3SH3 Homework 2

Bradley Kohler

Question 5.4

Consider the following program:

```
const int n = 50;

int tally;

 void total()

{

        int count;

        for (count = 1; count<= n; count++){

                tally++;

        }

 }

void main()

{

        tally = 0;

        parbegin (total (), total ());

        write (tally);

}
```

a) Determine the proper lower bound and upper bound on the final value of the shared variable tally output by this concurrent program. Assume processes can execute at any relative speed and that a value can only be incremented after it has been loaded into a register by a separate machine instruction.

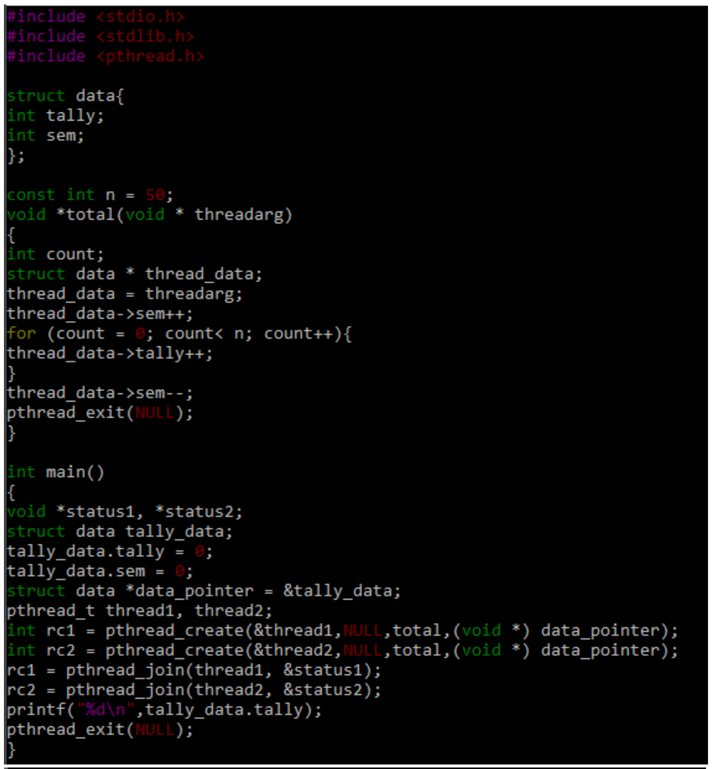
b) Suppose that an arbitrary number of these processes are permitted to execute in parallel under the assumptions of part (a). What effect will this modification have on the range of final values of tally?

Answer

a) The lower bound should be 50 due to a lack in mutual exclusion. The upper bound should be 100 if both processes run concurrently with mutual exclusion. There could be a scenario where the value of the shared variable tally is less than 50. One case where the value may be 2 is:

   i) Process 1 increments to one and does not store.
   ii) Process 2 increments to 49 and stores
   iii) Process 1 stores its value of 1
   iv) Process 2 loads 1
   v) Process 1 continues uninterrupted to 50 and stores
   vi) Process 2 continues with the loaded value of 1 and increments the value to 2 and stores

b) Tally will increase its upper bound by 50 for every parallel process. The lower bound will stay the same since all processes may run to completion with 2 left unexecuted. The scenario in (a) occurs between the two final processes that have not run yet.

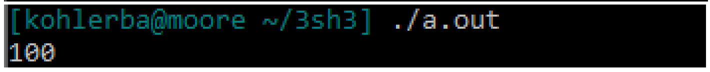


```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

struct data{
int tally;
int sem;
};

const int n = 50;
void *total(void * threadarg)
{
int count;
struct data * thread_data;
thread_data = threadarg;
thread_data->sem++;
for (count = 0; count< n; count++){
thread_data->tally++;
}
thread_data->sem--;
pthread_exit(NULL);
}

int main()
{
void *status1, *status2;
struct data tally_data;
tally_data.tally = 0;
tally_data.sem = 0;
struct data *data_pointer = &tally_data;
pthread_t thread1, thread2;
int rc1 = pthread_create(&thread1,NULL,total,(void *) data_pointer);
int rc2 = pthread_create(&thread2,NULL,total,(void *) data_pointer);
rc1 = pthread_join(thread1, &status1);
rc2 = pthread_join(thread2, &status2);
printf("%d\n",tally_data.tally);
pthread_exit(NULL);
}
```



```
[kohlerba@moore ~/3sh3] ./a.out
100
```

Question 5.6

Consider the following program:

```
boolean blocked [2];

 int turn;

void P (int id) {

     while (true) {

         blocked[id] = true;

         while (turn != id) {

              while (blocked[1-id])

                        /* do nothing */;

              turn = id;

              }
```

```
                    /* critical section */
                    blocked[id] = false;
                    /* remainder */
          }
}
void main()
{ blocked[0] = false;
blocked[1] = false;
turn = 0;
parbegin (P(0), P(1));
}
```

This software solution to the mutual exclusion problem for two processes is proposed in [HYMA66]. Find a counterexample that demonstrates that this solution is incorrect. It is interesting to note that even the Communications of the ACM was fooled on this one.

Answer

An example that shows this case is incorrect is when:

i) turn equals 0
ii) P(1) sets blocked[1] to true and sees blocked[0] set to false
iii) P(0) sets blocked[0] to true and sees turn equals zero
iv) P(0) enters critical section
v) P(1) sets turn to 1
vi) P(1) enters critical section

Mutual exclusion is violated because both processes are in their critical section.

Question 5.7

A software approach to mutual exclusion is Lamport's bakery algorithm [LAMP74], so called because it is based on the practice in bakeries and other shops in which every customer receives a numbered ticket on arrival, allowing each to be served in turn. The algorithm is as follows:

```
boolean choosing[n];
int number[n];
while (true) {
choosing[i] = true;
number[i] = 1 + getmax(number[], n);
choosing[i] = false;

for (int j = 0; j < n; j++){

        while (choosing[j]) { };

        while ((number[j] != 0) && (number[j],j) < (number[i],i)) { };

}

/* critical section */;

number [i] = 0;

/* remainder */;

}
```

The arrays choosing and number are initialized to false and 0, respectively. The ith element of each array may be read and written by process i but only read by other processes. The notation (a, b) < (c, d) is defined as:

$$(a < c) \text{ or } (a = c \text{ and } b < d)$$

a) Describe the algorithm in words.
b) Show that this algorithm avoids deadlock.
c) Show that it enforces mutual exclusion.

Answer

a) The newest process will choose a number that is one larger than the largest number currently given to a process in que. If no processes are currently choosing a number, then each process will check their number among others. If a process has a smaller non-zero number than any other process, and is first in que, then that process will enter its critical section while the others wait. This is to avoid processes with the same number entering their critical section at the same time. When a process finishes its critical section, it resets its number to zero.
b) This algorithm avoids deadlock by guaranteeing that no two processes are waiting on each other. Each process is given a number and is waiting on the highest process in que with the smallest number. This means that the lower processes in que are always waiting on the higher processes in que. Deadlock could only happen if a higher process in que is waiting on a lower process in que while a lower process in que is waiting on a higher process in que.
c) Mutual exclusion in guaranteed since every process is different (i.e. process 1, process 2, process 3). This means that even if two processes were given the same number, only one would enter its critical section. Each process checks that it is the lowest number and the highest in que before entering its critical section. It is impossible for two processes to meet that criteria simultaneously.

Question 5.13

3SH3 Homework 2

Bradley Kohler

Consider a sharable resource with the following characteristics: (1) As long as there are fewer than three processes using the resource, new processes can start using it right away. (2) Once there are three process using the resource, all three must leave before any new processes can begin using it. We realize that counters are needed to keep track of how many processes are waiting and active, and that these counters are themselves shared resources that must be protected with mutual exclusion. So we might create the following solution:

```
semaphore mutex = 1, block = 0;          /* share variables: semaphores, */
int active = 0, waiting = 0;                            /* counters, and */
boolean must_wait = false;                          /* state information */

semWait(mutex);                               /* Enter the mutual exclusion */
if(must_wait) {                           /* If there are (or were) 3, then */
    ++waiting;                           /* we must wait, but we must leave */
    semSignal(mutex);                         /* the mutual exclusion first */
    semWait(block);                 /* Wait for all current users to depart */
    SemWait(mutex);                         /* Reenter the mutual exclusion */
    --waiting;                              /* and update the waiting count */
}
++active;                               /* Update active count, and remember */
must_wait = active == 3;                           /* if the count reached 3 */
semSignal(mutex);                             /* Leave the mutual exclusion */

/* critical section */

semWait(mutex);                                   /* Enter mutual exclusion */
--active;                                  /* and update the active count */
if(active == 0) {                                     /* Last one to leave? */
    int n;
    if (waiting < 3) n = waiting;
    else n = 3;                                    /* If so, unblock up to 3 */
    while( n > 0 ) {                                   /* waiting processes */
        semSignal(block);
        --n;
    }
must_wait = false;                          /* All active processes have left */
}
semSignal(mutex);                             /* Leave the mutual exclusion */
```

The solution appears to do everything right: All accesses to the shared variables are protected by mutual exclusion, processes do not block themselves while in the mutual exclusion, new processes are prevented from using the resource if there are (or were) three active users, and the last process to depart unblocks up to three waiting processes.

a) The program is nevertheless incorrect. Explain why.

b) Suppose we change the if in line 6 to a while. Does this solve any problem in the program? Do any difficulties remain?

Answer

a) Two problems exist in this code. First, new processes might beat a blocked process into the critical section. Second, many processes may access a resource together in their critical section. If multiple processes are waiting at line 9 and become unblocked by a process in lines 25-28 there is no way in knowing when they will update their status of execution. So a new process in line 6 may not be blocked. Multiple processes will therefore be unblocked and access a resource together.
b) This will make unblocked processes recheck to see if they can execute or not. However, because there is no que, starvation may be an outcome do to new processes continuously cutting ahead of old processes that are waiting to execute. This is not a solution because it introduces more problems than it fixes.

Question 5.16

It should be possible to implement general semaphores using binary semaphores. We can use the operations semWaitB and semSignalB and two binary semaphores, delay and mutex. Consider the following:

```
void semWait(semaphore s)

{

semWaitB(mutex);

s--;

if (s < 0) {

        semSignalB(mutex);

        semWaitB(delay);

        }

else SemsignalB(mutex);

}

void semSignal(semaphore s)

{

semWaitB(mutex);

s++;

if (s <= 0)

        semSignalB(delay);
```

semSignalB(mutex);

}

Initially, s is set to the desired semaphore value. Each semWait operation decrements s, and each semSignal operation increments s. The binary semaphore mutex, which is initialized to 1, assures that there is mutual exclusion for the updating of s. The binary semaphore delay, which is initialized to 0, is used to block processes.

There is a flaw in the preceding program. Demonstrate the flaw and propose a change that will fix it. Hint: Suppose two processes each call semWait(s) when s is initially 0, and after the first has just performed semSignalB(mutex) but not performed semWaitB(delay), the second call to semWait(s) proceeds to the same point. All that you need to do is move a single line of the program.

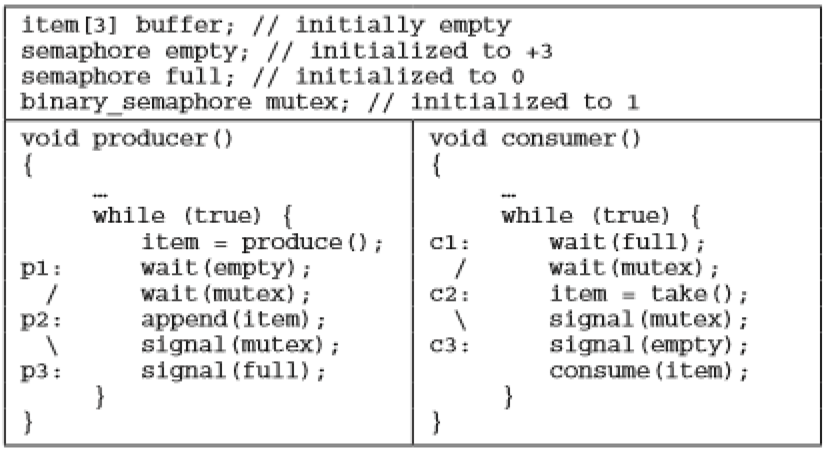
<u>Answer</u>

If two processes call semWait(s) when s is 0 and one process is running inbetween semSignalB(mutex) and semWaitB(delay), the other process proceeds to the same point because mutex in unlocked. Then if two other processes execute semsignal(s) they will both execute semSignalB(delay) but the second semsignalB is not defined.

A solution could be to move the the else line just before the final line in semWait to just before the final line in semSignal.

<u>Question 5.22</u>

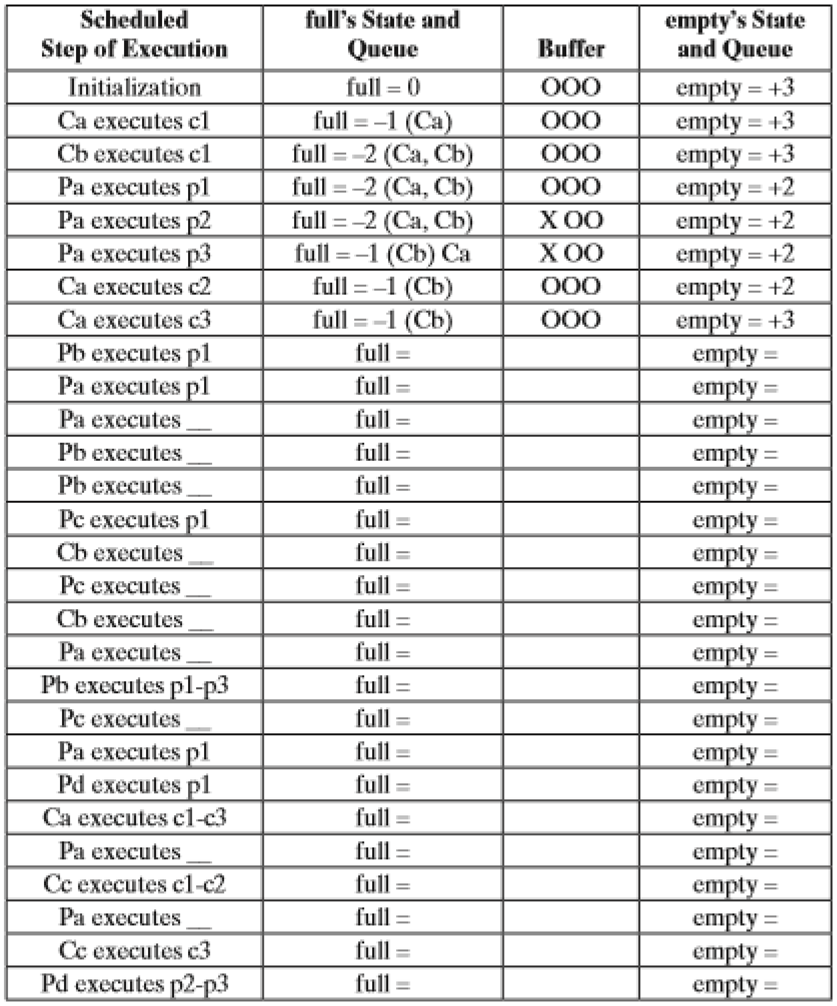
The following pseudocode is a correct implementation of the producer/consumer problem with a bounded buffer:



```
item[3] buffer; // initially empty
semaphore empty; // initialized to +3
semaphore full; // initialized to 0
binary_semaphore mutex; // initialized to 1

void producer()                      void consumer()
{                                    {
     ...                                  ...
     while (true) {                       while (true) {
         item = produce();           c1:      wait(full);
p1:      wait(empty);                 /       wait(mutex);
 /       wait(mutex);                c2:      item = take();
p2:      append(item);                \       signal(mutex);
 \       signal(mutex);              c3:      signal(empty);
p3:      signal(full);                        consume(item);
     }                                    }
}                                    }
```

Labels p1, p2, p3 and c1, c2, c3 refer to the lines of code shown above (p2 and c2 each cover three lines of code). Semaphores empty and full are linear semaphores that can take unbounded negative and positive values. There are multiple producer processes, referred to as Pa, Pb, Pc, etc., and multiple consumer processes, referred to as Ca, Cb, Cc, etc. Each semaphore maintains a FIFO (first-in-first-out) queue of blocked processes. In the scheduling chart below, each line represents the state of the buffer and semaphores after the scheduled execution has occurred. To simplify, we assume that scheduling is such that processes are never interrupted while executing a given portion of code p1, or p2, …, or c3. Your task is to complete the following chart.



Scheduled Step of Execution	full's State and Queue	Buffer	empty's State and Queue
Initialization	full = 0	OOO	empty = +3
Ca executes c1	full = –1 (Ca)	OOO	empty = +3
Cb executes c1	full = –2 (Ca, Cb)	OOO	empty = +3
Pa executes p1	full = –2 (Ca, Cb)	OOO	empty = +2
Pa executes p2	full = –2 (Ca, Cb)	X OO	empty = +2
Pa executes p3	full = –1 (Cb) Ca	X OO	empty = +2
Ca executes c2	full = –1 (Cb)	OOO	empty = +2
Ca executes c3	full = –1 (Cb)	OOO	empty = +3
Pb executes p1	full =		empty =
Pa executes p1	full =		empty =
Pa executes __	full =		empty =
Pb executes __	full =		empty =
Pb executes __	full =		empty =
Pc executes p1	full =		empty =
Cb executes __	full =		empty =
Pc executes __	full =		empty =
Cb executes __	full =		empty =
Pa executes __	full =		empty =
Pb executes p1-p3	full =		empty =
Pc executes __	full =		empty =
Pa executes p1	full =		empty =
Pd executes p1	full =		empty =
Ca executes c1-c3	full =		empty =
Pa executes __	full =		empty =
Cc executes c1-c2	full =		empty =
Pa executes __	full =		empty =
Cc executes c3	full =		empty =
Pd executes p2-p3	full =		empty =

Answer

Pb executes p1	full=-1(Cb)	OOO	empty=+2
Pa executes p1	full=-1(Cb)	OOO	empty=+1
Pa executes p2	full=-1(Cb)	XOO	empty=+1
Pb executes p2	full=-1(Cb)	XXO	empty=+1
Pb executes p3	full=0	XXO	empty=+1
Pc executes p1	full=0	XXO	empty=0
Cb executes c2	full=0	XOO	empty=0
Pc executes p2	full=0	XXO	empty=0
Cb executes c3	full=0	XXO	empty=+1
Pa execute p3	full=+1	XXO	empty=+1
Pb executes p1-p3	full=+2	XXX	empty=0
Pc executes p3	full=+3	XXX	empty=0
Pa executes p1	full=+3	XXX	empty=-1(Pa)

3SH3 Homework 2

Bradley Kohler

Pd executes p1	full=+3	XXX	empty=-2(Pa,Pb)
Ca executes c1-c3	full=+2	XXO	empty=-1(Pb) Pa
Pa executes p2	full=+2	XXX	empty=-1(Pb)
Cc executes c1-c2	full=+1	XXO	empty=-1(Pb)
Pa executes p3	full=+2	XXO	empty=-1(Pb)
Cc executes c3	full=+2	XXO	empty=0
Pd executes p2-p3	full=+3	XXX	empty=0