Stephen Chambers
Elizabeth Varki
CS 620

Homework 4

1. (20) The following is a variation of the critical section codes described in class. (Initially, flag[0] and flag[1] are both false.)

```
THREAD 0                          THREAD 1
for(;;)                           for(;;)
{                                 {
 <initial section>                 <initial section>
 flag[0]=true;                   while (flag[0]==true);
 while (flag[1] == true);          flag[1]=true;
 <critical section>                <critical section>
 flag[0]=false;                  flag[1]=false;
 <remainder section>             <remainder section>
}                                 }
```

Notice that the codes for the two threads are asymmetric. For the questions given below, justify your answer completely. Regarding this code:

(a) (4) Is mutual exclusion of the critical section guaranteed?

Yes, this code is mutually exclusive. If a thread is interrupted in the middle of its critical section, its corresponding flag variable is already set to true so that the other thread cannot run. Furthermore, the flag variable is only set to false after the critical section is complete.

(b) (4) Can Thread 0 starve out Thread 1?

Yes, Thread 0 could starve Thread 1 if it was interrupted anywhere but the remainder or initial sections. This is because Thread 1 sets its flag variable to true *after* the while loop. This means that Thread1 could be spinlocking while Thread 0 continuously uses the CPU.

(c) (4) Can Thread 1 starve out Thread 0?

No, Thread 1 cannot starve out Thread 0. The only way this would be possible is if Thread 1 was interrupted during the critical section, something that should happen anyway by the design of mutual exclusion. . This is because Thread 0 sets its flag variable to true before spinlocking and conceding the CPU to Thread 1. This serves as a protection; Thread1 would execute, eventually hit the while loop, and concede the CPU back to Thread 0 because of that protection put in place by Thread 0.

(d) (4) Is deadlock possible?

No, because the flag is set on either side of the while loop spin lock. If Thread 0 enters first, it will immediately set flag[0] to true, allowing it to execute. There is not a situation where Thread 0 would not reach the statement setting flag[0] to false, even though it may be interrupted before getting there. If Thread1 started first, it would spinlock until Thread 0 started executing, which would the result in the same logical reasoning described earlier. The key point is that flag[0] and flag[1] cannot be true at the same time due to the design of this code.

(e) (4) If a thread dies in its remainder section, can it block the other thread?

No, because it ensures the other thread can execute immediately before entering the remainder section by setting its own flag to false.

2. (16) You are not permitted to look-up a solution to the Baker's problem on-line (or anywhere else).

The Bakers Problem:

Consider a system with three baker processes and one chef process. Each baker continuously kneads the dough and then bakes bread. But to knead and bake bread, the baker needs three ingredients: flour, eggs, and yeast. One of the baker processes has flour, another has eggs, and the third has yeast. The chef has an infinite supply of all three ingredients. The chef places two of the ingredients on the table. The baker who has the remaining ingredient then kneads and bakes his bread signaling the chef on completion. The chef then puts out another two of the three ingredients, and the cycle repeats. Write the pseudocode to simulate this system using semaphores. You should write four processes: the chef process and the three baker processes. Note: the three bakers can be represented by a single process with a parameter value to indicate which ingredient the baker holds, e.g., baker(i) where i can be flour, eggs, or yeast.

Every baker has one resource. The baker waits on its respective semaphore until the chef tells him to start making bread. Once a baker is signaled, it makes the bread and signals the chef. The chef then arbitrarily picks which baker to go next. Initializing the three bakers to 0 ensures that only one will run at a time. Initializing the chef semaphore to 1 ensures the chef will free up resources before a baker begins making bread.

Note that every process (baker1, baker2, baker3, and the chef) is encapsulated in a while(1){} loop (an infinite loop).

```
Semaphore flourBaker = 0;
Semaphore eggsBaker = 0;
Semaphore yeastBaker = 0;
Semaphore chef = 1;
int curBaker;
```

baker1

```
p(flourBaker);
makeBread();
v(chef);
```

baker2

```
    p(eggsBaker);
makeBread();
v(chef);
```

baker3

```
        p(yeastBaker);
        makeBread();
 v(chef);
```

Chef

```
Randomly assign curBaker to value between 1 and 3.
p(chef)
if(curBaker == 1){
  setUpResources(flourBaker);
  v(flourBaker);}
else if(curBaker == 2){
  setUpResources(eggsBaker);
  v(eggsBaker);}
else if(curBaker == 3){
  setUpResources(yeastBaker);
  v(yeastBaker);}
```

3. (15)
r1 = 10 / a;
r2 = r1 * b;
r3 = e + f;
r4 = c * d;
r5 = r1 - r2;
r6 = 1 - r3;
r7 = r3 * r6;
r8 = 1 / r4;
r9 = r5 / r7;
r10 = r7 -r8;
r11 = r9 * r10;
r12 = r10 * r11;

In the above computation, the lower-case letters refer to memory variables. Assume that each of the above steps takes 1 time unit to complete. Hence, the entire computation will take 12 time units on a single CPU. Suppose you were running this code on a 2 processor machine and you decide to split the above code into 2 threads that can be run in parallel on the 2 processors. For example, r1 = 10 / a; and r3 = e + f; can be executed in parallel since these 2 commands do not affect each other. The goal is to execute the above code in the least possible time on 2 processors. The threads on the 2 CPUs need to be synchronized. For example, if Thread1 contains the code r4 = c * d; and Thread2 contains the code r8 = 1 / r4, you must ensure that r8=1 /r4 executes after r4=c * d. Use semaphores to synchronize the execution of the code on 2 processors. Assume that the P() and V() operations take 0 time units to complete (i.e., they are very fast operations compared to the computation time).
Semaphore x = 0;
Semaphore y = 0;

Thread 1	**Thread 2**
r1 = 10/a	r3 = e+f
r2 = r1 * b	r6 = 1-r3
r5 = r1 – r2	r7 = r3 * r6
p(x)	r4 = c*d
r8 = 1/r4	v(x)
v(y)	p(y)
r9 = r5/r7	r10 = r7 – r8
p(y)	v(y)
r11 = r9 * r10	p(x)
v(x)	r12 = r10*r11

(a) What is the least number of time units required to run the computation on 2 CPUs?
The least number of time units required to run the computation on 2 CPUs is 7 time units.

Time Unit	Registers Executed
1	r1, r3
2	r2, r6
3	r5, r7
4	r4
5	r9, r10
6	r11
7	r12

(b) What is the minimum number of semaphore variables required to run the above code on 2 CPUs?
The minimum number of semaphore variables required to run the above code on 2 CPUs is 2.

Semaphore x = 0;
Semaphore y = 0;
Semaphore z = 0;

Thread 1	Thread 2	Thread 3
r1 = 10/a	r3 = e+f	r4 = c*d
r2 = r1 * b	r6 = 1-r3	r8 = 1/r4
r5 = r1 – r2	r7 = r3*r6	p(x)
p(y)	v(x)	r10 = r7-r8
r9 = r5/r7	v(y)	p(x)
v(x)	p(z)	r11 = r9*r10
	r12 = r10*11	v(z)

(c) What is the least number of time units required to run the computation on 3 CPUs?
The least number of time units required to run the computation on 2 CPUs is 6 time units.

Time Unit	Registers Executed
1	r1, r2, r4
2	r2, r6, r8
3	r5, r7
4	r9, r10
5	r11
6	r12

(d) What is the minimum number of semaphore variables required to run the above code on 3 CPUs?
The minimum number of semaphore variables required to run the above code on 2 CPUs is 3.

4. (15) Consider the following synchronization problem: thread UNH adds 7 to a shared variable SCORE; then thread UM and thread BU must both read the value of SCORE (but they are allowed to read SCORE in any order); then after UM and BU read SCORE, thread UNH adds 3 to SCORE and then both BU and UM read the value of SCORE in any order. The entire process repeats. Give the code for these three threads, using semaphores, to guarantee the above sequence of events.

Semaphore write1 = 0;
Semaphore write2 = 0;
Semaphore read = 0;

Note that every process (UNH, BU, UM) is encapsulated in a while(1){} loop (an infinite loop).

UNH	**BU**	**UM**
Score += 7	p(read)	p(read)
v(read)	read(score)	read(score)
v(read)	v(write1)	v(write2)
p(write1)	p(read)	p(read)
p(write2)	read(score)	read(score)
Score += 3	v(write1)	v(write2)
v(read)		
v(read)		
p(write1)		
p(write2)		

5. (4) The semaphore P() operation is a blocking wait function, but it is implemented using the busy-wait hardware atomic function. Is this an inconsistency? Does this lower the efficiency of semaphores?

Yes, this lowers the efficiency of semaphores. A normal semaphore takes a blocked thread off of the ready queue and into a blocked queue. This means that the blocked thread does not get any time on the CPU. On the contrary, this example makes the thread spinlock on the CPU, wasting valuable resources. Assuming a standard scheduling policy, the OS will periodically switch to this process and give it a certain amount of time to run, and it will use this time to spinlock when the CPU could have been being used to process something else.

6. (6) Consider the following parallel program:

Thread x:	**Thread y:**	**Thread z:**
for(;;)	**for(;;)**	**for(;;)**
{	**{**	**{**
P(x)	**P(y)**	**P(z)**
P(x)		**P(z)**
P(x)		
write "x"	**write "y"**	**write "z"**
V(y)	**V(z)**	**V(x)**
V(y)	**V(x)**	
}	**}**	**}**

Initially, x = 3; y = z = 0. Describe the output format of this program.

Output*:* xyyzxyyzxyyzxyyz…..

The above output goes on infinitely. When the program starts, thread y and z are blocked due to their semaphores being initialized to 0. Thread x outputs 'x' because its semaphore starts at 3, and it is only decreased to 0 before writing 'x', allowing it to continue. It then unlocks Thread Y, which prints 'y'. Thread Y then unlocks Thread Z once, but Thread Z blocks twice. Therefore, it is impossible for Thread Z to print. Thread Y runs one additional time because Thread X unlocks it twice. The fact that Thread Y runs twice frees up z to run, and it prints 'z' and unlocks Thread X for the third time, allowing it to execute and repeat the process.

7. (6) Consider the following parallel program:

Thread x:	***Thread* y:**	**Thread z:**
for(;;)	**for(;;)**	**for(;;)**
{	**{**	**{**
P(x)	**P(y)**	**P(z)**
P(x)		**P(z)**
P(x)		
write "x"	**write "y"**	**write "z"**
V(y)	**V(z)**	**V(x)**
V(y)	**V(x)**	**P(z)**
V(z)		
}	**}**	**}**

Initially, x = 3; y = z = 0. Describe the output format of this program.

Output: xy(yz || zy) xy(yz || zy)…

When the program starts, thread y and z are blocked due to their semaphores being initialized to 0. Thread x outputs 'x' because its semaphore starts at 3, and it is only decreased to 0 before writing 'x', allowing it to continue. Thread X then unlocks Thread Z once. It then unlocks Thread Y, which prints 'y'. At this point, the output of the program is ambiguous. Thread Y can run more time because its semaphore value is 1, and Thread Z is also unlocked to execute. It is not known whether Thread Y will execute and print 'y' or if Thread Z will execute and print 'z' because the commands can happen in any order due to context switching. Therefore, there will always be either 'yz' or 'zy' after a 'xy'. This arbitrary "pattern" will repeat infinitely.

8. (3) Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

Interrupts are a critical part of a single-processor system, and disabling them will tamper with OS scheduling policies. It is possible to starve other processes by not allowing the CPU to context switch between processes; the running process would have full control of the CPU.

9. (3) Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

If synchronization is achieved by disabling interrupts in a program, mutual exclusion is not achieved. This is because interrupts will only be disabled on the particular processor that the program is running on. Other processers may execute the program's critical section and thereby prove mutual exclusion to be false.

10. (6) Servers can be designed to limit the number of open connections. For example, a server may wish to have only N socket connections at any point in time. As soon as N connections are made, the server will not accept another incoming connection until an existing connection is released. Explain how semaphores can be used by a server to limit the number of concurrent connections.

Semaphores can easily be used by a server to limit the amount of concurrent connections made by clients. If a semaphore is initialized to N, every client that connects the server would start by calling p() on the semaphore, decrementing it until it reached 0. When the semaphore is at 0, there are N clients connected and the server would not accept another incoming connection until a client called v() on the semaphore, or released its connection.

11. (6) Consider the code example for allocating and releasing processes shown below:

```
#define MAX_PROCESSES 255
int number_of_processes = 0;
/* the implementation of fork() calls this function */
int allocate_process() {
int new_pid;
if (number_of_processes == MAX_PROCESSES)
return -1;
else {
 /* allocate necessary process resources */
 ++number_of_processes;
 return new_pid;
 }
}
/*the implementation of exit() calls this function */
void release_process() {
/* release process resources */
--number_of_processes;
}
```

(a) Identify the race condition(s).

There are two race conditions associated with this problem.

1) The first is that a thread being created could be trying to compare the number_of_processes variable to MAX_PROCESSES in allocate_process() while another process is exiting and trying to decrement number_of_processes in release_process().
2) The second is that a thread could be trying to increases the number_of_processes by one in allocate_process() at the same time as another thread trying to decrement the number_of_processes variable by 1 in release_process().

(b) prevent the race condition using semaphores.

The race condition can be prevented by prevented using semaphores by initializing the semaphore to 1 and adding a p() operation at the beginning and a v() operationat the end of both the allocate_process() and release_process() functions. This locks down the shared variable so that neither race condition can occur.

```
#define MAX_PROCESSES 255
semaphore x = 1;
int number_of_processes = 0;
/* the implementation of fork() calls this function */
int allocate_process() {
int new_pid;
p(x)
if (number_of_processes == MAX_PROCESSES)
        return -1;
else
{
        /* allocate necessary process resources */
        ++number_of_processes;
        return new_pid;
}
v(x)
}
/*the implementation of exit() calls this function */
void release-process() {
/* release process resources */
p(x)
--number_of_processes;
v(x);
}
```