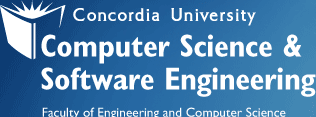
# COMP 346 – Fall 2018



**Theory Assignment 2**

**Tianlin Yang**

**40010303**

## Question # 1

Consider a single processor, single core environment. Somebody suggested the following solution to the critical section problem involving two processes P0 and P1. 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 Pi, i = 0 or 1. The other process is Pj, where j = 1 – i. Now answer the following questions:

1. 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.

**Answer:**

Mutual exclusion is ensured through the use of the flag and turn variables. If both processes set their flag to true, only one will succeed, namely, the process whose turn it is. The waiting process can only enter its critical section when the other process updates the value of turn.[[1]](#endnote-0)

Three possibilities for the critical sections:

1. Pi will enter the critical section when flag[i] is true and flag[j] is false.
2. Pj will enter the critical section when flag[j] is true and flag[i] is false.
3. If both flag[i] and flag[j] both true, when turn=i Pi will enter, turn=j Pj will enter.
4. Will the solution satisfy the “progress” requirement? You must prove your answer.

**Answer:**

Progress is provided, again through the flag and turn variables. This algorithm does not provide strict alternation. Rather, if a process wishes to access their critical section, it can set their flag variable to true and enter their critical section. It sets turn to the value of the other process only upon exiting its critical section. If this process wishes to enter its critical section again—before the other process—it repeats the process of entering its critical section and setting turn to the other process upon exiting.[[2]](#endnote-1)

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

**Answer:**

Bounded waiting is preserved through the use of the turn variable. Assume two processes wish to enter their respective critical sections. They both set their value of flag to true; however, only the thread whose turn it is can proceed; the other thread waits. If bounded waiting were not preserved, it would therefore be possible that the waiting process would have to wait indefinitely while the first process repeatedly entered—and exited—its critical section. However, Dekker’s algorithm has a process set the value of turn to the other process, thereby ensuring that the other process will enter its critical section next.[[3]](#endnote-2)

## Question # 2

Answer the following questions:

1. 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? (ii) Precisely two processes block permanently? (iii) No process blocks permanently? Justify your answers.

**Answer:**

i. If process B executes first and gets to the wait(goB) line the system will be in deadlock. The only place goB is signaled is in process A and the mutex lock is gone to process B and will never be released, similarly with process C it will never get mutex lock. The same thing occurs if Process C starts and gets stuck at wait(goC) the entire system (all 3 processes) will result in deadlock.

ii. If process a starts and signals(goB) then Process C starts next and gets stuck at wait(go C). Then process C and Process B are permanently blocked waiting for one another (deadlock).

iii. Process A starts signals(goB), process B starts having acquired wait(goB) signals(goC) and then process C starts having acquired wait(goC) successfully terminates.

1. Now consider a slightly modified example involving two processes:

Process A Process B

-------------- -------------

for (i = 0; i < m; i++) { for (i = 0; i < n; i++) { wait (mutex); wait (mutex);

… …

signal (goB); wait (goB);

… …

signal (mutex); signal (mutex);

} }

1. 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.

**Answer:**

i. Both processes block permanently if Process B starts first it will indefinitely wait for process A to release goB. As long as Process A executes at least once before each Process B, the program will not block.

1. 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.

**Answer:**

ii. Yes, once Process A has finished executing M processes, then everything is ok until Process B executes more than M processses. Any m<process<n will result in dead lock because there will be no wait to release wait(goB).

## Question # 3

Consider the following solution for the bounded-buffer producer-consumer problem. One requirement is that the producer must print when the buffer is full and the consumer must print when the buffer is empty:

semaphore mutex = 1; semaphore empty = N; semaphore full = 0; int inp = outp = 0;

Code for Producer process: Code for Consumer process:

do { do {

produce (item); wait (full);

wait (empty); wait (mutex);

wait (mutex); item = Buffer [outp];

Buffer[inp] = item; outp = (outp + 1) % N;

inp = (inp + 1) % N; signal (mutex);

signal (mutex); signal (empty);

signal (full); if (empty == N)

if (full == N) printf (“Buffer is empty\n”); printf (“Buffer is full\n”); consume (item);

} while (True) } while (True)

N is the buffer size. There is at least one serious bug (non-syntactic) in the above program.

1. Find out all the bugs and describe what problems they can cause.

**Answer:**

The bug is in both the methods producer as well as the consumer, the producer method is producing an item then checking for the condition if the buffer is full. The consumer method is buying an item then checking for the condition if the buffer is empty. These will cause the problem for overhead to the buffer and get wrong value of the buffer.

1. Fix all the bugs and rewrite the program. In your solution, the producer must print when the buffer is full and the consumer must print when the buffer is empty.

**Answer:**

Code for Producer process:

do {

produce (item);

if (full == N)

printf (“Buffer is full\n”);

wait (empty);

wait (mutex);

Buffer[inp] = item;

inp = (inp + 1) % N;

signal (mutex);

signal (full);

} while (True)

Code for Consumer process:

do {

if (empty == N)

printf (“Buffer is empty\n”);

wait (full);

wait (mutex);

item = Buffer [outp];

outp = (outp + 1) % N;

signal (mutex);

signal (empty);

consume (item);

} while (True)

## Question # 4

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 class to be filled in by you

…

…

}

FileControl fc; // An instance of the monitor

Writer Thread: Reader Thread:

while (True) { while (True) {

... ...

fc.WriterEntry(); fc.ReaderEntry();

Write (file); Read (file);

fc.WriterExit(); fc.ReaderExit();

… …

} }

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

**Answer:**

Monitor FileControl {

// Initialization code below

condition readers, writers;

int activereaders = 0;

boolean writing = False;

int waiting

writers = 0;

// Atomic methods below

WriterEntry() {

If (writing || activereaders > 0)) {

waitingwriters++;

writers.wait();

waitingwriters--;

}

writing = True;

}

WriterExit() {

if (waitingwriters > 0)

writers.signal();

else {

writing = False;

readers.signal();

}

}

ReaderEntry() {

if (writing || waitingwriters > 0)

readers.wait();

activereaders++;

}

ReaderExit() {

activereaders--;

if (activereaders == 0) {

if (waitingwriters > 0)

writers.signal();

else readers.signal();

}

}

}

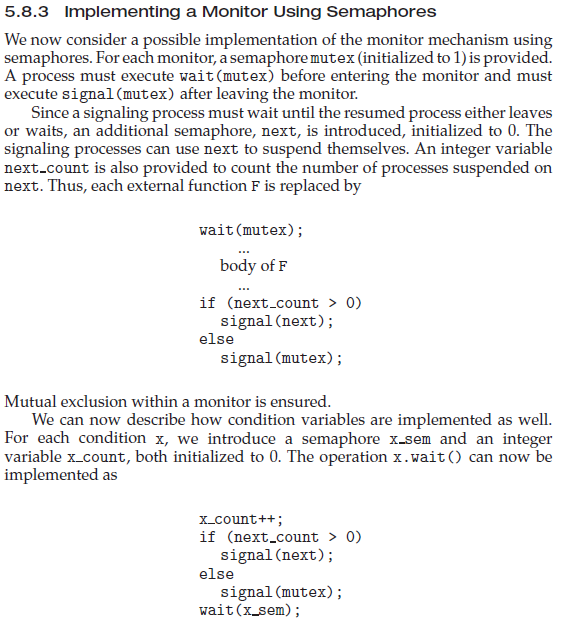
## Question # 5

The textbook (section 5.8.3 of the 9th edition) and the slides discuss about a monitor implementation using semaphores. Now, suppose we impose a restriction that the *signal()* operation of a condition variable (e.g., x.signal()) can only appear as the last statement in a monitor procedure. Suggest how the implementation described in section 5.8.3 can be simplified under this restriction.

**Answer:**

The given program has an atomicity bug due to which a reader and a writer can access the file simultaneously, which is not desired. Consider the following scenario: (i) Reader 1 arrives and since it is the first reader, it goes to the “if” part of the code, increments “numOfReaders” and starts reading the file. (ii) Reader 2 arrives and it enters the “else” part of the code. However, just before it calls “wait (mutex)”, there is a context switch and control passes to Reader 1. (iii) Reader 1 finishes reading, executes the rest of the code, decrements “numOfReaders” which becomes zero, and then subsequently does “signal(writeBlock)”. (iv) Subsequently, a writer arrives and starts writing to the file. (v) Reader 2 executes again from where it left, and gets access to the file. So, both Reader 2 and the writer have now concurrent access to the file which is not desired.

Note that the above scenario cannot arise in the code given in the textbook and the slides.



**Reference**

1. <https://quizlet.com/96311671/chapter-5-recommended-exercises-flash-cards/> [↑](#endnote-ref-0)
2. <https://quizlet.com/96311671/chapter-5-recommended-exercises-flash-cards/> [↑](#endnote-ref-1)
3. <https://quizlet.com/96311671/chapter-5-recommended-exercises-flash-cards/> [↑](#endnote-ref-2)