Do all questions.

1. [20 points] In the two-level model for implementing threads, user-level threads are executed in the context of kernel-level threads. There might well be more user-level threads than kernel-level threads; each user thread may run in the context of any kernel thread — there is no fixed mapping. In the one-level model, there is a one-to-one fixed mapping between user-level threads and kernel-level threads.

   a) [10 points] Describe a situation in which, if the two-level model is being used there is a deadlock, but if the one-level model were being used, there would be no deadlock. Be sure to explain why deadlock occurs in the two-level model but not in the one-level model. (Hint: consider Unix pipes, which are essentially an implementation of the producer-consumer problem in the kernel.)

   Assume we have one kernel thread and two user threads. Suppose the pipe is full and one user thread is attempting to write to the pipe, but is blocked. Not only is the user thread blocked, but so is the kernel thread in whose context it is running. The other user thread is available to read from the pipe, thereby unblocking the first user thread. However, since there is only one kernel thread and it is blocked, the other user thread cannot run and thus the system is deadlockd.

   b) [10 points] Describe a simple rule that could be implemented by the kernel to augment the two-level model so as to avoid this deadlock problem. (Hint: consider the deadlock situation you identified for part a. Can the kernel recognize that there’s a problem? What can it do about it?)

   The kernel might automatically create a new kernel thread for a process when all the process’s current kernel threads are blocked.

2. [15 points] Due to market pressure, the manufacturer of the Rhinopias disk drive has come out with a Rhinopias II drive that spins 33.33% faster than the original drive (now called the Rhinopias I). The specs for the new and old drives are as follows:

<table>
<thead>
<tr>
<th></th>
<th>Rhinopias II</th>
<th>Rhinopias I</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rotation speed</td>
<td>13,000 RPM (4 milliseconds/revolution)</td>
<td>10,000 RPM (6 milliseconds/revolution)</td>
</tr>
<tr>
<td>Number of surfaces</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Sector size</td>
<td>512</td>
<td>512</td>
</tr>
<tr>
<td>Sectors/track</td>
<td>500-1000 (750 average)</td>
<td>500-1000 (750 average)</td>
</tr>
<tr>
<td>Tracks/surface</td>
<td>100,000</td>
<td>100,000</td>
</tr>
<tr>
<td>Storage capacity</td>
<td>307.2 billion bytes</td>
<td>307.2 billion bytes</td>
</tr>
<tr>
<td>Average seek time</td>
<td>4 milliseconds</td>
<td>4 milliseconds</td>
</tr>
<tr>
<td>One-track seek time</td>
<td>.2 milliseconds</td>
<td>.2 milliseconds</td>
</tr>
<tr>
<td>Maximum seek time</td>
<td>10 milliseconds</td>
<td>10 milliseconds</td>
</tr>
</tbody>
</table>
a) [5 points] By what factor is the maximum (one-track) transfer speed of Rhinopias II faster than that of Rhinopias I? Explain.

Rhinopias II’s maximum (one-track) transfer speed is 1.3333 times faster than that of Rhinopias I. This is because the speed is the number of sectors that pass underneath a disk head per second.

b) [5 points] Assuming S5FS file systems are on both a new and an old disk drive, how much faster can we expect the average file to be read on the new drive than on the old?

On an S5FS file system, file-system blocks are scattered randomly throughout the disk. Thus the time to access each sector is the average seek time plus the average rotational latency (plus the average transfer time, but this is negligibly small). On Rhinopias II this average per-sector access time is 6 milliseconds (4 milliseconds average seek time plus 2 milliseconds average rotational latency). On Rhinopias I the average per-sector access time is 7 milliseconds. Thus the sustained throughput is 7/6 times faster on Rhinopias II.

c) [5 points] Assuming log-structured file systems are on both a new and an old disk drive, how much faster can 300KB of data be written on the new drive than on the old? Recall that in log-structured file systems, updates to files are written in contiguous disk locations (in the log).

300KB of data can fit on one track. Since in log-structured file systems data are written to consecutive sectors, the limiting factor in write speed is the disk’s rotation speed. Thus the data can be written 1.3333 times faster on the new drive than on the old.

3. [25 points] In many Unix-based systems (such as Linux and Weenix), a process’s address space is represented by a linked list of vmarea structures, each representing a separate piece of the address space and describing the access permissions and what has been mapped into the region of the address space. Such operating systems often employ “lazy evaluation” in which they postpone many operations in hopes of not having to do them.

a) [6 points] Assume the OS is running on a (32-bit) x86-based system, which employs a two-level page-translation mechanism with a top-level page-directory table that has $2^{10}$ entries that each refer to page tables with $2^{10}$ entries, each of which refer to pages of size $2^{12}$ bytes. Consider the life of a process, from its creation via fork to its termination via exit. What are the events that cause the page directory to be allocated and what are the events that cause the page tables to be allocated? In particular, what actions, such as system calls, memory references, etc., by threads within the process cause the page directory to be allocated and what are the events that cause page tables to be allocated? (Note this doesn’t happen all at once.)

The page-directory table is allocated either when the process is created, as part of the fork system call, or when it’s first needed to satisfy a memory reference. Page tables are allocated only when needed: when a thread in the process references a page that doesn’t belong to any of the current page tables.

b) [6 points] Recall that in 32-bit Linux, until recently all physical memory was directly mapped into the kernel’s address space. However, once it became feasible for the amount of physical memory in a machine to get close to $2^{30}$ bytes (one gigabyte), this was

---

1 What this means is that, for any real-memory address $r$ there is a virtual address $v$ in the kernel’s address space such that $M(v)=r$, and if $r+1$ is a legal real-memory address, then $M(v+1)=r+1$, where $M$ is the mapping from kernel virtual addresses to real-memory addresses.
no longer possible, and just the first gigabyte, minus a small amount, is directly mapped into the kernel. However, in 64-bit Linux, things are different: many, many gigabytes of physical memory can easily be directly mapped into the kernel’s address space. Assume that by Moore’s law the amount of physical memory one can afford doubles every two years, and assume one can now afford up to eight gigabytes. Will it still be possible to directly map all the physical memory one can afford at the end of the current century into the kernel address space of 64-bit Linux? (The last day of the current century is December 31, 2100.) Explain.

No. The end of the current century will occur in 85.5 years. To make the math slightly simpler, let’s round this down to 84 years. According to Moore’s law, the amount of primary memory we can afford will increase by a factor of $2^{84}$. Since we can currently afford $2^{33}$ bytes, we will then be able to afford $2^{35}$ bytes. Unless there are remarkable leaps in data-compression technology in the upcoming years, this amount of physical memory won’t fit into a 64-bit address space.

c) [6 points] In Linux each thread has a stack occupying contiguous locations in memory that may grow to be up to 2 MB in length. It is possible (it’s not done in Linux, and you don’t need to show how it’s done) to allocate an individual thread’s stack in a way so it doesn’t need to occupy contiguous locations in memory. Thus thread stacks could be allocated from the heap (the dynamic region) on demand as they grow. (The first page of the stack could be in one part of the address space, the next page allocated from another part of the address space, and so forth.) This requires more processor time than in the standard approach, so it’s not normally done. Explain why such an approach might be useful if one is to support thousands of concurrent threads on 32-bit Linux.

The problem is that there isn’t sufficient room in a $2^{32}$-byte address space to support thousands of threads in the same address space with a default (two-megabyte) stack size. However, many threads might get by with fairly small stacks, while just a few need large stacks. Thus by giving all threads small stacks initially, but allowing them to grow on demand, we are much more likely to fit thousands of the threads in a $2^{32}$-byte address space.

d) [7 points] We now want to support thousands of concurrent threads on 64-bit Linux (by “thousands”, let’s say we mean $2^{14}$). Can this be done without having to resort to the use of non-contiguous stacks? Explain. Be sure to pay attention to the cost of creating a thread’s stack: is it cheaper to create a 10-kilobyte stack than it is to create a two-megabyte stack (the default size in Linux)? Assume the architecture is x86-64, which is much like x86-32, except there are four levels of page tables rather than two.

Yes. If each has a stack of size $2^{21}$, we need $2^{35}$ bytes of address space to support them. But this is not a problem on 64-bit Linux. Creating a thread’s stack involves allocating both virtual memory and physical memory. The allocation of virtual memory simply requires setting up the appropriate vmarea structures, which would be the same whether creating a small stack or a large stack. There are clearly more page-table entries to set up for a large stack, but, as discussed in part a, allocating page tables and setting up their entries is done on demand, when stack pages are first referenced. Thus if a thread uses only a small portion of a large stack, the overhead for setting up the page-table resources is just that required for the stack portion used.

4. [20 points] A security problem popular in the late 1960s and early 1970s was the mutually suspicious users problem. User A has a proprietary program. User B has proprietary data. B wants to run A’s program on B’s data, but wants to make certain that A doesn’t get a copy of the data. A wants B to use A’s program (for a fee), but doesn’t want B to get a copy of it. The solution
is to set up a protection domain that has read access to B’s data, execute-only access to A’s code, write access to a solutions file that can be read only by B, and no other access rights.

Can such a protection domain be established in Unix? Explain. (Hint: consider the chroot system call, which allows a process to be restricted to a subtree of the directory hierarchy, and assume the problems with it that were discussed in class have been fixed. Also, your solution might take advantage of a trusted third party to set things up. You may assume that protection domains consist only of files — no direct communication between processes is possible and the internet does not exist.)

Yes. This solution requires the cooperation of a trusted third party, which could be root, but doesn’t have to be. An empty directory is created that is owned by user C, the trusted third party. C has rwx permission for it, B has execute-only permission, and A has no permissions. A gives C the pathname of an execute-only copy of its program; C puts a hard link to it in the directory. B gives C the pathname of the file containing B’s data. The file is not accessible by A; C creates a hard link to it in the directory. Similarly, B gives C the pathname of the file where B wants its output to go. The file is writable by B but is not accessible by A. Again, C creates a hard link to it in the directory. B additionally makes certain that the directories containing its data and output files are not accessible by A. B now runs a program (running as user ID B) that does a chroot to the directory, then execs A’s program. Since the program is running as B, it has access to the input and output files. Since these files are not accessible by A, A cannot view them. Since the subtree in which the program runs is not writable by B, even though B is running code provided by A, it cannot create any new files in the directory. Though A’s program (running as B) can change the permissions of the input and output files, it does A no good since it cannot access the directories referring to them (including the chroot directory).

5. [20 points] We’d like to virtualize a disk. The disk controller supports direct memory access (DMA) transfers between the disk and primary memory. Its interaction with the processor is via a set of registers that appear in the processor’s (real) address space. These registers include:

- disk address register (DAR): contains the address of the source or destination disk sector
- memory address register (MAR): contains the address of the source or destination buffer in primary memory
- length register (LR): contains the length of the transfer
- command and status register (CSR): contains the disk command (write-only) and the disk status (read-only)

To perform a transfer, the processor stores into the DAR the destination disk address for writes or the source disk address for reads. It stores into the MAR the source memory address for writes or the destination memory address for reads. It then stores the length of the transfer (in sectors) into LR. Finally, it checks the CSR to make sure the controller is ready, then stores either a WRITE or READ command into the CSR to initiate the transfer. The processor may then either check CSR to determine when the transfer completes, or, when it stores the transfer command (WRITE or READ), also set the interrupt-enable flag in CSR so that it is interrupted by the controller when the transfer completes.

a) [5 points] We are running a guest operating system in a virtual machine, such that the guest is oblivious to the fact that it’s running on a virtual machine rather than on a real machine. We would like its virtual machine to have sole access to the disk drive (i.e., the disk drive won’t be shared with other virtual machines). Explain what must be done by the virtual machine monitor (VMM) to make this happen. In particular, how are the controller’s registers made available to the virtual machine? What does the VMM do, if anything, when the guest OS puts commands into the controller registers and when it
reads from the controller registers? What happens when the disk controller interrupts the processor?

The VMM must protect the pages into which the controller’s registers are mapped on the virtual machine so that it is notified, via protection faults, of any attempt to access the registers by the virtual machine. When the VM does access these registers, the VMM must make sure the VM is in (virtual) privileged mode. If not, it causes a fault to occur on the VM. Otherwise it must translate the virtual real address the VM is putting in the MAR into a real address. When the controller interrupts the hardware, the VMM must pass the interrupt to the VM by emulating an interrupt on the VM.

b) [5 points] We would like to minimize the involvement of the VMM in dealing with the disk controller, in particular, the guest OS should be able to read and write the controller registers directly without the VMM’s having to do anything additional. Describe what hardware functionality would be required to make this happen. (Hint: we discussed this in one of the later lectures. It involves the mapping of addresses: consider which hardware unit is generating addresses and what needs to be done to these addresses.)

What is needed is an I/O memory management unit (IOMMU) that translates the virtual real addresses used by the disk controller (as given to it by the VM) into real addresses.

c) [5 points] Suppose now the disk device is to be shared among multiple virtual machines, such that each virtual machine is given private access to what appears to it to be a smaller version of the actual disk. (For example, the real disk might hold a terabyte of data, but each virtual machine sees a private disk holding 100 gigabytes.) Describe the VMM’s role, beyond that of part a, in making this happen.

The VMM now must translate the disk addresses stored into the DAR into the disk region assigned to the VM. It also must check the LR to make sure the VM isn’t accessing beyond the bounds of its disk region.

d) [5 points] Would it make sense for multiple virtual machines to completely share the entire disk? In other words, each virtual machine has access to the entire disk, but this access is through its device driver, which is the same device driver as in part a. Explain. (Note: this does not involve your answers to parts b and c.)

No. Since each VM would be unaware of what the other VM’s were doing with the disk, there would be uncoordinated access to the contents of the disk and hence total chaos.