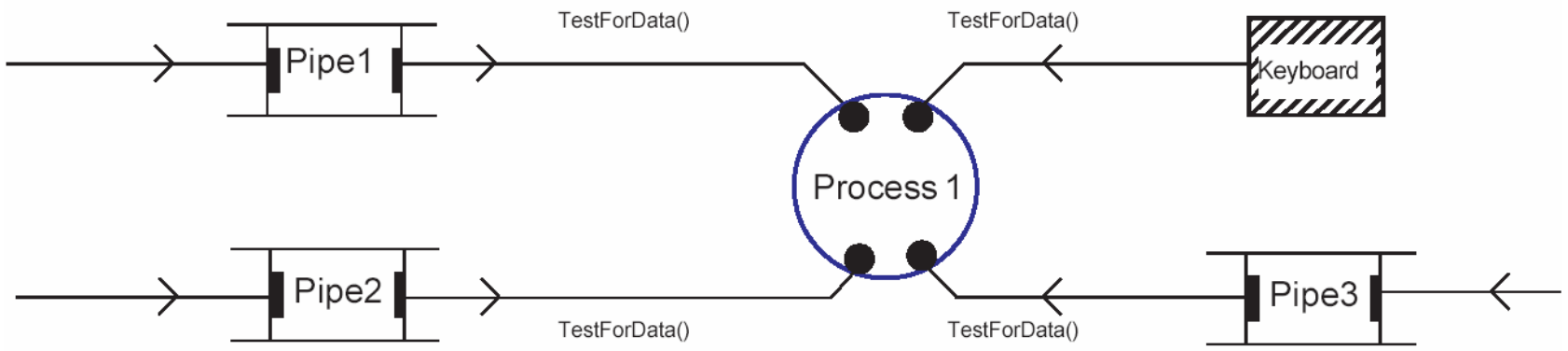
Voluntarily Relinquishing CPU time: The SLEEP() Primitive

- Another useful technique we can employ to improve the performance of our system is to deliberately design our processes so that they co-operate with other processes in the system to share out the available CPU time.
- As an example, imagine a process responsible for updating the display of graphical data on an operators terminal in some process control or manufacturing application.
- The operator gains very little by having his/her display updated 1000 times per second, since the human brain cannot process or respond to that rate of change of data.
- More importantly the effect of this continuous updating is to consume vastly more CPU processing power than is really necessary to carry out the task. The updating could be performed much less frequently with no real detrimental effect to the operator but with the benefit that other processes could receive more of the available CPU time.
- Thus, the designer could incorporate a SLEEP() primitive into their graphical display update loop so that after each iteration/update, the process voluntarily suspends itself for a given period. This forces the operating system to perform a process swap and consequently the CPU is then directed to a process than can make better use of the CPU time. As a result both responsiveness and utilisation improve.

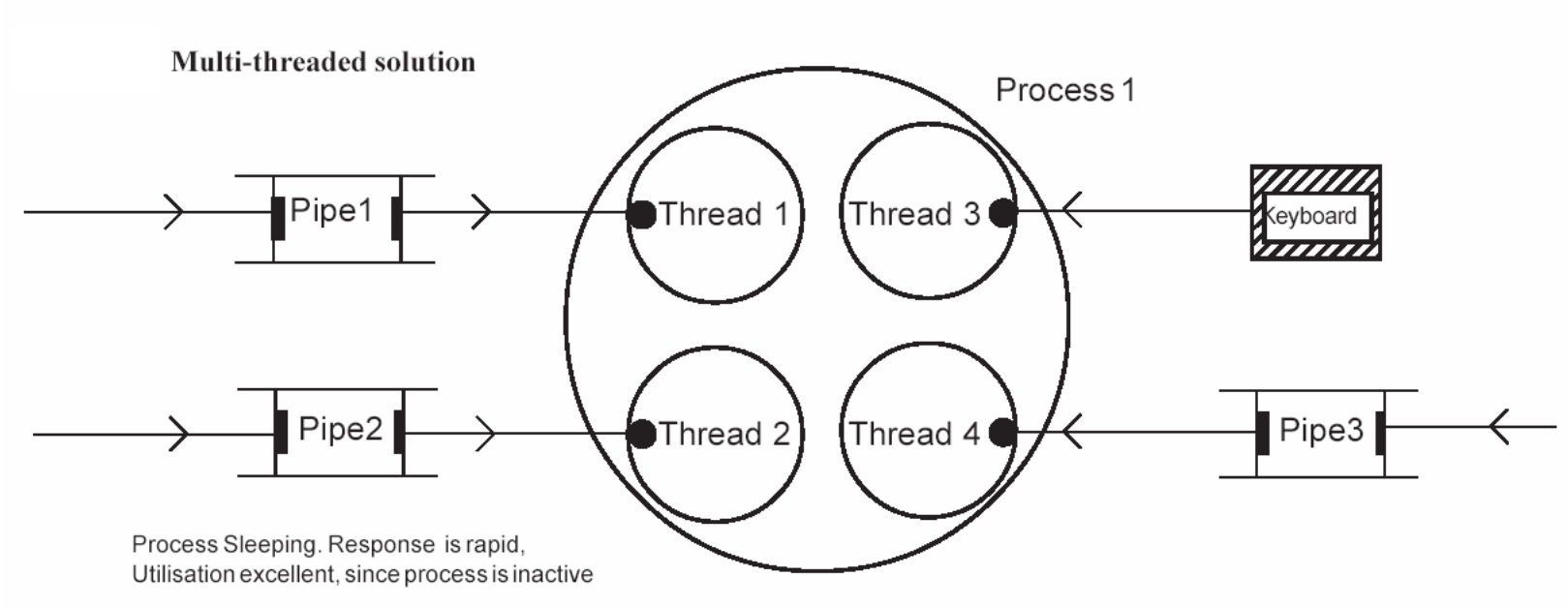
Extending the Scheme to cover Inter-Process Communication

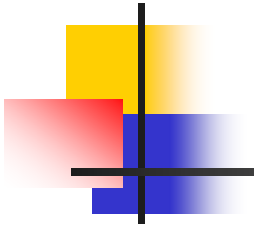
- Ideally we would like to extend this suspension of a process scheme to incorporate process-to-process communication.
- We have seen in previous discussions that a process reading from several pipelines and a keyboard, has to poll each in turn, using the TestForData() primitive to avoid getting suspended if it attempts to read from an empty pipeline (*See illustration below*).
- Now as we have seen, polling is very wasteful in terms of CPU time and utilisation. What we would like is a scheme that would allows a process to be suspended until there is data to read.
- Once solution would be to incorporate the SLEEP() primitive into the polling loop so that a process wakes up, has a quick look to see if data has arrived and if not, puts itself to sleep thereby wasting less of the CPUs time.



Extending the Scheme to cover Inter-Process Communication (cont…)

- A second, better solution would be to split the process up into multiple independent threads, each dedicated to reading from a single pipeline.
- This scheme would allow the thread to perform a read from a pipeline, and if it were found to be empty then it would get suspended. Now provided that the thread has nothing else to do, this is no big problem, in fact it's good, as it means that each thread now only consumes processor time when it has data to read. (See illustration below)
- This the creation of finer and finer process granularity using threads can improve the scheduling, responsiveness and utilisation of a system.





The Concept of Process Priority and Pre-emptive Scheduling

- Many commercial and especially real-time operating systems support the concept of 'Process Priority' which is simply a numerical value attached to a process/thread to indicate its importance in the system, relative to all other competing processes/threads. (The priority of the process/thread is generally assigned during its creation but can always be changed later if required.)

- In essence, assigning priorities means that a process with a higher priority should be able to pre-empt (i.e. *displace/dislodge/boot out*) a lower priority process when ever that higher priority process becomes active and is able to run (i.e. it is neither blocked nor sleeping).

- In other words the concept of priority guarantees that the CPU will at any point in time, always be executing the process/thread with the highest priority, until either that process terminates, or is blocked by a Wait() or Sleep() style call.

- This is particularly important when we consider Rate Monotonic Scheduling, RMS (see later) as a scheduling strategy for real-time systems.

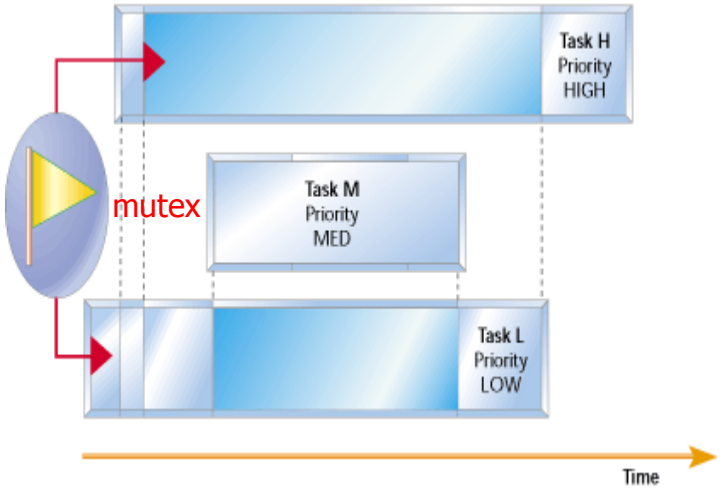
What is the Effects of Raising the Priority of an Interactive Processes?

- By *itself* raising the priority of an interactive process such as an editor achieves little, since the process is frequently blocked by the need to perform IO and thus giving it more CPU time does not necessarily mean it will get to use it. For example.
  - If the process polled its IO device then it would only waste any extra time that were given to it
  - If the process used interrupt driven IO then it would suspend itself if the device had no data and thus giving it extra CPU time again acheives little on the face of it.
- If however, the priority of an interrupt driven interactive process were raised above that of a numerically intensive process such as a C++ compiler, then it would mean that
  - The C++ compiler would still receive all the available CPU time whenever the interactive process was blocked/suspended.
  - The interactive process would receive all the available CPU time whenever it was not blocked thus it would become more responsive to its outside world whenever it was activated.
  - The interactive process would get to perform whatever actions or computations were required of it in the shortest period of time, such as search and replace operations in the case of an editor since no process swapping would be then performed between the C++ compiler and the editor whenever the editor were active.
  - As a consequence, the C++ compiler would now execute o completion more quickly with its lower priority. This is because if it had the same priority as the editor, the operating system would have to swap between then 50:50 whenever both were active. This increases the numbers of 't's (the time taken to respond to the RTC and perform a process swap). However because the editor has higher priority, the OS does not need to swap to the compiler whenever the editor is active, resulting in less 't's, thus the compiler completes slightly quicker (assuming the editor does not hang on the CPU forever which is unlikely because it is interactive)

Operating System Scheduling

Priority Inversion and Priority Inheritance

- A particularly nasty problem in RT systems associated with the introduction of priorities is that of priority inversion.
- Imagine a situation where we have a low priority process/thread currently executing on the CPU which has acquired some non-sharable resource such as a data pool by performing a WAIT() on a semaphore or mutex protecting it.
- Now imagine a second high priority process becomes activated (*displacing the low priority process*) and wishes to access that resource but cannot because it is blocked due to its WAIT() on the mutex by the lower priority process.
- The high priority process thus yields the CPU which returns to executing the low priority process so that it may complete and release the resource blocking the high priority task.
- However before that happens a medium priority task is activated and pre-empts the low priority task.
- Now even though the medium priority task does not require access to the resource, the effect of this is that the medium priority task is indirectly blocking the high priority task.
- This situation is referred to as priority inversion, i.e. a lower priority process takes precedence over a higher one, a situation which is *usually* harmless, serving only to delay the high priority process slightly, however this situation lead directly to the failure of the Mars Pathfinder mission in 1997 (see http://www.embedded.com/story/OEG20020321S0023 for details)
- The solution to this problem is Priority Inheritance. Here a low priority process which is blocking a high priority process will have its priority temporarily raised by the OS to the level of the high priority process until it has released the resource.



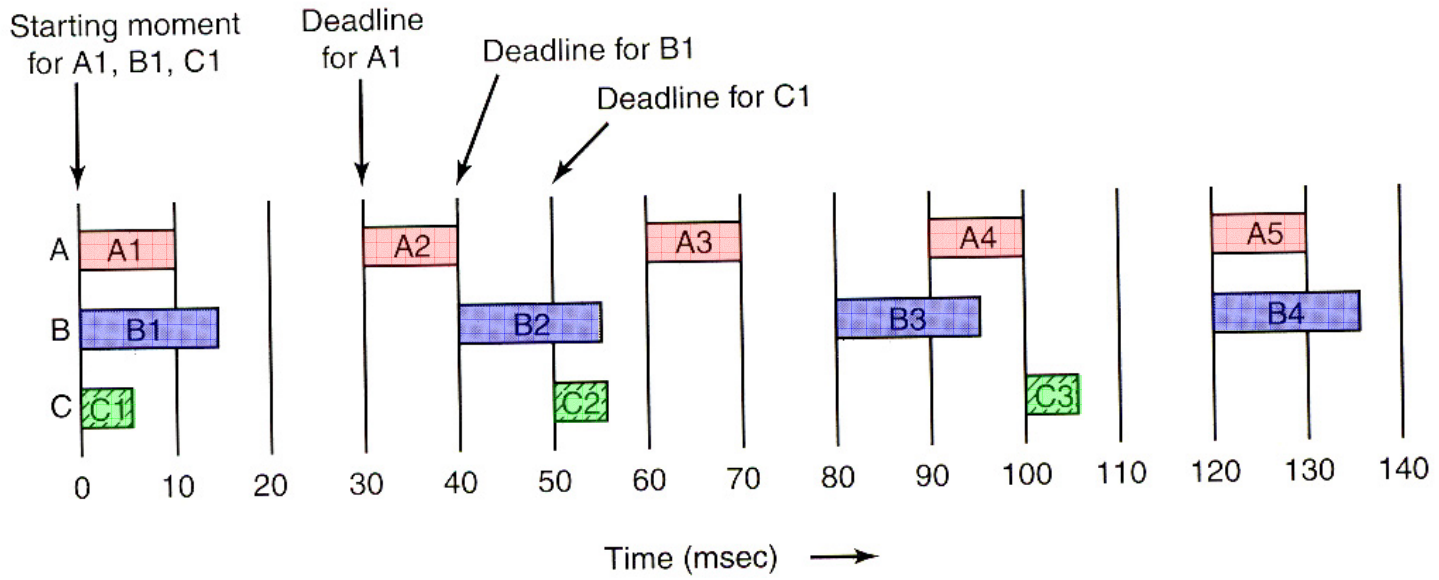
Real-Time Scheduling – Rate Monotonic Scheduling (RMS)

- In the opening lecture of this course we noted that time-based systems were an important classification of real-time systems, that is, there are a large number of real-time systems out there with processes that are designed to perform short repetitive tasks at regular time intervals, i.e. periodic processes such as a process or thread that responds to an event say every 30mS.
- The classic real-time scheduling algorithm for pre-emptable, periodic processes is RMS or Rate Monotonic Scheduling first published by Liu and Layland in 1973.
- Such a scheme can be used to guarantee the successful scheduling of processes so that they meet their periodic deadlines under the following conditions

  - Each process is periodic (i.e. it runs at a set time interval) and can complete its given task, i.e. generate its response, within that time period.
  - Each process is independent of all other process (i.e. it cannot get suspended waiting for example on a mutex that may be used by another process)
  - Each process consumes the same amount of CPU time every time it runs.
  - Any non-periodic processes have no deadlines (that is time is not important)
  - Process pre-emption occurs instantly and with no overhead, i.e. time 't' due to process swapping is 0 (not very realistic but it simplifies the modelling).

Operating System Scheduling

- The illustration below shows an example of just such a system with 3 process A, B and C being triggered by an event at 30, 40 and 50mS respectively.
- Each process will require a specified amount of CPU time i.e. a compute time in order to process the event and generate a response to it which in this example is 10, 15 and 5 mS respectively for A, B & C.
- The deadline for each process is simply the time between one occurrence of the event or trigger for that process and the next. During that time the system has to be able to generate the response.
- We see for each of these processes, that they would comfortable meet their deadline when working in isolation, that is, as the only process running in the system, since each of their compute times is less than the time between their successive events/triggers, however, things become more complicated when we let all three processes run concurrently on the system.
- Firstly can we guarantee that each would meet its deadline in the presence of the other processes.
- Secondly how do we schedule each process to ensure that each meets its deadline.



Operating System Scheduling

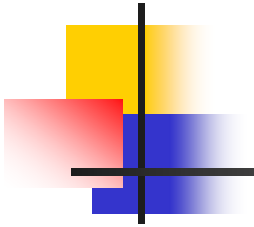
- First of all, Liu and Layland showed that any number of periodic process can be scheduled to meet their deadlines using a pre-emptable, priority based operating system if the following equation holds true

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- Where P_i is the time between successive activations of process 'i' and C_i is the computation time required by process 'i' to generate its response.
- For example, consider 3 processes A, B and C triggered by periodic events every 100, 200 and 500mS respectively.
- If each of these processes requires compute times of 50, 30 and 100mS respectively to generate their response then we see from the above equation that such a system is theoretically schedulable since

  (50/100 + 30/200 + 100/500) = 0.85 (which is < 1)

- From this we see that the ratio C_i / P_i is the fraction of CPU time being used by process 'i' on its own. That is, process A would consume 50% of the available CPU time if it were running on its own and thus the above equation ensures we don't try to schedule a number of processes that between them would consume more than 100% of the CPU processing power.



RMS Scheduling Strategy

- Successful RMS scheduling relies on being able to assign each process a fixed priority equal to the frequency of the event that triggers the process into action.
- For example, a process that is triggered every 50mS (i.e. 20 times per second) will be assigned a priority of 20 under this scheme.
- Likewise a process that runs every 20mS (i.e. 50 times per second) is assigned a priority of 50.
- That is, the higher the frequency of activation of the process, the higher its priority, that is, each process is assigned monotonically increasing priorities based upon the rate of activation, hence the name.
- The operating system can then use pre-emptive scheduling to ensure that only the highest priority process is running at any instant.
- The success of this scheme is dependant upon the OS being able to support multiple priority levels ideally in the hundreds or thousands. This is one area where Windows shows its shortcomings as a Real Time Operating Systems (RTOS), as it only has 32 priority levels and not all of them are available.

Operating System Scheduling

Example in Action

- The figure below shows three process A, B and C having static priorities of 33, 25 and 20 respectively, which means that whenever process A wants to run, it can pre-empt either process B or C. Likewise process B can pre-empt C but not A.
- Assume A, B and C have compute times of 10, 15 and 5 mS respectively as shown below.
- Initially, all processes are ready to run, but A with the highest priority is allowed to run first.
- After 10mS, B is allowed to run followed by C which between them consume 30mS. All meet their deadlines before A is triggered again at t=30mS
- This rotation carries on until the system becomes idle at t=70mS.
- At t=80mS B becomes ready and runs, however at t=90mS, A becomes ready and pre-empts B until time t=100mS. At this point the OS can either complete B or start C, so it chooses the former based on priority.