COMP-3500

Introduction to Operating Systems

FROM:	Midas Oden
TO:	Dr. Xiao Qin
DATE:	October 29, 2021
LAB SECTION:	001

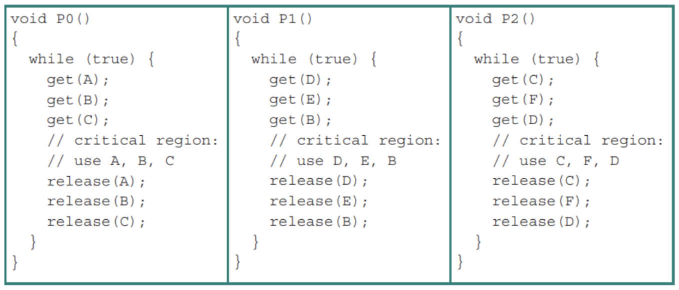
Homework #2

Banner ID: 903567097

Problem 1

1. [**40 points**]

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

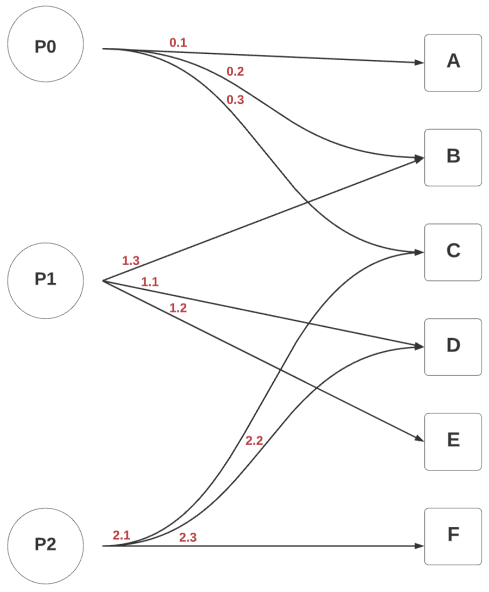


```
void P0()
{
  while (true) {
    get(A);
    get(B);
    get(C);
    // critical region:
    // use A, B, C
    release(A);
    release(B);
    release(C);
  }
}
```

```
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);
  }
}
```

a. Using a resource allocation graph, show the possibility of a deadlock in this implementation.



→ All three processes run on a single processor. The processor can execute one and only one process at a time. This means that if a process is executing its Get() or Release() function or critical region, no other process can execute.

→ After executing a statement in any process (such as get(), a statement in the critical section or release()), the operating system can switch to another process and execute the other process' next statement.

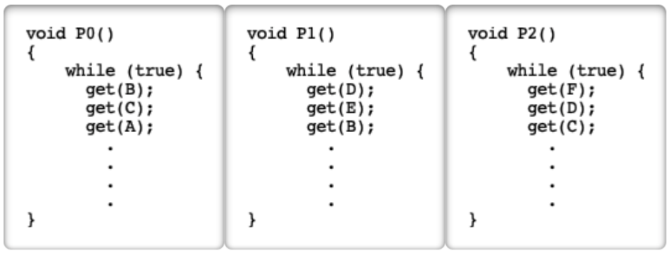
Deadlock occurs through events:

- **P$_0$ request A — Resource A is now free and assigned to** P_0
- **P$_1$ request D — Resource D is now free and assigned to** P_0
- **P$_2$ request C — Resource C is now free and assigned to** P_0
- **P$_0$ request B — Resource B is now free and assigned to** P_0

- **P$_1$ request E — Resource E is now free and assigned to P_0**
- **P$_2$ request F — Resource F is now free and assigned to P_0**
- **P$_0$ request C — Process P_0 waits for resource C to be free**
- **P$_1$ request B — Process P_1 waits for resource B to be free**
- **P$_2$ request D — Process P_2 waits for resource D to be free**

$\rightarrow$ At this point, we have reached a deadlock since all processes are in a cyclic waiting state — waiting for other preceding processes to release.

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.

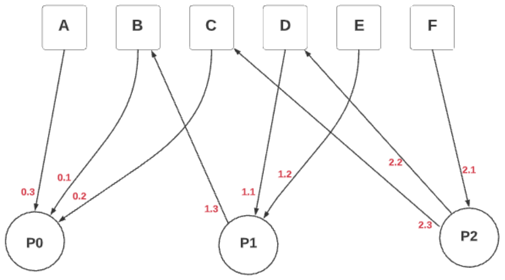


```
void P0()
{
    while (true) {
      get(B);
      get(C);
      get(A);
        .
        .
        .
        .
}
```

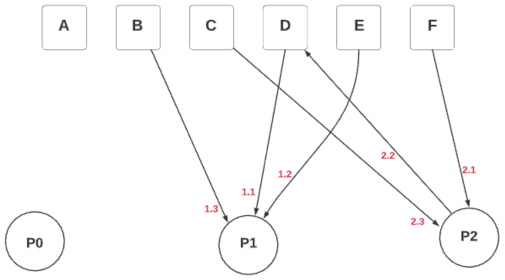
```
void P1()
{
    while (true) {
      get(D);
      get(E);
      get(B);
        .
        .
        .
        .
}
```

```
void P2()
{
    while (true) {
      get(F);
      get(D);
      get(C);
        .
        .
        .
        .
}
```

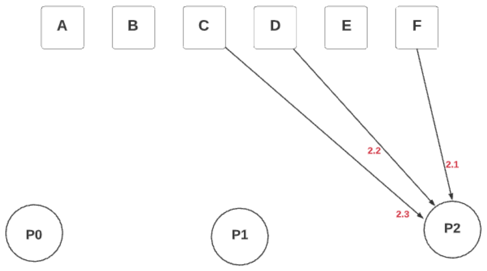
→ Process P_0 gets all its resource requirements fulfilled and hence will enter its critical section. After finishing execution in its critical section, P_0 will release all the resources held by it.



→ Resource B is now free and process P_1 is requesting for it — in which B will now be assigned to process P_1. Similarly, resource C will also be assigned to process P_1. Process P_1 now holds all required resources and will enter its critical section.



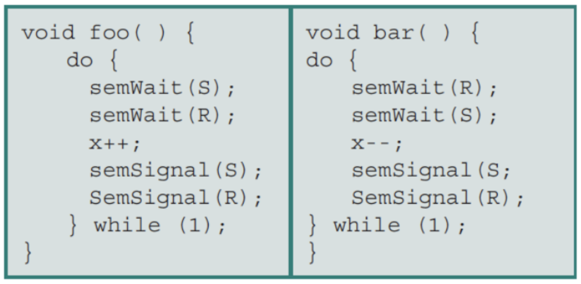
→ After exiting from its critical section, process P_1 will release all resources held by it. Process P_2 will be assigned resource D and will enter its own critical section. Hence, all processes will have completed their respective executions, resulting in the system being deadlock-free.



Problem 2

2. [**20 points**]

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

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

→ **The concurrent execution of these two processes results in both being blocked forever.**

→ **Initially, both semaphore variables S and R are available. If we were to suppose that once semWait(S) executes within foo( ), the value of S becomes 0. Also, once semWait(R) executes within bar( ), the value of R becomes 0.**

→ **Process foo( ) will block on semWait(R) since the value of R is 0, so it must wait until R becomes 1.**

→ **Process bar( ) will block on semWait(S) since value of S is 0, and must wait until it becomes 1.**

→ **Process foo( ) will indefinitely wait for R and bar( ) for S resulting in both being blocked forever since neither process will be able to proceed.**

Problem 3

3. [**20 points**]
What is the difference among deadlock avoidance, detection, and prevention?

- **Deadlock Avoidance**
  - **Mechanism ensures that the system does not enter an unsafe state.**
  - **A system is safe when it is possible to allocate resources to all processes in some order without causing a deadlock.**
  - **System requires information on the existing resources, resources availability, and resource requests to find whether the system is in a safe or unsafe state.**
  - **Requests for resources are manipulated until at lease one safe path is found.**
  - **The most common technique to avoid deadlock is the Banker's algorithm. It helps to find a safe path to execute all the processes. Also, this algorithm considers the total resources and current requested resources.**
  - **Other techniques include (1) Wait/Die and (2) Wound/Wait.**
  - **Can block processes for too long.**

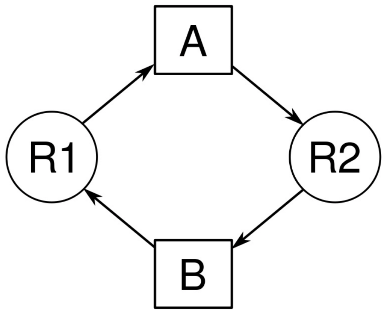


Figure 1: Process A holds resource R1 and requires R2 while Process B holds resource R2 and requires R1.

- **Deadlock Detection:**
  - **The goal is to detect the deadlock after it occurs or before it occurs.**
  - **Detecting the possibility of a deadlock before it occurs is much more difficult and is, in fact, generally undecidable. However, in specific environments, using specific means of locking resources, deadlock detection may be decidable.**
  - **The system does not requires additional information regarding the overall potential use of each resource for each process in all cases.**
  - **In order for the system to detect the deadlock condition it does not need to know all the details of all resources in existence, available and requested.**
  - **A deadlock detection technique example is model checking. This approach constructs a Finite State-model on which it performs a progress analysis and finds all possible terminal sets in the model.**
  - **Resource allocation strategy for deadlock detection is very liberal. Resources are granted as requested.**
  - **Needs to be invoked periodically to test for deadlock.**

- **Deadlock Prevention:**
  - **Mechanism ensures that at least one of the necessary conditions for deadlock can never occur.**
  - **System does not require information on the existing resources, the resource availability or the resource requests.**
  - **All resources are requested at once.**
  - **Examples of deadlock prevention algorithms include: (1) non-blocking synchronisation, (2) serialising tokens, and (3) Dijkstra's algorithm.**
  - **Resource allocation strategy for deadlock prevention is conservative, it under commits the resources.**
  - **It doesn't have any cost involved because it has to just make one of the conditions false so that deadlock doesn't occur.**
  - **Deadlock prevention has low device utilization.**

Problem 4

4. [20 points]

Consider a system consisting of four resources of the same type that are shared by three processes, each of which needs at most two resources. Show that the system is deadlock-free.

P0
P1
P2
R1
R2
R3
R4
MAX
2
AVAILABLE
4

→ We can show that the system is deadlock-free by using Proof of Contradiction. If we were to suppose that the system is currently in deadlock, this would imply that each process is holding a resource and is waiting for more.

→ Since we have three process and four resources, once process must be able to obtain two resources. This process requires no more resources and therefore it will return its resources when done.

→ The system is only in deadlock if a process cannot access the maximum amount of resources it requires. If three process need a maximum of 2 resources, and the system has 4 of those total — then none of those process must already have the max amount of resources it needs to cycle through. Once that process is finished, more resources will become free for other processes to execute.

→ In the worst-case scenario, all three process will be acquiring one resources and asking for the other one, while the fourth resources will fulfill the requirement of the processes, resulting in a deadlock-free system.