Alexandria University
Faculty of Engineering
Comp. & Comm. Engineering
CC373: Operating Systems

جامعة الاسكندرية
كلية الهندسة
برنامج هندسة الحاسب والاتصالات
مادة نظم التشغيل

Sheet6

CONCURRENCY: DEADLOCK AND STARVATION

1) Give examples of reusable and consumable resources.

Examples of reusable resources are processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases, and semaphores. Examples of consumable resources are interrupts, signals, messages, and information in I/O buffers.

2) What are the three conditions that must be present for deadlock to be possible?

Mutual exclusion. Only one process may use a resource at a time. Hold and wait. A process may hold allocated resources while awaiting assignment of others. No preemption. No resource can be forcibly removed from a process holding it.

3) What are the four conditions that create deadlock?

The above three conditions, plus: Circular wait. A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.

4) How can the hold-and-wait condition be prevented?

The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time, and blocking the process until all requests can be granted simultaneously.

5) List two ways in which the no-preemption condition can be prevented.

First, if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource. Alternatively, if a process requests a resource that is currently held by another process, the operating system may preempt the second process and require it to release its resources.

6) How can the circular wait condition be prevented?

The circular-wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type R, then it may subsequently request only those resources of types following R in the ordering.

7) What is the difference among deadlock avoidance, detection, and prevention?

Deadlock prevention constrains resource requests to prevent at least one of the four conditions of deadlock; this is either done indirectly, by preventing one of the three necessary policy conditions (mutual exclusion, hold and wait, no preemption), or directly, by preventing circular wait. Deadlock avoidance allows the three necessary conditions, but makes judicious choices to assure that the deadlock point is never reached. With deadlock detection, requested resources are granted to processes whenever possible; periodically, the operating system performs an algorithm that allows it to detect the circular wait condition.

8) Given the following state for the Banker's Algorithm.
6 processes P0 through P5
4 resource types: A (15 instances); B (6 instances); C (9 instances); D (10 instances)
Snapshot at time T0:

Available

A	B	C	D
6	3	5	4

	Current Allocation				Maximum Demand			
Process	A	B	C	D	A	B	C	D
P0	2	0	2	1	9	5	5	5
P1	0	1	1	1	2	2	3	3
P2	4	1	0	2	7	5	4	4
P3	1	0	0	1	3	3	3	2
P4	1	1	0	0	5	2	2	1
P5	1	0	1	1	4	4	4	4

a) Verify that the Available array has been calculated correctly.
15 – (2+0+4+1+1+1) = 6
6 – (0+1+1+0+1+0) = 3
9 – (2+1+0+0+0+1) = 5
10 – (1+1+2+1+0+1) = 4

b) Calculate the Need matrix.
Need Matrix = Max Matrix – Allocation Matrix

Need

Process	A	B	C	D
P0	7	5	3	4
P1	2	1	2	2
P2	3	4	4	2
P3	2	3	3	1
P4	4	1	2	1
P5	3	4	3	3

c) Show that the current state is safe, that is, show a safe sequence of processes. In addition, to the sequence show how the Available (working array) changes as each process terminates.
The following matrix shows the order in which the processes and shows what is available once the give process finishes)

Available

Process	A	B	C	D
P5	7	3	6	5
P4	8	4	6	5
P3	9	4	6	6
P2	13	5	6	8
P1	13	6	7	9
P1	15	6	9	10

d) Given the request (3, 2, 3, 3) from Process 5. Should this request be granted? Why or why not?

ANSWER is NO for the following reasons: IF this request were granted, then the new allocation matrix would be:

	Allocation			
Process	A	B	C	D
P0	2	0	2	1
P1	0	1	1	1
P2	4	1	0	2
P3	1	0	0	1
P4	1	1	0	0
P5	4	2	4	4

Then the new need matrix would be

	Allocation			
Process	A	B	C	D
P0	7	5	3	4
P1	2	1	2	2
P2	3	4	4	2
P3	2	3	3	1
P4	4	1	2	1
P5	0	2	0	0

And Available is then:

Available			
A	B	C	D
3	1	2	1

Which means we could NOT satisfy ANY process' need.

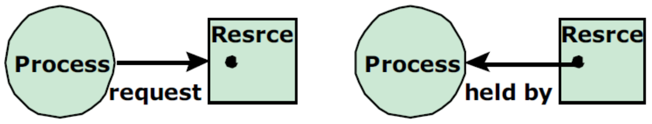
9) In the code below, three processes are competing for six resources labeled A to F.

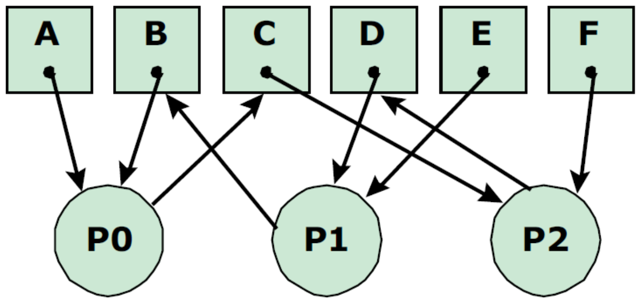
```
void P0()
{
   while(true) {
      get(A);
      get(B);
      get(C);
      // critical section
      // use A, B, C
      release(A);
      release (B);
      release (C);
   }
}
```

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

```
void P2()
{
   while(true) {
      get(C);
      get(F);
      get(D);
      // critical section
      // 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.





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.

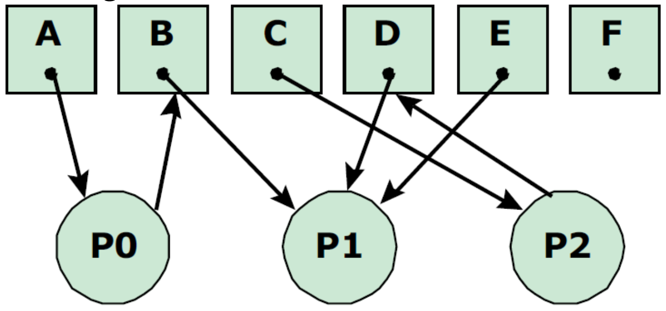
b) Modify the order of some of the get requests to prevent the possibility of any deadlock. You cannot move requests across procedures, only change the order inside each procedure. Use a resource allocation graph to justify your answer.

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.

10) 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.

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) 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. Note: Indefinite postponement is equivalent to starvation.

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.

11) Consider the following ways of handling deadlock: (1) banker's algorithm, (2) detect deadlock and kill thread, releasing all resources, (3) reserve all resources in advance, (4) restart thread and release all resources if thread needs to wait, (5) resource ordering, and (6) detect deadlock and roll back thread's actions.

a) One criterion to use in evaluating different approaches to deadlock is which approach permits the greatest concurrency. In other words, which approach allows the most threads to make progress without waiting when there is no deadlock? Give a rank order from 1 to 6 for each of the ways of handling deadlock just listed, where 1 allows the greatest degree of concurrency. Comment on your ordering.

In order from most-concurrent to least, there is a rough partial order on the deadlock-handling algorithms:

(1) detect deadlock and kill thread, releasing its resources
detect deadlock and roll back thread's actions
restart thread and release all resources if thread needs to wait

None of these algorithms limit concurrency before deadlock occurs, because they rely on runtime checks rather than static restrictions. Their effects after deadlock is detected are harder to characterize: they still allow lots of concurrency (in some cases they enhance it), but the computation may no longer be sensible or efficient. The third algorithm is the strangest, since so much of its concurrency will be useless repetition; because threads compete for execution time, this algorithm also prevents useful computation from advancing. Hence it is listed twice in this ordering, at both extremes.

(2) ***banker's algorithm***
resource ordering
These algorithms cause more unnecessary waiting than the previous two by restricting the range of allowable computations. The banker's algorithm prevents unsafe allocations (a proper superset of deadlock-producing allocations) and resource ordering restricts allocation sequences so that threads have fewer options as to whether they must wait or not.

(3) ***reserve all resources in advance***
This algorithm allows less concurrency than the previous two, but is less pathological than the worst one. By reserving all resources in advance, threads have to wait longer and are more likely to block other threads while they work, so the system-wide execution is in effect more linear.

(4) ***restart thread and release all resources if thread needs to wait***
As noted above, this algorithm has the dubious distinction of allowing both the most and the least amount of concurrency, depending on the definition of concurrency.

b) Another criterion is efficiency; in other words, which requires the least processor overhead. Rank order the approaches from 1 to 6, with 1 being the most efficient, assuming that deadlock is a very rare event. Comment on your ordering. Does your ordering change if deadlocks occur frequently?

In order from most-efficient to least, there is a rough partial order on the deadlock-handling algorithms:

(1) ***reserve all resources in advance***
resource ordering
These algorithms are most efficient because they involve no runtime overhead. Notice that this is a result of the same static restrictions that made these rank poorly in concurrency.

(2) ***banker's algorithm***
detect deadlock and kill thread, releasing its resources
These algorithms involve runtime checks on allocations which are roughly equivalent; the banker's algorithm performs a search to verify safety which is O(n m) in the number of threads and allocations, and deadlock detection performs a cycle-detection search which is O(n) in the length of resource-dependency chains. Resource-dependency chains are bounded by the number of threads, the number of resources, and the number of allocations.

(3) ***detect deadlock and roll back thread's actions***
This algorithm performs the same runtime check discussed previously but also entails a logging cost which is O(n) in the total number of memory writes performed.

(4) ***restart thread and release all resources if thread needs to wait***
This algorithm is grossly inefficient for two reasons. First, because threads run the risk of restarting, they have a low probability of completing. Second, they are competing with other restarting threads for finite execution time, so the entire system advances towards completion slowly if at all.

This ordering does not change when deadlock is more likely. The algorithms in the first group incur no additional runtime penalty because they statically disallow deadlock-producing execution. The second group incurs a minimal, bounded penalty when deadlock occurs. The algorithm in the third tier incurs the unrolling cost, which is O(n) in the number of memory writes performed between checkpoints. The status of the final algorithm is questionable because the algorithm does not allow deadlock to occur; it might be the case that unrolling becomes more expensive, but the behavior of this restart algorithm is so variable that accurate comparative analysis is nearly impossible.

12) Comment on the following solution to the dining philosophers problem.

a) A hungry philosopher first picks up his left fork; if his right fork is also available, he picks up his right fork and starts eating; otherwise he puts down his left fork again and repeats the cycle.

The philosophers can starve while repeatedly picking up and putting down their left forks in perfect unison.

b) A hungry philosopher picks up both forks at the same time (in a critical section), only if both forks are available.

Some philosophers may be unlucky to pick both forks at the same time. So it can cause starvation to some philosophers. It can be avoided with some sort of synchronization using queues (ones of semaphores or condition variables) with FIFO discipline.

13) Suppose that there are two types of philosophers. One type always picks up his left fork first (a “lefty”), and the other type always picks up his right fork first (a “righty”). The behavior of a lefty and a righty is defined as follows.

```
void lefty(int i)
{
   while(true) {
      think();
      wait( fork[i] );
      wait( fork[(i+1) % 5] );
      eat();
      signal( fork[(i+1) % 5] );
      signal( fork[i] );
   }
}
```

```
void righty(int i)
{
   while(true) {
      think();
      wait( fork[(i+1) % 5] );
      wait( fork[i] );
      eat();
      signal( fork[i] );
      signal( fork[(i+1) % 5] );
   }
}
```

Prove the following:

a) Any seating arrangement of lefties and righties with at least one of each avoids deadlock.

Assume that the table is in deadlock, i.e., there is a nonempty set D of philosophers such that each Pi in D holds one fork and waits for a fork held by neighbor. Without loss of generality, assume that Pj ∈ D is a lefty. Since Pj clutches his left fork and cannot have his right fork, his right neighbor Pk never completes his dinner and is also a lefty. Therefore, Pk ∈ D. Continuing the argument rightward around the table shows that all philosophers in D are lefties. This contradicts the existence of at least one righty. Therefore deadlock is not possible.

b) Any seating arrangement of lefties and righties with at least one of each prevents starvation.

Assume that lefty Pj starves, i.e., there is a stable pattern of dining in which Pj never eats. Suppose Pj holds no fork. Then Pj's left neighbor Pi must continually hold his right fork and never finishes eating. Thus Pi is a righty holding his right fork, but never getting his left fork to complete a meal, i.e., Pi also starves. Now Pi's left neighbor must be a righty who continually holds his right fork. Proceeding leftward around the table with this argument shows that all philosophers are (starving) righties. But Pj is a lefty: a contradiction. Thus Pj must hold one fork.

As Pj continually holds one fork and waits for his right fork, Pj's right neighbor Pk never sets his left fork down and never completes a meal, i.e., Pk is also a lefty who starves. If Pk did not continually hold his left fork, Pj could eat; therefore Pk holds his left fork. Carrying the argument rightward around the table shows that all philosophers are (starving) lefties: a contradiction. Starvation is thus precluded.
