

Contents

1. Answers to Sample Paper.
 1. Q1
 2. Q2
 3. Q3
 4. Q5
2. More Sample Questions
 1. Describe the two general roles of an operating system, and elaborate why these roles are important.
 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.
 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?
 4. Is putting security checks in the C library a good or a bad idea? Why?
 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.
 6. Multi-programming (or multi-tasking) enables more than a single process to apparently execute simultaneously. How is this achieved on a uniprocoessor?
 7. What is a process? What are attributes of a process?
 8. What is the function of the ready queue?
 9. What is the relationship between threads and processes?
 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.
 11. Name some advantages and disadvantages of user-level threads.
 12. Why are user-level threads packages generally cooperatively scheduled?

13. List the advantages and disadvantages of supporting multi-threaded applications with kernel-level threads.
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.
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?
16. What is a race condition? Give an example.
17. What is a critical region? How do they relate to controlling access to shared resources?
18. What are three requirements of any solution to the critical sections problem? Why are the requirements needed? from lecture notes:
19. Why is turn passing a poor solution to the critical sections problem?
20. Interrupt disabling and enabling is a common approach to implementing mutual exclusion, what are its advantages and disadvantages?
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.
22. What is the producer consumer problem? Give an example of its occurence in operating systems.
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.
24. Describe how to implement a lock using semaphores.
25. What are monitors and condition variables?
26. What is deadlock? What is startvation? How do they differ from each other?
27. What are the four condtions required for deadlock to occur?
28. Describe four general strategies for

28. Describe four general strategies for dealing with deadlocks.
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.
30. Is the following system of four processes with 2 resources deadlocked?
31. Assuming the operating system detects the system is deadlocked, what can the operating system do to recover from deadlock?
32. What must the banker's algorithm know a priori in order to prevent deadlock?
33. Describe the general strategy behind dealock prevention, and give an example of a practical deadlock prevention method.
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.
35. Give an example of a scenario that might benefit from a file system supporting an append-only access write.
36. Give a scenario where choosing a large filesystem block size might be a benefit; give an example where it might be a hinderance.
37. Give an example where contiguous allocation of file blocks on disks can be used in practice.
38. What file access pattern is particularly suited to chained file allocation on disk?
39. What file allocation strategy is most appropriate for random access files?
40. Compare bitmap-based allocation of blocks on disk with a free block list.
41. Why might the direct blocks be stored in the inode itself?
42. 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 soley using triple indirection, why not simply use only triple indirection to locate

all file blocks?

43. 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.
44. 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).
45. What is the reference count field in the inode? You should consider its relationship to directory entries in your answer.
46. 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.
47. What does flushd do on a UNIX system?
48. Why might filesystems managing external storage devices do write-through caching (avoid buffering writes) even though there is a detrimental effect on performance.
49. Describe the difference between external and internal fragmentation. Indicate which of the two are most likely to be an issue on a) a simple memory management machine using base limit registers and static partitioning, and b) a similar machine using dynamic partitioning.
50. List and describe the four memory allocation algorithms covered in lectures. Which two of the four are more commonly used in practice?
51. 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?
52. Describe page-based virtual memory. You should consider pages

- memory. You should consider pages, frames, page tables, and Memory Management Units in your answer.
53. Give some advantages of a system with page-based virtual memory compared to a simply system with base-limit registers that implements swapping.
 54. 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.
 55. What is a translation look-aside buffer? What is contained in each entry it contains?
 56. Some TLBs support address space identifiers (ASIDS), why?
 57. Describe a two-level page table? How does it compare to a simple page table array?
 58. What is an inverted page table? How does it compare to a two-level page table?
 59. What are temporal locality and spatial locality?
 60. What is the working set of a process?
 61. How does page size of a particular achitecture affect working set size?
 62. What is thrashing? How might it be detected? How might one recover from it once detected?
 63. Enumerate some pros and cons for increasing the page size.
 64. Describe two virtual memory page fetch policies. Which is less common in practice? Why?
 65. 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)?
 66. Name and describe four page replacement algorithms. Critically compare them with each other.
 67. 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.
 68. 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?

69. Compare I/O based on polling with interrupt-driven I/O. In what situation would you favour one technique over the other?
70. Explain how the producer-consumer problem is relevant to operating system I/O.
71. What is disk interleaving? What problem is it trying to solve?
72. What is cylinder skew? What problem is it trying to solve?
73. Name four disk-arm scheduling algorithms. Outline the basic algorithm for each.
74. What is the difference between preemptive scheduling and non-preemptive scheduling? What is the issue with the latter?
75. Describe round robin scheduling. What is the parameter associated with the scheduler? What is the issue in choosing the parameter?
76. The traditional UNIX scheduler is a priority-based round robin scheduler (also called a multi-level round robin scheduler). How does the scheduler go about favouring I/O bound jobs over long-running CPU-bound jobs?
77. In a real-time system with a periodic task set, how are priorities assigned to each of the periodic tasks?
78. Describe why the use of spinlocks might be more appropriate than blocking locks on a multiprocessor when compared to a uniprocessor. Is there a trade-off between spinning and blocking on a multiprocessor? Discuss.
79. Is a single ready queue on a multiprocessor a good idea? Why?
80. What is thread affinity. Why might it improve performance?

Note: These answers were provided by student posts to the forum in past years, or by the lecturer. They are intended as a guide to the correct answers. Many thanks to those who have contributed.

For the questions without answers, post your attempts and I will correct them.

Good luck in the exam.

Answers to Sample Paper.

Q1

- A smaller page size leads to smaller page tables – False – need more entries because we have more pages
- A smaller page size leads to more TLB misses – True – less likely that page will encompass address we are after.
- A smaller page size leads to fewer page faults – True

In practice, initial faulting-in is usually less important than page faults generated once the application is up and running.

Once the app is up and running, reducing the page size results in a more accurate determination of work set, which is in-turn smaller, which requires less memory, which in-turn leads to a lower page fault rate.

- A smaller page size reduces paging I/O throughput – True
- Threads are cheaper to create than processes – True
- Kernel-scheduled threads are cheaper to create than user-level threads – False
- A blocking kernel-scheduled thread blocks all threads in the process – False. This is true for user level threads
- Threads are cheaper to context switch than processes – True – don't have to save the address space
- A blocking user-level thread blocks the process - True
- Different user-level threads of the same process can have different scheduling priorities – False. User level threads aren't scheduled, they yield() the CPU
- All kernel-scheduled threads of a process share the same virtual address space – True
- The optimal page replacement algorithm is the best choice in practice – False - it's impossible to implement
- The operating system is not responsible for resource allocation between competing processes – False – it is responsible for this

- System calls do not change to privilege mode of the processor – False – we trap into the kernel so we do change the privilege mode
- A scheduler favouring I/O-bound processes usually does not significantly delay the completion of CPU-bound processes – True – the I/O processes get the I/O and then yield the CPU again.

Q2

a)

99 TLB hits, which needs only a single memory access (the actual one required)

.0095 TLB miss, one read from memory to get the page table entry to load TLB, then one read to read actual memory.

.0005 pagefault plus eventual read from memory plus something like one of the following (I'd take a few variation here)

1) A memory reference to update the page table 2) A memory reference to update the page table and a memory reference for the hardware to re-read the same entry to refill the TLB 3) Any reasonable scebario that results in valid page table and refilled TLB.

I'll assume option 2 for final answer

thus

$.99 * 100\text{ns} + .0095 * 2 * 100\text{ns} + .0005 (3 * 100\text{ns} + 5\text{ms})$

would be one answer I would accept.

Q3

a)

i) FCFS order traversed: 119,58,114,28,111,55,103,30,75 tracks traversed: 547 ii) SSTF order traversed: 75,58,55,30,28,103,111,114,119 tracks traversed: 143 iii) SCAN order traversed: 103,111,114,119,75,58,55,30,28 tracks traversed: 130 iv) C-SCAN order traversed: 103,111,114,119,28,30,55,58,75 tracks traversed: 177

Q5

b)

It is in the multi process lecture notes. 🌐

<http://cgi.cse.unsw.edu.au/~cs3231/06s1/lectures/lect21.pdf>

Starting at page 30.

Just some notes that may be good to point out

Test-and-Set Hardware locks the bus during the TSL instruction to prevent memory accesses by any other CPU Issue: Spinning on a lock requires bus locking which slows all other CPUs down

- Independent of whether other CPUs need a lock or not
- Causes bus contention

Caching does not help this test and set.

Hence to reduce bus contention.

- Read before TSL
 - Spin reading the lock variable waiting for it to change
 - When it does, use TSL to acquire the lock
- Allows lock to be shared read-only in all caches until its released
 - no bus traffic until actual release
- No race conditions, as acquisition is still with TSL.

More Sample Questions

These are sample questions about topics taught in the course. The answers are from students, which have mostly been vetted by me. They may also be fresh changes, so you should look at them critically yourselves. Part of mastery of the material is being able to tell a good or correct answer from a bad or incorrect answer.

Note that you can correct/clarify/modify/provide answers yourself.

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

Operating systems provide: - a fairly standardised abstraction between hardware and userspace software, that allows userspace software to be written with little concern for the hardware it is running on, and having access to a bunch of useful things like memory, files, screens, i/o etc. - resource management and allocation to processes & threads. resources


include files, memory, cpu runtime, etc.

i) high-level abstractions why? - more friendly to programmers - provides common core for all applications - hides details of hardware to make application code portable

ii) resource manager why? - divides resources amongst competing programs or users according to some system policy to ensure fairness and no starvation where possible.

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.

The userspace program makes a function call to the C library, e.g. 'uptime', which sets up a system call in the appropriate manner for the hardware. The userspace program must set up parameters appropriately then trap, such that the kernel takes control and processes the request in userspace. In MIPS, this is achieved through the syscall instruction. Once the kernel has control, the user state is saved. The kernel performs necessary security & sanity checks, then attempts to fulfill the request. In the case of 'uptime', it would copy the value of a counter, which (presumably) has been counting seconds since powerup. The user state is restored, the counter is placed this in the stack or a register and warps back to userspace. The C library routine reads the stack / register that the kernel just wrote, and returns it to the userspace program.

 <http://cgi.cse.unsw.edu.au/~cs3231/07s1/lectures/lect03.pdf> Its all about system calls, and has the required info. Page 63 has a nice diagram of the producers.

The steps may be:

- Set up the required syscall arguments
- Call syscall execution
- Change to kernel stack
- Preserve registers by saving to memory (the stack)
- Leave saved registers somewhere accessible to
 - Read arguments
 - Store return values
- Do the "read()"

- Restore registers
- Switch back to user stack
- Return to application

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 needs to verify that the memory being written to / read from is a valid address, owned by the process making the request. It must prevent processes from corrupting other processes, or reading memory owned by other processes. The operating system must take care with inputs from memory as:

- a) The address given by the application as the address of the input or output could be - an address of an illegal area of memory - an address of a legal area that is paged to disk
- b) or, could change if the application is multithreaded.

Basically, the operating system must ensure that it cannot be corrupted, crashed, or bypassed as a result of accessing memory as directed by the application.

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

Putting security checks in the C library is no guarantee of security, because it is the choice/convenience of the userspace program to use the C library. There is nothing to stop a userspace program making a system call directly, with whatever parameters it pleases, which are potentially malicious. Security checks should be performed in kernel space, where the userspace program has no influence, and cannot circumvent security checks.

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.

Processes can be Running, Ready, or Blocked. A running process may block on an IO request. A running process may also be placed in a ready queue if it cooperatively or preemptively gets context switched, to allow another program to run. Blocked processes can be moved to a ready queue when the thing it was blocking on becomes available. Ready processes can start running in a context switch.

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

Preemptive multitasking is achieved by rapidly context switching between processes. Each process is given a window of time to execute, called a 'timeslice'. The processor then gets interrupted by a timer, which causes it to context switch to another process, which will run until it blocks, or until the next tick. By switching rapidly enough, it creates the illusion that several processes are running simultaneously.

Cooperative multitasking is achieved by having every process yield occasionally, such that other processes can run.

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

A process is one or more threads of execution that exist in a single virtual memory space. A process has its own set of file descriptors, vm space, timekeeping data, pwd, (maybe some other stuff)

1. a program in execution 2. an instance of running program 3. an owner of the resource allocated to program execution 4. a task or a job 5. encompass one or more threads

attribute of a process: Address Space Open files - (could be argued are part of resources) Process ID Process Group Parent Process signals etc.

attributes of threads: Scheduling Parameters Registers Program Counter Program Status Word Stack Pointer Thread state

If the OS does not support multi-threaded applications, then thread attributes become process attributes.

What is the function of the ready queue?

The ready queue stores threads that aren't currently running, that are capable of resuming execution. There may be several ready queues for each priority level, depending on the scheduling algorithm. The scheduler consults the ready queue to determine which process/thread to run next. As the name suggests, the ready queue is a queue, in order to schedule fairly.

What is the relationship between threads and processes?

A process is a container for threads, which has its own memory space. A process may contain one or more threads, which share that memory space, all of the file descriptors and other attributes. The threads are the units of execution within the process, they possess a register set, stack, program counter, and scheduling attributes - per thread.

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.

From the kernel's perspective, each process has its own address space, file descriptors, etc and one thread of execution. To support multiple threads at user-level, the process contains code to create, destroy, schedule and synchronise user-level threads - which can be thought of as multiplexing many user-level threads onto the single kernel thread, all managed within the process. The scheduler can run any arbitrary scheduling algorithms, and is independent of the kernel's scheduler.

User-level threads can also satisfy a huge number of threads if necessary, as it can take advantage of virtual memory to store user-level thread control blocks and stacks.

These are often cooperative schedulers, as there is usually only rudimentary support (if any) for delivering timer ticks to the user-level scheduler. However the application must be specifically written for cooperative scheduling, containing yields such that other threads have an opportunity to run; a single badly written thread without enough yields

can monopolise cpu time. Alternatively, these can also be preemptive multitaskers, where the user level thread package receives regular signals from the operating system, which may initiate the context switch. These have high granularity though and aren't really practical.

Another important consideration is making system calls non blocking, so that the user-level thread scheduler can schedule (overlap) execution of another thread with I/O latency rather than blocking the whole process. This is not always possible (either because of lack of async I/O system calls), or because events such as page faults (which are always synchronous) will block the process and all the user-level threads in side.

Name some advantages and disadvantages of user-level threads.

Advantages: User level threads are more configurable, as they may use any scheduling algorithm, and they can also run on any OS (probably even DOS) since they do not require kernel support for multiple threads. User level threading is also much faster, as they don't need to trap to kernel space and back again. (hmmm... prescott?)

Disadvantages: without kernel thread support, the threads must be cooperative. Yields are annoying, and require every thread to be well written, e.g. having enough yields and not monopolising cpu time. The I/O must be non blocking, then require extra checking to deal with cases that would normally block. This is not always possible (either because of lack of async I/O system calls, or polling system calls), or because events such as page faults (which are always synchronous) will block the process and all the user-level threads in side.. User level threads cannot take advantage of multiple cpus, as the os cannot dispatch threads to different processors, or potentially do true parallelism, because the OS only sees 1 process with 1 thread.

Why are user-level threads packages generally cooperatively scheduled?

User level threads packages are generally cooperatively scheduled as is usually only rudimentary support (if any) for delivering timer ticks to the user-level scheduler. They can be made preemptively if the operating system regularly signals the process, however this is high granularity, high enough that it isn't particularly useful, so cooperative is preferential to give the illusion of parallelism.

List the advantages and disadvantages of supporting multi-threaded applications with kernel-level threads.

Advantages

1. assuming that the user-threads package is cooperative, user programs do not need to be written with yields everywhere
2. each thread is guaranteed a fair amount of execution time (as fair as the os is)
3. userspace programs do not require extra checking around blocking i/o to avoid it - i.e. other ready threads within the process can be scheduled if the running thread blocks on a system call or page fault.
4. low granularity multitasking, gives the illusion of parallelism
5. can take advantage of multiprocessors, as the kernel can dispatch threads to other cpus as it sees appropriate

Disadvantages

1. the user program must use locks and mutexes appropriately around critical sections, since it cannot control when context switching will occur (...but so what)
2. code is less portable since this requires operating system support
3. cannot choose the scheduling algorithm
4. Thread management (create, delete, ect) is more expensive as it now requires system calls

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

- Thread 1 running happily - Timer tick - Trap into kernel space - switch to kernel stack for thread 1 - thread 1 user registers stored - find next thread to run (named thread 2) from scheduler - context switch to thread 2 (change to its kernel stack, make its thread control current, flush tlb) - thread 2 user registers loaded - warp into thread 2 - thread 2 running

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?

For a real example of such a function look at the OS/161 mips_switch code

In general, assembly language context switch code does:

- Saves enough registers on the stack to eventually return back to the calling C function
- stores the stack pointer in the current thread's control block
- load the stack pointer from the destinations thread's control block
- restores the destination threads registers from its stack
- returns to 'C' in the destination thread.

What is a race condition? Give an example.

A race condition occurs when there is an uncoordinated concurrent access to shared resources (e.g. memory or device registers), which leads to incorrect behaviour of the program, deadlocks or lost work.

An example would be two process updating a counter simultaneously. Say the counter has a value of 1. Process A reads the variable but before updating Process B reads and updates the counter to 2. Process A, having no knowledge of what Process B does, updates the counter to 2. The correct behaviour of the program would have the counter being incremented to 3, not 2.

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

A critical region is an area of code where the code expects shared resources not to be changed or viewed by other threads while executing inside the critical region.

What are three requirements of any solution to the critical sections problem? Why are the requirements needed? from lecture notes:

Mutual Exclusion: No two processes simultaneously in critical region

Progress: no process outside of a critical region may cause another process to block Bounded: No process must wait forever to enter its critical region

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

need to wait for whatever thread whose turn it is to actually use that section, which it might not be about to use. This can cause the thread waiting on the critical section to block unnecessarily long. if the different threads have different patterns of use of the critical region, performance is limited by the most infrequent user., i.e. the system progresses at the rate of the slowest thread.

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

Advantages: It's easy and efficient on a uniprocessor. Disadvantages: it doesn't work on multiprocessors as other processors can access resources in parallel - even if all interrupts are disabled on all processors. Userspace programs shouldn't be allowed to do it as they can disable preemption and monopolise the CPU. Slows interrupt response time.

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.

Test and set is an atomic instruction (meaning uninterruptable) which always returns the content of the tested memory location, and additionally set the location to 1 if (and only if) the memory location contained 0.

lecture notes: enter_critical_region: tsl register, lock // try to set the lock, and copy the lock to 'register' cmp register, 0 // if the register is zero jne critical_region // then we don't hold the lock. ret // return to caller after executing the critical region

leave_region: move 0, lock // set lock to zero to release it. ret

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

The producer consumer problem is a problem that involves filling and depleting a bounded buffer. The producer inserts entries into the buffer, and can sleep if it is full. The consumer must retrieve data from the buffer, possibly sleeping if it is empty. Transactions on the buffer itself may require a mutex. This is seen in operating systems in buffers for everything, particularly i/o device input queues.

Given a single, fix-sized queue, the producer process(es) generate(s) an item to put into the queue while the consumer(s) take(s) the item out of the queue and consume(s) it.

Take file system buffer cache as example, the file system writes are produced by the application requests. OS has to find a free entry in the buffer cache to store the write or blocks the request until a free slot is available on the bounded buffer cache.

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

A semaphore has two operations: Verhogen and Proberen, increment and decrement. P (decrementing) atomically reduces a counter by 1; if this drives it negative, it blocks until the counter is at least 0. V (incrementing) atomically increases a counter by 1, and wakes threads sleeping on this semaphore.

Interrupts are disabled around these code regions such that they are atomic.

Pseudocode: P: // while decrementing would drive it negative, sleep.
sem.count--; while (sem.count < 0) {

add this process to semaphore queue thread_block();

}

```
V: // increment and wake other threads sem.count++; if (sem.count <= 0) {  
    remove head of semaphore queue thread_wakeup(head);  
}
```

Describe how to implement a lock using semaphores.

A lock using semaphores is achieved by initialising the semaphore to 1. The count of a semaphore is effectively the number that may be allowed before preventing more in, so if it is 1, it acts as a lock. e.g.:

P(sem);

V(sem);

What are monitors and condition variables?

Monitors are a programming language construct designed to simplify control of concurrent access. Code related to a shared resource (and the shared resource itself) is placed in a monitor, then the compiler ensures that there is only one thread executing inside the monitor at a time.

Condition variables are entities used within monitors to block and wait for an event, the operations 'wait' waits for the event to occur and 'signal'. operation is used to generate the event. Basically, the thread invoking 'wait' sleeps until another thread wakes it with 'signal'.

What is deadlock? What is starvation? How do they differ from each other?

Deadlock is the state of a set of threads where all threads in the set are blocked waiting for another thread in the set to release a resource, and a required resource can only be released by a thread in the blocked set. The set remains blocked forever.

Example with a set size of 2 {1,2}: Thread 1 needs the resource A and B to proceed, it holds A. Thread 2 needs resource A and B to proceed, it holds B. Both are blocking on the resource they need and cannot release what they are holding.

Starvation is the phenomenon where the allocation of resources is 'unfair', and something cannot proceed because it never gets the

resources it requires even though the resources may become available as the resources are always granted to something else.

In deadlock, processes halt because they cannot proceed and the resources are never released. In starvation, the system overall makes progress using (and reusing) the resources, but particular processes consistently miss out on being granted their resource request.

What are the four conditions required for deadlock to occur?

Mutual exclusion: only one thread can use a resource at a time
Hold and wait: a resource can be held, then blocking whilst waiting for more resources
No preemption: resources cannot be forcibly taken away from process holding it
Circular wait: A circular chain of 2 or more processes, each of which are waiting for a resource held by the next member in the chain.

Describe four general strategies for dealing with deadlocks.

1) ignore it - easy - especially if the other strategies are very costly, and deadlock is a rare occurrence

2) detect and recover

detection: - do pages of maths - and some more maths - then some more - or perhaps explore a graph - with maths

recover : - then either steal preemptible resources, or try and save valid states of processes and roll back to them - looks complicated...

3) deadlock avoidance - disallow deadlock by trying to determine if allowing a particular resource allocation will cause deadlock - only allow progression of programs into 'safe states', which guarantee process completion. - difficult, because we need to know process requirements in advance, or do checks before allocating resources. this can often be impossible practically

4) deadlock prevention - negate one of the 4 requirements of deadlock. - solve hold and wait, by making threads drop resources they hold if they cannot get all of the resources they require for progress. this can cause starvation. - usually done by using ordered resources, to break the circular dependency condition. breaking other conditions is rarely a

feasable option.

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.

Deadlock detection can be done by finding closed loops in the graph, which involve two or more processes requesting resources which are hold by other processes.

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

Give 1 2 to P1: Current allocation matrix P1 2 5 P2 4 1 P3 1 2 P4 2 0

Current request matrix P1 0 0 P2 4 3 P3 1 7 P4 5 1 Availability 0 2

P1 finishes, releases 2 5. Current allocation matrix P2 4 1 P3 1 2 P4 2 0

Current request matrix P2 4 3 P3 1 7 P4 5 1 Availability 2 7

Give 1 7 to P3: Current allocation matrix P2 4 1 P3 2 9 P4 2 0 Current request matrix P2 4 3 P3 0 0 P4 5 1 Availability 1 0

P3 finishes, releases 2 9: Current allocation matrix P2 4 1 P4 2 0 Current request matrix P2 4 3 P4 5 1 Availability 3 9

Now there is no possible request that can be fulfilled, so it is deadlocked.

If the availability vector is as below, is the system above still deadlocked?
2 3

Scenario 2: 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 2 3

Give 1 2 to P1: Current allocation matrix P1 2 5 P2 4 1 P3 1 2 P4 2 0

Current request matrix P1 0 0 P2 4 3 P3 1 7 P4 5 1 Availability Vector 1 1

P1 finishes, releases 2 5: Current allocation matrix P2 4 1 P3 1 2 P4 2 0
Current request matrix P2 4 3 P3 1 7 P4 5 1 Availability Vector 3 6

Now there is nothing satisfyable. Deadlocked

Scenario 3: Is the system deadlocked if the availability is 2 4

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 2 4

Give 1 2 to P1 Current allocation matrix P1 2 5 P2 4 1 P3 1 2 P4 2 0
Current request matrix P1 0 0 P2 4 3 P3 1 7 P4 5 1 Availability Vector 1 2

P1 ends, releasing 2 5 Current allocation matrix P2 4 1 P3 1 2 P4 2 0
Current request matrix P2 4 3 P3 1 7 P4 5 1 Availability Vector 3 7

Run P3 (releases 1 2): Current allocation matrix P2 4 1 P4 2 0 Current
request matrix P2 4 3 P4 5 1 Availability Vector 4 9

Run P2 (releases 4 1): Current allocation matrix P4 2 0 Current request
matrix P4 5 1 Availability Vector 8 10

Then P4 can run. System isn't deadlocked.

General approach:

a) look for a process whose requests \leq the availability. we know we can then satisfy the requirements for this process, so we assume it gets its resources required and terminates. Thus, we can

b) release all resources *currently allocated* to that process, and add then to the pool.

c) repeat for all processes. If you eventually reach a point where you cannot do this, and there are still processes remaining, the system is deadlocked. If you release them all, you are safe.

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

An operating system can take resources if they are preemptible (meaning the process doesn't die if it's taken away). It can kill a process to free up resources. Otherwise, it can try and take snapshots of process states, and roll back to state where the system was not deadlocked.

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

The banker's algorithm must know the maximal resource requirements of all processes.

We assume that the banker can keep track of resource availability and usage, and the process that receives the resources they require eventually finishes and releases them.

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

Deadlock prevention is removing the capability of deadlock to occur, by negating one of the 4 conditions of deadlock. A practical example of how to achieve this is to use ordered resources, such that circular dependencies cannot form. see above

Note: avoiding deadlock via careful allocation (e.g. banker algorithm) is not prevention, it is avoidance.

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.

Sparse files mean that there aren't physical blocks on the disk dedicated to holding empty parts of the file. An application that might benefit from this is a CDROM image representing a not full disk. the filesize would be the size of the hard disk, however unused parts in the image do not mean blocks wasted. This means filesystems support this kind of file that if some entire blocks of one file contain nonmeaningful data (should be null), these blocks need not be allocated on the disk.

Some database applications that seek to large offsets obtained from large hash values may benefit for sparse files.

Give an example of a scenario that might

benefit from a file system supporting an append-only access write.

Log files (e.g. used for auditing), append-only prevents modifying (accidentally or otherwise) any existing log entries.

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

Large block sizes would be good when uber large files are being used, e.g. a video collection on a disk, and almost no little files. Particular benefits in sequential access. (and less metadata required to keep track of the smaller number of blocks)

It would be a waste of disk blocks in filesystems that store a lot of little files, like news, /etc, my physiology study so far, etc. It is also poorer performance when there is random access of small segments of data, as we need to load the entire block, even though we may only need a tiny portion of it.

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

CDROM file systems - you know how big all the files are prior to burning the disk.

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

Sequential file access is suited to chained file allocation. The chain is effectively a linked list that is traversed as the file is read. Random access is poor, because each prior link of the chain must be considered to get to any point of the file.

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

Indexed allocation is appropriate for random access, as there is a constant time (slowing down very slightly with increased indirection

levels), for accessing any part of the file.

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

Bitmap based block allocation is a fixed size proportional to the size of the disk. This means wasted space when it's full. A free block list shrinks as space is used up, so when the disk is full, the size of the free block list is tiny.

Contiguous allocation is easier to perform/find with a bitmap.

== 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 can be larger than the filesize, if the file is non-sparse, and additional blocks are required for holding metadata. This is the norm.

Block count can be smaller, in the case of a sparse file using only a small fraction of the file, the data and metadata is less than the notional size of the sparse file.

They can be equal with just the right tradeoff of the above cases.

Why might the direct blocks be stored in the inode itself?

Quick access to the start of the file stored in the inode itself. For small files, no direct blocks required, entire file can be in the inode. This improves the performance for small files.

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?

Triple indirection is slower, as it may result in multiple seeks to get to the desired block. Seeks take a long time. Tiny files that only use a few KB will perform much better if they do not have to do seeks to find each level of indirection... see 42 (lol).

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.

block size = 512

number of block numbers in an indirection block = block size / 4 = 128

number of blocks for file data in that file object

= $16 + 128 + 128^2 + 128^3$ Maximum file size: (direct + single indirect + double indirect + triple indirect) * (blocksize) = $(16 + 512/4 + (512/4)^2 + (512/4)^3) * (512) = 1\,082\,204\,160$ bytes (corrected value, was previously incorrect 68853964800 bytes)

The berkely 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).

Block groups are clustered arrangements of related data blocks and inodes. This increases performance through spacial locality of data arranged on the disk. System V didn't do this, so it didn't see performance gains through locality.

Also, the super-block was replicated in each group, so the file system was no longer lost if the one s5fs super block went bad.

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

The reference count is the number of hard links (names in directories) presently pointing at a particular inode. When this reaches zero, it is an indication that that inode can be deleted, since nothing points at it anymore (it has no name).

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 when the unit of transfer/update is different between two entities (e.g. updating 1 byte in a disk block requires buffering the disk block in memory). Buffering can improve performance as writes to disk can be buffered and written to disk in the background - this does reduce file system robustness to power failures and similar.

Caching is loading used data into the buffercache, such that subsequent reads to the same area of data are accelerated by using the data in cache and not having to access disk. The acceleration is seen due to the principle of locality; if data is used, data nearby will be used soon, so the application will run a lot faster if it is in RAM rather than requiring a disk seek, which is several orders of magnitude slower.

What does flushd do on a UNIX system?

Periodically syncs (writes) dirty blocks to disk (every 30 seconds) to avoid significant data loss on unexpected OS shutdown/powerloss.

Also indirectly improves performance as it increases the number of clean blocks, and clean blocks can be booted immediately from buffer cache rather than writing them back, which should be a speedup.

Why might filesystems managing external storage devices do write-through caching (avoid buffering writes) even though there is

a detrimental affect on performance.

External devices have no guarantee of connectivity. To ensure filesystem consistency, writes should be made without caching and be made as atomic as possible in case the drive is removed before the write has been committed to the drive.

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 management machine using base limit registers and static partitioning, and b) a similar machine using dynamic partitioning.

- Internal fragmentation: the space wasted internal to the allocated region. Allocated memory might be slightly larger than requested memory. - External fragmentation: the space wasted external to the allocated region. memory exists to satisfy a request but it's unusable as it's not contiguous.

Static partitioning is more likely to suffer from internal fragmentation. Dynamic partitioning is more likely to suffer from external fragmentation.

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

First fit - allocate into the first available gap found of adequate size, starting from the beginning of memory. Next fit - allocate into the first available gap found, resuming the search from the last allocation. Best fit - search all of memory to find the smallest valid gap and allocate into it, on the basis that it'll leave the smallest unusable gap. Worst fit - search and place into the largest area, on the basis that it is likely to leave a gap big enough for something else to use.

First and next fit are used in practice as they are faster, with comparable

performance. First-fit Scan memory region list from start for first fit. Must always skip over potentially many regions at the start of list.

Next-fit Scan memory region list from point of last allocation to next fit. Breaks up large block at the end of memory

Best-fit Pick the closest free region in the entire list Leaves small unusable regions and slower due to searching of entire list.

Worst-fit Finds the worst fit in the entire list Slower as it searches entire list. Fragmentation still an issue.

First-fit and next-fit most commonly used as it is easier to implement and works out better.

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 act of running each whole process in main memory for a time then placing back onto disk and vice versa. It is used when the system does not have enough main memory to hold all the currently active processes.

Assuming that an application is one program and hence a single process, swapping will not permit an application(process) requiring 16M of memory to run on a machine with 8M RAM (main memory) as the whole process is too large to fit into main memory.

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

Each process has its own page table, which is a mapping between virtual pages and physical frames. A page is a chunk of virtual memory (commonly 4KB) that exists within the process's virtual memory space. This is mapped to a physical frame (4KB of real 1s and 0s in RAM) using a page table.

The program is given the illusion of running in its own address space.

CPU instructions execute the program, which will perform memory requests to virtual addresses. These are converted to physical addresses by looking up in the Page Table, (more specifically the Translation Lookaside Buffer (TLB)), then the physical memory is accessed.

The TLB is the fast hardware implementation of a page table lookup. It contains recently used page table entries, which allow a really quick mapping between virtual pages and physical frames. In the event that the TLB doesn't contain the virtual address requested, a page fault is triggered. The operating system is called to find the appropriate mapping from the page table, stored in RAM, and load it into the the TLB as appropriate. (in MIPS... Pentium architectures have some dedicated hardware for pagetables, and therefore some hardware support for tlb refill).

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

A simple base limit with swapping system requires that all of the data from base to limit must be resident in contiguous main memory for execution. This can lead to external fragmentation, and swapping is much more likely to occur when memory is becoming full. This incurs a huge penalty when context switching, because a large chunk must be swapped out to disk, then a large chunk retrieved from disk before execution can resume. Disk is horridly slow compared to ram.

In a paging system, physical memory doesn't have to be contiguous, and there is no external fragmentation, so only the minimum amount of memory needs to be moved in a context switch; only the pages required to hold the resident set are required to be in memory for execution.

The paging sceme makes it possible for the resident sets of many processes to be present in memory, which drastically increases the performance of a context switch as there is much less likely to be a disk access on context switching.

Also, paging supports running a process who's virtual memory usage is greater than the size of physical RAM in the machine.

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 is made up of many small base limit portions (segments), each having a segment number and a size. The segments allow multiple processes to share data on a per segment basis, and since segments may be arbitrarily defined to contain code/stack/whatever, this could be a per subroutine basis. It also allows permissions of each segment to be set appropriately. An annoyance is that the programmer must be aware of the segmentation, and this isn't trivial. Segments are loaded into contiguous regions of physical memory. The segment number, base, limit & permissions are stored in a per-process segment table.

An instruction looks like this: operation, whatever, segment number, offset.

When the instruction is computed, the segment table is looked up to find the appropriate segment based on segment number. If the segment number is present in a table entry, and offset is smaller than limit - base, the physical address is formed out of base + offset. Then the physical memory is fetched and the operation performed. If the segment isn't present, or the offset is out of bounds, a fault is handed to the operating system that would then deal with the process.

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

TLB is a fast hardware implementation of a page table lookup. Recent page table entries used are loaded into the TLB, such that virtual to physical address conversion is really fast. When a virtual address cannot be successfully translated by the TLB, a pagefault is thrown and the OS finds the appropriate frame in the page table, or if it is an invalid operation, kills the process. see 53

Each TLB entry has page number (for matching), and frame number associated with that page, privileges, and ASID (sometimes). The frame number is the significant bits of the physical memory address. Privileges are what operations can be performed; this provides some memory protection to shared libraries and code segments; e.g. a process cannot

execute in it's stack, or cannot write to code, etc.

Address spaces, and the page tables used to map them, are a per process entity. The ASID (address space identifier) is used to check if tlb entries present are those of the by the currently running process, and therefore the currently active page table. ASIDs allow multiple processes to have hot TLB entries by not requiring complete flushing of the TLB on a context switch. This will boost performance just after a context switch.

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

see previous question

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

For a 32 bit (4GB) virtual address space, with 4K pages/frames: A two level page table uses indirection to save memory and at the expense of time. The top-level page table (an array of 1024 pointers to 2nd-level arrays) would be indexed such that the most significant bits of the virtual address ($VADDR[31:22]$) are used to look up the 2-level array (of page-table entries - frame numbers and permissions), where some of the less significant bits $VADDR[21:12]$ are used to index that table.

This is preferential to simple page table arrays as it allows second level pages to be omitted if they're not required. It also means that a ridiculously long array is not required either. 2^{20} page table entries, 4 bytes each, is 2^{22} bytes, is 4MB of memory required to hold a 1-level page table. If the page table itself can be swapped, this is a big disk hit. If not, 4MB must be resident in physical memory as contiguous segments, which can cause fragmentation.

2-level page table is slower to look up as it requires 2 memory references per lookup. assume we got 32-bits virtual address and 4k page size, top level table A, and second level table B, first 10 is to locate A's index, and next 10-bits is the offset to locate B's index, using A's content to locate the B's base address, and plus the offset to B's index, we got the physical frame index; then using the last 12 bits plus the B's content, finally the physical address located.

The two-level table avoids allocating pagetables for all possible translations, if the translations are not required.. A disadvantage is that it

requires 2 memory references to look up a page table entry, instead of one.

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

An inverted page table is a list of pages sorted by frame number. Inverted page tables hash the virtual address page number to lookup into the page table, which then matches the page number (or uses a linked list structure to find the page). This index is the frame number.

Page table size is proportional to size of physical memory, not address space size. This is particularly important on 64 bit machines, where virtual address space size, if 2^{64} bytes, is 16 Giga Giga bytes; storing a multi level page table for this would require too many levels of indirection and be slow, and require alot of space to store. Inverted page table is an array of page numbers sorted (indexed) by frame number. Virtual address is hashed to locate the entry in the frame table. The most important advantage is that its size is related to physical memory size, not virtual memory. Inverted page table is good for system with large addressing space but significantly less physical memory (e.g. 64-bit addressing with 2GB of ram)

What are temporal locality and spatial locality?

temporal locality (locality of time) is the phenomenon where if a piece of memory is accessed, it is likely that that piece of memory will be accessed again in the near future. spatial locality is where if a piece of memory is accessed, it is likely that future accesses will be to similar addresses. These phenomena are why caching works; memory accesses are rarely random.

What is the working set of a process?

The pages/frames currently relevant for the execution of a process. It includes the current top of stack, areas of the heap being accessed, the current area of code segment being executed, and any shared libraries recently used. The workset of a process consists of all pages accessed within in the recent past (a small, current, time window).

How does page size of a particular

architecture affect working set size?

If the page size is small, the work set size is smaller (in terms of memory used, not absolute number of pages) as the pages more accurately reflect current memory usage (i.e. there is less unrelated data within the page).

If the page size is large, working set also increases as more unrelated data within the larger pages are included in the current working set.

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

Thrashing is when you're trying to run java 'hello world', and any other process on a system. The memory requirements of the currently running processes are so large that the working sets of the executing processes cannot be contained in physical memory. So one process tries to run, gets a page fault, retrieves the page from swap, boots out another processes' frame doing that. Then it context switches and the other process tries to run, but it's memory has been booted, so it page faults to swap, etc. The time is spent swapping frames rather than getting anything done.

It could be detected by monitoring swapping levels compared to real cpu utilization. If swapping is huge, and utilization is down, thrashing is occurring. Suspending processes can alleviate thrashing; hopefully some processes will go below the thrashing threshold, run, terminate, then free up memory for other processes. Installing more ram helps, and not running java hello world helps. Thrashing occurs when too many processes are run on a processor at a given time. What occurs is the the number of page faults increase dramatically and the virtual memory subsystem is constantly paging pages in and out of memory. This occurs when the working set of all processes is larger than the amount of RAM available on a system.

This can be detected by monitoring the page fault frequency and CPU utilisation. If increasing the number of processes results in increasing page fault rate and decreasing CPU utilisation, then the system is thrashing.

To recover from this condition the number of processes currently in the running/ready queue must be reduced. This can be accomplished by suspending processes (pushing them onto the sleeping queue), so that pressure on physical memory is reduced (suspended processes

eventually get swapped out), and thrashing subsides.

Enumerate some pros and cons for increasing the page size.

pros: reduces pagetable size increases tlb coverage increases swapping I/O throughput, as small disk transaction times are dominated by seek & rotational delays.

cons: increases page fault latency, as swap response time is slower due to more data. increases internal fragmentation of pages, as there is more 'wasted page' in the working set

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 a process before they're accessed. Wastes I/O bandwidth if pages are loaded unnecessarily, and worse if an unnecessary page kicks out a necessary page. Demanding paging (or fetch on demand) : load pages when page fault occur, and is more common.

Prepaging policy : Pre-paging brings in more pages than needed at the moment. IO is improved due to reading large chunks but waste bandwidth if pages aren't used. Especially bad if we eject pages in working set in order to pre-fetch unused pages. Hard to get in right in practice.

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)?

Monitor page fault frequency of an application - high frequency => needs more memory allocated to resident set size

Name and describe four page replacement

algorithms. Critically compare them with each other.

From best to worst (lecture notes):

- 1) Optimal - use time travel to find the page that won't be used for the most time and boot it. Impossible to implement - used only as a theoretical reference point for comparing over algorithms.
- 2) Least Recently Used - calculate whatever page hasn't been used the longest, and boot it. impossible to implement efficiently in practice - requires a timestamp on every memory reference.
- 3) Clock page replacement - set 'referenced' bit to 1 when something is used. when looking for an entry to boot, set these bits to 0 if they're 1, and kick the first 0 candidate found. resume searches from the last one booted. Efficient (implementable) approximation of LRU - used in practice.
- 4) FIFO - remove the page that has been there longest - does not consider actual memory usage in its decision - it will evict the most frequently used page in the system.

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 in the i/o subsystem is required to match two entities when the unit of transfer is different (e.g. updating 1 byte in a disk block requires buffering the disk block in memory).

Buffering is advantageous if the peak rates of producing and consuming data is different. Data can arrive asynchronously, pile up in a buffer whilst the processor is doing something else useful, then it'll context switch to the process waiting on that buffer which can then do an operation on lots of data;

Too much buffering in really high speed networks can reduce performance, if the time required to copy between buffers is comparable to the time spent acting on data.

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?

Pushing more I/O functionality into device controllers relieves the operating system from having to perform it (e.g. adding DMA to a device controller relieve the OS from having to copy data to memory). This results in improved performance as the OS only has to supervise the device, instead of doing all the work itself in the device driver, also the I/O functionality can now happen in parallel with the OS/application doing something else. It may also reduce device driver complexity and improve reliability, but that is dependent on whether the complex device is actually reliable itself.

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

In polling based I/O, the OS simply busy waiting on a loop, continually checking for I/O, whereas in interrupt-driven I/O, when an I/O job arrive an interrupt is generated and the OS runs the interrupt-handler, after that it return to what it was doing before the interrupt came.

If the cost for interrupt handling (the cost of switching to kernel-mode and preserving the user-level context and then eventually returning again) is bigger than the time spent busy-waiting in polling and the rate of I/O jobs arrive is high then polling is preferred, otherwise the latter is a better choice. 1 poll is cheaper than 1 interrupt; but 1 interrupt guarantees an event, polling may have thousands of wasted polls. Interrupt driven I/O is preferential when input is unpredictable, e.g. packets, typing, mouses. Polling is favourable when the time spent busy waiting is less than the time spent dealing with interrupt overheads

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

Many I/O devices utilise a buffer to store incoming information. Upon an event, such as entering the 'new line' character, it notifies the OS / user program. Data is then copied from the buffer to a location which it can be used, and the consumed data is cleared from the buffer.

This is an example of a bounded-buffer producer-consumer problem, where the producer is the I/O device, the consumer is the OS / user program and the buffer is bounded by its size. Therefore, concurrency control must be implemented to ensure that race conditions do not occur. Additionally, the OS / program must consume the buffer quickly enough to prevent it from overflowing and lose information.

What is disk interleaving? What problem is it trying to solve?

Disk interleaving is putting blocks like 1 - 2 - 3 - 4 rather than 1 2 3 4 5 6, such that each read occurs at a nice time. It addresses latency between successive disk block reads lining up with latency of disk rotation, so that when the next read comes along, it's just before the block requested rather than just after.

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

cylinder skew is vaguely like interleaving, except it is an offset of the next block between cylinders, such that rotation and cylinder head seek latency are taken into account. The idea is for when the request for the block on the disk results in a seek to the next track, the block is further around on the disk such that when the seek is complete the disk block is just arriving under the head, instead of having been missed. T

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

SSTF: shortest seek time first: service the requests that requires the least head movement next. quick, but can suffer starvation. FCFS: process requests in the order they were received. deteriorates to random with multiple processes accessing the disk. SCAN: service all pending requests

in ascending order until the end of the disk platter is reached, then service all requests in descending order until the start of the disk platter is reached, repeat ad infinitum. C-SCAN: SCAN, but only the ascending pass is done. When it reaches the end it just goes back to the beginning. Increases performance due to ascending disk block numbers of contiguous files, that are likely to be missed on the downward pass of SCAN. 90. Why is it generally correct to favour I/O bound processes over CPU-bound processes? i/o is many orders of magnitude slower than cpu. therefore favouring i/o bound process won't actually give them more execution time than a cpu bound process, because they're just going to block again on the next i/o transaction. favouring them gives them a chance to operate when the i/o requests appear (system is more responsive, and I/O utilisation is higher as requests are re-issued faster); it also decreases the probability of packet loss, as the i/o buffers are given more chances to be read.

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

preemptive scheduling means that processes are context switched at regular time intervals, by the kernel. non pre-emptive scheduling, means a process returns control to other processes upon termination, or by yielding (cooperative multitasking). fair share of cpu-time in a non-preemptive system requires that all processes are yielding at similar time intervals. a single process can monopolise cpu time by not yielding.

Describe round robin scheduling. What is the parameter associated with the scheduler? What is the issue in choosing the parameter?

Round robin scheduling runs each process for timeslice t , then preempts it, puts it on the end of a ready queue. then runs the next process for timeslice t .

If t is too large, it is unresponsive. if t is too small, much time is wasted in context switching overhead.

The traditional UNIX scheduler is a priority-based round robin scheduler (also called a

multi-level round robin scheduler). How does the scheduler go about favouring I/O bound jobs over long-running CPU-bound jobs?'

It uses multiple ready queues of each priority. priorities are gradually increased over time to prevent starvation of low priority processes. They are increased by boosting the priority based on the amount of CPU consumed. I/O bound jobs are indirectly favoured as they consume less CPU.

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

In a rate monotonic scheduler, priorities are assigned based on the period of each task. In an earliest deadline first scheduler, priorities are assigned dynamically based on the deadlines of each task.

+ Rate monotonic : the shorter period, the higher priority the process is + EDF : the earlier deadline, the higher priority. 98. What is an EDF scheduler? What is its advantage over a rate monotonic scheduler? EDF is guaranteed to produce a feasible schedule if CPU utilisation is $\leq 100\%$

RMS only provides such a guarantee if the utilisation is less than that specified in the formula in the lecture notes, which is significantly less than 100%

Describe why the use of spinlocks might be more appropriate than blocking locks on a multiprocessor when compared to a uniprocessor. Is there a trade-off between spinning and blocking on a multiprocessor? Discuss.

Spinlocking on a uniprocessor is useless, as another thread on the same processor needs to release it, so blocking asap is desirable.

On a multiprocessor, the thread holding the lock may be presently active on another processor, and it could release the lock at any time. On a

multiprocessor, spin-locking can be worthwhile if the average time spent spinning is less than the average overheads of context switch away from, and back to, the lock requestor.

Is a single ready queue on a multiprocessor a good idea? Why?

No, as it is a shared resource, which introduces lock contention when processors are trying to schedule work, as they must all wait on a single lock on the ready queue to determine who to schedule next.

What is thread affinity. Why might it improve performance?

Thread affinity is where a thread has a preferential processor for scheduling on a multiprocessor machine. This increases the probability of being scheduled on the processor previously scheduled on, which in turn increases the probability of being scheduled onto a processor that has a hot cache. Hot cache is good, it means less cache misses and higher performance.