

A1. CPU speed for a given cost is improving by a factor of 2 every 18 months. DRAM cost is improving at a rate of a factor of 2 every 3 years. Which of the following statements is true, if it currently costs twice as much as you'd like to build a PDA to a given specification?

d you must wait for both but the DRAM price will be the bigger holdup

A2. You are designing a high-availability system. This means

c you are trying to minimize downtime

A3. In terms of requirements for systems programming

c Java's support for primitive types like int would be useful for OS coding

A4. When an interrupt occurs, the following has to be taken into account:

e both *b* and *c*

A5. A given interrupt handler should not be interrupted, and it is permissible that interrupts of this kind be missed. The following are options to avoid problems when this actually happens:

d *b* or *c*

A6. Compared with a synchronous I/O request

d both *b* and *c*

A7. In general terms finding other work instead of waiting for I/O

c is less an issue for I/O than keeping the CPU busy

A8. The Solaris approach to threads

b provides flexibility according to dynamically determined needs

A9. The UNIX fork system call

a can be efficient with copy on write

A10. For a round-robin scheduler

d both *b* and *c*

A11. Initializing a semaphore S [algorithm p 201 Silberschatz *et al.* 2003; lecture week 5 slide 25] with a negative counter (N) results in a situation where

e none of the above

A12. Initializing a semaphore S [algorithm p 203 Silberschatz *et al.* 2003; lecture week 5 slide 28] with a negative counter (N) results in a situation where

e both *b* and *c* : why is *b* not correct?

A13. An atomic memory modification and test instruction

c makes implementing synchronization easier

A14. A simple spinlock (loop testing a variable, while attempting to set it atomically)

c may be useful for a very short critical section with a low probability of contention

- A15. A deadlock will *never* occur if
d *any of the above*
- A16. Contiguous allocation in main memory is
d *can lead to external fragmentation*
- A17. Segmentation in main memory
b *can lead to external fragmentation*
- A18. In a situation where the physical address space is very small (compared to virtual)
d *either a or b depending on memory usage*
- A19. A multilevel page table
d *all of the above*
- A20. An optimal page replacement strategy is
b *efficient at minimizing page faults but almost impossible to implement*
- A21. Storing swapped out virtual memory as normal files means
d *b and c*
- A22. Using a memory-mapped file differs from ordinary file I/O in that
b *you need not read the file in explicitly*
- A23. A tree-structured file system
a *allows multiple items to have the same name*
- A24. Aliases are a problem in file systems because
e *b and c but not a*
- A25. The SCAN disk scheduling algorithm
e *none of the above*
- A26. The C-LOOK disk scheduling algorithm
c *is organized to avoid looking at recently serviced parts of a disk*
- A27. A distributed operating system
b *allows remote files to be used the same way as local files*
- A28. A malicious code fragment which can only run when attached to another code fragment is
b *a virus*
- A29. Language-based security (e.g., as in Java) makes sense if
e *both a and d*

A30. To implement secure code in C++

b would require major limitations compared with standard C++

B BONUS QUESTION FOR 5 MARKS

i) Why are priorities useful in a scheduler? [3 marks].

1. *can ensure I/O bound process get fair share of CPU: I/O-bound process high priority, CPU-bound process low priority (2 points)*
 2. *can avoid starvation by increasing priority of stalled processes*
- Other points possible.*

ii) Explain why an inode-based multilevel pointer scheme does not require linear searches to find a given location in a file. [2 marks].

1. *pointers are arranged in a tree-like structure with indexing within nodes rather than linear search*
2. *can get quickly to right part of "tree" in fewer than i steps to reach location i .*

Part C – Answer one of PART C or PART D not both PARTS
Design Issues: 20 MARKS TOTAL

Question C1 Threads and Processes – 7 marks

- a. A proposed new operating system does not support threads kernel-level threads, but does have primitives to do the following:
- on an IO interrupt, restart a (possibly stalled) handler function in the same process as that which caused the interrupt; the API for such functions is:
 - `// set up a handler routine to take control on an interrupt`
`install_handler (interrupt_name, functionname);`
 - `// stall the handler, returning control to OS`
`suspend ();`
 - the handler routine replaces the OS's standard handling of the given interrupt once installed
 - such a handler, once installed, has its own stack and can access global variables; if it returns, the interrupt will be handled by the OS
 - generate an IO interrupt after a fixed time interval
 - allow a user-level program to define its own interrupt

Discuss how these features could be used to implement user-level threads. [4 marks]

Need to think about how user-level threads can be implemented using a handler routine to actually launch each thread; use suspend (); to give up the CPU. Need to set up the interrupt handler for timeouts and for IO operations. Any reasonable strategy would do for 4 marks. A bit more challenging than real exam questions.

b. Explain why threads are useful to obtain a responsive feel to a user interface. [3 marks]

1. *can have a thread for each major user interface element*
2. *thread responding to user interface events can keep active while I/O happens*

3. do not need to have complex programming strategy to achieve same effect

Other points possible.

Question C2 General–13 marks (from various sections of the course)

- a. The following is proposed as a solution to the bounded-buffer producer-consumer problem, with the intent that the whole buffer should be used (as opposed to the approach in the text book p 109; lecture week 3 slides 36–38) which leaves an entry unused:

```

initialized filled = 0, emptied = 0;

// if filled-emptied == size-1 all slots filled
// since arrays are indexed from zero
producer:
    while (true) {
        while (filled-emptied == size-1); // spin
        buffer [filled % size] = produced;
        filled++;
    }
consumer:
    while (true) {
        while (filled-emptied == 0); // spin
        consumed = buffer [emptied % size];
        emptied++;
    }

```

Comment on strengths and weaknesses of this solution. **[5 marks]**

1. *Good: appears to be correct and simple to understand:*
 - 1.1. *filled-emptied is number of used spaces*
 - 1.2. *no race conditions (each variable only modified in one process)*
- Bad:*
 2. *counters never decremented so will eventually overflow*
 3. *adds another variable so not necessarily less memory than before*
 4. *still only handles 1 reader, 1 writer*
- b. For the FAT allocation scheme, under the following assumptions, calculate the size in bytes of a table needed to represent 64 files, each sized 1GB, on an 64GB drive:
 - 1Kbyte block size
 - minimum bits to represent number of blocks, plus 1 bit to represent end of file
 - i State all assumptions and show detailed working **[5 marks]**
Assume we can implement "pointers" as an unsigned integer of any number of bits (other reasonable assumptions accepted).
First find no. blocks, then work out bits to represent:
 $64 \times 1\text{GB} / 1\text{KB} = 64\text{M blocks}$. $1\text{M} = 2^{20}$, $64 = 2^6$, so $64\text{M} = 2^{26}$ so need 26 bits per "pointer", total 27 to represent EOF. Now for total size, we need 27 bits x number of blocks = $27 \times 64\text{M} / 8 \text{ bytes} = 216\text{Mbytes}$ (remember to convert bits to bytes).
2 for assumptions, explanation, 2 for working, 1 for answer.
 - ii Discuss how your answer would differ with a UNIX-type inode scheme with direct, indirect double and triple-indirect pointers. **[3 marks]**

You are not asked here to do a detailed calculation ...

The inode scheme requires allocating index blocks. Since the number of index blocks will not be an exact fit to 1M blocks, there will be some wasted space at the end of the last index block, so the overheads even with minimal-sized pointers will be slightly higher than for the FAT scheme – any good 3 points showing you know the difference between inode and FAT schemes ...

Part D – Answer *one* of PART C or PART D *not both* PARTS Systems Programming: 20 MARKS TOTAL – based on assignments

Question D1 Multithreaded Code–10 marks

In the following code fragment (part of a possible solution to Assignment 1):

```
// sort thread 2*i joins read thread i
void launch_threads (int N, char *fileNames []) {
    int i;
    FileInfo filesetup[N];
    ThreadInfo sortsetup[N];
    init_threads (2*N);
    for (i = 0; i < N; i++) {
        sortsetup[i].which_thread = 2*i;
        sortsetup[i].N = N;
        launch_thread (sort, (void*)&sortsetup[i], 2*i);
        filesetup[i].filemode = "r";
        filesetup[i].filename = fileNames[i+1];
        launch_thread (readFile, (void*)&filesetup[i], i);
    }
}
```

- a. Explain why there are timing-based bugs in this code. [4 marks]

Sorts are being launched before reads and so may hit a join before the sort thread has launched, which is an error in Pthreads. (1 for error, 1 for explaining)

Also, variables local to launch_threads are being passed to the threads and these will not exist anymore when launch_threads returns. This could result in incorrect data being in the location accessed by the tread. (1 for error, 1 for explaining)

- b. Rewrite the code to correct the problem(s). If you want to only show the parts which differ from the original, mark the places where changes are made in the listing at the top of the page, but write the changes below – *make it clear what you have done*. [6 marks]

Change the order of the calls to launch_thread so the reads are launched first. Move the arrays to outside the function (but you must allocate their space in the function because N isn't known before the program starts):

*FileInfo *filesetup;*

*ThreadInfo *sortsetup;*

Then in launch_threads, before the loop:

*sortsetup = (ThreadInfo *) malloc (sizeof (ThreadInfo) * N);*

*filesetup = (FileInfo *) malloc (sizeof (FileInfo) * N);*

3 for explaining, 3 for getting details right

Question D2 Virtual Memory–10 marks

Explain how the implementation of a FIFO policy differs from the implementation of a clock algorithm. Outline any differences in data structures and algorithms used.

Clock needs to keep track of order accessed; a bit of state which it periodically clears: needs a simple representation of order of physical frames with this extra “used” bit. [2]

Key issue, for speed: must be able to find frame in fixed number of operations (i.e. not proportional to n, log n, etc.) [1].

FIFO needs to organize memory 2 ways:

- *To find frames in order allocated to handle victim selection [1]*
- *To find arbitrary frame when it’s allocated or deallocated by other parts of VM system [1]*

Clock can therefore use a simple array [1] but FIFO needs (or other options with same effect) an array ordered on and indexed by page frame number for more arbitrary allocations and deallocations [1] and a FIFO structure e.g., a doubly-linked list [1]. A doubly-linked list is useful because it’s easy to add to one end and remove from other [1] which are the basic operations for a FIFO. For example: add to list at front, remove tail – both can be done in fixed number of operations; also true of finding page through array [1].

Extra points: FIFO list should point to array (or contain page frame, used as index), and both should be kept accurate with respect to each other.