Example Sheet for Operating Systems I (Part IA)

Solutions for Supervisors
Lent 2015
3 Feb 2015
Note these may be updated in the light of feedback. (Check update time.)

1. (a) Modern computers store data values in a variety of "memories", each with differing size and access speeds. Briefly describe each of the following:
   i. cache memory
   ii. main memory
   iii. registers

   (b) Give an example situation in which operating systems effectively consider disk storage to be a fourth type of "memory".

This is from the first section of the course, basic computer organization, with a brief reference to the "swapping" concept mentioned in later lectures.

Different memory types

Note: don't require details on implementation although expect some students will provide such. General comments on size, speed and basic purpose will suffice.

Cache memory is a relatively fast memory made from SRAM (details of SRAM internals not required). It sits between the processor and the main memory being typically faster and smaller than the latter.

Main memory is usually DRAM; this means that it can be far larger than cache memory (typically approximately 1 transistor per bit: not required for answer). However it is also slower.

Registers are the fastest kind of 'memory'; this is in part due to their implementation, and in part due to their co-location with the ALU etc on chip. However the amount of register space is very limited due to the expense and complexity (ports etc).

Disk as memory

Note: looking for swapping since do not cover demand paging in detail. However it is mentioned in passing and an answer using this as an example is fine.

If an operating system is short on main memory in a multi-programming environment, it can *swap out* a process (i.e. all of its memory contents, etc) to disk, thereby freeing up memory. At this or a later stage a process may also be *swapped in.*

2. (a) Describe (with the aid of a diagram where appropriate) the representation in main memory of:

i. An unsigned integer
ii. A signed integer
iii. A text string
iv. An instruction

(b) Does an operating system need to know whether the contents of a particular register represent a signed or unsigned integer?

(c) Describe what occurs during a *context switch*.

Note: this is from a series of a slides in the first section of the course in which the students learn the way in which various forms of data might be stored in the memory of a simple computer.

Unsigned integer: just binary number interpreted as an integer between 0 and $2^w - 1$, where w is the word size (in bits).

Signed integer: 2's complement – viz. invert (complement) all bits and add 1 to perform unary minus. I expect a diagram or indication in this case about what is different from the unsigned case.

Text string: either as an array of characters or a null terminated sequence of characters, possibly a mention of character coding schemes (ASCII,..)

Instruction: as a structured set of bit fields in which various parts denote the "instruction" and various others the operand(s). Miscellaneous other flags may also be present (e.g. write-back result?, set flags?, etc). *Note: anything plausible*

3. Describe with the aid of a diagram how a simple computer executes a program in terms of the *fetch-execute cycle*, including the ways in which arithmetic instructions, memory accesses and control flow instructions are handled.

The simple computer will fetch an instruction from memory, and then execute it, and then repeat – hence the "fetch-execute cycle". Instructions are decoded by the processor to determine what precisely must be done. Some common instruction types include arithmetic & logical instructions such as add, sub, and, xor, ... Most of these are performed by the ALU and have two input operands (registers or perhaps one literal) and an output operand (a register).

Since these instruction operate on registers, there must be a way to load and store values from and to memory. These instructions must *address* memory in some fashion; that is, allow the computation of a byte address referent. The addressing modes in the simple computer are just register indirect and variants. Loads and stores to memory go over a bus – a shared set of data and control wires.

The address of the "next" instruction to be fetched by the processor is held in an internal register called the program counter (PC). In normal operation, this is incremented by a fixed amount after every instruction fetch so that consecutive instructions in memory will be loaded. To alter this default (e.g. to allow branches, loops or procedure calls). control flow instructions effectively update the PC. Conditional codes are often combined with these so that the PC update is predicated on the outcome of a simple ALU-style operation.

4. *System calls* are part of most modern operating systems.

   (a) What is the purpose of a system call?

   (b) What mechanism is typically used to implement system calls?

A system call is the means by which an application (user-space process) can invoke services within the operating system.

They are typically implemented by using *software interrupts* or *traps*: hardware supported instructions which transfer control to a well-defined entry point in the operating system, switching the processor mode to "privileged" in the process.

5. (a) Operating systems need to be able to prevent applications from crashing or locking up the system, or from interfering with other applications. Which three kinds of hardware support do we require to accomplish this?

   (b) How do applications request that the operating system perform tasks on their behalf?

   (c) What could we do if we did not have the requisite hardware support?

Required Hardware Support

First and foremost we require processor support for at least two modes of operation: user and kernel/supervisor modes. This allows us to have instructions which may only be used in supervisor mode (such as direct access to I/O devices via `inb` etc.).

Secondly we require memory protection hardware to prevent applications from trashing each other / the OS. Configuring this hardware can only be done in supervisor mode.

Thirdly we require a timer device so that we can implement preemptive scheduling. Otherwise an application can execute `while(1);` and lock the system.

Interface with OS

Applications request system services by means of *system calls*. These are implemented by using special instructions which cause a secure transition to supervisor mode.

Coping without hardware support

Main thing we could do would be to use language-level techniques. E.g. we could require that all applications were written in a "safe" language like java, and only load such applications. This would involve severely constraining the interface to the user (i.e. no "poke" command!), but might just work.

6. Explain how a program accesses I/O devices when:

(a) it is running in supervisor-mode

(b) it is running in user-mode

When running in supervisor mode, the program has direct access to I/O devices. In some machines there may be explicit instructions to access devices; in other cases one may simply perform memory reads and writes to a special area of physical memory; these reads and writes propagate across the bus to devices which then interpret them in some fashion. Communicating data to/from devices may involve reads and writes, or may use direct memory access. In either case, interrupts may be used to signal the end of an operation – this will reset the PC to a defined handler which can then do whatever may be necessary.

In user-mode, programs cannot directly access I/O devices; if explicit I/O instructions are supported, they will fault in user mode; if memory access is being used, then these parts of the physical address space will not be mapped into the processes virtual address space. This is for reasons of safety.

Instead, user-mode programs must invoke the operating system to perform I/O tasks on their behalf. This involves a system call or trap which switches from user mode into a well defined handler within the operating system which can then perform the requested operation.

7. From the point of view of the device driver, data may be read from an I/O device using *polling*, *inerrupt-driven programmed I/O*, or *direct memory access* (DMA). Briefly explain each of these terms, and in each case outline using pseudo-code (or a flow chart) the flow of control in the device driver when reading data from the device.

Polling is used in the absence of interrupts, and requires that the host CPU explicitly check ("poll") the status of the device (e.g. by reading its 'status' register). If the device is busy, the CPU simply has to check again, either immediately or at some later time.

Hence a routine to read a character into `(char *)buf` might look like:

```
do {
  status = read_status_register();
} while (status == DEVICE_BUSY);
write_cmd_register(DO_READ);
do {
  status = read_status_register();
} while (status == DEVICE_BUSY);
*buf = read_data_register();
```

In the interrupt-driven programmed I/O case, the "read" routine may *begin* the read operation, but will not wait for it to complete. In fact if the device is busy, it will not even begin the read operation but will rather queue it. It might look like:

```
        status = read_status_register();
        if(status == DEVICE_BUSY) {
           enqueue(buf, request_queueu);
           return;
        }
        current = buf;
        write_cmd_register(DO_READ);
```

The main control flow logic is in the interrupt handler. This is where read completions are handled, and where enqueued operations are begun. The 'read-done' portion might look something like:

```
        *current = read_data_register();        // not reqd if DMA
        if(!is_empty(request_queue)) {
           current = dequeue(request_queue);
           write_cmd_register(DO_READ);
        }
        clear_interrupt();
```

The DMA case is almost identical to the previous one save that the data does not need to be explicitly read when the interrupt occurs; it will already be present in memory. While this doesn't really give us a big win here (with a single byte), it's well worth it when transferring e.g. 4K of data from a disk.

8. From the point of view of the application programmer, data may be read from a device in a *blocking*, *non-blocking* or *asynchronous* fashion. Using a keyboard as an example device, describe the expected behaviour in each case.

In blocking I/O, the invocation does not return until 'complete' — i.e. until the data has been written or read. Hence in our keyboard example, the blocking call would return promptly if data were available in the keyboard buffer, but otherwise would block the process until a key was pressed.

In non-blocking I/O, the invocation always returns immediately, but may or may not be complete. In our keyboard example, the non-blocking read would return immediately with e.g. a return code stating many characters (≥ 0) have been read from the keyboard buffer.

In asynchronous I/O, the invocation always comprises two halves: a request followed sometime later by a response. In our keyboard case, the initial invocation would enqueue a read request along with an I/O completion handler. Then whenever keyboard data became available (perhaps with no delay), the completion handler would be invoked and could then process the data.

9. Process scheduling can be *preemptive* or *non-preemptive.* Compare and

contrast these approaches, commenting on issues of simplicity, fairness, performance and required hardware support.

Preemptive scheduling is where a process can be interrupted or *preempted* in its execution at any time by the operating system (or, more precisely, an interrupt). It is relatively complex, but has scope to implement fairness by ensuring no process "hogs" the CPU. In terms of performance there is some additional overhead in *context switches* which aren't, strictly speaking, necessary. The hardware support required is a programmable or periodic timer which can generate interrupts.

Non-preemptive scheduling refers to schemes in which a process only relinquishes the processor voluntarily (e.g. via `yield`) or implicitly (e.g. when blocking on I/O). This is somewhat less complex to implement since there are fewer times when scheduling decisions need to be made, and less worry about concurrency in user-space. Fairness cannot in general be guaranteed since a rogue process can fail to yield the processor for an arbitrarily long time; however performance is usually good since there are no unnecessary context switches. No additional hardware support is required beyond the ability to save and restore process context.

10. (a) Describe how the CPU is allocated to processes if static priority scheduling is used. Be sure to consider the various possibilities available in the case of a tie.

    (b) "All scheduling algorithms are essentially priority scheduling algorithms."

    Discuss this statement with reference to the first-come first-served (FCFS), shortest job first (SJF), shortest remaining time first (SRTF) and round-robin (RR) scheduling algorithms.

    (c) What is the major problem with static priority scheduling and how may it be addressed?

    (d) Why do many CPU scheduling algorithms try to favour I/O intensive jobs?

Description of Static Priority Scheduling

In static priority scheduling, each process/job/task (wlog: process) is assigned a priority value, usually an integer within some pre-defined range. The scheduler then chooses to schedule the runnable/ready process with the highest priority.

In the case of a tie, two main options are available[1] preemptive or non-preemptive round-robin. In the former case each of the highest-priority processes is allowed to run for up to Q time units, where Q is called the *quantum* . If a process is still running after its quantum is up, it is preempted and placed on the back of the queue for this priority level. Preemption typically also occurs if a higher priority process becomes runnable once more.

Non-preemptive round-robin allows the highest-priority process to run to completion (or, more likely, "to blocking"). This is seldom used.

"All scheduling is priority scheduling..." The RR answer is pushing it a bit; I'd be happy with an answer which says that RR isn't really a priority algorithm.

FCFS can be considered non-preemptive static priority scheduling where the time of arrival into the system is the priority (earlier values corresponding to higher priorities).

SJF can be considered a non-preemptive dynamic priority scheduling algorithm where the (estimated) job completion time is the priority, and with shorter times corresponding to higher priorities.

SRTF an be considered a preemptive dynamic priority scheduling algorithm in which the remaining time is the priority (shortest first). It is more difficult to shoehorn RR into this taxonomy, but just about possible: RR is a preemptive dynamic priority scheduling algorithm in which priority is the amount of time one has been ready waiting for the CPU (with longer waits corresponding to higher priorities), and where a process's "wait time" is reset to '0' (and hence its priority is set to the lowest possible value) once it has been current for an amount of time Q.

Problem with Static Priority Scheduling

The major problem with static priority scheduling is that processes may be starved. This may be solved by making the priorities dynamic, e.g. by aging processes, or by using process accounting information to explicitly adjust priorities.

Why scheduling algorithms favour I/O intensive jobs

Operating systems achieve good throughput by multiplexing together a set of processes/jobs/tasks (wlog: processes). When e.g. a process blocks shortly after being run, this allows a new process to be scheduled while the first process makes progress in its own way (i.e. waiting for I/O completion). As such, I/O intensive processes are "good value" for the scheduler.

11. An operating system uses a single queue round-robin scheduling algorithm for all processes. You are told that a *quantum* of three time units is used.

   (a) What can you infer about the scheduling algorithm?

   (b) Why is this sort of algorithm suitable for a multi-user operating system?

   (c) The following processes are to be scheduled by the operating system.

Process	Creation Time	Required Computing Time
P_1	0	9
P_2	1	4
P_3	7	2

   None of the processes ever blocks. New processes are added to the tail of the queue and do not disrupt the currently running process. Assuming context switches are instantaneous, determine the *response time* for each process.

   (d) Give one advantage and one disadvantage of using a small quantum.

Inference

You can infer that the scheduling algorithm is preemptive. It makes no sense to talk about a quantum for non-preemptive scheduling.

Multi-User OS

A preemptive scheduling algorithm is suitable for a multi-user OS because it does not require that processes co-operate (as a non-preemptive one does).

Determining Response Times

The point of this question is to test the understanding of preemptive scheduling. The first 4 marks should be awarded for something resembling the table below.

The schedule would proceed as follows, where R_i is the amount of time for which process P_i must yet be run.

Time	Current	Queue	R_1	R_2	R_3
0	P_1	P_1	9	–	–
1	P_1	P_1,P_2	8	4	–
2	P_1	P_1,P_2	7	4	–
3	P_2	P_2,P_1	6	4	–
4	P_2	P_2,P_1	6	3	–
5	P_2	P_2,P_1	6	2	–
6	P_1	P_1,P_2	6	1	–
7	P_1	P_1,P_2,P_3	5	1	2
8	P_1	P_1,P_2,P_3	4	1	2
9	P_2	P_2,P_3,P_1	3	1	2
10	P_3	P_3,P_1	3	0	2
11	P_3	P_3,P_1	3	0	1
12	P_1	P_1	3	0	0
13	P_1	P_1	2	0	0
14	P_1	P_1	1	0	0
15			0	0	0

A further two marks should go for the final computation; out-by-one errors (due to misunderstanding the nature of discrete time) probably deserve at least one mark.

P_1 starts at 0, finishes by 15: hence its response time is 15.

P_2 starts at 1, finishes by 10: hence its response time is 9.

P_3 starts at 7, finishes by 12: hence response time is 5.

Small Quantum

Having a small quantum is an advantage in that it can reduce the average response time, particularly for short (e.g. I/O bound) runs. Processes do not have to wait as long until they get their chance to run (in general will need to wait $(n-1) \times q$ time units, where there are n processes in the run queue, and the quantum is q time units).

The down side is that context switches are not free. The smaller the quantum, the more frequent the context switches, and hence the larger the overall scheduling overhead.

12. (a) Describe with the aid of a diagram the life-cycle of a process. You should describe each of the states that it can be in, and the reasons it moves between these states.

    (b) What information does the operating system keep in the process control block?

    (c) What information do the shortest job first (SJF) and shortest remaining time first (SRTF) algorithms require about each job or process? How can this information be obtained?

    (d) Give one advantage and one disadvantage of non-preemptive scheduling.

    (e) What steps does the operating system take when an interrupt occurs? Consider both the programmed I/O and DMA cases, and the interaction with the CPU scheduler.

    (f) What problems could occur if a system experienced a very high interrupt load? What if the device[s] in question were DMA-capable?

Process Life-cycle

Looking for a five-state (or more) picture like that in notes. The states involved will be something like:

(a) *New*: state for newly created processes. Moves from here to *Ready*.

(b) *Ready*: state for any runnable process (i.e. one which could use CPU if granted it). Typically many processes in this state. Moves into *Running* when dispatched by scheduler.

(c) *Running*: state for processes currently executing; only one of these per CPU. Moves to state *Ready* if preempted, *Exited* if exits, or *Blocked* if waits on I/O or some other event.

(d) *Blocked*: state for all those processes currently awaiting an event of some sort. Typically many processes here. Moves into *Ready* once the requisite event occurs.

(e) *Exited*: state for all those processes who have [willingly or unwillingly] called exit.

Could also have various swapped states in there.

Contents of PCB

Lots of things in the PCB including:

(a) Process state.

(b) Memory management information.

(c) Program counter.

(d) CPU registers.

(e) CPU scheduling information.

(f) Accounting information.

(g) I/O Status information.

I expect that 1, 2, 3/4 and 5 will be the most common answers, but the others or anything else plausible will be ok.

SJF and SRTF

They require the length of time each job/task needs to execute (or, more generally, the length of its next CPU "burst").

This cannot usually be obtained easily or precisely, but rather must be estimated. One possibility is to use an exponential average e.g. $\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$ where τ_n is the n^{th} estimate, t_n is the n^{th} sample and $0 \leq \alpha \leq 1$.

Non-preemptive Scheduling

Main advantage is that it is simple. Also can be slightly more efficient since only ever schedule and context switch when one has to [e.g. when a process has just blocked].

Main disadvantage is lack of protection; cannot prevent denial of service. Similarly get larger variance in waiting times.

Action on Interrupt

When an interrupt occurs, the CPU transfers control to a/the interrupt handler. This is low-level software which may need to e.g. demultiplex the interrupt, save some state, etc.

Next an interrupt-specific device driver is invoked which will:

(a) for programmed I/O device:
   - transfer data
   - clear interrupt (sometimes a side effect of transfer)

(b) for DMA device:
   - acknowledge transfer

(c) request another transfer if any more I/O requests pending on device

(d) unblock any waiting processes.

(e) enter scheduler or return

In summary, on an interrupt OS will (a) try to keep I/O device[s] busy and (b) potentially unblock (and dispatch) a process/processes.

Problems with Interrupt Load / DMA

Two main problems can occur. The most notable is that if the interrupt load is high enough, one doesn't get any time to schedule processes at all ("interrupt livelock").

Secondly, if a large amount of DMA traffic is going on, contention on the bus might cause degradation in processor performance.

13. (a) What is the *address binding* problem?
    (b) The address binding problem can be solved at compile time, load time or run time. For *each* case, explain what form the solution takes, and give one advantage and one disadvantage.

This concerns memory management and, in particular, one of the motivations for virtual addressing. The "address binding problem" is explicitly mentioned in the notes. The answers below give more detail than I would expect from the students.

The address binding problem

This is the problem of coordinating the memory addresses issued by a program (e.g. accessing variables, making function calls) with the correct addresses to be used at run time.

Solutions to the address binding problem

One mark for the solution, one mark for an advantage, and one for a disadvantage.

We can solve this at compile time by enforcing a load address a priori (e.g. as for DOS .com files)[2]. An advantage is that this is very simple (and requires no load time or run time support). Major disadvantage is inflexiblity; it is not possible to load or run a program unless sufficient memroy at the required location is available.

We can solve this at load time by *relocating* – i.e. the loader patches up addresses once it knows the base address at which the program is to be loaded. An advantage is that we now have more flexibility than the static compile time option and can use any free (contiguous) memory. Disadvantages include the additional time required for performing relocation, the fact that this must be done on every load (including swap in), and that we still require contiguous memory.

We can solve this at run time by using *virtual addresses*; i.e. dynamically mapping from "program addresses" into "read addresses". Usually this means that the processor will have either paging or segmentation hardware. Advantages include greater flexibility (e.g. with paging can now use any frame of physical memory for any page of a process's image). Disadvantages are the additional hardware and run-time costs (and complexity).

14. For each of the following, indicate if the statement is true or false, and explain why:
    (a) Preemptive schedulers require hardware support.
    (b) A context switch can be implemented by a flip-flop stored in the translation lookaside buffer (TLB).
    (c) Non-blocking I/O is possible even when using a block device.
    (d) Shortest job first (SJF) is an optimal scheduling algorithm.

(e) Round-robin scheduling can suffer from the so-called 'convoy effect'.

(f) A paged virtual memory is smaller than a segmented one.

(g) Direct memory access (DMA) makes devices go faster.

(h) System calls are an optional extra in modern operating systems.

(i) In UNIX, hard-links cannot span mount points.

(j) The Unix shell supports redirection to the buffer cache.

15. Most operating systems provide each process with its own *address space* by providing a level of indirection between virtual and physical addresses.

    (a) Give *three* benefits of this approach.

    (b) Are there any drawbacks? Justify your answer.

    (c) A processor may support a *paged* or a *segmented* virtual address space.

        i. Sketch the format of a virtual address in each of these cases, and explain using a diagram how this address is translated to a physical one.

        ii. In which case is physical memory allocation easier? Justify your answer.

        iii. Give *two* benefits of the segmented approach.

Benefits of per-process address space

There are a number of benefits including:

- Independence: each process can use (nearly) all of the address space without worrying about whatever other processes might currently be running.
- Portability: the size of physical memory is hidden behind the level of indirection $\Rightarrow$ programmers do not need to know about the precise amount of RAM installed on the machine.
- Protection: a process cannot by default read or modify information within another process's address space.
- Static relocation: having the extra layer of indirection means that processes can be linked at a fixed virtual address, and so do not need to be relocated at load-time.
- (related to above): means that demand paging/segmentation is possible.
- Sharing: the same piece of physical memory can be mapped into multiple address spaces (e.g. shared libraries, kernel).

There are certainly more benefits and/or different ways to phrase things. Any valid benefit should receive the marks.

Drawbacks of per-process address space

The main drawback is the fact that sharing between processes needs to be mediated by the operating system, whether explicit (e.g. IPC via mailboxes, pipes, ports, . . .) or implicit (shared libraries as mentioned above).

A second drawback is that additional overhead is incurred upon context switches due to the necessity to swizzle page or segment tables, and the subsequent cost of re-filling address translation caches (TLBs, segment registers or CAMs, virtual tagged caches, etc).

A third drawback is that, even in the best case, additional time (e.g. cost of TLB hit) and complexity (e.g. transistor cost of TLB & extra control logic for handling misses) is required. In practice a certain percentage of accesses will be even more expensive due to page-table or segment table lookups.

Paged/Segmented Virtual Addresses

A virtual address on a paged machine looks something like:

```
        m bits                    n bits
+-------------------------+----------------------+
I Virtual Page Number    I          Offset       I
+-------------------------+----------------------+
```

The first m bits (the virtual page number (VPN)) are translated into a k-bit physical frame number (PFN), where k ¡in. This translation occurs using a page table: a h/w or software supported data structure which maps VPNs to page table entries (PTEs). Each PTE contains the PFN to which the VPN should be mapped, along with various other bits. The diagram explaining the translation process should show the translation of the VPN to the PFN via the page table. It is not necessary to mention TLBs, etc.

A virtual address on a segmented machine looks something like:

```
          x bits                   y bits
+-------------------------+----------------------+
I       Segment Number   I          Offset       I
+-------------------------+----------------------+
```

The first x bits (the segment number) are an index into a segment table, each entry of which contains a segment descriptor (SD). The SD contains a base and a limit, along with various other bits. The translation proceeds by a) checking that the offset is K the limit and, if so, b) adding the offset to the base to produce the physical address. The diagram explaining the translation should show the segment table (and the base and limit fields within it). It is not necessary to mention CAMs, or two level segment tables.

Physical Memory Allocation

Physical memory allocation is simpler when using a paged scheme. This is since allocation is performed at a (relatively) coarse granularity (the physical frame).

This reduces the amount of book-keeping the operating system must do. It also removes the problem of external fragmentation.

Benefits of Segmentation

Although memory allocation is more difficult, segmentation provides some nice features:

- The virtual address space is divided into logically distinct units. This provides a nice abstraction from both the OS and programmer points of view.
- Protection occurs at the segment level ⇒ can occur at an arbitrary (e.g. byte) granularity.
- Since segments represent logical units, the segment level is ideal for supporting sharing.
- By using an explicit "attach", the OS (and the segment table) must only be concerned with segments which are actually in use.
- The above, coupled with the fact that segments tend to be much larger than a page, mean that less space is required in the CAM for equivalent performance.

The above is not an exhaustive list.

16. File systems comprise a *directory service* and a *storage service.*

    (a) What are the two main functions of the directory service?

    (b) What is a directory *hierarchy*? Explain your answer with the aid of a diagram.

    (c) What information is held in file *meta-data*?

    (d) What is a *hard link*? Does file system support for hard links place any restrictions on the location of file meta-data?

    (e) What is a *soft* (or *symbolic*) link? Does file system support for soft links place any restrictions on the location of file meta-data?

Functions of Directory Service

One mark for each function.

The directory service is responsible for name resolution (the means by which pathuames/filenames are resolved to file meta-data), and for access control.

Directory Hierarchy

A directory hierarchy is a tree with variable branching factor (the maximum being implementation- specific). Each non-leaf node in the tree is a directory. Most leaf nodes are files, although some may be directories.

The illustration should clearly show these distinctions.

File meta-data

The most important part of file meta-data is the information describing where on disk the contents of the file are to be found. This may be a simple table of blocks, or a hierarchy of tables, etc.

Other file meta-data information includes the file size (i.e. how much of last block valid), the file type (dir, link, reg), the owner of the file, permissions on the file, the time of last access/creation/modification, the number of links (references) to the directory, etc.

Hard links

A hard link is a reference-counted copy of a file, a directory entry which refers to the same meta-data.

A system using hard links cannot keep file meta-data embedded within directories; an extra level of indirection is required. A creative answer which describes how one could still keep meta-data in directories also deserves a mark providing sufficient detail is given.

Soft links

A soft link is a "file" containing the pathname/filename of another file. In some implementations, the pathname will actually be contained within the directory entry; in others, a real file will be created.

Soft links do not place restrictions on the location of meta-data.

17. (a) In the context of memory management, under which circumstances do *external* and *internal* fragmentation occur? How can each be handled?

    (b) What is the purpose of a page table? What sort of information might it contain? How does it interact with a TLB?

    (c) Describe with the aid of a diagram a two-level page table. Explain the motivation behind the structure and how it operates.

Fragmentation

External fragmentation occurs when we are allocating variable sized partitions/segments; after a number of segments of variable size have been allocated and subsequently freed, small portions of free memory will be left between the allocated regions. These may be too small to be useful. We can "solve" this problem by compacting memory.

Internal fragmentation occurs when we are allocating fixed size pages; requests must be rounded up to a multiple of the fixed page size, and so space is wasted. We cannot really solve this per se, although we can reduce the average waste by using smaller page sizes. Unfortunately this is at odds with other concerns. . .

Page Table Basics

A page table is responsible for mapping from *pages* [fixed size portions of the virtual address space] to *frames* [fixed size portions of the physical address space].

Each entry in a page table contains the corresponding frame (if any), protection information, and a set of flags such as valid, referenced and modified.

The TLB acts as a *cache* for entries in the page table. The page table is only searched whenever an entry is not found in the TLB. Once the information has been retrieved from the page table it is inserted into the TLB.

Two-level Page Table

Diagram as per notes.

Two level page table is essentially a tree of depth two in which every node has the same [large] branching factor. Typically a node will be the same size as a page/frame, although this is not necessary.

Lookup operates by splitting the virtual address into three portions; the lowest s bits are the page offset (where the page size is 2^s) and are ignored. The remaining bits are divided into two [probably] equal portions. The most significant of these is used to index into the *root page table.* The result of this operation is either (a) a pointer to a second-level page table or (b) a "translation fault" [a discussion of superpages is not required]. In the former case, the second less significant portion of the address is then used to index into the second-level page table. This results in a page table entry which describes the mapping [if any] for the original page.

The motivation is that a linear table would be far too large to manage, and it is expected that virtual addresses will be allocated and accessed with some sort of locality of reference. Hence ideally few second-level page tables will be required, and those that are will be relatively full.

File meta-data

The most important part of file meta-data is the information describing where on disk the contents of the file are to be found. This may be a simple table of blocks, or a hierarchy of tables, etc. etc.

Other file meta-data information includes the file size (i.e. how much of last block valid), the file type (dir, link, reg), the owner of the file, permissions on the file, the time of last access/creation/modification, the number of links (references) to the directory, etc.

18. Suppose we have a system with three users a, b and c and ten files f_0,f_1,...f_9.

    Further suppose we have four operations for which we wish to control access: *read*, *append*,*replace* and $modify$.

    (a) Do we require all of these or can some be described by combinations of others?

    (b) Create a nontrivial example set of access tuples of the the form (user, file, permission) and show how it might be represented as i) an access matrix, ii) access control lists, iii) capability sets

19. Describe the basic access control scheme used in the Unix filing system. How does UNIX support more advanced access control policies?

For the first bit, just want the 3 bits (RWX) for owner, group, world along with some explanation of the user and group concept. Semantics when used with directories would be nice, but not necessary.

Unix allows fairly arbitrarily complex access control policies to be implemented by means of the setuid/setgid bits. A program with one/both of these bits set will execute with the relevant effective uid/gid. In addition, the program can programatically check other factors such as calling uid/gid, time of day, day of week, etc, etc, and decide to terminate or continue based on this.

20. (a) Describe with the aid of a diagram the on-disk layout of a Unix V7 filesystem. Include in your description the role of the *superblock*, and the way in which free inodes and data blocks are managed.

    (b) Describe with the aid of a diagram a Unix V7 *inode*.

    (c) Estimate the largest file size supported by a Unix V7 filesystem.

    (d) Suggest *one* reliability enhancement and *two* performance enhancements which could be made to the Unix V7 filesystem.

This question concerns the Unix V7 case study; in particular the first section on the filesystem.

The Unix *V7 filesystem*

A diagram here should show the layout on disk; for the V7 filesystem this is essentially the superblock followed by the inode table followed by the data blocks.

The superblock holds filesystem metadata such as logical block size, size of file system, mount status, number of inodes, start of free-inode list, start of free-block list, etc. Storage is managed by means of these free lists which contain inodes (respectively) blocks which have not been allocated. When a new inode/block is needed, the head of the relevant list is chosen. When inodes/blocks become free (e.g. after a deletion), they are returned to the relevant list.

Description of a Unix *inode*

A diagram here should show the hierarchical use of indirect blocks, etc.

Unix holds file (and directory) meta data in inodes. Each inode holds information such as the ower of the file, the time it was created (and modified, and accessed), the size of the file in bytes, the access permissions, and the location of the file data blocks on disk.

The addresses of the first n (e.g. 10) disk blocks are held directly in the inode, followed by the location of a single indirect block, followed by the location of a double indirect block, followed by the location of a triple indirect block.

Indirect blocks represent a trade-off between the speed of block location and the size of an mode. Since many files are small, having too many “direct” pointers in the mode would waste space. Using (the three kinds of) indirect blocks allows very large files to be supported, while not penalising small ones.

Largest file size estimate

Precise answer unimportant, but means of estimate should incorporate direct, indirect, etc blocks.

Assuming block size of 512 bytes and 4 byte block addresses, each indirect block holds 128 pointers. Hence we expect a max file size of approx $(10 + 128 + 128^2 + 128^3)$ blocks.

Improvements to UNIX *V7 filesystem*

Reliability could be improved by having multiple copies of important data structures (e.g. superblock, root inode).

Performance could be improved by changing the layout so that inodes and data blocks are not so far apart; it could also be improved by attempting to allocated contiguous blocks on disk to contiguous parts of a file (or to related files). Could also use a larger block size to reduce the impact of direct blocks, although would also then need to take into account internal fragmentation (e.g. via using explicit fragment blocks).

21. (a) Compare and contrast *blocking*, *non-blocking* and *asynchronous* I/O.
    (b) Give *four* techniques which can improve I/O performance.

Blocking, non-blocking and async I/O

In blocking I/O, the invocation does not return until 'complete' — i.e. until the data has been written or read. Hence, using a keyboard for example, the blocking call would return promptly if data were available in the keyboard buffer, but otherwise would block the process until a key was pressed.

In non-blocking I/O, the invocation always returns immediately, but may or may not be complete. In our keyboard example, the non-blocking read would return immediately with e.g. a return code stating many characters (≥ 0) have been read from the keyboard buffer.

In asynchronous I/O, the invocation always comprises two halves: a request followed sometime later by a response. In our keyboard case, the initial invocation would enqueue a read request along with an I/O completion handler. Then whenever keyboard data became available (perhaps with no delay), the completion handler would be invoked and could then process the data.

Improving I/O

Options include: reducing number of copies, using DMA wherever possible, using buffering to match device and processor speeds, using caching to avoid I/O wherever possible, and scheduling device requests.

22. (a) Describe, with the aid of diagrams where appropriate, how Unix implements and manages:
        i. a hierarchical name space for files

ii. allocation of storage on disk

iii. file-system and file meta-data

iv. pipes

(b) A Unix system administrator decides to make a ‘versioned’ file-system in which there are a number of directories called `/root-dd-mm-yyyy`, each of which holds a copy of the file-system on day `dd`, month `mm` and year `yyyy`. The idea is that at any particular time only the most recent snapshot will be used as the ‘real’ filesystem root, but that all previous snapshots will be available by explicitly accessing the directory in question. In this way the system administrator hopes to allow resilience to mistaken edits or unintentional deletions by users, or to hardware problems such as a disk head crash.

To implement this, the system administrator arranges for a program to run every morning at 01:00 which recursively ‘copies’ the current snapshot to the new one. However to save disk space, hardlinks are used in place of actual copies. Once the ‘copy’ is complete, the new snapshot is used as the new root.

To what extent will this scheme provide the functionality the system administrator hopes for? What advantages and disadvantages does it have?

Hierarchical name space

Unix supports a hierarchical name by using a tree of directories with a designated root. Each directory contains a list of names and the numbers of their associated inodes; these inodes in turn refer to either further directories, or to files (leaves of the tree).

Storage allocation

The Unix file system (the version covered in lectures) manages storage by keeping a *free list* of blocks which have not been allocated. When a new block is needed, the head of this list is chosen; when blocks become free (e.g. after a deletion), they are returned to the list. The head pointer is kept in the superblock, which lives at the start of the partition.

Filesystem metadata

A diagram here should show the hierarchical use of indirect blocks, etc. Unix holds filesystem metadata in the superblock, and file (and directory) meta data in inodes. The superblock holds information such as logical block size, size of file system, mount status, number of inodes, start of free list, etc.

The per-file [and directory] inodes hold information about the ower of the file, the time it was created (and modified, and accessed), the size of the file in bytes, the access permissions, and the location of the file data blocks on disk.

The addresses of the first n (e.g. 10) disk blocks are held directly in the inode, followed by the location of a single indirect block, followed by the location of a double indirect block, followed by the location of a triple indirect block.

Indirect blocks represent a trade-off between the speed of block location and the size of an mode. Since many files are small, having too many "direct" pointers in the mode would waste space. Using (the three kinds of) indirect blocks allows very large files to be supported, while not penalising small ones.

Pipes

Unix pipes are implemented by using a shared memory buffer and a producer-consumer model; the writer can insert data (byte by byte) only up to a certain threshhold – attempting to write more will cause them to be blocked. The read can read any existing data, but will block if there is nothing to be read. Pipes just look like files (in the sense that they're accessed by file descriptors); however the original ones had no filesystem-visible name, and hence could only be used for IPC between processes with a common parent.

Using links for reliability

As outlined, the scheme does not provide the resilience the system administrator would like; firstly, since hard links all refer to the same inode, a mistaken 'edit' will propagate to all snapshots; secondly, since hards links must refer to files on the same filesystem, there is not really any additional protection to hardware faults. It does mitigate against unintentional deletions since a file must be deleted from every snapshot in order to be truly deleted (although this is arguably a disadvantage too). Finally, the cost will not be quite so 'free' since of course directories must be copied rather than hardlinked to.

23. Describe *how* CPU scheduling algorithms favour I/O intensive jobs in the Unix operating systems.

In Unix, the priority of process j at the beginning of time interval i is given by:

$$P_j(i) = Base_j + \frac{CPU_j(i-1)}{4} + 2 \times nice_j$$

where $CPU_j i$ is a measure of how much CPU time process j has been using.

Since higher integer values in Unix correspond to lower priorities, this means that processes which get a lot of CPU time (i.e. CPU-bound processes) will have their priority dropped over time. This will allow I/O intensive processes to have a go.

Note: the equation is not required; an answer which just explains that Unix *reduces the priority of a CPU-intensive process is fine.*

In Windows NT/2K, an explicit priority boost is given to I/O bound processes when they are unblocked after the I/O they were waiting for completes. The amount of the boost is device (or device-driver) dependent, and decays over time as and when the process completes a quantum.

24. Describe with the aid of a diagram a Unix *inode*. You should explain the motivation behind *indirect blocks*, and how they are used when accessing a file.

An UNIX mode holds file meta-data as described previously. It also holds the locations of the first n (e.g. 10) disk blocks are held in the mode, followed by the location of a single indirect block, followed by the location of a double indirect block, followed by the location of a triple indirect block.

Indirect blocks represent a trade-off between the speed of block location and the size of an mode. Since many files are small, having too many "direct" pointers in the mode would waste space. Using (the three kinds of) indirect blocks allows very large files to be supported, while not penalising small ones.

In a typical UNIX system, the first 10 blocks of a file are directly pointed to. Once this is determined, the file system can load the relevant block from disk.

Given a 512-byte block size, and a 4-byte block address, the next 128 blocks are obtained by a) first reading the single indirect block, and b) indexing into this using $b - 10$, where b is the block offset in the file, and c) loading the relevant block from disk.

The next 1282 blocks are found by reading the double indirect block, indexing into this using $(b - 138)/128$ to obtain the location of a single indirect block, reading the single indirect block, indexing into this using $(b-138)\%128$, and finally reading the relevant block from disk.

The final 128 are found via the triple indirect block. This contains the locations of up to 128 further double indirect blocks. Indexing follows the obvious scheme.

25. Describe the operation of the Unix shell with reference to the process management system calls it makes use of. You might like to use pseudo-code or a diagram to aid with your description.

The main process management system calls used by a shell are `fork()`, `exec()`, `wait()` and `exit()`. The key thing here is to get the ordering / interaction of these correct. Don't require info about backgrounding, job control, process groups, etc.

In its steady state, the shell (parent) repeatedly executes the following steps: read command line, parse command line, locate executable, invoke `fork()`, invoke `wait()`, repeat. The *child* process comes into being after the invocation of `fork()` and will then invoke `exec()` to replace its image with that of the executable, run for a while and eventually invoke `exit()`. This will unblock the parent and allow it to read in a new command line. Note: since the above `exit()` system call is not actually present in the text of the parent it is reasonable for the answer to omit this providing understanding is shown.

Or, in pseudo-code, something like the below:

```
while(!done) {
   line    = read_command_line());
   command = parse(line, /* out /* &args);
   procid  = fork();
   switch(procid) {
      case -1: /* error return */
         printf("fork() failed, error %d\n", errno);
         break;
      case 0:  /* child process */
         execve(command, args);
         break;
      default: /* parent process */
         wait(&status);
         break;
    }
}
```

Or use a diagram from the notes.