CS-323: OS Exercise Week 10 (Fall 2023)

November 22, 2023

Last name, first name: _____

1 Condition Variables

(a) Consider the following code in which we (the parent/main thread) want to wait for a particular condition (for e.g. the child thread is done). What is the advantage of using a condition variable (CV) over a shared variable as in the code below?

```
1   volatile int done = 0;
2
3   void* child(void*arg) {
4     printf("child\n");
5     done = 1;
6     return NULL;
7   }
8
9   int main(void) {
10    printf("parent: begin\n");
11    pthread_t c;
12    pthread_create(&c, NULL, child, NULL); // create child
13    while (done == 0)
14      ; // spin
15    printf("parent: end\n");
16    return 0;
17  }
```

In this case, shared variables are a possible, though inefficient, solution. When the parent spins on the lock, it wastes CPU cyles. When using a CV, the parent thread will be put to sleep until the condition (e.g., the child is done executing) becomes true. In the meantime, the OS can schedule a different thread to execute on the core, and the CPU cycles can be used for useful work instead of spinning.

Note: There are cases where the approach of using a shared variable might just be incorrect. Why?
Reason 1: Compiler optimization. The compiler might not see the done variable being modified (in this case, we purposefully declare it as volatile to eliminate this possibility) and eliminate the check in the while loop. Some memory accesses might also be re-ordered by the compiler.
Reason 2: CPU memory models (optional, taught in multiprocessor architecture). The visibility of a shared variable that enforces no constraints on memory can be complicated due to cache coherence and flushes. Changes made to variables can be seen from another thread in an unexpected order, or even never under the correct circumstances. Using locks and/or atomic instructions will prevent such issues.

Name: _____

(b) What is a condition variable?

A condition variable is an explicit sleeping queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired (by waiting on the condition). Some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition).

(c) What is wrong with the following code snippet? Answer with an example of what can go wrong.

```
1   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
2   pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
3
4   void thr_exit() {
5     pthread_mutex_lock(&m);
6     pthread_cond_signal(&c);
7     pthread_mutex_unlock(&m);
8   }
9
10  void thr_join() {
11    pthread_mutex_lock(&m);
12    pthread_cond_wait(&c, &m);
13    pthread_mutex_unlock(&m);
14  }
15
16  void* child(void*arg) {
17    printf("child\n");
18    thr_exit();
19    return NULL;
20  }
21
22  int main(void) {
23    printf("parent: begin\n");
24    pthread_t p;
25    pthread_create(&p, NULL, child, NULL);
26    thr_join();
27    printf("parent: end\n");
28    return 0;
29  }
```

Suppose the child thread executes the entire `thr_exit` function before the main thread enters `thr_join`. The child will lock the mutex, signal the CV (which has no waiting threads) and then unlock the mutex. Later, when the parent thread calls wait on the CV, it will block forever. This is known as a lost wakeup. This is the reason why a variable with a value needs to be tied to the condition variable, set by the child and checked by the main thread before going to sleep. Only using the condition variable (the queue) itself is not enough for proper synchronization.

(d) Now, consider the code snippet bellow. What problem could occur here?

```
1
2   // Note that this example is not "real" code, because the call to
3   // pthread_cond_wait() always requires a mutex as well as a condiiton
4   // variable. For the sake of this example, here,
5   // we just pretend that the interface does not ask for the mutex.
6
7   int done  = 0;
8   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
9   pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
10
```

Name: _____

```
11      void thr_exit() {
12        done = 1;
13        pthread_cond_signal(&c);
14      }
15
16      void thr_join() {
17        if (done == 0)
18          pthread_cond_wait(&c);
19      }
20
21      void* child(void*arg) {
22        printf("child\n");
23        thr_exit();
24        return NULL;
25      }
26
27      int main(void) {
28        printf("parent: begin\n");
29        pthread_t p;
30        pthread_create(&p, NULL, child, NULL);
31        thr_join();
32        printf("parent: end\n");
33        return 0;
34      }
35
```

There is a race condition in the above code where the parent might see the variable done as 0, but miss the signal by the child thread. For this race condition, the parent has to do the check in line 15 *before* the child sets the done variable, but call the wait function after the child signals the condition. This interleaving of execution is illustrated below.

Thread 0 (parent)	Thread 1 (child)
if(done == 0)	
----------	----------
	done = 1
	pthread_cond_signal()
----------	----------
pthread_cond_wait()	

This is why a lock needs to be held when checking the variable, and passed to the wait function to be released atomically when going to sleep.

(e) Considering the case with a single producer and a single consumer, does the following code work?

```
1
2       int buffer;
3       int count = 0;      // initially, empty
4       int loops;
5       cond_t  cond;
6       mutex_t mutex;
7
8       void put(int value) {
9         assert(count == 0);
10        count = 1;
11        buffer = value;
12      }
```

Name: _____

```
13
14      int get() {
15        assert(count == 1);
16        count = 0;
17        return buffer;
18      }
19
20      void* producer(void* arg) {
21        int i;
22        for (i = 0; i < loops; i++) {
23          pthread_mutex_lock(&mutex);           // p1
24          if (count == 1)                       // p2
25            pthread_cond_wait(&cond, &mutex);   // p3
26          put(i);                               // p4
27          pthread_cond_signal(&cond);           // p5
28          pthread_mutex_unlock(&mutex);         // p6
29        }
30      }
31
32      void* consumer(void* arg) {
33        int i;
34        for (i = 0; i < loops; i++) {
35          pthread_mutex_lock(&mutex);           // c1
36          if (count == 0)                       // c2
37            pthread_cond_wait(&cond, &mutex);   // c3
38          int tmp = get();                      // c4
39          pthread_cond_signal(&cond);           // c5
40          pthread_mutex_unlock(&mutex);         // c6
41          printf("%d\n", tmp);
42        }
43      }
```

Yes. This example prevents the race condition from the previous example by not allowing one thread to check the count variable (and take actions based on that) while the other thread is in a critical region which modifies the same variable. It does this by using the mutex.

Consider the producer, for example. When it is checking the value of count (line p2), it holds the mutex. Therefore, it is sure that the count will not change by the time it waits on the CV (line p3). Similar analysis assures us that the value of count does not change between lines c2 and c3, preventing the consumer from incorrectly waiting forever.

Another consideration is during the functions put and get. During both, we are assured that there is no concurrent running thread because the producer/consumer is holding the mutex (from lines c3/p3 respectively) before calling get/put.

(f) Let us continue with the code from the example above. Now, assume that we have two consumer threads (Tc1 and Tc2) and one producer thread (Tp). All the threads run on a single CPU, and the figure bellow shows the actions each thread takes, as well as its scheduler state (Ready, Running, or Sleeping). Please fill in the count value in the table bellow based on the provided thread trace.

Name: _____

Tc1	State	Tc2	State	Tp	State	Count
c1	Run		Ready		Ready	0
c2	Run		Ready		Ready	0
c3	Sleep		Ready		Ready	0
	Sleep		Ready	p1	Run	0
	Sleep		Ready	p2	Run	0
	Sleep		Ready	p4	Run	1
	Ready		Ready	p5	Run	1
	Ready		Ready	p6	Run	1
	Ready		Ready	p1	Run	1
	Ready		Ready	p2	Run	1
	Ready	c1	Run	p3	Sleep	1
	Ready	c2	Run		Sleep	1
	Ready	c4	Run		Sleep	0
	Ready	c5	Run		Sleep	0
	Ready	c6	Run		Ready	0
c4	Run		Ready		Ready	0

(g) What causes the problem in previous question? What would you change to solve it?

Tc1 is woken up as if the count was 1 the end of the previous answer, even though the value has already been consumed by Tc2 in the meantime.

This issue arises due to an issue in the semantics of a CV (discussed in the next question). In short, waking the thread Tc1 does not assure us that it will run immediately. In fact, in the trace above, Tc2 is able to run first and steal the produced item.

The problem in this example is based on the use of if statement before the wait. As we can see from the text of the previous question, there are two consumers (Tc1 and Tc2) and one producer (Tp). Consider the following sequence of steps:

1. A consumer (Tc1) runs first; it acquires the lock (c1), checks if buffer is ready for consumption (c2), and finding that it is not, waits (c3) (which releases the lock)
2. The producer (Tp) runs. It acquires the lock (p1), checks if buffer is full (p2), and finding that it is not the case, goes ahead and fills the buffer (p4).
3. The producer then signals that a buffer has been filled (p5). Critically, this moves the first consumer (Tc1) from sleeping on a condition variable to the ready queue; Tc1 is now able to run (but not yet running).
4. The producer then continues until realizing the buffer is full, at which point it sleeps (p6, p1-p3).
5. Another consumer (Tc2) sneaks in and consumes the existing value in the buffer before Tc1 is scheduled (c1, c2, c4, c5, c6, skiping the wait at c3 because the buffer is full).
6. Tc1 runs: just before returning from the wait, it re-acquires the lock and then returns. It then calls get() (c4), but there is no buffer to consume!

The code does not function as desired. The problem: there is no guarantee that the woken thread will run immediately after being woken. In the end, Tc1 was woken up even though the condition has not changed in Tc1's point of view (still empty after being woken up). These are called spurious wakeups. We need a way to prevend Tc1 from trying to consume and put it back to sleep because Tc2 snuck in and consumed the value in the buffer that had been produced, which means the buffer is empty again! Solution: always use while loops instead of if statements when waiting on condition variable. With a while loop, Tc1 will safely go back to sleep after checking the variable again. Do note that this works because it is holding the mutex once woken up and can check the value safely.

Condition variables that work in this way are said to have Mesa semantics. Mesa semantics does not guarantee that the woken thread will run immediately after it has been woken. Another type is Hoare semantics, which immediately switches execution to the thread being woken up and prevents this issue. However, Mesa semantics are much more common in practice due to the added complexity to make Hoare semantics work.

Refer to the book, chapter 30 (Condition Variables) of OSTEP for more details.

Name: _____

(h) What does `pthread_cond_broadcast()` signaling do?

It wakes up *all* waiting threads. In contrast, `pthread_cond_signal()` only wakes up *one* waiting thread.

(i) What are the advantages and disadvantages of broadcast signaling?

Advantages: guarantees all the threads will be woken at once, without needing to call signal a set number of times. Disadvantages: can have a negative performance impact, as we might needlessly wake up many other waiting threads that don't yet need to be awake. Those threads will simply wake up, re-check the condition, and then go immediately back to sleep, wasting CPU cycles. Broadcasting is therefore more suited to a case when other threads are waiting on the same signal or in more nuanced cases, rather than something they can *consume*.

2 Semaphores

(a) The concept of semaphores comes from railways, where they are used to prevent accidents due to multiple trains using the same track. In computer science, semaphores may be binary or counting. Which type of computer science semaphore does the railway implement?

Binary semaphore, since there can only be at most one train using a particular track at a time.

(b) Consider, instead, that a semaphore is used to control access of trains to Lausanne gare. What sort of semaphore is useful, and how would you use it?

In this case, we should use a counting semaphore, where the value represents the number of platforms available. An incoming train waits on the semaphore while an outgoing train posts on the semaphore.

(c) The following code snippet presents declaration and initialization of a semaphore `s`. Explain the `sem_init` function.

```
1    #include <semaphore.h>
2    sem_t s;
3    sem_init(&s, 0, 1);
```

`sem_init()` initializes the semaphore s to the value provided as the third argument (in this example semaphore s is initialized to the value 1). Its second argument indicates that the semaphore is shared between threads in the same process. If set to a non-zero value, it indicates that this semaphore will be shared between multiple processes and should be allocated in a shared memory region.

(d) Take into consideration the following code snippet: we have expressed the behavior of wait and post operations as pseudo-code.

```
1  int sem_wait(sem_t* s) {
2    wait for the value of s to become positive
3    decrement the value of semaphore s by one
4  }
5
6  int sem_post(sem_t* s) {
7    increment the value of semaphore s by one
8    if there are one or more threads waiting, wake one
9  }
```

The initial value of the semaphore m (initialized to `X` in the below code) is essential to implement the functionality we want from it. Consider the case where we want to implement a simple mutex locking mechanism with semaphore `m`:

```
1  sem_t m;
2  sem_init(&m, 0, X); // initialize to X;
```

Name: _____

```
3    sem_wait(&m);
4    // critical section here
5    sem_post(&m);
```

Go through definition of `sem_wait()` and `sem_post()` shown in the above code and answer the question - what should `X` be?

X should be set to 1.

Let's imagine a scenario with two threads. The first thread (Thread 0) calls `sem_wait()`; it will first decrement the value of the semaphore, changing it to 0. Having a non-negative value, `sem_wait()` will simply return and the calling thread will continue; thread 0 is now free to enter the critical section.

If no other thread tries to acquire the lock while thread 0 is inside the critical section, when it calls `sem_post()`, it will simply restore the value of the semaphore to 1 (and not wake a waiting thread, because there are none).

If, however, thread 1 *also* called sem_wait(), it would decrement the value of the semaphore to -1, restore it, and then wait until it becomes positive again.

Now, consider the following example.

```
1    sem_t s;
2    void* child(void* arg) {
3      printf("child\n");
4      sem_post(&s);                    // signal here: child is done
5      return NULL;
6    }
7
8    int main(void) {
9      sem_init(&s, 0, X);              // what should X be?
10     printf("parent: begin\n");
11     pthread_t c;
12     pthread_create(&c, NULL, child, NULL);
13     sem_wait(&s);                    // wait here for child
14     printf("parent: end\n");
15     return 0;
16   }
```

We want to gain the following order in execution:

```
1    // We want parent thread to wait until
2    // child thread finishes its execution
3    parent: begin
4    child
5    parent: end
```

What should be the initial value of semaphore to achieve this effect?

The initial value of the semaphore should be set to 0. Consult chapter 31, page 5 for a detailed explanation.

(e) What is a reader-writer lock?

A reader-writer lock implements a useful feature: shared data allows multiple concurrent readers, but requires exclusive access when writing. Compared to a mutex/spin lock, it has the advantage of allowing several concurrent readers in the critical section.

Consider the situation where we have data structure on which we apply insert and lookup operations. Reader-writer lock is used to guarantee that while reading the data structure no inserts are concurrently going on, while also allowing concurrent lookups.

Name: _____

(f) In which situation will a reader-writer lock work best?

Imagine you are storing Beyonce's tweets in a database. This database will be read millions of times, whenever fans load her twitter page, but gets written to less frequently (perhaps a few times a day).

This database can be concurrently read and written, and therefore requires synchronization. A mutex would only allow a single user to see her twitter at a time. In contrast, a reader-writer lock will allow all fans to access the page at the same time.

(g) In what situation would a reader-writer lock perform worse?

A reader-writer lock will perform worse than a mutex in a write-heavy workload (50% or more writes). In this situation, the lock is unable to leverage reader parallelism, mostly stuck with a single writer executing at a time. At the same time, it has a higher overhead per operation because it has to track all readers and writers separately.

(h) Does reader-writer lock have any negative aspects?

A reader-writer lock is not inherently unfair, but most trivial implementations may be. A simple implementation is one where the writer waits until all readers have completed before making the update. In this case, Beyonce would have to wait for millions of user requests to complete before her tweet gets uploaded. This would cause the starvation of writer threads while prioritizing reader threads. The implementation details need to be considered based on the application.

(i) The five philosophers problem is as follows: five philosophers sit around a table with five forks between them. To eat, each philosopher requires a fork in their left and right hand. How can this problem lead to a deadlock?

The incorrect (though intuitive) algorithm for each philosopher is:

1. Grab the left fork
2. Grab the right fork
3. Eat
4. Release both forks

This algorithm can lead to deadlock: if each philosopher happens to grab a fork on their left before any philosopher can grab the fork on their right, each will be stuck holding one fork and waiting for another.

(j) What is the simplest way to solve deadlock problem from previous question?

We can prevent a deadlock by changing the order of acquiring forks for only one of the philosophers.

For example, we can choose the philosopher 4 (the highest numbered one) to get the forks in a different order than the others. Philosopher 4 will in this case first grab right fork and then left one.

For a deadlock, each philosopher must have exactly one fork (which, as per the algorithm, must be the first fork they pick up). This means that philosopher 4 has the right fork and philosopher 0 has the left fork, implying that both picked up the same fork (which is impossible). This contradiction proves that the deadlock is impossible.

Convince yourself that this would work!