3. [12 pts] Here is an attempted solution to the 2-process mutual exclusion problem. Processes P0 and P1 share three boolean variables, `csFree`, `x0`, and `x1`, all initially true. In addition to the usual atomicity assumption of the mutual exclusion problem, assume that each of the if-statements A3 and B3 is executed atomically.

P0 executes

```
   do forever {
A1:  NCS;
A2:  while x0 do {
A3:    if csFree then
         { csFree := FALSE; x0 := FALSE };
     };
A4:  CS;
A5:  csFree := TRUE; x0 := TRUE ;
   }
```

P1 executes

```
   do forever {
B1:  NCS;
B2:  while x1 do {
B3:    if csFree then
         { csFree := FALSE; x1 := FALSE };
     };
B4:  CS;
B5:  csFree := TRUE; x1 := TRUE ;
   }
```

For each of the following properties, say whether or not the algorithm satisfies the property. If your answer is yes, give a proof that covers all possible executions. If your answer is no, give a counter-example execution.

a.

Safety property: At most one process is in its CS.

b.

Progress property: For each process, if the process is hungry then it eventually eats provided that no process stays eating forever and everything other than NCS is executed with weak fairness.

Answer:

3 [12 pts]. Solution

Part a [6 points].

Safety property holds.

Proof

Let Z0 denote "(P0 at A2 with x0=FALSE) or (P0 at CS) or (P0 in CS) or (P0 at A5)". Define Z1 symetrically (i.e., Z0 with P0, x0, Ai, replaced by P1, x1, Bi). It suffices to show that at any time Z0 and Z1 cannot both be true.

Consider the first time t1 that Z0 and Z1 both are both true.

Suppose at t1, Z0 becomes true and Z1 was already true. Then at t1, csFree was TRUE, P0 executed A3, and csFree became FALSE. And at some time t2 (< t1), csFree was TRUE, P1 executed B3, and csFree became FALSE. Thus csFree became TRUE between t2 and t1. Thus P0 executed A5 between t2 and t1 (P1 could not have executed B5 since Z1 holds between t2 and t1). But then Z0 and Z1 held at time t2, contradicting the assumption that t1 was the first such time.

The symetric argument holds if at t1, Z1 becomes true and Z0 was already true.

End of proof

COMMON MISTAKES:

- *A common mistake is to show that if P0 enters CS at time t1, then P1 cannot enter CS until P0 leaves CS. This does not account for the case where P1 may already be in CS at t1.*
- *To treat "P0 enters CS" to be the same as the execution of A3 with csFree=TRUE, whereas the former is actually the execution of A2 with x0=FALSE. Similarly with P1.*

Part b [6 points].

Progress does not hold.

Counter-example execution (each entry signifies execution of the statement):

```
A1, A2, A3, A4, B1, B2, B3, B2, A5, A1, A2, A3, A4, B1, B2, B3, B2, ...
```

Here, P1 is hungry for ever even though P0 does not stay in CS forever and P0 and P1 execute with weak fairness

1. [9 pts] Below are the arrival and service times (in seconds) for a stream of jobs arriving to a queue. Service times are known to the scheduler.

	arrival	service
J1	0	12
J2	2	8
J3	3	4
J4	7	9

This repeats every 35 seconds [i.e. J5 arrives at 35 with service 12, J6 arrives at 37 with service 8, etc.]

a.

For FCFS discipline, determine the average response time, the maximum number of jobs in the system at any time, and the average number of jobs in the system.

b.

Repeat part (a) for SJFP (i.e. Shortest Job First Preemptive) discipline.

c.

Repeat part (a) but with the server replaced by a server that is 10% slower; that is, if the old server required 10 s to serve a customer, the new server requires 11 s to serve that customer.

2. [9 pts] Implement binary semaphores in terms of counting semaphores. Do not use any other synchronization constructs. For constructs other than semaphores, assume only read/write atomicity and weak fairness progress. Your solution must be less than 30 lines and have no busy waiting. Elegance counts.

3. [8 pts] Consider an operating system that provides user processes with the following message-passing service:

- Send(pid, m): send message m to process pid. Blocks until process pid is at a corresponding receive.
- Receive(pid, m): receive message m from process pid. Blocks until process pid is at a corresponding send.
- If two processes are at corresponding send and receive, then the communication eventually takes place.

The following questions are with regard to processes using the message-passing service. Give brief and precise answers. Irrelevant material will cost you points.

a.

Explain how processes (using the message-passing service) can become deadlocked.

b.

Does it makes sense for the OS to provide a deadlock prevention method. If yes, describe one such method.

c.

Does it makes sense for the OS to provide a deadlock avoidance method. If yes, describe one such method.

d.

Does it makes sense for the OS to provide a deadlock detection/recovery method. If yes, describe one such method.

4. [4 pts] Consider a computer system where processes can have user-level threads. The cpu scheduler uses two-level round-robin as follows: A ready process is either in level A or level B. A-processes are served in round-robin order. B-processes are served in round-robin order but only when only when there are no A-processes. A A-process that accumulates more than 20 milliseconds of cpu time becomes a B-process. A B-process remains so until it stops being ready (i.e. ready or running).

Suppose the system has a set of processes that access a shared data structure using critical sections That is, the processes share a semaphore mutex initially 1, and every access is preceded by P(mutex) and succeeded by V(mutex).
Is it possible for the cpu scheduling and the processes to interact in a way that severely degrades the performance. Explain briefly?

Answers:
Below, Queue shows the sequence of (job, remaining service time) entries in order of their service, with the head (job being served) at the left.

Event	Time	NumJobs	Queue
	0-	0	( )
J1 arrives	0+	1	(J1, 12)
J2 arrives	2	2	(J1, 10), (J2, 8)
J3 arrives	3	3	(J1, 9), (J2, 8), (J3, 4)
J4 arrives	7	4	(J1, 5), (J2, 8), (J3, 4), (J4, 9)
J1 leaves	12	3	(J2, 8), (J3, 4), (J4, 9)
J2 leaves	20	2	(J3, 4), (J4, 9)
J3 leaves	24	1	(J4, 9)
J4 leaves	33	0	( )

```
Response time of J1 = 12 - 0 = 12
   " "     "   " J2 = 20 - 2 = 18
   " "     "   " J3 = 24 - 3 = 21
   " "     "   " J4 = 33 - 7 = 26

Average Response Time = (1/4)(12 + 18 + 21 + 26) = 77/4

Maximum Number of Jobs at any time = 4

Average Number of Jobs = (1/35)[(33 -0) + (24-2) + (20-3) + (12-7)] = 77/35

[This last metric is more easily obtained by Little's Law:
  AverageNumberOfJobs = AverageResponseTime * Throughput (= 4/35) = 77/35
]
```

Part b. [3 pts]

Event	Time	NumJobs	Queue
	0-	0	( )
J1 arrives	0+	1	(J1, 12)
	2-	1	(J1, 10)
J2 arrives	2+	2	(J2, 8), (J1, 10)
	3-	2	(J2, 7), (J1, 10)
J3 arrives	3+	3	(J3, 4), (J2, 7), (J1, 10)
	7-	3	(J3, 0+), (J2, 7), (J1, 10)
J3 leaves,			
J4 arrives	7+	3	(J2, 7), (J4, 9), (J1, 10)
J2 leaves	14	2	(J4, 9), (J1, 10)
J4 leaves	23	1	(J1, 10)
J1 leaves	33	0	( )

```
Response time of J1 = 33 - 0 = 33
   " "     "   " J2 = 14 - 2 = 12
   " "     "   " J3 =  7 - 3 =  4
   " "     "   " J4 = 23 - 7 = 16

Average Response Time = (1/4)(33 + 12 + 4 + 16) = 65/4

Maximum Number of Jobs at any time = 3 (or 4 for an instant)

Average Number of Jobs = 65/35   (easy from Little's Law)
```

Part c. [3 pts]

With the above server, 33 seconds of work enters every 35 seconds. If the service times are increased by 10%, then 36.3 (= 33 + 3.3) seconds of work would enter every 35 seconds. In the long term, the queue would blow up and average response time, max number of customers, average number of customers would all be unbounded (or infinity).
GRADING: For parts a and b, roughly 1 point for the individual response times and 2 points for the rest. For part c, all or nothing.

2 [9 pts]. Solution

Below, Vb() and Pb() to denote V and P ops on a binary semaphore, and V() and P() to denote the operations on a counting semaphore.
Declaration of binary semaphore S is implemented as

```
 record S {
   integer val initially the value of S ; // value of S
   CountingSemaphore wait initially 0 ;   // process stuck at Pb(S) waits
here
   integer waitCount initially 0 ;        // number of processes stuck on
wait
   CountingSemaphore mutex initially 1 ;  // to protect val and waitCount
}
```

Pb(S) is implemented as

```
 Pb(S) {
  P( S.mutex );
  if ( S.val = 0 )
     then { S.waitCount ++ ;
            V( S.mutex );
            P( S.wait );
     }
     else { S.val := 0 ;
            V( S.mutex );
     }
}
```

Vb(S) is implemented as

```
 Vb(S) {
  P( S.mutex );
  if ( S.waitCount > 0 )
     then { S.waitCount -- ;
            V( S.mutex );      // can be moved to after P(S.wait) in Pb(S)
     }
     else { S.val := 1 ;
            V( S.mutex );
     }
}
```

A variation is to combine val and waitCount in the usual way.
GRADING: Roughly 3 points for each part (variables definition, Pb(S), Vb(S)). -3 points if the approach is correct but the solution does not work completely.
Grievious errors leading to a max score of 0 or 1 [or 2 if the approach was otherwise correct]: using other synchronization constructs or busy waiting [0 points]; treating binary semaphore as just a counting semaphore initialized to either 0 or 1 [1 point]; subjecting a semaphore to operations other than P and V [1 point]; subjecting a non-semaphore variable to P or V operation [1 point].

3 [8 pts]. Solution

Part a. [2 pts]

A set of two processes, i and j, becomes deadlocked if either

i is at Send(j,.) and j is at Send(i,.), or
i is at Receive(j,.) and j is at Receive(i,.).

A set of three or more processes, i1, i2, i3, ..., iN, become deadlocked if

i1 is at Send/Receive(i2,.),
i2 is at Send/Receive(i3,.),
...,
iN is at Send/Receive(i1,.)

Part b. [2 pts]

It does not make sense for the OS to do deadlock prevention. Deadlock prevention would mean constraining when a process can do a send or a receive. Because the OS has no idea of the application. such constraints would be meaningless.

Part c. [2 pts]

It does not make sense for the OS to do deadlock avoidance. Deadlock avoidance would mean constraining when a send or receive returns. But this depends entirely on the peer user process, and not on any resource controlled by the OS. (Equivalently, deadlock avoidance and deadlock prevention result in the same set of "resource-allocation" states.)

Part d. [2 pts]

It does make sense for the OS to do deadlock detection/recovery. The OS can detect the presence of a deadlock by searching for a cycle as described in part a. This search can be initiated periodically or whenever a process calls a send or receive operation. Upon finding such a cycle, it can terminate all the processes involved in the cycle (or do appropriate exception handling if defined).
GRADING: Mostly all or nothing for each part. A common mistake was to redefine the semantics of the send and receive primitive.

4 [4 pts]. Solution

The only way for the *interaction* between the cpu scheduling and the critical section activity to cause performance degradation is if the system can enter the situation where a B-process has grabbed mutex [is in the critical section] and runs very slowly because of A-processes. This degrades the performance of all processes contending for the critical section.
Case 1: If the P operations are implemented without busy waiting, then the A-processes are not contending for the critical section, i.e., not be waiting on P(mutex). In this case, only processes waiting at P(mutex) are degraded by the extra slowness of the process holding mutex.
Case 2: If the P operations are implemented with busy waiting, then A-process can be waiting on P(mutex). In this case, none of the waiting or eating processes can make progress. This devastates the performance.
GRADING: 4 points for cases 1 and 2. 2 points for case 1 only.