OSD 19CS2106S
Session – 39
Deadlocks
Lecture Notes

Mutex Deadlocks

Sometimes, a thread needs to simultaneously access two or more different shared resources, each of which is governed by a separate mutex. When more than one thread is locking the same set of mutexes, deadlock situations can arise. Figure shows an example of a deadlock in which each thread successfully locks one mutex, and then tries to lock the mutex that the other thread has already locked. Both threads will remain blocked indefinitely.

Thread A	**Thread B**
1. pthread_mutex_lock(mutex1);	1. pthread_mutex_lock(mutex2);
2. pthread_mutex_lock(mutex2);	2. pthread_mutex_lock(mutex1);
blocks	blocks

Figure: A deadlock when two threads lock two mutexes

The simplest way to avoid such deadlocks is to define a mutex hierarchy. When threads can lock the same set of mutexes, they should always lock them in the same order. For example, in the scenario in Figure, the deadlock could be avoided if the two threads always lock the mutexes in the order mutex1 followed by mutex2. Sometimes, there is a logically obvious hierarchy of mutexes. However, even if there isn't, it may be possible to devise an arbitrary hierarchical order that all threads should follow.

Program to demonstrate deadlock.

```
/* deadlock.c */
/* On running this, nothing will be printed as both the threads are waiting to get the lock
acquired by the other process. Hence, never enter their critical sections.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
/* Create and initialize the mutex locks */
pthread_mutex_t m1 = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t m2 = PTHREAD_MUTEX_INITIALIZER;
/* function both threads will execute in */
void* worker(void* arg) {
   if ((long long) arg == 1) { /* true for first thread  */
    /* Get the first mutex lock if it's available */
        pthread_mutex_lock(&m1);
    /* Get the second mutex lock if it's available */
        pthread_mutex_lock(&m2);
    /* Critical section start */
    printf("Inside Thread 1\n");
    /* Critical section ends */
  } else {  /* true for second thread  */
    /* Get the second mutex lock if it's available */
    pthread_mutex_lock(&m2);
    /* Get the first mutex lock if it's available */
    pthread_mutex_lock(&m1);
    /* Critical section start */
    printf("Inside Thread 2\n");
    /* Critical section ends */
  }
  /* Release the locks */
 pthread_mutex_unlock(&m1);
 pthread_mutex_unlock(&m2);
 /* Exit the thread */
 pthread_exit(0);
}
/* Note the order of waiting on the locks.
The second thread gets the second mutex lock and waits for the first mutex lock, which is
held by the first thread, which is waiting for the second mutex lock to be released. This
leads to DEADLOCK. */
int main(int argc, char *argv[]) {
 pthread_t p1, p2; /* the thread identifiers */
/* Create thread one */
 pthread_create(&p1, NULL, worker, (void *) (long long) 1);
/* Create thread two */
 pthread_create(&p2, NULL, worker, (void *) (long long) 2);
/* Wait for the two threads to complete */
 pthread_join(p1, NULL);
 pthread_join(p2, NULL);
 return 0;
}
/*
vishnu@mannava:~/threads$ cc deadlock.c -lpthread
vishnu@mannava:~/threads$ ./a.out
```

Semaphores	*Condition Variables*
Can be used anywhere in a program, but should not be used in a monitor	**Can only be used in monitors**
Wait() does not always block the caller (*i.e.*, when the semaphore counter is greater than zero).	**Wait() always blocks the caller.**
Signal() either releases a blocked thread, if there is one, or increases the semaphore counter.	**Signal() either releases a blocked thread, if there is one, or the signal is lost as if it never happens.**
If Signal() releases a blocked thread, the caller and the released thread *both* continue.	**If Signal() releases a blocked thread, the caller yields the monitor (Hoare type) or continues (Mesa Type). Only one of the caller or the released thread can continue, but not both.**

Deadlocks:

There are four conditions that are necessary to achieve deadlock:

1. **Mutual Exclusion** - At least one resource must be held in a non-sharable mode; If any other process requests this resource, then that process must wait for the resource to be released.
2. **Hold and Wait** - A process must be simultaneously holding at least one resource and waiting for at least one resource that is currently being held by some other process.
3. **No preemption** - Once a process is holding a resource (i.e. once its request has been granted), then that resource cannot be taken away from that process until the process voluntarily releases it.
4. **Circular Wait** - A set of processes { P_0, P_1, P_2, . . ., P_N } must exist such that every $P_{[i]}$ is waiting for $P_{[(i+1)\%(N+1)]}$. (Note that this condition implies the hold-and-wait condition, but it is easier to deal with the conditions if the four are considered separately.)

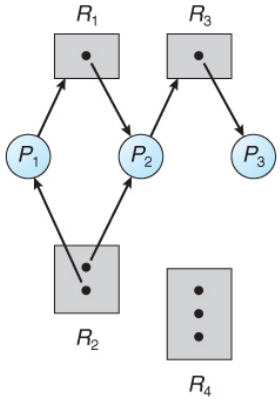
Resource-Allocation Graph

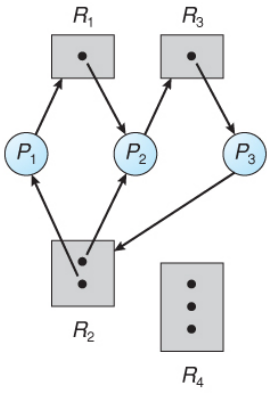
• **Request Edges -** A set of directed arcs from P_i to R_j, indicating that process P_i has requested R_j, and is currently waiting for that resource t**o** become available.

• **Assignment Edges -** A set of directed arcs from R_j to P_i indicating that resource R_j has been allocated to process P_i, and that P_i is currently holding resource R_j.

• Note that a **request edge** can be converted into an **assignment edge** by reversing the direction of the arc when the request is granted. (However note also that request edges point to the category box, whereas assignment edges emanate from a particular instance dot within the box.) For example:

- If a resource-allocation graph contains no cycles, then the system is not deadlocked. (When looking for cycles, remember that these are ***directed*** graphs.) See the example in Figure 7.2 above.
- If a resource-allocation graph does contain cycles **AND** each resource category contains only a single instance, then a deadlock exists.
- If a resource category contains more than one instance, then the presence of a cycle in the resource-allocation graph indicates the *possibility* of a deadlock, but does not guarantee one. Consider, for example, Figures 7.3 and 7.4 below:





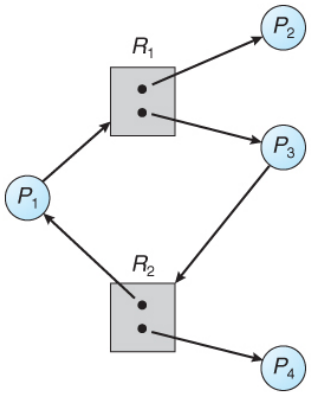


Figure 7.1 - Resource allocation graph,
Figure 7.2 - graph with a deadlock,
Figure 7.3 – graph with a cycle but no deadlock

Methods for Handling Deadlocks

- Generally speaking there are three ways of handling deadlocks:
  1. Deadlock prevention or avoidance - Do not allow the system to get into a deadlocked state.
  2. Deadlock detection and recovery - Abort a process or preempt some resources when deadlocks are detected.
  3. Ignore the problem all together - If deadlocks only occur once a year or so, it may be better to simply let them happen and reboot as necessary than to incur the constant overhead and system performance penalties associated with deadlock prevention or detection. This is the approach that both Windows and UNIX take.

Deadlock Avoidance

- In some algorithms the scheduler only needs to know the *maximum* number of each resource that a process might potentially use. In more complex algorithms the scheduler can also take advantage of the *schedule* of exactly what resources may be needed in what order.
- When a scheduler sees that starting a process or granting resource requests may lead to future deadlocks, then that process is just not started or the request is not granted.
- A resource allocation **state** is defined by the number of available and allocated resources, and the maximum requirements of all processes in the system.

Safe State

- A state is ***safe*** if the system can allocate all resources requested by all processes (up to their stated maximums) without entering a deadlock state.
- More formally, a state is safe if there exists a ***safe sequence*** of processes $\{P_0, P_1, P_2, ..., P_N\}$ such that all of the resource requests for Pi can be granted using the resources currently allocated to Pi and all processes Pj where j < i. (I.e. if all the processes prior to Pi finish and free up their resources, then Pi will be able to finish also, using the resources that they have freed up.)
- If a safe sequence does not exist, then the system is in an unsafe state, which ***MAY*** lead to deadlock. (All safe states are deadlock free, but not all unsafe states lead to deadlocks.)

Banker's Algorithm

- For resource categories that contain more than one instance the resource-allocation graph method does not work, and more complex (and less efficient) methods must be chosen.

Banker's Algorithm for One Resource

This is modelled on the way a small town banker might deal with customers' lines of credit. In the course of conducting business, our banker would naturally observe that customers rarely draw their credit lines to their limits. This, of course, suggests the idea of extending more credit than the amount the banker actually has in her coffers.

Suppose we start with the following situation

Customer	Credit Used	Credit Line
Andy	**0**	**6**
Barb	**0**	**5**
Marv	**0**	**4**
Sue	**0**	**7**
Funds Available	**10**	
Max Commitment		**22**

Our banker has 10 credits to lend, but a possible liability of 22. Her job is to keep enough in reserve so that ultimately each customer can be satisfied over time: That is, that each customer will be able to access his full credit line, just not all at the same time. Suppose, after a while, the bank's credit line book shows

Customer	Credit Used	Credit Line
Andy	1	6
Barb	1	5
Marv	2	4
Sue	4	7
Funds Available	2	
Max Commitment		22

Eight credits have been allocated to the various customers; two remain. The question then is: Does a way exist such that each customer can be satisfied? Can each be allowed their maximum credit line in some sequence? We presume that, once a customer has been allocated up to his limit, the banker can delay the others until that customer repays his loan, at which point the credits become available to the remaining customers. If we arrive at a state where **no customer** can get his maximum because not enough credits remain, then a deadlock *could* occur, because the first customer to ask to draw his credit to its maximum would be denied, and all would have to wait.

To determine whether such a sequence exists, the banker finds the customer closest to his limit: If the remaining credits will get him to that limit, The banker then assumes that that loan is repaid, and proceeds to the customer next closest to his limit, and so on. If all can be granted a full credit, the condition is *safe*.

In this case, Marv is closest to his limit: assume his loan is repaid. This frees up 4 credits. After Marv, Barb is closest to her limit (actually, she's tied with Sue, but it makes no difference) and 3 of the 4 freed from Marv could be used to award her maximum. Assume her loan is repaid; we have now freed 6 credits. Sue is next, and her situation is identical to Barb's, so assume her loan is repaid. We have freed enough credits (6) to grant Andy his limit; thus this state safe.

Suppose, however, that the banker proceeded to award Barb one more credit after the credit book arrived at the state immediately above:

Customer	Credit Used	Credit Line
Andy	**1**	**6**
Barb	**2**	**5**
Marv	**2**	**4**
Sue	**4**	**7**
Funds Available	**1**	
Max Commitment		**22**

Now it's easy to see that the remaining credit could do no good toward getting anyone to their maximum.

So, to recap, the banker's algorithm looks at each request as it occurs, and tests if granting it will lead to a safe state. If not, the request is delayed. To test for a safe state, the banker checks to see if enough resources will remain after granting the request to

satisfy the customer closest to his maximum. If so, that loan is assumed repaid, and the next customer checked, and so on. If all loans can be repaid, then the request leads to a safe state, and can be granted. In this case, we see that if Barb is awarded another credit, Marv, who is closest to his maximum, cannot be awarded enough credits, hence Barb's request can't be granted —it will lead to an unsafe state[3].

Banker's Algorithm for Multiple Resources

Suppose, for example, we have the following situation, where the first table represents resources assigned, and the second resources still required by five processes, A, B, C, D, and E.

Resources Assigned				
Processes	**Tapes**	**Plotters**	**Printers**	**Toasters**
A	**3**	**0**	**1**	**1**
B	**0**	**1**	**0**	**0**
C	**1**	**1**	**1**	**0**
D	**1**	**1**	**0**	**1**
E	**0**	**0**	**0**	**0**
Total Existing	**6**	**3**	**4**	**2**
Total Claimed by Processes	5	3	2	2
Remaining Unclaimed	**1**	**0**	**2**	**0**

Resources Still Needed				
Processes	**Tapes**	**Plotters**	**Printers**	**Toasters**
A	**1**	**1**	**0**	**0**
B	**0**	**1**	**1**	**2**
C	**3**	**1**	**0**	**0**
D	**0**	**0**	**1**	**0**
E	**2**	**1**	**1**	**0**

The vectors ***E, P*** and ***A*** represent Existing, Possessed and Available resources respectively:

```
E = (6, 3, 4, 2)
P = (5, 3, 2, 2)
A = (1, 0, 2, 0)
```

Notice that

```
A = E - P
```

Now, to state the algorithm more formally, but in essentially the same way as the example with Andy, Barb, Marv and Sue:

1. Look for a row whose unmet needs don't exceed what's available, that is, a row where ***P*** <= ***A***; if no such row exists, we are deadlocked because no process can acquire the resources it needs to run to completion. If there's more than one such row, just pick one.
2. Assume that the process chosen in 1 acquires all the resources it needs and runs to completion, thereby releasing its resources. Mark that process as virtually terminated and add its resources to ***A***.
3. Repeat 1 and 2 until all processes are either virtually terminated (safe state), or a deadlock is detected (unsafe state).

Going thru this algorithm with the foregoing data, we see that process D's requirements are smaller than ***A***, so we virtually terminate D and add its resources back into the available pool:

```
E = (6, 3, 4, 2)
P = (5, 3, 2, 2) - (1, 1, 0, 1) = (4, 2, 2, 1)
A = (1, 0, 2, 0) + (1, 1, 0, 1) = (2, 1, 2, 1)
```

Now, A's requirements are less than ***A***, so do the same thing with A:

```
P = (4, 2, 2, 1) - (3, 0, 1, 1) = (1, 2, 1, 0)
A = (2, 1, 2, 1) + (3, 0, 1, 1) = (5, 1, 3, 2)
```

At this point, we see that there are no remaining processes that can't be satisfied from available resources, so the illustrated state is *safe*.

In order to apply the Banker's algorithm, we first need an algorithm for determining whether or not a particular state is safe.

- This algorithm determines if the current state of a system is safe, according to the following steps:
  1. Let Work and Finish be vectors of length m and n respectively.
     a. Work is a working copy of the available resources, which will be modified during the analysis.
     b. Finish is a vector of booleans indicating whether a particular process can finish. (or has finished so far in the analysis.)
     c. Initialize Work to Available, and Finish to false for all elements.
  2. Find an i such that both (A) Finish[i] == false, and (B) Need[i] < Work. This process has not finished, but could with the given available working set. If no such i exists, go to step 4.
  3. Set Work = Work + Allocation[i], and set Finish[i] to true. This corresponds to process i finishing up and releasing its resources back into the work pool. Then loop back to step 2.
  4. If finish[i] == true for all i, then the state is a safe state, because a safe sequence has been found.

Resource-Request Algorithm (The Bankers Algorithm)

- Now that we have a tool for determining if a particular state is safe or not, we are now ready to look at the Banker's algorithm itself.
- This algorithm determines if a new request is safe, and grants it only if it is safe to do so.
- When a request is made (that does not exceed currently available resources), pretend it has been granted, and then see if the resulting state is a safe one. If so, grant the request, and if not, deny the request, as follows:
    1. Let Request[n][m] indicate the number of resources of each type currently requested by processes. If Request[i] > Need[i] for any process i, raise an error condition.
    2. If Request[i] > Available for any process i, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
    3. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must wait until its request can be granted safely.The procedure for granting a request (or pretending to for testing purposes) is:
        - Available = Available - Request
        - Allocation = Allocation + Request
        - Need = Need - Request

An Illustrative Example

- Consider the following situation:

	Allocation	Max	Available	Need
	A B C	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2	7 4 3
P_1	2 0 0	3 2 2		1 2 2
P_2	3 0 2	9 0 2		6 0 0
P_3	2 1 1	2 2 2		0 1 1
P_4	0 0 2	4 3 3		4 3 1

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- And now consider what happens if process P1 requests 1 instance of A and 2 instances of C. (Request[1] = (1, 0, 2))
- What about requests of (3, 3,0) by P4? or (0, 2, 0) by P0? Can these be safely granted? Why or why not?

/* Dining philosophers Problem solution */

HOLD AND WAIT STRATEGY: THE LR ALGORITHM

***The LR algorithm*:** The philosophers are assigned fork acquisition strategies as follows: the philosophers whose numbers are even are R-type, and the philosophers whose numbers are odd are L-type. Concurrency in the context of the dining philosophers problem is a measure of how many philosophers can eat simultaneously (in the worst case) when all the philosophers are hungry. Having high concurrency is a most desirable property. The LR algorithm does not achieve high concurrency: it is possible to get into a situation in which only a *quarter* of the philosophers will be able to eat simultaneously when all of them are hungry.

```
/* Implementation in C  phil-lr.c:
Odd numbered philosophers pick up the right chopstick first and then the left, while even
numbered philosophers pick up the left chopstick first and then the right. */
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#define NUMBER_OF_PHILOSOPHERS 5
void *philosopher(void *);
void think(int);
void pickUp(int);
void eat(int);
void putDown(int);
pthread_mutex_t chopsticks[NUMBER_OF_PHILOSOPHERS];
pthread_t philosophers[NUMBER_OF_PHILOSOPHERS];
pthread_attr_t attributes[NUMBER_OF_PHILOSOPHERS];
int main() {
	int i;
	srand(time(NULL));
	for (i = 0; i < NUMBER_OF_PHILOSOPHERS; ++i) {
		pthread_mutex_init(&chopsticks[i], NULL);
	}
	for (i = 0; i < NUMBER_OF_PHILOSOPHERS; ++i) {
		pthread_attr_init(&attributes[i]);
	}

	for (i = 0; i < NUMBER_OF_PHILOSOPHERS; ++i) {
		pthread_create(&philosophers[i], &attributes[i], philosopher, (void *)(i));
	}
	for (i = 0; i < NUMBER_OF_PHILOSOPHERS; ++i) {
		pthread_join(philosophers[i], NULL);
	}
	return 0;
```

```
}
void *philosopher(void *philosopherNumber) {
	while (1) {
		think(philosopherNumber);
		pickUp(philosopherNumber);
		eat(philosopherNumber);
		putDown(philosopherNumber);
	}
}
void think(int philosopherNumber) {
	int sleepTime = rand() % 3 + 1;
	printf("Philosopher %d will think for %d seconds\n", philosopherNumber, sleepTime);
	sleep(sleepTime);
}
void pickUp(int philosopherNumber) {
	int right = (philosopherNumber + 1) % NUMBER_OF_PHILOSOPHERS;
	int left = (philosopherNumber + NUMBER_OF_PHILOSOPHERS) % NUMBER_OF_PHILOSOPHERS;
	if (philosopherNumber & 1) {
		printf("Philosopher %d is waiting to pick up chopstick %d\n", philosopherNumber, right);
		pthread_mutex_lock(&chopsticks[right]);
		printf("Philosopher %d picked up chopstick %d\n", philosopherNumber, right);
		printf("Philosopher %d is waiting to pick up chopstick %d\n", philosopherNumber, left);
		pthread_mutex_lock(&chopsticks[left]);
		printf("Philosopher %d picked up chopstick %d\n", philosopherNumber, left);
	}
	else {
		printf("Philosopher %d is waiting to pick up chopstick %d\n", philosopherNumber, left);
		pthread_mutex_lock(&chopsticks[left]);
		printf("Philosopher %d picked up chopstick %d\n", philosopherNumber, left);
		printf("Philosopher %d is waiting to pick up chopstick %d\n", philosopherNumber, right);
		pthread_mutex_lock(&chopsticks[right]);
		printf("Philosopher %d picked up chopstick %d\n", philosopherNumber, right);
	}
}
void eat(int philosopherNumber) {
	int eatTime = rand() % 3 + 1;
	printf("Philosopher %d will eat for %d seconds\n", philosopherNumber, eatTime);
	sleep(eatTime);
}
void putDown(int philosopherNumber) {
	printf("Philosopher %d will will put down her chopsticks\n", philosopherNumber);
	pthread_mutex_unlock(&chopsticks[(philosopherNumber + 1) % NUMBER_OF_PHILOSOPHERS]);
	pthread_mutex_unlock(&chopsticks[(philosopherNumber + NUMBER_OF_PHILOSOPHERS) %
NUMBER_OF_PHILOSOPHERS]);
}
/*
[vishnu@mannava ~]$ cc phil-lr.c -lpthread
[vishnu@mannava ~]$ ./a.out
Philosopher 1 will think for 2 seconds
Philosopher 0 will think for 3 seconds
Philosopher 2 will think for 2 seconds
Philosopher 4 will think for 2 seconds
Philosopher 3 will think for 2 seconds
Philosopher 1 is waiting to pick up chopstick 2
Philosopher 1 picked up chopstick 2
Philosopher 1 is waiting to pick up chopstick 1
Philosopher 1 picked up chopstick 1
Philosopher 1 will eat for 3 seconds
Philosopher 2 is waiting to pick up chopstick 2
Philosopher 4 is waiting to pick up chopstick 4
Philosopher 4 picked up chopstick 4
Philosopher 4 is waiting to pick up chopstick 0
Philosopher 4 picked up chopstick 0
Philosopher 4 will eat for 3 seconds
Philosopher 3 is waiting to pick up chopstick 4
Philosopher 0 is waiting to pick up chopstick 0
Philosopher 1 will will put down her chopsticks
Philosopher 1 will think for 2 seconds
Philosopher 2 picked up chopstick 2
Philosopher 2 is waiting to pick up chopstick 3
Philosopher 2 picked up chopstick 3
Philosopher 2 will eat for 3 seconds
Philosopher 4 will will put down her chopsticks
Philosopher 4 will think for 1 seconds
Philosopher 0 picked up chopstick 0
Philosopher 0 is waiting to pick up chopstick 1
Philosopher 0 picked up chopstick 1
Philosopher 3 picked up chopstick 4
Philosopher 3 is waiting to pick up chopstick 3
Philosopher 0 will eat for 1 seconds
^C
```

/* Program to demonstrate Semaphores - Reader Writer Problem */

```
#include<stdio.h>
#include<pthread.h>
#include<semaphore.h>
sem_t mutex,writeblock;
int data = 0,rcount = 0;
void *reader(void *arg)
{
  int f;
  f = ((int)arg);
  sem_wait(&mutex);
  rcount = rcount + 1;
  if(rcount==1)
   sem_wait(&writeblock);
  sem_post(&mutex);
  printf("Data read by the reader%d is
%d\n",f,data);
  sleep(1);
  sem_wait(&mutex);
  rcount = rcount - 1;
  if(rcount==0)
   sem_post(&writeblock);
  sem_post(&mutex);
}/*
[vishnu@mannava ~]$ ./a.out
Data read by the reader0 is 0
Data read by the reader1 is 0
Data read by the reader2 is 0
Data writen by the writer1 is 1
Data writen by the writer0 is 2
Data writen by the writer2 is 3 */
```

```
void *writer(void *arg)
{
  int f;
  f = ((int) arg);
  sem_wait(&writeblock);
  data++;
  printf("Data writen by the writer%d is
%d\n",f,data);
  sleep(1);
  sem_post(&writeblock);
}
int main()
{
  int i,b;
  pthread_t rtid[5],wtid[5];
  sem_init(&mutex,0,1);
  sem_init(&writeblock,0,1);
  for(i=0;i<=2;i++)
  {
    pthread_create(&wtid[i],NULL,writer,(void *)i);
    pthread_create(&rtid[i],NULL,reader,(void *)i);
  }
  for(i=0;i<=2;i++)
  {
    pthread_join(wtid[i],NULL);
    pthread_join(rtid[i],NULL);
  }
  return 0;
}
```