COLLEGE of INFORMATICS and ELECTRONICS

Department of Computer Science
and Information Systems
End-of-Semester Assessment Paper

Academic Year:	**2004/05**	Semester:	**Semester 1**
Module Title:	**Operating Systems Overview**	Module Code:	**CS4145**
Duration of Exam:	**2 Hours**	Percent of Total Marks:	**100**
Lecturer:	**J Sturdy**	Paper marked out of :	**100**

Instructions to Candidates

- Answer any 5 questions.
- Each question is worth 20 marks.

1. Explain the concept of "process address space", and the differences between processes and threads. When would you use processes, and when would you use threads, in designing an application?

   The address space is the range of addresses (or memory) that the process can access. Processes have separate address spaces, but the threads within a process share the address space of the process. Processes can be reached from other processes, e.g. by signals, but threads are visible only within their own process.

   What are the conceptual differences between paging and segmentation? You should mention any ways that they have an effect on writing programs to run on systems using them. Does a segmented architecture have any particular implications for a multi-tasking environment?

   Addresses in a segmented system have two parts, a segment and the offset within a segment, and this is typically visible when programming using it (although a simple use might put everything in a single data segment). Paging can be used to implement either flat or segmented address spaces, and is not visible to the application programmer. Segments may be shared between processes, and some kind of co-ordination may be needed for this.

   Explain what "swapping" is, and when it occurs, both in the loose sense of the word, and in an older sense that is now little-used. Why is the old type of swapping less useful than it was in more primitive systems?

   Swapping can mean transferring pages in from disk when they are needed but are not in memory, and out to disk when not used for a while and the space is needed to make way to bring some in. It can also mean forcing the entire memory of a process out to disk to make room for other processes. Efficient page-based swapping has reduced the need for the relatively lumbering process-based swapping.

   The following answer is a particularly good real student answer, which got full marks:

   The process address space is the memory space allocated to a particular process. Thanks to virtual memory, the process has the illusion that its address space begins at 0$_{\surd}$and goes up to the end of memory. This is a good security point, as each process is only aware of its own address space and cannot access to any address space allocated to another process, since it cannot even see it$_{\surd}$.

   Threads are a subdivision of one process$_{\surd}$, they share the same address space (hence they share their data)$_{\surd}$, they have less overhead than processes which means context switching is faster between threads$_{\surd}$. Processes are separate entities with different address spaces$_{\surd}$(even if they can share some of it via paging$_{\surd}$). They are represented by different entries in the process table, so context switching takes more time than with threads.

Threads may be a good idea when writing a program if there is a need for fast context switching for a light part of the application – to keep a user interface up-to-date, for instance. Threads should be used if most of the data has to be shared$_{\surd}$, since threads share the same address space. Processes are heavier data structures but should be used if not a lot of data needs to be shared, if the program is clearly divided into parts$_{\surd}$: one dedicated to the user, and another related to a device, for instance.

- *Paging occupies a flat and linear space$_{\surd}$, it is exactly the size of a memory frame and is invisible to the programmer$_{\surd}$.*
- *Segmentation allows more flexibility in memory allocation$\times$, avoids fragmentation of the memory space, and enables address space expansion. It is visible to the programmer$_{\surd}$.*
- *Both can be combined$_{\surd}$.*

They make better use of the memory space$_{\surd}$, by avoiding gaps and enabling memory sharing (if two processes have the same page or segment in their address space)$_{\surd}$.
Swapping happens when the memory is full$_{\surd}$, it writes the pages out on disk to make space. It is also called "backing store"$_{\surd}$. When a process is needed but is not the memory anymore, it is called from the swap space and put back into memory$_{\surd}$.
[Clear, well structured answer$_{\surd}$.]

2. What interrupt-driven operating system action links memory and concurrency? Under what circumstances does it occur? What are the different kinds of this event, and how does the operating system handle each of them, and what data structures does it use and modify in doing so? Your answer should include explanations of the flag bits in the data structures concerned.

   A page fault is an interrupt generated by the Memory Management Unit, that indicates that an attempted memory access can not yet be completed. Page faults occur when:

   - *a process tries to access memory that it does not have; this is the default if the flag bits do not match one of the other cases described below – it is passed on in the form of a signal*
   - *a process accesses memory that is currently paged out onto disk; this is marked by the Presence flag bit being clear – the page is brought in, and the access retried*
   - *a process tries to write to memory that is mapped as read-only (according to the Writable flag bit being clear) – this could be a fault to be signalled, or could be a Copy-on-Write in progress, which will be indicated in another part of the process memory description – this is handled by taking a real copy of that page, and setting the page to writable in the processes that were sharing it*

   When a page fault happens, the process may have to be moved from the ready queue to a device queue, if paging from a device is required. The page fault will modify the page table for the process that caused it.

   Which system calls (functions within the operating system that can be called from ordinary programs) are made more efficient by things done using this mechanism, and how? Explain the actions done by these system calls, both with and without this assistance.

   The `fork()` operation is made more efficient by Copy-on-Write, as it does not have to do a bulk copy of memory, does not have to make room for a whole second copy at once, and does not copy data that is not written again.
   The `exec()` operation is made more efficient by the paging (and Presence bit) mechanism, as it does not have to load the whole executable file into memory at once, and does not have to make room for the whole file at once, and never loads parts of the program that are not used in that run, and does not (on the whole) load pieces of program that will be paged out again before being used.

`fork()` has to create a process header, and the associated data structures, and add the new process to the ready queue. Without the assistance of copy-on-write, it also has to copy the entire memory of the process into newly-allocated memory. With copy-on-write, it sets up the page table for the new process to point to the same frames as the parent process, and marks the area that needs copying as read-only in both.

`exec()`, without demand-loading, has to load the new program into the process memory. With demand-loading, it adjusts the page table for the process, to point the text segment to the executable file, and mark it as currently paged out.

The following answer is a particularly good real student answer, which got nearly full marks (19/20):

The action that links memory and concurrency is an interrupt called page fault$_{\surd}$. It occurs when the memory management unit is trying to access memory that it does not have$_{\surd}$, or when it is trying to read a page whose frame is not in physical memory$_{\surd}$or when it is trying to write a page which is for read only access$_{\surd}$. The first case is the default one when none of the below described happens.

When the memory management unit is trying to translate a virtual page to its physical frame and sees that there is no mapping for this page (because the Present$_{\surd}$bit is clear) it causes an interrupt to the CPU which copies the required page from disk into main memory, and adds that entry to the translation lookaside buffer$_{\surd}$for future accesses to that page and gives control back to the process that was trying to read the page that can now resume its work and try the operation again$_{\surd}$.

In the second case when the memory management unit realises that there is an attempt to write in a page that is for read-only access (the Writable bit is clear) it also causes a page fault interrupt which checks if there is a copy-on-write$_{\surd}$in action, in which case copies the physical frame$_{\surd}$that was shared among the two processes and makes it appear in both translation lookup tables as writable setting the writable bit$_{\surd}$. Once again, control is given back to the process that caused the page fault interrupt to happen that can now write on that frame$_{\surd}$.

Fork is made more efficient by the page fault interrupt as mentioned above by implementing it doing the copy-on-write technique$_{\surd}$so that only the pages that are really going to be different are the ones that is going to copy while the others can be shared among the parent and the child process$_{\surd}$. Without this it would have to copy the whole process address space at a time although many pages would be automatically overwritten by an exec call$_{\surd}$. Besides this makes that it is more time to start up a process.

The other system call that is made more efficient by the page fault interrupt is the exec system call$_{\surd}$. By using the demand loaded executable mechanism which only brings pages from disk as they are needed$_{\surd}$, on demand, it saves a lot of work swapping pages into memory and then out as they are not used and there is need to make room for other pages$_{\surd}$. The demand loaded executable works attending to the Present bit. First no page is present on memory and after a few page fault interrupt the process has got its working set already in memory and so avoid the CPU to be thrashing because there is no sufficient space and has to swap too much$_{\surd}$.

[Clear structure$_{\surd}$.]

3. Explain what software is executed when any kind of interrupt occurs. You should mention its effect on data structures such as process lists and memory management information.

An interrupt handler runs immediately, which will take care of any specific requirements of the interrupting device (such as a peripheral interface, or the system clock timer, or the Memory Management Unit) and then possibly move a process from one queue to another; in particular, if the interrupt indicated an I/O operation completing, a process will move from a

device queue to the ready queue. The interrupt handler then calls the scheduler, which picks a process (from the ready queue) to run next.

Some kinds of interrupt may be propagated to user processes in the form of signals. What are signals, and what are the operations that a process can do concerning them, either as they happen, or in advance? How could you use signals in an application?

Signals are events that cause a process to do something other than the next stage of executing their programs. A process can "handle" a signal, that is, run a specified part of the process' program; it can set up a signal handler, or mark that the signal is to be ignored, or return to the default action for the signal.

You can use signals for any asynchronous event handling, that is, when a process must be diverted from what it is doing. You can use them to catch errors within the application (for example, arithmetic exceptions) and also to tidy up when the application has to be shut down from outside.

4. What are files? How are they identified on a Linux file system?

   A file is a piece of persistent data consisting of a sequence of bytes. Each file is identified within the filing system by a number, called the inode number.

   What are directories? How are they related to files?

   Directories are a special kind of file, containing mappings from names to files.

   What are file systems? What factors do you have to consider in setting up a filing system on a storage device?

   File systems contain files and directories. You should consider block size and number of inodes when instantiating a file system.

   Explain how these three concepts fit together. Your answer should include musings on the following questions:

   - Is it possible for a file to contain another file?
   - Is it possible for a directory to contain another directory?
   - Are directories a kind of file?
   - Is it possible for a file system to contain another file system?
   - Does reflecting on "Are directories a kind of file?" affect your answer to "Is it possible for a directory to contain another directory?"?

   A file cannot contain another file, although an application such as `tar` *or* `zip` *could treat a file as a set of files that can be passed around as a bundle. Directories are arranged as a hierachy, but subdirectories are not really inside their parent directories, as directories are only files containing mappings from names to files. Files (including directories) are contained within filesystems. This is a better view than regarding files and subdirectories being inside directories, particularly as there may, on a system such as Linux, be more than one name for the same file (possibly in different directories). File systems can have other file systems mounted over any directory within them, but, just as subdirectories are not inside their parent directories but inside the file system, all the file systems reside within storage devices, so it is not helpful to regard a file system as being inside the one within which its mount point is held.*

Why do operating systems such as Linux and Unix present hardware devices (such as USB interfaces, the keyboard, disk drives, etc) as part of the file system?

This allows programs to deal with varied sources and destinations of data in the same way, without having to have separate I/O sections for devices and for files. It also unifies the naming scheme for indicating where data is to come from or to go to.

5. Explain the concept of "Buffered input/output". Is it visible from the viewpoint of the author of a typical application program? Or to any other users of the computer?

   I/O buffering means storing data in memory (working memory, RAM) between creating it and writing it out to a peripheral such as a disk drive or a printer; or storing it in memory between reading it from a device and using it in a program. It happens behind the scenes; the program simply reads and writes files, devices etc. It is sometimes visible as device activity may continue for some time after a program has finished, and also in that removable media devices have to be dismounted or otherwise made ready to remove, which includes flushing any buffered data onto the device.

   How does it make the operation of a computer system more efficient?

   Buffering allows more efficient use of devices, as data can be transferred in quantities and at times suited to the device (which is typically more demanding on timing than the software and CPU). For example, it would be inefficient to write each byte of an output file to disk individually; at least a whole disk block of data can be accumulated and written in a single operation. Even better, data heading for a disk drive could be kept buffered until it is convenient for the disk head to be in the right place for that block, thus reducing unnecessary head seeking. Similar things apply to input, and the system may do some read-ahead on the assumption that a file is being read sequentially, in order not to delay the program by having to fetch the data from disk at the time it is asked for.

   Are there any circumstances in which it is better not to buffer I/O, or in which is it better for a program to take specific control of the buffering? If so, what are they, and what control is needed? Give arguments for or against (or, for a really balanced answer, both for and against) the view that if the operating system were doing its job properly, this would not be needed.

   Buffering is usually useful, especially for sequential files. For files with odd access patterns, such as databases, it may be appropriate to make special arrangements, and allow the application more control. The operating system cannot be written to handle every possible form of program behaviour at the greatest possible efficiency, as programs vary so much; on the other hand, it might be possible to provide flexible buffering arrangments that a program can tune to its requirements.
   Buffering, in the general sense, is not appropriate for some kinds of real-time operation.

6. Some operating systems are provided with scripting languages, and use of such languages is often regarded as part of "systems programming". Are the scripting languages really part of the operating system? If so, why, and if not so, why are they often treated as if they are?

   Sometimes, part of the operating system is written in a scripting language, the example par excellence being the Unix or Linux startup sequence. Certainly something is needed to launch applications, and where this something is text-based, it is a simple step to make it read commands from an easily-edited text file. Further language features may then be added for more flexibility. A well-modularized system should allow for a variety of scripting languages to be used on it; it only strictly need provide any that are themselves used as or by parts of the system.

Explain how operating systems may be configured and customized for the uses and users of specific installations (such as web servers, mail servers, and personal desktops). You should mention what kinds of things can be configured, how the configuration information is held, how it is applied, and two approaches to how it may be changed, with arguments for and against each approach, as seen both by novice or casual users, and by "industrial-strength" users such as IT professionals.

Configuration typically includes the overall file system layout, and the processes started at system boot time (or marked for starting on demand later, to service particular requests). Directories for certain kinds of file, such as printer spool files, must be specified, and for a machine which handles server-type activity, it may need to know how it fits in with other servers on the network (for example, where to pass on requests that it does not handle itself, for each kind of service). On a smaller scale, users' personal environments (command paths, and preferences such as screen background etc) may also be configured.

Configuration data is normally held in files. If it is held in simple text-files, it can be edited conveniently either with a general-purpose text editor of the users' choice, or with a special application program, typically with a GUI. If it is held in a binary format, the user is forced to use the provided application. The GUI approach is supposed to reduce the learning curve, but typically makes it harder to do bulk changes, such as changing a large number of directory names all in the same way, which would be easier with a text editor. The text-based approach is typically easier to apply to multiple machines too, as the files can be copied around, whereas when the configuration mechanism is hidden, it is not necessarily possible to copy the information from one machine to another – this is inconvenient when updating all machines on a company network, for example.

7. Although email is an application (or group of applications), it is often treated as a facility provided by the operating system.

   List, with brief notes, the main arguments for and against the operating system handling various aspects of email. *(There is no single right answer to this part of the question. It will be marked in terms of the understanding of the topic that you display in your arguments for or against the statement.)*

   On a shared machine, mail delivery must be handled centrally, in the sense of "not under the control of an individual", to make sure that each message gets delivered to the right person, and no-one can interfere with other people's mail.

   The Mail User Agent (the mailer the user uses) is best left up to the user to choose. The OS may provide one or more mail handling utilities, but it is probably best to allow the user to bring their own, as well – although a delivery component may have to limit the action of the user program, for example, to stop them sending spam (and getting the whole machine blacklisted).

   There should be an agreed interface between the user part of the mail system, and the delivery part of it.

   List and describe the programs, processes, files and protocols involved in processing email.

   - *delivery program(s)*
   - *user agent*
   - *incoming and outgoing spool files and directories*
   - *delivery control configuration file*
   - *delivery techniques database*
   - *mail transfer protocols such as SMTP and IMAP*
   - *user-level and system-level mail alias databases*

   (a couple of points about most of these should be sufficient)

8. Why is it important that only operating system code can be executed when the CPU is in supervisor mode? What is the mechanism that enforces this?

   It would be possible for user code to take malicious actions, such as copying private files, or modifying data that the user has no permission to modify, if it could run in supervisor mode. It would also be possible for mistakes made in the code to be damaging.

   A "software interrupt" instruction is used to access system calls. This causes two things to happen, indivisibly:

   - *a call is made via the interrupt vector*
   - *the CPU switches to supervisor mode*

   Explain the concepts of user-IDs, and how they apply to files and to processes. How does their use by processes determine their use by filing systems, and vice versa?

   A user-ID or uid (on a Linux system, for example) is a number identifying files and processes belonging to a user. The number is looked up from the user name the user gives when logging in. When a process tries to access a file, the uid of the process is checked against the uid of the file, and the appropriate set of permission bits is checked for whether that form of access (read, write, or execute) is allowed to that process – if the file and the user have the same uid, the "user" or "owner" permissions are used, and so on. When a process creates a file, the file will be owned by the owner of the process.

   What is a "root user-ID"? Is it necessary to have such a user, or are there alternatives? Is it desirable to have one, or not to have one, or does this depend on the circumstances? (There is no single right answer to the last part of this question. It will be marked in terms of the understanding of the topic that you display in your arguments for or against the statement.)

   "root" is the super-user, and processes owned by root bypass various security checks made by the system (such as file permissions).

   In some circumstances, this is too much power to give to users who have to have some extra powers, so it may be preferable to have several kinds of more specialized users with extra powers; for example, someone might be allowed to make backups (potentially accessing files that they could not access as a normal user) but without being able to change other people's files.

   On the other hand, it is simpler to have a single user with control of the entire system, at least when there is someone with suitable skills to do so with relatively low risk of system damage.