CS 3423 Operating Systems
Fall Semester 2018
Prof. Pai H. Chou

Assignment 5

(scope: Chapter 6: Synchronization)
Due Date: Sunday, November 4, 2018, 11:59pm
Submit electronically on iLMS
105065532 彭成全

What to submit: One zip file named <studentID>-hw5.zip (replace <studentID> with your own student ID). It should contain three files:

- one PDF file named **hw5.pdf** for Section 1. Write your answers in **English**. Check your spelling and grammar. Include your name and student ID!
- Sections 2.1, 2.2, 2.3: turn in ONE Python source code named **hw5.py** . Explanations should be given as comments in the source file. Questions should be answered in the PDF.
- Section 2.4: **typescript**; explain your answer in the PDF file.

1. [40 points] Problem Set

1. [20 points] 6.2 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i (i == 0 or 1) is shown in Figure 6.21. The other process is Pj (j == 1 or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

---------- (The following is Fig. 6.21) ----------

```
do {
   flag[i] = true;
   while (flag[j]) {
      if (turn == j) {
         flag[i] = false;
         while (turn == j) ; /* do nothing */
         flag[i] = true;
      }
    }
    /* critical section */
    turn = j;
    flag[i] = false;
    /* remainder section */
} while (true);
```

Figure 6.21 The structure of process P_i in Dekker's algorithm.

1. Mutual exclusion: When both flag set True, only one who have turn will proceed and enter its critical section.

2. Progress: If a thread wishes to access its critical section, set flag = True, if there are no other threads have True flag, this thread can enter its critical section, or this thread waits a while for others and then enter its critical section.

3. Bounded Waiting:Since code i give precedence to j first, this makes flag[i] = False after “while flag[j]” and will wait turn j, then get flag[i] = True to proceed its critical section, after proceeding, it set turn to another thread j, this avoids a process repeatedly enter. Hence, every thread have a chance to proceed after waiting other thread.

2. [10 points] 6.4 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.

Ans: If a program can disable timer interrupt, a program can prevent context switching and other processes won’t have a chance to execute until a program finished synchronization primitives.

3. [10 points] Consider how to implement a mutex lock using an atomic hardware instruction in 8051. Look up the JBC *bit*, *rel* instruction in the 8051 instruction set, which says

   JBC executes the jump if the addressed bit is set, and also clears the bit. Thus a flag can be tested and cleared in one operation.

   3.1 Explain how you should interpret the bit value when using the JBC instruction? Does the value 0 or 1 encode locked or unlocked state? What value would you initialize the bit?

   JBC can be a mutex lock by using the flag of global variable and rel as “JBC flag, rel”. Value 1 encodes unlocked state, when flag is 1, JBC will jump to the rel and clear the bit. The initialize bit is 1 for everyone jump to the critical section at the initial.

   3.2 write a 8051 assembly code fragment to use JBC to acquire the mutex. You can declare your own labels as branch targets if needed.

```
TestAndSet:

        SETB        P1.1            ;; mutex lock
        JBC         P1.1,           Critical ;; jump to critical
        JMP         TestAndSet

Critical:
        …       …       ;; critical process block
        ...     ...
        RET
```

2. [60 points] Programming Exercise

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 [20 points] 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 module. What would you choose and why?

```
sem = threading.Semaphore(N)
spotsSync = threading.Lock()
```

Since the threads need to access shared data synchronically, we need to avoid data writing over the maximum spots and to limit maximum cars enter the spot, so we use 1 lock to check and write spot at one time with lock, also restricted maximum cars with semaphore.

2.2 [5 points] 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 [30 points] Next, you are to 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, spotSync
```

```
        # 2.3.1 [5 points]  ############################
        # if spot available, grab it; otherwise wait until available.
        #  Hint: don't use if/else statement!  this is just one line.
        # 2.3.2  [10 points] ##########################################
        # 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, explain why not.
```

I use code to protect block of code.

```
            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 [5 points] ###################################
        # (1) give the spot back to the pool (hint: semaphore operation)
        # 2.3.4 [10 points] ###################################
        # (2) update the spots data structure by replacing the spot
        #    that current car occupies with the value None; protect code if needed
        # (3) print out the status of the spots
        print("Car %d left after %d sec, %s" %
              (car, st, myCopySpots))
# 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 [5 points] Show your typescript. Run your code multiple times. Does it show the same or different output? Why?

Different. For leaving time, it results in random.randrange(1,10) with random time. For got spot, sometimes may be different because there might be more than one car exits at the same time. After that, there might be more than one car can enter in at the same time. Hence, the output order of got spot is also different.