Q1

(a)(i) The exceptions commonly found are:

- Interrupts
- Traps
- Faults
- Aborts

Interrupts are typically used for handling hardware events.

Traps are used to invoke system calls called from software.

Faults are used to handle conditions which the handler may be able to correct; a page fault is a good example.

Aborts are invoked as a result of an unrecoverable error.

[3 marks]

ii) To execute a system call, a user process would issue a trap which, as well as, simultaneously transferring the processor to priveleged mode, would transfer execution to a handler function in the kernel. Typically, a wrapper

library routine would write the (index) number of the system call into a pre-defined register as well as copying user parameters into other registers before invoking the trap. The system call handler would then push the user parameters onto the kernel's stack before invoking the specific system call required. On return, an error code would be written to a pre-defined register before execution is returned to the user program and the processor put back into user mode. [4 marks]

iii) A kernel needs to be paranoid about system call parameters because invalid (or inappropriate) values might corrupt the kernel; this might be an accidental or malicious act. [3 marks]

(b) Test-&-set (@ m) returns the value of the boolean variable at address @ m and sets the value of the variable to true; these operations are atomic and so cannot be interrupted. This instruction can be used to implement synchronisation by initially setting the

lock variable (@ m) to false. Any program ~~[illegible]~~ wishing to enter its critical section performs a while-loop on the result of the Test-&-Set. If the lock is false on entry, the while-loop proceeds to the process's critical section after (atomically) setting the lock to true, thereby preventing any other process from passing its entry section. If the lock is already true, the process loops until the lock becomes false. On exit from the critical section, the lock is set to false. [5 marks]

c) When a signal is transmitted from one process to another using the kill (PID, signal) system call, the kernel alerts the receiving process by setting the appropriate bit of a signal mask contained in the process control block (PCB) of the receiving process (identified by 'PID'). When the scheduler runs a process with a pending signal it would typically restore the context of the user process, push the ~~[illegible]~~ register contents onto the user process's stack and transfer execution to the

process nominated signal handler. [5 marks]

Q2.

a) If a low-priority process is holding a resource required by a high-priority process, the low-priority process must continue executing long enough to free the resource. At this point the fully-preemptible kernel typical of real-time OSs would switch execution to the high-priority process. However, if a process of intermediate priority is present, this would take priority over the low-priority process (indirectly) thereby preventing the high-priority process from running. This is priority inversion and is undesirable because an intermediate-priority process is preventing a high-priority process from running. The solution is for a low-priority process which is holding a resource needed by a high-priority process to temporarily acquire the priority of the high priority process until it has released the resource.

[5 marks]

b) Livelocks are related to deadlocks in the sense that livelocked processes cannot make progress. Whereas neither deadlocked proces can run, two processes which are livelocked appear to be running but neither is actually making any progress. Detecting livelock is generally very difficult: usually this is only possible by deduction and observation. [5 marks]

c) At each priority level, the OS maintains a circular queue of processes. The objective is to clear the highest priority queue by cycling round the priority queue - so-called round robin scheduling. When the highest priority occupied queue has been cleared, the OS drops down to the next highest priority queue and attempts to clear that. If a process of higher priority than that of the queue currently being processed is created, the OS immediately switches its attention to the new, higher-priority proces.

A proces would typically be promoted to a higher

priority queue if it consistently context switched before exhausting its timeslice (I/O bound), or maybe because it had 'aged' (not run for a long time).

A process would typically be relegated if it consistently exhausted its timeslice (compute-bound).

[5 marks]

d) When a process blocks on a semaphore, its PID is typically added to a list of process IDs maintained by the particular semaphore instance. (The process would also be put into a 'waiting' state.) When the semaphore unblocks it would run over the list of processes it is blocking, switching each to the 'Ready' state.

[5 marks]

Q3.

a) The dilemma involved in implementing a threading system in a micro-kernel OS is where to place the threading functionality: in the micro-kernel, thus deviating from the micro-kernel concept and possibly making the

kernel (arguably) less robust; or in user-space where the additional run-time overheads would make it slower than if it were implemented in the micro-kernel. [4 marks]

b) Two independent processes can communicate via shared memory typically by one process declaring a block of memory as shared (and typically setting access attributes for other processes). A second process would then attach to the block of shared memory. (Both the above operations involve system calls.) After that, each process would be free (subject to access rights) to read from and write to the block of common memory. [4 marks]

c) Two processes could synchronise by creating some file - possibly of zero length - and then only proceeding to any synchronisation-dependent section of code if it was able to obtain the (mutually exclusive) write lock to the file. [4 marks]

d) A simple, single program option would be to

Concatenate the two sections of code, however the program would block whenever data were unavailable and the execution time would be long. An alternative would be for a single program to loop, repeatedly trying to read data from X and Y, and examining the other data source if no data were available. This alternative would only be viable if non-blocking API functions were available to read the data. Execution time could be reduced.

Splitting the handling of X and Y across two processes would allow the OS to interleave the read operations between the two data sources, switching to, say, Y while the I/O request on X was being processed. It is likely this solution would yield the shortest time to completion.

[8 marks]

Q4

a) The advantages of the buddy system are twofold. Firstly, allocation of a memory block is fast (compared to searching tables of free memory blocks).

Secondly, when a block of memory is freed and its buddy is also free, the two can be merged to create a larger contiguous block of memory. If that larger block's buddy is also free the merging process can be continued. The buddy system can thus produce a less fragmented memory layout.

[8 marks]

b) One of the emerging requirements of a computing system is the ability to inter-operate with other systems. A virtual file system layer would present generic file commands to the user but these commands would be translated into filesystem-specific commands by the virtual layer. It is thus possible, for example, to seamlessly access files on a Windows machine from a UNIX

system. [4 marks]

c) Many file operations comprise large numbers of low-level disk operations. There is thus a danger that if the system crashes in the middle of this sequence of low-level operations, the file system may be left in an inconsistent, or in some cases, unusable state.

A 'journalling file system would split the sequence of operations into three phases. Typically:

i) The intended operations are logged to a journal file

ii) The operations are commited to the actual files

iii) The just-completed operations are (optionally) journalled.

If the system crashes in phase (i) the file operations are lost but the integrity of the filesystem is not compromised.

If the system crashes in phase (ii) or (iii) the inconsistency between the two journals means the operations need to be repeated during

some subsequent recovery phase. Filesystem integrity is thus maintained.

The disadvantage of a journalling system is the time penalty incurred by doubling or tripling the number of disk accesses. This penalty can be reduced by journalling only the so-called metadata, the information relating to the file layout rather than the actual data itself. Although faster, it reduces the ability of the filesystem to recover from a crash without data loss but the integrity of the filesystem will still be maintained.

[8 marks]