CHAPTER 9

MULTI-TASKING

Multi-tasking is very important in real time applications, and is essential for multi-user systems. Although traditionally these uses have required very different operating systems, the multi-tasking features of OS-9 are suited to both without compromise or limitation. Microware have designed a simple and very elegant scheduling algorithm that is quick to execute, gives great flexibility, but is very easy to use. The default method of operation gives a prioritized automatic "round robin" scheduler, but a number of options are available to alter the behaviour of the scheduler. In the most extreme case, the scheduler can be made to operate in a purely hierarchical prioritized mode, such as is commonly found in simple real time kernels.

9.1 OS-9 PROCESS SCHEDULING

All of the process scheduling features of OS-9 are in the kernel module. The aim of the scheduler is to permit multiple processes (tasks) to be requesting processor time, and to divide up the processor time between them. To do this, OS-9 maintains a linked list of the processes that are requesting processor time, known as the "active queue". Each such process will eventually get some processor time, unless one of the pre-emptive features of the scheduler is in use. An important aspect of the OS-9 scheduler is the concept of the "current process". The current process is the process that is actually running now (except for the execution of interrupt service routines). The current process is not in the active queue. It is known because the System Globals field **D_Proc** points to its process descriptor. Therefore the active queue is the list of processes that want processor time, but are not currently receiving it.

Processes not in the active queue are not run by the scheduler. A process must be moved to the active queue to be requesting processor time. A process can only cease to be active by its own request, such as a "wait for event" or a "sleep" (which may be executed from within a system call, such as an I/O call), or if the process is terminated by the kernel in response to a "kill" signal, or a hardware exception, or a signal received by a process that has not installed a signal handler routine. A process is put in the active queue when it is first forked (unless it is forked for debugging), when it receives a signal, or when the condition it is waiting for (such as an event) occurs. This is the function of the **F$AProc** system call.

Scheduling can use the automatic (time-slicing) scheduler, or the pre-emption mechanisms described below, or a mixture. The main work of the scheduler is carried out when a process is put in the active queue. The scheduler must decide at what position within the linked list to insert the new process. As described below, this is done in such a way that the next process to execute is always at the head of the queue. Therefore when the time comes to switch processes (a task switch), the decision of which process to make the current process is a very simple one. The kernel removes the first process in the queue from the linked list, and makes it the current process. This is the function of the **F$NProc** system call.

A process switch (call to **F$NProc**) is only carried out when the current process makes a system call that suspends it (such as a sleep request, perhaps from within a device driver), or the current process dies, or the processor descriptor of the current process is marked as "timed out" when the operating system is about to return to user state (to continue execution of the current process) after a system call or an interrupt. In this last case, the current process is inserted into the active queue exactly as if it had just become active, before the first process in the queue is removed from the queue to become the current process. It is therefore possible for the same process to become the current process again, for example if the process has a high priority. Note that if the active queue is empty (the current process is the only active process), the kernel does not waste time re-inserting the process in the active queue – it ignores the "time out".

The tick routine of the kernel is called on each clock tick interrupt. It decrements the **D_Slice** field of the System Globals, which contains the number of ticks remaining in the time slice of the current process. If this field is now zero, the time slice of the current process has finished. The kernel sets the "timed out" flag (bit 5 of **P$State**) in the process descriptor of the current process. It also resets **D_Slice** to one, rather than zero, in case

the current process is the only active process - it is thus given another tick of processor time.

The actual process switch is not executed until the kernel is about to return to executing the current process in user state, and notices that the current process is timed out. At this time the kernel executes the **F$APROC** system call to put the current process back in the active queue (as described above), and the **F$NPROC** system call to start execution of the next process. This feature ensures that system calls - which are executed in system state - are indivisible. That is, the current process will not be switched out while it is executing a system call, unless the system call explicitly puts the process to sleep.

The **F$NPROC** system call starts the processor running the next process. It removes the first process in the active queue from the linked list, and makes it the current process. It then resets the **D_Slice** field of the System Globals, using the value in **D_TSlice** (ticks per time slice). This initializes the count of the number of ticks in the time slice for the process. If the **F$NPROC** system call finds that there is no process to run, the kernel will execute the 68000 **stop** instruction, causing the processor to stop executing instructions until an interrupt occurs - the kernel then checks the active queue again (the interrupt handler may have woken a process by sending a signal, or changing the value of an event). If a flag is set in the first compatibility byte of the **init** module, the kernel will not execute the **stop** instruction, so it simply loops, checking the active queue until it is not empty (this is to satisfy some processor boards that cannot support the **stop** instruction).

Note that the number of ticks per time slice is typically two. This is because the system can only resolve time to an integral number of ticks. A process switch does not necessarily happen on a tick interrupt. It will also happen if the current process goes to sleep, perhaps to wait for an I/O operation to complete. If the number of ticks per time slice were one, a process could become the current process just before a tick interrupt, and so get very little time. With two ticks per time slice the process will get at least one full tick (unless it goes to sleep, or is pre-empted during its time slice).

9.2 THE SCHEDULER FEATURES

The OS-9 scheduler implements four types of scheduling. These are described briefly below, and then in greater detail in the following sections. Any combination of the scheduling mechanisms can be in use together.

a) "Round robin" automatic scheduling. The processor time is divided into "time slices", and each process in the active queue is given a time slice in turn. A priority value assigned to each process causes high priority processes to receive time slices more frequently than low priority processes.

b) "Minimum process priority" process suspension. Processes with a priority value less than a designated threshold receive no time slices, even if they are in the active queue. The process will receive time slices if the threshold is lowered, or the process's priority is raised (**F$SPrior**), so that the process's priority is no longer below the threshold.

c) Pre-emptive prioritized scheduling. This is the scheduling familiar to real time kernel users. The highest priority active process remains the current process until it ceases to be active, or another process with a higher priority becomes active, or its process priority is reduced so that it is no longer the highest priority process, or the priority of another active process is increased above the priority of the current process. Only processes whose priorities are greater than or equal to a threshold are treated in this way. Processes with a priority below the threshold continue to be scheduled in the "round robin" manner, but receive no time slices while any process is active with a priority greater than or equal to the threshold.

d) Single process pre-emption. This is a mechanism that hands over scheduling to the application programmer. A process is specified as the pre-empting or "seizing" process. Only this process will be given processor time, until the specified process ID is changed, or the mechanism is suspended by specifying a process ID of zero. The scheduler will not give processor time to any other process, even if the pre-empting process goes to sleep, or dies.

9.3 ACTIVATING A PROCESS

As described above, a process is put in the active queue by the **F$APROC** system call. This is a privileged system call (it can only be made from system state), and it is normally only used by other system calls within the kernel, such as the **F$Send** system call (send a signal). Note that the process being put in the active queue may already be active (for example, the current process at the end of its time slice), and may even already be in the active

queue (for example, when a process's priority is changed by the **F$SPrior** system call).

The kernel maintains a "current system active queue age" value in the **D_ActAge** field of the System Globals. This is a long word value, initially set to $7FFF0000, and decremented at the start of each call to the **F$AProc** routine. If it decrements below zero, it is reset to $7FFF0000 (and the process descriptors in the active queue are updated, as described below). The term "system age" is perhaps a little misleading, as this value decreases as the system gets older!

The **F$AProc** routine, called to insert a process in the active queue, decrements the system age, and then calculates a "scheduling constant" (as described below) for the process. It is this scheduling constant that is used to determine the position of the process in the active queue. A process will be placed in the active queue ahead of a process with a lower scheduling constant. Note that a process that is being inserted into the active queue will be inserted *after* any processes with an equal scheduling constant. Once a process has been placed in the active queue its scheduling constant (written to the **P$Sched** field of its process descriptor) is not changed, unless the threshold of one of the scheduling pre-emptive mechanisms is changed, or the priority of the process is changed, or the system age is decremented below zero. In all cases the active queue is kept ordered by scheduling constant.

Normally, the scheduling constant of a process being put in the active queue is calculated by adding the (decremented) system age to the process's priority. Because the system age never exceeds $7FFF0000, and the process priority is a 16-bit word (and so cannot exceed $FFFF), the scheduling constant cannot exceed $7FFFFFFF. The process descriptor of the process is unlinked from any queue it may be in (the sleeping queue, the waiting queue, an event queue, or even the active queue), the new scheduling constant is written to the **P$Sched** field, and the kernel searches the active queue to find the place to insert the process.

The kernel also checks whether the new process has a higher process priority than the current process. If so, it marks the current process as "timed out" (sets bit 5 of the **P$State** field of its process descriptor). This causes the process now at the head of the active queue to become the current process when the currently executing system call or interrupt handler finishes. The aim is to allow a high priority process that has just woken up to immediately become the current process without waiting for the current process to finish

its time slice, unless a higher priority process is already active. This feature can be very important in real time applications.

If any of the three pre-emptive mechanisms is in use, the behaviour of **F$AProc** described above is modified. The following sections describe the effect of the automatic scheduling mentioned above, and the behaviour of the scheduler if the pre-emptive mechanisms are used.

9.4 AUTOMATIC SCHEDULING

The aim of the automatic scheduler is to share the processor's time among multiple processes according to the following principles:

a) Time slices are shared evenly, not given in a block to one process, followed by a block to another process.

b) A process priority mechanism is available, whereby a process of a given priority will receive more time slices than one of a lower priority.

c) All process of the same priority receive the same share of time slices, on a "round-robin" basis.

d) The mechanism must execute quickly, so that scheduling does not consume a significant fraction of the processor's time.

The OS-9 scheduler satisfies the principles described above. In order to execute quickly it uses a simple algorithm (described above) that cannot easily be expressed mathematically. As required, a high priority process receives more time slices than a lower priority process. The relationship depends on the *absolute difference* between the priorities. Thus two processes with priorities of 100 and 105 share time slices in the same ratio as if they had priorities of 50 and 55, or 5 and 10.

The sharing of time slices can be calculated as follows. If the lowest priority active process has priority A, and other active processes have priorities which are B, C, D, and E respectively *greater than A*, then the proportion of time slices going to processes other than process A - for example, process D - is:

$$(D+1)/(2+B+1+C+1+D+1+E+1)$$

while the proportion of time slices going to process A is:

$$2/(2+B+1+C+1+D+1+E+1)$$

This has the interesting corollary that if the lowest priority process (or processes) has a priority only one less than that of the process with the next highest priority, it will get the same proportion of time as that process.

Another important effect of this algorithm is that quite a small difference in priority between processes will produce a large difference in processor time allocation. For example, if two processes are active, and one has a priority that is five higher than the other, the first process will get 5.5 times has much processor time as the second process. However, this is not usually of great importance, as a typical multi-tasking real time application will have a group of low priority processes, all with the same low priority, and a group of high priority processes, all with the same much higher priority.

9.5 AN EXAMPLE OF SCHEDULING

Below is shown an example of time slicing between three processes. It gives an empirical demonstration of how the processor time would be divided between three compute-bound processes[11] at different priorities.

The system age starts at 60. There are three processes, two of priority 10, and one of priority 8. All three are continuously active during the 10 time slices observed. The top row of the table shows the system age at each successive time slice. It is decremented by one each time as the current process is put back in the active queue. The three rows below show the scheduling constant for each of the processes at each time slice. The current process is marked with a '►'. The priority of the process is shown at the left of the row, and the total number of time slices for which the process was the current process is shown at the right of the row. Note that at each time slice only the scheduling constant of the process that was the current process in the previous process is recalculated – because the current process is put back in the active queue before the first process in the active queue becomes the new current process.

	System Age										
Priority	59	58	57	56	55	54	53	52	51	50	Slices
10	69	► 69	67	67	► 67	64	64	► 64	61	► 61	4
10	► 70	68	68	► 68	65	► 65	63	63	► 63	60	4
8	68	68	► 68	64	64	64	► 64	60	60	60	2

11 Processes that are continuously working, and do not ask to go to sleep.

Remember that the scheduling constant is calculated by adding the system age to the process's priority, and that if the scheduling constant of a process being put in the active queue is equal to that of a process already in the queue, the new process is put in the queue behind the process already in the queue. For the sake simplicity, the above example assumes that all three processes were initially put in the queue at the same system age (60). In fact, because the system age is decremented before a process is put in the queue, this would not happen in practice.

The example shows two important results. Firstly, the two processes of the same priority received the same number of time slices, while the lower priority process received less time slices. And secondly, the time slices were very evenly distributed between the processes. This illustrates that despite using a very simple algorithm, the OS-9 kernel achieves the aims of an automatic round robin scheduler.

9.6 SCHEDULING PRE-EMPTION MECHANISMS

OS-9 provides three mechanisms for the programmer to pre-empt the round robin scheduler. The mechanisms are all controlled by changing user-writable values in the System Globals. This is done using the **F$SetSys** system call (made by the C library function **_setsys()**). This system call must be used to modify these controlling variables, even if the system is not using the SSM, (in which case the program could write to the variables directly). This is because the kernel takes action when these variables are changed, to ensure a correct change in the behaviour of the scheduler.

9.6.1 Minimum Priority

This facility allows a group of low priority processes to be suspended (given no processor time), and re-activated at a later time. This mechanism uses the System Globals field **D_MinPty**. If a process with a priority less than the value in this field is put in the active queue (by the **F$APROC** system call), its scheduling constant is set to zero, rather than calculating the scheduling constant from the system age and the process's priority. (Note that the normal calculation is used if the process is in system state - to allow it to complete a system call). Because the process's scheduling constant is zero, it is put at the tail of the active queue, along with the other processes whose priorities are below the "minimum priority".

The **F$NProc** system call checks the priority of the process it is about to make the current process. If the priority is less than the value in the

D_MinPty field, it marks the process's process descriptor as "timed out" (bit 5 set in the **P$State** field. In addition, if the process is not in system state, the kernel calls the **F$AProc** routine to re-insert the process in the active queue (which will set its scheduling constant to zero, and put it at the tail of the queue), and takes the next process from the head of the queue to be the current process. The process is marked as "timed out" so that if it is in system state (processing a system call), a task switch will occur as soon as the system call finishes. This allows the process to finish a system call (which must be permitted, otherwise system resources could be locked up), but not to execute any more of its program.

In this way, any process that was already in the active queue before the **D_MinPty** field was set above its priority is allowed to finish any system call it is executing, and then is re-inserted in the active queue, with a scheduling constant of zero. If the **F$NProc** routine finds that the process at the head of the active queue has a scheduling constant of zero, it acts as if the active queue were empty, by suspending the processor's execution of instructions. It does not need to check the rest of the queue, as the queue is always kept sorted by scheduling constant, so any other processes in the queue must also have a scheduling constant of zero.

From OS-9 version 2.3 onwards, if the kernel finds on task switch that the current process is the only active process, but its priority is less than the value in **D_MinPty**, it re-inserts the process in the active queue, and calls the **F$NProc** routine to activate the next process. As there is no other process in the active queue, this causes the current process to be suspended (its priority is less than **D_MinPty**), and processor execution to be suspended. This guarantees that processes with a priority below **D_MinPty** are immediately suspended (after completing any system call). This may be needed to prevent these processes making a system call that takes some time to execute, possibly impairing the real time response of the high priority processes. Prior to OS-9 version 2.3, if the current process was the only active process it continued execution, even if its priority became less than **D_MinPty**.

The result of this algorithm, in conjunction with the fact that the current process is marked as "timed out" if a higher priority process is put in the active queue, is that a high priority process can set the **D_MinPty** field to immediately suspend a group of low priority processes, and then allow them to run at a later time by clearing the **D_MinPty** field.

To ensure that the low priority processes are re-activated when the **D_MinPty** threshold is lowered, the **F$SetSys** system call (used to change

fields in the System Globals) takes special action if this field is being changed, and the new value is less than the present value. It scans through the active queue, and re-inserts any process whose current scheduling constant is zero (using the **F$AProc** routine), causing its scheduling constant to be recalculated. It is therefore essential that the **D_MinPty** field is changed using the **F$SetSys** system call or the **_setsys()** C library function, rather than by directly writing to the System Globals, as otherwise the low priority processes will never be re-activated. Note that a process is simply re-inserted in the active queue when the "minimum priority" is lowered - it is not necessarily re-activated, because its priority may still be below the "minimum priority". This permits any number of groups of processes at different priority levels to be suspended and re-activated in a hierarchy.

9.6.2 Maximum Age

The term "maximum age" used to refer to this mechanism is something of a misnomer, as the mechanism acts on the process's priority, not its "age". (See the chapter on the OS-9 Internal Structure for a discussion of a process's "age", which is a value invented only when a copy of the process descriptor is requested.)

The "maximum age" field in the System Globals - **D_MaxAge** - sets a threshold. A process with a priority less than this threshold is scheduled in the normal "round robin" way, while processes with priorities greater than or equal to the threshold are scheduled in a strictly prioritized manner. If the **D_MaxAge** field is zero (the default on startup), this mechanism is disabled.

If **D_MaxAge** is not zero, and a process has a priority greater than or equal to the threshold, then the **F$AProc** routine calculates its scheduling constant in a different way. Instead of adding the process priority to the current system age, it adds the process priority to $80000000. As described above, the normal method of calculating the scheduling constant cannot produce a result greater than $7FFFFFFF. Therefore all processes in the group with priorities equal to or above the threshold will always have scheduling constants greater than all processes in the lower group, and so any process in the upper group will be inserted in the active queue ahead of all processes in the lower group.

The first effect of this mechanism is that processes in the lower group will not run so long as any process in the higher group is active. The second effect is that the processes in the upper group that are active are always ordered strictly by priority in the active queue, irrespective of how much processor

time they have already used. This means that the highest priority active process will always be the current process. It must cease to be active (or have its priority changed) in order for the process with the next highest priority to become the current process. Therefore processes in the upper group are subject to a pre-emptive prioritized scheduling mechanism - there is no "round robin" distribution of processor time. This is the scheduler familiar to users of real time kernels.

Changing an active process's priority (using the **F$SPrior** system call,or **setpr()** C library function) causes it to be re-inserted in the active queue, and if a process with a higher priority than the current process is inserted in the active queue, then the current process is marked as "timed out". Therefore, as with the "minimum priority" mechanism, processes in the upper group immediately pre-empt processes in the lower group. That is, if the current process is a process in the lower group, and a process in the upper group becomes active, the time slice of the current process is immediately terminated.

The **F$SetSys** system call checks whether the **D_MaxAge** field is being changed. If so, it calls the **F$APROC** routine to re-insert every active process back into the active queue. This ensures that a change in threshold is immediately acted upon, with a re-ordering of the upper and lower groups. Also, if the current process is now in the lower group, and any process in the upper group is active, the current process is marked as "timed out", as described above. It is therefore essential that **D_MaxAge** is changed by using the **F$SetSys** system call or the **_setsys()** C library function, rather than by writing directly to the System Globals.

9.6.3 Seizing Control

This mechanism completely pre-empts the scheduler, leaving all scheduling to be done by the application. It uses the System Globals field **D_Sieze** (note the spelling). The mechanism is enabled if this field is not zero, and is disabled again if the field is set to zero. When the mechanism is enabled, the **D_Sieze** field is assumed to contain the ID of a process. When the process with this ID is put in the active queue by the **F$APROC** system call it is given a scheduling constant of $FFFFFFFF, forcing it to the front of the queue. As described above, if it also has a higher priority than the current process, the current process is marked as "timed out".

When the current process is switched out, and the kernel looks for the next process to run, the **F$NProc** system call will make the "seizing" process the current process (because it is at the head of the queue). At the end of its time

slice the process will again become the current process, because the **F$APROC** routine will again force it to the front of the active queue. In addition, if the process goes to sleep (or even if it dies!), the **F$NProc** routine will refuse to run any other process, and will suspend execution just as if the active queue were empty. This mechanism therefore leaves the scheduling entirely in the hands of the programmer, and clearly it must be used with extreme care. Indeed, because of the dangers involved, this mechanism should only be used if there is absolutely no alternative, which is extremely rare.

The **F$SetSys** system call takes no special action when the **D_Sieze** variable is changed.

9.6.4 The Precedence of the Mechanisms

Although any or all of the mechanisms described above can be activated at any one time, in some respects they are clearly in conflict. It is therefore useful to know in what order of precedence the kernel acts on them.

The "seizing" mechanism has the highest precedence. If **D_Sieze** is not zero, the other mechanisms are inoperative. Otherwise, the priority of a process is first checked against the **D_MinPty** field, and only if it is not below this threshold, or the process is in system state, is the priority also checked against the **D_MaxAge** field. Therefore, if a process's priority is below **D_MinPty** the process will be suspended, even if its priority is equal to or greater than **D_MaxAge**, unless it is in system state (presumably executing a system call).

9.7 SCHEDULING IN REAL TIME APPLICATIONS

The processes in a typical real time application will be divided into two groups:

a) High priority processes that are reacting to I/O events. These processes sleep, waiting for an I/O event, wake up to deal with the I/O event, and then go to sleep again. These processes are real time - they must respond to the I/O event within the specified time, or the system has failed.

b) Low priority processes that are handling non-real-time functions. User interface and reporting processes usually fall into this category.

Although the pre-emption mechanisms described above are available, they are very rarely needed. In almost every case it is sufficient to give the first group of processes all the same priority, which is significantly higher than that of the second group, which also all have the same priority.

If one of the high priority processes is woken it will get processor time ahead of all the low priority processes, although it may execute after one or more other high priority processes. In addition, if a low priority process is the current process when a high priority process wakes up, the time slice of the low priority process is immediately terminated, so the high priority process immediately becomes the current process.

To make one process execute to the exclusion of all others for a short time it is only necessary to give it a significantly higher priority. For example, if the high priority group of processes has a priority of 1000, while the low priority group has a priority of 100, then a high priority process will (to a very rough approximation), get 900 time slices before any low priority process receives any processor time. As this typically equates to 18 seconds of processor time, the high priority process will have plenty of time to finish its job and go back to sleep, without worrying that it may lose processor time to a low priority process.

This mechanism is made even more flexible by the ability of a process to change its own priority, using the **F$SPrior** system call (made by the **setpr()** C library function). In addition, a process can change the priority of another process, provided the process making the **F$SPrior** system call is owned by the same user (same group number and user ID), or it is owned by a super user (group zero).

If a greater degree of control is required in very time critical applications, the "maximum age" pre-emption mechanism can be used. This retains the benefits of the automatic scheduling for the lower priority group of processes, while giving a deterministic prioritized scheduling for the upper group of processes.

Note that a task switch is not performed if the current process is executing in system state. This causes system calls to be indivisible, but it also means that task switching is suspended while a system state process is the current process. Because a system call is allowed to proceed to completion (or until it explicitly goes to sleep), a lengthy system call that does not sleep - such as a large disk transfer without DMA - can cause a significant delay before even a high priority process gets processor time. This should be taken into account when writing operating system components such as device drivers. The

device driver, knowing that it is taking a long time to complete its operation, could sleep for one tick (which causes the process to be immediately re-inserted in the active queue) at regular intervals, allowing other processes an opportunity to gain processor time.

The same caution should be applied to interrupt service routines. The execution of processes is naturally suspended while an interrupt is being serviced, because the interrupt causes the processor to change the flow of control. Therefore interrupt routines should be as short as possible, to avoid compromising the real time response of high priority processes.