Assignment 3

Philip Dumaresq

40082638

COMP 346 - Operating Systems

Goswami Dhrubajyoti

Question 1

[15 marks] Consider a single processor, single core environment. Somebody suggested the following solution to the critical section problem involving two processes P_0 and P_1. It uses two shared variables, `turn` and `flag`. Note that this is not the Peterson's solution discussed in class, but looks similar:

```
boolean flag [2]; // Initially False
int turn; // Initially 0
do {
  flag[i] = true;  // i == 0 for P0 and 1 for P1
  while (flag[j] == true) { // j = 1-i
      if (turn == j) {
          flag[i] = false;
          while (turn == j) ;  // Do nothing: just busy wait flag[i] = True;
      }
  }
  // Critical section code here
  // ...
  turn = j;
  flag[i] = false;
  // Remainder section code here// ...
} while (true)
```

The above is the code for process P_i, $i = 0$ or 1. The other process is P_j, where $j = 1–i$. Now answer the following questions:

a) Will the solution satisfy mutual exclusion of the critical section? You must prove or argue (in a way similar to we did in class for Peterson's solution) your answer.

Yes, it will satisfy mutual exclusion. In this method process P_i is blocked from running until P_j is finished, or vice versa. The loop before the critical section will busy wait until the other process's critical section is done running, at which point the flags will be flipped and the next process can run.

Let's say we have process P_0 and process P_1 running. P_0 gets to go first, it will check if `flag[0]` is true, at which point it won't be, so P_0 will get to execute it's critical section.

b) Will the solution satisfy the “progress” requirement? You must prove or argue your answer.

No, the progress requirement is not met. If process P_0 runs, then process P_1 tries to run 3 times in a row, then it will get blocked at the third process. It will sit there doing nothing until a P_0 comes along to unblock it. P_0 will have no problem executing right away, but after the second P_1 in a row, all subsequent P_1 will sit there doing nothing until getting unlocked by a P_0.

c) Will the solution satisfy the bounded waiting requirement? If so, what is the bound? You must prove or argue your answer.

No, the bounded waiting requirement is not met. If Process P_0 and P_1 are alternating evenly, then it works out, but if process P_0 runs, then process P_1 tries to run 3 times in a row, then it will get blocked at the third process. P_0 can eventually comes along and unblock it but changing the turn, but otherwise it'll block permanently.

Question 2

[20 marks] Answer the following questions:

a) Consider three concurrent processes `A`, `B`, and `C`, synchronized by three semaphores `mutex`, `goB`, and `goC`, which are initialized to 1, 0 and 0 respectively:

Process A	Process B	Process C
wait (mutex)	wait (mutex)	wait (mutex)
...	...	...
signal (goB)	wait (goB)	wait (goC)
...	signal (goC)	...
signal (mutex)	...	signal (mutex)
	signal (mutex)	

Does there exist an execution scenario in which:

(i) All three processes block permanently?

If process C goes first then it will block everything because `wait(mutex)` will make mutex 0, thus blocking any subsequent call to mutex. Then once it hits `wait(goC)` it too will be blocked because `goC` is initialized to 0, which will make it drop to -1 and block Process C. So all processes will be blocked.

(ii) Precisely two processes block permanently?

Yes, precisely two processes can block permanently. If process A goes first, then when it's done executing `mutex` will be 1 and `goB` will be 1. If Process C goes after that, then it will get blocked at `goC` since it's 0. Then process B goes, it will be able to hit `signal(goC)` because `goB` is 1, which will unlock process C, but then it will immediately lock itself again next time and continue indefinitely.

(iii) No process blocks permanently? Justify your answers.

If process A goes first, at the end of it's running `mutex` will be 1 and `goB` will be 1. If That's followed by process B, then when it exits, `mutex` will be 1 again, `goB` will be 0 and `goC` will be 1. Then Process C executes, it won't block because `goC` 1 will be 1. When it exits, `mutex` will be 1, `goB` will be 0 and `goC` will be 0. This is back to the same as the initial values. So if process A, B and C execute in order, then none of them will ever block.

b) Now consider a slightly modified example involving two processes:

- Process A

```
for (i = 0; i < m; i++) {
    wait (mutex);
    ...
    signal (goB);
    ...
    signal (mutex);
}
```

- Process B

```
for (i = 0; i < n; i++) {
  wait (mutex);
  ...
  wait (goB);
  ...
  signal (mutex);
}
```

(i) Let $m > n$. In this case, does there exist an execution scenario in which both processes block permanently? Does there exist an execution scenario in which neither process blocks permanently? Explain your answers.

Both scenarios exist.

- If Process B goes first, then the call to `wait(goB)` will block Process B from finishing, which will cause both processes to wait indefinitely since Process A can't enter while Process B is running.
- If process A goes first, then when it's done, `goB` will be 1, which will allow process B to execute unblocked. They can continue alternating like this without either ever getting blocked. So long as Process B doesn't execute enough times in a row that `goB` becomes negative, they won't block permanently.

(ii) Now, let $m < n$. In this case, does there exist an execution scenario in which both processes block permanently? Does there exist an execution scenario in which neither process blocks permanently? Explain your answers

Only the scenario where both processes block permanently exists.

- If Process B goes first, the same things happens as when $m > n$.
- Since `wait(goB)` will *always* be called more than `signal(goB)`, there doesn't exist a scenario where neither will block permanently because `wait(goB)` will always end up getting blocked and will never be unblocked by another call to `signal(goB)`.

Question 3

[15 marks]Consider the below implementations of a semaphore’s wait and signal operations:

```
wait () {
    Disable interrupts;
    sem.value--;
    if (sem.value < 0) {
        save_state (current) ; // current process
        State[current] = Blocked; //A gets blocked
        Enqueue(current, sem.queue);
        current = select_from_ready_queue();
        State[current] = Running;
        restore_state(current);  //B starts running
    }
    Enable interrupts;
}
```

```
signal(){
    Disable interrupts;
    sem.value++;
    if (sem.value <= 0){
        k = Dequeue(sem.queue);
        State[k] = Ready;
        Enqueue (k, ReadyQueue);
    }
    Enable interrupts;
}
```

a) What are the critical sections inside the wait and signal operations which are protected by disabling and enabling of interrupts?

- `wait`: The critical section of wait will save the current process state, push it into a queue and dequeue the next process from the queue to start it. The important things that need to be critical here are the setting of state and operating on the queue. You want these operations to be in the C.S to keep the queue's state sane. If `signal` got called before `wait` was done, then you could end up trying to select from an empty queue, which could cause a lot of problems.
- `signal`: The critical section of the signal procedure will indicate that the dequeued process is ready to execute, then it'll enqueue the process again so that it'll be run. This needs to be critical so that state[k] remains consistent with process k in the ready queue. If `signal` were to be intrupted and something managed to get into the queue before the enqueue operation could run, then the queue would be out of sync.

b) Give example of a specific execution scenario for the above code leading to inconsistency if the critical sections inside implementation of `wait()` and `signal()` are not protected (by disabling of interrupts).

If the critical sections aren't protected by blocking interrupts then you could end up with something like the state of the ready queue getting mixed up. For example in the `signal` procedure, if something else got enqueued between the enqueue and the dequeue, then the queue would get out of order and could mess things up.

c) Suppose that process `A` calling semaphore `wait()` gets blocked and another process `B` is selected to run (refer to the above code). Since interrupts are enabled only at the completion of the wait operation, will `B` start executing with the interrupts disabled? Explain your answer.

Yes, process `B` will start running with the interrupts still disabled. It's state is restored as the last operation in the `wait()` procedure before interrupts are re-enabled, so interrupts will be turned on once the process has begun, but process `B` will initially have them disabled.

Question 4

[20 marks] A file is shared between several reader and writer threads. Design a monitor to control the access of the file by the different threads so that the following constraints are satisfied:

(i) at most one writer can be active on the file at a particular time.

(ii) When a writer is writing to the file, no reader can read from the file.

(iii) More than one reader can be reading from the file simultaneously.

(iv) When a writer is waiting to write, no more new reader should be allowed to read.

(v) When a writer is writing and some other writer is waiting to write, then the writer is given more preference over a reader waiting to read.

The general structure of each reader and writer thread is shown in the following:

```
Monitor FileControl {
        // Definition of the monitor to be filled in by you
        int readers = 0;
        int writerIdSeq = 0;
        FifoQueue writerQueue;

        writerEntry() {
                writerId = writerIdSeq++; // Generate a UID for this writer
                // signal immediately that a writer is waiting since writers have priority
                writerQueue.enqueue(writerId);
                // New readers can't get added because they need to wait for there to be no waiting writers
                while (readers > 0) {}
                // Wait until this writer is the next one in the queue, then allow it to write
                while (writerQueue.peek() != writerId) {}
        }

        writerExit() {
                writerQueue.dequeue();
        }

        readerEntry() {
                while (!writeQueue.isEmpty()) {} // Wait until all writers are done
                readers++;      // Then add this reader
        }

        readerExit() {
                readers--; // Show that this reader is done
        }
}

FileControl fc;   // An instance of the monitor
```

- Writer Thread:
- Reader Thread:

```
while (True) {
    ...
    fc.WriterEntry();
    Write (file);
    fc.WriterExit();
    ...
}
```

```
while (True) {
    ...
    fc.ReaderEntry();
    Read (file);
    fc.ReaderExit();
    ...
}
```

Fill in the pseudo-code for the monitor FileControl as shown above.

Question 5

[10 marks] Someone wrote the following solution to solve the First Reader Writer Problem, where multiple readers can access the file at the same time; a writer however has exclusive access to the file (i.e. when a writer accesses the file, no other reader or writer can access the file).

```
semaphore writeBlock = 1, mutex = 1;
int numOfReaders = 0;
```

```
//Reader Code - runs in a loop
if (numOfReaders == 0) {
    wait(writeBlock);
    wait(mutex);
    numOfReaders++;
    signal(mutex);
}
else {
    wait (mutex);
    numOfReaders++;
    signal (mutex);
}
read (file, ...)
wait (mutex);
numOfReaders--;
if (numOfReaders == 0)
    signal(writeBlock);

signal (mutex);
```

```
//Writer Code - runs in a loop
wait (writeBlock);
write (file, ...)
signal (writeBlock);
```

Does this solution suffer any problem(s)? Whether yes or no, explain your answer very clearly and in full details (preferably with an example scenario of what can go wrong, if any)

According to the description of the problem, this solution does satisfy it. Writers clearly have exclusive access to the file because they can't write if any readers are active, and readers will check for active writers before attempting to read. There can also be any number of active readers at a a time because the `mutex` semaphore won't prevent new readers from getting added. So while it does satisfy the constraints of this question, it doesn't mean that it's without problem. In this solution the writers can starve, because the only way they get unblocked is when there's no more readers. So if new readers constantly get added, then the writer will never get a chance to write, and just sit there waiting indefinitely.