UNIVERSITY OF EDINBURGH

FACULTY OF SCIENCE AND ENGINEERING

COMPUTER SCIENCE THREE

OPERATING SYSTEMS

Thursday 6 June 2002

14:00 to 15:30

Examiners:

Prof. K. Turner (External)

Prof. C. Stirling (Chairman)

INSTRUCTIONS TO CANDIDATES

Answer any TWO questions.

All questions carry equal weight.

1. (a) Without any synchronisation, if one process is updating the value of a shared variable, another process may or may not get the new value, depending on the order of execution of the processes. In general, the order of execution of processes is unpredictable due to the occurrence of asynchronous events such as interrupts to the kernel which may result in processes being rescheduled. Which value a process gets when it reads the variable will be entirely unpredictable.

   Each process may have a region of program code which is critical in the sense that only one of the processes may be executing in their region at any one time. Any accessing and updating of shared variables can therefore be done inside such a critical region. The difficulty is achieving the desired mutual exclsuion.

   (b) In theory, if interrupts were disabled by a process before entering a critical section and enabled after leaving it, the kernel of the OS would not be entered during execution of a process's critical section and therefore would not have the opportunity to reschedule another process before the critical section is complete. That may be the theory but the practice is different! Interrupts are usually too critical to be turned off for very long - even by the kernel. Time-critical events may be lost. How long they might be off for the execution of a critical section is potentially unknown. Furthermore, interrupts such as page faults will still be required to trigger a demand page for a page of the processes critical section is not present in memory. The idea of giving a user-level process such a level of control is anathema to OS designers - its effects are far too devastating, for example if the process failed to complete its critical section for some reason, or failed completely, while interrupts were off, the whole system would crash.

   (c) Each process picks up a 'token', with a value nominally one greater than all existing tokens. The process with the lowest token value gets to enter its critical section next. Unfortunately, two processes can get the the same value token when they happen to execute the 'choosing' code simultaneously. In this case, the process with the lower process number wins. Each process decides for itself whether it is its own turn to enter its critical section. To do this it has to look at all the other processes (in any order) to check that none of them have priority. For each other process, it must first wait until it has finished choosing a token (in case the one it will get comes before this one's token); then it must wait, if the other process had priority, until that process has left its critical section.

   (d) This has aspects of bookwork in respect of the conditions necessary for deadlock to occur but hopefully thereafter the question will allow them to demonstrate their understanding of the importance of having some strategy for dealing with deadlocks. In some circumstances, this may just be to ignore the possibility but the consequences of so-doing must be spelled out.

Avoiding and preventing deadlock almost invariably involve knowing what is likely to happen in the future. For that reason and because processes and hardware components are always coming and going, trying to prevent deadlocks is almost always not a practical approach. Detecting deadlocks, or least deciding when and how often to try detecting deadlocks, can be difficult and is certainly time-consuming. Most existing operating systems do their best to avoid the possibility of deadlocks arising as far as they can by careful design but do not preclude their happening. Deadlocks are a real concern to operating system designers.

2. (a) A microkernel based OS is one where the kernel is as small as possible, essentially just fielding interrupts, dispatching and dealing with virtual memory structures. It passes control to system processes as soon as possible to deal with all other organisational matters such as time and space scheduling. The kernel can be set uninterruptable for short periods, for instance when initially handling interrupts, but system processes will be run interruptably (though with software interlocks where necessary). In a monolithic structure all parts of the OS are linked together into one program (possibly with dynamic linking and unlinking of kernel modules as and when required e.g. Linux RPMs). Communication between parts of a microkernel OS is by inter-process communcation e.g. by message passing, whereas in the monolithic type it is just by procedure calling. A microkernel's system processes are likely to be much more independent of each other and will result in a 'cleaner' interface between the parts of the system. Monolithic kernels may well be faster but much less flexible and more error prone.

   (b) A process is a program running in its own virtual machine with one context of execution i.e. program counter, stack etc. A thread is also a running program having its own program counter and local stack etc. but multiple threads may co-exist within the same virtual machine. Threads share all the resources allocated to that process - virtual space, open files etc. and cooperate among themselves. Processes compete for resources. Threads can be more efficient since they have a smaller context are therefore more quickly created. They do not need to save a complete process context when they are interrupted. Caches, TLBs etc. are also potentially more efficient because several threads exist in the same space.

   Threads are very useful in programming terms. Asynchronous activities can each be assigned a separate thread which can possibly run without interaction and synchronisation with other activities. A server process is an example, providing some service such as file access, asynchronously to other processes. In the applications domain, a typical example might be an interactive program one thread of which performs the interaction, one performs the computation requested and a third updates the screen, all notionally concurrently. If more than one processor is available, then separate threads can run actually concurrently on separate processors.

(c) A question which will allow them to demonstrate their knowledge and understanding of CPU scheduling. They will wish to take into account the likely mix of processes and threads, response time for interactive users, predictability, throughput on servers, CPU utilisation, fairness, likelyhood of starvation, system overheads, memory size, disk speed and capacity etc. Any system that results will probably be a combination of various schemes. It will certainly involve aspects of process priority whether fixed or dynamic, and probably round robin time-slicing for the interactive context. Pre-emption and priority inversion may also be discussed. Most such schedulers e.g. Windows 2000, benefit from manual tuning to optimise for different contexts e.g. changing the length of time-slice, although an adaptable system should be entirely possible.

3. The kernel designer wants to have as efficient an interrupt and context switching system as possible to avoid overheads. If the context is likely to be substantial, then hardware assist in storing and restoring will probably be useful. Making the context as small as possible would be even better. A major loss of performance on process context switching can arise from loss of TLB and cache contexts. Some way of alleviating this would be very important in a new architecture. For instance, adding a process ID to virtual addresses and then caching on virtual rather than physical addresses may help. Then multi-processing can get more difficult. Other issues include dealing with multiple interrupts, how to organise threads efficiently, handling synchronisation issues etc.

   From a user's, or at least application writer's, point of view, segmentation is likely to be important allowing a logical partitioning and protection of the virtual space, both program and data. He will also wish for some form of paging to optimise his use of physical memory. Segment and page sizes may come under scrutiny. The kernel is fundamentally affected by the VM organisation in many aspects, memory management in particular. Kernel table organisation and size will follow the VM structure. Parameterisable VM systems are common (having developed with backwards compatibility in mind) but make the kernel more complex.