CS 342302 Operating Systems

Fall Semester 2021

Prof. Pai H. Chou

Weekly Review 8

(scope: Chapter 6-7: Synchronization)

## 1. Definitions and Short Answers

PART-I

1. In the version of producer-consumer code that uses both in/out and count, race conditions could occur.

|  |  |
| --- | --- |
| // Producer  while (1) {  nextItem = getItem();  while (count==BUFSIZE) ;  buffer[in] = nextItem;  in = (in+1) % BUFSIZE;  count++;  } | // Consumer  while (1) {  while(count==0) ;  item = buffer[out];  out = (out+1) % BUFSIZE;  count--;  } |

* 1. Give one example where one process's update is lost due to overwriting by the other process, assuming preemptive scheduling.
  2. Explain how the race condition causes incorrect results.
  3. Where can you convert into critical sections to eliminate the race condition?

A: In the assembly instructions corresponding to count++ or count --.

1. What are the three requirements of a **critical section**?

A: The three requirements for a critical section (CS) are

1. **Mutual Exclusion:** At most one process in the CS at the time. Other processes are excluded from entering the critical section.

2. **Progress:** Once a CS is entered by a process, it must exit it. If there are no other processes in the CS, processes that wish to enter the CS must not be blocked indefinitely.

3. **Bounded waiting**: Processes must exit the CS in a bounded amount of time. In addition, once a process has requested entering the CS, there must be a bound on the number of times that other processes are allowed to enter their CS after such request.

1. Of the two **software solutions** to the critical section problem, why do they work, and what are their assumptions?
   1. Non-preemptive scheduling – It works because there is no race condition. It assumes no process or thread yields in the middle of a critical section.
   2. preemptive scheduling - using Peterson's Solution – Because the three requirements for a CS are met. However, it is a software solution and thus it assumes no hardware support.
2. If you perform **nonpreemptive** scheduling, can there be a race condition? On a single-threaded uniprocessor? On a multiprocessor? Why or why not?

A: In a single threaded processor race conditions for non-preemptive scheduling are impossible.

1. Consider Peterson' solution,
   1. Does it need to **temporarily disable interrupts** in the critical section?

A: No, it doesn’t. It doesn’t rely on hardware support, so it doesn’t rely on disabling interrupts temporarily. Context switching may happen at any moment.

* 1. Does it work on processors with **two hardware threads**? With what kind of assumptions? When can it fail when the assumptions don't hold?

A: No, it doesn’t work. It would work assuming code order is followed. In multiple hardware threads, CPU sees values not in code order and thus Peterson’s solution wouldn’t work in practice.

1. What does a **memory-barrier** instruction do? How can it be used as part of a synchronization primitive?

A: A memory barrier instruction ensures memory is loaded in code order.

1. Why is it inefficient to **disable interrupts** on a **multiprocessor**?

A: Because it is very expensive to arrange all other cores to disable their interrupts. In addition, the delay overhead in the pipeline flush (instruction delay) is high.

1. When atomic test-and-set is called as a C function, you **successfully acquired the lock** when it returns true or returns false? What is the lock value after?

A: When it returns false since the atomic test-and-set returns the previous value (unlocked value). After the process is done executing, the lock must be unlocked and then lock = FALSE. It is a protocol.

1. How can **atomic** **compare-and-swap** be used to implement a lock? What **additional information** can such a locking data type provide that is otherwise not available in atomic test-and-set?

A: The atomic compare-and-swap lock mechanism is very similar to the atomic test and set, however is more general since the lock value types are not restricted to be Boolean. You can encode additional information such as process id or somehow encode the new owner of the lock.

1. Can an atomic variable replace a critical section in general? In what case may it fail, if any?

A: No, it may not replace a critical section in general. Example where it may fail: Single producer, multiple consumer problem. Mutual exclusion is not guaranteed since multiple consumers could take control at the same time. Its applicability is limited to single update of shared data, rather than being guards on other shared data structure.

1. How do you pronounce "Dijkstra"?

A: “DAIKS”– “TRA”

1. After declaring a semaphore S,
   1. What does wait(S) do? Under what condition would it block, and under what condition would it not block? What is its effect on S's value upon returning? What causes wait(S) to unblock?

A: Wait(S) decrements the value S of a semaphore. A process that wishes to use a resource performs a wait(S) operation on the semaphore. The process blocks if S <= 0 (nonpositive). The process won’t block if S is positive. Upon returning S decrements its value by 1. Calling signal(S) causes the thread to unblock if S > 0.

* 1. What does signal(S) do? Does it ever block? If so, under what condition, or if not, why not? What is its effect on S's value upon returning?

A: signal(S) increments the value of the semaphore S. If the value is negative it puts a blocked process from the semaphores queue to the ready queue. Upon returning S increments its value by 1. It never blocks.

1. What does the **value of semaphore S** represent when it is positive? When it is negative? When it is 0?

A: When it is a positive int, a semaphore represents the number of instances of some resource. When it is 0 it means the resource is not available.

1. How can a mutex be implemented using a semaphore?

A: A mutex can be implemented using a semaphore when the S value is restricted to 0 or 1.

1. How can **barrier synchronization** be implemented using a semaphore? If a process P1 needs to execute A before Process P2 executes B, how can the code of the two processes be written? Explain how the waiting process can be unblocked even though the semaphore's value is initialized to 0.

A: Initializing S = 0. If process P1 is to execute A before process P2 can execute B, then P1 must call signal(S), incrementing the value of S and thus lifting the barrier. If B arrives first, then it calls wait(S) and waits for A to call signal(S).

1. Is it always more efficient to use *non-busy-wait semaphores*? When is it more efficient to use *busy-wait semaphores*, if ever?

A: Usually it is more efficient to use non-busy-wait semaphores, although in presence of **low congestion** and low unlocking time, the spinlock can be. quite efficient.

From lecture -> Depends on the length of the critical section (Is atomic!).

For short CS, we could use busy-wait since there is low overhead, especially for multiprocessor. For long CS, busy waiting is inefficient.

1. In the classical bounded-buffer problem of n-buffers, it declares three semaphores  
    semaphore mutex = 1;  
    semaphore full = 0;  
    semaphore empty = n;
   1. What is the purpose of the semaphore mutex? What resource does it protect?

A: It is used for mutual exclusive access – Allows direct modification of the shared queue.

* 1. What is the purpose of the semaphore full?

A: It represents the number of full buffers as a resource. There are at most N buffers in this problem.

1. Continuing with the classical bounded-buffer problem, Producer's code looks like infinite loop with body:  
    1 *produce next item*;  
    2 wait(empty);  
    3 wait(mutex);  
    4 *enqueue next item*;  
    5 signal(mutex);  
    6 signal(full);  
   Consumer's code looks like infinite loop with body  
    7 wait(full);  
    8 wait(mutex);  
    9 *deque the next item*;  
   10 signal(mutex);  
   11 signal(empty);  
   Fill in the blanks above (lines 2, 3, 5, 6, 7, 8, 10, 11) with the proper semaphores. Explain why they need to go in those places.
2. In the Readers-Writers classical synchronization problem, two semaphores rw\_mutex and mutex are declared, in addition to an int readcount=0;
   1. What is the purpose of semaphore rw\_mutex;? Why is it initialized to 1?

A: This semaphore allows readers to block writers and writers to block readers and writers. It is initialized to 1 for the first reader or writer to access it and block other writers or reader/writers respectively. There is not reader-writer precedence so initializing it to 0 would cause starvation.

* 1. What is the purpose of the semaphore named mutex in the code? Why is it initialized to 1 and use a separate int readcount = 0; instead of using a *counting semaphore* to keep track of the number of readers?

A: The purpose of the mutex semaphore is to ensure mutual exclusion when updating the int variable readcount. The readcount variable keeps track on how many processes are currently reading. The variable readcount is not a resource so a counting semaphore is not required.

It is initialized to 1 such that the first reader

1. Continuing with the Readers-Writers classical synchronization problem, fill in the blanks below with the proper semaphores (rw\_mutex, mutex):  
    1 Writer():  
    2 while (TRUE):  
    3 wait(rw\_mutex)  
    4 *code for writing*  
    5 signal(rw\_mutex)  
    6 Reader():  
    7 while (TRUE):  
    8 wait(mutex)  
    9 readcount += 1  
   10 if (readcount == 1):  
   11 wait(rw\_mutex)  
   12 signal(mutex)  
   13 *code to read data*  
   14 wait(mutex)  
   15 readcount -= 1  
   16 if (readcount == 0):  
   17 signal(rw\_mutex)  
   18 signal(mutex)
2. In the Dining Philosophers problem, if the code for each philosopher is written as the following infinite loop that make use of an array of semaphores chopstick[5] = {1, 1, 1, 1, 1};  
    1 do {  
    2 wait(chopstick[i]);   
    3 wait(chopStick[(i + 1) % 5]);   
    4 *eat rice*;  
    5   signal (chopstick[i]);    
    6 signal (chopstick[ (i + 1) % 5] );   
    7 *think*;  
    8 } while (TRUE);
   1. Explain a situation where a **deadlock** can occur.

A: Deadlock can occur if a context switch happens between the two wait statements for each philosopher process.

* 1. Explain a situation where a philosopher might **starve**.

A:

* 1. What is the difference between deadlock and starvation?

A: A deadlock occurs when every process holds a resource and waits for a resource held by another process. Starvation on the other hand, is when a process waits or is blocked indefinitely waiting for a resource.

## 2. Programming Exercise: Parking Simulation

In this assignment, you are to implement a parking simulation program in Python using semaphores.

A parking lot is a good match with (counting) semaphores because it is a resource with multiple instances (i.e., N parking spots). So, it will allow up to N simultaneous users to use the shared resources. Any time the occupancy is less than N, there is no blocking; but if more than N, then some will have to block.

2.1 You will need several data structures for the parking lot:

* a counting semaphore for the number of parking spots
* a list to represent the spots (i.e., record which car is parked in which position)
* another synchronizing data structure of your choice when modifying the list of spots

Use the following template for making the parking lot data structure  
  
import threading  
def MakeParkingLot(N):

global sem # semaphore for the parking lot

global spots # list for the spots

global spotsSync # for synchronizing access to spots

spots = [None for i in range(N)]

# your code to initialize sem and spotsSync

You have several choices of data structures for spotsSync and spots. You may even choose some alternative to spots instead of the code shown here, but if you use a plain list, then you would need something like a mutex, a lock, or another semaphore for spotsSync. Check out the available synchronization primitives from [threading](https://docs.python.org/3/library/threading.html) module. What would you choose and why?

2.2 Each car can be represented by a thread. In the next function, MakeCars(C), create C threads and return a list of them.  
  
def MakeCars(C):  
 # your code here to spawn threads  
 # don’t forget to return the list

2.3 Next, write the function to be attached to each thread, i.e., the action of parking the car, leaving it there for some time, and leaving. it will make use of the same global data structures declared earlier. Use the comments in the following template code to fill in the necessary statements.  
  
**def** Park(car):  
 **global** sem, spots, spotsSync  
 # 2.3.1 ############################  
 # if spot available, grab it; otherwise wait until available.  
 # Hint: don’t use if/else statement! this is just one line.  
 # 2.3.2 ###########################################  
 # after confirming one parking spot, modify the spots   
 # (Python list or your choice of list-like data structure to  
 # put this car into the spot. The following is an example  
 # of what it can do, but you may have a different way of  
 # grabbing parking spots.  
 # Do you need to protect access to the following block of  
 # code? If so, add code to protect it; if not, why not?  
 for i in range(len(spots)):  
 if spots[i] is None:  
 spots[i] = car  
 break  
 snapshot = spots[:] # make a copy for printing  
 # now let us print out the current occupancy  
 print("Car %d got spot: %s" % (car, snapshot))  
 # leave the car on the lot for some (real) time!  
 import time  
 import random  
 st = random.randrange(1,10)  
 time.sleep(st)  
 # now ready to exit the parking lot. What do we need to   
 # 2.3.3 ################################  
 # update the spots data structure by replacing the spot   
 # that current car occupies with the value None;   
 # protect code if needed  
 # (2) print out the status of the spots  
 print("Car %d left after %d sec, %s" %   
 (car, st, myCopySpots))  
 # 2.3.4 #################################  
 # (3) give the spot back to the pool   
 # (hint: semaphore operation)

# Finally, have the main program run it:  
if \_\_name\_\_ == '\_\_main\_\_':  
 MakeParkingLot(5)  
 cars = MakeCars(15)  
 for i in range(15): cars[i].start()

Here is sample output. Your output may be in a different order, but it must be consistent.

$ python3 parking.py

Car 0 got spot: [0, None, None, None, None]

Car 1 got spot: [0, 1, None, None, None]

Car 2 got spot: [0, 1, 2, None, None]

Car 3 got spot: [0, 1, 2, 3, None]

Car 4 got spot: [0, 1, 2, 3, 4]

Car 0 left after 3 sec, [None, 1, 2, 3, 4]

Car 5 got spot: [5, 1, 2, 3, 4]

Car 2 left after 3 sec, [5, 1, None, 3, 4]

Car 6 got spot: [5, 1, 6, 3, 4]

Car 3 left after 4 sec, [5, 1, 6, None, 4]

Car 7 got spot: [5, 1, 6, 7, 4]

Car 6 left after 1 sec, [5, 1, None, 7, 4]

Car 8 got spot: [5, 1, 8, 7, 4]

Car 5 left after 3 sec, [None, 1, 8, 7, 4]

Car 9 got spot: [9, 1, 8, 7, 4]

Car 1 left after 8 sec, [9, None, 8, 7, 4]

Car 4 left after 8 sec, [9, None, 8, 7, None]

Car 10 got spot: [9, 10, 8, 7, None]

Car 11 got spot: [9, 10, 8, 7, 11]

Car 10 left after 3 sec, [9, None, 8, 7, 11]

Car 12 got spot: [9, 12, 8, 7, 11]

Car 7 left after 7 sec, [9, 12, 8, None, 11]

Car 13 got spot: [9, 12, 8, 13, 11]

Car 11 left after 5 sec, [9, 12, 8, 13, None]

Car 14 got spot: [9, 12, 8, 13, 14]

Car 8 left after 9 sec, [9, 12, None, 13, 14]

Car 9 left after 9 sec, [None, 12, None, 13, 14]

Car 13 left after 6 sec, [None, 12, None, None, 14]

Car 14 left after 6 sec, [None, 12, None, None, None]

Car 12 left after 9 sec, [None, None, None, None, None]

2.4 Show your typescript. Run your code multiple times. Does it show the same or different output? Why?