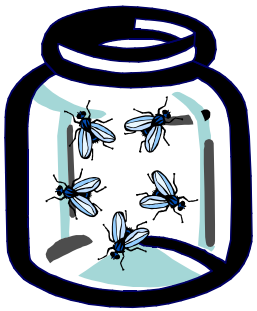
Real Time Systems

Introduction

A *thread* is a "light-weight" (cheap) process which has low start-up costs, low context switch costs and is intended to come and go easily. Threads are schedulable activities that are attached to a process. They work within the domain of the process. Threads would share the resources of a process among themselves, and take CPU time from that given to the controlling process.

Thus this technique provides a kind of shared resource multiprogramming (e.g., sharing the controlling process' memory). Clearly a thread can be created and deleted much more quickly than a process.

Flies in a bottle



A process may be likened to fly in a bottle. The bottle represents the execution environment. When a process "forks" it creates a new process, which is like another fly in its own bottle. However, a process creating a thread is like a master fly laying eggs and producing companion flies. Companions are created and deleted by the master. Still, like processes, threads in one environment can communicate with threads in another.

Real-time kernel

A **real-time kernel** is the software skeleton that provides for:

task scheduling
task initialization, and
inter task communication and synchronization.

There are four key points with real-time systems (RTS):

- Multitasking
- Foreground/Background processing
- Task Control Block (TCB) model
- Simplicity

Multitasking

Real-time multitasking can be achieved *without interrupts*, and this leads to systems that are easier to analyze.

As a consequence:

- I/O devices are usually not allowed to interrupt the CPU, but rather the I/O devices are polled to determine their status and to service a pending “interrupt”. Clearly this is a much slower way of handling I/O, but it is more controlled and uniform.
- One priority action will not be superseded by another.
- The performance of critical sections more predictable, making it possible that all deadlines are met.

Foreground/Background

Foreground/Background processing is widely used even in embedded applications.

An embedded system is one where the computing element is integral to a larger unit. The hardware and software are highly specialized to the application.

Most time-critical tasks or processes are kept in the foreground, while book-keeping and service functions run at a lower priority in the background.

QNX and Harmony are Canadian examples of Unix-based real time operating systems.

Task Control Block model

The *Task Control Block (TCB)* model is used in commercial real-time executives, and in full-featured operating systems where the number of tasks is dynamic or indeterminate.

In normal operating systems courses the terms task and process are more or less interchangeable. Thus TCB = PCB.

The term *task* is preferable in real-time systems because the word “process” is often used to describe the system being controlled, as in paint mixing process or chemical process.

Simplicity

The *more features* a real-time kernel provides, the more complex it is, the *poorer its performance*, and the *more difficult* it is to *analyze*.

Usually the bigger and more complex systems provide a better user interface and other facilities. Extra features just increase the cost by requiring more machine than necessary to do the job.

In embedded RTS low cost is important. But this factor is problematic, since a general-purpose computer may be cheaper than a specialized one. Also a simple machine with few capabilities may offer less debugging support.

Hierarchy of kernel types

There are five variants of the definition of “kernel”. Increasing complexity leads to more code and slower response time.

- *Nano-kernel*: Simple flow-of-control (thread of execution.) Provides only task initiation.
- *Micro-kernel*: Adds task scheduling, hence multi-programming.
- *Kernel*: Provides inter task synchronization and communication, with semaphores, mailboxes or communication ports.
- *Executive*: Adds memory management and protection, I/O services and other high-level features.
- *Operating System*: User interface, full resource sharing.

Scheduling

In real time systems even a simple *Round-Robin* scheduler can be good enough, and it is analyzable. Like polling it is fair and guarantees service to everyone.

It is not so good at distinguishing important processes and giving them special service when necessary. More useful is a *preemptive priority (PP)* scheduler. This has the advantage that a CPU intensive task can be preempted to provide some service to a more important task. They work well in systems where the priorities are set at task initiation time, and do not change later.

Unfortunately PP scheduling can be inflexible and may not handle the growing urgency of deadlines.

Scheduling continued

It is easy to provide a more *dynamic* preemptive priority scheduler (improvement on Shortest Job First) if you are prepared to set the priority inversely proportional to the CPU time (or elapsed time) to completion.

Unfortunately this implies that you know exactly how much CPU time is required when the task is started, or that you can always provide a deadline for the completion of every task (even the most unimportant ones).

Further, tasks may languish for long periods early in their work, only receiving crisis-type attention towards the end. This is a very poor way to manage and use resources.

Rate monotonic scheduling (RMS)

With RMS, priority is proportional to the frequency with which a task is requesting the CPU.

Good for I/O oriented tasks, and also for CPU intensive tasks that give up the CPU at their time-slice end.

But totally unimportant tasks can be either I/O oriented (taking characters from a keyboard, say) or CPU intensive (a background process computing *pi* or *e*). Hence a task may have high priority despite the uncritical nature of its work.

RMS is much loved, since the priority can change automatically as the task moves though various phases of execution.

This scheduler is well-accepted in the RT community.

Rate monotonic systems

In rate monotonic systems (systems with cyclic tasks), it can be shown that no deadline will be missed if the CPU utilization is less than 70% (the number is actually *ln(2)*, which is derived by constructing and then analyzing an event tree.)

Even if the CPU utilization is above 70%, a schedule may still be feasible (no missed deadlines), but in general there are no guarantees.

The theory itself does not take into account practical issues such as: *context switch time* or *resource contention delays.*

Most solutions that allow for these delays are impossible to analyze.

Priority inversion

There are several priority inversion problems, where an unimportant process with a high frequency of execution is given a higher RMS priority than a task that is more critical, but has a lower execution rate.

One solution is to place tasks in priority bands, and not allow the priority to rise beyond that of a more critical band. This may require some external intervention, or decision at task initiation time.

Another problem occurs when a lower priority task locks a resource that is needed by a higher priority action. Thus the critical task is blocked and waits while the low priority task with the resource struggles to get the CPU and use the resource.

Priority ceiling protocol

This solution requires that a task blocking a higher priority one inherit the priority of the more critical task until the block can be removed. The idea is excellent and deals directly with the problem of low priority tasks blocking high priority ones.

Another approach might be to preempt the resource from the other task, and force it to return to some earlier checkpoint.

Both are complex strategies, and are not without their difficulties. In particular, when waiting for a resource, many systems would not know which process currently holds it. Modifying the priorities might not be too easy either, though this can probably be done through another entry in the TCB.

Priority scheduling strategies

In all cases, assume that tasks are periodic once they are released and they arrive together at regular intervals.

- Prioritized inversely to their duration or frequency (Rate Monotonic); that is, inversely to $F_i = D_i - R_i$
- Prioritize according to their first deadline time, D_i.
- Prioritize proportional to their load A_i / F_i.

The relevant data is shown in the following table:

					Priority		
Task	R_i	D_i	A_i	F_i	**RMS**	$1/D_i$	A_i/F_i
T_1	0	10	4	10	5	2	5
T_2	3	5	1	2	1	1	4
T_3	6	12	3	6	4	4	3
T_4	7	10	2	3	2	3	2
T_5	10	15	4	5	3	5	1

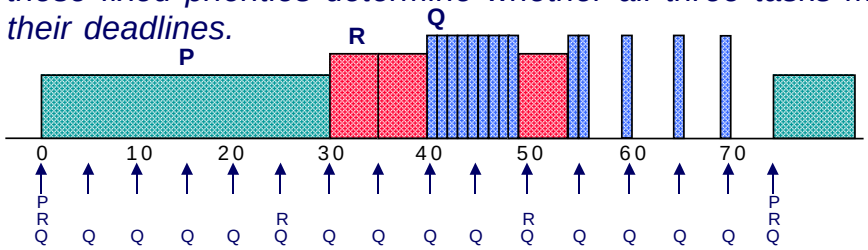
R_i = release time
D_i = duration
A_i = activity
F_i = frequency

All T_i arrive together, but have different “frequency” after release.

Exercise A

Consider three tasks, P, Q and R. P has a frequency of 75 milliseconds in which it requires 30 milliseconds of processing. These values are given as a frequency–processing pair: (75,30). The corresponding values of Q and R are (5,1) and (25,5), respectively.

Consider the case when P is the most important task, followed by R and then Q. *Using a preemptive scheduler and these fixed priorities determine whether all three tasks meet their deadlines.*



Exercise A continued

What is the processor utilization of P, Q and R?

P = 30/75 = 0.4, Q = 1/5 = 0.2, R = 5/25 = 0.2

For these same three processes, *compute the rate monotonic schedulability condition*, and *sketch the resultant timing chart*. *Show in both theory and practice whether all processes meet their deadlines*.

In the general case of M processes we must confirm, for first *k* processes in [1,M], that the following relationship holds:

$$\sum_{i=1}^{k} (A_i/F_i) < U(k) = k \cdot (2^{1/k}-1)$$

For M=1, 0.4 < 1; for M=2, 0.6 < 0.82; but for M=3,0.8 > 3*($2^{1/3}$-1). Therefore, an RMS schedule is not guaranteed, but one does exist.

Exercise B

A feature of many real-time systems is that they run a set of cyclic tasks. In the table below there are three tasks. Task T_i is released R_i seconds after task T_1 starts, and restarts every F_i seconds thereafter. During its execution-window, task T_i must receive A_i seconds of CPU activity before its deadline (i.e., the start time of the next instance of the same task.)

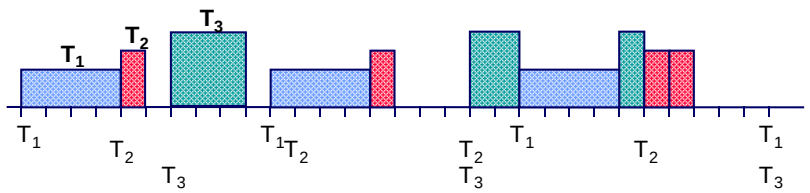
(a) *In the respective columns of the table below, fill in the priorities of these 3 tasks according to the two strategies: RMS (most frequent first) and U (largest utilization, A_i/F_i, first.)*

Task	Rel. R_i	Freq. F_i	Activ. A_i	RMS Prio.	Utilization Priority
T_1	0	10	4	2	1 = 4/10 = 0.4
T_2	4	7	1	1	3 = 1/7 = 0.143
T_3	6	12	3	3	2 = 3/12 = 0.25

Exercise B continued

(b) *Draw a sketch (a Gantt chart is appropriate), showing how a priority preemptive scheduler allocates the CPU to these three tasks according to the U priority strategy of part (a), for the first 30 seconds of execution. The sketch (diagram) should show not only when each task is released, but also when it actually receives the CPU. Do the tasks meet all their deadlines?*

No, the third instantiation of T_2 misses its deadline. However the system is RMS schedulable on theoretical grounds, so all deadlines can be met by the use of RMS priority.



Allow for blocking

Now assume that tasks T_1 and T_3 both access the same shared variable, and thus that T_3 (with lower priority) can block the more important task T_1 for 0.1 time units. For Rate Monotonic Schedulability the condition:

$$\sum_{i=1}^{k} (A_i/F_i) + B_i < U(k) = k \cdot (2^{1/k}-1)$$ *B_i = time T_i may be blocked some T_j, $j > i$*

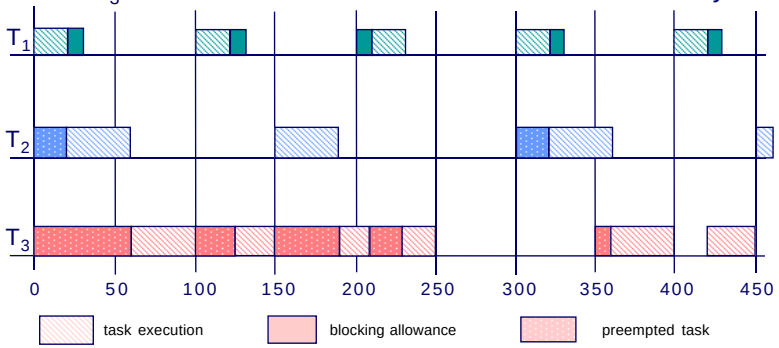
must hold.

Task	Period F_i (ms)	Active A_i (ms)	Util. A_i/F_i	Priority
T_1	100	20	0.2	1 (high)
T_2	150	40	0.267	2
T_3	350	100	0.286	3 (low)

How do we represent blocking in our process-schedule diagram?

Allow for blocking continued .

Task	A_i	A_i/F_i	B_i	Util.		U(k)	Doable?
T_1	20	0.2	0.1	0.3	<	1.0	yes
T_2	40	0.267	0	0.467	<	0.828	yes
T_3	100	0.286	0	0.752	<	0.779	yes



Note if the T_3 period had been 300, then lock step or convoy effect. However, in the case here T_3 gets an earlier start on its second cycle.

Deadline categories