Implementing Concurrent Programming Facilities on a Uniprocessor Machine Using Concurrent Pascal-S

Michael Camillone
Master of Science Thesis
CS 694
Pace University
Professor C.T. Zahn
March, 1992

INTRODUCTION AND HISTORY

The use of parallel computers has not yet reached the level of the smaller university; therefore, the only way to gain hands-on experience with parallel programming is through simulation. This involves using a PC and a software package designed to emulate multiprocessor activity on a uniprocessor machine. Such a software package is Concurrent Pascal-S, whose description and history follow. (Note: Although similar in name, this package is not related to Brinch Hansen's Concurrent Pascal implementation [4].)

The original Pascal-S compiler/interpreter was written by N. Wirth in 1976. It was able to compile a subset of Pascal into P-code (pseudo-code), each instruction consisting of an instruction field and zero, one, or two operands. Once the program was entirely compiled, the instruction set was then interpreted to simulate execution on a hypothetical stack machine. The stack was used for all runtime computation; therefore, there was no need for registers or other temporary storage places. There were no facilities for concurrent programming in the original Pascal-S.

M. Ben-Ari modified Wirth's original compiler/interpreter in 1980 to include some basic features that were able to simulate concurrent programming. First, a **cobegin** s_1; ...; s_n **coend** block structure was added, allowing concurrent execution of the statements s_1 ... s_n, which were required to be global procedure calls. These **cobegin** ... **coend** blocks could not be nested within one another. Second, the use of semaphores was introduced, with a semaphore data type (really synonymous with the integer data type) and the semaphore operations **wait(**s**)** and **signal(**s**)**, corresponding to Dijkstra's **P(**s**)** and **V(**s**)**, respectively. The remaining features of the Pascal subset compiled by Ben-Ari's kit can be found in the Appendix of [3]. Concurrent process execution was simulated by letting each process use the PC's processor for a random amount of time, and then giving all other processes a turn for another random amount of time. In this context, "time" refers to a count of instruction

executions, making the simplifying assumption that each instruction takes the same amount of time to execute. This procedure is called process switching, and it tries to best emulate the results that would be produced on a true multiprocessor machine.

The next modification of the Concurrent Pascal-S compiler was done by C.T. Zahn in 1988, implementing the package using Borland's Turbo Pascal. Zahn modified the **cobegin** ... **coend** syntax, replacing the semicolon with a vertical bar as the separator between concurrent execution calls. Also, regular program statements, as well as global procedure calls, were now allowed to be placed within the **cobegin** ... **coend** block, and these blocks could now be nested within one another. The code for **cobegin** and **coend** was modified as well, so that the concurrent processes signified within the block were treated as child processes forked by a parent process. After all the child processes have been forked, i.e. the **coend** keyword has been encountered, the parent process is put to sleep and does not execute any instructions until all its child processes are killed (finished executing). There is a limited number of "processors" available in the implementation, controlled by the constant **pmax**. Therefore, by simply changing the value of this constant, the user can simulate execution of the same program on a "machine" with different numbers of processors. When a parent attempts to fork a child, a wait is performed on the internal semaphore **FBSema**, the Free Block Semaphore. Free blocks correspond to processors currently not in use. **FBSema** checks to see if there is a free block eligible for use by the child process; if there is not, the parent process is temporarily suspended until a block becomes available.

Another key revision made by Zahn was the implementation of a circular Ready Queue containing all processes eligible for execution at any given moment. The scheduling was done via a round-robin approach using a time-slice quantum (a random number chosen from within a given

interval) to determine the amount of time each process had to use the processor for a given turn of execution. The Ready Queue made use of the already existing **ptab** processor table to simulate a linked list of processes by having each entry in the table have a **nextp** field that contained the index of another process. The queue was circular, so the last process in the list points to the first process.

At this time the reader should have a good grasp of the Wirth/Ben-Ari/Zahn Concurrent Pascal-S kit in order to fully understand the newest set of changes that have been made. The first modification involves keeping accurate track of the time of execution of each process in a given program in order to give the user access to various statistics about the program's performance. The key assumption made here is that one machine instruction equals one time unit; this assumption was made in order to simplify the duties of time-keeping. The second modification is the addition of a CSP/Occam-type synchronous message-passing scheme via the declaration of channels, the functions **send** and **receive**, and a guarded command structure using the keyword **select**. A description of these modifications will comprise the remainder of this paper, and they involve a fairly detailed description of the Pascal code in which the kit has been written.

TIMING AND STATISTICS

Process switching

In the existing implementation, process switching was performed by way of the variable **stepcount** and the procedure **setquantum**. Upon a call to **setquantum**, **stepcount** would be assigned a random value representing the number of machine instructions to be executed for the given turn. After that number of instructions has been performed and **stepcount** is reduced to zero, we say that the process will be "timed out," meaning that another process (if any) will now have the

opportunity to use the PC's processor. **Setquantum** is again called to determine how many instructions this second process is allowed to execute. This nondeterministic method of choosing random numbers is intended to most accurately simulate the behavior of a true concurrent computer, since multiple runs of the same program would yield different values for **stepcount**, yet the results of the program should be identical. This allows the user to detect program bugs that might have remained hidden otherwise. The randomness also simulates the unpredictability of the relative speed of various concurrent processes running on different processors; in other words, a multiprocessor machine may contain different kinds of processors, each with its own rate of instruction execution.

In order to accurately keep track of each process' running time and to correctly interpret the activity of **wait** and **signal** commands, it was necessary to perform process switches after each instruction. This was achieved by introducing the constant **USEQUANTUM**; when set to false, the **setquantum** procedure is never called and **stepcount** is always equal to zero, effectively forcing a process switch after every instruction. When set to true, the interpreter proceeds as initially described. To more fully understand why this change was necessary, consider the following situation: the main program forks five child processes. The first three processes issue **signal**s at times 10, 20, and 30, respectively (these times, as well as all times in this example, reflect the real time of execution over the entire program, not the time particular to any process). The fourth process then issues a **wait** at time 15; naturally, it is matched with the **signal** at time 10 from process one, since the other **signal**s could not have occurred yet. Then process five issues a **wait** at time 5, which will be matched with the **signal** at time 20; but was it really intended to be matched with the **signal** at time 10? There is no way to know for sure. So depending on the values given to **stepcount** throughout the program, different results may occur, thus defeating the nondeterministic

approach that needs to be achieved. Process switching after each instruction solves this problem, since it keeps the real time of all processes synchronized.

The processor table

The **ptab** processor table is the main data structure of the Concurrent Pascal-S kit. It is an array of **pmax**+1 records that contain key descriptive information about each process that is currently running. Several fields were added to the record data structure in order to keep track of each process' running and waiting times. The most important of these fields are **tstart**, which records the real time that a process began; **timer**, which is the running time of a process, or number of instructions executed; and **totalwait**, which shows the total time that a given process has spent waiting. An invariant of this method of timekeeping is that the current real time of any process can be obtained at any time by adding the values of **tstart**, **timer**, and **totalwait**.

Tstart of the main process is given the value of zero, since it is the first process to run. When subsequent processes are forked, their respective **tstart**s are obtained by summing the values of their parent process' **tstart**, **timer**, and **totalwait**, as stated in the invariant above.

The **timer** field of each **ptab** entry is incremented by one upon execution of every instruction. It starts out at zero, and is reset to zero when the **ptab** entry is returned to the free list.

The **totalwait** field, as previously stated and as its name indicates, records the total time a process has spent waiting as a result of **wait**s, waiting for a rendezvous, or going to sleep, as a parent process does after forking its children. ("Rendezvous" is a term that will be dealt with in greater detail later, but suffice it to say at this point that a rendezvous occurs when two processes come to a point in their execution where a "sending" process sends information to a "receiving" process, and

then each process goes about its own duties.) The calculation of waiting time involves the use of another new **ptab** field called **tblocked**. When a process encounters a semaphore **wait**, waits for a rendezvous, or goes to sleep, **tblocked** records the current time by summing **tstart**, **timer**, and **totalwait**. When the corresponding **signal** is issued, the rendezvous is accomplished, or the process wakes up from sleeping, the value of **tblocked** is subtracted from the current time of the signalling process, and this new value is added to **totalwait**.

A fifth field that was added to **ptab** is **procno**, which is used to uniquely identify a process. Its use is described in greater detail in the discussion of **sumtab** below.

Wait and Signal times

Upon conclusion of the user program's execution, a runtime graph is displayed tracing the activity of each processor throughout the program. In order to show when a process was in a waiting or sleeping state, it is necessary to keep track of the times that each process was blocked and when each process resumed execution. These times are stored in a new data structure, which is a linked list of **waitnode**s. Each **waitnode** contains **wait** and **signal** fields, as well as a **next** field which is a pointer to the next node. This list is part of the **sumtab** structure described below.

Two new procedures were written to populate the fields of this linked list. **AddWait** searches to the end of the list and then creates a new node, assigning to its **wait** field the value of the time the **wait** command is being issued, the rendezvous is sought, or the process goes to sleep. Similarly, **AddSignal** searches the list until it finds the first node with a wait time but no signal time, and then fills in the **signal** field.

Granularity

As pointed out in [15], in every concurrent program there is more absolute work done than in an equivalent serial program on account of the overhead involved in administering the concurrency. For this reason, a statistic that concurrent programmers are frequently interested in is the granularity of an application. Granularity is a ratio of this overhead to the actual work being performed. "Fine-grained" applications are those in which the amount of work approaches the amount of overhead (i.e. the ratio approaches 1); "coarse-grained" applications perform large amounts of work compared to relatively little overhead.

In the Concurrent Pascal-S kit, the concurrency overhead amounts to five instructions per process: load **FBSema**; **wait**; **fork**; **jump**; **kill**. Finding the total overhead, therefore, only requires keeping track of the number of forks performed throughout the program and multiplying this number by five. For this reason, the variable **numforks** has been introduced into the program. Its use is simple as it only needs to be incremented when a **fork** statement is encountered. At the conclusion of the program, this overhead figure is taken and divided into the total number of instructions executed throughout the entire program; this percentage is the granularity of the application.

The summary table

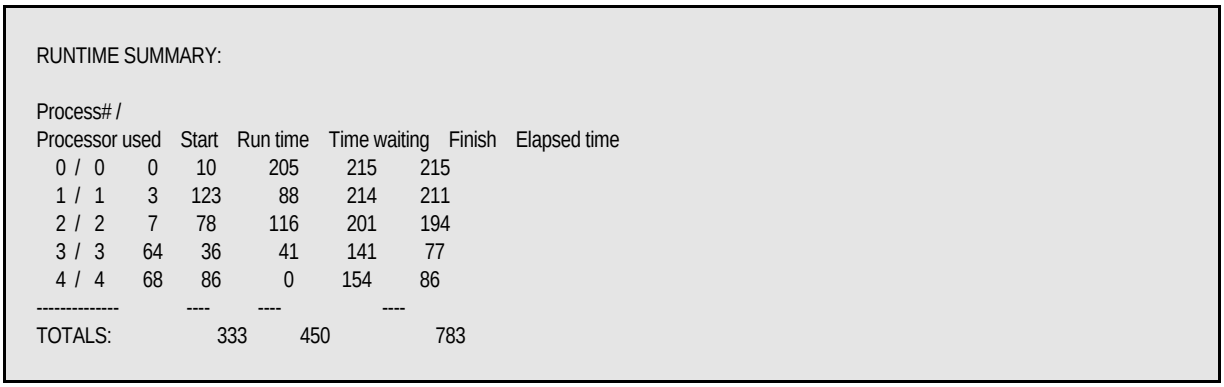
If the reader has not already noticed, there is a problem with the method of using the **ptab** entries to store all information about processes if we want to recall all this information at the program's termination. Suppose, for example, that we allow our implementation to have ten processors (ten **ptab** entries). During program execution, ten processes are forked, each occupying one of the ten processors. Subsequently, an eleventh process is forked; it waits until a processor is

free. When it is assigned a processor, that processor's **ptab** entry is erased to make way for the eleventh process. There is no way to recover this information at the program's end to perform any kind of statistics.

To remedy this problem, a new data structure called **sumtab** was introduced. Intended for use as a summary table of processes, its structure is similar to that of **ptab**. It is an array of records indexed by a unique process identifier; this is the same identifier assigned to the **procno** field of **ptab**. It has a field called **processor_used**, which is the index of **ptab** that was used by this process. Its next three fields are **start**, **running**, and **waiting**, which reflect the **ptab** values of **tstart**, **timer**, and **totalwait**, respectively. Its final field is **waitlist**, which is a pointer to a set of **waitnode**s giving the corresponding wait and signal times of that process. The variable **proccount** is used to keep track of how many processes have lived during the program's execution, and it serves as an index into **sumtab**.

In order to transfer the **ptab** information to **sumtab**, the procedure **ReportProcess** was introduced. As its name suggests, it simply reports vital process information from one table to the other upon the process' termination.

Runtime summary



RUNTIME SUMMARY:

Process# / Processor used		Start	Run time	Time waiting	Finish	Elapsed time
0 / 0	0	10	205	215	215	
1 / 1	3	123	88	214	211	
2 / 2	7	78	116	201	194	
3 / 3	64	36	41	141	77	
4 / 4	68	86	0	154	86	
TOTALS:			450		783	

Figure 1: Runtime Summary

As previously mentioned, when the program has finished its execution, various statistics are displayed to the user. By default these statistics are displayed on the screen, but the user also has the option of redirecting the output to a disk file or the printer; this redirection is accomplished at the command line in the following manner:

cps <myprog.in >myprog.out

where *cps* is the command to invoke the Concurrent Pascal-S compiler/interpreter, *myprog.in* is the file containing the user's program, and *myprog.out* is a file that will be created and will contain the output generated by the kit. Alternatively, the user can specify *prn* in place of *myprog.out* in the example above to send the output to the printer.

The first statistic is performed by the procedure **RuntimeSummary**, which shows the following information for each process: which processor it used, when it began executing, its run time, its time spent waiting, when it finished executing, and its elapsed time (running time + time waiting). These figures are calculated by simply accessing each record of **sumtab** and printing its information. A sample runtime summary from an actual printout is shown in Figure 1.

Relative utilization

Relative utilization is a measure of the run time of a process compared to its elapsed time. In other words, it is a percentage that expresses how much time of a process' life was spent actually executing instructions, as opposed to time spent waiting. The procedure **RelUtilization** accomplishes this task in much the same way as **RuntimeSummary** did its job: by displaying key information from each **sumtab** entry. For each process, **RelUtilization** shows the process number, the ratio of run time to run time plus wait time, and the percentage that this ratio represents. This percentage is called the utilization of the process. The user should note that the total run time plus the total wait time should be equal to the total elapsed time shown in the Runtime Summary.

```
RELATIVE UTILIZATION PERCENTAGES (across processes):

Process#    Run/ Run+Wait  =  Utilization
    0        10/  10+ 205  =      4.7%
    1       123/ 123+  88  =     58.3%
    2        78/  78+ 116  =     40.2%
    3        36/  36+  41  =     46.8%
    4        86/  86+   0  =    100.0%
--------   --------------       ------
TOTALS:     333/ 333+ 450  =     42.5%

GRANULARITY of this application =
ratio of concurrency overhead to actual work being performed =
20 / 333 =   6.0%
(High percentage = Fine-grained; Low percentage = Coarse-grained)
```

Figure 2: Relative Utilization Percentages and Granularity

RelUtilization is also responsible for displaying the granularity of the application, as discussed above in the **Granularity** section. Figure 2 shows a sample of both the Relative Utilization and Granularity displays.

Absolute utilization

Whereas relative utilization is a measure of processes, absolute utilization is a measure of processors. For each processor, the latter shows a ratio of the sum of run time and wait time as compared to the total time the processor spent in the following three states: running, waiting, or idle. The terms "running" and "waiting" have been referred to before, so for the sake of clarity, it is pointed out that "idle" refers to that time when a processor is not attached to any process; it is not running, it is not waiting, it is simply available for use to any process that needs it.

```
ABSOLUTE UTILIZATION PERCENTAGES & GRAPHS (across processors):

Processor    Run+Wait/ Run+Wait+Idle  =  Utilization
    0         10+ 205/  10+ 205+   0  =    100.0%
    1        123+  88/ 123+  88+   4  =     98.1%
    2         78+ 116/  78+ 116+  21  =     90.2%
    3         36+  41/  36+  41+ 138  =     35.8%
    4         86+   0/  86+   0+ 129  =     40.0%
---------   ------------------------      ------
TOTALS:      333+ 450/ 333+ 450+ 292  =     72.8%
```

Figure 3: Absolute Utilization Percentages

Calculating these numbers is not as easy as finding the relative utilizations because it is neccessary to access all of the **sumtab** entries and pick out those that made use of the same processor and sum their running and waiting times. In addition, idle times must be determined as the sum of the times when a processor was between attachments to processes as well as the time before its initial attachment. These figures and their corresponding percentages are then displayed. There is also an internal check that is performed to ensure that the sum of the run and wait and idle times of each processor are all equal. Figure 3 shows a sample of the Absolute Utilization statistics.

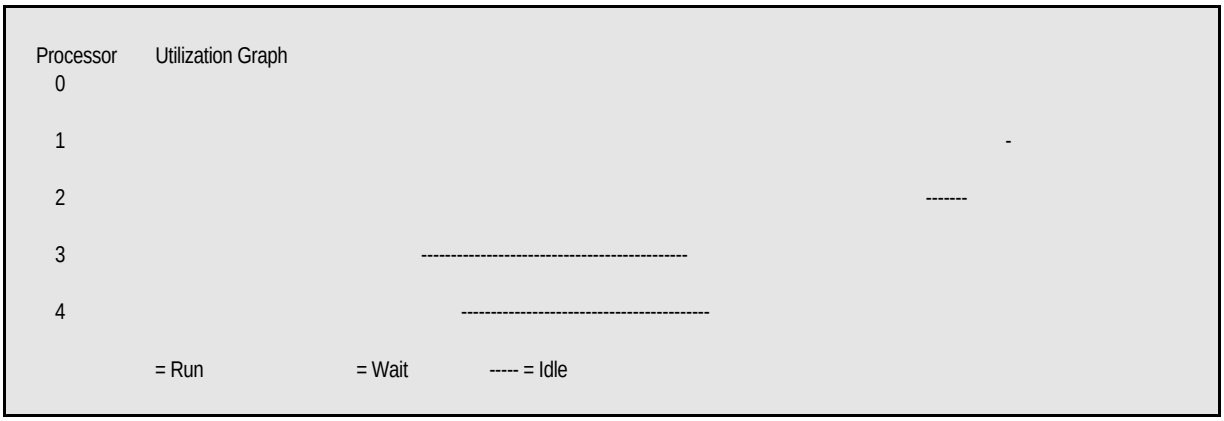


Figure 4: Absolute Utilization Graph

The final action performed by **AbsUtilization** is a call to its internal procedure **BarChart**. This procedure takes the run, wait, and idle times obtained previously and displays them in bar chart form, so that the user can get a visual sense of the activity of the program. A sample bar chart is shown in Figure 4.

A useful bit of information that the user can glean from both the relative and absolute utilization percentages and graphs is the possible presence of bottlenecks in the program. If a certain processor spent most of its time waiting, the user can see this and take appropriate actions. Also, since the total number of processors used to execute the program is displayed, the user should note that more processors could not have been used with the algorithm driving the program. In other words, if the user wants to make use of more processors, the algorithm must be changed.

Runtime graph

The final output provided to the user about his/her submitted Concurrent Pascal-S program is a runtime graph that shows the activity of each processor throughout execution of the entire program. The activity at each time interval is shown in the following manner:

- if the processor was busy executing instructions of a particular process, that process number is displayed
- if the processor was delayed on account of a semaphore, a rendezvous request, or sleep, a dot is displayed
- if the processor was idle, that space is left blank

The time interval is controlled by the variable **displayincr**. At the start of a run, the user can choose between regular output mode and an "adaptive" output mode. In the case of the regular mode, **displayincr** is set to 1 so the user can see every action at every time instant; in the adaptive mode, the output is restricted so that it does not exceed a length of two pages. This latter mode is accomplished by dividing (via **div**) the total execution time of the program by 100 and giving that value to **displayincr**; thus, a one-unit "snapshot" is taken every **displayincr** units. This ensures that the printout will not go beyond the 132-line limit of two full pages of text. Figure 5 shows a portion of the output obtained from a particular example using the adaptive mode.

```
RUNTIME GRAPH:
----  ---------- Processors ----------
Time   0  1  2  3  4  5  6  7  8  9  10
----  ------------------------------------------------------
  0   0
  4   0
  8   0
 12   0
 16   0
 20   0
 24   0
 28   0  1
 32   0  1  2
 36   0  1  2  3
 40   0  .  2  3  4
 44   0  .  2  3  4  5
 48   0  1  2  3  4  5  6
 52   0  1  2  .  4  5  6  7
 56   .  .  2  .  .  5  6  7  8
 60   .  .  2  .  .  .  6  7  8
 64   .  .  2  3  .  .  .  7  8
 68   .  .  2  3  .  .  .  .  8
 72   .  1  2  3  .  .  .  .  .
 76   .  1  2  3  4  .  .  .  .
 80   .  .  2  3  4  .  .  .  .
 84   .  .  2  3  4  .  .  .  .
 88   .  .  2  3  4  .  .  .  .
 92   .  .  2  3  4  5  .  .  .
 96   .  1  2  3  4  5  .  .  .
100   .  1  2  3  4  5  .  .  .
104   .  .  2  3  4  5  .  .  .
108   .  .  2  3  4  5  6  .  .
112   .  .  2  3  4  5  6  .  .
116   .  .  2  3  4  5  6  .  .
120   .  .  2  3  4  5  6  .  .
124   .  1  2  3  4  5  6  7  .
128   .  .  2  3  4  5  6  7  .
132   .  .  2  3  4  5  6  7  .
136   .  .  2  3  4  5  6  7  8
140   .  .  2  3  4  5  6  7  8
144   .  .  2  3  4  5  6  7  8
148   .  1  2  3  4  5  6  7  .
152   .  1  2  3  4  5  6  7  .
156   .  .  2  3  4  5  6  7  .
160   .  .  2  3  4  5  6  7  .
164   .  .  2  3  4  5  6  7  8
168   .  .  2  3  4  5  6  7  8
172   .  1  2  3  4  5  6  7  .
176   .  1  2  3  4  5  6  7  .
180   .  .  2  3  4  5  6  7  .
184   .  .  2  3  4  5  6  7  .
188   .  .  2  3  4  5  6  7  8
192   .  .  2  3  4  5  6  7  8
196   .  1  2  3  4  5  6  7  .
200   .  1  2  3  4  5  6  7  .
204   .  .  2  3  4  5  6  7  .
208   .  .  2  3  4  5  6  7  .
212   .  .  2  3  4  5  6  7  8
216   .  .  2  3  4  5  6  7  8
220   .  .  2  3  4  5  6  7  8
224   .  1  2  3  4  5  6  7  .
```

Figure 5: Runtime Graph (Adaptive Mode)

In terms of the code used to write this segment, for each time interval, the procedure **DisplayGraph** simply scans the **sumtab** entries looking for the processes that used processor 0, then processor 1, etc., up to processor **pmax**. When it has found a desired **sumtab** entry, it checks its beginning and end times via the **start** field and the sum of the **start**, **running**, and **waiting** fields, respectively. If this time interval is the one sought, another check is made to see if the process was in a delayed state, via the **waitlist**. If it is found to be delayed, a dot is printed; otherwise, the process number is printed. Processors are determined to be idle for a time interval if no **sumtab** entry is found such that that time interval falls between its **start** field and the sum of its **start**, **running**, and **waiting** fields. In this case, the space under that processor for the time interval is left blank.

View process summary table

The procedure **DumpSumtab** is provided as a debugging aid to the programmer. The contents of **sumtab** can be printed on the screen as part of the analysis printout if the user so chooses at the start of the session. Internally, the state of the Boolean variable **ShowInternal** determines whether or not this additional information will be displayed. These extra details consist of the index of each **sumtab** entry as well as the **processor_used**, **start**, **running**, and **waiting** fields. Figure 6 shows an example of a **sumtab** dump.

```
Dump Sumtab:
sumtab[ 0]  processor_used= 0  start=  0  running= 10  waiting= 205
sumtab[ 1]  processor_used= 1  start=  3  running= 123  waiting=  88
sumtab[ 2]  processor_used= 2  start=  7  running=  78  waiting= 116
sumtab[ 3]  processor_used= 3  start= 64  running=  36  waiting=  41
sumtab[ 4]  processor_used= 4  start= 68  running=  86  waiting=   0
```

Figure 6: **Sumtab** dump

View list of wait and signal times

Along the lines of **DumpSumtab**, the procedure **DumpWaitlist** is another debugging aid. Also making use of the variable **ShowInternal**, this procedure gives the user the option of seeing the times at which each process executed a **wait** (or waited for a rendezvous or was put to sleep) and the times when they were awakened. A sample **wait** and **signal** list dump is shown in Figure 7.

```
Dump Waitlist:
 0)  9 & 214
 1)  16 & 19 / 76 & 161
 2)  30 & 38 / 49 & 57 / 70 & 154 / 172 & 180 / 191 & 199
 3)  76 & 81 / 88 & 100 / 107 & 119 / 126 & 138
 4)  No waits.
```

Figure 7: Dump of **Wait** and **Signal** Times

Sleeping processes

When the main process has completed forking all of its child processes, it goes to sleep and is removed from the Ready Queue. However, in the existing implementation, the **status** field of the process' **ptab** entry still reported that the process was in the **ready** state (the constant **ready** is really synonymous with the value 2). This fact can cause problems with time measurements. For

example, it is possible for the first child process forked by its parent to be so short that it completes execution before the parent has forked its second child. When the first child executes its **kill** command, it checks the parent's **childcount** field. If this field is equal to zero, vital time measurements are taken and the parent is awakened and re-inserted back into the Ready Queue. In the example at hand, though, the **childcount** field will be equal to zero, and even though the parent has not yet gone to sleep because it has more children to fork, it will be "awakened" and re-inserted into the Ready Queue prematurely. This is obviously a problem.

The solution is simple, as it follows a simple rule: the parent cannot be awakened before it goes to sleep! In order to implement this rule, a fifth possible processor status was added: that of **sleeping**, given the value 4. This also required modifying the range of the **status** field of **ptab** to read **0 .. 4**. Now, when a parent process goes to sleep, its **status** is given the value **sleeping**. And when a child executes a **kill** and checks to see if its parent should be awakened, not only does it check the value of **childcount** but it also checks to see if the parent's **status** is **sleeping**. If so, the parent is awakened and its **status** is returned to **ready**; if not, the parent is not awakened.

Access to the list of free processors

In a true concurrent environment, we would like to disallow simultaneous access to the list of free processors by more than one process for a given time interval. One reason for this is that **kill**s and **fork**s issued at the same time are dangerous. For example, if there are no processors available and a **fork** is issued, the response from the system will be to make the inquiring process wait. If a **kill** is issued at the same time but dealt with by the PC's processor after it deals with the **fork**, a process block will be returned to the free list. So why did the process desiring to be forked

have to wait if a process block was indeed available at that time interval? The problem is now obvious.

There are various ways to solve this problem in the Concurrent Pascal-S implementation. A mutex semaphore could be used to ensure mutual exclusion with regard to access of the free list. Alternatively, an internal check could be made inside the program to disallow **fork**s and **kill**s at the same time. A third method, however, was chosen that has the same desired effect. After each process completes an instruction, the interpreter checks to see if its next instruction will be a **kill** instruction. If it is not, this process will be put at the end of the Ready Queue as usual. If it is, this process will be inserted into the Ready Queue so that it is at the beginning of the next executing time interval. In other words, for each time interval, the interpreter will always execute all **kill** instructions first (if any). That way, when subsequent **fork** instructions are encountered, all possible process blocks will have been returned to the free list.

SYNCHRONOUS MESSAGE PASSING

Reasons for implementation

The implementation of concurrent processes is a significant achievement; however, this achievement needs to be enhanced by allowing the processes to communicate with each other. Various methods have been proposed (and implemented) by others to facilitate process communication, and several of them are described in [6]. Of these assorted methods, only one was available in the existing Concurrent Pascal-S implementation, namely, communication via shared variables. Since several processes can access the same variables, values can be deposited into the variables by one process and picked up by another, thus achieving communication. However, this

method immediately brings up the problem of mutual exclusion; that is, a situation where two or more processes unknowingly access the same variable in such a way that the actions of one process are subsequently overwritten by another. This problem can be solved by the use of semaphores to protect access to shared variables. Semaphores, along with their corresponding functions **wait** and **signal**, were already a part of the existing Ben-Ari implementation. Monitors (also summarized in [6]) can also be used to solve the mutual exclusion problem. They are not explicitly defined in the Concurrent Pascal-S kit, but they can be simulated by the user by the correct use of semaphores. Refer to Appendix A for such a simulation, which was taken from Hoare's 1974 paper in *Communications of the ACM* [9] and was translated into a Concurrent Pascal-S program by C.T. Zahn.

Neither of these methods allows direct process communication. Concurrent programmers would surely benefit from some sort of "message passing" scheme, which is indeed the central idea of Hoare's Communicating Sequential Processes (CSP) notation, as presented in [10] and [11]. This programming model allows two processes to directly communicate with each other by issuing input and output commands in the following fashion:

PROCESS_NAME **?** VAR

allows the issuing process to input (signified by the **?** operator) a value from process PROCESS_NAME and store it in the location VAR. Similarly,

PROCESS_NAME **!** EXPR

is a command whereby the value of EXPR is output (the **!** operator) to process PROCESS_NAME by the issuing process.

Message passing can be of either the synchronous or the asynchronous variety. In the

synchronous scheme, a process wishing to output (or input) a value must wait until another process issues a corresponding input (or output) command. The point at which the two processes are actually performing the exchange of information is called the rendezvous. After the rendezvous, the two processes each go on their separate ways.

In contrast, the asynchronous message passing scheme permits the process issuing the output command to continue on its way immediately after the command is issued, whether or not a corresponding input command has been issued; it does not have to wait for a rendezvous. The process performing the input is still required to wait for a rendezvous, since it does not make much sense for a process to issue and input command and then have nothing to show for it. If several processes have output messages to a particular process that has not yet performed an input, those messages are simply put on a queue for that input process and must be serviced one at a time.

The synchronous paradigm also allows for process synchronization. This is due not only to the aforementioned rendezvous scenario, but it is also possible for processes to exchange "empty" or essentially meaningless messages whose sole intent is to synchronize the pair of processes. In other words, the message allows the processes to reach a common point in time and then proceed anew.

CSP also incorporates the concept of Dijkstra's guarded command [8]. A typical guarded command is constructed in the following manner:

if $\text{Guard}_1 \rightarrow \text{Command}_1$
$\text{Guard}_2 \rightarrow \text{Command}_2$
.
.
.
$\text{Guard}_n \rightarrow \text{Command}_n$
fi

The darkened bars separate the several guarded commands present in the construct. Each of the *n*

guards is a Boolean expression, usually incorporating one of the communication primitives discussed above. For some guard that evaluates to true, its corresponding command set is executed. For example, in the above set of guarded commands, if $Guard_2$ evaluated to true, $Command_2$ would be performed. If more than one guard is true, then one of the corresponding command sets will be chosen at random. In the case of all guards evaluating to false, the program will abort.

Occam is a concurrent programming language that embodies the basic concepts of CSP (see [12], [13], and [17], as well as [16] and [6] for discussions of Occam). A notable difference is that the process communication is not signified by naming the process with which a rendezvous is desired; rather, channels are established between processes as a means of conveying values, and they are explicitly named as the vehicles through which communication is achieved. In the Concurrent Pascal-S implementation, we have incorporated the ideas of CSP and Occam in order to provide a synchronous message passing mechanism to allow the user to experience this concurrent programming paradigm. A description of its syntax, semantics, and implementation follows.

Syntax and semantics

Four new commands have been added to the Concurrent Pascal-S language to facilitate synchronous message passing. The first of these is the **getchannel** command, which has the following syntax:

getchannel(chan **)**

A **getchannel** command is issued by a process with the effect of attaching a certain channel exclusively to itself such that it can now receive messages from that channel. Processes must issue a **getchannel** before they can attempt to receive from that channel, thereby establishing a many-to-one

send-receive environment, as presented in [7] and [14].

The input (**?**) and output (**!**) commands from the CSP discussion above have been implemented as the second and third new Concurrent Pascal-S commands:

send(expr, chan **)** ≈ chan **!** expr
receive(chan, var **)** ≈ chan **?** var

A process may **send** an expression to any channel, even if that channel has not yet been attached to a process. Multiple **send**s are put into a queue associated with their channel. Processes that **send** to a channel must wait until a corresponding **receive** is issued on that channel; after the rendezvous, both the sending and receiving processes proceed on their own.

As stated above, a **receive** cannot be issued unless the issuing process already owns that channel (has performed a **getchannel**). If it does own the channel, though, the first item on the send queue is transferred to the location specified by *var*. As is the case with the **send** command, a process issuing a **receive** must wait until the corresponding **send** command is issued.

The fourth and final new command added to the Concurrent Pascal-S implementation is the guarded command structure. The syntax is as follows:

select
    B_1 & C_1 → commands
#
    B_2 & C_2 → commands
#
    .
    .
    .
#
    B_n & C_n → commands
end

The semantics of the Concurrent Pascal-S guarded command are the same as described above. Here,

each *B* and *C* combination comprises a guard, where *B* is a Boolean expression and *C* is a communication command (either **send** or **receive**). A guard need not contain both the Boolean and communication components; only one of the two need be present. (The reader will note that a guarded command structure with guards containing only Boolean expressions is essentially a non-deterministic selection statement.)

Implementation

The channel table

The scheme for implementing synchronous message passing is now presented. Since the message passing is accomplished via channels, we need a data structure to store the key information about the channels. An array of records called **chantab** was introduced for this purpose, to serve as the channel table. The table is indexed by the same number that indexes its location in the identifier table **tab**. Each channel entry contains the process ID number of its owner (i.e. the process that performed a **getchannel** on that channel) in a field called **owner**, as well as a field called **uniqueowner** that stores the unique process ID number of the owner process in case of a post-mortem dump.

The remaining fields of **chantab** are used to maintain the list of **send** requests on that channel and the list of **receive** requests on that channel; the former list is called **sendq**, while the latter list is called **receiveq**. Both of these lists are stored as FIFO queues, since the first communication requests posted are the first to be serviced. They are composed of nodes of type **qptr**, which will be discussed more thoroughly in a following section.

The final field of **chantab** is **sendcount**, an integer variable used to keep track of the number

of **send** requests that still require service. As with the type **qptr** mentioned above, the use of **sendcount** will be explored later.

At this point it would be appropriate to note that when the interpreter is first invoked, each entry of the channel table has its **owner** and **uniqueowner** fields initialized to **nilproc** (a constant equal to -1, showing that no process owns the channel), **sendcount** initialized to zero, and **sendq** and **receiveq** initialized to nil.

The send and receive queues

At first glance, it may seem fairly obvious how to go about storing **send** and **receive** requests in their respective queues: every time a request is issued, simply create a new node containing the proper information and append it to the end of the appropriate queue. But an additional challenge presented itself in the inclusion of the guarded command into the language. The above method will no longer suffice because of the following reason: **select** statement blocks may contain several guards, which in turn may contain several communication requests. To aid in ease of compilation, these requests all need to be posted as they are encountered by the interpreter (assuming that the Boolean portion of the guard, if present, is true). But as program execution proceeds, only one of those communication requests will be processed, and the others will be cancelled. So a way is needed to post all the requests found within a guarded command, and yet be able to go back and cancel those that will never be serviced.

The solution to this problem involves the inclusion of the set of all **send** and **receive** requests of a particular guarded command block in a circular linked list. This is not a separate list, but rather it is incorporated into the **sendq**'s and **receiveq**'s of the channels involved. An illustration is shown in Figure 8. In this example, there are three channels, each represented by a darkly-outlined box.

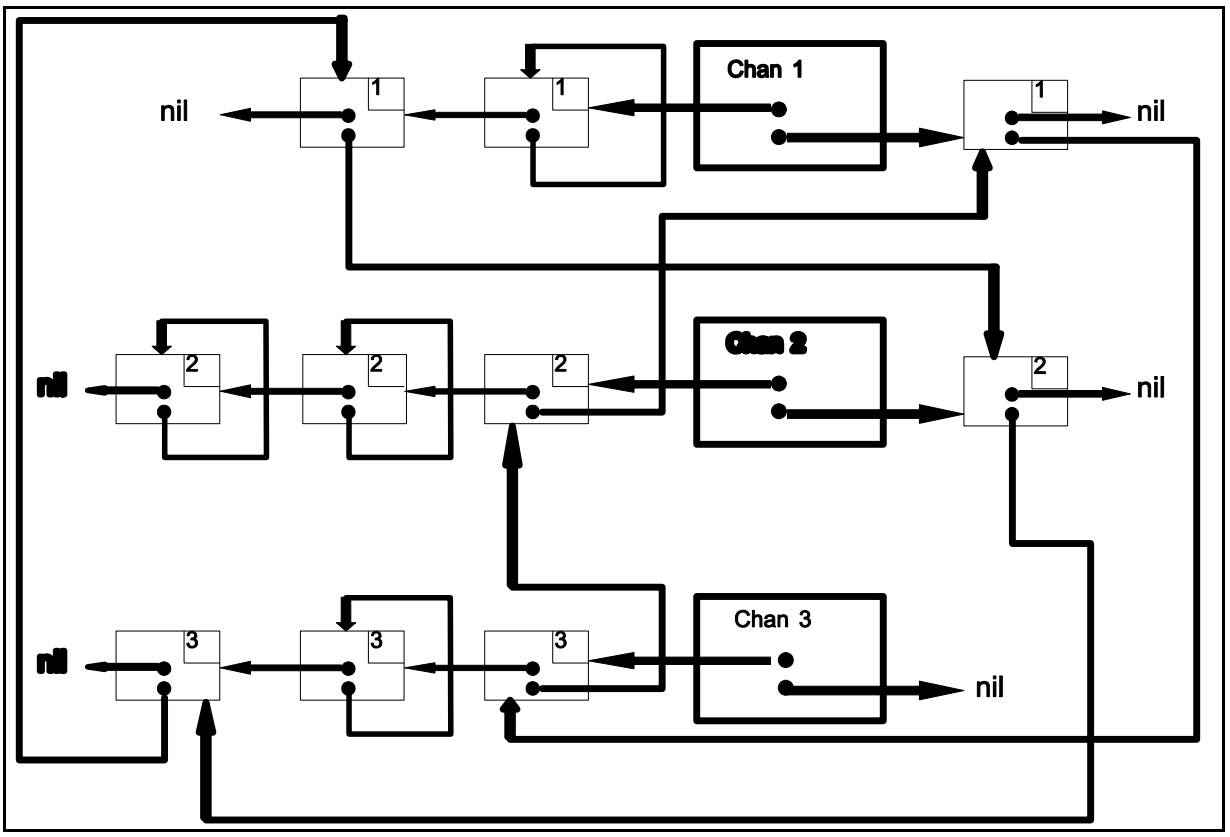


Figure 8: Synchronous message passing scheme

Each channel is named and has a **sendq** pointer, represented by the higher of the two circles in the box, and a **receiveq** pointer, represented by the lower circle. Messages that have been posted are portrayed as smaller, lightly-outlined boxes, reflecting the **qentry** data structure in the kit (**qptr** has been established as a pointer to a node of type **qentry**; therefore **sendq** and **receiveq** are of type

qptr). Each **qentry** node has seven important fields; the first is the **process** field, reflecting the process ID number of the process that issued the communication request. Second is the **channel** field, which tells to which channel that particular request was issued. In Figure 8, this number is shown in the upper right-hand corner of each message node. It is needed because when we are traveling around the circular linked list, we need to be able to refer back to the proper channel, since the list does not have backward pointers.

The third field in the **qentry** node is called **direction**; it tells whether a node is a **send** request or a **receive** request. Next comes the **comm** field, which in the case of a **send** request contains the expression that was sent, and in the case of a **receive** request contains the address of the variable where the message is to be stored. The **resume** field tells where program execution is to resume should this node be serviced.

The final two fields of the **qentry** data structure are both pointers of type **qptr**. The first is called **next** and simply points to the next entry in either the **sendq** or the **receiveq** for that channel, and pointing to **nil** at the end of the list. In the figure, this pointer is represented by the upper circle in each node. The other pointer, illustrated by the lower circle in each node, is named **setlink**, and it is used to link together, in a circular linked list, the set of all nodes that were posted as the result of a guarded command block. In the case of individual **send** and **receive** requests that were issued independent of a guarded command, this pointer simply points to its own node, reflecting a singleton set. So in Figure 8 we see an example where each channel has been issued a combination of singleton requests as well as requests resulting from a guarded command block. In this particular example, it is evident that two guarded command blocks have been encountered since there are two separate circular linked lists. The exact process that is used to populate and maintain these lists will

be explored later in the detailed explanation of the code.

New variables

To accommodate the inclusion of the guarded command structure, it was necessary to introduce two new variables to the **interpret** procedure. The first of these is **selcount**, an integer that reflects the number of communication requests that were posted as the result of a guarded command block. This number is not simply equal to the number of requests present in the actual text of the block, but shows how many Boolean portions of the guards evaluated to true, thereby resulting in the posting of their corresponding communication requests. In the case of normal **send** and **receive** statements, **selcount** has the value of 1.

The second variable that was added is **setptr**, a pointer of type **qptr** that points to that process' circular set of communication requests. After the circular linked list has been constructed, **setptr** is pointed at one of the nodes to provide immediate access to the list.

The Getchannel instruction

Three new P-code instructions were added to the Concurrent Pascal-S kit to accommodate the synchronous message passing scheme. These are the instructions that are emitted by the parser and then passed on to the interpreter. The first of them is the Getchannel instruction, which has been given instruction number 42. This instruction simply accesses the **owner** and **uniqueowner** fields of the proper **chantab** entry and assigns to them the process ID number of the running process (**runpr**) and that process' unique ID number (**procno** field of **ptab**), respectively.

The Post instruction

Instruction number 41 is the Post instruction, and as evidenced by its name, it is used to post communication requests, both inside and outside of guarded command blocks. The Post instruction is the third instruction of the three-instruction set that comprises both the **send** and **receive** commands. The first two instructions of this set cause the channel identifier and expression/variable address to be loaded onto the stack. Post then pushes three more values on top of these two: the process issuing the request, the resume address, and an integer identifying the request as either **send** or **receive**. These five values will be used by the Comm instruction (below).

The Comm instruction

The purpose of the Comm instruction (number 40) is to go through the set of communication requests that have been posted and select one for execution if a rendezvous is possible; otherwise, the process must cease execution temporarily and wait for a rendezvous. It works in the following manner:

First, Comm checks to see if a valid request has been made, and issues an error if this is not the case. Otherwise, it proceeds to enqueue each request in turn onto the appropriate queue of the appropriate channel, incrementing its local variable **selcount** for each node that is constructed. Since the channel identifier and the expression/variable address are pushed onto the stack in opposite order for **send** and **receive** requests, it is the responsibility of Comm to put them all in the same order before it enqueues the request. This ensures the uniformity of the code for the **EnQueue** procedure. It is this procedure, along with its local procedure **FillNode**, that actually searches along the **sendq** or **receiveq** of the proper channel, appends a new node, and fills that node with the

required information. In addition, **FillNode** sets the values of the **setlink** and **setptr** pointers to construct the circular linked list request set. **EnQueue** also does a bit of maintenance work, as it deletes those nodes on the **sendq** that have a channel value of -1; these nodes were posted as a result of a previous **select** block, and they were "cancelled" to show that they were no longer eligible for servicing.

After all requests have been enqueued, Comm picks a random number greater than or equal to zero and less than **selcount** and advances **setptr** that many times around the circular linked list. This is done in order to provide a way of nondeterministically selecting a communication request from the current guarded command set to be serviced. (Note: This procedure works for regular **send** and **receive** requests also, since in their cases **selcount** is equal to one; therefore, the random number chosen is always zero and **setptr** is not advanced.) Comm then starts at the node at which **setptr** is now pointing and goes around the entire set of requests one by one. At each stop, Comm examines the channel on which that node has been issued to see if it is ready for a rendezvous (if **sendcount** <> 0 and **receiveq** <> **nil**). If it is indeed ready for a rendezvous, then procedure **Rendezvous** is called. This procedure does the actual exchange of information through the channel between the two processes, wakes up the processes that had been put to sleep waiting for the rendezvous, and adjusts their **tblocked** and **totalwait** fields to reflect the time spent waiting. The final act of **Rendezvous** is to call the procedure **CancelNodes**, which goes through the request set associated with both the sending process and the receiving process. For each node encountered, if it is a **send** node, its **channel** field is set to -1 (the request is "cancelled") and the **sendcount** of that channel is decreased by one; if it is a **receive** node, that node is deleted and the **receiveq** pointer of its channel is set to **nil**. The reason for this difference in activity is that **receiveq**'s can only contain

one node in this implementation, so they can easily be deleted immediately. **Sendq**'s, though, can be many nodes long, so it will suffice to simply mark them as cancelled now and do the actual deletion later.

Returning to the Comm instruction itself, it will continue to go around the set of enqueued requests, searching for one that is ready for a rendezvous. If the search is exhausted and no rendezvous has been performed, then that process must wait for a corresponding communication request to be issued. At this point the procedure **WaitforRen** ("Wait for Rendezvous") is called. This procedure removes the current process from the ready queue and sets its **status** field to reflect the cause of the waiting (i.e. whether blocked on a **send**, **receive**, or **select** statement). It also fills in the **blocker** field with the identifier of the channel on which the request has been issued.

That is the end of the Comm instruction. It is called once for every regular **send** or **receive** request, and once for every set of requests encountered in a **select** block. The kit can immediately tell whether a Comm instruction is to be used for a singleton request or a **select** block by examining the *y* argument of the Comm instruction. This field has the value 1 (represented by the constant **single**) if it is a single communication request, or it has the value 0 (denoted by the constant **select**) if it is used for a **select** block.

Entering the select statement block

Before the various components of the **select** statement block are processed, the parser emits an instruction 24 (push literal) code with the argument **select** (0). This zero value is pushed onto the stack to act as a sentinel value to the Comm instruction; that is, Comm will continue to pop 5-tuples of communication request information until it reaches the sentinel value (0).

Parser modifications

Another facet of the modifications necessary to implement synchronous message passing is the changes made in the code for the Concurrent Pascal-S parser. The parsing of the **getchannel**, **send**, and **receive** commands is fairly straightforward as it basically follows the style already used to parse the **wait** and **signal** commands. An important thing to note, however, is the use of the variable **parseguard**, which is set to true when the parser is in the middle of going through a guarded command block. The code for **send** and **receive** will only emit a Comm instruction (40) if **parseguard** is false; otherwise, it is up to the parse code for the **select** statement to emit the Comm instruction. Also, it should be noted that the Post instruction (41) is emitted with one argument: either that of the constant **send** (1) or the constant **receive** (2).

The parsing of the **select** statement can be derived from the following EBNF grammar:

```
selcom          = 'select' guardcomm { '#' guardcomm } 'end' .

guardcomm       = guard '->' commands .

guard           = boolean [ '&' communication ] | communication .

communication   = 'send(' expr ',' channel ')'
                | 'receive(' channel ',' var addr ')' .
```

The **selcom** portion above was actually called **selectstatement** in the program to conform to the nomenclature of other procedures such as **ifstatement** and **whilestatement**. The first action performed by **selectstatement** is to emit instruction 65 (Enter Select), and its last action is to emit instruction 66 (Exit Select). In between it repeatedly calls procedure **guardcomm** to process all of the guards in the block; **guardcomm** in turn calls procedure **guard** to process the actual guard. A

key responsibility of all three of these procedures working together is to keep track of return addresses that must be inserted into previously-generated code to enable the interpreter to jump around sections of code when necessary (eg. when the Boolean portion of a guard is false, or after parsing the code for a communication request).

For the sake of completeness, we note that the words **channel**, **getchannel**, **send**, and **receive** were all entered into the identifier table **tab**. Also, the keyword **select** was added to **key**, the table of keywords, and its corresponding symbol **selectsy** was added to **ksy**, the table of keyword symbols. Two additions were made to the special symbols table **sps** as well: **sps['#']** was set to **pound**, and **sps['&']** was set to **ampersand**.

ADDITIONAL FEATURES

Post-mortem dump

Every programmer realizes the importance of a good post-mortem dump. When his or her program aborts because of some kind of error, it is always helpful to see information concerning the state of the system at the time of the abort. The existing Concurrent Pascal-S kit did perform a dump of this nature, but it has now been further enhanced as well as given its own procedure, **PostMortemDump**. Like before, it still issues the same initial message:

Halt at xx in process yy because of zz

where *xx* represents the value of **pc** at the time of the halt, *yy* is the process ID number that was currently executing, and *zz* is a specific message displaying the cause of the abort. Two such messages were added to the implementation to reflect errors associated with channels and guarded commands. The first message is *illegal channel access*, and its associated **ps** status is **chanchk**.

This message is shown when a process tries to send a message to itself or receive a message from a channel owned by another process. The second new message is *false guards in select statement* and its associated **ps** status **guardchk**. This warning is displayed if none of the Boolean portions of the guards of a guarded command block are true. The user must construct his or her program in such a way that at least one Boolean guard evaluates to true.

PostMortemDump then displays a table showing, for each active process, the status of that process, what semaphore or channel that process may be blocked on (or it may be asleep), the value of the Program Counter, how long that process has been running, and how long that process has been waiting as a result of **wait** commands, waiting for a rendezvous, or sleeping.

Finally, another table is shown, containing information about the user's program's global variables. Their name, type, offset, and current value are displayed, as well as a comment about their present state. For example, semaphores tell whether or not they are blocking any particular process, and channels tell what process they are owned by (if any), whether or not a **receive** request is still pending, and what **send** requests may be still pending. See Figure 9 for a sample post-mortem dump produced from an actual Concurrent Pascal-S program.

```
Halt at 26 in process 5 because of illegal channel access


 Process  Status    Blocker    PC    Run    Wait

  0)    sleep     sleep      92     38      6

  1)    send      ch  7      4      6      18

  2)    wait      sem 16     7      4      16

  3)    receive   ch 10      14     8      8

  4)    send      ch  7      19     6      6

  5)    HALT                26     8      0


Global Variables:
NAME       TYPE      OFFSET    VALUE    COMMENT
----       ----      ------    -----    -------
FBsema     special     5         5    Free process blocks still available
chan       channel     7              Owner = process 5
                                      No receives are pending
                                      Sent messages pending:
                                        24 from process 1
                                         5 from process 4
chan2      channel     10             Owner = process 3
                                      Receives are pending
                                      No sent messages are pending
i          integer     13         1
b          boolean     14       TRUE
c          character   15         A
s          semaphore   16         0    Blocking process 2
```

Figure 9: Sample post-mortem dump

System date and time

Another convenience provided for the user is the procedure **DateAndTime**, which is immediately invoked at the start of each Concurrent Pascal-S session. As its name implies, it simply prints the current date and time at the top of the output produced by the kit. This greatly aids the user in sorting and maintaining his or her output records, whether on disk file or hard copy. As for the code of this procedure, the Turbo Pascal procedures **GetDate** and **GetTime** were invoked to

obtain the current system date and time. The results from these procedures were stored in local variables and presented in the most readable way to the user.

User options

At the start of each Concurrent Pascal-S session, the user is allowed to customize the output he or she will receive from the kit. He or she can choose whether or not to include the following information in the output: a listing of the machine code, a trace of the execution, and the contents of **sumtab** and a list of the wait and resumption times of each process. The Boolean variables used to control these actions are **CODELIST**, **TRACE**, and **ShowInternal**. The user's fourth option is whether or not to limit the runtime execution graph to two pages (this is the "adaptive mode" discussed previously). This can be quite a useful feature since the graph for programs of sufficient complexity can sometimes go on for pages. If the user chooses this option, the variable **Adaptive** is set to true and processor activity is shown only at given intervals, such that the output does not exceed two pages (132 lines). See Figure 10 for a display of the actual questions and a set of answers that were supplied by the user.

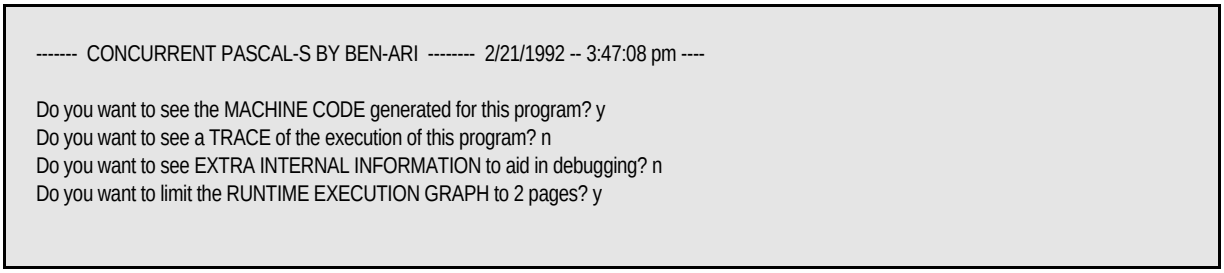


Figure 10: Sample user session

PEDAGOGY

We would like to stress the importance of this programming kit in the world of academics. As stated at the outset, many students are not afforded the opportunity to program in a true concurrent environment. The Concurrent Pascal-S kit enables students to practice and get a better feel for the concurrent programming techniques they have learned in the classroom. Specifically, this kit allows the use of semaphores and synchronous message passing; monitors may also be implemented, requiring the use of semaphores and an extra bit of programming by the user (see Appendix A).

These three paradigms are the most popular and therefore the most widely known methods of problem solving using concurrent programming. They can be used to solve some of the key problems that are very frequently faced in the world of concurrent programming. The bounded buffer is one of these problems.

The bounded buffer problem is also referred to as the producer-consumer problem. In this problem, as the latter name implies, there is a producer process that is continually calculating values and sending them out to a consumer process, which continually takes in the values. In between the two processes is a depository for these values which is called a bounded buffer. This buffer is usually implemented as an array where values can be deposited as well as withdrawn, and the number of elements inside the array grows and shrinks as the producer and consumer go about their duties. The producer can put values into the rear of the buffer as long as there is a slot open, and the consumer can remove values from the front of the buffer whenever there is one present. The question is how to coordinate the two processes so that each can spend as much time as possible doing actual work and as little time as possible waiting to send or receive values. Good discussions of the bounded buffer problem can be found in chapter four of [2] and chapter four of [16].

One solution to this problem is achieved by the use of semaphores, where one semaphore is used to prevent the producer from depositing values into a full buffer and another semaphore is used to prevent the consumer from removing values from an empty buffer. A Concurrent Pascal-S program using this method is shown in Appendix B.

A second solution uses the synchronous message passing scheme, and is presented in Appendix C. Here, in addition to the usual producer and consumer processes, there is a third process (in this case named *Copy*) established to manage the bounded buffer. Using channels and a guarded command, *Copy* is able to input values from the producer as long as there is space available in the buffer and the producer has a value ready for output. Similarly, *Copy* also removes values from the buffer and sends them to the consumer as long as the buffer is nonempty and the consumer is ready for input.

Another problem that is familiar amongst concurrent programmers is the readers and writers problem. This problem involves two kinds of processes: readers and writers. These processes all share access to the same database. The readers merely wish to obtain information from the database; this is a non-destructive action, so many readers may concurrently access the database. The writers, though, wish to read information from the database and update it as well, thereby requiring exclusive access to the database to prevent interference with other readers and writers.

The first solution to the problem uses semaphores, and is shown in chapter four of [2]. Both readers and writers contain critical sections, which are blocks of code that cannot be interrupted during execution by other processes because they use shared resources. One semaphore in this solution is used to protect the critical section between readers, while a second semaphore protects the critical section between readers and writers.

Another solution to the readers and writers problem uses monitors and can be found listed in Appendix A. As previously mentioned, this algorithm was taken from [9] and coded by C.T. Zahn to demonstrate the implementation of monitors in Concurrent Pascal-S. Basically, it uses the example of a banking situation to more concretely embody the actions of readers and writers. Semaphores are then used in the simulation of monitors to provide exclusive access to the database (a bank account, in this case) when necessary.

Yet another concurrent programming paradigm that is continuing to grow in popularity is the method of dataflow programming. Good overviews of this topic are presented in [16] and [5]. Briefly, dataflow programming looks at the whole concept of programming from a different angle: instead of having the program statements and their textual order in the program determine the sequence of execution, why not let as many statements as possible be performed in parallel as soon as the data are ready for the operation? In other words, the data are allowed to "flow" into their operators with results "flowing" out into other operators; it can immediately be seen that dependencies among data need to be identified. As a small example, consider the matrix multiplication program in Appendix D whose algorithm was taken from [16]. Through the use of parallel execution blocks, it can be seen how the two multiplications of the inner product procedure need to be calculated before the addition operation can take place; the data flow from the multiplication to the addition. Also, all four inner products can be calculated at the same time since the result of one does not affect the operation of any other. To contrast this method of matrix multiplication, another solution is given in Appendix E using a message passing approach. This program was originally an Occam program presented in [16] but has been translated to Concurrent Pascal-S.

CONCLUSION

The desire for greater speed in problem-solving is the primary driving force behind concurrent programming research. The Concurrent Pascal-S kit does its best to simulate real-time concurrency. The timing and statistics features added to the implementation are helpful for debugging and efficiency checks, and they give the user a feel for how the program would work on a true concurrent machine. Of course, the same speed is not achieved, but that is not the aim of this exercise (nor is it possible!). Therefore, the Concurrent Pascal-S programmer should always keep in mind the fact that the performance of his or her programs should be measured not by how long the PC spends executing the program, but rather by observing the statistics that are now provided at the end of the run.

We wholeheartedly recommend the use of this kit and others like it to better familiarize students with the principles and practice of concurrent programming!

Appendix A

CPS Program to Implement Monitors As Suggested by C.A.R. Hoare [9] And Coded by C.T. Zahn

```
program rwmon2;  (* rwmon2.in *)

  (*     This is a simplified implementation of
   *  our RWMon program as suggested by Hoare on
   *  page 551 of his CACM article.  It assumes that
   *  all signal operations occur as the last command
   *  in a monitor procedure, and therefore the
   *  urgency variables can be omitted.
   *)

  const trace = true;
        firstbal = 100;  maxchng = 10;
  var mutex : semaphore;
     readcnt : integer;  (* actually reading *)
     readsem, writesem : semaphore;
     okrcnt, okwcnt : integer; (* waiting to access *)
     busy : boolean;  (* someone writing *)
     balance, transcnt,   (* bank account *)
     actors : integer;    (* live updaters *)
     printsem : semaphore;  (* output mutex *)

  function abs( x : integer ) : integer;
  begin
    if x<0 then abs := -x else abs := x
  end;

  procedure initialize;
    var j, k : integer;
  begin
    mutex := 1;  readcnt := 0;
    busy := false;
    readsem := 0;  writesem := 0;
    okrcnt := 0;  okwcnt := 0;
  end;  (* initialize *)

  procedure startread;
  begin
    wait(mutex);
    if busy or (okwcnt > 0) then
       begin
          okrcnt := okrcnt+1;
          signal(mutex);
          wait(readsem);
```

```
        okrcnt := okrcnt-1
      end;
    readcnt := readcnt+1;
    if okrcnt > 0 then signal(readsem)
    else signal(mutex)
  end;  (* startread *)

  procedure endread;
  begin
    wait(mutex);
    readcnt := readcnt-1;
    if (readcnt = 0) and (okwcnt > 0) then
      signal(writesem)
    else signal(mutex)
  end;  (* endread *)

  procedure startwrite;
  begin
    wait(mutex);
    if (readcnt <> 0) or busy then
      begin
        okwcnt := okwcnt+1;
        signal(mutex);
        wait(writesem);
        okwcnt := okwcnt-1
      end;
    busy := true;
    signal(mutex)
  end;  (* startwrite *)

  procedure endwrite;
  begin
    wait(mutex);
    busy := false;
    if okrcnt > 0 then signal(readsem)
    else if okwcnt > 0 then signal(writesem)
    else signal(mutex)
  end;  (* endwrite *)

  procedure debit;
    const reps = 10;
    var k, db : integer;
  begin
    for k := 1 to reps do
      begin
        db := 17*k mod maxchng;
        startwrite;
        balance := balance-db;  transcnt := transcnt+1;
        if trace then
          begin
```

```
            wait(printsem);
            writeln( 'debit ', k, db, balance, transcnt );
            signal(printsem)
          end;
        endwrite
      end;
  actors := actors-1;
  writeln( 'exit from debit ****** ' )
end;  (* debit *)

procedure credit;
  const reps = 15;
  var k, cr : integer;
begin
  for k := 1 to reps do
    begin
      cr := 53*k mod maxchng;
      startwrite;
      balance := balance+cr;  transcnt := transcnt+1;
      if trace then
        begin
          wait(printsem);
          writeln( 'credit ', k, cr, balance, transcnt );
          signal(printsem)
        end;
      endwrite
    end;
  actors := actors-1;
  writeln( 'exit from credit ******** ' )
end;  (* credit *)

procedure audit( n : integer );
  var lastbal, lastcnt, newbal, newcnt, k : integer;
begin
  newbal := firstbal;  newcnt := 0;
  repeat
    lastbal := newbal;  lastcnt := newcnt;
    for k := 1 to 200 do (* delay a bit *) ;
    startread;
    newbal := balance;  newcnt := transcnt;
    endread;
    if trace then
      begin
        wait(printsem);
        writeln( 'audit ', n, lastbal, lastcnt );
        signal(printsem)
      end;
    if abs(newbal-lastbal) > (newcnt-lastcnt)*maxchng then
      writeln( '***** ??? bad balance found by audit.' )
  until actors = 0;
```

```
      wait(printsem);
      writeln( 'exit from audit ', n, ' ********' );
      signal(printsem)
    end;  (* audit *)

begin  (* rwmon *)
  initialize;  balance := firstbal;  transcnt := 0;  actors := 2;
  printsem := 1;
  cobegin
    credit
  | debit
  | audit(1)
  | audit(2)
  | audit(3)
  coend;
  writeln;  writeln( 'All done!' )
end.
```

Appendix B

Bounded Buffer Solution Using Semaphores
As Coded by C.T. Zahn

```
program boundedbuffer;
  const bufsize = 3;  nitems = 10;
        FastP = true;  (* makes producer fast *)
  var q : array[1..bufsize] of integer;
      front, rear, count : integer;
      mutex, filled, empty : semaphore;

  procedure initializeq;
  begin
    count := 0;  front := 1;  rear := 0
  end;  (* initializeq *)

  procedure addtoq( x : integer );
  begin
    count := count+1;
    rear := (rear mod bufsize) + 1;
    q[rear] := x
  end;  (* addtoq *)

  procedure deletefromq( var x : integer );
  begin
    count := count - 1;
    x := q[front];
    front := (front mod bufsize) + 1
  end;  (* deletefromq *)

  procedure produce;
    var item : integer;  t : integer;
  begin
    for item := 1 to nitems do
      begin
        if not FastP then
          for t := 1 to 20 do
            (* nothing *) ;
        wait(empty);
          wait(mutex);
            addtoq( item );
            writeln( item, ' from P' );
          signal(mutex);
        signal(filled)
      end;
      writeln( 'exit P' )
  end;  (* produce *)
```

```
  procedure consume;
    var item, last, t : integer;
  begin
    last := 0;
    repeat
      wait(filled);
        wait(mutex);
          deletefromq( item );
          writeln( item, ' from C' );
        signal(mutex);
      signal(empty);
      if FastP then
        for t := 1 to 20 do
          (* nothing *) ;
      if item = last+1 then
        last := item
      else
        begin
          writeln( 'Bad Q' );
          abort
        end
    until item = nitems;
    writeln( 'exit C' )
  end;  (* consume *)

begin  (* MAIN code *)
  mutex := 1;   (* noone in critical section yet *)
  filled := 0;  empty := bufsize;  (* buffer is empty *)
  initializeq;
  cobegin
    produce
  |
    consume
  coend;
  writeln( ' Both Finished ' )
end.  (* bounded buffer *)
```

Appendix C

Bounded Buffer Solution Using Message Passing
As Suggested by G. Andrews [2]
And Coded by M. Camillone

```
program buff;  (* bounded buffer from Andrews text, p. 428 *)

var East, West : channel;

procedure Producer;
var x : integer;
begin
  while true do
    for x := 1 to 100 do
      send( x, West )
end;  (* Producer *)

procedure Consumer;
var y : integer;
begin
  getchannel( East );
  while true do
    begin
      receive( East, y );
      writeln( y )
    end
end;  (* Consumer *)

procedure Copy;
var buffer : array[1..10] of integer;
   front, rear, count : integer;
begin
  front := 1;
  rear := 1;
  count := 0;
  getchannel( West );
  while true do
    select
      count > 0  & send( buffer[front], East )  ->
        begin
          count := count - 1;
          front := (front mod 10) + 1
        end
    #  count < 10 & receive( West, buffer[rear] )  ->
        begin
          count := count + 1;
          rear := (rear mod 10) + 1
        end
```

```
    end
end;  (* Copy *)

begin
  cobegin
    Producer
  | Consumer
  | Copy
  coend
end.
```

Appendix D

Matrix Multiplication Solution Using Dataflow
As Suggested by R.H. Perrott [16]
And Coded by M. Camillone

```
program matflow;

(* Dataflow matrix multiplication *)
(* See Perrott, p. 244 *)

const N = 2;

type sglary = array[1..N] of integer;
    dblary = array[1..N] of sglary;

var Mat1, Mat2, Ans : dblary;

procedure PrintMatrix( M:dblary; inv:boolean );
var i, j : integer;
begin
  for i := 1 to N do
    begin
      write( '|' );
      for j := 1 to N do
        if inv then write( M[j,i] )
        else write( M[i,j] );
      writeln( ' |' )
    end
end;

function InnerP( X,Y:sglary ) : integer;
var Z : sglary;
begin
  cobegin
    Z[1] := X[1] * Y[1]
  | Z[2] := X[2] * Y[2]
  coend;
  InnerP := Z[1] + Z[2]
end;

procedure MatrixMult( A,B:dblary; var C:dblary );
begin
  cobegin
    C[1,1] := InnerP( A[1],B[1] )
  | C[1,2] := InnerP( A[1],B[2] )
  | C[2,1] := InnerP( A[2],B[1] )
  | C[2,2] := InnerP( A[2],B[2] )
  coend
```

```
end;

begin  (* main *)
  Mat1[1,1] := 1;  Mat1[1,2] := 2;
  Mat1[2,1] := 3;  Mat1[2,2] := 4;
  Mat2[1,1] := 5;  Mat2[1,2] := 7;
  Mat2[2,1] := 6;  Mat2[2,2] := 8;

  MatrixMult( Mat1,Mat2,Ans );

  PrintMatrix( Mat1,false );
  writeln( '   *' );
  PrintMatrix( Mat2,true );
  writeln( '   =' );
  PrintMatrix( Ans,false )
end.
```

Appendix E

Matrix Multiplication Solution Using Message Passing
As Suggested by R.H. Perrott [16]
And Coded by M. Camillone

```
program matmult;

(* Concurrent matrix multiplication program *)
(* From "Parallel Programming," R.H. Perrott, p. 127 *)

const n = 2;

var column, row : array[0..5] of channel;

procedure mult( top,bottom,right,left:integer );
var a,b,t,i : integer;
begin
  t := 0;
  for i := 1 to n do
    begin
      cobegin
        getchannel( column[top] );  receive( column[top], a )
      | getchannel( row[left] );    receive( row[left], b )
      coend;
      t := t + (a * b);
      cobegin
        send( a, column[bottom] );
      | send( b, row[right] )
      coend
    end;
  writeln( t )
end;  (* mult *)

procedure initmatrix;
begin
  cobegin
    send( 2, row[0] )
  | send( 7, column[0] )
  | send( 4, row[1] )
  | send( 8, column[1] )
  coend;
  cobegin
    send( 1, row[0] )
  | send( 5, column[0] )
  | send( 3, row[1] )
  | send( 6, column[1] )
  coend
end;  (* initmatrix *)
```

```
procedure cleanup;
var a,b,c,d,e : integer;
begin
  for e := 1 to n do
    cobegin
      getchannel( row[4] );     receive( row[4], a );
    | getchannel( row[5] );     receive( row[5], b );
    | getchannel( column[4] );  receive( column[4], c );
    | getchannel( column[5] );  receive( column[5], d );
    coend;
end;  (* cleanup *)

begin  (* main *)
  cobegin
    initmatrix
  | mult( 0,2,2,0 )
  | mult( 1,3,4,2 )
  | mult( 2,4,3,1 )
  | mult( 3,5,5,3 )
  | cleanup
  coend
end.  (* main *)
```

Appendix F

Complete Concurrent Pascal-S Session

```
-------  CONCURRENT PASCAL-S BY BEN-ARI  --------  3/9/1992 -- 9:35:08pm ----
USER PROGRAM:
    0 program pipeline;
    0
    0 (* Calculate nCr in pipeline fashion using channels *)
    0
    0 var pipe : array[1..7] of channel;
    0     last : integer;
    0
    0 procedure timer;
    0    var t : integer;
    0 begin
    0    for t := 1 to last do
    4       send( t, pipe[1] );
   12    send( -1, pipe[1] )
   20 end;
   20
   20 procedure choose( left, right : integer );
   21    var ncr, x : integer;
   21 begin
   21    getchannel( pipe[left] );
   25    ncr := 0;
   28    repeat
   28       receive( pipe[left], x );
   35       if x >= 0 then
   39          begin
   39             ncr := ncr + x;
   44             send( ncr, pipe[right] )
   51          end
   51    until x < 0;
   55    send( -1, pipe[right] )
   63 end;
   63
   63 procedure results;
   64    var t, x : integer;
   64 begin
   64    getchannel( pipe[7] );
   68    for t := 1 to last+1 do
   74       begin
   74          receive( pipe[7], x );
   81          writeln( x )
   83       end
   84 end;
   85
   85 begin  (* main *)
   86    writeln( '** Welcome to the n C 7 Calculator **' );
   89    writeln;
   90    write( 'Please enter a value for n : ' );
   92    readln( last );
   95    writeln;
   96    writeln( 'Solution for', last, ' C 7 :' );
  103    last := last - 6;
  108    cobegin
  108       timer
  112    |  choose( 1,2 )
```

```
  122    |  choose( 2,3 )
  131    |  choose( 3,4 )
  140    |  choose( 4,5 )
  149    |  choose( 5,6 )
  158    |  choose( 6,7 )
  167    |  results
  173    coend
  176 end.


Global Variables:
   FBsema     OFFSET:     5
   pipe       OFFSET:     7
   last       OFFSET:    28


CODE:
    0)    0         2         5
    1)   24         0         1
    2)    1         1        28
    3)   14         0        12
    4)   24         0         0
    5)    1         2         5
    6)   24         0        22
    7)   24         0         1
    8)   21         0         1
    9)   41         0         1
   10)   40         0         1
   11)   15         0         4
   12)   24         0         0
   13)   24         0         1
   14)   36         0         0
   15)   24         0        22
   16)   24         0         1
   17)   21         0         1
   18)   41         0         1
   19)   40         0         1
   20)   32         0         0
   21)   24         0        22
   22)    1         2         5
   23)   21         0         1
   24)   42         0         0
   25)    0         2         7
   26)   24         0         0
   27)   38         0         0
   28)   24         0         0
   29)   24         0        22
   30)    1         2         5
   31)   21         0         1
   32)    0         2         8
   33)   41         0         2
   34)   40         0         1
   35)    1         2         8
   36)   24         0         0
   37)   50         0         0
   38)   11         0        51
   39)    0         2         7
   40)    1         2         7
   41)    1         2         8
```

42)	52	0	0
43)	38	0	0
44)	24	0	0
45)	1	2	7
46)	24	0	22
47)	1	2	6
48)	21	0	1
49)	41	0	1
50)	40	0	1
51)	1	2	8
52)	24	0	0
53)	47	0	0
54)	11	0	28
55)	24	0	0
56)	24	0	1
57)	36	0	0
58)	24	0	22
59)	1	2	6
60)	21	0	1
61)	41	0	1
62)	40	0	1
63)	32	0	0
64)	24	0	22
65)	24	0	7
66)	21	0	1
67)	42	0	0
68)	0	2	5
69)	24	0	1
70)	1	1	28
71)	24	0	1
72)	52	0	0
73)	14	0	85
74)	24	0	0
75)	24	0	22
76)	24	0	7
77)	21	0	1
78)	0	2	6
79)	41	0	2
80)	40	0	1
81)	1	2	6
82)	29	0	1
83)	63	0	0
84)	15	0	74
85)	32	0	0
86)	24	0	37
87)	28	0	0
88)	63	0	0
89)	63	0	0
90)	24	0	29
91)	28	0	37
92)	0	1	28
93)	27	0	1
94)	62	0	0
95)	63	0	0
96)	24	0	12
97)	28	0	66
98)	1	1	28
99)	29	0	1
100)	24	0	6
101)	28	0	78

102)	63	0	0
103)	0	1	28
104)	1	1	28
105)	24	0	6
106)	53	0	0
107)	38	0	0
108)	0	1	5
109)	6	0	0
110)	4	0	1
111)	10	0	115
112)	18	0	24
113)	19	0	4
114)	5	0	0
115)	0	1	5
116)	6	0	0
117)	4	0	1
118)	10	0	124
119)	18	0	26
120)	24	0	1
121)	24	0	2
122)	19	0	6
123)	5	0	0
124)	0	1	5
125)	6	0	0
126)	4	0	1
127)	10	0	133
128)	18	0	26
129)	24	0	2
130)	24	0	3
131)	19	0	6
132)	5	0	0
133)	0	1	5
134)	6	0	0
135)	4	0	1
136)	10	0	142
137)	18	0	26
138)	24	0	3
139)	24	0	4
140)	19	0	6
141)	5	0	0
142)	0	1	5
143)	6	0	0
144)	4	0	1
145)	10	0	151
146)	18	0	26
147)	24	0	4
148)	24	0	5
149)	19	0	6
150)	5	0	0
151)	0	1	5
152)	6	0	0
153)	4	0	1
154)	10	0	160
155)	18	0	26
156)	24	0	5
157)	24	0	6
158)	19	0	6
159)	5	0	0
160)	0	1	5
161)	6	0	0

```
  162)    4         0         1
  163)   10         0       169
  164)   18         0        26
  165)   24         0         6
  166)   24         0         7
  167)   19         0         6
  168)    5         0         0
  169)    0         1         5
  170)    6         0         0
  171)    4         0         1
  172)   10         0       176
  173)   18         0        31
  174)   19         0         4
  175)    5         0         0
  176)    9         0         0
  177)   31         0         0


------------- INTERPRET
** Welcome to the n C 7 Calculator **

Please enter a value for n : 17
Solution for    17 C 7 :
     1
     8
    36
   120
   330
   792
  1716
  3432
  6435
 11440
 19448
    -1


RUNTIME SUMMARY:

Process# /
Processor used    Start    Run time    Time waiting    Finish    Elapsed time
    0  /   0        0          56           391          447          447
    1  /   1       25         104           217          346          321
    2  /   2       29         333             0          362          333
    3  /   3       33         333            12          378          345
    4  /   4       37         333            24          394          357
    5  /   5       41         333            36          410          369
    6  /   6       45         333            48          426          381
    7  /   7       49         333            60          442          393
    8  /   8       53         146           247          446          393
--------------              ----          ----                       ----
TOTALS:                     2304          1035                       3339


RELATIVE UTILIZATION PERCENTAGES (across processes):

Process#    Run/ Run+Wait  =  Utilization
    0        56/  56+ 391  =     12.5%
    1       104/ 104+ 217  =     32.4%
    2       333/ 333+   0  =    100.0%
```

```
    3       333/ 333+  12  =     96.5%
    4       333/ 333+  24  =     93.3%
    5       333/ 333+  36  =     90.2%
    6       333/ 333+  48  =     87.4%
    7       333/ 333+  60  =     84.7%
    8       146/ 146+ 247  =     37.2%
--------   --------------       ------
TOTALS:    2304/2304+1035  =     69.0%

GRANULARITY of this application =
ratio of concurrency overhead to actual work being performed =
40 / 2304 =   1.7%
(High percentage = Fine-grained; Low percentage = Coarse-grained)


ABSOLUTE UTILIZATION PERCENTAGES & GRAPHS (across processors):

Processor    Run+Wait/ Run+Wait+Idle  =  Utilization
    0         56+ 391/  56+ 391+   0  =    100.0%
    1        104+ 217/ 104+ 217+ 126  =     71.8%
    2        333+   0/ 333+   0+ 114  =     74.5%
    3        333+  12/ 333+  12+ 102  =     77.2%
    4        333+  24/ 333+  24+  90  =     79.9%
    5        333+  36/ 333+  36+  78  =     82.6%
    6        333+  48/ 333+  48+  66  =     85.2%
    7        333+  60/ 333+  60+  54  =     87.9%
    8        146+ 247/ 146+ 247+  54  =     87.9%
---------   ------------------------       ------
TOTALS:     2304+1035/2304+1035+ 684  =     83.0%

Processor        Utilization Graph
    0


    1                                                  --------------------

    2                                                    ------------------

    3                                                      ----------------

    4                                                        --------------

    5                                                           ------------

    6                                                             ----------

    7                                                               ---------

    8                                                                --------

                = Run                        = Wait                  ----- = Idle
```

RUNTIME GRAPH:

	Processors									
Time	0	1	2	3	4	5	6	7	8	9	10
0	0										
4	0										
8	0										
12	0										
16	0										
20	0										
24	0										
28	0	1									
32	0	1	2								
36	0	1	2	3							
40	0	.	2	3	4						
44	0	.	2	3	4	5					
48	0	1	2	3	4	5	6				
52	0	1	2	.	4	5	6	7			
56	.	.	2	.	.	5	6	7	8		
60	.	.	2	.	.	.	6	7	8		
64	.	.	2	3	.	.	.	7	8		
68	.	.	2	3	.	.	.	.	8		
72	.	.	2	3	.	.	.	.	.		
76	.	1	2	3	.	.	.	.	.		
80	.	1	2	3	4	.	.	.	.		
84	.	.	2	3	4	.	.	.	.		
88	.	.	2	3	4	.	.	.	.		
92	.	.	2	3	4	.	.	.	.		
96	.	.	2	3	4	5	.	.	.		
100	.	.	2	3	4	5	.	.	.		
104	.	1	2	3	4	5	.	.	.		
108	.	1	2	3	4	5	.	.	.		
112	.	.	2	3	4	5	6	.	.		
116	.	.	2	3	4	5	6	.	.		
120	.	.	2	3	4	5	6	.	.		
124	.	.	2	3	4	5	6	.	.		
128	.	1	2	3	4	5	6	7	.		
132	.	1	2	3	4	5	6	7	.		
136	.	.	2	3	4	5	6	7	.		
140	.	.	2	3	4	5	6	7	.		
144	.	.	2	3	4	5	6	7	8		
148	.	.	2	3	4	5	6	7	8		
152	.	.	2	3	4	5	6	7	8		
156	.	1	2	3	4	5	6	7	.		
160	.	1	2	3	4	5	6	7	.		
164	.	.	2	3	4	5	6	7	.		
168	.	.	2	3	4	5	6	7	.		
172	.	.	2	3	4	5	6	7	8		
176	.	.	2	3	4	5	6	7	8		
180	.	.	2	3	4	5	6	7	8		
184	.	1	2	3	4	5	6	7	.		
188	.	1	2	3	4	5	6	7	.		
192	.	.	2	3	4	5	6	7	.		
196	.	.	2	3	4	5	6	7	.		
200	.	.	2	3	4	5	6	7	8		
204	.	.	2	3	4	5	6	7	8		
208	.	.	2	3	4	5	6	7	.		
212	.	1	2	3	4	5	6	7	.		
216	.	1	2	3	4	5	6	7	.		

220	.	.	2	3	4	5	6	7	.
224	.	.	2	3	4	5	6	7	8
228	.	.	2	3	4	5	6	7	8
232	.	.	2	3	4	5	6	7	8
236	.	1	2	3	4	5	6	7	.
240	.	1	2	3	4	5	6	7	.
244	.	.	2	3	4	5	6	7	.
248	.	.	2	3	4	5	6	7	.
252	.	.	2	3	4	5	6	7	8
256	.	.	2	3	4	5	6	7	8
260	.	.	2	3	4	5	6	7	8
264	.	1	2	3	4	5	6	7	.
268	.	1	2	3	4	5	6	7	.
272	.	.	2	3	4	5	6	7	.
276	.	.	2	3	4	5	6	7	.
280	.	.	2	3	4	5	6	7	8
284	.	.	2	3	4	5	6	7	8
288	.	.	2	3	4	5	6	7	8
292	.	1	2	3	4	5	6	7	.
296	.	1	2	3	4	5	6	7	.
300	.	.	2	3	4	5	6	7	.
304	.	.	2	3	4	5	6	7	.
308	.	.	2	3	4	5	6	7	8
312	.	.	2	3	4	5	6	7	8
316	.	.	2	3	4	5	6	7	.
320	.	1	2	3	4	5	6	7	.
324	.	1	2	3	4	5	6	7	.
328	.	.	2	3	4	5	6	7	.
332	.	.	2	3	4	5	6	7	8
336	.	.	2	3	4	5	6	7	8
340	.	.	2	3	4	5	6	7	8
344	.	1	2	3	4	5	6	7	.
348	.		2	3	4	5	6	7	.
352	.		2	3	4	5	6	7	.
356	.		2	3	4	5	6	7	.
360	.		2	3	4	5	6	7	8
364	.			3	4	5	6	7	8
368	.			3	4	5	6	7	8
372	.			3	4	5	6	7	.
376	.			3	4	5	6	7	.
380	.				4	5	6	7	.
384	.				4	5	6	7	.
388	.				4	5	6	7	8
392	.				4	5	6	7	8
396	.					5	6	7	8
400	.					5	6	7	.
404	.					5	6	7	.
408	.					5	6	7	.
412	.						6	7	.
416	.						6	7	8
420	.						6	7	8
424	.						6	7	.
428	.							7	.
432	.							7	.
436	.							7	.
440	.							7	8
444	.								8
447									

Appendix G

Concurrent Pascal-S Source Listing
File 1: CPS.PAS

```
(*$D-,B-,R+,U+    Concurrent Pascal-S       *)
program pascals( input, output );
(* author:   N. Wirth, E.T.H. ch--8092 Zurich, 1.3.76 *)
(* modified: M. Ben-Ari, Tel Aviv Univ, 1980 *)
(*           C. T. Zahn, Pace Univ, 1988 *)
(*           M. Camillone, Pace Univ, 1992 *)
uses DOS, CRT;
const DEBUG = true;  (* to control extra output *)
    WAKESEM = false; (* controls semaphore signal behavior *)
    USEQUANTUM = false;  (* to context switch after quantum or 1 instr. *)
    pmax = 10;   (* max no. of simultaneous processes *)
    stmax = 5000;  (* total stacksize *)
    mainsize = 2000; (* global data in main process *)
    quantmin = 10;   (* minimum quantum size *)
    quantwidth = 10; (* variance of quantum size *)
    single = 1;     (* identifies a single communication request *)
    select = 0;     (*     integer values     *)
    send = 1;       (*          of           *)
    receive = 2;    (* communication commands *)
    purebool = 3;   (* "direction" for guard only containing Boolean *)

    nkw = 27;   (* no. of keywords *)
    alng = 10;   (* no. of significant chars in ident *)
    llng = 70;   (* input line length *)
    kmax = 5;   (* max no. of significant digits *)
    tmax = 70;   (* size of table *)
    bmax = 20;   (* size of block-table *)
    amax = 10;   (* size of array table *)
    cmax = 500;   (* size of code *)
    lmax = 7;   (* maximum level *)
    smax = 500;   (* size of string table *)
    omax = 70;   (* highest order code *)
    xmax = 30000;
    nmax = 30000;
    lineleng = 80;   (* output line length *)
    linelimit = 400;  (* max lines to print *)


type symbol = (intcon, charcon, stringsy,
          notsy, plus, minus, times, idiv, imod, andsy, orsy,
          eql, neq, gtr, geq, lss, leq,
          lparent, rparent, lbrack, rbrack, comma, semicolon,
          period, bar, ampersand, arrow, pound,
          colon, becomes, constsy, typesy, varsy, functionsy,
          proceduresy, arraysy, programsy, ident, selectsy,
          beginsy, cobegsy, ifsy, repeatsy, whilesy, forsy,
          endsy, coendsy, elsesy, untilsy, ofsy, dosy, tosy,
          thensy);
   index = -xmax .. +xmax;
   alfa = packed array [1..alng] of char;
   object = (konstant, variable, type1, prozedure, funktion);
   types = (notyp, ints, bools, chars, semas, chans, arrays);
   er = (erid, ertyp, erkey, erpun, erpar, ernf,
```

```
        erdup, erch, ersh, erln);
    symset = set of symbol;
    typset = set of types;
    item = record typ : types; ref : index; end;
    order = packed record
              f : -omax .. +omax;
              x : -lmax .. +lmax;
              y : -nmax .. +nmax;
            end;
var sy : symbol;   (* last symbol read by insymbol *)
   id : alfa;   (* identifier from insymbol *)
   inum : integer;   (* integer from insymbol *)
   rnum : real;   (* real number from insymbol *)
   sleng : integer;   (* string length *)
   ch : char;  (* last character read from source *)
   line : array[1..llng] of char;
   cc : integer;   (* character counter *)
   lc : integer;  (* program location counter *)
   ll : integer;  (* length of current line *)
   errs : set of er;
   errpos : integer;
   progname : alfa;
   skipflag : boolean;
   constbegsys, typebegsys, blockbegsys, facbegsys,
     statbegsys : symset;
   key : array[1..nkw] of alfa;
   ksy : array[1..nkw] of symbol;
   sps : array[char] of symbol;   (* special symbols *)
   t, a, b, sx, c1, c2, k : integer;   (* indices to tables *)
   stantyps : typset;
   display : array[0..lmax] of integer;
   parseguard : boolean;
   c : char;  (* user's response to questions *)
   Screen, Keys : text;

   tab : array[0..tmax] of      (* identifier table *)
         packed record
           name : alfa;  link : index;
           obj : object;  typ : types;
           ref : index;  normal : boolean;
           lev : 0..lmax;  adr : integer;
         end;
   atab : array[1..amax] of    (* array table *)
          packed record
            inxtyp, eltyp : types;
            elref, low, high, elsize, size : index;
          end;
   btab : array[1..bmax] of    (* block table *)
          packed record
            last, lastpar, psize, vsize : index;
          end;
   stab : packed array[0..smax] of char;   (* string table *)
   code : array[0..cmax] of order;
   CODELIST : boolean; (* for listing of machine code *)
   TRACE : boolean;  (* to trace execution -- wait, signal etc. *)
   ShowInternal : boolean;  (* display sumtab and list of waits & signals *)
   Adaptive : boolean;  (* for Adaptive printout mode of Runtime Graph *)

procedure DumpGlobalNames;
  var k : integer;
begin
```

```
  writeln;  writeln;
  writeln('Global Variables:');
  writeln(' ':3, 'FBsema    ', ' ', 'OFFSET: ', 5:5 );
  for k := 0 to t do with tab[k] do
    if (lev = 1) and (obj = variable) then
      writeln(' ':3, name, ' ', 'OFFSET: ', adr:5 );
  writeln;  writeln;
end;  (* DumpGlobalNames *)

procedure DateAndTime;
  var yr, mon, day, wkdy, hr, min, sec, s100 : word;
begin
  GetDate( yr, mon, day, wkdy );
  GetTime( hr, min, sec, s100 );
  write( ' ', mon, '/', day, '/', yr, ' -- ' );
  if hr = 0 then write( '12:' )
  else if hr > 12 then write( hr - 12, ':' )
  else write( hr, ':' );
  if min < 10 then write( '0' );
  write( min, ':' );
  if sec < 10 then write( '0' );
  write( sec );
  if hr > 11 then write( 'pm' )
  else write( 'am' );
  writeln( ' ----' )
end;  (* DateAndTime *)

(*$I cpslex.pas *)

(* ----------------------------------------------  BLOCK  --- *)

procedure block( fsys : symset; isfun : boolean; level : integer );
  type conrec =
    record tp : types; i : integer end;
  var dx : integer;   (* data allocation index *)
      prt : integer;   (* t-index of this procedure *)
      prb : integer;   (* b-index of this procedure *)
      x : integer;

  procedure skip( fsys : symset; n : er );
  begin
    error(n);  skipflag := true;
    while not ( sy in fsys ) do insymbol;
    if skipflag then endskip
  end;  (* skip *)

  procedure test( s1, s2 : symset; n : er );
  begin
    if not ( sy in s1 ) then skip( s1+s2, n )
  end;  (* test *)

  procedure testsemicolon;
  begin
    if sy = semicolon then insymbol else error( erpun );
    test( [ident]+blockbegsys, fsys, erkey )
  end;  (* testsemicolon *)

  procedure enter( id : alfa; k : object );
    var j, l : integer;
  begin
    if t = tmax then fatal(1)
```

```
  else
    begin
      tab[0].name := id;
      j := btab[display[level]].last;  l := j;
      while tab[j].name <> id do j := tab[j].link;
      if j <> 0 then error( erdup )
      else
        begin
          t := t+1;
          with tab[t] do
            begin
              name := id; link := l;
              obj := k; typ := notyp;
              ref := 0; lev := level; adr := 0
            end;
          btab[display[level]].last := t
        end
    end
end;  (* enter *)

function loc( id : alfa ) : integer;
  var i, j : integer;
begin  (* locate id in table *)
  i := level;  tab[0].name := id;  (* sentinel *)
  repeat
    j := btab[display[i]].last;
    while tab[j].name <> id do j := tab[j].link;
    i := i-1
  until (i<0) or (j<>0);
  if j = 0 then error( ernf );
  loc := j
end;  (* loc *)

procedure entervariable;
begin
  if sy = ident then
    begin enter( id, variable ); insymbol end
  else error( erid )
end;  (* entervariable *)

procedure constant( fsys : symset; var c : conrec );
  var x, sign : integer;
begin
  c.tp := notyp;  c.i := 0;
  test( constbegsys, fsys, erkey );
  if sy in constbegsys then
    begin
      if sy = charcon then
        begin c.tp := chars; c.i := inum; insymbol end
      else
        begin
          sign := 1;
          if sy in [plus,minus] then
            begin
              if sy = minus then sign := -1;
              insymbol
            end;
          if sy = ident then
            begin
              x := loc(id);
              if x <> 0 then
```

```
                if tab[x].obj <> konstant then error( ertyp )
                else
                  begin
                    c.tp := tab[x].typ;
                    c.i := sign*tab[x].adr
                  end;
              insymbol
            end
          else if sy = intcon then
            begin c.tp := ints; c.i := sign*inum; insymbol end
          else skip( fsys, erkey )
        end
    end
end;  (* constant *)

procedure typ( fsys : symset; var tp : types; var rf, sz : integer );
  var x : integer;
      eltp : types;  elrf : integer;
      elsz, offset, t0, t1 : integer;

  procedure arraytyp( var aref, arsz : integer );
    var eltp : types;
        low, high : conrec;
        elrf, elsz : integer;
  begin
    constant( [colon,rbrack,ofsy]+fsys, low );
    if sy = colon then insymbol else error( erpun );
    constant( [rbrack,comma,ofsy]+fsys, high );
    if high.tp <> low.tp then
      begin error( ertyp ); high.i := low.i end;
    enterarray( low.tp, low.i, high.i );
    aref := a;
    if sy = comma then
      begin
        insymbol; eltp := arrays; arraytyp( elrf, elsz )
      end
    else
      begin
        if sy = rbrack then insymbol else error(erpun);
        if sy = ofsy then insymbol else error(erkey);
        typ( fsys, eltp, elrf, elsz )
      end;
    with atab[aref] do
      begin
        arsz := (high-low+1)*elsz;  size := arsz;
        eltyp := eltp;  elref := elrf;  elsize := elsz
      end
  end;  (* arraytyp *)

begin  (* typ *)
  tp := notyp;  rf := 0;  sz := 0;
  test( typebegsys, fsys, erid );
  if sy in typebegsys then
    begin
      if sy = ident then
        begin
          x := loc(id);
          if x <> 0 then
            with tab[x] do
              if obj <> type1 then
                error( ertyp )
```

```
                else
                  begin
                    tp := typ;  rf := ref;  sz := adr;
                    if tp = notyp then error( ertyp )
                  end;
              insymbol
            end
          else if sy = arraysy then
            begin
              insymbol;
              if sy = lbrack then insymbol else error(erpun);
              tp := arrays;  arraytyp( rf, sz )
            end
          else test( fsys, [], erkey )
        end
end;  (* typ *)

procedure parameterlist;  (* formal parameter list *)
  var
    tp : types;
    rf, sz, x, t0 : integer;
    valpar : boolean;
begin
  insymbol;
  tp := notyp;  rf := 0;  sz := 0;
  test( [ident,varsy], fsys+[rparent], erpar );
  while sy in [ident,varsy] do
    begin
      if sy <> varsy then
        valpar := true
      else
        begin
          insymbol;  valpar := false
        end;
      t0 := t;  entervariable;
      while sy = comma do
        begin
          insymbol;  entervariable
        end;
      if sy = colon then
        begin
          insymbol;
          if sy <> ident then error( erid )
          else
            begin
              x := loc(id);  insymbol;
              if x <> 0 then
                with tab[x] do
                  if obj <> type1 then error(ertyp)
                  else
                    begin
                      tp := typ;  rf := ref;
                      if valpar then sz := adr
                      else sz := 1
                    end;
            end;
          test([semicolon,rparent],
               [comma,ident]+fsys, erpun)
        end
      else error(erpun);
      while t0 < t do
```

```
          begin
            t0 := t0+1;
            with tab[t0] do
              begin
                typ := tp;  ref := rf;
                normal := valpar;  adr := dx;
                lev := level;  dx := dx+sz
              end
          end;
        if sy <> rparent then
          begin
            if sy = semicolon then insymbol
            else error(erpun);
            test( [ident,varsy], [rparent]+fsys, erkey )
          end
      end;  (* while *)
    if sy = rparent then
      begin
        insymbol;
        test( [semicolon,colon], fsys, erkey )
      end
    else error(erpun)
  end;  (* parameterlist *)

  procedure constantdeclaration;
    var c : conrec;
  begin
    insymbol;
    test( [ident], blockbegsys, erid );
    while sy = ident do
      begin
        enter( id, konstant );  insymbol;
        if sy = eql then insymbol
        else error(erpun);
        constant( [semicolon,comma,ident]+fsys, c );
        tab[t].typ := c.tp;  tab[t].ref := 0;
        tab[t].adr := c.i;  testsemicolon
      end
  end;  (* constantdeclaration *)

  procedure typedeclaration;
    var tp : types;  rf, sz, t1 : integer;
  begin
    insymbol;
    test( [ident], blockbegsys, erid );
    while sy = ident do
      begin
        enter( id, type1 );  t1 := t;  insymbol;
        if sy = eql then insymbol
        else error(erpun);
        typ( [semicolon,comma,ident]+fsys, tp, rf, sz );
        with tab[t1] do
          begin
            typ := tp;  ref := rf;  adr := sz
          end;
        testsemicolon
      end
  end;  (* typedeclaration *)

  procedure variabledeclaration;
    var t0, t1, rf, sz : integer;  tp : types;
```

```
  begin
    insymbol;
    while sy = ident do
      begin
        t0 := t;  entervariable;
        while sy = comma do
          begin
            insymbol;  entervariable
          end;
        if sy = colon then insymbol
        else error( erpun );
        t1 := t;
        typ( [semicolon,comma,ident]+fsys, tp, rf, sz );
        while t0 < t1 do
          begin
            t0 := t0+1;
            with tab[t0] do
              begin
                typ := tp;  ref := rf;
                lev := level;  adr := dx;
                normal := true;  dx := dx+sz
              end
          end;
        testsemicolon
      end
  end;  (* variabledeclaration *)

  procedure procdeclaration;
    var isfun : boolean;
  begin
    isfun := (sy = functionsy);
    insymbol;
    if sy <> ident then
      begin
        error(erid);  id :=  '         '
      end;
    if isfun then enter(id, funktion)
    else enter(id, prozedure);
    tab[t].normal := true;
    insymbol;
    block( [semicolon]+fsys, isfun, level+1 );
    if sy = semicolon then insymbol
    else error(erpun);
    emit( 32+ord(isfun))   (* exit *)
  end;  (* procdeclaration *)

(* ----------------------------------------------  statement  *)

  procedure statement( fsys : symset );
    var i : integer;  x : item;

    procedure expression( fsys : symset;  var x : item ); forward;

    procedure selector( fsys : symset;  var v : item );
      var x : item;  a, j : integer;
    begin
      if sy <> lbrack then error(ertyp);
      repeat
        insymbol;
        expression( fsys+[comma,rbrack], x );
        if v.typ <> arrays then error(ertyp)
```

```
    else
      begin
        a := v.ref;
        if atab[a].inxtyp <> x.typ then error(ertyp)
        else emit1(21,a);
        v.typ := atab[a].eltyp;
        v.ref := atab[a].elref
      end
  until sy <> comma;
  if sy = rbrack then insymbol else error(erpun);
  test( fsys, [], erkey )
end;  (* selector *)

procedure call( fsys : symset; i : integer );
  var x : item;  lastp, cp, k : integer;
begin
  emit1(18,i);  (* mark stack *)
  lastp := btab[tab[i].ref].lastpar;  cp := i;
  if sy = lparent then
    begin  (* actual parameter list *)
      repeat
        insymbol;
        if cp >= lastp then error(erpar)
        else
          begin
            cp := cp+1;
            if tab[cp].normal then
              begin  (* value parameter *)
                expression(fsys+[comma,colon,rparent], x);
                if x.typ = tab[cp].typ then
                  begin
                    if x.ref <> tab[cp].ref then
                      error(ertyp)
                    else if x.typ = arrays then
                      emit1(22, atab[x.ref].size)
                  end
                else if x.typ <> notyp then error(ertyp)
              end
            else
              begin  (* variable parameter *)
                if sy <> ident then error(erid)
                else
                  begin
                    k := loc(id);  insymbol;
                    if k <> 0 then
                      begin
                        if tab[k].obj <> variable then
                          error(erpar);
                        x.typ := tab[k].typ;
                        x.ref := tab[k].ref;
                        if tab[k].normal then
                          emit2(0,tab[k].lev,tab[k].adr)
                        else
                          emit2(1,tab[k].lev,tab[k].adr);
                        if sy = lbrack then
                          selector(fsys+[comma,colon,
                            rparent],x);
                        if (x.typ<>tab[cp].typ)
                          or (x.ref<>tab[cp].ref) then
                            error(ertyp)
                      end
```

```
                    end
                end
            end;
          test([comma,rparent], fsys, erkey)
        until sy <> comma;
        if sy = rparent then insymbol else error(erpun)
      end;
    if cp < lastp then error(erpar);  (* too few parms *)
    emit1(19, btab[tab[i].ref].psize-1);
    if tab[i].lev < level then
      emit2(3,tab[i].lev, level)
  end;  (* call *)

  function resulttype( a,b : types ) : types;
  begin
    if (a>ints) or (b>ints) then
      begin
        error(ertyp);  resulttype := notyp
      end
    else if (a=notyp) or (b=notyp) then
      resulttype := notyp
    else
      resulttype := ints
  end;  (* resulttype *)

  procedure expression;
    var y : item;  op : symbol;

    procedure simpleexpression(fsys : symset; var x : item);
      var y : item;  op : symbol;

      procedure term(fsys : symset; var x : item);
        var y : item;  op : symbol; ts : typset;

        procedure factor(fsys : symset; var x : item);
          var i,f : integer;
        begin
          x.typ := notyp;  x.ref := 0;
          test(facbegsys, fsys, erpun);
          while sy in facbegsys do
            begin
              if sy = ident then
                begin
                  i := loc(id);  insymbol;
                  with tab[i] do case obj of
                    konstant:
                      begin
                        x.typ := typ;
                        x.ref := 0;
                        emit1(24,adr)
                      end;
                    variable:
                      begin
                        x.typ := typ;
                        x.ref := ref;
                        if sy = lbrack then
                          begin
                            if normal then
                              f := 0
                            else f := 1;
                            emit2(f,lev,adr);
```

```
                    selector(fsys,x);
                    if x.typ in stantyps then
                      emit(34)
                  end
                else
                  begin
                    if x.typ in stantyps then
                      if normal then
                        f := 1
                      else f := 2
                    else
                      if normal then
                        f := 0
                      else f := 1;
                    emit2(f,lev,adr)
                  end
              end;

            type1, prozedure :
              error(ertyp);

            funktion :
              begin
                x.typ := typ;
                if lev <> 0 then
                  call(fsys,i)
                else emit1(8,adr)
              end
          end  (* with case *)
        end
      else if sy in [charcon,intcon] then
        begin
          if sy = charcon then
            x.typ := chars
          else x.typ := ints;
          emit1(24,inum);
          x.ref := 0;  insymbol
        end
      else if sy = lparent then
        begin
          insymbol;
          expression(fsys+[rparent],x);
          if sy = rparent then insymbol
          else error(erpun)
        end
      else if sy = notsy then
        begin
          insymbol;
          factor(fsys,x);
          if x.typ = bools then emit(35)
          else if x.typ <> notyp then
            error(ertyp)
        end;
      test(fsys,facbegsys,erkey)
    end
  end;  (* factor *)

begin  (* term *)
  factor(fsys+[times,idiv,imod,andsy],x);
  while sy in [times,idiv,imod,andsy] do
    begin
```

```
        op := sy;  insymbol;
        factor(fsys+[times,idiv,imod,andsy],y);
        if op = times then
          begin
            x.typ := resulttype(x.typ,y.typ);
            if x.typ = ints then emit(57)
          end
        else if op = andsy then
          begin
            if (x.typ = bools) and (y.typ = bools) then
              emit(56)
            else
              begin
                if (x.typ <> notyp)
                  and (y.typ <> notyp) then
                    error(ertyp);
                x.typ := notyp
              end
          end
        else  (* op in [idiv,imod] *)
          begin
            if (x.typ = ints) and (y.typ = ints) then
              if op = idiv then emit(58)
              else emit(59)
            else
              begin
                if (x.typ <> notyp)
                  and (y.typ <> notyp) then
                    error(ertyp);
                x.typ := notyp
              end
          end
      end
  end;  (* term *)

begin  (* simpleexpression *)
  if sy in [plus,minus] then
    begin
      op := sy;  insymbol;
      term(fsys+[plus,minus],x);
      if x.typ > ints then error(ertyp)
      else if op = minus then emit(36)
    end
  else
    term(fsys+[plus,minus,orsy],x);
  while sy in [plus,minus,orsy] do
    begin
      op := sy;  insymbol;
      term(fsys+[plus,minus,orsy],y);
      if op = orsy then
        begin
          if (x.typ = bools) and (y.typ = bools) then
            emit(51)
          else
            begin
              if (x.typ <> notyp)
                and (y.typ <> notyp) then
                  error(ertyp);
              x.typ := notyp
            end
        end
```

```
        else
          begin
            x.typ := resulttype(x.typ,y.typ);
            if x.typ = ints then
              if op = plus then emit(52)
              else emit(53)
          end
      end
  end;  (* simpleexpression *)

begin  (* expression *)
  simpleexpression(fsys+[eql,neq,lss,leq,gtr,geq],x);
  if sy in [eql,neq,lss,leq,gtr,geq] then
    begin
      op := sy;  insymbol;
      simpleexpression(fsys,y);
      if (x.typ in [notyp,ints,bools,chars])
          and (x.typ = y.typ) then
        case op of
          eql : emit(45);
          neq : emit(46);
          lss : emit(47);
          leq : emit(48);
          gtr : emit(49);
          geq : emit(50)
        end
      else error(ertyp);
      x.typ := bools
    end
end;  (* expression *)

procedure assignment(lv,ad : integer);
  var x,y : item;  f : integer;
  (* tab[i].obj in [variable,prozedure] *)
begin
  x.typ := tab[i].typ;  x.ref := tab[i].ref;
  if tab[i].normal then f := 0 else f := 1;
  emit2(f,lv,ad);
  if sy = lbrack then
    selector([becomes,eql]+fsys,x);
  if sy = becomes then
    insymbol
  else error(erpun);
  if x.typ = semas then
    emit(37);  (* for double store *)
  expression(fsys,y);
  if x.typ = semas then
    begin  (* semaphore initialization *)
      if y.typ = ints then
        begin
          emit(38);
          (* field 2 of sema set to nilproc *)
          emit1(24,1); emit(52);
          emit1(24,-1); emit(38);
        end
      else error(ertyp)
    end
  else if x.typ = y.typ then
    begin
      if x.typ in stantyps then emit(38)
      else if x.ref <> y.ref then error(ertyp)
```

```
      else if x.typ = arrays then
        emit1(23,atab[x.ref].size)
    end
  else error(ertyp)
end;  (* assignment *)

procedure compoundstatement;
begin
  insymbol;
  statement([semicolon,endsy]+fsys);
  while sy in [semicolon]+statbegsys do
    begin
      if sy = semicolon then insymbol
      else error(erpun);
      statement([semicolon,endsy]+fsys)
    end;
  if sy = endsy then insymbol else error(erkey)
end;  (* compoundstatement *)

procedure parblock;
  var lc1 : integer;

  procedure process;
    begin
      insymbol;
      (* wait on FBsema *)
      emit2(0,1,5);  emit(6);
      emit1(4,level);  (* fork new child *)
      lc1 := lc;  emit1(10,0);  (* branch around *)
      statement([semicolon,bar,coendsy]+fsys);
      while sy in [semicolon]+statbegsys do
        begin
          if sy = semicolon then insymbol
          else error(erpun);
          statement([semicolon,bar,coendsy]+fsys)
        end;
      emit(5);  (* kill child *)
      code[lc1].y := lc  (* fix early branch *)
    end;  (* process *)
begin
  process;
  while sy = bar do
    process;
  if sy = coendsy then insymbol else error(erkey);
  emit(9)  (* put parent to sleep *)
end;  (* compoundstatement *)

procedure ifstatement;
  var x : item;  lc1, lc2 : integer;
begin
  insymbol;
  expression(fsys+[thensy,dosy],x);
  if not (x.typ in [bools,notyp]) then error(ertyp);
  lc1 := lc;  emit(11);  (* jmpc *)
  if sy = thensy then insymbol else error(erkey);
  statement(fsys+[elsesy]);
  if sy = elsesy then
    begin
      insymbol;  lc2 := lc;  emit(10);
      code[lc1].y := lc;
      statement(fsys);
```

```
      code[lc2].y := lc
    end
  else
    code[lc1].y := lc
end;  (* ifstatement *)

procedure repeatstatement;
  var x : item;  lc1 : integer;
begin
  lc1 := lc;  insymbol;
  statement([semicolon,untilsy]+fsys);
  while sy in [semicolon]+statbegsys do
    begin
      if sy = semicolon then insymbol
      else error(erpun);
      statement([semicolon,untilsy]+fsys)
    end;
  if sy = untilsy then
    begin
      insymbol;
      expression(fsys,x);
      if not (x.typ in [bools,notyp]) then error(ertyp);
      emit1(11,lc1)
    end
  else error(erkey)
end;  (* repeatstatement *)

procedure whilestatement;
  var x : item;  lc1,lc2 : integer;
begin
  insymbol;  lc1 := lc;
  expression(fsys+[dosy],x);
  if not (x.typ in [bools,notyp]) then error(ertyp);
  lc2 := lc;  emit(11);
  if sy = dosy then insymbol else error(erkey);
  statement(fsys);
  emit1(10,lc1);  code[lc2].y := lc
end;  (* whilestatement *)

procedure forstatement;
  var cvt : types;  x : item;
     i,lc1,lc2 : integer;
begin
  insymbol;
  if sy = ident then
    begin
      i := loc(id);  insymbol;
      if i = 0 then cvt := ints
      else if tab[i].obj = variable then
        begin
          cvt := tab[i].typ;
          if not tab[i].normal then error(ertyp)
          else emit2(0,tab[i].lev,tab[i].adr);
          if not (cvt in [notyp,ints,bools,chars]) then
            error(ertyp)
        end
      else
        begin
          error(ertyp);  cvt := ints
        end
    end
```

```
    else
      skip([becomes,tosy,dosy]+fsys,erid);
    if sy = becomes then
      begin
        insymbol;
        expression([tosy,dosy]+fsys,x);
        if x.typ <>cvt then error(ertyp)
      end
    else
      skip([tosy,dosy]+fsys,erpun);
    if sy = tosy then
      begin
        insymbol;
        expression([dosy]+fsys,x);
        if x.typ <> cvt then error(ertyp)
      end
    else
      skip([dosy]+fsys,erkey);
    lc1 := lc;  emit(14);
    if sy = dosy then insymbol else error(erkey);
    lc2 := lc;
    statement(fsys);
    emit1(15,lc2);  code[lc1].y := lc
  end;  (* forstatement *)

  procedure standproc(n : integer);
    var i,f : integer;  x,y : item;
  begin
    case n of
      1,2 :    (* read *)
        begin
          if sy = lparent then
            begin
              repeat
                insymbol;
                if sy <> ident then error(erid)
                else
                  begin
                    i := loc(id);  insymbol;
                    if i <> 0 then
                      if tab[i].obj <> variable then
                        error(ertyp)
                      else
                        begin
                          x.typ := tab[i].typ;
                          x.ref := tab[i].ref;
                          if tab[i].normal then
                            f := 0
                          else f := 1;
                          emit2(f,tab[i].lev,tab[i].adr);
                          if sy = lbrack then
                            selector(fsys+[comma,rparent],x);
                          if x.typ in [ints,chars,notyp] then
                            emit1(27,ord(x.typ))
                          else error(ertyp)
                        end
                  end;
                test([comma,rparent],fsys,erkey)
              until sy <> comma;
              if sy = rparent then insymbol else error(erpun)
            end;
```

```
    if n = 2 then emit(62)
  end;

3,4 :
  begin   (* write *)
    if TRACE then emit(12);
    if sy = lparent then
      begin
        repeat
          insymbol;
          if sy = stringsy then
            begin
              emit1(24,sleng);  emit1(28,inum);
              insymbol
            end
          else
            begin
              expression(fsys+[comma,colon,rparent],x);
              if not (x.typ in stantyps) then
                error(ertyp);
              emit1(29,ord(x.typ))
            end
        until sy <> comma;
        if sy = rparent then insymbol
        else error(erpun)
      end;
    if n = 4 then emit(63);
    if TRACE then emit(13)
  end;

5,6 :   (* wait, signal *)
  if sy <> lparent then error(erpun)
  else
    begin
      insymbol;
      if sy <> ident then error(erid)
      else
        begin
          i := loc(id);  insymbol;
          if i <> 0 then
            if tab[i].obj <> variable then
              error(ertyp)
            else
              begin
                x.typ := tab[i].typ;
                x.ref := tab[i].ref;
                if tab[i].normal then
                  f := 0
                else f := 1;
                emit2(f,tab[i].lev,tab[i].adr);
                if sy = lbrack then
                  selector(fsys+[rparent],x);
                if x.typ = semas then
                  emit(n+1)
                else error(ertyp)
              end
        end;
      if sy = rparent then insymbol
      else error(erpun)
    end;
```

```
 7 :  (* halt *)
   emit(31);

 8 :  (* abort *)
   emit(17);

 9 : begin  (* getchannel *)
       if sy <> lparent then error(erpun)
       else
         begin
           insymbol;
           if sy <> ident then error(erid)
           else
             begin
               i := loc(id);  insymbol;
               if i <> 0 then
                 if tab[i].obj <> variable then
                   error(ertyp)
                 else
                   begin
                     x.typ := tab[i].typ;
                     x.ref := tab[i].ref;
                     emit1(24,i);  (* load channel *)
                     if sy = lbrack then
                       selector(fsys+[rparent],x);
                     if x.typ = chans then
                       emit(42)
                     else error(ertyp)
                   end
             end;
           if sy = rparent then insymbol
           else error(erpun)
         end
     end;  (* getchannel *)

10 : begin  (* send *)
       if sy <> lparent then error(erpun)
       else
         begin
           if not parseguard then emit(24);  (* sentinel *)
           insymbol;
           if sy = ident then
             begin
               i := loc(id);  insymbol;
               if i <> 0 then
                 if tab[i].obj <> variable then
                   error(ertyp)
                 else
                   begin
                     x.typ := tab[i].typ;
                     x.ref := tab[i].ref;
                     emit2(1,tab[i].lev,tab[i].adr);
                     if sy = lbrack then
                       selector(fsys+[rparent,comma],x)
                   end
             end
           else
             begin
               expression(fsys+[comma,rparent],x);
               if not (x.typ in stantyps) then
                 error(ertyp)
```

```
            end;
          if sy <> comma then error(erpun)
          else insymbol;
          if sy <> ident then error(erid)
          else
            begin
              i := loc(id);  insymbol;
              if i <> 0 then
                if tab[i].obj <> variable then
                  error(ertyp)
                else
                  begin
                    x.typ := tab[i].typ;
                    x.ref := tab[i].ref;
                    emit1(24,i);  (* load channel *)
                    if sy = lbrack then
                      selector(fsys+[rparent],x);
                    if x.typ <> chans then error(ertyp)
                  end
            end;
          emit1(41,send);  (* post *)
          if not parseguard then emit1(40,single);  (* comm *)
          if sy = rparent then insymbol
          else error(erpun)
        end
    end;  (* send *)

11 : begin  (* receive *)
       if sy <> lparent then error(erpun)
       else
         begin
           if not parseguard then emit(24);  (* sentinel *)
           insymbol;
           if sy <> ident then error(erid)
           else
             begin
               i := loc(id);  insymbol;
               if i <> 0 then
                 if tab[i].obj <> variable then
                   error(ertyp)
                 else
                   begin
                     x.typ := tab[i].typ;
                     x.ref := tab[i].ref;
                     emit1(24,i);  (* load channel *)
                     if sy = lbrack then
                       selector(fsys+[comma,rparent],x);
                     if x.typ <> chans then error(ertyp)
                   end
             end;
           if sy <> comma then error(erpun)
           else insymbol;
           if sy <> ident then error(erid)
           else
             begin
               i := loc(id);  insymbol;
               if i <> 0 then
                 if tab[i].obj <> variable then
                   error(ertyp)
                 else
                   begin
```

```
                    x.typ := tab[i].typ;
                    x.ref := tab[i].ref;
                    if tab[i].normal then
                      f := 0
                    else f := 1;
                    emit2(f,tab[i].lev,tab[i].adr);
                    if sy = lbrack then
                      selector(fsys+[rparent],x)
                  end
              end;
            emit1(41,receive);  (* post *)
            if not parseguard then emit1(40,single);  (* comm *)
            if sy = rparent then insymbol
            else error(erpun)
          end
      end;  (* receive *)

  end  (* case *)
end;  (* standproc *)

procedure selectstatement;
  type jmpary = array[0..50] of integer;
  var lc1, i : integer;
     lc3 : jmpary;

  procedure guardcomm( var lc1 : integer; var lc3 : jmpary );
    var lc2 : integer;

    procedure guard( var lc1, lc2 : integer );
      var x : item;
    begin
      parseguard := true;
      if (id = 'send      ') or (id = 'receive   ') then
        begin
          statement( fsys+[arrow] );
          lc2 := lc;
          emit(10);
          lc1 := -1  (* no boolean *)
        end
      else
        begin
          expression( fsys+[ampersand,arrow], x );
          if not (x.typ in [bools,notyp]) then error(ertyp);
          lc1 := lc;
          emit(11);  (* jmpc *)
          if sy = ampersand then
            begin
              insymbol;
              if (id = 'send      ') or (id = 'receive   ') then
                statement( fsys+[arrow] )
              else error(erkey)
            end
          else
            emit(43);  (* post pure Boolean *)
          lc2 := lc;
          emit(10)
        end;
      parseguard := false
    end;  (* guard *)

  begin  (* guardcomm *)
```

```
        guard( lc1, lc2 );
        if sy = arrow then insymbol
        else error(erpun);
        statement( fsys+[pound] );  (* commands *)
        lc3[0] := lc3[0] + 1;
        lc3[lc3[0]] := lc;
        emit(10);  (* jmp to end *)
        code[lc2].y := lc
      end;  (* guardcomm *)

    begin  (* selectstatement *)
      emit1(24,select);  (* enter select statement block - push sentinel *)
      lc3[0] := 0;
      insymbol;
      guardcomm( lc1, lc3 );
      if lc1 <> -1 then
        code[lc1].y := lc;
      while sy = pound do
        begin
          insymbol;
          guardcomm( lc1, lc3 );
          if lc1 <> -1 then
            code[lc1].y := lc
        end;
      if sy = endsy then insymbol
      else error(erkey);
      emit1(40,select);  (* comm *)
      for i := 1 to lc3[0] do
        code[lc3[i]].y := lc
    end;  (* selectstatement *)

  begin  (* statement *)
    if sy in statbegsys+[ident] then
      case sy of
        ident :
          begin
            i := loc(id);  insymbol;
            if i <> 0 then
              case tab[i].obj of
                konstant,type1 : error(ertyp);
                variable : assignment(tab[i].lev,tab[i].adr);
                prozedure :
                  if tab[i].lev <> 0 then call(fsys,i)
                  else standproc(tab[i].adr);
                funktion :
                  if tab[i].ref = display[level] then
                    assignment(tab[i].lev+1,0)
                  else error(ertyp)
              end  (* case *)
          end;

        beginsy : compoundstatement;

        cobegsy : parblock;

        ifsy : ifstatement;

        whilesy : whilestatement;

        repeatsy : repeatstatement;
```

```
          forsy : forstatement;

          selectsy : selectstatement

        end;  (* case *)
      test(fsys,[],erpun)
    end;  (* statement *)

begin  (* block *)
  dx := 5;
  (* global FBsema variable *)
  if level = 1 then dx := dx+2;
  prt := t;
  if level > lmax then fatal(5);
  test([lparent,colon,semicolon],fsys,erpun);
  enterblock;  display[level] := b;  prb := b;
  tab[prt].typ := notyp;  tab[prt].ref := prb;
  if (sy = lparent) and (level > 1) then
    parameterlist;
  btab[prb].lastpar := t;  btab[prb].psize := dx;
  if isfun then
    if sy = colon then
      begin
        insymbol;  (* function type *)
        if sy = ident then
          begin
            x := loc(id);  insymbol;
            if x <> 0 then
              if tab[x].obj <> type1 then
                error(ertyp)
              else if tab[x].typ in stantyps then
                tab[prt].typ := tab[x].typ
              else error(ertyp)
          end
        else skip([semicolon]+fsys,erid)
      end
    else error(erpun);
  if sy = semicolon then insymbol else error(erpun);
  repeat
    if sy = constsy then constantdeclaration;
    if sy = typesy then typedeclaration;
    if sy = varsy then variabledeclaration;
    btab[prb].vsize := dx;
    while sy in [proceduresy,functionsy] do
      procdeclaration;
    test([beginsy],blockbegsys+statbegsys,erkey)
  until sy in statbegsys;
  tab[prt].adr := lc;
  insymbol;  statement([semicolon,endsy]+fsys);
  while sy in [semicolon]+statbegsys do
    begin
      if sy = semicolon then insymbol
      else error(erpun);
      statement([semicolon,endsy]+fsys)
    end;
  if sy = endsy then insymbol else error(erkey);
  test(fsys+[period],[],erkey)
end;  (* block *)

(*$I cpsint.pas *)
```

```
(* ------------------------------------------  MAIN  -------*)

begin  (* main *)
  ClrScr;
  AssignCrt( Screen );  rewrite( Screen );
  AssignCrt( Keys );    reset( Keys );
  CheckEOF := true;
  Assign( input, '' );   reset( input );
  Assign( output, '' );  rewrite( output );

  write( '-------  CONCURRENT PASCAL-S BY BEN-ARI  -------- ' );
  DateAndTime;
  writeln( Screen );
  write( Screen,
       'Do you want to see the MACHINE CODE generated for this program? ');
  readln( Keys, c );
  if (c = 'y') or (c = 'Y') then CODELIST := true
  else CODELIST := false;
  write( Screen,
       'Do you want to see a TRACE of the execution of this program? ' );
  readln( Keys, c );
  if (c = 'y') or (c = 'Y') then TRACE := true
  else TRACE := false;
  write( Screen,
      'Do you want to see EXTRA INTERNAL INFORMATION to aid in debugging? ');
  readln( Keys, c );
  if (c = 'y') or (c = 'Y') then ShowInternal := true
  else ShowInternal := false;
  write( Screen,
       'Do you want to limit the RUNTIME EXECUTION GRAPH to 2 pages? ' );
  readln( Keys, c );
  if (c = 'y') or (c = 'Y') then Adaptive := true
  else Adaptive := false;
  writeln( Screen );
  writeln( 'USER PROGRAM:' );

  key[ 1] := 'and      ';  key[ 2] := 'array    ';
  key[ 3] := 'begin    ';  key[ 4] := 'cobegin  ';
  key[ 5] := 'coend    ';  key[ 6] := 'const    ';
  key[ 7] := 'div      ';  key[ 8] := 'do       ';
  key[ 9] := 'else     ';  key[10] := 'end      ';
  key[11] := 'for      ';  key[12] := 'function ';
  key[13] := 'if       ';  key[14] := 'mod      ';
  key[15] := 'not      ';  key[16] := 'of       ';
  key[17] := 'or       ';  key[18] := 'procedure ';
  key[19] := 'program  ';  key[20] := 'repeat   ';
  key[21] := 'select   ';  key[22] := 'then     ';
  key[23] := 'to       ';  key[24] := 'type     ';
  key[25] := 'until    ';  key[26] := 'var      ';
  key[27] := 'while    ';

  ksy[ 1] := andsy;        ksy[ 2] := arraysy;
  ksy[ 3] := beginsy;      ksy[ 4] := cobegsy;
  ksy[ 5] := coendsy;      ksy[ 6] := constsy;
  ksy[ 7] := idiv;         ksy[ 8] := dosy;
  ksy[ 9] := elsesy;       ksy[10] := endsy;
  ksy[11] := forsy;        ksy[12] := functionsy;
  ksy[13] := ifsy;         ksy[14] := imod;
  ksy[15] := notsy;        ksy[16] := ofsy;
  ksy[17] := orsy;         ksy[18] := proceduresy;
  ksy[19] := programsy;    ksy[20] := repeatsy;
```

```
ksy[21] := selectsy;     ksy[22] := thensy;
ksy[23] := tosy;         ksy[24] := typesy;
ksy[25] := untilsy;      ksy[26] := varsy;
ksy[27] := whilesy;

sps['+'] := plus;        sps['#'] := pound;
sps['('] := lparent;     sps[')'] := rparent;
sps['='] := eql;         sps[','] := comma;
sps['['] := lbrack;      sps[']'] := rbrack;
sps[''] := neq;          sps['&'] := ampersand;
sps[';'] := semicolon;   sps['*'] := times;
sps['|'] := bar;

constbegsys := [plus,minus,intcon,charcon,ident];
typebegsys := [ident,arraysy];
blockbegsys := [constsy,typesy,varsy,proceduresy,
            functionsy,beginsy,cobegsy];
facbegsys := [intcon,charcon,ident,lparent,notsy];
statbegsys := [beginsy,cobegsy,ifsy,selectsy,
           whilesy,repeatsy,forsy];
stantyps := [notyp,ints,bools,chars,semas,chans];
parseguard := false;
lc := 0;  ll := 0;  cc := 0;  ch := ' ';
errpos := 0;  errs := [];  insymbol;
t := -1;  a := 0;  b := 1;  sx := 0;  c2 := 0;
display[0] := 1;
skipflag := false;
if sy <> programsy then error(erkey)
else
  begin
    insymbol;
    if sy <> ident then error(erid)
    else
      begin progname := id;  insymbol end
  end;

enter( '         ',  variable,notyp,0);     (* sentinel *)
enter( 'false    ',  konstant,bools,0);
enter( 'true     ',  konstant,bools,1);
enter( 'char     ',  type1,chars,1);
enter( 'boolean  ',  type1,bools,1);
enter( 'integer  ',  type1,ints,1);
enter( 'semaphore ',  type1,semas,2);
enter( 'channel  ',  type1,chans,3);
enter( 'eof      ',  funktion,bools,17);
enter( 'eoln     ',  funktion,bools,18);
enter( 'read     ',  prozedure,notyp,1);
enter( 'readln   ',  prozedure,notyp,2);
enter( 'write    ',  prozedure,notyp,3);
enter( 'writeln  ',  prozedure,notyp,4);
enter( 'wait     ',  prozedure,notyp,5);
enter( 'signal   ',  prozedure,notyp,6);
enter( 'halt     ',  prozedure,notyp,7);
enter( 'abort    ',  prozedure,notyp,8);
enter( 'getchannel',  prozedure,notyp,9);
enter( 'send     ',  prozedure,notyp,10);
enter( 'receive  ',  prozedure,notyp,11);
enter( '         ',  prozedure,notyp,0);     (*  ???  *)

with btab[1] do
  begin
```

```
      last := t;  lastpar := 1;  psize := 0;  vsize := 0
    end;
  block( blockbegsys+statbegsys, false, 1 );
  if sy <> period then error(erpun);
  if btab[2].vsize > mainsize then error(erln);
  emit(31); (* halt *)
  if not eof(input) then readln( input );
  if errs=[] then
    begin
      if DEBUG then
        DumpGlobalNames;
      if CODELIST then  (* print code *)
        begin
          writeln;
          writeln( 'CODE:' );
          for k := 0 to lc-1 do with code[k] do
            writeln( k:5, ') ', f:4, x:10, y:10 );
          writeln
        end;
      writeln;  write( '---------- INTERPRET ' );
      if WAKESEM then write( 'with semaphore wakeup.' );
      writeln;
      interpret
    end
  else errormsg;
  Close( Screen );  Close( Keys )
end.
```

Appendix H

Concurrent Pascal-S Source Listing
File 2: CPSLEX.PAS

```
(*  CPSLEX.PAS  *)
procedure errormsg;
  var k : er;  msg : array[er] of alfa;
begin
  msg[erid] := 'identifier';  msg[ertyp] := 'type      ';
  msg[erkey] := 'keyword   ';  msg[erpun] := 'punctuatio';
  msg[erpar] := 'parameter ';  msg[ernf] := 'not found ';
  msg[erdup] := 'duplicate ';  msg[erch] := 'character ';
  msg[ersh] := 'too short ';  msg[erln] := 'too long  ';
  writeln( 'compilation errors' );
  writeln;  writeln( 'keywords' );
  for k := erid to erln do if k in errs then
    writeln( ord(k), ' ', msg[k] )
end;  (* errormsg *)

procedure endskip;
begin  (* underline skipped part of input *)
  while errpos < cc do
    begin write( '_' );  errpos := errpos+1 end;
  skipflag := false
end;  (* endskip *)

procedure error( n : er );
begin
  if errpos = 0 then write( ' ****' );
  if cc > errpos then
    begin
      write( ' ': cc-errpos, "", ord(n):2 );
      errpos := cc+3;  errs := errs+[n]
    end
end;  (* error *)

procedure nextch;  (* read ch; process line end *)
begin
  if cc = ll then
    begin
      if ll = llng then error(erln);
      if eof(input) then
        begin
          writeln;  writeln( 'program incomplete');
          errormsg;  halt
        end;
      if errpos <> 0 then
        begin
          if skipflag then endskip;
          writeln;  errpos := 0
        end;
      write( lc:5, ' ' );  ll := 0;  cc := 0;
      while (not eoln(input)) and (ll < llng-2) do
        begin ll := ll+1;  read(input,ch);  write(ch);  line[ll] := ch end;
      writeln;  ll := ll+1;  line[ll] := ' ';
      if not eoln(input) then
        begin ll := ll+1;  line[ll] := ' ' end;  (* to force error *)
```

```
      readln( input )
    end;
  cc := cc+1;  ch := line[cc]
end;  (* nextch *)

procedure fatal( n : integer );
  var msg : array[1..6] of alfa;
begin
  writeln;  errormsg;
  msg[1] := 'identifier';  msg[2] := 'procedures';
  msg[3] := 'strings   ';  msg[4] := 'arrays    ';
  msg[5] := 'levels    ';  msg[6] := 'code      ';
  writeln( ' compiler table for ', msg[n], ' is too small');
  halt
end;  (* fatal *)

(* ---------------------------------------------------  INSYMBOL  ---- *)

procedure insymbol;  (* reads next symbol *)
  label 1,2,3;
  var i, j, k, e : integer;
begin
1: while ch = ' ' do nextch;
  case ch of
    'A' .. 'Z', 'a' .. 'z' :
      begin  (* ident or wordsymbol *)
        k := 0;  id := '          ';
        repeat
          if k < alng then
            begin k := k+1;  id[k] := ch end;
          nextch
        until not ( ch in ['A'..'Z','a'..'z','0'..'9'] );
        i := 1;  j := nkw;  (* binary search *)
        repeat
          k := (i+j) div 2;
          if id <= key[k] then j := k-1;
          if id >= key[k] then i := k+1
        until i > j;
        if i-1 > j then sy := ksy[k] else sy := ident
      end;
    '0' .. '9' :
      begin  (* number *)
        k := 0;  inum := 0;  sy := intcon;
        repeat
          inum := inum*10 + ord(ch)-ord('0');
          k := k+1;  nextch
        until not ( ch in ['0'..'9'] );
        if (k > kmax) or (inum > nmax) then
          begin error(erln); inum := 0; k :=0 end
      end;
    ':' :
      begin
        nextch;
        if ch = '=' then
          begin sy := becomes;  nextch end
        else sy := colon
      end;
    '-' :
      begin
        nextch;
        if ch = '>' then
```

```
        begin sy := arrow;  nextch end
      else sy := minus
    end;
  '<' :
    begin
      nextch;
      if ch = '=' then
        begin sy := leq;  nextch end
      else if ch = '>' then
        begin sy := neq;  nextch end
      else sy := lss
    end;
  '>' :
    begin
      nextch;
      if ch = '=' then
        begin sy := geq;  nextch end
      else sy := gtr
    end;
  '.' :
    begin
      nextch;
      if ch = '.' then
        begin sy := colon;  nextch end
      else sy := period
    end;
  "" :  (* strings *)
    begin
      k := 0;
    2: nextch;
      if ch = "" then
        begin nextch;  if ch <> "" then goto 3 end;
      if sx+k = smax then fatal(3);
      stab[sx+k] := ch;  k := k+1;
      if cc = 1 then
        begin (* end line *) k := 0 end
      else goto 2;
    3: if k = 1 then
        begin sy := charcon;  inum := ord(stab[sx]) end
      else if k = 0 then
        begin error(ersh);  sy := charcon;  inum := 0 end
      else
        begin
          sy := stringsy; inum := sx;
          sleng := k; sx := sx+k
        end
    end;
  '(' :
    begin  (* possible comment *)
      nextch;
      if ch <> '*' then sy := lparent
      else
        begin  (* comment *)
          repeat
            while ch <> '*' do nextch;
            nextch
          until ch = ')';
          nextch;
          goto 1
        end
    end;
```

```
    '+', '&', '*', ')', '=', ',', '[', ']', ';', '|', '#' :
      begin sy := sps[ch];  nextch end;
    else
      begin error(erch);  nextch;  goto 1 end
  end
end;  (* insymbol *)

(* ---------------------------------------------------  ENTER  --- *)

procedure enter( x0 : alfa; x1 : object; x2 : types; x3 : integer );
begin  (* standard identifier *)
  t := t+1;
  with tab[t] do
    begin
      name := x0;  link := t-1;  obj := x1;
      typ := x2;  ref := 0;  normal := true;
      lev := 0;  adr := x3
    end
end;  (* enter *)

procedure enterarray( tp : types; l,h : integer );
begin
  if l > h then error(ertyp);
  if (abs(l)>xmax) or (abs(h)>xmax) then
    begin error(ertyp); l:=0; h :=0 end;
  if a = amax then fatal(4)
  else
    begin
      a := a+1;
      with atab[a] do
        begin inxtyp := tp; low := l; high := h end
    end
end;  (* enterarray *)

procedure enterblock;
begin
  if b = bmax then fatal(2)
  else
    begin
      b := b+1;  btab[b].last := 0;
      btab[b].lastpar := 0
    end
end;  (* enterblock *)

procedure emit( fct : integer );
begin
  if lc = cmax then fatal(6);
  with code[lc] do
    begin f := fct; x := 0; y := 0 end;
  lc := lc+1
end;  (* emit *)

procedure emit1( fct,b : integer );
begin
  if lc =cmax then fatal(6);
  with code[lc] do
    begin f := fct; x := 0; y := b end;
  lc := lc+1
end;  (* emit1 *)

procedure emit2( fct,a,b : integer );
```

```
begin
  if lc = cmax then fatal(6);
  with code[lc] do
    begin f := fct; x := a; y := b end;
  lc := lc+1
end;  (* emit2 *)
```

Appendix I

Concurrent Pascal-S Source Listing
File 3: CPSINT.PAS

```
(* ----------  CPSINT.PAS  ---------- *)

procedure interpret;
const
  (* status of a process block *)
    free = 0; running = 1; ready = 2; semablock = 3; sendblock = 4;
    recblock = 5; selblock = 6; sleeping = 7;
  nilproc = -1;    (* nil pointer for ptype references *)
  main = 0;        (* index of main process *)
  mainquant = 10000; (* quantum for main process *)
  tru = 1;         (* integer value of true *)
  fals = 0;        (* integer value of false *)
  charl = 32;      (* lowest print character ordinal *)
  charh = 126;     (* highest print character ordinal *)
  procmax = 200;   (* maximum number of processes *)
  sentinel = 0;    (* for communication statements *)
type
  ptype = nilproc .. pmax;   (* index over processors *)
  proctype = 0 .. procmax;
  waitptr = ^waitnode;
  waitnode = record
    wait, signal : integer;
    next : waitptr
  end;
  qptr = ^qentry;
  qentry = record
    process : main .. pmax;
    channel : integer;
    direction : send .. purebool;
    comm : integer;    (* expression or var address *)
    resume : integer;  (* resumption address *)
    next,
    setlink : qptr     (* circular set of comm requests *)
  end;
var
  ir : order;     (* instruction buffer *)
  ps :            (* processor status *)
      ( run, timeout, sleep, suspended, kill, wakeup,
        fin, divchk, inxchk, stkchk, procchk, guardchk,
        linchk, lngchk, redchk, chanchk, deadlock, abort );
  lncnt,          (* number of lines *)
  chrcnt : integer;   (* number of characters in line *)
  h1, h2, h3, h4 : integer;    (* local variables *)
  s : array[1..stmax] of integer;      (* the stack *)
  numforks : integer;  (* number of forks performed during program execution *)
  selcount : integer;  (* no. of communications in select statement *)
  setptr : qptr;       (* circular set of comm requests *)

  (* processor table --- one entry for each processor *)
  ptab : array[0..pmax] of
    record  (* process descriptor *)
      procno : proctype;   (* unique process identifier *)
      parent : ptype;
```

```
        childcount : integer;
        status : free .. sleeping;
        blocker : integer;    (* index of semaphore *)
        nextp : ptype;
        t, b,                 (* top, bottom of stack *)
        pc,                   (* program counter *)
        stacklimit : integer;
        display : array[1..lmax] of integer;
        tstart : integer;     (* time process began *)
        timer : integer;      (* no. of instructions executed *)
        tblocked : integer;   (* time process blocked on last wait *)
        totalwait : integer   (* total time spent waiting *)
      end;
    freetop : ptype;    (* top of free blocks list *)
    lastready : ptype;  (* circular ready process queue *)
    lastpr,             (* previous running process *)
    relpr,              (* released process to be waked up *)
    childpr,            (* newly forked child process *)
    runpr : ptype;      (* current running process *)
    stepcount : integer;    (* number of steps before timeout *)
    seed : integer;     (* random seed *)
    lock : boolean;     (* to suppress timeouts *)
    stkincr,            (* stacksize per process *)
    Indent : integer;   (* for trace printouts per process *)
    inserted : boolean;    (* to re-insert a process into Ready Q *)
    before, after : ptype; (* pointers into Ready Q *)
    clock,                 (* clock of process to be inserted *)
    afterclock : integer;  (* clock of process pointed to by after *)
    serviced : boolean;    (* to find a serviceable comm request *)
    last, tmpptr : qptr;

    (* summary table of processes *)
    sumtab : array[proctype] of
      record
        processor_used : 0 .. pmax;
        start,
        running,
        waiting : integer;
        waitlist : waitptr
      end;
    proccount : proctype;   (* index to sumtab *)

    (* channel table *)
    chantab : array[1..tmax] of
      record
        owner : ptype;
        uniqueowner : nilproc .. procmax;
        sendcount : integer;
        sendq,
        receiveq : qptr
      end;

  procedure RuntimeSummary;
    var i : integer;
        runsum, waitsum, elapsum : longint;
  begin
    runsum := 0;  waitsum := 0;  elapsum := 0;
    writeln;
    writeln( 'RUNTIME SUMMARY:' );
    writeln;
    writeln( 'Process# /' );
```

```
  write  ( 'Processor used    Start    Run time    Time waiting    ' );
  writeln( 'Finish    Elapsed time' );
  for i := 0 to proccount do
    with sumtab[i] do
      begin
        writeln( i:5, '/':3, processor_used:4, start:10, running:10,
              waiting:14, start+running+waiting:13, running+waiting:12 );
        runsum := runsum + running;
        waitsum := waitsum + waiting;
        elapsum := elapsum + running + waiting
      end;
  writeln( '--------------', '----':18, '----':14, '----':25 );
  writeln( 'TOTALS:', runsum:25, waitsum:14, elapsum:25 )
end;  (* RuntimeSummary *)

procedure RelUtilization;
  var i : integer;
     runsum, waitsum : longint;
begin
  runsum := 0;  waitsum := 0;
  writeln;
  writeln;
  writeln( 'RELATIVE UTILIZATION PERCENTAGES (across processes):' );
  writeln;
  writeln( 'Process#    Run/ Run+Wait  =  Utilization' );
  for i := 0 to proccount do
    with sumtab[i] do
      begin
        writeln( i:5, running:10, '/', running:4, '+', waiting:4, '=':3,
              running/(running+waiting)*100:9:1, '%' );
        runsum := runsum + running;
        waitsum := waitsum + waiting
      end;
  writeln( '--------', '--------------':17, '------':13 );
  writeln( 'TOTALS:', runsum:8, '/', runsum:4, '+', waitsum:4, '=':3,
        runsum/(runsum+waitsum)*100:9:1, '%' );
  writeln;
  writeln( 'GRANULARITY of this application =' );
  writeln( 'ratio of concurrency overhead to actual work being performed =' );
  write  ( numforks * 5, ' / ', runsum, ' = ' );
  writeln( numforks * 5 / runsum * 100 :5:1, '%' );
  writeln( '(High percentage = Fine-grained; Low percentage = Coarse-grained)')
end;  (* RelUtilization *)

procedure AbsUtilization;
  var used : set of 0..pmax;
     curproc, run, wait, idle, last, i, no_procs : longint;
     runsum, waitsum, idlesum, checksum : longint;
     AbsUtilTab : array[0..pmax] of
       record R, W, I : longint end;

  procedure BarChart;
    var count, len, j, k : integer;
       ratio, rlen : real;
  begin
    ratio := 70 / (sumtab[main].start+sumtab[main].running+
              sumtab[main].waiting);
    writeln;
    writeln( 'Processor       Utilization Graph' );
    for j := 0 to no_procs do
      begin
```

```
        count := 0;
        write( j:5, ' ':5 );
        rlen := AbsUtilTab[j].R * ratio;
        if (rlen < 1) and (rlen > 0) then rlen := 1;
        len := round( rlen );
        for k := 1 to len do
          begin
            count := count + 1;
            if count <= 70 then write( chr(219) )    (* Run *)
          end;
        rlen := AbsUtilTab[j].W * ratio;
        if (rlen < 1) and (rlen > 0) then rlen := 1;
        len := round( rlen );
        for k := 1 to len do
          begin
            count := count + 1;
            if count <= 70 then write( chr(177) )    (* Wait *)
          end;
        for k := count+1 to 70 do write( '-' );       (* Idle *)
        writeln;
        writeln
      end;
    writeln( ' ':10, chr(219), chr(219), chr(219), chr(219), chr(219),
          ' = Run', ' ':17, chr(177), chr(177), chr(177), chr(177),
          chr(177), ' = Wait', ' ':17, '----- = Idle' )
  end;  (* BarChart *)

begin
  no_procs := -1;
  runsum := 0;  waitsum := 0;  idlesum := 0;
  used := [];
  writeln;
  writeln;
  writeln( 'ABSOLUTE UTILIZATION PERCENTAGES & GRAPHS (across processors):' );
  writeln;
  writeln( 'Processor    Run+Wait/ Run+Wait+Idle  =  Utilization' );
  for curproc := 0 to proccount do
    if not (sumtab[curproc].processor_used in used) then
      begin
        run := 0;
        wait := 0;
        idle := 0;
        used := used + [sumtab[curproc].processor_used];
        for i := curproc to proccount do
          if sumtab[i].processor_used = sumtab[curproc].processor_used then
            begin
              if run = 0 then
                idle := idle + sumtab[i].start
              else
                idle := idle + sumtab[i].start - (sumtab[last].start +
                      sumtab[last].running + sumtab[last].waiting);
              last := i;
              run := run + sumtab[i].running;
              wait := wait + sumtab[i].waiting
            end;
        idle := idle + sumtab[main].start + sumtab[main].running +
              sumtab[main].waiting - (sumtab[last].start +
              sumtab[last].running + sumtab[last].waiting);
        writeln( sumtab[curproc].processor_used:5, run:11, '+', wait:4, '/',
              run:4, '+', wait:4, '+', idle:4, '=':3,
              (run+wait)/(run+wait+idle)*100:9:1, '%' );
```

```
          with AbsUtilTab[sumtab[curproc].processor_used] do
            begin
              R := run;  W := wait;  I := idle
            end;
          no_procs := no_procs + 1;
          runsum := runsum + run;
          waitsum := waitsum + wait;
          idlesum := idlesum + idle
        end;
  writeln( '---------', '-------------------------':27, '------':13 );
  writeln( 'TOTALS:', runsum:9, '+', waitsum:4, '/', runsum:4, '+', waitsum:4,
        '+', idlesum:4, '=':3,
        (runsum+waitsum)/(runsum+waitsum+idlesum)*100:9:1, '%' );
  checksum := sumtab[main].start + sumtab[main].running + sumtab[main].waiting;
  for i := 0 to no_procs do   (* consistency check *)
    if (AbsUtilTab[i].R+AbsUtilTab[i].W+AbsUtilTab[i].I) <> checksum then
      writeln( '-- Error in consistency of Run, Wait, & Idle times for ',
            'processor ', i, ' --' );
  BarChart
end;  (* AbsUtilization *)

procedure DisplayGraph;
  var time, endofprog, spaces, dashes, x : integer;
     displayincr : integer;  (* time increment for graph display *)
     slotempty : boolean;
     process : proctype;
     processor : 0 .. pmax;
     ptr : waitptr;
begin
  writeln;
  writeln;
  endofprog := sumtab[main].start+sumtab[main].running+sumtab[main].waiting;
  time := 0;
  if ((endofprog+1) > 132) and Adaptive then
    displayincr := (endofprog+1) div 100
  else
    displayincr := 1;
  writeln( 'RUNTIME GRAPH:' );
  write( '----    ' );
  spaces := pmax * 5 + 1;
  if pmax > 9 then
    spaces := spaces + pmax - 9;
  dashes := spaces - 12;
  for x := 1 to (dashes div 2) do
    write( '-' );
  write( ' Processors ' );
  for x := 1 to (dashes div 2) do
    write( '-' );
  if ((dashes div 2) * 2) <> dashes then
    write( '-' );
  writeln;
  write( 'Time' );
  for processor := 0 to pmax do
    write( '    ', processor );
  writeln;
  write( '----    -' );
  for processor := 1 to pmax do
    begin
      write( '-----' );
      if processor > 9 then write( '-' )
    end;
```

```
  writeln;
  while time < endofprog do
    begin
      write( time:4 );
      for processor := 0 to pmax do
        begin
          slotempty := true;
          process := 0;
          while slotempty do
            begin
              if (sumtab[process].processor_used = processor) and
                (sumtab[process].start <= time) and
                (time < (sumtab[process].start+sumtab[process].running+
                      sumtab[process].waiting)) then
                begin
                  ptr := sumtab[process].waitlist;
                  while (ptr <> nil) and slotempty do
                    if (ptr^.wait <= time) and
                      (time < ptr^.signal) then
                      begin
                        write( '.':5 );
                        slotempty := false
                      end
                    else
                      ptr := ptr^.next;
                  if slotempty then
                    begin
                      write( process:5 );
                      slotempty := false
                    end
                end
              else
                begin
                  process := process + 1;
                  if process > proccount then
                    begin
                      write( ' ':5 );
                      slotempty := false
                    end
                end
            end;
        end;
      time := time + displayincr;
      writeln
    end;
  write( endofprog:4 );
  if ps <> fin then writeln( 'HALT':(runpr*(Indent+1)+Indent+4) )
  else writeln
end;  (* DisplayGraph *)

procedure AddWait( id : ptype; time : integer );
  var ptr : waitptr;
begin
  ptr := sumtab[ptab[id].procno].waitlist;
  if ptr = nil then
    begin
      new( sumtab[ptab[id].procno].waitlist );
      sumtab[ptab[id].procno].waitlist^.wait := time;
      sumtab[ptab[id].procno].waitlist^.next := nil
    end
  else
```

```
    begin
      while ptr^.next <> nil do
        ptr := ptr^.next;
      new( ptr^.next );
      ptr^.next^.wait := time;
      ptr^.next^.next := nil
    end
end;  (* AddWait *)

procedure AddSignal( id : ptype; time : integer );
  var ptr : waitptr;
begin
  ptr := sumtab[ptab[id].procno].waitlist;
  while ptr^.next <> nil do
    ptr := ptr^.next;
  ptr^.signal := time
end;  (* AddSignal *)

procedure DumpWaitlist;
  var ptr : waitptr;
      i : integer;
begin
  writeln;
  writeln( 'Dump Waitlist:' );
  for i := 0 to proccount do
    begin
      write( i:2, ')  ' );
      ptr := sumtab[i].waitlist;
      if ptr = nil then write( 'No waits.' )
      else
        while ptr <> nil do
          begin
            write( ptr^.wait, ' & ', ptr^.signal );
            ptr := ptr^.next;
            if ptr <> nil then write( ' / ' )
          end;
      writeln
    end
end;  (* DumpWaitlist *)

procedure DumpSumtab;
  var i : integer;
begin
  writeln;
  writeln( 'Dump Sumtab:' );
  for i := 0 to proccount do
    with sumtab[i] do
      writeln( 'sumtab[', i:2, ']   processor_used=', processor_used:3,
               '   start=', start:4, '   running=', running:4,
               '   waiting=', waiting:4 )
end;  (* DumpSumtab *)

procedure AddReady( new : ptype );
  (* assumes non-empty Ready Q *)
  var follow : ptype;
begin
  follow := ptab[lastready].nextp;
  ptab[lastready].nextp := new;
  with ptab[new] do
    begin
      nextp := follow;
```

```
        status := ready;
        blocker := 0
      end;
  lastready := new
end;  (* AddReady *)

procedure RemoveReady;
begin
  if lastready = runpr then
    lastready := nilproc (* now empty *)
  else
    ptab[lastready].nextp := ptab[runpr].nextp
end;  (* RemoveReady *)

procedure Suspend( sema : integer );
  var lastw, follow : ptype;
begin
  with ptab[runpr] do begin
    status := semablock;
    blocker := sema;
    tblocked := tstart + timer + totalwait;
    AddWait( runpr, tblocked )
  end;
  lastw := s[sema+1];
  if lastw = nilproc then
    ptab[runpr].nextp := runpr
  else
    begin
      follow := ptab[lastw].nextp;
      ptab[lastw].nextp := runpr;
      ptab[runpr].nextp := follow
    end;
  s[sema+1] := runpr;
end;  (* Suspend *)

procedure Release( sema : integer );  (* assuming someone waits *)
  var lastw : ptype;
      time : integer;
begin
  lastw := s[sema+1];
  relpr := ptab[lastw].nextp;
  if relpr = lastw then
    s[sema+1] := nilproc
  else
    ptab[lastw].nextp :=
      ptab[relpr].nextp;
  with ptab[relpr] do begin
    time := ptab[runpr].tstart + ptab[runpr].timer + ptab[runpr].totalwait;
    tblocked := time - tblocked;
    totalwait := totalwait + tblocked
  end;  (* with *)
  AddSignal( relpr, time )
end;  (* Release *)

procedure EnQueue( tos, reqno : integer; var last : qptr; base : boolean );
  var ptr, delptr : qptr;

  procedure FillNode( node : qptr; var last : qptr );
  begin
    with node^ do
      begin
```

```
        direction := s[tos];
        process := s[tos-2];
        channel := s[tos-3];
        comm := s[tos-4];
        resume := s[tos-1];
        next := nil;
        with ptab[process] do begin
          if reqno = 1 then
            begin
              setptr := node;
              setlink := node;
              last := node
            end
          else if base then
            begin
              last^.setlink := node;
              setlink := setptr
            end
          else
            begin
              last^.setlink := node;
              last := node
            end
        end
      end;  (* FillNode *)

begin  (* EnQueue *)
  if s[tos] = send then
    begin
      while chantab[s[tos-3]].sendq^.channel = -1 do
        begin
          delptr := chantab[s[tos-3]].sendq;
          with chantab[s[tos-3]] do sendq := sendq^.next;
          delptr^.next := nil;
          dispose( delptr )
        end;
      with chantab[s[tos-3]] do begin
        ptr := sendq;
        sendcount := sendcount + 1
      end
    end
  else if s[tos] = receive then
    ptr := chantab[s[tos-3]].receiveq
  else  (* pure Boolean *)
    new( ptr );
  if s[tos] = purebool then
    FillNode( ptr, last )
  else if ptr = nil then
    begin
      new( ptr );
      if s[tos] = send then
        chantab[s[tos-3]].sendq := ptr
      else  (* receive *)
        chantab[s[tos-3]].receiveq := ptr;
      FillNode( ptr, last )
    end
  else
    begin
      while ptr^.next <> nil do
        ptr := ptr^.next;
```

```
        new( ptr^.next );
        FillNode( ptr^.next, last )
      end
end;  (* EnQueue *)

procedure CancelNodes( dirptr : qptr );
  var delptr, start : qptr;
      numlinks, x : integer;
begin
  numlinks := 0;
  start := dirptr;
  repeat
    numlinks := numlinks + 1;
    dirptr := dirptr^.setlink
  until dirptr = start;
  for x := 1 to numlinks do
    if dirptr^.direction = send then
      begin
        with chantab[dirptr^.channel] do sendcount := sendcount - 1;
        dirptr^.channel := -1;
        delptr := dirptr;
        dirptr := dirptr^.setlink;
        delptr^.setlink := delptr
      end
    else if dirptr^.direction = receive then
      begin
        delptr := dirptr;
        dirptr := dirptr^.setlink;
        chantab[delptr^.channel].receiveq := nil;
        dispose( delptr )
      end
    else  (* pure Boolean *)
      begin
        delptr := dirptr;
        dirptr := dirptr^.setlink;
        dispose( delptr )
      end
end;  (* CancelNodes *)

procedure Rendezvous( ptr : qptr );
  var sendptr, recptr : qptr;
      chanid, wakeup1, wakeup2, time : integer;

begin  (* Rendezvous *)
  chanid := ptr^.channel;
  sendptr := chantab[chanid].sendq;
  recptr := chantab[chanid].receiveq;
  s[recptr^.comm] := sendptr^.comm;  (* exchange *)
  time := ptab[runpr].tstart + ptab[runpr].timer + ptab[runpr].totalwait;
  wakeup1 := sendptr^.process;
  wakeup2 := recptr^.process;
  if (ptab[wakeup1].status <> ready) and (ptab[wakeup1].status <> running) then
    begin
      AddReady( wakeup1 );
      with ptab[wakeup1] do
        begin
          tblocked := time - tblocked;
          totalwait := totalwait + tblocked
        end;
      AddSignal( wakeup1, time )
    end;
```

```
  ptab[wakeup1].pc := sendptr^.resume;
  if (ptab[wakeup2].status <> ready) and (ptab[wakeup2].status <> running) then
    begin
      AddReady( wakeup2 );
      with ptab[wakeup2] do
        begin
          tblocked := time - tblocked;
          totalwait := totalwait + tblocked
        end;
      AddSignal( wakeup2, time )
    end;
  ptab[wakeup2].pc := recptr^.resume;
  CancelNodes( sendptr );
  CancelNodes( recptr )
end;  (* Rendezvous *)

procedure WaitforRen( dir, ch : integer );   (* Wait for Rendezvous *)
begin
  RemoveReady;
  with ptab[runpr] do
    begin
      case dir of
        send    : status := sendblock;
        receive : status := recblock;
        select  : status := selblock
      end;
      blocker := ch;
      tblocked := tstart + timer + totalwait;
      AddWait( runpr, tblocked )
    end;
  ps := suspended
end;  (* WaitforRen *)

procedure ReportProcess( id : ptype );
  (* retain vital information of each process before  *)
  (* returning its process block to the free list     *)
begin
  with sumtab[ptab[id].procno] do
    begin
      processor_used := id;
      start := ptab[id].tstart;
      running := ptab[id].timer;
      waiting := ptab[id].totalwait
    end
end;  (* ReportProcess *)

procedure nextproc;
  (* schedule next running process via round-robin *)
begin
  (* circular shift of ready queue *)
  lastready := ptab[lastready].nextp;
end;  (* nextproc *)

procedure setquantum;
begin
  if runpr = main then stepcount := mainquant
  else stepcount := quantmin + trunc(random*quantwidth)
end;  (* setquantum *)

(* functions to convert integers to booleans and conversely *)
```

```
function itob(i : integer) : boolean;
begin
  itob := ( i = tru )
end;

function btoi( b : boolean ) : integer;
begin
  if b then btoi := tru else btoi := fals
end;

procedure PostMortemDump;
  var i, tt : integer;
      ptr : qptr;
begin
  writeln;  writeln;
  with ptab[runpr] do
    write( 'Halt at ', pc, ' in process ', runpr, ' because of ' );
  case ps of
    deadlock : writeln( 'deadlock' );
    guardchk : writeln( 'false guards in select statement' );
    divchk : writeln( 'division by zero' );
    inxchk : writeln( 'invalid index' );
    stkchk : writeln( 'storage overflow' );
    linchk : writeln( 'too much output' );
    lngchk : writeln( 'line too long' );
    redchk : writeln( 'reading past end of file' );
    procchk: writeln( 'too many process forks' );
    chanchk: writeln( 'illegal channel access' );
    abort  : writeln( 'programmer abort' )
  end;
  writeln;  writeln;
  writeln( ' Process  Status   Blocker   PC    Run    Wait ' );
  for i := 0 to pmax do with ptab[i] do
    if status <> free then
      begin
        if (status <> running) and (status <> ready) then
          begin
            tt:=ptab[runpr].tstart+ptab[runpr].timer+
               ptab[runpr].totalwait;
            tblocked := tt - tblocked;
            totalwait := totalwait + tblocked;
            AddSignal( i, tt )
          end;
        ReportProcess( i );
        writeln;
        write( procno:4, ')    ' );
        case status of
          running   : write( 'HALT           ' );
          ready     : write( 'run            ' );
          semablock : write( 'wait     sem ', blocker:2 );
          sendblock : write( 'send     ch ', blocker:2, ' ' );
          recblock  : write( 'receive  ch ', blocker:2, ' ' );
          selblock  : write( 'guard          ' );
          sleeping  : write( 'sleep    sleep ' )
        end;
        writeln( pc:7, timer:7, totalwait:8 )
      end;
  writeln;
  writeln;  writeln( 'Global Variables:' );
  writeln( 'NAME       TYPE      OFFSET    VALUE    COMMENT' );
  writeln( '----       ----      ------    -----    -------' );
```

```
  write  ( 'FBsema      special       5', s[5]:11, ' ':4 );
  if s[6] = -1 then writeln( 'Free process blocks still available' )
  else writeln( 'No free block available for pr ', s[6] );
  for i := btab[1].last+1 to tmax do
    with tab[i] do
      if (lev = 1) and (obj = variable) then
        if typ in stantyps then
          case typ of
            ints : writeln( name, '  integer    ', adr:4, s[adr]:11 );
            bools : writeln( name,'  boolean    ', adr:4,itob(s[adr]):11);
            chars : writeln( name, '  character  ', adr:4,
                        chr(s[adr] mod 256):11 );
            semas : begin
                      write(name,'  semaphore  ',adr:4,s[adr]:11, ' ':4);
                      if s[adr+1] = -1 then
                        writeln( 'Not blocking any process' )
                      else writeln( 'Blocking process ', s[adr+1] )
                    end;
            chans : begin
                      write( name, '  channel    ', adr:4, ' ':15 );
                      if chantab[i].uniqueowner = nilproc then
                        writeln( 'Channel not owned' )
                      else
                        writeln( 'Owner = process ',
                              chantab[i].uniqueowner );
                      write( ' ':42 );
                      if chantab[i].receiveq = nil then write( 'No r' )
                      else write( 'R' );
                      writeln( 'eceives are pending' );
                      write( ' ':42 );
                      if chantab[i].sendcount = 0 then
                        writeln( 'No sent messages are pending' )
                      else
                        begin
                          writeln( 'Sent messages pending:' );
                          ptr := chantab[i].sendq;
                          while ptr <> nil do
                            begin
                              if ptr^.channel <> -1 then
                                writeln( ptr^.comm:48,
                                      ' from process ',
                                      ptr^.process);
                              ptr := ptr^.next
                            end
                        end
                    end
          end
end;  (* PostMortemDump *)

begin  (* interpret *)
  numforks := 0;
  proccount := 0;
  stkincr := (stmax-mainsize) div pmax; (* partition the stack *)
  Indent := 50 div (pmax+1);  (* avoid wrap for tracing *)
  (* initialize the main process *)
  s[1] := 0;  s[2] := 0;  s[3] := -1;  s[4] := btab[1].last;
  with ptab[main] do  (* main process block *)
    begin
      procno := proccount;  (* main process will be 0 *)
      parent := nilproc;  childcount := 0;
      status := running;  blocker := 0;
```

```
    nextp := nilproc;
    b := 0;    display[1] := 0;
    t := btab[2].vsize-1;  pc := tab[s[4]].adr;
    stacklimit := mainsize;
    tstart := 0;
    timer := 0;
    tblocked := 0;
    totalwait := 0
  end;

(* initialize the free block list *)
for h1 := 1 to pmax do with ptab[h1] do
  begin
    if h1 < pmax then nextp := h1+1
    else nextp := nilproc;
    status := free;
    b := ptab[h1-1].stacklimit+1;
    stacklimit := b+stkincr-1;
    timer := 0;
    tblocked := 0;
    totalwait := 0
  end;
freetop := 1;

(* initialize sumtab's waitlist *)
for h1 := 0 to procmax do
  sumtab[h1].waitlist := nil;

(* initialize channel table *)
for h1 := 1 to tmax do with chantab[h1] do
  begin
    owner := nilproc;  (* no one owns this channel *)
    uniqueowner := nilproc;
    sendcount := 0;
    sendq := nil;
    receiveq := nil
  end;

(* initialize ready process queue to main alone *)
lastready := main;
ptab[main].nextp := main;
runpr := main;
if USEQUANTUM then
  setquantum
else
  stepcount := 0;

(* initialize FBsema *)
s[5] := pmax;  s[6] := nilproc;

lock := false;
randomize;
lncnt := 0;  chrcnt := 0;
repeat
  ps := run;
  with ptab[runpr] do
    begin
      ir := code[pc];
      pc := pc+1;
    end;
```

```
with ptab[runpr] do
 begin
  timer := timer + 1;
  case ir.f of

    0 : begin  (* load address *)
          t := t+1;
          if t > stacklimit then ps := stkchk
          else s[t] := display[ir.x] + ir.y
        end;

    1 : begin  (* load value *)
          t := t+1;
          if t > stacklimit then ps := stkchk
          else s[t] := s[display[ir.x] + ir.y]
        end;

    2 : begin  (* load indirect *)
          t := t+1;
          if t > stacklimit then ps := stkchk
          else s[t] := s[s[display[ir.x] + ir.y]]
        end;

    3 : begin  (* update display *)
          h1 := ir.y;  h2 := ir.x;  h3 := b;
          repeat
            display[h1] := h3;  h1 := h1-1;  h3 := s[h3+2]
          until h1 = h2
        end;

    4 : begin  (* fork new child process *)
          numforks := numforks + 1;
          childpr := freetop;
          freetop := ptab[freetop].nextp;
          AddReady( childpr );
          if TRACE then
            writeln( ' ':Indent*runpr,
                  'FORK ', childpr, ' at ', pc );
          proccount := proccount + 1;
          with ptab[childpr] do
            begin
              procno := proccount;
              parent := runpr;
              childcount := 0;
              pc := ptab[runpr].pc+1;  (* after TRA *)
              t := b-1;
              (* inherit parent's environment *)
              tstart := ptab[parent].tstart + ptab[parent].timer +
                        ptab[parent].totalwait;
              timer := 0;  tblocked := 0;  totalwait := 0;
              for h1 := 1 to ir.y do
                display[h1] := ptab[runpr].display[h1];
              ptab[parent].childcount :=
                ptab[parent].childcount+1
            end
        end;

    5 : begin  (* kill terminates child process *)
          if TRACE then
            writeln( ' ':Indent*runpr,
                  'KILL ', runpr, ' at ', pc );
```

```
      ptab[parent].childcount :=
        ptab[parent].childcount-1;
      if (ptab[parent].childcount = 0) and
        (ptab[parent].status = sleeping) then  (* awaken parent *)
        begin
          ptab[parent].status := ready;
          ptab[parent].tblocked := tstart + timer + totalwait -
                                   ptab[parent].tblocked;
          ptab[parent].totalwait := ptab[parent].totalwait +
                                    ptab[parent].tblocked;
          AddSignal( parent, tstart+timer+totalwait );
          AddReady( parent )
        end;
      RemoveReady;
      ReportProcess( runpr );
      (* signal FBsema *)
      if TRACE then
        writeln(' ':Indent*runpr, 'SIGNAL FB',
              ' at ', pc);
      if s[6] = nilproc then
        s[5] := s[5]+1
      else
        begin
          Release( 5 );
          AddReady( relpr );
          if WAKESEM then
            begin  (* awaken waiting process *)
              while ptab[lastready].nextp <> relpr do
                lastready := ptab[lastready].nextp;
              if TRACE then
                writeln( ' ':Indent*runpr, 'WAKEUP ',
                      relpr);
              ps := wakeup
            end
          else if TRACE then
            writeln(' ':Indent*runpr, 'RELEASE ',
                  relpr)
        end;
      ps := kill;
      (* add to free list *)
      status := free;
      nextp := freetop;  freetop := runpr
    end;

  6 : begin  (* semaphore wait *)
      h1 := s[t];  t := t-1;
      if s[h1] > 0 then
        begin
          s[h1] := s[h1]-1;
          if TRACE then
            writeln(' ':Indent*runpr, 'PASS ', h1,
                  ' at ', pc)
        end
      else
        begin  (* suspend running process *)
          RemoveReady;
          Suspend( h1 );
          if TRACE then
            writeln( ' ':Indent*runpr, 'SUSPEND ', h1,
                  ' at ', pc);
          ps := suspended
```

```
        end
    end;

7 : begin  (* semaphore signal *)
      h1 := s[t];  t := t-1;
      if TRACE then
        writeln(' ':Indent*runpr, 'SIGNAL ', h1,
              ' at ', pc);
      if s[h1+1] = nilproc then
        (* noone awaits this signal *)
        s[h1] := s[h1]+1
      else
        begin  (* someone awaits this signal *)
          Release( h1 );
          AddReady( relpr );
          if TRACE then
            writeln( ' ':Indent*runpr, 'Time waiting = ',
                  ptab[relpr].tblocked, ' for ', h1 );
          ptab[relpr].tblocked := 0;
          if WAKESEM then
            begin  (* awaken waiting process *)
              while ptab[lastready].nextp <> relpr do
                lastready := ptab[lastready].nextp;
              if TRACE then
                writeln( ' ':Indent*runpr, 'WAKEUP ',
                      relpr);
              ps := wakeup
            end
          else if TRACE then
            writeln(' ':Indent*runpr, 'RELEASE ',
                  relpr)
        end
    end;

8 : case ir.y of
      17 : begin
             t := t+1;
             if t > stacklimit then ps := stkchk
             else s[t] := btoi(eof(Keys))
           end;
      18 : begin
             t := t+1;
             if t > stacklimit then ps := stkchk
             else s[t] := btoi(eoln(Keys))
           end;
    end;

9 : begin  (* put parent process to sleep *)
      if childcount > 0 then
        begin
          tblocked := tstart + timer + totalwait;
          AddWait( runpr, tblocked );
          RemoveReady;
          if TRACE then
            writeln( ' ':Indent*runpr, 'SLEEP ',
              runpr, ' at ', pc);
          status := sleeping;
          ps := sleep
        end;
    end;
```

```
10 : pc := ir.y;  (* jump *)

11 : begin  (* conditional jump *)
       if s[t] = fals then pc := ir.y;
       t := t-1
     end;

12 : begin  (* block all timeouts *)
       lock := true
     end;

13 : begin  (* allow timeouts again *)
       lock := false
     end;

14 : begin  (* for1up *)
       h1 := s[t-1];
       if h1 <= s[t] then s[s[t-2]] := h1
       else
         begin t := t-3;  pc := ir.y end
     end;

15 : begin  (* for2up *)
       h2 := s[t-2];  h1 := s[h2]+1;
       if h1 <= s[t] then
         begin s[h2] := h1;  pc := ir.y end
       else t := t-3
     end;

17 : ps := abort;  (* abort *)

18 : begin   (* mark stack *)
       h1 := btab[tab[ir.y].ref].vsize;
       if t+h1 > stacklimit then ps := stkchk
       else
         begin t := t+5; s[t-1] := h1-1; s[t] := ir.y end
     end;

19 : begin   (* call *)
       h1 := t-ir.y;
       h2 := s[h1+4];  (*  h2 points to tab *)
       if TRACE then
         writeln( ' ':Indent*runpr,
            'CALL ', tab[h2].name );
       h3 := tab[h2].lev;  display[h3+1] := h1;
       h4 := s[h1+3]+h1;
       s[h1+1] := pc;  s[h1+2] := display[h3];
       s[h1+3] := b;
       for h3 := t+1 to h4 do s[h3] := 0;
       b := h1;  t := h4;  pc := tab[h2].adr
     end;

21 : begin  (* index *)
       h1 := ir.y;   (* h1 points to atab *)
       h2 := atab[h1].low;  h3 := s[t];
       if h3 < h2 then ps := inxchk
       else if h3 > atab[h1].high then ps := inxchk
       else
         begin
           t := t-1;
           s[t] := s[t] + (h3-h2)*atab[h1].elsize
```

```
        end
    end;

22 : begin  (* load block *)
      h1 := s[t];  t := t-1;  h2 := ir.y+t;
      if h2 > stacklimit then ps := stkchk
      else
        while t<h2 do
          begin
            t := t+1; s[t] := s[h1]; h1 := h1+1
          end
    end;

23 : begin  (* copy block *)
      h1 := s[t-1];  h2 := s[t];  h3 := h1+ir.y;
      while h1 < h3 do
        begin
          s[h1] := s[h2];
          h1 := h1+1;  h2 := h2+1
        end;
      t := t-2
    end;

24 : begin  (* literal *)
      t := t+1;
      if t > stacklimit then ps := stkchk
      else s[t] := ir.y
    end;

27 : begin  (* read *)
      if eof(Keys) then ps := redchk
      else
        case ir.y of
          1 : read(Keys,s[s[t]]);
          3 : begin read(Keys,ch); s[s[t]] := ord(ch) end
        end;
      t := t-1
    end;

28 : begin  (* write string *)
      h1 := s[t];  h2 := ir.y;  t := t-1;
      chrcnt := chrcnt+h1;
      if chrcnt > lineleng then ps := lngchk;
      repeat
        write(stab[h2]);
        h1 := h1-1;  h2 := h2+1
      until h1=0
    end;

29 : begin  (* write1 *)
      if ir.y=3 then h1 := 1 else h1 := 10;
      chrcnt := chrcnt+h1;
      if chrcnt > lineleng then ps := lngchk
      else
        case ir.y of
          1 : write(s[t]:6);
          2 : write(itob(s[t]));
          3 : if (s[t] < charl) or (s[t] > charh) then
                ps := inxchk
              else write(chr(s[t]))
        end;
```

```
      t := t-1
    end;

31 : ps := fin;  (* halt *)

32 : begin  (*  exit procedure  *)
      t := b-1;  pc := s[b+1];
      b := s[b+3]
    end;

33 : begin  (* exit function *)
      t := b;  pc := s[b+1];  b := s[b+3]
    end;

34 : s[t] := s[s[t]];

35 : s[t] := btoi(not itob(s[t]));

36 : s[t] := -s[t];

37 : begin  (* duplicate *)
      t := t+1;
      if t>stacklimit then ps := stkchk
      else s[t] := s[t-1]
    end;

38 : begin  (* store *)
      s[s[t-1]] := s[t];  t := t-2
    end;

40 : begin  (* comm *)
      selcount := 0;
      setptr := nil;
      if s[t] = sentinel then
        ps := guardchk
      else if ((s[t] = send) and (chantab[s[t-3]].owner = runpr)) or
          ((s[t]=receive) and (chantab[s[t-4]].owner<>runpr)) then
        ps := chanchk
      else
        begin
          last := nil;
          while s[t] <> sentinel do
            begin
              if s[t] = receive then  (* swap addr & chan *)
                begin
                  h1 := s[t-3];
                  s[t-3] := s[t-4];
                  s[t-4] := h1
                end;
              selcount := selcount + 1;
              EnQueue( t, selcount, last, (s[t-5]=sentinel) );
              t := t-5
            end;
          h1 := random( selcount );
          for h4 := 1 to h1 do
            setptr := setptr^.setlink;  (* pick a comm stmt *)
          tmpptr := setptr;
          serviced := false;
          repeat
            h2 := tmpptr^.channel;
            if h2 = 0 then
```

```
            begin
              pc := tmpptr^.resume;
              serviced := true;
              CancelNodes( tmpptr )
            end
          else if (chantab[h2].sendcount <> 0) and
               (chantab[h2].receiveq <> nil) then
            begin
              Rendezvous( tmpptr );
              serviced := true
            end
          else
            tmpptr := tmpptr^.setlink
        until serviced or (tmpptr = setptr);
        if not serviced then
          if ir.y = select then  (* comm is part of guard *)
            WaitforRen( select, 0 )
          else  (* normal comm *)
            WaitforRen( s[t+5], tab[s[t+2]].adr );
        t := t-1  (* pop sentinel *)
      end
  end;

41 : begin  (* post *)
      s[t+1] := runpr;
      s[t+2] := pc + 1;
      s[t+3] := ir.y;  (* send/receive *)
      t := t+3
  end;

42 : begin  (* getchannel *)
      chantab[s[t]].owner := runpr;
      chantab[s[t]].uniqueowner := ptab[runpr].procno;
      t := t-1
  end;

43 : begin  (* post pure Boolean *)
      s[t+1] := 0;        (* comm *)
      s[t+2] := 0;        (* channel *)
      s[t+3] := runpr;    (* process *)
      s[t+4] := pc + 1;   (* resume *)
      s[t+5] := purebool;  (* direction *)
      t := t+5
  end;

45 : begin t := t-1; s[t] := btoi(s[t]=s[t+1]) end;
46 : begin t := t-1; s[t] := btoi(s[t]<>s[t+1]) end;
47 : begin t := t-1; s[t] := btoi(s[t]<s[t+1]) end;
48 : begin t := t-1; s[t] := btoi(s[t]<=s[t+1]) end;
49 : begin t := t-1; s[t] := btoi(s[t]>s[t+1]) end;
50 : begin t := t-1; s[t] := btoi(s[t]>=s[t+1]) end;
51 : begin
      t := t-1;
      s[t] := btoi(itob(s[t]) or itob(s[t+1]))
  end;
52 : begin t := t-1; s[t] := s[t]+s[t+1] end;
53 : begin t := t-1; s[t] := s[t]-s[t+1] end;
56 : begin
      t := t-1;
      s[t] := btoi(itob(s[t]) and itob(s[t+1]))
  end;
```

```
    57 : begin t := t-1; s[t] := s[t]*s[t+1] end;
    58 : begin
           t := t-1;
           if s[t+1] = 0 then ps := divchk
           else s[t] := s[t] div s[t+1]
         end;
    59 : begin
           t := t-1;
           if s[t+1] = 0 then ps := divchk
           else s[t] := s[t] mod s[t+1]
         end;
    62 : if eof(Keys) then ps := redchk else readln(Keys);
    63 : begin
           writeln;  lncnt := lncnt+1;  chrcnt := 0;
           if lncnt > linelimit then ps := linchk
         end;

   end;  (* case *)
  end;  (* with *)
lastpr := runpr;
if stepcount > 0 then stepcount := stepcount-1;
if lastready = nilproc then
  ps := deadlock
else
  begin
    if (ps = run) and (stepcount = 0) and not lock then
      begin
        if TRACE then
          writeln( ' ':Indent*runpr, 'TIMEOUT at ',
                ptab[runpr].pc);
        ps := timeout;
        if code[ptab[runpr].pc].f = 5 then
          begin
            inserted := false;
            with ptab[runpr] do
              clock := tstart + timer + totalwait;
            (* temporarily remove Runpr from Ready Q *)
            ptab[lastready].nextp := ptab[runpr].nextp;
            before := lastready;
            after := ptab[lastready].nextp;
            while not inserted do
              begin
                with ptab[after] do
                  afterclock := tstart + timer + totalwait;
                if afterclock = clock then
                  begin
                    ptab[runpr].nextp := after;
                    ptab[before].nextp := runpr;
                    inserted := true
                  end
                else
                  begin
                    before := ptab[before].nextp;
                    after := ptab[after].nextp;
                    if before = lastready then
                      begin
                        ptab[runpr].nextp := after;
                        ptab[before].nextp := runpr;
                        inserted := true;
                        nextproc
                      end
```

```
                    end
                  end
                end
              else
                nextproc
            end;
          if ps in [timeout,kill,suspended,sleep,wakeup] then
            begin
              if ps = timeout then
                ptab[runpr].status := ready;
              runpr := ptab[lastready].nextp;
              ptab[runpr].status := running;
              if TRACE then
                begin
                  writeln;
                  writeln( ' ':Indent*runpr, 'RESTART ',
                    runpr, ' at ', ptab[runpr].pc)
                end;
              if USEQUANTUM then setquantum
            end
        end
  until not (ps in [run,timeout,sleep,suspended,kill,wakeup]);

  (*  ps in [fin, xxxchk, or deadlock]  *)

  writeln;
  if ps = fin then
    ReportProcess( main )
  else
    PostMortemDump;

  RuntimeSummary;    (* Print times for all processes *)

  RelUtilization;    (* Print relative utilization percentages *)

  AbsUtilization;    (* Print absolute utilization percentages & graphs *)

  if ShowInternal then
    begin
      DumpSumtab;   (* Print contents of Sumtab *)
      DumpWaitlist  (* Print wait & signal times for each process *)
    end;

  DisplayGraph;      (* Print graphical display of runtime use of processors *)

end;   (* interpret *)
```

References

1. Akl, S.G. *The Design and Analysis of Parallel Algorithms*. Prentice-Hall. Englewood Cliffs, New Jersey. 1989.

2. Andrews, G.R. *Concurrent Programming: Principles and Practice*. Benjamin Cummings. Redwood City, California. 1991.

3. Ben-Ari, M. *Principles of Concurrent Programming*. Prentice-Hall. Englewood Cliffs, New Jersey. 1982.

4. Brinch Hansen, P. "The Programming Language Concurrent Pascal." *IEEE Transactions on Software Engineering*, 1 (Jun. 1975), 199-207.

5. Camillone, M. "Dataflow Programming: An Overview." Term paper for CS 691, Pace University. 1991.

6. Camillone, M. "Parallel Processing and Parallel Programming." Term paper for CS 693, Pace University. 1991.

7. Collado, M., Morales, R., and Moreno, J.J. "A Modula-2 Implementation of CSP." *ACM SIGPLAN Notices*, 22 (Jun. 1987), 25-38.

8. Dijkstra, E.W. "Guarded Commands, Non-determinacy, and Formal Derivation of Programs." *Communications of the ACM*, 18 (Aug. 1975), 453-457.

9. Hoare, C.A.R. "Monitors: An Operating System Structuring Concept." *Communications of the ACM*, 17 (Oct. 1974), 549-557.

10. Hoare, C.A.R. "Communicating Sequential Processes." *Communications of the ACM*, 21 (Aug. 1978), 666-677.

11. Hoare, C.A.R. *Communicating Sequential Processes*. Prentice-Hall. Englewood Cliffs, New Jersey. 1985.

12. INMOS Ltd. *Occam Programming Manual*. Prentice-Hall. Englewood Cliffs, New Jersey. 1984.

13. May, D. "Occam." *ACM SIGPLAN Notices*, 18 (Apr. 1983), 69-79.

14. Morales-Fernandez, R., and Moreno-Navarro, J.J. "CC-Modula: A Modula-2 Tool to Teach Concurrent Programming." *ACM SIGCSE Bulletin*, 21 (Sep. 1989), 19-25.

15. Musciano, C. Letter to the editor. *ACM SIGPLAN Notices*, 22 (Jun. 1987), 2-3.

16. Perrott, R.H. *Parallel Programming*. Addison-Wesley. Wokingham, England. 1987.

17. Pountain, D. "Occam II." *Byte*, 14 (Oct. 1989), 279-284.