CSc33200: Operating Systems, CS-CCNY, Fall 2003   Jinzhong Niu October 24, 2003

Concurrency: Deadlock and Starvation

1 The conditions for deadlock

We have seen many examples that may result in *deadlock*, which may be defined as the permanent blocking of a set of processes that either compete for system resources or communicate with each other.

Take two processes, P and Q, as an example. Both processes require resources, A and B. They execute respectively in the following ways:

```
Process P:                          Process Q:
  ...                                 ...
  Get A                               Get B
  Get B                               Get A
  ...                                 ...
  do_something()                      do_something_else()
  ...                                 ...
  Release A                           Release B
  Release B                           Release A
  ...                                 ...
```

Clearly, if P obtains A at the same time Q obtains B, a deadlock occurs since neither of them can proceed to obtain the other resource they need.

If we examine this example and many others, we may find the following conditions that must be present for a deadlock to be possible:

1. **Mutual exclusion**. Only one process may use the shared resource at a time.
2. **Hold and wait**. More than one resource is involved and they are to be obtained one by one. That is processes may hold allocated resources while awaiting assignment of others.
3. **No preemption**. Once a resource is held by a process, it cannot be forcibly removed from the process.

Note that strictly speaking these are not absolutely necessities, since a deadlock may also occur when no resource is even involved. We have mentioned an example before in which two processes make RPC requests to each other before they receive the requests, then they will both be blocked waiting for response from the other. In the situation, no progress could be made and a deadlock is resulted in, however no resource is required in the environment. We would rather to view the above conditions are necessary in the sense that resource sharing is the most common case that leads to deadlocks.

These conditions are said necessary but no sufficient for a deadlock to happen. To make sure a deadlock will happen or did happen, a fourth condition is required:

4. **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. In terms of graph theory, there is a directed circuit, in which processes and resources alternate exactly one by one, as Figure 1 depicts.

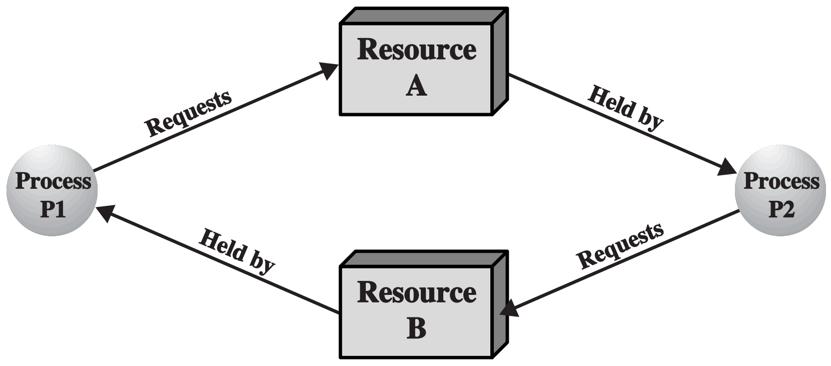

Figure 1: Circular wait

The fourth condition is actually a potential consequence of the first three. If related concurrent processes are coded in a specific way like the first example we gave above, then circular waiting may happen. Once circular wait occurs, a deadlock is obvious unavoidable.

2 Deadlock prevention

A deadlock is of course not desirable, so naturally we need some methods to prevent deadlocks. A simple strategy is to prevent the occurrence of one of the above four conditions.

Mutual exclusion

In general, this condition cannot be disallowed. For example, it is not realistic for two processes to print on a same sheet of paper at the same time.

Hold and wait

This condition is preventable since a process may request for the resources it needs all at once. If its need can be met, then the operating system just does it, otherwise blocks the process until all the resources are available. However this method raises two problems:

- Processes may need some resources for part of their lives. It is inefficient for them to own all the resources all the time.
- In some cases, it is impossible for a process to know in advance what resources it will need during its execution.

In a modular application, it is unimaginable for a programmer to put all the code making requests for resources in `main()` instead of the various modules where those requests are supposed to be made.

No preemption

This condition can be prevented in two ways. First, if a process is denied a further request while it has already held some resources, it must release all them and request them again together with the additional resources. Alternatively, if a process requests a resource that is currently held by another process, the operating system may preempt the resource from the second process and allocate it to the first one. Note that this approach is practical only when the resource of concern could be in some way restored to its original state.

Circular wait

This condition is also preventable. Let's go back to the first example, but change the code of process Q a little bit as follows:

```
Process P:                          Process Q:
  ...                                 ...
  Get A                               Get A
  Get B                               Get B
  ...                                 ...
  do_something()                      do_something_else()
  ...                                 ...
  Release A                           Release B
  Release B                           Release A
  ...                                 ...
```

That is all processes request for the resources in the same order, thus the one who succeeds at the first step will be allocated all the resources first. Other processes have no chance until those resource are released.

In a systematic way, we may assign an index to each resource in the system, and all the processes are required to request for the resources in the order of increasing index. However again this method may be inefficient. For example, a process may need to manipulate a resource with a larger index far before another resource associated with a smaller index is needed. To comply with this hold-and-wait prevention strategy, the second resource will have to be requested first, thus be held without utilization for a long time.

3 Deadlock avoidance

Different from *deadlock prevention*, where one of the four necessary conditions is prevented in some way, *deadlock avoidance* takes another approach, which is the progress of resource allocation in the operating system is monitored dynamically and whenever a deadlock is going to happen, some measure is taken to avoid it. Thus an evaluation process should be performed, when a resource allocation is requested, to make sure a deadlock will not happen.

3.1 Resource allocation model

To do such an evaluation, if there are totally n processes and m different types of resources in the system, then the availability of the resources and the processes' needs for resources may are presented in the following vectors and matrices, where C_{ij} is the requirement of process i for resource j and A_{ij} the current allocation of resource j to process i:

Resources	=	$(R_1, R_2, \ldots, R_m)$	total amount of each resource in the system
Available resources	=	$(V_1, V_2, \ldots, V_m)$	total amount of each resource not allocated to a process
Claims	=	$\begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & \cdots & C_{nm} \end{pmatrix}$	requirement of each process for each resource
Allocations	=	$\begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & \cdots & A_{nm} \end{pmatrix}$	current allocations

The following relationships can be observed easily:

1. $R_i = V_i + \sum_{k=1}^{n} A_{ki}$, for all i: All resources are either available or allocated.
2. $C_{ki} \leq R_i$, for all k: No process can claim more than the total amount of resource in the system.
3. $A_{ki} \leq C_{ki}$, for all k, i: No process is allocated more resources of any type than the process originally claimed to need.

3.2 The banker's algorithm

Based on the above vectors and matrices, we introduce an algorithm of deadlock avoidance, called the *banker's algorithm*, that was first proposed Dijkstra in 1960s. It is named in such a way because in a bank a same situation exists where the bank has to skillfully satisfy customers' request for loan at the same time keeping itself from bankruptcy.

Terms

We define the **state** of the system the current allocation of resources to processes, thus the state may be fully defined by the above two vectors and two matrices. A **safe state** is one in which there is at least one sequence that does not result in a deadlock (i.e., all of the processes can run to completion finally). An **unsafe state** is of course a state that is not safe.

Note that by definition, a system may still reach a deadlock from a safe state by following some sequence, since a safe state merely needs one safe case and does not care what may be led to in other cases. But it is sufficient to avoid deadlock if all the states of the system are guaranteed to be safe.

Examples

The idea of the algorithm may be illustrated by examples as follows. Suppose there are 4 processes, P_1, P_2, P_3, and P_4, and 3 types of resources, R_1, R_2, and R_3, whose amounts are respectively 9, 3, and 6. Figure 2 (a) depicts a state in which 1 unit of R_2 and 1 unit of R_3 are still available.

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	6

Resource Vector

R1	R2	R3
0	1	1

Available Vector

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
6	2	3

Available Vector

(b) P2 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation Matrix

R1	R2	R3
7	2	3

Available Vector

(c) P1 runs to completion

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim Matrix

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation Matrix

R1	R2	R3
9	3	4

Available Vector

(d) P3 runs to completion

Figure 2: Determination of a safe state

Before we determine if further request for resources should be satisfied at the initial state, we need first answer whether it is safe or not. To obtain the answer, we need to examine if any of the four processes be run to completion with the resources available.

Clearly, P_1 cannot be satisfied since it requires 2 more units of R_1 but none available. On the contrary, P_2 can run to completion after allocated 1 more unit of R_3. Suppose that this is accomplished, then we have the resulting state depicted in Figure 2 (b). Now we again need to answer if all the processes can be completed (i.e., if the resulting state is safe). In this case, each of the remaining processes could be completed. Suppose we choose P_1 to run first until its completion, then we arrive at the state shown in Figure 2 (c). Then we may choose P_3, resulting respectively in the state of Figure 2 (d). Finally we can complete P_4. Thus we have actually found a sequence that leads to the completion of all the processes. So the state of Figure 2 (a) is a safe one.

Let us consider the state of Figure 3, which may be viewed as one preceding the state of Figure 2 (a) before 1 unit of R_1 and 1 unit of R_3 are allocated to P_2. If at this state, P_2 does make the request, obviously it is safe to satisfy the request since the resulting state is safe. But let us consider alternatively that P_1 requests for an additional unit each of R_1 and R_3. If we grant the request, the state of Figure 3 (b) is obtained. Obviously at this point, no process can run to completion since each will need at least one unit of R_1, and there are none available. Thus, the request by P_1 should be denied, and P_1 be blocked.

The strategy we use above is: When a process makes a request for a set of resources, assume first that the request is granted and update the system state accordingly, then determine if the result is a safe state. If so, grant the request; otherwise block the process until it is safe to grant the request.

Note that the requirement of this strategy is over strict since we assume that processes will hold the allocated resources to completion. Actually they may release some of their resources in the middle of their execution for other processes' use. So an unsafe state may even be *safe*. Our strategy is simply sufficient for avoiding deadlock though probably more than necessary.

The following gives the skeleton of the banker's algorithm:

```
struct state {
  int resource[m];
  int available[m];
  int claim[n][m];
  int allocation[n][m];
};
```

Claim Matrix

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation Matrix

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Resource Vector

R1	R2	R3
9	3	6

Available Vector

R1	R2	R3
1	1	2

(a) Initial state

Claim Matrix

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Allocation Matrix

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Available Vector

R1	R2	R3
0	1	1

(b) P1 requests one unit each of R1 and R3

Figure 3: Determination of an unsafe state

```
/**
 * i: the index of the requesting process.
 * request: a vector expressing a resource request by Pi.
 */
boolean allocate(int i, int request[*]) {
  boolean safe = true;

  if (allocation[i, *] + request[*] > claim[i, *])
    <error>;
    return false;
  else if (request[*] > available[*])
    <block process Pi>
    return false;
  else {
    <define newstate by
      allocation[i, *] = allocation[i, *] + request[*];
      available[i, *] = available[i, *] - request[*];
    >;
  }

  if (safe(newstate))
    <carry out the allocation>;
    return true;
  else {
    <restore original state>;
    <block process Pi>;
```

```
      return false;
    }
  }

  boolean safe(state S) {
    int currentavail[m];

    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;

    while (possible) {
      find a Pk in rest such that
        claim[k, *] - allocation[k, *] <= currentavail;
      if (found) {
        currentavail = currentavail + alloc[k, *];
        rest = rest - {Pk};
      } else
        possible = false;
    }

    return (rest == null);
  }
```

Although deadlock avoidance has the advantage that it is not necessary to preempt and rollback processes, as in deadlock detection that we will cover, and is less restrictive than deadlock prevention, it has a number of disadvantages:

- The maximum resource requirement for each process can hardly be determined in advance.
- The resources involved should remain static. For example, a resource available cannot become unavailable suddenly due to some reason.
- The processes under consideration must be independent. That is if some processes need to communicate with each other, the banker's algorithm cannot guarantee a deadlock will be avoided.

4 Deadlock detection

It is possible to attack deadlock even after a deadlock has happened. The deadlock detection approach is quite different from deadlock prevention in the sense that it does not limit resource access or restrict process actions. What is needed is that the operation system performs an algorithm periodically that detects the existence of circular wait.

4.1 Deadlock detection algorithm

Suppose a request matrix, Q, is defined such that Q_{ij} represents the amount of resources of type j requested by process i, and the Allocation matrix, A, and Available vector, V, presented above are also defined. Then a deadlock detection algorithm, which marks processes that are not deadlocked, goes as follows:

1. Mark each process that has a row of all zeros in the Allocation matrix, A.
2. Initialize a temporary vector W, to equal V.
3. Find a process, P_i, such that it is currently unmarked and the ith row of Q is less than or equal to W. That is $Q_{ik} \leq W_k$, for $1 \leq k \leq m$. If no such row is found, terminate the algorithm.
4. If such a row if found, mark P_i and add the corresponding row of the allocation matrix to W. That is, set $W_k = W_k + A_{ik}$, for $1 \leq k \leq m$. Return to step 3.

Clearly a deadlock exists if and only if there are unmarked processes at the end of the algorithm. Each of them is deadlocked. The strategy of this algorithm is to check all the processes whether they may run to completion one by one if their requests are granted. If there is a way, then all the processes will be marked and no deadlock is detected. Take the state of Figure **??** as an example. Then P_4, P_3 will be marked sequentially, but P_1 and P_2 not, which shows the latter two are deadlocked.

Note that actually Q may be used instead in step 1 of the algorithm, or equivalently the whole step may be omitted (Why? What is the difference between the original version and the revised one?).

4.2 Recovery

Once a deadlock is detected, some strategy is needed for recovery. The possible approaches are:

1. Abort all deadlocked processes.

2. Rollback each deadlocked process to some previously defined checkpoint where no deadlock was detected, and restart them. Though the processes may execute in the same sequence as before and result in a deadlock again, the nondeterminism of concurrent processing may probably lead to another sequence and involve no deadlock.
3. Successively abort deadlocked processes until deadlock no longer exists. The processes should be selected to abort on the basis of minimum cost.
4. Successively rollback deadlocked processes by preempting resources from them until deadlock no longer exists. A cost-based selection should also be used.

Many factors may be considered to minimize the abortion or rollback cost: process priority, time that has been consumed by a process, time that will be needed by a process, etc.

5 A classic example: dining philosophers problem

The dining philosophers problem is another classic problem besides the producer / consumer problem. As Figure 4 shows, five philosophers meet together to think about

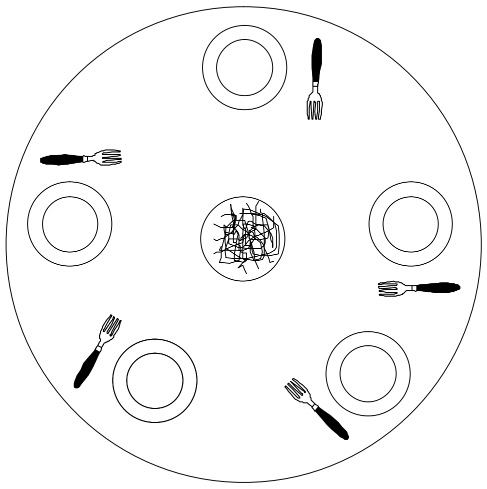

Figure 4: Dining arrangement for philosophers

what you have mutual interests on and eat spaghetti. They may be busy thinking or busy eating, and nothing else. When they feel hungry, they may feel free to eat by using the two forks respectively on the left and on the right; after completion, the forks are replaced on the table. Two adjacent philosophers share the fork between them on the table.

This problem is of interest since every philosopher may be viewed as a concurrent process and they share global resources - forks. The first solution to this problem is given in Figure 5. In the algorithm, every philosopher picks up first the fork on the left and then the fork on the right. It unfortunately leads to deadlock: If all of the philosophers are hungry at the same time, they all pick up the fork on their left, and then they all reach out for the other fork, which is not there. Thus all philosophers starve in this case.

```
/* program dining philosophers */
semaphore fork[5] = {1};

/**
 * i: 0 .. 4
 */
void philosopher(int i) {
  while (true) {
    think();
    wait(fork[i]);
    wait(fork[(i+1) mod 5];
    eat();
    signal(fork[(i+1) mod 5];
    signal(fork[i]);
  }
}

void main() {
  parbegin(philosopher(0), philosopher(1), philosopher(2),
           philosopher(3), philosopher(4));
}
```

Figure 5: A first solution to Dining philosophers problem

To overcome the risk of deadlock, we could allow only at most four philosophers at a time to begin to eat, thus at least one philosopher will have two forks. Figure 6 shows the solution, which is free of deadlock and starvation.

```
/* program dining philosophers */
semaphore fork[5] = {1};
semaphore token = {4};

void philosopher(int i) {
  while (true) {
    think();
    wait(token);
    wait(fork[i]);
    wait(fork[(i+1) mod 5];
    eat();
    signal(fork[(i+1) mod 5];
    signal(fork[i]);
    signal(token);
  }
}

void main() {
  parbegin(philosopher(0), philosopher(1), philosopher(2),
           philosopher(3), philosopher(4));
}
```

Figure 6: A second solution to Dining philosophers problem