GROUP PROBLEMS

1(a) Assume no clock interrupt hardware. The key challenge is how to limit a process’s CPU burst to a small enough time quantum to support timesharing.

**Non-compiler-based solution:** Before the scheduler selects the next process to run, it can itself make an I/O request to guarantee that an I/O interrupt will occur in the near future. With some careful experimental evaluation of the time needed for minimal I/O, it can try to make an I/O request that will cause an interrupt in as short as possible a time quantum (as close as possible to a typical scheduling time quantum for timesharing). This solution is elegant and ‘cheap’. Alternatively, the OS can create an I/O bound process of highest priority that is always run before any ‘regular’ user processes. The shortcoming of this solution is that time quanta are very small compared to time to service I/O (which often requires mechanical movement in the I/O device) so it is likely that interactive processes will be served as quickly as needed.

*Note:* common error is to not explicitly indicate what entity issues this I/O interrupt and when. Many solutions were too vague: “we,” “the OS” (when?), “the system,” etc.

**Compiler-based approaches:** The compiler is re-designed to modify user code and insert voluntary kernel calls to the OS to give up control. The kernel calls must be placed carefully since it is impossible to predict the runtime behavior of a program. Thus, every possible branch direction must be covered, and there must be a call within every loop. The compiler can do some analysis to guess how much execution time segments of code take and choose the insertion points to try to match a reasonable clock interrupt interval. Regardless, this method is not perfect and some processes may run for longer intervals than others; in addition, timesharing processes may not get the smooth round robin service they need.

Other possible compiler-based approaches: forcing divide by zero interrupt or forcing I/O interrupt to occur by writing to a temp file on disk. These ideas are OK.

(b) Next, do the same for a system without I/O interrupts. The key challenge is how to know when I/O is done.

Without I/O interrupts, the OS or the user process must poll to check to see if the I/O is complete. This was actually done in early computers before interrupt hardware was developed. To do the polling, every time there is a timer interrupt, the OS can check to see if there is outstanding I/O for this process, and check to see if that I/O is complete yet. (This information can be recorded in specific I/O management data structures.) When complete, the OS will then notify the user process and move it from blocked state to ready state.

(c) Then argue which option you would choose with the goal of providing the best performance for the user. What kind of situations would your OS perform well? What kind of situations would it perform poorly? Note: Remember that this is for a uniprocessor system and that user processes and the OS must take turns on the CPU.

**Solution:** Any answer is acceptable here with good arguments.

**Keep timer interrupt arguments:** Having the OS or a high priority I/O process issue “fake I/O requests” is cheaper, but there is the extra overhead. A compiler generated solution is too costly since (i) it requires all language compilers to be modified and (ii) all user code be augmented with this wasteful extra code. Desirable to keep the timer
interrupt in systems that support a lot of CPU-intensive applications (scientific computing, AI search algorithms, etc.)

Keep the I/O interrupt argument: polling is wasteful since it introduces overhead of OS checking to see if I/O is complete. However, it is straightforward and cheap and is used in real computer systems. Highly desirable to keep I/O interrupts in systems with lots of I/O bound jobs such as game consoles, interactive database systems, and any OS that supports a lot of user interaction.

2. Observing Processes Under Unix (20 pts)
Almost everyone got these correct. Hope you had fun snooping around. Email me with any ideas you have for a more exciting version of this problem that does not overload the system with renegade processes.

INDIVIDUAL PROBLEMS

3. Silbershatz Problems (20 pts)

2.8 As in all cases of modular design, designing an operating system in a modular way has several advantages. The system is easier to debug and modify because changes affect only limited sections of the system rather than touching all sections of the operating system. Information is kept only where it is needed and is accessible only within a defined and restricted area, so any bugs affecting that data must be limited to a specific module or layer.

2.19 Why is the separation of mechanism and policy desirable?
Answer: Mechanism and policy must be separate to ensure that systems are easy to modify. No two system installations are the same, so each installation may want to tune the operating system to suit its needs. With mechanism and policy separate, the policy may be changed at will while the mechanism stays unchanged. This arrangement provides a more flexible system.

3.6 Describe the differences among short-term, medium-term, and long-term scheduling.
Answer:
a. Short-term (CPU scheduler) — selects from jobs in memory those jobs that are ready to execute and allocates the CPU to them.
b. Medium-term — used especially with time-sharing systems as an intermediate scheduling level. A swapping scheme is implemented to remove partially run programs from memory and reinstate them later to continue where they left off.
c. Long-term (job scheduler) — determines which jobs are brought into memory for processing.

The primary difference is in the frequency of their execution. The short-term must select a new process quite often. Long-term is used much less often since it handles placing jobs in the system and may wait a while
for a job to finish before it admits another one.

3.7 Describe the actions taken by a kernel to context-switch between processes.

**Answer:** In general, the operating system must save the state of the currently running process and restore the state of the process scheduled to be run next. Saving the state of a process typically includes the values of all the CPU registers, value of the PC, CPU condition codes, in addition to memory allocation. Context switches must also perform many architecture-specific operations, including flushing data and instruction caches.

4.2 (1) User-level threads are unknown by the kernel, whereas the kernel is aware of kernel threads. (2) On systems using either M:1 or M:N mapping, user threads are scheduled by the thread library and the kernel schedules kernel threads. (3) Kernel threads need not be associated with a process whereas every user thread belongs to a process. Kernel threads are generally more expensive to maintain than user threads as they must be represented with a kernel data structure.

4. fork and exec (20 pts)

Almost everyone got this right except for one detail. It is necessary for psched to be passed the PID of the process that is being sent to a specific scheduler. Thus, since the parent is the only one who knows the child’s PID, it is the parent who has to call psched. Alternatively, the child could mypid=getpid(); psched(mypid,i); then execl("bin/parallel-code",i);

```c
main() {
    for (i = 1; i<= 100; i++) {
        pid1=fork();
        if (pid1 < 0) /* error */
        elseif (pid1 == 0 /* child */
            execl("bin/parallel-code", i); /* get the child process going */
        elseif (pid1 > 0 /* parent: I have the child's pid */
            printf("child PID = %d\n", i);
            psched(pid1, i); /* notify parallel scheduler to run */
        pid1 on processor i */
    }
}
```

More sensible to spawn threads since they all use the same code.