1 Processes and Scheduling

1.1.a Modern computers store data values in a variety of "memories", each with differing size and access speeds. Briefly describe each of the following:

- Cache Memory: Fast Random Access Memory stored inside/close to the processor. Used to *cache* heavily used sections of instructions in use such that the processor can access the instructions much faster. Much faster to access Cache Memory compared to Main Memory. Modern machines this is of order of KB.
- Main Memory: Re-writable Random Access Memory, for storing instructions to be loaded by the processor for execution. Modern computers O(GB).
- Registers: Small sections of accessible fast storage (O(8/16/32/64 bits)) available for the processor, which are filled with data and instructions that are then manipulated by the processor.
- Hierarchy: Registers - Cache - Main

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

The disk / secondary storage can be used as an additional form of virtual memory. If additional memory is required (i.e main memory filled up), temporary page files / swap files can be created in the disk storage to act in an identical fashion to the main memory, albeit slower.

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

- Unsigned Int: Represented by n bits, (commonly 8, 16, 32). Values stored in binary for 2^n distinct values. Can be referred to in base 16 / hex for ease.
- Signed Int: Two options
  - Sign / Magnitude: Same as unsigned it, leftmost bit flags (1)negative, others are the absolute value.
  - 2's complement: convert a positive to negative by applying NOT and adding 1
- Text String: An individual character is represented in different fashions based on the encoding used. ASCII uses 7 bits to encode a set of characters, Unicode has 8, 16, and 32 bit versions for larger character sets. Entire text strings are typically stored as arrays of characters. The length of the array can be stored as a precursor bit sequence or implictly through a termination bit sequence.
- Instruction: At an abstract level, instructions comprise an opcode specifying the which action should be executed, along with successive operands detailing where to retrieve values.

Section	Cond	00	I	Opcode	S	Ra	Rd	Operand 2
Bit Range	31-28	27-26	25	24-21	20	19-16	15-12	11-0

As a specific example, the ARM ALU data processing operations:

- Cond: Condition code, execution rules for instruction
- I: Immediate Operand: dictates what Operand 2 is
- Opcode: Explicit instruction
- S: Set condition codes
- Rn: 1st operand register
- Rd: destination register
- Operand 2: if I == 0, Op2 is a register with a shift and a 2nd register. If I == 1, Op2 is an intermediate value with some shift to be applied.

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

An operating system does need to know if the contents of a register are signed or unsigned, especially depending on the representation. With two's complement, addition (as an example CPU operation) is identical regardless of the sign. However, more complex comparisons and operations require knowledge of the type. If the negative value is represented in sign/magnitude, additional knowledge is required to check the sign bit before executing the operations.

1.2.c Describe what occurs during a context switch

A CPU in operation on a single task will run through a set of instructions in a linear transactional fashion. Given processors need to manage multiple tasks at once (user-space and kernel-space processes) in any practical implementation, the CPU is responds to and is directed between processes and events using interrupts and system calls (software triggered interrupt).

This allows software and hardware signals to affect the CPU, and allows support for multitasking in CPU wait situations (IO for example) and decoupling the CPU requests from device responses. In the IO situation, the IO device will raise an interrupt when the data has been fetched. After interrupt the CPU vectors to the handler, reads the data, and resumes where it left of prior to waiting for the IO.

The crucial section here is that the processor must be able to resume processing. To this end, the CPU typically preserves the values of most/all registers in the Process Control Block (PCB, context information for a process).

This process, of an interrupt signaling that the CPU should change tasks, followed by preservation of the current process state and transfer / restore to an alternative task is a context switch. Given no work can occur during a switch, it's all overhead time spent switching.

1.3 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

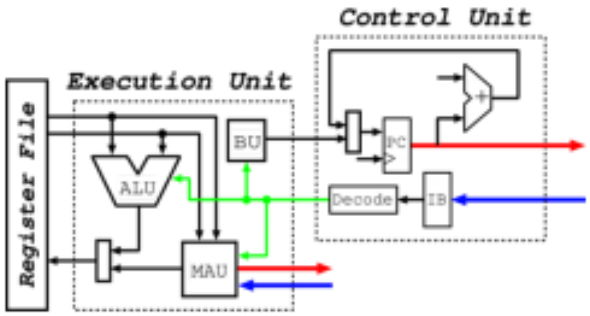

Figure 1: Fetch Execute Cycle

- PC: Program counter, keeps track of the memory address of the next instruction
- MAR: Memory address register, holds address of memory block for r/w
- MDR: Memory data register, holds data fetched from memory or data to be stored in memory
- IR: Instruction Register, Stores fetched instruction
- CU: Control Unit, Records the instruction in the IR, selects and coordinates resources
- ALU: Arithmetic Logic Unit, Performs maths/logic operations
- FPU: Floating point unit, floating point operations

1. Fetch: Fetch the next instruction from the PC, and store in IR.
2. Decode: Interpret the stored instruction
3. Read: If required read from main memory into data registers
4. Execute: Execute the instruction, CU passes decoded information as control signals to other components to read data from registers, instruct ALU to manipulate values and write back to register. If the ALU feed s back a signal, the PC may be updated.

This cycle repeats.

1.4 Process scheduling can be preemptive or non-preemptive. Compare and contrast these approaches, commenting on issues of simplicity, fairness, performance and required hardware support.

Process scheduling methods dictate how processes from some ready queue are taken off whenever the CPU becomes idle. In terms of CPU scheduling, there are 4 states that force a scheduling decision:

1. Process switches from running to waiting/blocked, e.g. due to an I/O request
2. A process terminates
3. A process switches from running to ready, due to a timer expiry or an interrupt
4. A process switches from waiting/blocked to ready, e.g. completion of I/O

If scheduling decisions are carried out only on conditions 1 and 2, the scheduling is *non-preemptive*, otherwise the scheduling is *preemptive*.

In non-preemptive scheduling any process that has the CPU allocated to it will retain the CPU until either of the two conditions are met. This is simpler than preemptive; there's no need to support interrupts and the lack of process switching aside from termination or blocking means no overhead related to process switching, nor a need to implement timers on the CPU. However, this means that buggy or long running processes can hog the CPU, denying service to other processes. MS-DOS and Windows 3.X made use of this scheduling method.

For preemptive scheduling, the flexibility of the CPU is much higher; long-running processes can be preempted to allow execution of short running processes or high priority processes without starvation, it allows for interrupts and is used for all modern OSs. This adjustment can lead to low-priority processes starving and never being executed if higher priority processes are added to the ready queue sufficiently frequently. There are also associated costs in terms of complexity, as timers need to be implemented in the software, and using preemptive scheduling also affects the kernel design as if the kernel is required to process a system call while the prior activity involved modifying the kernel data. If the process is preempted while the kernel structure is in an inconsistent state, problems could ensue without a specific kernel execution model or hardware support. It also leads to race conditions, for example if several processes have shared data and a process is preempted while updating the shared store. There is also the execution overhead of having to manage the process switching from running to ready and vice versa.

Category	Non-preemptive	Preemptive
Simplicity	More simple	Less simple
Fairness	Weighted towards long running processes, short high priority tasks may starve	weighted towards high priority processes, low priority processes may starve if sufficient high priority processes arrive
Performance	Efficient given no overhead from process switching, and no race conditions between processes. Hindered by buggy or long running processes	Supports interrupts for critical / high priority processes, avoids long running / slow code derailing or hindering processes. Overhead due to support for timers and interrupts means code is slower in the perfect case, but better overall
Hardware	No specific hardware support aside from CPU	Requires kernel logic to be designed to support interrupts and changes when the kernel is in an inconsistent state, and requires the hardware to have a timer and interrupt channel to allow for processes to be preempted.

1.5.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.

In static priority scheduling processes arrive with some object measure of priority associated with the process (typically represented by an integer). With *static* priority scheduling, these priority levels are fixed for the process across all time. When the CPU is free to be reallocated, the current highest priority task in the queue will be executed.

Considering the example given in the notes, for pure static scheduling where the CPU is not interrupted by higher priority tasks:

Process	Arrival Time	Priority	Burst Time	Execution Time	Wait Time
P_1	0	3	7	0	0
P_2	2	2	4	8	6
P_3	4	1	1	7	3
P_4	5	2	4	12	7

Here we've used first in to tie break which of P_1 and P_4 should have run first. One option for tie breaking is to use round robin scheduling on those tasks, where the tasks are split over the CPU based on a set unit of time and alternate. Or as given in the example, use First Come First Served to assign the CPU.

1.5.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.

The statement can be see to arise from consideration of SJF scheduling; while the scheduling is technically derived from the burst time of the job, this is in terms of execution identical to priority scheduling where the priority of a process is $\propto \frac{1}{bursttime}$. Considering the algorithm in this fashion, you can describe the algorithm as static priority with a calculated priority per task rather than provided.

Considering FCFS, the algorithm can be contorted to be described as a priority algorithm, albeit it's probably an over complication of what is carried out by a simple FIFO queue. To frame FCFS as a priority algorithm, each process that is added is assigned a priority based on the order it arrived, e.g. by incrementing a value (a hypothetical not practical solution) per new process and assigning that new value to the new process, with low values taking priority.

SRTF can also be couched in the logic of priority, where SRTF will always allocate (and interrupt if necessary) the CPU to the current highest priority process, where the priority is the inverse of the remaining time for any given process.

RR scheduling seems to contradict the statement, as it treats all processes as equally important. You could stretch the definition as with FCFS considering the circular queue that underpins RR, but each process will receive an equal share of the CPU up to the point of completion, and given the fact that all processes are worked on unlike in plain FCFS it seems incorrect to describe it as a priority queue, although you could do at a stretch.

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

The principal problem with static priority is the starvation problem: given a sufficient volume of incoming higher priority processes, lower priority processes will be ignored; there is no guarantee a low priority process will ever be executed. The rumour/urban legend of a 6 year low priority process that was never allocated at MIT in the 70s is illustrative of this. This can be addressed by changing from static priority to *dynamic* priority, where the priority of a process changes after its arrival in response to other factors. A simple approach is aging: as a process sits without being allocated, its priority is steadily increased in proportion to the time, which would make it eventually eclipse the incoming higher priority tasks in priority and have CPU allocated. Using a RR style scheduler would also mean that low priority tasks are executed, as the RR has no concept of task priority.

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

I/O intensive jobs are those that spend a good fraction of their total process time blocked waiting for I/O functions. When an IO intensive job is scheduled it will be worked on until it is blocked waiting for the I/O. At this point the CPU is then idle. By favouring I/O intensive jobs over CPU intensive jobs, the CPU can start the I/O process then complete or progress the CPU intensive job while the I/O task is blocked waiting for the I/O. This is more efficient than if the process prioritised the CPU intensive job, which would result in the CPU intensive job being completed, the I/O job started and then the CPU sitting idle waiting for the I/O in this example with one of each process. Overall favouring I/O processes can result in more efficient use of the CPU.

1.6 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.

1.6.a What can you infer about the scheduling algorithm?

Not sure beyond what is already given, round-robin scheduling. Preemptive scheduling with support for interrupts in the kernel. OS is looking for responsiveness overall. Unclear without more detail as to whether the quantum chosen is on the small or the large scale.

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

As mentioned, responsiveness is a key aspect of RR scheduling. With each process guaranteed to be allocated a section of CPU within a short amount of time, processes that originate from distinct users will show a result or at least a response, instead of lower priority / late arriving processes being forced to wait, the source of which would be unclear for the user in question.

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

Process	Creation t	Required t	Completion t	Response t	First Switch t	First Response t
P_1	0	9	15	15	3	3
P_2	1	4	10	9	6	5
P_3	7	2	12	5	12	12

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

RR scheduling involves significant context switching, which possesses a degree of overhead. High frequency context switching if the quantum is too small can result in significant overhead on the processes. The benefit of a smaller time quantum is that it can reduce turnaround time for processes with smaller burst times, when being handled with longer burst time processes. The small turnaround time means less time is initially spent on the long process and the smaller burst processes can finish within a single time quanta.

1.7.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.

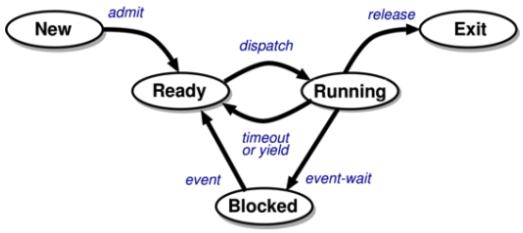

Figure 2: Fetch Execute Cycle

- New: The process is being created
  - Process can be *admitted* from new, moving to Ready
- Running: the process instructions are being executed
  - Process can *released* or can *exit* at which point it is Terminated
  - Process can be interrupted/timeout/yield the CPU and move back to Ready
  - Process can be made to wait for an event or I/O moving to Waiting
- Waiting: The process is stopped waiting for an event (I/O completion, signal reception)
  - Process moves back to Ready at the completion of the event
- Ready: The process is ready to be allocated a CPU to run, waiting
  - Process is dispatched from Ready to Running at the behest of the scheduler
- Terminated: The process has finished executing

1.7.b What information does the operating system keep in the process control block?

The operating system keeps a Process Control Block for each running process, which contains:

- Process State: The current state / lifecycle stage of the process
- Program Counter: The address of the next instruction for the process
- Process Number / ID / PID: Identifier for the process
- CPU Registers: This section varies significantly with the architecture, but consists of a collection of accumulators, index and general purpose registers, stack pointers and any condition code information. This state information is saved on interrupt to allow the process to be restarted.
- Scheduling Information: Information pertinent to the process scheduling: priority, pointers to queues and other parameters
- Memory Management Information: Contain values of the base (smallest legal memory address) and limit (range of allowed memory addresses) registers, page tables (base address of each individual page in memory), segment tables (mappings between the allocated segments of the overall 1D linear memory)
- I/O status information: list of allocated devices, open files etc.
- Accounting information: CPU and Real Time elapsed, time limits, account, job and process numbers, name of executable, owner
- References to adjacent PCBs

1.7.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?

SJF requires the burst time for a process, and does not require any modification to that value past the point of first receipt. SRTF requires the burst time for a process, and also the elapsed time on the process to calculate the remaining time.

However, knowing the burst time ahead of time is difficult for long running jobs and very difficult at low level short term scheduling. As a result, the typical approach is to approximate or calculate a predicted value for the next CPU burst, and use that as the value. The conventional method of approximating the burst time τ_{n+1} is the exponential average of the prior burst times, using a constant $0 \leq \alpha \leq 1$, for the time of the previous burst t_n and estimated time of previous burst τ_n

$$\tau_{n+1} = \alpha t_n + (1-\alpha)\tau_n = \alpha t_n + ... + (1-\alpha)^j \alpha t_{n-j} + ... + (1-\alpha)^{n+1}\tau_0 \qquad (1)$$

for some constant τ_0. The value of α is chosen based on some understanding of the system (if prior tasks are irrelevant $\alpha \approx 1$). This average is a good predictor if the variance of process bursts is small, and weighs older times less than more recent times. These values can then be fed into SJF and SRTF as the burst time.

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

Non-preemptive scheduling is simpler to implement and has less overhead than preemptive scheduling, due to the lack of a requirement for any kind of timer or interrupt system. The principle disadvantage is the vulnerability of the scheduler to long running / buggy process which deny service to the other processes and cause their starvation.

1.7.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.

When an interrupt occurs, it is detected by the CPU on a specific interrupt-request line that is checked after each instruction execution. When detected, the CPU saves the state in the PCB for the process and jumps to an interrupt handler. This handler determine the source of the interrupt, performs any necessary processing and the performs a state restore and returns control to the CPU.

CPU typically have two interrupt lines, maskable (which can be ignored) and nonmaskable. Each interrupt mechanism accepts an address, typically an offset value in the *interrupt vector* which is a memory address to a specialised set of interrupt handlers for different issues.

Programmed I/O is where the CPU manages transfers within the OS, watching status bits and feeding data to IO controllers. This can be considered wasteful, as the basic I/O commands aren't comparable to the range of a general purpose CPU register. Modern system avoid this issue by offloading the work to a specialised Direct Memory Access (DMA) Controller. A DMA transfer writes a command block describing the transfer to memory, which the DMAC then handles by transferring from disk directly to memory using the memory bus directly without the CPU.

An interrupt driven I/O cycle is as follows:

1. The device driver in the CPI initiates an I/O operation
2. The I/O Controller initiates the I/O operation on its end
3. The I/O operation generates an interrupt signal (i.e. input ready, output complete, error)
4. The CPU, which has been handling other instructions and checking for interrupts receives the interrupt and transfers control to the interrupt handler via the interrupt vector
5. The interrupt handler processes the data
6. The interrupt handler returns from the interrupt
7. The CPU resumes the processing on whatever task were interrupted

A DMA transfer cycle, using FreeBSD as an example when an IO device has data it wants to transfer to main memory, is as follows:

1. The CPU initiates a DMA transfer with target address and byte count C, by alerting the I/O device
2. The IO device is prepared, and alerts the DMAC
3. The DMA performs internal checks and requests the memory bus from the CPU
4. The CPU detects the interrupt, progresses to where it can release the memory bus and releases it to the DMAC
5. The CPU can progress up to the point it requires the memory bus again
6. The DMAC then signals the I/O device to transfer the byte via the memory bus to the target address in main memory
7. The DMAC completes work for this byte and releases the memory bus, decrementing C
8. The CPU begins its pre-existing work again
9. This loops until C reaches 0, at which point the DMAC interrupts the CPU to signal completion

Interrupts are the primary point at which the scheduler is called upon. The CPU receives an interrupt, handlers the interrupt appropriately and is then faced with which task to resume with. At this point, the scheduler is used to determine which process to allocate the CPU to. Interrupts that cause the scheduler to be invoked would typically be external interrupts from I/O devices and so on but would also arise (in modern processors) from clock interrupts where the internal clock timer on the CPU determines sufficient time has passed to reassess, similar to RR scheduling.

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

Primarily, the issue is with high interrupt load the core work of the CPU is neglected, whatever processes were attempting to use the register will continually be set aside as the CPU devotes time and resources to handling the incoming interrupts. In the absurd case, you could imagine that the CPU exclusively processes interrupt handling instructions, and never manages to free itself for the main processes. The net result would be processes would starve and remain incomplete, or at least take a significant portion of time.

If the device was DMA capable, interrupts relating to transferring from I/O to main memory could be have their processing offloaded onto the DMAC, alleviating the pressure on the CPU and allowing the general processes to be worked on. This would cut down the interrupt rate to a degree, but only for memory access issues.

In summary, high volumes of interrupts load the processor and result in less processing time spent on the other tasks.

2 Memory Management

2.2 What is the address binding problem?

In the machinecode of a source program, addresses to variables or memory locations are typically symbolic, such as assigning a variable `count`. This symbolic reference to a memory location is useful for the source program, but cannot be used in with main memory as firstly, the memory has no concept of the symbol used and additionally the memory space for that program is not guaranteed to be constant. This is the address binding problem, the difficulty of translating the symbolic or relative address references from a source program to absolute addresses within the hardware's memory space.

2.3 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.

- Compile Time: assuming at the time of compilation it is known where within the memory space the program will reside, *absolute* addresses can be generated for the software, say be beginning at the designated start point for the code R and moving up from that address. If for whatever reason the execution point of the program changes, complete recompilation is required.
  - Requires no additional recalculation post compile time, the binary can be loaded and executed
  - Requires strictly setting a specific start point, hard to then guarantee different programs can work together and hardware changes would necessitate unique compilation.
- Load Time: if the base start point is unknown, it's possible to generate relocatable code. The addresses can be envision as being generated relative to an unknown starting value which when know at load time, each address is finally bound to an absolute address. If the start point changes, the code can be reloaded with the new start point without needing recompilation.
  - Can adapt to a new start point by reloading
  - Although that may be a destructive operation from the perspective of the process, and needs to be calculated every time the process starts.
- Execution Time: If the memory space for the OS is constantly changing and processes need to be moved to better accommodate the full set of processes, the code needs to support being shuffled around, and the binding of the symbolic references needs to be delayed until run time. This requires special hardware such as a Memory Management unit to facilitate the dynamic relocation of the program. Most modern operating systems use this approach.
  - No need to reload or adapt the software when the location changes in the memory
  - Needs specialised MMU hardware

2.4 For each of the following, indicate if the statement is true or false, and explain why:

2.4.a Preemptive schedulers require hardware support.

True: to enable preemptive scheduling the hardware needs to allow for a process to be preempted/interrupted. This is done by dedicated interrupt lines monitored by the CPU and / or a timer / clock.

2.4.b A context switch can be implemented by a flip-flop stored in the translation lookaside buffer (TLB).

Translation Lookaside buffer is a small fast lookup hardware memory cache used to store previously calculated translations between virtual memory addresses and absolute memory addresses, for the sake of rapid memory access in the case of sophisticated virtual memory management (paging / segmenting)

Flip-flop: binary state storage

True The translation lookaside buffer provides access to memory. If the section it is providing access to is the stack for one process, and the flip-flop changes the TLB to point to a section of memory used as the stack for a separate process, a context switch has been enacted.

2.4.c Non-blocking I/O is possible even when using a block device.

True: all devices are effectively blocking, but with threading or context switching you can, provided there is a call back mechanism for the device to signal completion. The first thread starts the IO, tells a second thread to wait for the call back and continues occasionally checking the request.

2.4.d Shortest job first (SJF) is an optimal scheduling algorithm.

True: SJF scheduling, in the perfect theoretical sense is an optimal algorithm. Running SJF results in maximum throughput for a system and minimum average waiting time for processes. The caveat is that SJF cannot be truly applied in practice, as knowing the burst time ahead of execution is impossible.

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

False: The convoy effect is an issue in FCFS scheduling where all tasks end up waiting behind a single massive long running task. With RR scheduling each process is allocated a block of fixed time, so the long running problem process will be interrupted and other processes executed in its place.

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

Example: The paged and segmented strategies of 32-bit Intel architecture, with the total addressable memory space of 2^32.

Segmented: IA-32 permits maximum segment size of 4GB and a maximum segment number per process of 16K, divided between 8K private to that process and 8K shared between all processes. With a segmentation table for each partition. A logical address is 48bits, with a 16 bit segment selector and 32 bit offset. Each entry in the selector table is a base (32 bit address) and a limit (20 bit). As each process can have it's own local segment table, this could get rather large. (8000 * 52 + N * 8000 * 52)

Paged: IA-32 allows two page sizes (4kB and 4MB), with a two level paging scheme for a 32 bit linear address,

Page Size	10	10	12
4kB	Page Directory, points to inner page table	Inner page table, points to page	offset in 4kb page
4MB	Page Directory, points to actual page	combined with 12 bit section to form 22 bit offset	

In IA-32, each structure has 1024 entries of 32 bit each, meaning the total possible size of the page structures is (1024 * 4 * 1024) + 1024 * 4, so just over 4MB of paging table.

So for a worst case scenario, the segmentation would require 10 processes each occupying all 8K possible segment to match the memory commitment of page tables.

False Paged virtual memory requires the full set of page tables given the architecture in question, which will be larger than a comparable segmentation table, as the segments can be sized as required.

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

If devices mean the Machine as a whole, **True**, if devices refers to components or I/O Hardware, **False**. DMA doesn't speed up the I/O hardware, it frees up the processor to operate while it would be waiting for the I/O or blocking operation, which can improve the overall processing speed of the device, but wouldn't speed up the particular action that is blocking the processor.

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

False: System calls allow processes to execute low level operating system functionality. As such low level functionality will always be required, it can be said that system calls will always be required. However, it's not necessary that the processes use these low level system calls, they can be concealed behind an API or a System Call Interface. With this set up you could make an argument that the system calls are optional as only these interfaces are required.

2.4.i In Unix, hard-links cannot span mount points.

2.4.j The Unix shell supports redirection to the buffer cache.

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

2.5.a Give three benefits of this approach

- Simplicity: Each process can be present with a neat linear block of memory allocated to it, irrespective of the overall set of processes in the OS. Each process doesn't have to worry about fitting its memory allocations into the physical memory space.
- Memory Space Increase: By using virtual memory, the total allocated memory for the system can exceed the physical memory of the machine.
- Security benefits: By allowing each process its own address space, a process cannot access memory outside of its allocated space, which has obvious security benefits. *This is not necessarily true, out of memory / stack exploits are a common attack method*

2.5.b Are there any drawbacks? Justify your answer.

In terms of raw efficiency, overhead. As an example, for the IA-32 paging architecture an address lookup has to go through two tables before accessing the byte in question. This can be alleviated with hardware; translation lookaside buffers registers and so on but is still slower than direct absolute lookup. As mentioned though, this requires specific hardware designs, and increased complexity of the operating system.

2.5.c A processor may support a paged or a segmented virtual address space.

- 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.

  Paging:

Page Number	Page Offset
p	d
m - n	n
10 + 10	12

  Logical address consists of page number and offset. The page number refers to a page table that translates to an occupied page in memory. The offset is then used to access the bytes in question.

  Segmented: Logical Address is a segment number and an offset. The exact sizes of which vary. The segment number refers to an entry in the segment table for that process. That entry is a limit and a base. The base refers to the absolute memory address of the allocated segment and the limit is the size of the segment. The

logical address maps to the segment, the segment maps to the absolute address and the offset gives the exact byte. If the offset exceeds the limit the processor will complain.

- In which case is physical memory allocation easier? Justify your answer.
- Give two benefits of the segmented approach.