ECE391: Computer Systems Engineering **Fall 2023**
Study Guide

Sample Questions for the Second Midterm

This guide contains example questions to help you study for the coming midterm. Roughly speaking, all materials covered in class, in the notes (to the point reached in class before the test), on the problem sets, or on the machine problems are fair game. The test will focus primarily on the material covered since the last midterm, and in particular the topics listed below, but the test will also assume an understanding of the topics and study guide provided for the first midterm.

As was the case with the previous guide, most of these questions are short answer, whereas the test will include a larger portion of questions that require some code or analysis of code. The questions here are also not as carefully designed for clarity, in part so as to encourage you to think about different angles. On tests, we try to make our questions more precise. We suggest that you also attempt the previous exams available on the class web page.

Again as before, we are generally less interested in system-specific details than in whether you understand how systems are designed and why they are designed in a particular way.

You may bring **two** 8.5x11" sheets (both sides) of notes to the test. No other materials are allowed. No calculators or other electronic devices are allowed.

- MP2
  - memory fences
  - copy direction for overlapping memory copies
  - data structures for graphical images
  - build buffer and drawing model
  - image clipping
  - mode X image mapping
  - simple device drivers (example: Tux)
  - octrees
- hard interrupt execution
  - interactions with processor and PIC
  - assembly linkage
  - descriptor abstraction and usage for control
- tasklets
  - declaration and use
  - scheduling semantics and implementation
- task/process abstraction
  - terminology: task, process, thread
  - elements of a task, such as continuation, memory map, signal behavior
  - sharing elements between tasks
  - machine state
- filesystems
- system calls
  - assembly linkage (user and kernel sides)
  - calling convention
- scheduling
  - types of jobs (batch, interactive, real-time)
  - job state (example: runnable) and transitions
  - real-time jobs
  - epoch-based scheduler
  - static and dynamic priority
- hardware protection models
  - x86 ring model
  - protection levels for memory, I/O
  - stack swapping
- virtual memory
  - rationale for use
  - x86 segmentation
  - paging
  - translation lookaside buffers (TLBs)
- memory management abstractions (partial)
  - Linux dynamic allocation API
  - memory maps
  - virtual memory regions
  - exponentially scaled bins and the buddy system

19

8kB 8kB 8kB
8 8 8
8kB PD PT1 PT2

DE

256 2^{10}

DE ≤ 8KB

8kB

8kB
8
$2 \cdot 10^3$

2MB
2kB
$2 \cdot 10^6$
$2 \cdot 10^3$
256

1. How wide should a memory fence be? Describe the tradeoffs involved in choosing the size.
2. When does the direction used in copying data from one place in memory to another matter?
3. Explain the purpose of callbacks. Write a definition for a structure containing three function pointers, each of which takes two arguments (you pick the types) and returns a `double`.
4. Some of the higher resolution color modes supported by our emulated video cards support 24-bit color, in which each pixel is specified by 8-bit red, green, and blue values. Suggest two C data structures that you could use to hold a 16-pixel-wide by 12-pixel-high image for such a mode.
5. In the MP2 code, the game provides a virtual world of pixels, and the mode X code maps that world into mode X planes before storing the pixels to be displayed in the build buffer. As a result, planes in the build buffer must be remapped into the VGA planes when displaying a new screen. Consider two alternatives. First, one might instead draw a graphical image in a form convenient to C in the build buffer, then map to planes when copying to video memory. Second, one might map the window being displayed (a sub-portion of the virtual world provided by the maze) into logical planes, allowing the mapping to video memory to occur without plane-to-plane remapping. Explain why neither alternative is attractive.
6. Why is double-buffering useful?
7. If color bits could be packed tightly into bytes, how much space is saved by using an 8-bit palette rather than individual pixel colors (6-bit R, G, and B) in mode X (320 by 200 pixels)? How much more space is needed to extend the color space to 24-bit (8:8:8 R:G:B) using the same palette?
8. Explain how a virtual address is translated to a physical address using a two-level page table hierarchy. Count the number of memory accesses involved.
9. How does use of virtual memory prevent a malicious program from contaminating or destroying another program's memory?
10. Give an example of how memory fragmentation can negatively impact a system without virtual memory.
11. Explain the advantages and disadvantages of using a 4-MB page size rather than a 4-kB page size to support paging.
12. Virtual memory is useful for protection, sharing, avoiding fragmentation, and avoiding code and data relocation. Compare the effectiveness of segmentation with that of virtual memory for these four goals.
13. Explain the advantages of abstracting a memory map as a set of contiguous regions rather than relying directly on the page tables for descriptive information. *Hint: don't forget one the original goals of using virtual memory, in addition to any other advantages.*
14. Explain the tradeoffs between handling translation lookaside buffer (TLB) misses in hardware and handling them in software (via an exception mechanism).
15. Explain the purpose and benefits of stack swapping. What additional support is necessary for tracking stack pointers when more than one program is allowed to exist at a time?
16. In what cases is having more than two levels of protection rings supported by hardware useful?
17. As part of debugging your device driver, you decide to count the number of command requests made to the device that have yet to generate an interrupt. One approach to doing so is to use a shared variable incremented by the command request code and decremented by the interrupt handler. Suggest a better method that does not involve sharing data between these two pieces of code.
18. Suppose that you have decided to go with the increment/decrement approach in the previous problem. You declare a shared integer variable, `n_outstanding`, and execute `n_outstanding++` in the command request code and `n_outstanding--` in the interrupt handler. Explain the meaning of the term atomicity in this context, why atomicity is necessary to make this code work properly, and what mechanisms you can use to ensure that atomicity.

19. Three programs running on a multiprocessor arrive at critical sections protected by a common lock. Assuming that program A arrives first, followed by B, followed by C, and that a Linux spin lock is used to protect the critical sections, list the possible orderings of critical section executions (for example, C followed by B followed by A, which is not possible) and explain why the order is not uniquely determined by the order of arrival at the lock calls.

20. Describe the advatages and disadvantages of dynamically allocating objects for a device driver using slab cache relative to allocation using the kmalloc interface.

21. Why are I/O ports protected using a processor's hardware privilege level support?

22. Threads share a single address space, which in Linux is represented as a memory map. Explain how Linux supports sharing of memory maps between threads.

23. Look up the paging mechanism defined for the AMD-64 extensions to the x86 ISA. Describe the mechanism and how it works, then compare it to the mechanism described in class, particularly with regard to which aspects seem to have been driven by compatibility needs as opposed to other engineering requirements.

24. As part of implementing their Tux controller code, a friend adds two new ioctl's assigning them integer values -42 and 42 using an enumeration construct. Do these values reach your friends ioctl function? If not, explain what happens instead.

25. Draw a picture of the page tables (including the page directory) used for single process under your operating system. Be sure to indicate how the shell process' tables differ from those of another process.

26. Rather than re-writing the blocks of an executable image, you could instead use more complex page tables to re-map the blocks into a contiguous region of virtual memory. Explain how this mapping might work and what added complexity you must handle with this approach (remember that some parts of the file represent static program data).

27. Describe the system call calling convention for Linux in general terms and explain why (and where) indirection is used, why parameters are passed in the way that they are passed, why return values are defined as they are defined, and why specific registers are callee- or caller-saved.

28. Why is assembly linkage necessary between the address specified for a system call in the IDT and the C function that implements the system call? In other words, why don't operating systems simply use the C calling convention for system calls?

29. Explain how long it takes to access a random block on a disk and why. What implications does disk random access time (time to access a block, averaged over all blocks) have for filesystems?

30. Why does the ext2 filesystem overload the block pointers in index nodes to allow symbolic links with short target names to avoid using an extra data block? Does the storage saved matter?

31. For a filesystem with predominantly small files and a few very large ones, is it better to use a larger block size or to extend the levels of indirection supported by block lookup (with an extra level, for example)?

32. Why do many filesystems reserve part of the disk for empty space?

33. Explain the purpose and contents of an index node.

34. What difficulties or problems can arise when a filesystem caches information in memory? In light of these hazards, why is caching used?

35. When do tasklets run in Linux?

36. Why does a Linux memory map maintain two data structures (a linked list and a red-black tree) for keeping track of virtual memory regions?

37. Explain why the action of clearing the bits used to control tasklet scheduling and execution in Linux (`TASKLET_STATE_SCHED` and `TASKLET_STATE_RUN`) can be done with a simple store instruction as opposed to some type of synchronization primitive.

38. Rewrite (in pseudocode) the tasklet scheduling and execution code to make use of a single spinlock rather than atomically managed bits. Explain any changes necessary to data structures, and compare the ability of your solution with the original approach in terms of which pieces of code can execute in parallel on separate processors.

39. Using the x86 ISA as an example, explain the concept of a continuation and how it relates to the state of the machine.

40. Explain how atomic bit operations are used to ensure that a tasklet is does not execute on more than one processor at a time in SMP.

41. What are the difference between a batch job, an interactive job, and a real-time job? Give examples of each type.

42. What event or events might make a job runnable?

43. Describe two possible task states other than runnable and explain how and when they are used.

44. Explain how double-buffering is used to support the notion of epochs in the Linux scheduler.

45. Why are real-time jobs considered to have higher priority than interactive jobs in the Linux scheduler.

46. Under what conditions might Linux execute a batch job rather than a real-time job? Why was this priority inversion allowed in early kernels?

47. What happens to non-real-time jobs if a real-time job never leaves the runnable state in Linux?

48. How does software disablement of hard interrupts work in Linux? Also comment on how happens if the system's PIC does not support masking of individual interrupts. In particular, how is a hard interrupt deferred, and when and how is it eventually executed?