1. [28%] We’ve decided to design a new system that doesn’t have interrupts. For each source of external events, such as devices and timers, a thread is dedicated to handling these events, i.e., there is one thread per event source. These threads all run at a higher priority than other threads, and threads handling higher-priority events have a higher scheduling priority than threads handling lower-priority events. In this problem we explore some of the consequences of this design decision. Assume we intend to have a fully preemptive kernel with a preemptive, priority-based scheduler. You may also assume the existence of a thread_switch function that saves the caller’s register context in its thread_t and restores the context of the thread that is to run next (chosen from the run queues) from its thread_t, idling, if necessary, until there is such a thread. You may also assume there is just a single processor.

   a. [7%] In an interrupt-based system, when an interrupt occurs, the current thread’s registers are saved on its kernel stack, the stack pointer is set to point to the kernel stack, and the instruction pointer is set to point to the handler for the interrupt. What does the interrupt handler do to switch from the interrupted thread to another thread? You may assume that what was interrupted was running at IPL 0. You need not discuss how the processor goes back to user mode from privileged mode.

   The interrupt handler, if it is about to return to IPL 0, simply calls thread_switch.

   b. [7%] In our new, interrupt-less system, when a timer event occurs, the timer thread should be scheduled to handle it. What must be done, in terms of saving and setting registers, for this to happen? Note that what is effectively happening is that the currently running thread is being forced to make a coroutine call to the timer thread. The expected answer is short.

   The effect must be as if the currently running thread had made a call thread_switch, which then chose the timer thread to run. Thus the currently running thread’s registers are saved in its thread_t and the registers of the timer thread are restored from its thread_t.

   c. [7%] If the timer thread determines that the time slice of the previously running thread has expired, what should it do to cause the next thread of the same priority to run? Assume there is a separate queue of runnable threads for each priority level.

   The timer thread must move the previously running thread’s thread_t to the end of its priority queue, then call thread_switch to give control to the next thread.

   d. [7%] Explain why, in an interrupt-based system, interrupt handlers may not, in general, lock mutexes. May event handlers (such as the handler of timer events) safely lock mutexes in the interrupt-less system?

   In an interrupt-based system, the interrupt handler runs on the interrupted-thread’s stack and thus the interrupted thread cannot resume execution until the interrupt handler returns. If the interrupt handler attempts to lock a mutex that’s currently locked by the interrupted thread, there would be a deadlock.

   In the interrupt-less system, interrupt handlers run on separate stacks and thus may safely lock mutexes without the danger of causing a deadlock with interrupted threads.
We would like to adapt stride scheduling for use with multithreaded processes. The intent is that each process would purchase a certain number of tickets, as in standard stride scheduling. These tickets would be used to determine the processor time to be allocated to the process as a whole. Within a process, the tickets would be distributed among the process’s threads. For example, a four-threaded process might purchase 10 tickets and distribute two tickets to each of three threads, and four tickets to the fourth thread. Another process, say a two-threaded process, might purchase six tickets, with one going to one of its threads and five to the other. In general, an n-threaded process must purchase at least n tickets so that all of its threads can get processor time. The scheduler, when deciding which thread to run, will first determine which process to select a thread from based on per-processor meters governed by the number of tickets held by each process. Then, given the process, it will decide which thread within the process to run based on per-thread meters governed by the number of tickets held by each thread within the process. Recall that a meter runs at a speed that is 1/n, where n is the number of tickets held. A process’s meter runs only when one of its threads is running; a thread’s meter runs only when that thread is running.

Assuming the two aforementioned processes are the only active processes, one will have a meter running at 1/10 the “fair” meter rate, and the other will have a meter running at 1/6 the “fair” rate. When the first (four-threaded) process is chosen to execute (i.e., when its meter has a lower value than the other process’s), whichever of its threads that has the lowest value on its meter will run. The process’s meter will be incremented by 1/10 of a clock tick, the thread’s meter by 1/n of a clock tick, where n is the number of tickets given to that thread. Thread and process meters are initialized to 1/n, where n is the number of tickets assigned to them.

a. Between the two processes (again, assuming there are no other active processes) there are 16 tickets outstanding. With normal stride scheduling, with only per-thread meters as explained in lecture, after 16 clock ticks the value of each thread’s meter would be increased by 1. Would this be different using the adapted stride scheduling described above? Explain.

After 16 clock ticks, the meters on the two processes would each have been increased by one, with the first having been chosen during 10 of those ticks and the other during 6. Thus the four-threaded process would have had 10 ticks for its threads to run. Its four-ticket thread would have run during four of those ticks, with its meter increasing by 1/4 tick each time. The other three threads would have each run during two of those ticks, with their meters increasing by 1/2 tick each time. And, similarly, the two-threaded process would have its five-ticket thread run during five of its six ticks, with its meter increasing by 1/5 tick each time, and the other thread running once, with its meter increasing by one. Thus the effect on each thread’s meter would be the same as with normal stride scheduling.

b. In normal stride scheduling, the thread with five tickets would run first. Which thread runs first with adapted stride scheduling?

Since the four-threaded process, with 10 tickets, is selected first, its four-ticket thread would be the first to run.

c. Suppose we make a further modification to stride scheduling, allowing each process to distribute any number of tickets to its threads, regardless of the number of tickets it, itself, has purchased. Thus, for example, our second (two-threaded) process might distribute one ticket to each of its threads, rather than distributing six tickets to them. After 16 clock ticks, how much actual processor time would each of these threads have run?
The two-threaded process, as before, would be selected for six of the 16 ticks. Since its threads had paid equal bribes and thus have meters running at identical rates, each would run for three ticks.

3. [20%] We have an operating system whose only file system is S5FS. Suppose we decide to employ a RAID level 4 or RAID level 5 disk system. Can S5FS take advantage of all the features of these systems? Explain. Which would be better: level 4 or level 5? Assume that the block size of S5FS is equal to the disk-sector size. Recall that in RAID level 4 one disk is dedicated as the parity disk, while in RAID level 5 parity information is spread across all disks. (Hint: recall that the features of RAID that are potentially available are: larger effective disk (n disks provide more usable space than one), per-request speed-up for a single (single-threaded) user, and system-wide speed-up for a collection of users.)

Using S5FS on either RAID level clearly has the advantage of a larger effective disk (with redundancy) than obtained using S5FS on a single disk. Since S5FS’s block size is equal to the sector size, there is no per-request speedup resulting from parallel transfers — each request can utilize only one disk. However, there can be a fair amount of “concurrency” — if there are multiple requests queued up, in general they can be spread across multiple disks. Thus the effective throughput is faster than it would be on a system with a single disk. The parity disk on a RAID level-4 system will be a bottleneck, but the approach taken in RAID level-5, spreading the parity information across all disks, will eliminate the bottleneck.

4. [25%] Recall that with redo journaling, new values are put in the journal before the transaction is committed. With undo journaling, old values are put in the journal before the transaction is committed.

   a. [12%] What is the advantage of undo journaling over redo journaling? (Hint: consider when data in the cache can be freed relative to when a transaction is committed.)

The advantage of undo journaling is that it allows cached file-system pages to be freed before transactions involving them commit: once the cached page is checkpointed, i.e., written to disk, updating the file system, the cache entry can be freed. If the system crashes before the commit, the undo entries in the journal are used to undo all the uncommitted changes.

   b. [13%] NTFS employs both redo and undo journaling. This, of course, requires twice as much space for the journal. What is the advantage to doing both, rather than just undo journaling? (Hint: consider crash recovery.)

Suppose the system crashes after the commit, but before checkpointing has completed. If only undo journaling is done, then the entire transaction must be undone, since no information is available to complete it. With redo journaling, the transaction can be completed during file-system recovery.
If you do all of the following correctly, you'll get an A regardless of how well you do on the first four problems. If you miss any of the following, your grade will be based solely on how well you do on the first four problems.

The names of some operating systems are rather boring and what they mean can be easily figured out. For example, it doesn’t take much thought to figure out that “OS/360” is the operating system for the IBM 360. But some names are more interesting and less easy to figure out. Explain the following names of operating systems.

5. **Multics**
   
   Multiplexed information and computing service. It was intended to be a utility, shared by a vast number of users, not unlike cloud services such as today’s AWS.

6. **Unix**
   
   Unix was initially developed by former Multics developers who were disillusioned by the grand scheme behind Multics. Thus they chose the name Unix (rather than Unics) to emphasize that it wasn’t such a large, multiplexed system.

7. **Linux**
   
   This was the system developed by Linus Torvalds and is much like Unix. Wouldn’t you try to blend your name into Unix if you were to develop such a system?

8. **OS/2**
   
   Ok, there had to be one easy one. This stands for “operating system 2”, brought to you by the same company that brought you the programming language “PL/1”. It was supposed to be the replacement for PC-DOS (which everyone but IBM called MS-DOS). It was a pretty reasonable system, but couldn’t compete with Windows NT.

9. **Tenex**
   
   It stands for “ten extended”, and was an advancement over the TOPS-10 operating system for the Digital PDP-10. It supported virtual memory, which was pretty much lacking in its predecessor. The system was popular for AI research. It was developed at BBN, but rights to it were purchased by Digital, who continued its development and called the newer version TOPS-20, which was referred to as Twenex by some.

10. **Aegis**
   
   This was the operating system of the Apollo computer, the first commercially available computer workstation (and which played a prominent role in the history of the Brown CS department). The name comes from Greek mythology and referred to something that provided physical protection.