Operating Systems
Sample Exam Questions and Answers

Tommy Sailing

1. Describe the two general roles of an operating system, and elaborate why these roles are important.

The first general role of an operating system is to provide an ABSTRACTION layer for software to run on a machine without needing to know hardware-specific implementation details. It is important in order to reduce the burden on application software developers, extend the basic hardware with added functionality and provided a common base for all applications. The second general role of an operating system is to provide RESOURCE MANAGEMENT to the machine's users, by ensuring progress, fairness and efficient usage of computing resources.

2. Using a simple system call as an example (e.g. getpid, or uptime), describe what is generally involved in providing the result, from the point of calling the function in the C library to the point where that function returns.

A system call is completed as follows:
- As the function is called, an interrupt of the type "software exception" is placed on the processor, causing a Context Switch to take place between the calling function and the kernel.
- The exception handler will clear out and save user registers to the kernel stack so that control may be passed on to the C function corresponding to the syscall.
- The syscall is executed.
- The value(s) returned by the syscall is placed into the correctly corresponding registers of the CPU (the same ones that a user function normally places its return values in).
- The handler takes this value, restores user registers and returns said value to the user programme that called it.

3. Why must the operating system be more careful when accessing input to a system call (or producing the result) when the data is in memory instead of registers?

The operating system may access memory without restriction (as opposed to user mode where memory access is highly regulated by the OS… we hope). When the data is in memory the OS must be careful to ensure that it is only accessing data that it needs to, since carelessness might result in overwriting data pertaining to still-running user mode functions, breaking their operating when the OS shifts scope from kernel mode back to user mode.

4. Is putting security checks in the C library a good or a bad idea? Why?

It is maybe a good idea if performance is a concern, as it bypasses the context switch (from user mode to kernel mode and back) which is an expensive (time-consuming) operation, while providing a basic level of protection from badly written programmes. However, it would be stupid not to put security checks in the kernel libraries because the user-land C libraries can be freely attacked by malicious users and programmes.

5. Describe the *three state process model*, describe what transitions are valid between the three states, and describe an event that might cause such a transition.

The three-state process model dictates that a process may take the form of one of three states, RUNNING, READY and BLOCKED. Valid transitions include:
- RUNNING to READY (timeslice process management)
- BLOCKED to READY (when a H/W peripheral becomes free)
- READY to RUNNING (the scheduler decides it should run)
- RUNNING to BLOCKED (the process needs some input)

6. Multi-programming (or multi-tasking) enables more than a single process to apparently execute simultaneously. How is this achieved on a uniprocoessor?

Multiprogramming is achieved on a uniprocessor by the concept of “threading”. Every process' total running time is divided up into threads, which are a subset of the process' instructions that can be completed in a certain amount of time, called a timeslice. When a thread's timeslice is finished, CPU time has to switch to a different thread. On a large scale, these timeslices are nanoseconds long, so it appears to the user that the processor is processing processes concurrently. The ultimate goal is to keep the system responsive while really maximising the processor's ability to process.

The above scenario is known as Pre-Emptive multitasking. An alternative scheme is Cooperative Multitasking, where each process occasionally yields the CPU to another process so that it may run.

7. What is a process? What are attributes of a process?

A process is a 'task' or a 'job' that an individual programme has pushed to the CPU. It is allotted a set of resources and will encompass one or more threads. Its attributes fall under the headings of PROCESS MANAGEMENT (including registers, PC, PID etc), MEMORY MANAGEMENT (pointers to CODE, DATA and STACK segments) and FILE MANAGEMENT (root directory, CWD, UID, GID etc).

8. What is the function of the *ready queue*?

The ready queue is a queue of processes in the READY state of the three state process model. A process will enter the READY queue when it may be executed without waiting for a resource. The queue exists to establish a fair and efficient order for processes to be executed. One way to implement such a queue is in a first-in, first-out (FIFO) round robin scheme.

9. What is the relationship between threads and processes?

A thread is a subset of a process, of which numerous threads can be formed. Threads have advantages such as sharing the address space and global variables, while also having its own stack, program counter and alotted registers.

10. Describe how a multi-threaded application can be supported by a *user-level threads package*. It may be helpful to consider (and draw) the components of such a package, and the function they perform.

The kernel sees each process as having:
- Its own address space,
- Its own file management descriptors, and
- **A single thread of execution.**

Incorporating a user-level thread package into the programme will have a beneficial effect on its performance, a way to visualise it is **multiplexing user level threads onto a single kernel thread**. The process's scheduler we note is **separate from the kernel's scheduler**, and is often cooperative rather than pre-emptive so we must be careful of blocking operations. Generally we can attain good performance as we take advantage of virtual memory to store user level control blocks and stacks.

11. Name some advantages and disadvantages of user-level threads.

Advantages of user-level threads include:
- Theoretically greater performance, as the OS does not need to perform expensive context switches every time a thread changes.
- More configurable, as you are not tied to the kernel to decide a scheduling algorithm, nor do you require kernel support for multiple threads.

Disadvantages of user-level threads include:
- Realistically worse performance, because many threads require a context switch regardless (say a syscall is called, such as on an I/O event). This will force a switch into kernel mode and BLOCK every single other user-level thread in that process from running, because the kernel treats the entire user-space of threads as a single process.
- User level threads are generally co-operative rather than pre-emptive, so must manually yield() to return control back to the dispatcher – a thread that does not do this may monopolise the CPU.
- I/O must be non-blocking – this requires extra checking just in case there is an action that *should* block.
- We cannot take advantage of multiprocessors, since the kernel sees **one process with one thread.**

12. Why are user-level threads packages generally cooperatively scheduled?

User-level thread packages are co-operatively scheduled because generally they form part of a single kernel-level thread. Most often this means that the process runs on its own user-level scheduler, separate from the kernel scheduler. This means it does not have access to the strict timing-based pre-emptive scheduling that kernel level threads enjoy, so must use cooperative scheduling.

13. Enumerate the advantages and disadvantages of supporting multi-threaded applications with *kernel-level threads*.

The advantages of kernel-level threads include: Ability to pre-emptively schedule threads as user level scheduling generally only supports the (potentially buggy) co-operative scheduling – with the added benefit of not requiring yields everywhere; Each thread is guaranteed a fair amount of execution time

Disadvantages of kernel-level threads include: No benefit to applications whose functions stay staunchly in user land; a significantly smaller address space; a significantly smaller and possibly close to full stack; code is less portable as OS support is required; thread management is expensive as it requires syscalls

(Hmmm. Not sure about this. Must recheck tute)

14. Describe a sequence the sequence of step that occur when a timer interrupt occurs that eventually results in a context switch to another application.

- Timer Interrupt is called
- Trap into kernel space, switching to kernel stack
- Current application's registers are saved
- Scheduler gives next thread that should be run
- Context switch to other thread (load its stack, flush the TLB)
- Load user registers from the kernel stack
- PC register shifts to the beginning of the other application's instruction

15. Context switching between two threads of execution within the operating system is usually performed by a small assembly language function. In general terms, what does this small function do internally?

Saves the current registers on the stack, then stores the stack pointer in the current thread's control block. We then load the stack pointer for the new thread's control block and restore the registers it had saved, from its stack.

16. What is a *race condition?* Give an example.

When two (or even more) processes try to concurrently access a shared resource, leading to odd behaviour, deadlocks or mistakenly overwriting memory addresses.

Typical example from lectures: Two processes simultaneously updating a counter (lol, Assignment 1 math.c). Say Process A reads the variable but process B steps in and reads and writes to it. Control is passed back to process A, who just writes to it. You've just written to it twice. Ouch.

17. What is a critical region? How do they relate to controlling access to shared resources?

A critical region is a section of code that essentially contains a variable(s) that can be accessed by multiple threads, but will affect the operation of the entire program, or system. Correctness relies on critical regions not being modified by two or more different processes at the same time.

18. What are three requirements of any solution to the critical sections problem? Why are the requirements needed?

Mutual Exclusion – One thread cannot access the critical region while another is inside.
Progress – The critical sections solution must not impede an entire thread's progress, halting the CPU.
Boundedness – We shall not *starve* any process, the resource must be freed so that each thread can access the critical region fairly.

19. Why is *turn passing* a poor solution to the critical sections problem?

Turn passing is a poor solution because we introduce **busy waiting.** While a process is accessing a critical section, we don't want a process waiting it's turn for the critical section to be idling away, doing nothing useful. We should be utilising the power of the CPU at all times. We lose the **progress** requirement.

20. Interrupt disabling and enabling is a common approach to implementing mutual exclusion, what are its advantages and disadvantages?

Advantages:
- It actually succeeds in enforcing mutual exclusion.

Disadvantages:
- You have the result of **busy waiting** again, a poor approach. All other processes must simply wait for the currently running one to finish, and any IRQs sent to the CPU during that period will be ignored. Which, y'know, impedes **progress.**
- Obviously, only works in kernel mode.
- Does not work on multiprocessor systems.

21. What is a test-and-set instruction? How can it be used to implement mutual exclusion? Consider using a fragment of psuedo-assembly language aid you explanation.

A test and set instruction is a method of using the lock-variable solution, but with hardware support (instructions are in ASM and called before entering any critical region). Prior to entering the critical region, it copies the lock variable to a register, sets it to 1 if it was 0 (free), acquire the lock and execute. If not, another process had it. By being called as ASM instructions, it is guaranteed to be atomic:

```
enter_region:
      tsl register, lock
      cmp register, #0
```

```
     b enter_region        ; if non-zero the lock was set, so loop!
     b critical_region     ; else, the lock was 0, so get in there

leave_region:
     mov lock, #0          ; store a 0 in the lock
     b caller
```

22. What is the *producer consumer* problem? Give an example of its occurence in operating systems.

The producer-consumer problem is a idea where two threads exist, one is “producing” data to store in the buffer and the other is “consuming” that data from said buffer. Concurrency problems arise when we need to keep track of the number of items in the buffer, which has a fixed limit on how many items can be inside it at any one time.

23. A *semaphore* is a blocking synchronisation primitive. Describe how they work with the aid of pseudo-code. You can assume the existance of a thread_block() and a thread_wakeup() function.

Semaphores work by blocking processes with P, calling thread_block(), waiting for a resource if it is not available, then being put into a queue of processes that want to access this resource. This is more efficient than other solutions because we avoid busy waiting – other processes can do other things while blocked ones are in the queue! When a resource is released, V is run, which calls thread_wakeup() to signal the next thread to use it. See:

```
typedef struct _semaphore {
     int count;
     struct process *queue;
} semaphore;

semaphore sem;

void P(sem) {
     sem.count--;
     if (sem.count < 0) {
          /* add process to sem.queue */
          thread_block();
     }
}

void V(sem) {
     sem.count++;
     if (sem.count <= 0) {
          /* remove a process from sem.queue */
          thread_wakeup();
     }
}
```

24. Describe how to implement a lock using semaphores.

You place a P(mutex) before writing to a critical variable, then a V(mutex) to signal that the operation is complete so that other processes may have their turn.

25. What are monitors and condition variables?

Monitors are a higher level synch primitive that takes the form of a data type, in which the compiler defines mutual exclusion. When a thread calls a monitor which is already used, it is queued and sleeps until the monitor is free again – very much like a semaphore, but needs to be defined beforehand as part of the OS. A **condition variable** is a wait() and signal() primitive for monitor functions.

26. What is *deadlock?* What is *startvation?* How do they differ from each other?

Deadlock is the phenomenon that arises when a number of concurrent processes all become blocked waiting for another thread to make available a resource, but it can only be released by a thread that is still blocked. **Starvation** is the phenomenon which arises when a process does not ever receive the resource it is waiting for, even if it repeatedly becomes available, as it is always allocated to another waiting process.

Processes halt in **deadlock** becayse they cannot proceed and the resources are never made available. Therefore, no progress can be made. With **starvation**, progress is made overall at the expense of a particular process or processes, which consistently miss out on being allocated their requested resource.

27. What are the four condtions required for deadlock to occur?

The four conditions required for deadlock to occur are:
Mutual Exclusion – the processes must be trying to access the same resource at the same time
Circular Wait – the processes exist in a circular chain, where each is waiting for the resource held by the next member of the chain.
Hold & Wait – process holds a resource, DOESN'T GIVE IT BACK, and blocks because it's waiting for more
No Preemption – resource can't be forcibly taken from the process holding it

28. Describe four general strategies for dealing with deadlocks.

Ignorance – If deadlocks are not liable to happen, the effort required to deal with them outweighs the problem of deadlocks actually occurring.
Detection and Recovery – Keep log of resource ownership and requests. If no progress is made, recover from said deadlock by pre-emption (steal a resource from another process), rollback (make checkpoints – but operating systems aren't *Halo* or *Call of Duty*, this is difficult), or crudely killing processes in the deadlock cycle.
Deadlock Avoidance – The most difficult option. We disallow deadlock by setting “safe states”, in which process completion is always guaranteed.
Deadlock Prevention – Negate one of the four deadlock conditions. Most commonly we deal with the *circular wait* condition. Attacking mutex is infeasible, attacking hold and wait is prone to starvation, and attacking no preemption is downright idiotic.

29. For single unit resources, we can model resource allocation and requests as a directed graph connecting processes and resources. Given such a graph, what is involved in deadlock detection.

We detect deadlocks by finding *closed loops* in the graph, where two or more processes requesting resources are held by other processes.

30. Is the following system of four processes with 2 resources deadlocked?

Current allocation matrix

P1	1	3
P2	4	1
P3	1	2
P4	2	0

Current request matrix

P1	1	2
P2	4	3
P3	1	7
P4	5	1

Availability Vector

1	4

Process 1 uses 1 of resource 1, 2 of resource 2, making the availability vector 0 2.
It then runs to completion, freeing all its resources, making the availibility vector 2 7.
Process 3 uses 1 of resource 1, 7 of resource 2, making the availability vector 1 0.
It then runs to completion, freeing all its resources, making the availability vector 3 9.
The only processes left are P2 and P4 – the system is deadlocked because we don't have enough of resource 1 – P2 wants 4, and P4 wants 5 – but we only have 3.

If the availability vector is as below, is the system above still deadlocked?

2	3

Is the system deadlocked if the availability is

2	4

31. Assuming the operating system detects the system is deadlocked, what can the operating system do to recover from deadlock?

The operating system can recover from deadlock by:
- **Killing the deadlocked process** (crude, and requires that it will be restarted)
- **Rolling Back** (going back a few instructions until it can enter a non-deadlocked state, and retry)
- **Pre-emption** (forcing the handing over of a resource to another process, at the expense of the one it's currently deadlocked on)

32. What must the *banker's algorithm* know a priori in order to prevent deadlock?

The **banker's algorithm** must calculate whether giving a process a resource will lead to a *safe state* or not. In order to work, it must know how many resources the process in question may be granted at any one time, and how many resources the OS is able to give. This is difficult because how can we know what future requests might be?

To see if a state is safe – the algorithm checks to see if enough resources exist to satisfy a process. If so, the “loans” are assumed to be “repaid”.

33. Describe the general strategy behind *deadlock prevention*, and give an example of a practical deadlock prevention method.

Deadlock prevention is theorised on the basis that if one nullifies one of the four conditions for deadlock to occur (mutual exclusion, hold and wait, circular wait and no preemption), a deadlock cannot occur. Attacking mutual exclusion and no preemption has no practical basis, so we commonly prevent the circular wait condition. We do this by **globally numbering all resources**. (e.g. Blu-Ray drive #1 and USB Hard Drive #2). At every instant – one of the processes will have the highest numbered resource. The process holding it will never ask for a lower one, because we only allow processes to access higher numbered resources. Eventually it will finish and free all its resources. All processes can finish.

34. Filesystems can support *sparse files*, what does this mean? Give an example of an application's file organisation that might benefit from a file system's sparse file support.

A sparse file is a file that has a start, and end, but no middle. The bytes in between the starting set and the ending set are simply unused. An application that may benefit from sparse file support might be a virtualisation platform, in which it presents the user with an “empty” virtual hard drive. Another is a partially completed file accessed via the BitTorrent Protocol.

35. Give an example of a scenario that might benefit from a file system supporting an append-only access write.

An application where an append-only access write would be useful could be a “drop box” to hand in finished assignments where no further adjustments can be made. Another could be a very rudimentary version control system in which every single revision of the file is stored in full. A third could be a log file, where append-only prevents editing existing log entries.

36. Give a scenario where choosing a large filesystem block size might be a benefit; give an example where it might be a hinderance.

Large file system block sizes are a help (where performance is concerned) when the files in question are very large, and contiguous reading/writing is prevalent. For example, multimedia files. A large file system block size, unfortunately, wastes a lot of space if your file system largely consists of small sized files. Random access to small segments of data require loading entire blocks of data, even though you only need a small amount of it.

37. Give an example where contiguous allocation of file blocks on disks can be used in practice.

Contiguous allocation of file blocks on disks can be used when you're writing a memory dump (of your known 2GB or so of memory) because your OS crashed ;) Jesting aside, contiguous allocation is painful because it is necessary to know the file's final size prior to even allocating any space. It is still used on write-once optical media, because prior to burning your CD, DVD or BD the system knows exactly how much space each file uses on said disk.

38. What file access pattern is particularly suited to chained file allocation on disk?

Chained, or 'Linked List' allocation, like contiguous allocation, is useful for when dealing with large, sequential files. The first word of each block is used to **point** to the next one in sequence. It is fine for sequential access because every block needs to be read regardless. It's a living nightmare for random access files because we introduce a lot of unnecessary, wasted “read” operations to the FS.

39. What file allocation strategy is most appropriate for random access files?

An inode (indexed node) based file allocation strategy. It is a form of file allocation table that offers temporal locality – the inode only needs to be located in memory when its corresponding file is open. The inode is a data structure which lists all attributes and points to all the addresses of the disk blocks corresponding to that file.

40. Compare bitmap-based allocation of blocks on disk with a free block list.

The two methods of managing free space on the inode-based system are:
- Linked List of free blocks – the pointers are stored in the free blocks themselves, only a block of pointers needs to be kept in main memory. Advantageous in that it gets smaller as the disk is used up.
- Bitmap Allocation – individual bits in a bit vector flags used and free blocks, but is large and of a fixed size as it corresponds to the entire disk (perhaps too large to hold in main memory) and expensive to search. However, it's simple to find contiguous free space.

41. How can the block count in an inode differ from the (file size / block size) rounded up to the nearest integer. Can the block count be greater, smaller, or both.

Block count frequently differs from file size. File size is the offset of the highest byte written, and is what you commonly notice in files that you deal with everyday. Block count is a count of the number of physical disk blocks that are used by the file. In the example of a sparsely populated file, such as a BitTorrent download of a 700 MB Linux ISO – it uses up '700 MB' on your system at the beginning, long before the actual 700 MB of the file's contents are actually written to your disk blocks. However, most files are non sparse and disk block usage should be equal to file size. In reality, disk block usage is actually a little higher, as there are additional blocks (metadata, etc) not considered in the file size. Therefore disk block count can differ in any way from the file size, or even be equal with the right trade-off between sparsity and additional block usage.

42. Why might the *direct blocks* be stored in the inode itself?

To adhere to the principle of spatial locality – we can avoid extra disk seek operations simply by reading/writing to the first “spare” 12 blocks of the inode – there's plenty of room in it, and it makes the inode space not a complete waste of disk blocks.

43. Given that the maximum file size of combination of direct, single indirection, double indirection, and triple indirection in an inode-based filesystem is approximately the same as a filesystem solely using triple indirection, why not simply use only triple indirection to locate all file blocks?

Did you not read the answer to Question 42?! Spatial locality! For smaller files, it keeps the amount of disk accesses (which are expensive, given that there is a mechanical component otherwise known as the disk head) low to read any block in the file. The amount of disk accesses grows with the levels of indirection we have – a direct block only requires one operation to read, a single indirection requires 2 operations (one for the indirect block, one of the block it's pointing to), and so on.

44. What is the maximum file size supported by a file system with 16 direct blocks, single, double, and triple indirection? The block size is 512 bytes. Disk block numbers can be stored in 4 bytes.

Number of blocks:
- Direct Blocks = 16 blocks
- Single Indirect Blocks = 512 / 4 = 128 blocks
- Double Indirect Blocks = 128 * 128 = 16384 blocks
- Triple Indirect Blocks = 128 * 128 * 128 = 2097152 blocks

Total number of blocks = direct + single + double + triple = 16 + 128 + 16384 + 2097152 = 2113680
Total number of bytes = 2113680 * 512 = 1.08220416 E 9 = 1.08 GB

45. The Berkeley Fast Filesystem (and Linux Ext2fs) use the idea of block groups. Describe what this idea is and what improvements block groups have over the simple filesystem layout

of the System V file system (s5fs).

The System V file system contained four blocks – the **boot block, super block, inode array** and **data blocks.** This was inefficient, because seek times would be massive – inodes are at the start of the disk and the actual data could be anywhere from the start to the end! There was also only a single super block (block containing attributes of the entire filesystem). If this was corrupted, it's bye-bye file system. The Berkeley Fast Filesystem (and ext2) extended the System V filesystem by creating block groups – all equally sized and each somewhat replicating the System V structure (aside from boot block). The inode array was split into **group descriptors, data block bitmap, inode bitmap and inode table**. This solves the major problems with s5fs, as proximity of inode tables and data blocks is spatial locality-friendly, and you can no longer corrupt the entire filesystem by way of superblock.

46. What is the reference count field in the inode? You should consider its relationship to directory entries in you answer.

The reference count field is a counter of how many times the inode is “referenced” by name. Adding a directory entry increments this counter. When the count falls to zero, (i.e. there are no longer any directory entries to that file), its inode and all corresponding disk blocks can be safely deallocated.

47. The filesystem *buffer cache* does both buffering and caching. Describe why buffering is needed. Describe how buffering can improve performance (potentially to the detriment of file system robustness). Describe how the caching component of the buffer cache improves performance.

Buffering is required in order to ensure that performance is not degraded as a file is sequentially read or written to, perhaps from different parts of the physical disk, since a large number of the disk reads occur before the file is actually “read” by the user (or in the case of writes, the data is flushed to disk long after individual “writes” occur). This minimises expensive disk I/O operations, because I/O occurs in sequential bursts, but possibly at the detriment of robustness – for example if a power failure occurs, the buffer cache is lost, and its contents were not written to disk. The caching component improves performance still, by storing frequently used parts of files in fast memory, minimising lengthy disk access times.

48. What does *flushd* do on a UNIX system?

Flushd forces a write of the contents of the buffer cache to the hard disk every 30 seconds, avoiding data loss on an unexpected OS termination.

49. Why might filesystems managing external storage devices do *write-through* caching (avoid buffering writes) even though there is a detrimental affect on performance.

Write-through caching is necessary on external drives in order to maintain reliability and avoid data loss in situations where the drive controller is compromised through an event (a kernel panic, power failure, or most commonly – simply being unplugged) where the buffer cache is lost. It is always much safer to have critical data written to physical disk blocks, despite the high cost of disk I/O operations.

50. Describe the difference between *external* and *internal* fragmentation. Indicate which of the two are most likely to be an issues on a) a simple memory memory mangement machine using base limit registers and static partitioning, and b) a similar machine using dynamic partitioning.

External fragmentation refers to fragmentation (any unused space inside the partition is wasted) outside of individual partitions, while internal fragmentation refers to that of the inside of partitions. In situation a, internal fragmentation is likely to be an issue, and in situation b, external fragmentation is likely to be an issue.

51. List and describe the four memory allocation algorithms covered in lectures. Which two of the four are more commonly used in practice?

The four memory allocation algorithms (in the scheme of dynamic partitioning placement) are:
First-Fit – in the linked list of available memory addresses, we place the data in the first entry that will fit its data. Its aim is to minimise the amount of searching, but leads to external fragmentation later on.
Next-Fit – similar to first fit, but instead of searching from the beginning each time, it searches from the last successful allocation. Greatly reduces the amount of searching but leaves external fragmentation at the beginning of memory.
Worst-Fit – traverses the memory and gives the partitions as large spaces as possible – to leave usable fragments left over. Needs to search the complete list and such is a poor performer.
Best-Fit – carefully scours the memory for spaces that perfectly fit the RAM we want. However, the search is likely to take a very long time.

We most commonly use first-fit and next-fit in practise. They're easier to implement and are faster to boot.

52. Base-limit MMUs can support swapping. What is *swapping*? Can swapping permit an application requiring 16M memory to run on a machine with 8M of RAM?

Swapping is the process of saving memory contents belonging to a particular process to a backing store (most commonly a fast and large disk) temporarily and reloading it for continued execution. Unfortunately, it does not permit an application requiring 16MiB of memory to run on a system with less RAM, as the program requires the full contents of its memory to be in the physical memory at any one time.

53. Describe *page-based virtual memory*. You should consider *pages, frames, page tables,* and *Memory Management Units* in your answer.

Page-based virtual memory is the idea that physical memory space can be divided into **frames**, while each process owns a virtual address space divided into **pages** which are the same size as frames. The **page table** contains the location of a frame corresponding to a page, which is generally obfuscated using some algorithm (commonly two-level and inverting).

54. Give some advantages of a system with page-based virtual memory compared to a simply system with base-limit registers that implements swapping.

Page based virtual memory ensures that every process has its own address space. It allows processes access up to 2^(system bits) bytes of address space, and allows programs requiring more RAM to use only what it needs to at any one point. Physical memory need not be contiguous. No external fragmentation. Minimal internal fragmentation. Allows sharing (you can map several pages to the same frame).

55. Describe *segmentation-based virtual memory.* You should consider the components of a memory address, the segment table and its contents, and how the final physical address is formed in your answer.

Segmentation-based virtual memory is a scheme supporting the user view of memory, that is, a program is divided into segments such as main(), function(), stack etc. A segmented memory address is distinguished by its segment number and offset. The segment table has a base (starting PADDR where a segment resides) and limit (length of the segment) column – physical addresses are formed by coalescing base and limit registers.

56. What is a *translation look-aside buffer?* What is contained in each entry it contains?

A translation look-aside buffer (TLB) is a high speed cache for page table entries (that can either be virtual or part of the physical MMU) containing those that are recently used. It consists of two different entries, EntryLo and EntryHi. EntryHi contains a virtual address and sometimes ASID. EntryLo contains a corresponding physical address and a number of bits signifying whether the address is 'dirty', 'empty' or otherwise.

57. Some TLBs support *address space identifiers* (ASIDS), why?

Multitasking operating systems would better perform when the physical memory addresses loaded from the TLB are identifiable, as they are process specific. On a context switch, one needs to flush the TLB (invalidate all entries), which is an expensive operation, and on a system that is frequently context switching could provide so much of a performance issue that the TLB almost becomes useless.

58. Describe a two-level page table? How does it compare to a simple page table array?

A two level page table is a form of nested page table structure. It works in terms of bases and offsets, where portions can be contatenated to form the original physical addresses. Second level page tables representing unmapped pages are left unallocated, saving a bit of memory.

59. What is an inverted page table? How does it compare to a two-level page table?

It uses hashing to achieve being an array of page numbers indexed by physical frame number. It grows with size of RAM, rather than virtual address space, saving a vast amount of space.

60. What are *temporal locality* and *spatial locality*?

Temporal locality is the concept (most commonly referred to in the subject of caching), that if you access a piece of data at any one time, you are likely to want to access that same piece again soon. Spatial locality, conversely, is the concept that if you access a given piece of data, you are likely to want to access its neighbouring data.

61. What is the *working set* of a process?

The working set of a process is the allocated pages/segments at any one time window (delta), consisting of all pages accessed during that time. Includes current top of stack, areas of the heap, current code segment and shared libraries.

62. How does page size of a particular achitecture affect working set size?

Page size of a particular architecture affects working set size because the larger the pages, the more memory that can be wasted on irrelevant data, vastly increasing the working set size for no good reason. If they're smaller, the pages accurately reflect current memory usage.

63. What is *thrashing?* How might it be detected? How might one recover from it once

detected?

Thrashing is the phenomenon that occurs when the total sum of all working set sizes becomes greater than the available physical memory. Productivity drops since the number of instructions that is able to be sent to the CPU drops. It could be detected if a threshold value was put in place for each working set size, and recovery is as simple as suspending processes until total working set size decreases.

64. Enumerate some pros and cons for increasing the page size.

Pros:
- Reduce total page table size, freeing some memory (hey, it uses up memory too!)
- Increase TLB coverage
- Increase swapping I/O throughput

Cons:
- Increase page fault latency (more page to search through)
- Increase internal fragmentation of pages

65. Describe two virtual memory page fetch policies. Which is less common in practice? Why?

Demand Paging – relevant pages are loaded as page faults occur
Pre Paging – try to load pages for processes before they are accessed

Prepaging is less common in practise because it wastes bandwidth if it gets it wrong (and it will be wrong) and wastes even more if it kicks known good pages because it's trying to stuff it full of bad ones!

66. What operating system event might we observe and use as input to an algorithm that decides how many frames an application receives (i.e. an algorithm that determines the application's resident set size)?

This was not covered in the 2013 course.

67. Name and describe four page replacement algorithms. Critically compare them with each other.

OPTIMAL: Impossible to achieve, but perfect. Works on the Nostradamus-like basis that we can predict which page won't be used for the longest time, and elect that that one should be replaced.
LEAST RECENTLY USED: Most commonly used, not quite optimal but better than the rest. We mark every page with a timestamp and the one which has been least recently accessed gets the flick. However we need to account for the extra overhead of adding/reading the time stamp, and who knows, the next page we want to access might just be the one we've sent back to the aether. It's quite a performer though.
CLOCK: Each page is marked with a 'usage' bit, and each is given a 'second chance' upon page replacement time. The one that becomes unmarked first disappears. Not quite as accurate as LRU, but has less of an overhead (one extra bit, as opposed to many for the timestamp).
FIFO: The simple and dodgiest option, we make the foolish assumption that the oldest page in the queue is the one that gets tossed. This is not quite always the case.

68. Describe *buffering* in the I/O subsystem of an operating system. Give reasons why it is required, and give a case where it is an advantage, and a case where it is a disadvantage.

Buffering is the action I/O devices take in order to efficiently transfer data to or from the operating system. Instead of directly transferring data bit-by-bit (literally) via system calls and registers, which is slow and requires context switch after context switch, we fill a buffer (or two, or three) with data, which is periodically transferred to or from main program control. We can also perform double or triple buffering where two-way efficiency is gained. It can be advantageous when we need to transfer large volumes of data, and at the same time may be disadvantageous due to overheads – if the network is high-speed the time to copy between buffers might be comparable to the time spent actually transferring the data!

69. Device controllers are generally becoming more complex in the functionality they provide (e.g. think about the difference between implementing a serial port with a flip-flop controlled by the CPU and a multi-gigabit network adapter with the TCP/IP stack on the card itself). What effect might this have on the operating system and system performance?

Device controllers' improved functionality can at times really increase the performance of the overall system as we are offloading tasks from the CPU (think of the difference between running 3D games with and without a dedicated GPU). However we may end up with the problem of lack of hardware support, or could be bounded by slow interfaces.

70. Compare I/O based on *polling* with *interrupt-driven* I/O. In what situation would you favour one technique over the other?

Polling means that we are continually using CPU cycles to check whether any I/O is occurring or not. However, it means the CPU is busy-waiting on any other task so as to not miss any I/O. Interrupt-driven I/O allows the CPU to work on other tasks and handle requests on demand, however it requires a context switch into the interrupt handler to process it. We normally favour interrupt-driven I/O for any sort of human interface device, as it is much slower than the data processing it is doing, so a few hundred interrupts each second won't matter. We might favour polling for high bandwidth raw data transfer applications, as a million context switches tends to slow things down.

71. Explain how the producer-consumer problem is relevant to operating system I/O.

The action of performing buffering on inputs and outputs is a real-life manifestation of the producer-consumer problem. The input/output to/from a buffer acts as the producer, while the OS or device receiving the data becomes the consumer, bounded by the limited capacity of the buffer.

72. What is disk *interleaving*? What problem is it trying to solve?

Disk interleaving is the action of putting sequentially accessed data into non-sequential sectors. It was used to adjust timing difference between receiving and reading data. Because buffer storage has become more than sufficient in recent times, interleaving has fallen out of use.

73. What is cylinder skew? What problem is it trying to solve?

Not in the 2013 COMP3231 course.

74. Name four disk-arm scheduling algorithms. Outline the basic algorithm for each.

FIFO: It processes requests as they come. If there are too many processes, it deteriorates to "random". However, it avoids starvation.

Shortest Seek Time First: Out of a list of requests, pick the ones that minimise seek time. Performance is excellent, but is susceptible to starvation.

Elevator (SCAN): Out of a list of requests, we move the head in one direction and back again. It services requests in track order until reaching the highest, then reverses. Performance is good, not quite as good as SSTF, but avoids starvation.

Modified Elevator (Circular-SCAN): Similar to elevator, but reads sectors in only one direction. It does not down-scan, instead electing to go back to the first track and start again. This gives it better locality on sequential reads and reduces maximum delay to read a particular sector.

75. Why is it generally correct to favour I/O bound processes over CPU-bound processes?

We favour I/O bound processes over CPU bound processes because they are generally many, many, orders of magnitude slower, so delaying CPU bound processes to take care of a tiny bit of I/O is nary a problem. However, choosing to run a CPU bound process prior to an I/O one delays the next I/O request hugely.

76. What is the difference between preemptive scheduling and non-preemptive scheduling? What is the issue with the latter?

Pre-emptive scheduling is based on timer interrupts, where a running thread may be interrupted by the OS and switched to the **ready** state at will (usually if something more important comes through) or when it has exceeded its timing allocation. Non-preemptive scheduling means once a thread is in the running state, it *continues until it completes,* or at least gives up the CPU voluntarily. Threads that do this are likely to monopolise the CPU.

77. Describe *round robin scheduling.* What is the parameter associated with the scheduler? What is the issue in chosing the parameter?

Round-robin scheduling works by giving each process a “timeslice” to run in, implemented by a ready queue and a regular timer interrupt. When a timeslice expires, the next process pre-empts the current process and runs for its timeslice, putting the pre-empted process at the end of the queue. The parameter concerned with it is the timeslice, which has to be exactly the right size. If it was too short, there is a large overhead every time it expires and the context switches, if it was too long, then we might end up with an unresponsive system.

78. The traditional UNIX scheduler is a priority-based round robin scheduler (also called a multi-level round robin schduler). How does the scheduler go about favouring I/O bound jobs over long-running CPU-bound jobs?

The traditional UNIX scheduler assigns each process a priority and places them in multiple ready queues. Priorities increase over time to prevent saturation of low priority processes, boosted based on the amount (as in lack thereof) of CPU time consumed. I/O bound jobs are favoured, naturally, as they consume very little CPU time.

79. In a real-time system with a periodic task set,, how are priorities assigned to each of the periodic tasks?

There are two types of real-time scheduling – rate monotonic and earliest deadline first. In rate monotonic, priorities are assigned based on the period of each task, and in earliest deadline first, priorities are dynamically assigned based on the deadlines of each task.

80. What is an *EDF* scheduler? What is its advantage over a rate monotic scheduler?

EDF scheduler = Earliest Deadline First scheduling. Rate monotonic scheduling, comparatively, assigns priorities based on the period of each task. However, it only works if CPU utilisation is not too high. EDF is more difficult to implement, however it always works so long as the tasks are actually possible to schedule!