CS 390
Chapter 6 Homework Solutions

6.1 `In Section 6.4, we mentioned ...`

Modern computer systems contain a special clock chip, powered by a battery, that keeps track of the date and time. Many system calls need to use the date and time. For example, most file systems track the time a file was last accessed, so every file read from or write to a file requires the kernel to read the date and time.

Because accessing any hardware is an expensive operation requiring DMA setup and teardown, and at the minimum the processing of an interrupt, the kernel only reads the date and time from the clock chip when the system boots. After that, the time is stored in a variable in kernel memory, which the kernel updates periodically. It does this by updating the value each time a timer interrupt occurs. If interrupts were to be disabled for a long time, we might miss some timer interrupts, causing the kernel's internal clock variable to run slowly. These effects can be minimized by disabling interrupts for only a short period of time.

The timer chip is also used for scheduling. At every timer interrupt, most scheduling algorithms will determine if the executing process's time quantum is expired. If the kernel ignores timer interrupts, the scheduling algorithm will not have accurate data with which to make decisions.

6.2 `What is the meaning of ...`

A "wait" occurs when a process wants to delay its execution until some condition becomes true. One way to accomplish this is by a "busy wait", where a process repeatedly checks the condition in a tight loop. The busy wait ends when the value of the condition changes. Here is an example:

```
/* A busy wait. */
while (the_condition_i_care_about_is_false) {
   ;
}
```

The only other kind of wait in an operating system is an "idle wait". In this type of wait, a process is placed on a queue and does not use CPU cycles until some event occurs. It is the responsibility of the OS to remove the process from the queue when the event occurs. Processes in idle wait are found either on the ready queue or on the wait queue.

Instead of busy waiting, a process can request that it be placed in idle wait. It is then up to either another process or the kernel to monitor the condition, and arrange for the process to be removed from the queue when the condition changes.

Adding a process to a wait queue, and then waking it and moving it back to the ready queue will require some non-zero amount of CPU time. On a multiprocessor system where the wait is expected to be short, it may actually be more efficient for a process to busy wait than to idle wait. (See next question.)

6.3 `Explain why spinlocks are not ...`

Spinlocks are never appropriate for single-processor systems for two reasons:

1. A spinlock uses CPU cycles even though it performs no useful work.
2. The condition that will break the process out of the spinlock can be obtained only be executing another process. Since we only have one CPU, the condition cannot change while the spinlock is being executed.

On a multiprocessor system, it is sometimes more efficient to allow a process to busy-wait rather than to place the process on a wait queue.

Placing the process on the wait queue requires a context switch. Another context switch is required when the process is removed from the queue and placed on the ready queue. If the critical section that a process is waiting on is short, it may become available quickly. In this case, it may be faster and use fewer CPU cycles to simply execute a spinlock rather than incur the overhead of two context switches. Even if one process is using CPU cycles by spinning, the condition can still change.

It is up to the kernel designers to determine which critical sections in the kernel code are long enough to require wait queues, and which can be better protected by spinlocks.

Additional Questions

A The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1 share a Boolean array with two elements, and an `int` variable. The structure of process $P_i, i == 0$ or 1 is shown below. (The other process is $P_j, j == 1$ or 0.)

```
boolean flag[2] = {false, false};
int turn;

while (true) {

   /* entry section */
   flag[i] = true;

   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j) {
            ;
         }
         flag[i] = true;
      }
   }

   /* critical section */

   /* exit section */
   turn = j;
   flag[i] = false;

   /* remainder section */
}
```

B Imagine a system where the kernel implements synchronization primitives, and then allows processes to use these primitives via system calls. Explain why implementing these primitives by disabling interrupts is not appropriate.

C Describe two data structures in which race conditions are possible.

D Consider this code example that a kernel might execute during process creation and destruction. For example, the first segment of code might be executed when a process calls `fork()` to create another process. The second might be executed when a process calls the `exit()` system call.

```
#define MAX_PROCESSES 255
int n_processes = 0;

/* the implementation of fork() calls this function */
unsigned int allocate_process()
{
   if (n_processes == MAX_PROCESSES) {
      return -1;
   } else {
      /* allocate necessary process resources */
      unsigned int new_pid = n_processes++;
      return new_pid;
   }
}

void release_process()
{
   /* release process resources */
   --n_processes;
}
```

1. Identify the race condition(s) in this code.
2. Assume you have a mutex lock name `mutex` with the operations `acquire()` and `release()`. Indicate where the locking needs to be placed to prevent the race condition(s) you identified.

3. Assume the programming language you are implementing the kernel in has an atomic data type that guarantees all statements that manipulate a variable act atomically. Could we replace `int n_processes = 0;` with `atomic_t n_processes = 0;` to prevent the race condition(s)?

Answers to Additional Questions

A It's easier to understand the algorithm if you know the purpose of the shared variables. The condition `Flag[i] == TRUE` means P_i is trying to enter or has already entered its critical section. If `Turn == i` in the code, it means that it is P_i's turn to enter the critical section, should both processes try to enter concurrently. Let us consider two processes P_0 and P_1.

Mutual Exclusion Assume that P_0 is in the critical section. We must show that P_1 cannot enter the critical section.

If P_0 is in its critical section, then `flag[0] == TRUE`, so P_1 will loop on the outer loop `while (flag[0]) ...` until P_0 changes `flag[0])` when it leaves the critical section.

Progress We need to show that a process can not be blocked forever when it attempts to enter the critical section. Consider process P_0. Suppose P_0 wants to enter its critical section. If P_1 is in its remainder section, a hand trace of the code shows that P_0 can enter its critical section immediately.

Now suppose that P_0 is somehow blocked in the entry section. It can be blocked at two places: the inner loop of the entry condition (`while (turn == 1) ...`) or the outer loop (`while (flag[1]) ...`).

Suppose that it is blocked at the outer loop. If this is true, we must have `flag[1] == TRUE`. If `turn == 1`, then P_0 will reset its flag to `FALSE`, allowing P_1 into its critical section. When P_1 exits, it will set `turn` to 0 and `flag[1]` to FALSE, allowing P_0 to enter its critical section.

Bounded Waiting We will show the bounded waiting requirement by showing that, when both processes always want to enter their critical sections, the processes alternate. First, note that the

variable `turn` is only ever assigned to in the exit section: P_0 sets `turn` to 1, and P_1 sets `turn` to 0.

Suppose both processes begin to execute their entry sections, and `turn == 1`. Both `flag` variables will be true, and both processes will execute the body of the outer `while` loop. P_0 will find `turn == 1`, clear its `flag` and busy wait on `while (turn == 1)`. P_1, on the other hand, will see that `turn == 1`, and `flag[0] == false`, and will then enter its critical section. (This follows from our Progress argument above.)

After P_1 exits its critical section, it will set `turn = 0`. Any attempt by P_1 to "sneak back around" and enter its critical section immediately will be blocked by the `if (turn == 0) ...` statement. Thus, P_0 will eventually find `flag[1] == FALSE` and enter its critical section.

B Suppose such a synchronization primitive was available to a user-land program. A program calling a sync primitive would then be executing with interrupts turned off. If the CPU is ignoring DMA interrupts, data from IO devices can be lost. (Think about an arriving network packet stored in a buffer on a network card.) Even worse, ignoring timer interrupts means that the process would never enter the kernel involuntarily: the CPU-scheduler would never run, essentially allowing the process to monopolize the CPU until it turned interrupts back on.

C Any kernel data structure that is accessed by multiple processes and that cannot be accessed atomically is vulnerable to race conditions. It is easier to understand how a race condition could occur if you imagine multiple processes executing in kernel mode on a multiprocessor system.

Here are four data structures that could exist in a kernel and are vulnerable to races.

1. When a new process is created, the kernel reads the next process number from a kernel variable, and then increments the variable. What would happen if two processes read or wrote the variable at the same time?
2. When the timer goes off, the executing process's PCB is added to the ready queue. What would happen if two processes

attempted to manipulate the ready queue at the same time?

3. When a process makes an IO request, the process's PCB is added to the appropriate wait queue. What would happen if two processes attempted to manipulate the wait queue at the same time?
4. When DMA occurs, the kernel must allocate a chunk of free memory for the device to use during IO. Suppose that chunks of free memory are kept in a list. What would happen if two processes attempted to manipulate the free memory list at the same time?

D A kernel might contain code like this in order to prevent a user from creating too many processes (for example, to prevent a `fork()` bomb.)

1. `N_processes` is a shared variable that is manipulated by both segments of code. Since different processes may execute this code concurrently, it is subject to race conditions if not accessed in a critical section. In particular, imagine what might happen if multiple processes were to execute the `if` statement in `allocate_processes()` simultaneously.
2. A process should lock the mutex lock before it executes any segment of code where it needs exclusive access to variables. It should unlock the mutex at the end of that segment.

   In the example, `allocate_process()` should take the mutex lock before the `if` statement starts. Because there are multiple ways to leave the if statement, it should unlock the mutex at each termination: that is, just prior to the two return statements.

   `Release_process()` should take the lock before decrementing `number_of_processes` and release it afterwards.
3. This idea sounds promising, but unfortunately, like many ideas that sound good when you first hear them (Twitter, FaceBook, the Windows phone, going out for $1 pints on a Sunday night, etc.) it doesn't work out in practice.

   The reason the race can occur is because `n_processes` is tested in `allocate_processes()`, and then, depending on the value of that test, updated later. What we really need is to make the test

and the update of the variable execute with a guarantee that no other process can intervene between those two actions.

This type of guarantee, where two separate actions are combined into one indivisible action, is referred to as an "atomic transaction". Such actions are very important in databases, fault-tolerant computing and distributed algorithms. All atomic transactions are constructed using the concept of a mutex lock as a basic building block.