Operating Systems I
Supervision Exercises

Stephen Kell
Stephen.Kell@cl.cam.ac.uk

November 16, 2009

These exercises are intended to cover all the main points of understanding in the lecture course. There are roughly 1–1$\frac{1}{2}$ questions per lecture, and each question is supposed to take roughly the same amount of time to complete. Don't expect to be able to answer everything. You are advised not to spend more than an hour on any one question – unless you really want to.

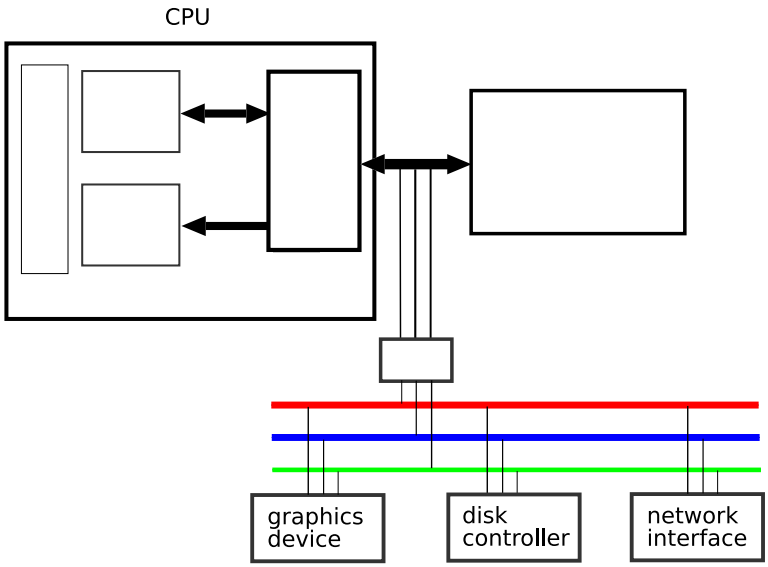
Compared to Tripos questions, these questions will generally be more structured and require smaller steps of reasoning. However, with the exception of the first few, they should be no more or less difficult conceptually. Unlike most Tripos questions, most of these can be answered quite briefly, so you should expect to spend more time reading and thinking than you spend writing answers.

Be warned that Tripos exams frequently contain "bookwork" questions, and you will be expected to remember incidental facts from the lecture notes which are not covered here.

Criticisms or comments about any aspect of these questions would be very gratefully received, however large or small.

1. *Architecture, memory hierarchy and the fetch-execute cycle*

   The following diagram shows a simple Von Neumann machine.



   (*a*) Copy the diagram above, and add the following labels in the appropriate places:

   - register file
   - control unit
   - execution unit
   - cache memory
   - processor bus
   - memory bus
   - main memory
   - I/O bus

   Draw arrows to show the paths of communication between the register file, control unit and execution unit.

   Draw a dashed box around any data paths which were not present in architectures predating the Von Neumann machine.

   (*b*) An execution unit may contain many different functional sub-units, but three in particular are always present. Give the names of these, and identify which of them might output a signal connecting to an *input* on the control unit.

   (*c*) (*i*) Explain why all modern systems have cache memory in addition to main memory.

(*ii*) A naive system designer calculates a system's total memory as the sum of sizes of its caches and the size of its main memory. Explain why this does not give a meaningful result.

(*iii*) If one memory device is said to be "below" another in a memory hierarchy, what *two* comparisons can be deduced between the two devices' storage technologies?

(*iv*) Suggest what lies *above* the cache memory in the memory hierarchy of the machine you drew in part (*a*).

(*d*) Briefly describe the sequence of events occurring as the processor undergoes a series of clock cycles. Assume that the processor is initially about to fetch an instruction. Make sure you explain the roles of memory, execution unit, control unit and program counter.

(*e*) The *Harvard* architecture is similar to the Von Neumann architecture except that it has separate caches for instructions and data. Suggest one plausible reason why this might be advantageous, and one situation where it might cause problems.

2. *Arithmetic, memory access, addressing modes and endianness*

(*a*) (*i*) In eight-bit signed two's complement binary arithmetic, write the bit-patterns for the decimal numbers 0, 1, 127, -1, -127 and -128. [The bits should be ordered from most-significant to least-significant.]

(*ii*) Explain why the range of an n-bit two's complement representation is always one greater than that of a sign/magnitude representation.

(*iii*) Explain why two's complement is used in preference to other schemes in all modern architectures.

(*b*) (*i*) Explain what is meant by "shifting", as might be done in the the arithmetic & logical unit (ALU) of a processor.

(*ii*) Explain how shifting can sometimes be used instead of multiplication and division, and why this can be advantageous.

(*iii*) Explain the distinction between logical and arithmetical shifting, with reference to signed arithmetic.

(*c*) (*i*) Apart from operands (arguments), give two examples of information which information which might be included in the representation of a machine instruction.

(*ii*) Identify three ways in which arguments to an instruction can be specified.

(*iii*) Explain the benefits and drawbacks of a fixed-length instruction encoding (such as on the ARM architecture) compared to a variable-length encoding (such as on x86).

(*d*) (*i*) Explain the term "byte order" (or "endianness") by drawing a diagram to show how how the 32-bit hexadecimal number `00C0FFEE` is represented differently in the memory of 32-bit big- and little-endian machines.

(*ii*) Similarly, draw a digram showing how the string "`Hello, world!`" is stored in memory on big- and little-endian machines. You need not convert the characters to numerical representations, and should assume a one-byte-per-character encoding.

(*iii*) Explain why there is no need for architectures offering only 8-bit arithmetic and 8-bit memory accesses to specify a byte ordering convention.

(*iv*) Suppose you wanted to write an assembly-language routine which counted the length of a *null-terminated* string by iterating over the characters until you reached the end. Explain how you would store the current location in the string, and which *addressing mode* you would use when reading the next character. Assuming you could only read and compare four bytes at a time, what arithmetic and/or logical instructions could you use to detect the string's null terminator?

[A null-terminated string is one which is represented in memory as a sequence of character values followed by a zero-valued "null" byte. You should assume a one-byte encoding as previously. An answer in pseudo-assembly code is acceptable.]

3. *I/O: buses, interrupts and direct memory access*

   (*a*) Briefly explain the term *bus*. Explain the purpose of address, data and control lines. In a *synchronous* bus, what specific control line is always found?

   (*b*) Explain the terms "master" and "slave" in relation to buses. Give an example of a device which is always a master, and one which is always a slave.

   (*c*) Explain what an *interrupt line* is, and why it is used. Suggest which functional unit of the processor it connects to. Explain what happens to the processor's flow of execution when an interrupt line is asserted.

   (*d*) Explain the key difference between direct memory access (DMA) and simple interrupt-driven I/O. Why might DMA introduce additional complexity to the processor's cache memory?

4. *Operating system concepts and structure*

   (*a*) Consider the statement that "an operating system should securely multiplex a computer's hardware resources". Give four examples of "resources" to which this statement might be referring. Explain the term "multiplex". Give an example of an *insecure* way to multiplex one of the resources you mentioned.

   (*b*) Explain the term "multiprogramming". Explain why a system which is not multiprogrammed does not need secure multiplexing of resources. Suggest what function(s) an operating system might still be used for on such a system.

   (*c*) Consider multiplexing of CPU time, I/O devices and memory. In each case, explain what hardware support is necessary to ensure *secure* multiplexing.

   (*d*) What is a *software interrupt*? Explain why these might not necessarily be found in a monolithic operating system, but are always found in kernel-based systems. [Note that, for reasons which will become clear, most monolithic OSes do, in fact, provide software interrupts.]

   (*e*) How does a *microkernel* differ from a conventional kernel? Briefly list the motivations and difficulties behind this.

5. *Processes and scheduling*

   (*a*) Briefly explain the terms terms "process" and "context switch".

   (*b*) Draw a state diagram showing the three basic states a process may be in, and the events which cause it to transition between these. Give a specific example of an event which might cause a process to transition from "blocked" to "ready".

   (*c*) Explain why *non-preemptive* scheduling is more likely to be fair for I/O-bound processes than for CPU-bound processes.

   (*d*) For each of the following scheduling algorithms, briefly describe their operation and list their advantages and disadvantages.

      (*i*) first-come-first-served (FCFS)

      (*ii*) shortest job first (SJF)

      (*iii*) round robin (RR)

      (*iv*) shortest remaining time first (SRTF)

   (*e*) Explain the concepts of *priority* and *quantum* in scheduling algorithms. Which of the above algorithms make use of these concepts?

   (*f*) Explain the difference between *static* and *dynamic* priority, and the motivation behind the latter. In what way might SJF and SRTF be considered dynamic priority algorithms?

6. *Memory management basics: the relocation problem*

   A simple assembler outputs a program executable in the form of a list of instructions (or *text*), a list of words containing global constant data and a list of words containing initialised global variable data. When the program is loaded, each of these is read into a particular area of memory.

   (*a*) Name *two* other regions of memory which will typically be allocated when the program is run, and will be accessible by the instructions of the program. Briefly state the purpose of each.

   (*b*) A *direct branch* instruction transfers control to an instruction at a known position (or *offset*) within the program text, where this offset is included as an argument to the instruction. How might the presence of this kind of instruction in a program lead to an instance of the *relocation problem*?

   (*c*) Describe a way of expressing the destination address of a direct branch, such that code which uses this instruction may be *position-independent*, *without* requiring virtual addressing.

   (*d*) DOS, a single-tasking monolithic operating system, allows some executables to directly specify the physical memory addresses of their constituents. Explain why a multiprogramming operating system cannot allow this.

   (*e*) A system architect suggests that all memory addresses in programs should be rewritten as the program is loaded into memory. Another suggests that instead, additional hardware is introduced to support *virtual addressing*. Briefly explain the advantages and disadvantages of the latter technique as opposed to the former.

   (*f*) In modern systems which use virtual addressing, it is nevertheless common to rewrite addresses at load time. Suggest a useful feature which might make this necessary, and explain why it does so.

7. *Allocation, fragmentation, paging and segmentation*

(*a*) Consider the following statements about memory allocation schemes. In each case, say whether the statement is true or false. If it is false, write one or two sentences explaining why. If it is true, explain why the converse would not work or is not possible.

(*i*) "A static multiprogramming system can only have one process in main memory at a time."

(*ii*) "A dynamic partitioning system requires a process to specify how much memory it requires in advance."

(*iii*) "Compaction can only be used on machines with support for run-time relocation."

(*iv*) "Paged virtual memory systems do not suffer from fragmentation."

(*v*) "A pure segmentation design always suffers from slower allocation than an equivalent pure paged memory system."

(*b*) Consider a process wishing to allocate a minimum k kilobytes of contiguous virtual memory. Stating any assumptions you make, give the big-O execution time cost of performing the allocation on:

(*i*) a pure page-based virtual memory system, where each page is f kilobytes and the total number of pages already allocated is p;

(*ii*) a pure segment-based virtual memory system, using a first-fit allocation policy on a linked list of s previously-allocated segments stored as (*base*, *limit*) pairs.

[You should assume that all the symbolic quantities above are variables, so it is likely that your big-O expressions will contain most of them.]

8. *Page tables, translation, sharing*

(*a*) (*i*) In a process's page table, why are many entries marked "invalid"? What happens if a process tries to access an invalid page? Briefly describe *two* techniques which can be used to reduce the amount of space wasted on invalid page table entries. [You need not go into detail.]

(*ii*) What extra information, specific to paging, must be saved as part of a process's *context* when switching to another process?

(*iii*) A system designer optimises the saving and restoring of registers, so that the cost of doing so is negligible. He claims that there is now no performance penalty in frequently switching between address spaces. Supposing that his optimisations are correct, and assuming that the system has a simple translation lookaside buffer (TLB), why is his claim still not true?

(*b*) Two processes are *sharing* a particular page at the same virtual address. This means that in both processes's address spaces, a particular paged-sized region of memory maps to the same frame of physical memory.

(*i*) What might (possibly) differ about the entries for the shared page between the two processes' page tables?

(*ii*) For two concurrently-running instances of the same program, each with code, data, heap and stack segments, which segments might benefit from *copy-on-write*? In what circumstances, if any, could these instead be shared read-only?

9. *Demand paging, page replacement*

(*a*) (*i*) What might an operating system do to ensure that it is notified the next time a process accesses a particular page?

(*ii*) In a demand-paged virtual memory system, what additional information must an operating system record for pages, other than what would ordinarily be included in a page table entry? Suggest two different places where this information could be stored.

(*iii*) The total capacity of a demand-paged virtual memory system is (approximately) the sum of the main memory size and the available disk swap space. This contrasts with caches, where the effective capacity is simply the size of main memory. Explain as fully as you can why the two cases are different. [Hint: think about how cache, main memory and disk space are *addressed*, and how they are managed.]

(*b*) FIFO, Clock and least-recently-used (LRU) are three page replacement algorithms.

(*i*) Consider a process starting up on a pure demand-paged virtual memory system. Thinking about the kinds of data (including code) that are read in to memory during the process's initial start-up phase, give an example of data for which FIFO would make a good page replacement decision, and another where it would make a poor one.

(*ii*) LRU cannot be implemented efficiently on conventional hardware. Suggest *two* alternative hardware features which could allow LRU to be implemented efficiently. Suggest why these are not implemented in practice.

(*iii*) What optimisation does the availability of a *dirty* bit in page table entries allow the operating system to make when doing page replacement? How can a dirty bit be simulated if the hardware does not provide it?

(*iv*) Explain how Clock approximates LRU using the *reference* bit. State one way in which Clock is similar to FIFO, and another in which it is fundamentally different.

(*v*) Write some pseudocode, operating on a large array of data, for which LRU would be a bad choice of algorithm. What assumption does LRU rely on but your code contradict? [Your code need not do anything useful.]

10. *Frame allocation, thrashing; segmentation, more sharing and protection*

(*a*) The *working set* of a process is defined as the minimal set of pages which it requires to be resident in order to make any progress in its task.

(*i*) With reference to the working set, explain why, if too many processes are started, the system enters the state known as *thrashing*. What is characteristic of the CPU load observed under thrashing?

(*ii*) Suggest how the working set of a process may be estimated from a snapshot of its memory space (including page tables). State what assumptions your approximation depends on for its accuracy.

(*iii*) An *allocation policy* is logic used to decide what share of physical memory a process should receive. With reference to the working set and thrashing avoidance, outline one situation where a simple "proportional shares" policy results in wasted frames.

(*b*) (*i*) Give an example of a privileged operation which can be performed more efficiently under a pure segmented architecture than under a pure paging architecture.

(*ii*) When a process performs a memory access in a paged architecture, to check that the access is valid, the hardware checks the valid bit and access permissions of the relevant page table entry (normally cached in the TLB). Outline the steps of the equivalent check in a pure segmented architecture. [Assume that segmentation is implemented in hardware.]

(*iii*) *Demand segmentation* is analogous to demand paging, but it is rarely implemented. Suggest two reasons why demand segmentation is less useful or less efficient than demand paging.

(*iv*) Suggest one optimisation which could be made to save space in a multi-level page table scheme. Likewise, suggest an optimisation to overcome external fragmentation in a segmentation scheme. [Hint: two possible optimisations both result in systems which are part-way between segmentation and paging, without going so far as to implement both techniques at the same time.]

11. *I/O: polling, interrupts and DMA*

A processor is connected to I/O devices by an I/O bus, and can communicate with them by *polling*, interrupt-driven I/O or direct memory access.

(*a*) (*i*) In what way does polling *waste* CPU cycles? Why is it nevertheless still widely supported and used?

(*ii*) When using interrupt-driven I/O, the number of CPU cycles consumed by I/O operations is determined by the speed of which device?

(*iii*) Describe one way in which interrupts can be used to aid fair process scheduling, and another in which they hinder it.

(*iv*) Briefly describe *two* ways in which DMA slows down the CPU's access to memory.

(*b*) A process wishes to transfer to main memory n bytes of data from an I/O device which can internally buffer up to b ($b < n$) bytes, over a bus which is w ($w \leq b$) bytes wide and takes c CPU cycles per bus transfer.

(*i*) Using interrupt-driven I/O, how many interrupts must be serviced to complete the transfer? How many CPU cycles are spent transferring data over the bus?

(*ii*) Using DMA, how many interrupts must be serviced to complete the transfer? How many CPU cycles are spent transferring data over the bus?

12. *Device characteristics, application-level interfaces to I/O*

(*a*) For each of the following, select the device or event which best matches the stated criterion, and very briefly explain your choice.

(*i*) Least suited to polled I/O: {network interface, mouse, modem}

(*ii*) Least likely to generate an interrupt: {key pressed, CD ejected, disk read completed}

(*iii*) Least likely to use DMA: {sound card, disk, printer}

(*iv*) Most likely to use memory-mapped I/O: {disk, 2-D display adapter, keyboard}

(*v*) Most likely to be buffered with circular buffers: {network interface, CD-ROM reader, timer}

(*vi*) Most likely to be buffered *and* cached: {network interface, display adapter, disk}

(*vii*) Most likely to be scheduled using a nontrivial algorithm: {printer, network interface, disk}

(*b*) An operating system provides system calls allowing applications to perform blocking, non-blocking and asynchronous I/O. Describe an application which might use all three of these. In each case outline one function for which the application might use that kind of I/O, and justify its suitability. [It might help to consider primarily disk-based I/O, using a familiar example such as a word processor, web browser or e-mail client.]

(*c*) Many operating systems separate devices into classes, such as block, character and network devices, and provide a slightly different programming interface for each. Give an example of an operating system service which allows access to one a device of one class by making it look more like one of another class. Explain, using an example, why this makes error handling more problematic. [Hint: think about how an operating system may provide access to *files*.]

(*d*) Identify a possible inefficiency in the interaction of buffering, caching and demand paging. Briefly outline how this might be solved.

13. *File systems: storage service implementation*

A disk consists of a large number of fixed-size blocks, numbered from 0 upwards. The disk controller supports operations to read or write exactly one block at a time. The operating system wishes to implement a file system on top of this block-level interface, so that applications can store variable-length *files* instead of individual blocks.

The storage service of this file system is to support files consisting of zero or more blocks. Each file will also have some *metadata* stored in a *file control block*. This metadata includes the file length, timestamps, access permissions, the owning user ID and information used to locate the file's data blocks. Disk blocks are not shared: each contains only one kind of data and pertains to at most one file.

(a) The storage service implementation divides blocks into three categories, based on their contents. One of these is "unused". What are the other two? [Note: in the design so far outlined, the mapping of blocks to categories is *not* stored on the disk, and there is no directory structure.]

(b) The file metadata encodes a mapping from a system file identifier (SFID) to an ordered sequence of block numbers. Suggest what piece of information could be used as the SFID for a given file.

(c) Suggest *two* simple data structures which could be used to maintain (on disk) the mapping from SFID to sequence of block numbers. Give an advantage and a disadvantage of each.

(d) Why is the file length stored in the metadata?

(e) As described so far, it is impossible to safely create a new file in the storage system. Explain why, and suggest what additional data structure could be stored on the disk to rectify the problem.

(f) A *file system integrity checker* is a program which walks a file system (typically after a power failure or other system crash) and ensures that file metadata is self-consistent. Explain why it is impossible to check the integrity of the storage system as described so far, and suggest changes which would make this possible *without* adding a directory service.

14. *File systems: the directory service, user access and other issues*

A directory service is implemented on top of the storage service. A *directory* is a new type of file, whose contents are a table mapping variable-length "human-friendly" names to SFIDs.

(a) What new piece of metadata must now be recorded in each file control block?

[Students taking the 25% option may wish to read the Wikipedia article "binary relation" before attempting the remainder of this question.]

The directory structure encodes a graph (V, E) where V is the set of files (including directories) on the file system, and E is a binary relation on V such that $(v, v') \in E$ iff v is a directory mapping some name to v'. A pathname is a list of such names.

(b) Given a particular file system and a pathname, what else is required to identify a particular file? How is this usually specified?

(c) What *two* problems occur if the set of directories is restricted to exactly one element?

(d) What problem remains if there may be any number of directories but E must encode a strict hierarchy (i.e. a tree)?

(e) What difficulty occurs with *existence control* if structure of the directory graph is not restricted? What restriction may be imposed to overcome this? Explain how *link counts* may be used to implement existence control on such a graph.

(f) Normally, user-level access to files proceeds according to an open-access-close pattern: first the user "opens" (or perhaps creates) a file, specifying pathname, and is returned a "user file identifier" (UFID). The user then performs some operations, and finally "closes" the file.

(i) Outline, as fully as you can, the steps performed by the operating system when asked to "open" a file.

(ii) Suggest *three* reasons why an open-access-close pattern is used in preference to specifying each operation independently.

15. *Unix file system case study*

(*a*) Give closest Unix equivalent for the following generic terms.

- file control block
- system file identifier
- user file identifier

(*b*) Describe the simplifications or restrictions which Unix makes to provide the following features or guarantees. Briefly mention any special cases or subtleties involved.

- efficient existence control
- unified interface to devices, files and interprocess communication streams
- simple but reasonably powerful access control

(*c*) Draw a diagram of a Unix *inode*. Explain the trade-off motivating the use of direct, indirect and multiply-indirect block references.

(*d*) What is a *mount point*? Explain the change it necessitates to pathname lookup.

(*e*) What is a symbolic link? Explain how it differs from a normal link (i.e. a directory entry), and its benefits and drawbacks.

(*f*) Draw a simple diagram showing how a disk is laid out in Unix, and what information each section contains.

(*g*) Give three ways in which Unix helps users to avoid the use of long pathnames, *not counting* facilities provided by the shell.

(*h*) Outline the procedure for file system integrity checking and recovery (following a crash). For both of the major inconsistencies which are possible, describe a sequence of operations which would have given rise to this inconsistency, had they been under way immediately before the crash.

16. *Protection basics*

(*a*) Below are some common security-oriented practices or features of operating systems. For each, identify one or more goals or properties which the feature or practice is designed to achieve, choosing from the following vocabulary: {secure authentication, audit trail, privacy, integrity, availability, compromise detection, covert channel avoidance, least privilege operation}. Briefly explain the particular instance of the property which the system is trying to achieve.

(*i*) Being required to "press Ctrl–Alt–Del to log in" (e.g. on a Windows NT box)

(*ii*) RSA encryption between terminal and server

(*iii*) Randomised delay after failed login (e.g. on Unix)

(*iv*) Digital signature of device drivers and downloaded software

(*v*) File access control lists

(*vi*) Running services (e.g. a web server) under their own special user identity

(*vii*) Limited (maximum) rate of process creation

(*viii*) Disallowing user access to DMA

(*ix*) Reserving a fixed amount of disk space for privileged users

(*b*) Give three possible techniques which improve computer security but which are *not* themselves implemented in computer hardware or software. In each case, mention which (using the above list, and adding your own if necessary) security properties they help enforce.

17. *Capabilities, and Unix access control*

A Unix system contains an executable called `web` with owner `bob`, group `spider` and mode `rwxr-x--x`. The group `spider` has members `alice`, `bob`, `charlie` and `dave`.

(*a*) (*i*) Draw an access control matrix for the file.

(*ii*) Give one benefit of using *groups* in access control as opposed to an arbitrary access control list (without groups), and one disadvantage.

(*iii*) What is a *capability*? Refer to the access control matrix in your answer.

(*b*) Bob wants his program to maintain some system-wide usage statistics, outputting to a file in his home directory. Anyone should be able to read the statistics, but only Bob should have write access to them.

(*i*) What feature of Unix access control might Bob use to provide this functionality?

(*ii*) Why is this modification likely to introduce security vulnerabilities?

(*iii*) Suggest a way that Bob can avoid the biggest potential vulnerability, by making careful use of Unix file descriptors. [Hint: consider file descriptors as capabilities.]

(*iv*) Suggest why the resulting system is still unlikely to be secure.

(*c*) Capabilities are often criticised for being difficult to *revoke*. Explain this statement, and suggest one or two simple solutions (highlighting any clear drawbacks within your suggestions).

(*d*) A system designer proposes using virtual addresses as capabilities. Suggest why this is not secure on a conventional instruction set architecture, and suggest one or more changes which would make this possible.

18. *More from the Unix case study*

(*a*) Explain why in Unix, the `/etc/passwd` file is publicly readable. What mechnism is used to protect passwords? Why is this inadequate on modern machines? What modification is made to password storage in modern implementations?

(*b*) Draw a diagram of a Unix process, explaining the different regions of the virtual address space and how each changes over the process's lifetime.

(*c*) How is the Unix kernel image loaded from disk at boot-time? What is the role of the `init` process?

(*d*) Explain the usage of the `fork()`, `exec()`, `exit()` and `wait()` system calls in Unix. What is the advantage of separating out `fork()` from `exec()` and what feature provided by other operating systems (such as Windows NT) makes this less useful?

(*e*) What are the three standard communication streams available to any Unix process? What is *I/O redirection*? What is a *pipeline* and what limitation with redirection does it address? Why must pipelined processes have a common ancestor? Give an example of a (non-linear) pipeline topology which cannot be created from the shell (but could be created from a hand-crafted program using system calls directly).

(*f*) To what feature of a (hardware) processor are Unix process *signals* analogous?

(*g*) Suggest (guess) what functionality might be provided by the *cooked* character I/O interface, and what system calls would be used to access it.

(*h*) Why might it be dangerous to cache filesystem metadata?

(*i*) Describe the Unix process scheduler in five words or fewer. What does the *nice* value of a process represent? What else does the scheduler account for when making decisions?

19. *NT design principles*

(*a*) Windows NT's design goals included portability, security, POSIX compliance, multiprocessor support, extensibility, internationalisation support and backwards-compatibility. List which of these design goals motivated each of the following implementation decisions.

(*i*) written in high-level languages

(*ii*) microkernel architecture

(*iii*) modular structure

(*iv*) use of a *hardware abstraction layer*

(*b*) What distinction concerning interrupts is made between the *executive* and the *kernel*? Why is the use of the word *kernel* in NT different from generic use of the word?

(*c*) Explain the relationship between a *process* and a *thread* in NT.

(*d*) What tradeoff is being exploited by the use of different thread scheduling *quantums* between Workstation and Server editions of NT?

(*e*) Draw a diagram of the internal structure of an *object* as found in the NT executive. Which fields are used to implement the *namespace*?

(*f*) To support what kind of application-level I/O interface does NT introduct *I/O request packets*?

(*g*) What is meant when NT's I/O cache system is described as *unified*? What features does the cache manager provide for application-directed "hinting"?

(*h*) Consider the FAT filesystem. What data structure is used to record the sequence of blocks containing each file's data? When is space for this structure allocated? Explain why "big cluster size is *bad*".

(*i*) Consider the NTFS filesystem. In what sense is a file in NTFS *structured*? Explain how use of a special *log file* aids recovery.

(*j*) Later versions of NT moved some graphical interface functionality into the executive. Where did this functionality previously reside? Suggest one service in the NT executive which might have been used by the graphical interface code, and why this could operate faster once the code was moved into the executive.

20. *NT vs Unix*

(*a*) What design goals of NT did Unix specifically not address?

(*b*) For each of the following features of Unix, specify in what way NT's closest analogous features are essentially different, and suggest a benefit and/or drawback in each case:

(*i*) filesystem namespace structured as directed acyclic graph;

(*ii*) process hierarchy;

(*iii*) buffer cache;

(*iv*) synchronous I/O.

(*c*) For each of the following features of NT, specify the closest analogous feature of Unix, if any, and suggest what design decisions justified Unix's approach.

(*i*) hardware abstraction layer (HAL)

(*ii*) NT "subsystems"

(*iii*) graphical user interface part of Win32 subsystem

(*iv*) thread creation

(*v*) local procedure call (LPC)