Sibling Rivalry

In 1991, Finnish programmer Linus Torvalds announced a new operating system to the world with this email:

“Hello everybody out there using minix -
I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torv...@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable [portable] (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.”

This home-brewed effort, originally called “Freax”, soon became “Linux” and swept the world, and today powers most of the world’s datacenters.

Meanwhile, a little known fact is that Linus had a number of twins (non identical), each of whom tried to create rival operating systems. Binus created **BeOS**; Dinus designed and implemented **DOS** (not Microsoft DOS, though); Minus invented **Minux**; and Zinus (the youngest) wrote the code for **ZOS** (all in one caffeine-fueled evening).

In this test, you will answer questions about each of these operating systems. Strangely, instead of focusing on all aspects of each OS, we will instead concentrate our evaluation upon CPU and memory virtualization. Hmm… I wonder why? In any case, good luck!

Facts:
- This exam contains **9** pages and just **32** questions.
- Each question **has only one answer. ANSWER ALL OF THEM!**
- **Guessing is OK; thinking and picking the right answer is preferred.**

PART I: *We begin with **BeOS** ("Be OS"), a minimal operating system. Each design decision in BeOS is meant to simplify. Why? Binus, well, she liked keeping things simple and to the point. Very direct, that Binus!*

1. BeOS keeps track of a small amount of information about each running process in a data structure called the "process list". However, to keep the amount of information in the process list small, Binus reduced the number of states a process could be in. Specifically, in BeOS, there are only two states a process can be in: **RUNNING** and **BLOCKED**. What process state found in most OSes was removed from BeOS?

- a STOPPED
- b READY
- c ACTIVE
- d WAITING
- e LOCKED

The other OS state is "READY" (or "RUNNABLE" as I just called it in class). Thus, B is the answer. The other terms are just kind of made up (STOPPED and WAITING could perhaps be names for BLOCKED, ACTIVE could be an alternate name for RUNNING, and LOCKED … well who knows what that means, exactly)

2. To keep scheduling simple in BeOS, Binus chose the **FIFO** policy (first in, first out) to decide which process to run. Which **one** of the following statements about FIFO scheduling is true?

- a It reduces average turnaround time
- b It reduces average response time
- c It suffers from the "convoy" problem
- d It is a preemptive scheduling policy
- e It is different from the FCFS policy

FIFO suffers from the convoy problem, where a single long job can make jobs behind it queue up and thus have worse response/turnaround times.

3. Assuming the BeOS FIFO scheduling approach, calculate the **turnaround time for job B**, assuming the following: job A arrives at time T=0, and needs to perform 10 seconds of compute; job B arrives at time T=5, and needs to perform 20 seconds of compute; job C arrives at time T=10, and needs to perform 5 seconds of compute. The jobs perform no I/O.

- a 10
- b 20
- c 30
- d 25
- e None of the above

Turn around for B is the time it ends minus the time it enters the system.

B arrives at T=5, but A still runs for 5 more seconds. Thus, B starts running at T=10. B then runs for 20 seconds. As such, B's turnaround is 30 - 5 = 25.

4. For simplicity, BeOS originally used a **"base and bounds"** approach to virtual memory. Which **one** of the following is true about this approach?

- a It translates addresses more slowly than most other approaches
- b It prevents internal fragmentation
- c It enables sharing of code between processes
- d It provides protection between processes
- e None of the above (all are false)

The bounds register is used for protection, and thus D.

It's not A because it is a fast approach;
It's not B because it doesn't prevent internal fragmentation (you could have a large address space with a big gap in the middle);
It's not C because it's hard to share code with base + bounds (easier with segmentation and paging).

5. Assume the following for the original version of BeOS: a 64-byte virtual address space, a system with 512 bytes of physical memory, a **base** register set to **10**, and a **bounds** register set to **20**. A process in BeOS wishes to access the following virtual memory addresses: 15, 6, 30, 21, 50. How many of these accesses are valid (legal) and thus will not trigger a fault?

- a 1
- b 2
- c 3
- d 4
- e 5

Bounds directly determines which virtual addresses are legal. With a bounds of X, addresses 0 through X-1 are all valid. Thus, in this case, 15 and 6. Thus, 2 are valid.

6. A later version of BeOS was developed for a newer machine and had a much larger virtual address space: **1 KB**. Assuming it is running on a system with 16 KB of physical memory, what is the largest virtual address a BeOS process can generate?

- a 1000
- b 1023
- c 1024
- d It depends on the base register
- e It depends on the bounds register

With a 1 KB virtual address space, a process can generate a reference to addresses 0 through 1023. Thus, 1023. Whether it's *legal* depends on the bounds, but the address can always be generated.

7. With a base and bounds registers in the MMU, BeOS must take care during process switching. In an early version of BeOS, it was so simple that the bounds register was never updated upon a process switch, instead using the first value — essentially, a random value — that was placed there. What could happen as a result? (choose **one** answer)

- a A process could access another process's memory
- b A process could fault even though it is accessing what should be valid memory
- c A process could access physical memory that isn't assigned to any process
- d A process could be prevented from accessing valid code
- e All of the above

Not updating the bounds would be a big problem, as each process has its own bounds. If it's essentially random, it could be "too big" (which could all you to access another process's memory, or physical memory not assigned to any process); if it's "too small", a fault could occur to a legal address or valid code may become inaccessible. Thus, E.

8. An earlier version of BeOS was so simple that it didn't use the base and bounds registers provided in hardware. Instead, when a process was loaded into memory, each of its address references was statically rewritten to match its location in physical memory. For example, if the program contained the instruction "load 10 into register R1", and the address space was loaded at physical address 100, the instruction would be rewritten to "load 110 into register R1". This approach (choose **one** answer):

- a Provides strong process isolation
- b Makes programs run slowly
- c Allows multiple processes to be active in the system
- d Affects the type of CPU scheduling policy you can implement
- e Helps realize a sparse virtual address space

Not having a bounds register and relying on software would not provide strong isolation (thus, not A); this seems actually like a fast approach as it just directly executes instructions after rewriting them (thus not B); this has nothing to do with scheduling (thus not D); this doesn't particularly help with realizing a sparse address space (it just does translation) (thus, not E). It does allow multiple processes, by loading them into different parts of physical memory. Thus, C. This is called "static relocation" btw, and is mentioned in the book.

PART II: *We now move on to **DOS**, designed by Dinus. DOS is based around the concept of **randomness**, because to be honest, Dinus was a pretty random kind of dude. You never knew what Dinus was up to!*

9. The DOS scheduler is based on a random policy. Specifically, every 10 milliseconds (ms), a timer interrupt goes off, and DOS picks a random process to run. This approach is most similar to which following policy?

- a First-in, First-out
- b Shortest Job First
- c Round Robin
- d Shortest Time To Completion First
- e Rank-order Scheduling

Picking a random choice every interrupt sounds a lot like round robin; indeed, with the "right" random choices, it can exactly become round robin. The choices A, B, and D are all "run to completion" approaches, usually, and thus not what we're looking for. Finally, "rank-order scheduling" is just made up.

10. With the DOS random scheduling policy, assume you have two jobs enter the system at the same time, each of which needs to run for 30 milliseconds. Assume again a 10 ms timer interrupt. What is the **best case average response time** for these two jobs?

- a 0 ms
- b 5 ms
- c 10 ms
- d 15 ms
- e None of the above

This question assumes that the timer interrupt timing is "aligned" with the start of the jobs; as such, the best case for response time would be job A runs immediately (response time: 0), and then, at the first tick (10 ms later), job B runs (response time: 10). Thus, average is 5.

11. With the DOS random scheduling policy, assume you have two jobs enter the system at the same time, each of which needs to run for 30 milliseconds. Assume again a 10 ms timer interrupt. What is the **worst case average turnaround time** for these two jobs?

- a 45 ms
- b 50 ms
- c 60 ms
- d 120 ms
- e None of the above

The worst turnaround occurs when each job runs for as long as possible. Thus, ABABAB, with each little chunk being 10 ms. This would mean the turnaround for A is 50, and B is 60, with average 55. Thus, E, none of the above.

12. DOS uses the MLFQ (multi-level feedback queue) scheduler. However, it changes some rules. The biggest change: new processes are added to a **random** queue (not the topmost one). What is the biggest effect this will have? (choose **one**)

- a Sometimes, short-running jobs won't get serviced quickly, thus decreasing interactivity
- b Over a long period of time, it will be unfair to long-running jobs
- c Jobs will generally finish more quickly
- d Jobs will be able to game the scheduler
- e None of the above

The idea with adding to the topmost queue is that a short (interactive) job could run quickly; thus, if you start at a lower queue, this isn't true (hence, A is the answer). Long-running jobs are treated more or less the same (round robin on the bottom queue), hence not B; jobs don't finish more quickly (hence not C); jobs don't have any new ability to game the scheduler (starting lower isn't some advantage), hence not D.

13. DOS uses paging. Before talking about how randomness was used, let's do a simple questions to make sure we understand it. In this system, the virtual address space size was 128 bytes (tiny!), and the page size was just 2 bytes. How many entries were in each page table? (assume a linear array)

- a 128
- b 64
- c 32
- d 16
- e None of the above

The number of virtual pages in a 128-byte virtual address space with 2-byte pages is: 128/2 or 64. This matches exactly the number of entries in a simple linear array page table. Thus, B.

14. Now assume the address space size is still 128 bytes, but the page size is 32 bytes. Here is the page table for a process, where the leftmost bit is the valid bit, and the rightmost 4 bits are the PFN.

0x8000000c
0x00000000
0x00000000
0x80000006

If 4 bits are needed for the PFN, how big is physical memory?

- a 128 bytes
- b 256 bytes
- c 512 bytes
- d 1024 bytes
- e None of the above

If each page is 32 (2^5) bytes, and there are 4 bits for the PFN, that means that there are 2^4 physical frames in the system; 2^4 frames * 2^5 bytes/frame = 2^9 bytes or 512 bytes.

15. Assume again that the address space size is 128 bytes, and the page size is 32 bytes. Using the page table above (Question 14), translate the virtual address 0x64.

- a 0x46
- b 0x04
- c 0xc4
- d 0x64
- e None of the above

0x64 in binary is 1100100. The top 2 bits (11) are the VPN; the bottom 5 (00100) are the offset. 11 tells us to use last entry in the page table. That entry contains 0x80000006. The top bit is valid, so we have a valid translation, and the physical frame is 0x6 (binary 0110). We concatenate the physical frame to the offset to get the physical address: 0 1100 0100. In hex, this is 0xc4.

16. When allocating a physical page to a process, DOS selects a **random** free physical page (instead of, for example, the first free one on a free list). This approach will:

- a Make the page table larger
- b Make physical memory more fragmented (externally)
- c Slow down address translation
- d Make heap allocation fail more often
- e None of the above

Choosing a "random" page in paging doesn't really change much about paging, and hence E. For example, memory is not externally fragmented (paging solves this issue); the page table is the same size (that is determined by the virtual address space size and page size); address translation is exactly the same; heap allocation is handled within an address space and thus not relevant.

PART III: *Next up is Minus, who created Minux. Minus, well, they really love system complexity, so Minux may indeed be the most complicated of the OSes. Let's analyze what Minus has created!*

17. Minux uses kernel mode, user mode, and a new mode called "supervisor" mode. When a user program runs in "supervisor" mode, it is still restricted (like user mode), but can do the following: change the length of the timer interrupt interval, including turning it off. Overall, would you say that supervisor mode (choose one):

- a Helps ensure streamlined round-robin scheduling
- b Ensures more efficient, application-aware timing interrupts
- c Better integrates scheduling and virtual memory mechanisms
- d Limits the OS's ability to retain control of the machine if user code runs in supervisor mode
- e None of the above

The answer is D, because if you could turn off the timer interrupt, you could prevent the OS from ever running again. The other answers don't really make sense.

18. Minux employs a (two-level) multi-level page table. Assume a 32-bit virtual address space, and 4 KB pages. Also assume each page table entry (PTE) is 4 bytes in size. How many PTEs fit onto one page?

- a 1
- b 32
- c 1024
- d 4096
- e None of the above

With a 4KB page, we just divide by the PTE size to get 4 KB / 4 bytes = 1024 entries on one page.

19. Assuming a two-level multi-level page table (32-bit virtual addresses, 4KB pages, 4-byte PTE size), what is the minimum number of valid virtual pages in an address space such that the multi-level page table becomes its maximum size?

- a 512
- b 1024
- c 2048
- d 4096
- e None of the above

To be at its maximum size, we must have a VALID entry on every page of the page table. In this page table, there are 1024 entries (2^10) per page of the page table (4 KB page / 4 byte PTE = 1024). We also know that there are 2^20 virtual pages, and thus the page table must have 2^20 entries, which means we have 2^10 pages of page tables (1024). If there are 1024 pages of the page table, each page must have one valid entry, and thus, there needs to be 1024 valid pages in the address space to make this page table use all of its pages, and become its max size.

20. Assuming a two-level multi-level page table (32-bit virtual addresses, 4KB pages, 4-byte PTE size), what is the minimum number of pages needed for the multi-level page table (including the page directory) when there are 1025 contiguous valid pages somewhere in the virtual address space?

- a 3
- b 4
- c 5
- d 1025
- e None of the above

A page of the page table has 1024 entries (4 KB page / 4 byte PTE = 1024). Thus, if there are 1025 contiguous valid pages, they must be pointed to by TWO pages of the page table. That, plus the ONE page directory gives the THREE page answer.

21. TLBs make hardware more complex, so Minux definitely uses one. The best description of what a TLB is as follows (choose one):

- a A memory to store page table entries
- b A collection of base/bounds pairs to help segmentation work quickly
- c An OS component to speed up translation
- d A hardware feature that ensures fair use of memory
- e An address-translation cache

The TLB is an address translation cache. It does not store page table entries (directly) and thus not A (though it has similar contents; it has nothing to do w/ base/bounds and thus not B; it is hardware not an OS component and thus not C; it has nothing to do w/ fair memory usage.

22. Assume the TLB, which has just four entries, has the following contents (numbers in the TLB are all decimal, and entries are all valid):

VPN 0 -> PFN 1
VPN 1 -> PFN 100
VPN 2 -> PFN 101
VPN 3 -> PFN 102

Assume this system has a 14-bit virtual address, and 4-KB pages. What virtual address will access the physical address 50 (decimal)?

- a 0x0032
- b 0x1023
- c 0x3012
- d 0x3132
- e None of the above

To access physical address 50, you need something that translates to the 0th physical frame. By looking at the page table, you can see there is no such entry. Thus, E.

23. When running on Minux, calculate the **hit rate** of the TLB assuming a process has 4-KB pages, and accesses every 128th byte on a series of pages. Assume the TLB begins empty, and ignore code accesses and just focus on this "strided" data access pattern.

- a Just about 99%
- b Just about 88%
- c Just about 50%
- d Just about 17%
- e None of the above

The intended meaning of every 128th byte was access bytes 0, 128, 256, 384, 512, … If you do this, you access 4KB/128 = 32 bytes on each page. The first would be a miss in the TLB, and the next 31 would be hits. Thus, the TLB hit rate would be 31/32, or about 97% Thus, E.

The other interpretation was that you access byte 128 on the first page, byte 128 on the second page, etc. This is all misses. The hit rate is thus 0%. The answer is also E.

24. The Minux scheduler uses a new scheduler called "highest process ID (PID) first" (i.e., the job with the highest PID always runs, and to completion). Assume job PID=1 arrives at time T=0, with length 10; PID=2 arrives at T=2, length=6; PID=3 arrives at T=4, with length 4. What is the **average response time** of this approach in this example?

- a 1
- b 2
- c 3
- d 4
- e None of the above

This question was corrected in the exam to "allow preemption". Thus, they would run like this: 11223333222211111111. However, the response time is just time(first run) - (time arrive), which in this example is 0. Thus, the answer is E.

PART IV: *Finally, ZOS, created by Zinus, is the most novel of the OSes. Zinus is quite creative! Each part of ZOS has some new ideas; are they good ideas? We'll see!*

25. To reduce the size of page tables, ZOS combines the idea of base/bounds and paging. A virtual address space is still chopped into pages. The page table is pointed to by the base register, and the bounds register holds the "size" of the page table (really, the max VPN that is valid, plus one). This approach (choose one):

- a Enables fast translation with only two extra memory references to fetch a PTE
- b Enables a compact page table, if you use the virtual address space carefully
- c Is always smaller than a linear page table
- d Supports a sparse address space while still also minimizing page table memory usage
- e None of the above

Having a base/bounds that points to a page table could allow for a very compact table, if the address space isn't sparsely used; hence, B. For example, the first page could be the only valid page, and thus the bounds would be set to 1 and the base would point to this small (1 entry) page table.

26. ZOS, as mentioned above (Question 25), combines base/bounds and paging. Assume the following: a 32-bit address space with 1-KB pages. Assume each page table entry (PTE) is 4 bytes. Assume there are 100 processes in the system. If each process uses only one virtual page, what is the **worst-case** total size of all of these page tables?

- a 16 MB
- b 160 MB
- c 1600 MB
- d 16 GB
- e None of the above

The worst case with this approach is that a full page table is still needed by each process, because it uses the last virtual page of the address space. As such, each process needs a 2^22 entry page table, which is 4 M entries * 4 bytes/PTE = 16 MB. If there are 100 processes, you need 1600 MB.

27. ZOS uses a new type of scheduler called a proportional-share scheduler. This scheduler makes sure each process gets a certain amount of CPU time, based on how many "tickets" it has. For example, if process A has 2 tickets, and process B has 1, A should get twice as much CPU time as B. Note that a process cannot change how many tickets it has, and all jobs only use the CPU (there is no I/O in this example). Which of the following traces (which each show which job was scheduled at each quantum over time) does **not** show the behavior of a proportional-share scheduler?

- a AABAABAABAABAA
- b ABABABABABAB
- c AAAAAAAAAABBBBB
- d ABBBABBBABBBABBB
- e None of the above (they all could be traces from a proportional share scheduler)

Really could be any of these. The first is A=2, B=1; the second A=1, B=1; the third is A=10, B=5; the last is A=1, B=3. Thus, answer is E, they all could be.

28. ZOS uses a new mechanism instead of timer interrupts. Instead of interrupting the CPU every so many milliseconds, the ZOS hardware is programmed to interrupt the CPU after every N TLB misses. How creative! As compared to the timer, this approach (choose one):

- a Is equally effective
- b Is faster to program
- c Is risky

This is risky, because if a program enters a tight endless loop that never suffers from TLB misses, it could potentially run forever.

- d Requires less memory
- e Requires virtual memory support

29. A later version of ZOS uses a different, clever approach to sharing the TLB among active processes. Assume the hardware does not have an address space identifier (or process identifier) field in the TLB. Instead of flushing the TLB when switching between processes, ZOS ensures that each process in the system uses different (unique) VPNs as compare to any other process. Which of the following is **not true** about this approach:

- a Allows for fully flexible use of the virtual address space by each process
- b Allows for faster context switching (as compared to the TLB flushing approach)
- c Allows for fully flexible use of physical memory
- d Allows sharing of code pages between processes
- e None of the above (all are true)

Because unique VPNs must be used per process, the approach clearly does NOT allow for fully flexible use of the virtual address space. Thus, A. B, C, and D are all true.

30. ZOS also later added support for "large" pages, a new and clever idea. Assume that in a given system, regular page size is usually 1 KB, and large pages are 1 MB. When possible, the OS uses large pages instead of a bunch of smaller ones (e.g., when there is a contiguous, aligned portion of the virtual address space in use). Why is using large pages a good idea?

- a They reduce system complexity
- b They speed up trap handling
- c They can increase TLB hit rates
- d They make physical memory allocation easier
- e None of the above

Large pages mean that for a contiguous range of the virtual address space, fewer TLB entries are needed, and, as such, can increase TLB hit rates. Large pages generally increase system complexity (more stuff to worry about); don't really change trap handling speed; and make physical memory allocation harder (have to be able to allocate both page sizes).

31. ZOS also added special hardware, in the form of general-purpose registers that only the kernel can use. Indeed, all kernel code has been written to only use these registers, not the regular (user-level) general purpose ones. Why might these registers for the kernel be a good idea?

- a Faster trapping into and returning from the kernel
- b Kernel code now easier to compile
- c Reduces need for context switching between processes
- d Now, no way to harm user-level register contents while in kernel code
- e None of the above

Kernel-only registers would mean that we don't have to save/restore user registers when trapping into the kernel/returning from kernel. As such, trapping is faster. This does not make the kernel easier to compile (perhaps harder, actually); it doesn't reduce the need to context switch (that is determined by scheduling policy); the kernel still could overwrite user registers if it wanted to.

32. Zinus also had this last question for you: which is **true** about operating systems?

- a They always make systems run faster
- b They always make systems use less memory and CPU
- c They generally make systems easier to use
- d They never crash
- e None of the above (these are all false)

As said on the first class day, OSes make systems easier to use. They don't always make things faster (think of additional work due to address translation, for example); they usually use more memory and CPU to do OS stuff; they do crash, despite our best intentions.