Solutions – EEE6032 – January 2009

Question 1

Why is it necessary for a multi-programmed operating system to provide facilities for inter-process communication (IPC)? [4 marks]

In a multi-programmed operating system, it is essential for user-level processes to be protected from corruption by other processes. Since the operating system is unable to distinguish between intended communication between processes and unintended corruption, separate mechanisms for IPC have to be provided.

How can a simple file model be generalised to pass messages between processes in a pipe model? What is the fundamental limitation of IPC by pipes? [6 marks]

The most naïve model of communication between two processes is for one process to write the information it wishes to transmit to a disk file, and for the second receiving process to read the information from the file. This, however, is a slow mechanism as it involves two sets of disk reads and writes. Since CPUs no longer directly control the hard disk but rather, disk reads/writes are routed to the disk via a memory buffer, it is possible to eliminate the (mechanical) disk and just write from one buffer to the other. Taking this one stage further, it is possible to eliminate one of the buffers so that two processes can communicate by writing/reading data from an area of memory, access to which is controlled by the operating system. In the same why that disk I/O is provided by a set of API functions, the operating system can provide a set of API functions for writing/reading to/from the shared memory buffer.

IPC via pipes has the fundamental limitation that both processes have to co-exist.

Many operating systems provide memory-mapped files as a possible IPC mechanism (as well as other IPC methods). What are the advantages and disadvantages of memory-mapped files for IPC? [5 marks]

Whereas pipes are conduits for data, memory-mapped files (MMFs) resemble normal disk-based files except that since the data are stored in memory rather then written to/ read from a mechanical disk, access is much faster. Since an MMF is organised as a serial entity, this may or may not be an advantage for a particular application. One advantage of MMFs is that, unlike pipes, the two processes do not have to co-exist. A disadvantage of MMFs is that because the data are held in RAM, the data will be lost on power down of the computer.

For transferring data between processes, which mechanism would you expect to be faster: pipes or memory-mapped files? Justify your answer. [1 mark]

Since both mechanisms involve writing to a block of memory via API calls, the speed is the same.

In addition to pipes and memory-mapped files, the UNIX operating system provides *signals* as a low-level IPC mechanism. Explain how two user processes can use signals to communicate. What are the limitations of using signals for IPC? [4 marks]

Two processes can communicate with each other using signals providing the sending process knows the unique process ID (PID) of the receiving process. The PID is an argument for the relevant API function. Thus, one process can send an integer to another process; typically this would be used to invoke a user-specified signal handler function which would take some action. The limitations of signals are that only an integer can be transmitted – thus signals are a rather limited information passing mechanism. Secondly, for non-realtime signals, there is no guarantee that the signal will reach its intended recipient.

Question 2

Describe the typical multi-thread library/API function to create a thread, paying particular attention to the parameters and their meaning. [4 marks]

Typically, the function to create a thread takes as (one of) its parameters, a reference to a function embedded within the enclosing process; in the C language, this will be in the form of a pointer to the function. This function is then launched as a separate path of execution within the process.

The return value from the thread creating function will typically be some identifier for the launched thread, such as a handle (in the case of Windows) or a unique thread ID number.

Multi-threading systems are implemented at either user level of at kernel level. What does this mean? What are the advantages and disadvantages of each implementation approach? [4 marks]

For threads implemented at kernel level, the operating system itself manages and schedules user threads. For user-level threads, the OS kernel is oblivious to threads and thread management and scheduling is implemented as a library which runs as part of the user's process.

The advantage of user-level threading is that context switching between threads in the same process has a very low overhead compared to kernel-level context switching. The disadvantage of user-level threads is that if one thread in the process performs an I/O operation, the kernel will context switch the whole process since it is unaware of the multiple threads.

Does the POSIX `pthreads` package use kernel-level or user-level thread management? [2 marks]

This depends on the implementation. In Solaris and Linux, the `pthreads` functions are part of the API to the OS and hence, thread management is carried out by the kernel. More generally, UNIX systems implement `pthreads` as a user-level library. In fact, there is no reason why both user-level and kernel-level methods could not be implemented on the same OS.

In a user-level multi-threading library, is there one stack per process, or one stack per thread? Explain your answer. [2 marks]

Since a thread is a concurrent path of execution, each thread has to have its own stack.

In a kernel-level multi-threading system, there are two processes, A and B. Each process has two threads, A1 and A2, and B1 and B2, respectively. If thread A1 is currently running, what considerations need to be taken into account by the kernel's thread scheduler when deciding to switch to thread A2 or to one of the threads in process B (B1 or B2)? How can this be addressed in practice? [4 marks]

The basic trade-off in the scheduler deciding whether to switch to A2 or a thread in process B. The advantage of switching to A2 is that the context switching time required to switch to another thread in the same process is much smaller than for switching to a thread in another process. The problem with not switching to a thread in process B is that process B is starved of CPU time. In practice, this problem can be addressed by dynamically changing the priorities of the threads such that if the scheduler switches to A2, the priorities of B1 and B2 are incremented, thus giving them a better chance of being chosen next time.

In some older user-level multi-threading libraries, switching between threads was co-operative, even though the underlying kernel was fully pre-emptive. What does this mean and what are the programming implications of using such a library? Can you identify the major flaw in such a system? [4 marks]

Although switching between processes will be fully pre-emptive, switching between threads requires each thread to periodically yield – that is, explicitly return control to the scheduling thread within the process. The major flaw in co-operative thread scheduling is the same as with the co-operative scheduling of processes, namely that if one thread does not perform a yield or an I/O then the other threads are starved of CPU time; more subtly, the user-level scheduling thread can never run.

Question 3

Outline the principal characteristics of deadlock. [4 marks]

The four principal characteristics of deadlock are:

i) Mutual Exclusion: When one process has been allocated a resource, it has exclusive access to that resource. Namely, resources cannot be simultaneously shared between processes.

ii) Hold-&-Wait: If a process is waiting on another resource, it holds the resources which it already holds.

iii) Circularity: Processes can get themselves into a state whereby process 1 is waiting on a resource held by process 2, which is waiting on a resource held by process 3, and so on to process *N*, which is waiting on a resource held by process 1. Consequently, none of the processes can proceed.

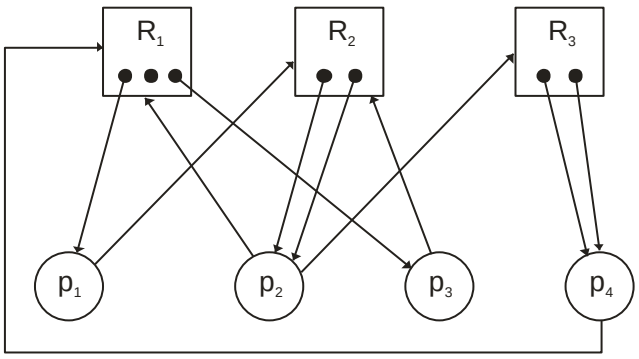
iv) No Pre-emption: A process can only release a resource by its own explicit action; a process cannot be removed from a resource.

A set of four processes, $\{p_1, p_2, p_3, p_4\}$ share three reusable resources, $\{R_1, R_2, R_3\}$. There are 3 instances of R_1, two instances of R_2 and two instances of R_3.

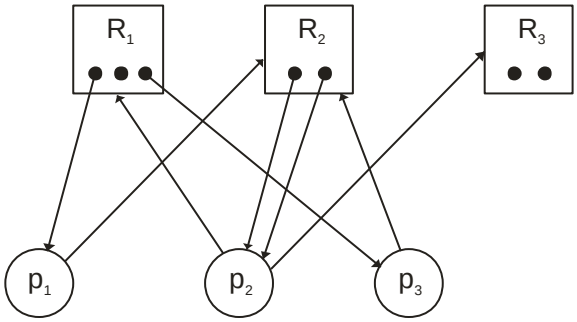
- p_1 holds one instance of R_1 and is requesting one instance of R_2
- p_2 holds two instances of R_2, and is requesting one instance of R_1 and one instance of R_3
- p_3 holds one instance of R_1 and is requesting one instance of R_2
- p_4 holds two instances of R_3 and is requesting one instance of R_1

Determine which, if any, of the four processes are deadlocked. Explain each logical deduction. [8 marks]

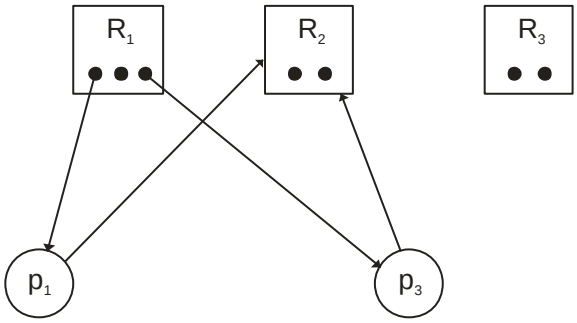
This situation can be most easily analysed using a resource allocation graph (RAG). Thus:



We can see that both p_4 and p_2 are vying for the only free instance of R_1. If it is allocated to p_2 we have deadlock because no process can make progress, but if it is allocated to p_4 then this process can run to completion, freeing both instances of R_3. The RAG can updated thus:



At this point, p_2 can be allocated an instance of R_1 and an instance of R_3 and hence run to completion, thus:



Finally, both p_1 and p_3 can obtain their requested resources and run to completion.

Thus the system is not deadlocked.

Describe the round robin scheduling algorithm with multiple level queues. What happens if the duration of the timeslice is increased to an arbitrarily large duration? [4 marks]

Within a given queue level, the operating system cycles around each process in the queue, giving each some slice of CPU time. Each level of queue contains processes of some given priority and the objective of the scheduler is to perform round robin cycling around the highest priority occupied queue. When it has emptied that queue, it proceeds to try t empty the next highest priority queue. If at any point, a higher priority queues becomes occupied, the scheduler immediately switches to trying to empty that higher priority queue.

If the duration of the timeslice is made arbitrarily large, round robin scheduling degenerates to first-come-first served (FCFS), which has no prioritisation.

Describe the hardware mechanism by which the process scheduler of a multi-programmed operating system is invoked. How can this mechanism be used to vary relative process priorities? [4 marks]

To support multi-programmed operating systems, CPUs have a built-in watchdog counter which is loaded with some number of clock ticks at the very end of a context switch. If the counter counts down to zero (*i.e.* the current process exhausts its timeslice), the counter generates an interrupt which invokes the scheduler process. The scheduler then performs a context switch to another process of its choosing.

By loading the counter with differing numbers of initial counts, different processes can be allocated timeslices of differing durations, hence varying the amount of CPU time they can use and hence their relative priorities.

Question 4

Outline how a virtual address is translated into a physical address in a basic virtual memory system. What is the principal performance penalty of this scheme compared to processes directly accessing physical memory? How can this disadvantage be reduced? [8 marks]

The processor generates a virtual address from with its contiguous virtual address space. This is translated to a page in physical memory using a page table, a look-up table which maps a logical page number to a physical page number. This page number is then used to construct a physical address which is either accessed from memory or loaded from disk if it has previously been swapped out.

The principal performance penalty concerns the look-up process since the page table will be held in main memory. Thus to access one memory location (containing code or data) will require two memory accesses: one to get access the page table and the other to access the actual data required.

This performance penalty can be reduced by caching the most recently used page table entries in fast access speed memory, usually called a *translation lookaside buffer* (TLB).

In practice, many operating systems use multiple-level page tables where a Level 1 page table contains the addresses of Level 2 page tables which contain the actual address mappings. Typically, for a 32-bit (4GB) virtual address space, the Level 1 table contains 4096 entries to Level 2 tables, each of which contain 1024 entries. What is the advantage of this multi-level page translation method? Apart from added complexity, what do you expect to be the greatest disadvantage? [4 marks]

A disadvantage of the above scheme is that a full page table has to be allocated for each process despite the fact that typically, most of the frames are unallocated and so their corresponding page table entries are unused. The advantage of the two level strategy is that only the Level 1 table has definitely to be allocated. Level 2 tables are only allocated when pages they are needed, thus saving a significant amount of memory that would otherwise be allocated to largely empty single-level page table.

The greatest disadvantage of multi-level page tables is that *three* memory accesses are now needed to access one datum (although in practice, when combined with a translation lookaside buffer which maintains a good hit rate, the penalty is quite modest).

The least recently used (LRU) algorithm is the most popular page replacement algorithm in virtual memory systems. Why is an exact implementation of the LRU algorithm impractical? Describe a simple approximation to the LRU page replacement algorithm which has acceptable performance. [4 marks]

To implement the LRU algorithm exactly would require timestamping each page with some (virtual) time of access on each access. To select a page for replacement would then require examining every page's timestamp which would be prohibitively expensive.

A more practical LRU approximation is to add an additional bit to the page table which is periodically cleared. When a page is accessed, this bit is set. When a page needs to be selected for replacement, any page with a clear bit is selected arbitrarily. This is a simple, and in many case, sufficiently good approximation to the LRU algorithm.

In connection with virtual memory, what is a ‘dirty’ (or ‘write’) bit used for? How can this be used in a page replacement algorithm to improve the efficiency of a virtual memory system? [4 marks]

Whenever a page in memory is modified, its ‘dirty’ (or ‘write’) bit – contained in each page table entry – is set. If this page is subsequently chosen for replacement, the OS knows that this page has to be written back to the page file to maintain consistency. Alternatively, if the page has not been modified, there is no need to update the page file with a saving in time.

When selecting a page for replacement, and given a choice between two pages which have not been accessed for equally long (within the resolution of the LRU approximation), it is advantageous to replace the page which has *not* been modified (*i.e.* a clear dirty bit), thus saving the time to write to the page file.