University of Newcastle
School of Electrical Engineering and Computing

COMP2240 - Operating Systems
Workshop 6 - Solution
Topics: Concurrency: Deadlock and Starvation

1 Consider the following software solution to the mutual exclusion problem. It happens to be incorrect. Find a condition under which this solution fails.

```
var     blocked: array[0..1] of boolean;
        turn: 0..1;
procedure P(id: integer);
begin
        repeat
                blocked[id] := true;
                while turn <> id do
                        begin
                                while blocked[1 - id] do
                                        {nothing};
                                turn := id;
                        end;
                <critical section>
                blocked[id] := false;
                <remainder>;
        until false
end;
```

```
begin
        blocked[0] := false;
        blocked[1] ;= false;
        turn := 0;
        parbegin
                P(0); P(1);
        parend
end.
```

Answer:

Consider the case where `turn` equals `0` and `P(1)` sets `blocked[1]` to `true`, and then finds `blocked[0]` set to `false`. `P(0)` will then set `blocked[0]` to `true`, find `turn = 0`, and enter its critical section. `P(1)` will then assign `1` to `turn` and will also enter its critical section.

2 At an instant, the resource allocation state in a system is as follows:

4 processes P1-P4

4 resource types: R1-R4

R1 (5 instances), R2 (3 instances), R3 (3 instances), R4 (3 instances)

Snapshot at time T_0:

	Allocation			
	R1	**R2**	**R3**	**R4**
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	1	0
P4	1	1	0	0

Request			
R1	**R2**	**R3**	**R4**
2	0	0	2
1	3	0	1
2	1	1	0
4	0	3	1

Available			
R1	**R2**	**R3**	**R4**
2	1	1	2

Run the deadlock detection algorithm and test whether the system is deadlocked or not. If it is, identify the processes that are deadlocked.

Answer:
Given allocation state is:

	Allocation			
	R1	**R2**	**R3**	**R4**
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	1	0
P4	1	1	0	0

Request			
R1	**R2**	**R3**	**R4**
2	0	0	2
1	3	0	1
2	1	1	0
4	0	3	1

Available			
R1	**R2**	**R3**	**R4**
2	1	1	2

Since `Request[3] ≤ Available`, *P3* can run into completion. So, mark *P3*.

`NewAvailable = Available + Allocation[3]`

Available			
R1	**R2**	**R3**	**R4**
2	2	2	2

Since `Request[1] ≤ Available`, P1 can run into completion. So, mark P1.

`NewAvailable = Available + Allocation[1]`

Available			
R1	**R2**	**R3**	**R4**
2	2	3	2

At this stage, neither P2 nor P4 can run to completion as the request vectors of P2 and P4 have at least one element greater than available.
Hence, the system is deadlocked and the deadlocked processes are P2 and P4.

3 In the code below, three processes are competing for six resources labeled A to F.

a) Using a resource allocation graph (Figures 6.5 and 6.6), show the possibility of a deadlock in this implementation.

b) Modify the order of some of the get requests to prevent the possibility of any deadlock. You cannot move requests across procedures, only change the order inside each procedure. Use a resource allocation graph to justify your answer.

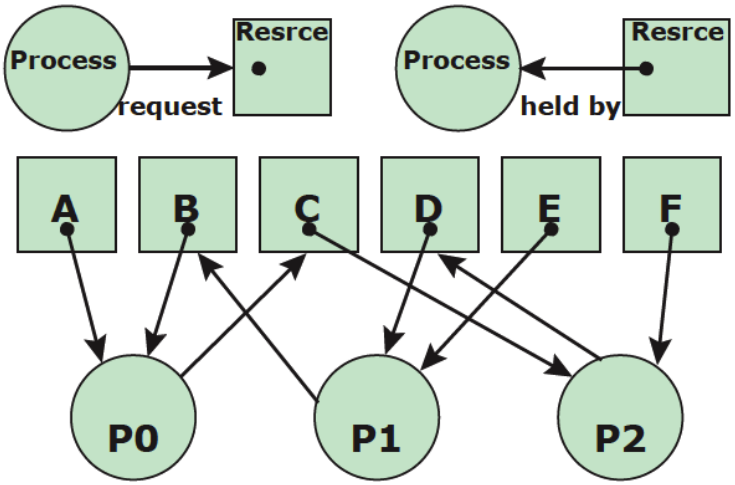
```
void P0()
{
  while (true) {
    get(A);
    get(B);
    get(C);
    // critical region:
    // use A, B, C
    release(A);
    release(B);
    release(C);
  }
}
```

```
void P1()
{
  while (true) {
    get(D);
    get(E);
    get(B);
    // critical region:
    // use D, E, B
    release(D);
    release(E);
    release(B);
  }
}
```

```
void P2()
{
  while (true) {
    get(C);
    get(F);
    get(D);
    // critical region:
    // use C, F, D
    release(C);
    release(F);
    release(D);
  }
}
```

Answer:

a)



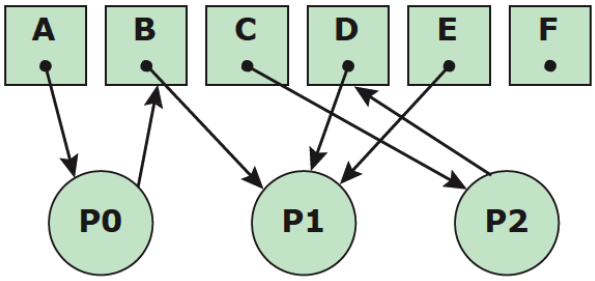
There is a deadlock if the scheduler goes, for example: P0-P1-P2-P0-P1-P2 (line by line): Each of the 6 resources will then be held by one process, so all 3 processes are now blocked at their third line inside the loop, waiting for a resource that another process holds. This is illustrated by the circular wait (thick arrows) in the RAG above:
P0→C→P2→D→P1→B→P0.

b) Any change in the order of the get() calls that alphabetizes the resources inside each process code will avoid deadlocks. More generally, it can be a direct or reverse alphabet order, or any arbitrary but predefined ordered list of the resources that should be respected inside each process.

Explanation: if resources are uniquely ordered, cycles are not possible anymore because a process cannot hold a resource that comes after another resource it is holding in the ordered list. See this remark in Section 6.2 about Circular Wait Prevention. For example:

A	B	C
B	D	D
C	E	F

With this code, and starting with the same worst-case scheduling scenario P0-P1-P2, we can only continue with either P1-P1-CR1… or P2-P2-CR2…. For example, in the case P1-P1, we get the following RAG without circular wait:



After entering CR1, P1 then releases all its resources and P0 and P2 are free to go. Generally the same thing would happen with any fixed ordering of the resources: one of the 3 processes will always be able to enter its critical area and, upon exit, let the other two progress.

4 Suppose the following two processes, foo and bar, are executed concurrently and share the semaphore variables S and R (each initialized to 1) and the integer variable x (initialized to 0).

```
void foo( ) {
    do {
      semWait(S);
      semWait(R);
      x++;
      semSignal(S);
      SemSignal(R);
    } while (1);
}
```

```
void bar( ) {
do {
    semWait(R);
    semWait(S);
    x--;
    semSignal(S;
    SemSignal(R);
} while (1);
}
```

a) Can the concurrent execution of these two processes result in one or both being blocked forever? If yes, give an execution sequence in which one or both are blocked forever.

b) Can the concurrent execution of these two processes result in the indefinite postponement of one of them? If yes, give an execution sequence in which one is indefinitely postponed.

Answer:

a) Yes. If `foo()` executes `semWait(S)` and then `bar()` executes `semWait(R)` both processes will then block when each executes its next instruction. Since each will then be waiting for a `semSignal()` call from the other, neither will ever resume execution.

b) No. If either process blocks on a `semWait()` call then either the other process will also block as described in (a) or the other process is executing in its critical section. In the latter case, when the running process leaves its critical section, it will execute a `semSignal( )` call, which will awaken the blocked process.

5 Given the following state of a system:
The system comprises of five processes and four resources.
P1-P5 denotes the set of processes.
R1-R4 denotes the set of resources.
Total Existing Resources:

Existing			
R1	**R2**	**R3**	**R4**
6	3	4	3

	Allocation				**Claim**			
	R1	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**
P1	3	0	1	1	6	2	1	1
P2	0	1	0	0	0	2	1	2
P3	1	1	1	0	3	2	1	0
P4	1	1	0	1	1	1	1	1
P5	0	0	0	0	2	1	1	1

a) Compute the Available vector.

b) Calculate the Need matrix.

c) Is the current allocation state safe? If so, give a safe sequence of processes. In addition, show how the Available (working array) changes as each process terminates.

d) If the request (1,1,0,0) from Process P1 arrives, will it be correct to grant the request? Justify your decision.

Answer:

Given that

Total number of existing resources:

R1	R2	R3	R4
6	3	4	3

Snapshot at the initial time stage:

	Allocation				Claim			
	R1	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**
P1	3	0	1	1	6	2	1	1
P2	0	1	0	0	0	2	1	2
P3	1	1	1	0	3	2	1	0
P4	1	1	0	1	1	1	1	1
P5	0	0	0	0	2	1	1	1

a) Total number of resources allocated to different processes:

R1	R2	R3	R4
3+1+1=5	1+1+1=3	1+1=2	1+1=2

Available Matrix = Total Existing – Total Allocation

Available			
R1	**R2**	**R3**	**R4**
1	0	2	1

b) Need matrix = Claim – Allocation

	Need			
	R1	**R2**	**R3**	**R4**
P1	3	2	0	0
P2	0	1	1	2
P3	2	1	0	0
P4	0	0	1	0
P5	2	1	1	1

c) Safety algorithm (Banker's Algorithm):

The following matrix shows the order in which the processes can run to completion. It also shows the resources that become available once a given process completes and releases the resources held by it.

At each step, a process `Pi` can be completed if `Need[i] ≤ Available`.

The Available matrix is updated as: `Available = Available + Allocation`

	Available			
	R1	**R2**	**R3**	**R4**
P4	2	1	2	2
P2	2	2	2	2
P3	3	3	3	2
P1	6	3	4	3
P5	6	3	4	3

Hence, the system is in a safe state and `<P4, P2, P3, P1, P5>` is a safe sequence.

d) Request from `P1` is (1, 1, 0, 0), whereas Available is (1, 0, 2, 1). As Request is greater than Available, this request cannot be granted.

Supplementary problems:

S1. Consider the following fragment of code on a Linux system.

```
read_lock(&mr_rwlock);
write_lock(&mr_rwlock);
```

Where `mr_rwlock` is a *reader–writer* lock. What is the effect of this code?

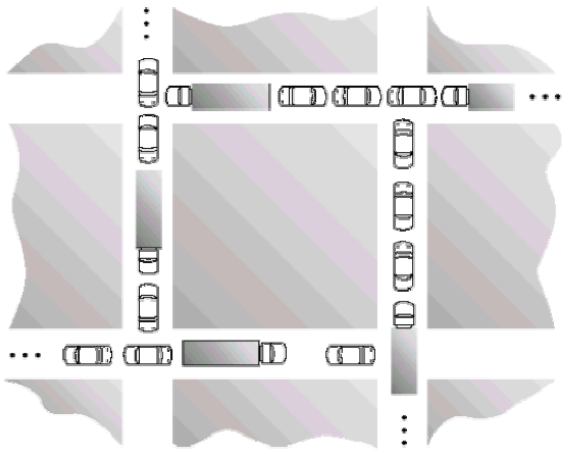
Answer:
This code causes a deadlock, because the writer lock will spin, waiting for all readers to release the lock, including this thread.

S2. A computer science student assigned to work on deadlocks thinks of the following brilliant way to eliminate deadlocks. When a process requests a resource, it specifies a time limit. If the process blocks because the resource is not available, a timer is started.
If the time limit is exceeded, the process is released and allowed to run again. If you were the professor, what grade would you give this proposal and why?

Answer:
I'd give it an F (failing) grade. What does the process do? Since it clearly needs the resource, it just asks again and blocks again. This is no better than staying blocked. In fact, it may be worse since the system may keep track of how long competing processes have been waiting and assign a newly freed resource to the process that has been waiting longest. By periodically timing out and trying again, a process loses its seniority.

S3. Is it possible to have a deadlock involving only one single-threaded process? Explain your answer.
Answer:
No. This follows directly from the hold-and-wait condition. There must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.

S4. Consider the traffic deadlock depicted in the following figure.



a) Show that the four necessary conditions for deadlock indeed hold in this example.

b) State a simple rule that will avoid deadlocks in this system.

Answer:

a) Consider each section of the street as a resource.
   i. *Mutual exclusion* condition applies, since only one vehicle can be on a section of the street at a time.
   ii. *Hold-and-wait* condition applies, since each vehicle is occupying a section of the street, and waiting to move on to the next section of the street.
   iii. *No-preemptive* condition applies, since a section of the street that is a section of the street that is occupied by a vehicle cannot be taken away from it.
   iv. Circular *wait* condition applies, since each vehicle is waiting on the next vehicle to move. That is, each vehicle in the traffic is waiting for a section of street held by the next vehicle in the traffic.

b) The simple rule to avoid traffic deadlock is that a vehicle should only enter an intersection if it is assured that it will not have to stop inside the intersection.

Existing			
R1	**R2**	**R3**	**R4**
6	3	4	3

	Allocation				Claim			
	R1	**R2**	**R3**	**R4**	**R1**	**R2**	**R3**	**R4**
P1	3	0	1	1	6	2	1	1
P2	0	1	0	0	0	2	1	2
P3	1	1	1	0	3	2	1	0
P4	1	1	0	1	1	1	1	1
P5	0	0	0	0	2	1	1	1

a)

Available			
R1	**R2**	**R3**	**R4**

b)

	Need			
	R1	**R2**	**R3**	**R4**
P1				
P2				
P3				
P4				
P5				

c)

	Available			
	R1	**R2**	**R3**	**R4**

	Allocation			
	R1	**R2**	**R3**	**R4**
P1	0	0	1	0
P2	2	0	0	1
P3	0	1	1	0
P4	1	1	0	0

Request			
R1	**R2**	**R3**	**R4**
2	0	0	2
1	3	0	1
2	1	1	0
4	0	3	1

Available			
R1	**R2**	**R3**	**R4**
2	1	1	2