Deadlocks (CS-351)

Agenda

- What is a deadlock?
- Resource-Allocation Graph
- Deadlock prevention.
- Deadlock avoidance.

What is a Deadlock?

- A set of processes $\{p_1 \dots p_n\}$ is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.
- Example: A system with one printer and one DVD drive.
  - Process P_i is holding the DVD drive.
  - Process P_j is holding the printer.
  - If P_i requests the printer and P_j requests the DVD drive, a dead lock occurs.
- Example: a system has 3 CD RW drives D_1, D_2, and D_3 and processes P_1, P_2, and P_3:
  - Each process P_i is holding drive D_i.
  - If each process requests access to another drive, a deadlock occurs.

What is a Deadlock?

- Example: "When two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone" – law passed by the Kansas legislature in the early 20th century.
- Example: Let S and Q be two semaphores set to the value of 1, and P_0 and P_1 processes sharing these semaphores.

P_0	P_1
wait (Q);	wait (S);
wait (S);	wait (Q);
.	.
.	.
.	.
signal(S)	signal(Q)
signal(Q)	signal(S)

- The processes may become deadlocked after the second line.

What is a Deadlock?

- Example: Pthreads library uses mutex locks (which behave like binary semaphores) to provide mutual exclusion:
  - pthread_mutex_t myMutex: declares a variable called myMutex of type mutex.
  - pthread_mutex_init(&myMutex,NULL): initializes myMutex and sets its state to “unlocked”.
  - pthread_mutex_lock(&myMutex): locks the mutex myMutex.
  - pthread_mutex_unlock(&myMutex): unlocks the mutex myMutex.
  - If myMutex is already locked, then subsequent calls to pthread_mutex_lock will cause the calling thread to block.
  - If myMutex locking fails, then calling pthread_mutex_lock will return an error.

What is a Deadlock? - One Mutex example

```
#include <pthread. h>
#include <stdio. h>
/* This data is shared by the thread(s) */
int count = 0;                    //The counter
variable.
pthread_t t1, t2;       //The thread variables.
pthread_mutex_t myMutex; //The mutex
/* the thread */
void *runner(void *param);
int sum = 0;
int main(int argc, char *argv[] )
{
  pthread_attr_t attr; /* Set of thread
attributes */
  /* Get the default attributes */
  pthread_attr_init(&attr);
 /* Initialize the mutex */
 pthread_mutex_init(&myMutex, NULL);
 /* create the thread */
  pthread_create(&t1, &attr, runner, NULL);
  pthread_create(&t2, &attr, runner, NULL);
 /* wait for the thread to exit */
  pthread_join(t1, NULL);
  pthread_join(t2,NULL);
} //End of main()

/* The thread will begin control in this
function */
void *runner(void *param)
{
  /* Lock the mutex to allow only one
thread */
  pthread_mutex_lock(&myMutex);
  for (int i = 0; i < 10; ++i) count += 1
  /* Unlock the mutex */
  pthread_mutex_unlock(&myMutex);

   pthread_exit(0);
}
```

Compare to Pthread multithreading sum example

What is a Deadlock? - Two mutex example

```c
/* thread_one runs in this function */
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;
pthread_t t1,t2;
pthread_attr_t attr;

int main()
{
 /* Initialize the mutex locks */
  pthread_mutex_init(&first_mutex, NULL);
  pthread_mutex_init(&second_mutex, NULL);

  /* Get the default attributes */
  pthread_attr_init(&attr);

   /* create the thread */
  pthread_create(&t1, &attr, do_work_one, NULL);

  pthread_create(&t2, &attr, do_work_two, NULL);
  pthread_join(t1,NULL);
  pthread_join(t2,NULL)
}
```

Deadlock is possible if thread_one acquires first_mutex while thread two acquires second_mutex.

```c
/* t1 runs in this function */
void *do_work_one(void* param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);

   /** Do some work **/

   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);
   pthread_exit(0);
}

/* t2 runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);

    /** Do some work */

    pthread_mutex_unlock(&first_mutex);
    pthread_mutex_unlock(&second_mutex);

    pthread_exit(0);
}
```

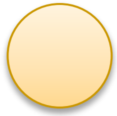
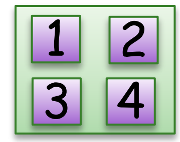
Why do Deadlocks Occur?

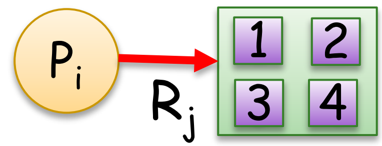
- A deadlock can arise only if ALL of the following conditions are met:
  - Mutual exclusion: the resources involved are non-sharable i.e. if process P_1 requests resource R held by process P_2, then P_1 must wait for P_2 to release the resource.
  - Hold and wait: a process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.
  - No preemption: resources cannot be preempted; a resource can be released only voluntarily by the process holding it, after that process has completed its task.
  - Circular wait: a set $\{P_0, P_1, ..., P_n\}$ of waiting processes must exist such that P_0 is waiting for a resource held by P_1, P_1 is waiting for a resource held by $P_2, ..., P_{n-1}$ is waiting for a resource held by P_n, and P_n is waiting for a resource held by P_0.

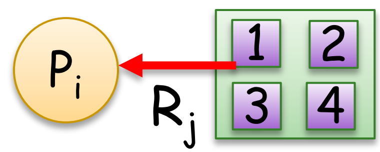
Resource-Allocation Graphs

- Used to precisely describe deadlocks.
- A set of vertices V and a set of edges E.
- V is partitioned into two types of vertices:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- Request edge: directed edge $P_i \rightarrow R_j$.
- Assignment edge: directed edge $R_j \rightarrow P_i$.

Resource-Allocation Graph

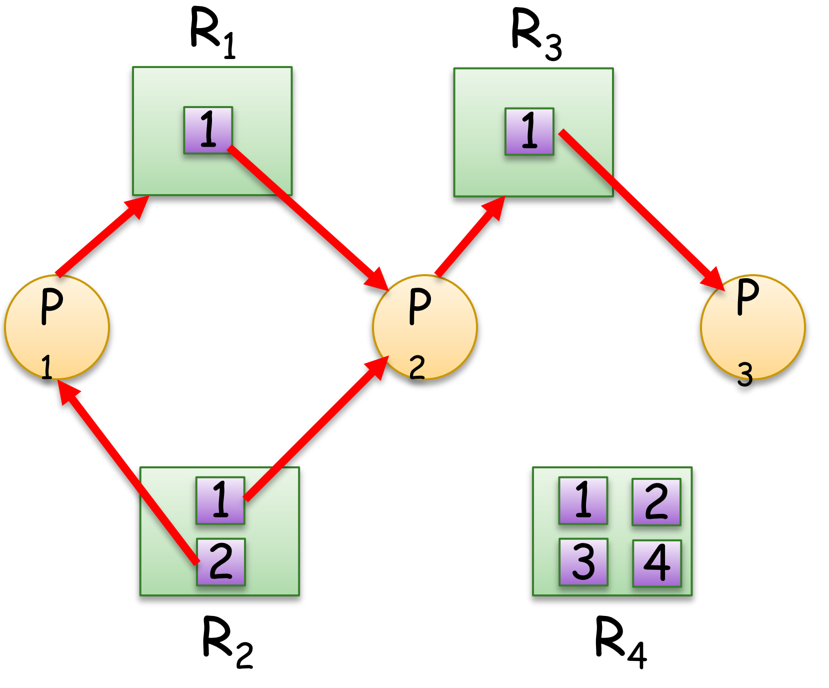
- Process: 
- Resource type with 4 instances: 

- P_i requests an instance of R_j: 

- P_i is holding an instance of R_j: 

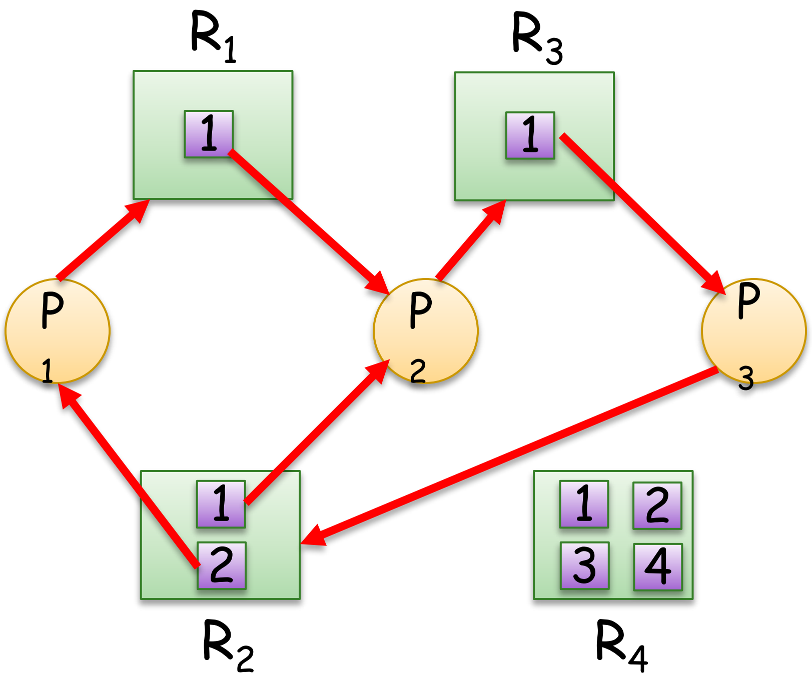
Resource-Allocation Graph - Example

- Example:
  - Process P_1 requests an instance of resource R_1.
  - Process P_2 requests an instance of resource R_3.
  - Instances of resource R_2 are assigned to processes P_1 and P_2.
  - Instance of resource R_3 is assigned to P_3.



Resource-Allocation Graph - Cycle

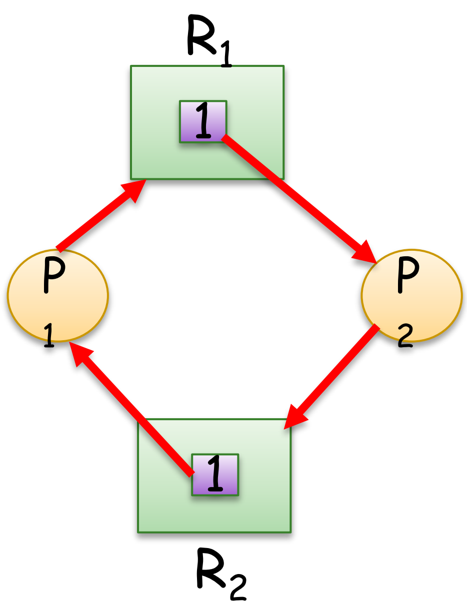
- Example:
  - P_3 now requests an instance of resource R_2.



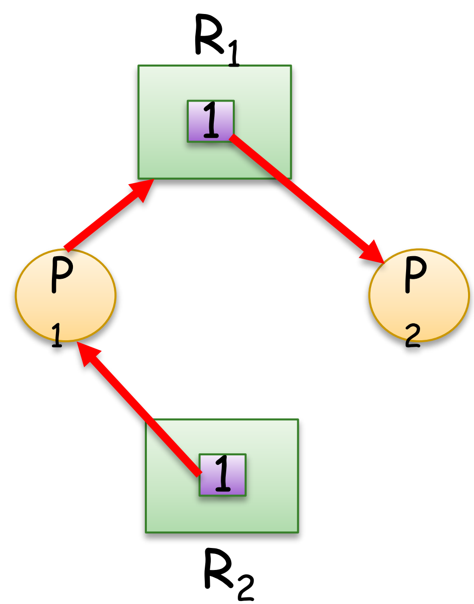
- The graph contains 2 cycles:
  - $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
  - $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$

Resource-Allocation Graph – Cycle vs Deadlock

- If graph contains no cycles then no deadlock is possible.
- If graph contains a cycle:
  - If only one instance per resource type, then deadlock.
  - If several instances per resource type, then possibility of deadlock.

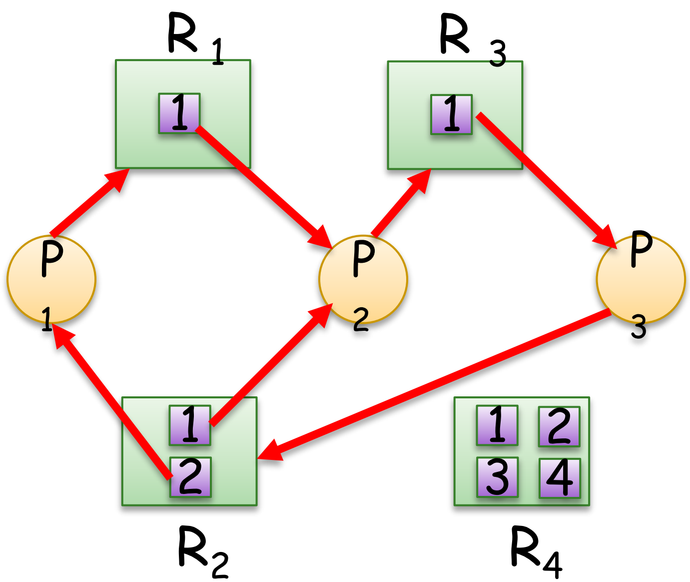


Cycle: $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_1$ and all resources in the cycle have a single instance = definite DEADLOCK!



No Cycles = definitely NO DEADLOCK!

Resource-Allocation Graph



Cycles:

$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$

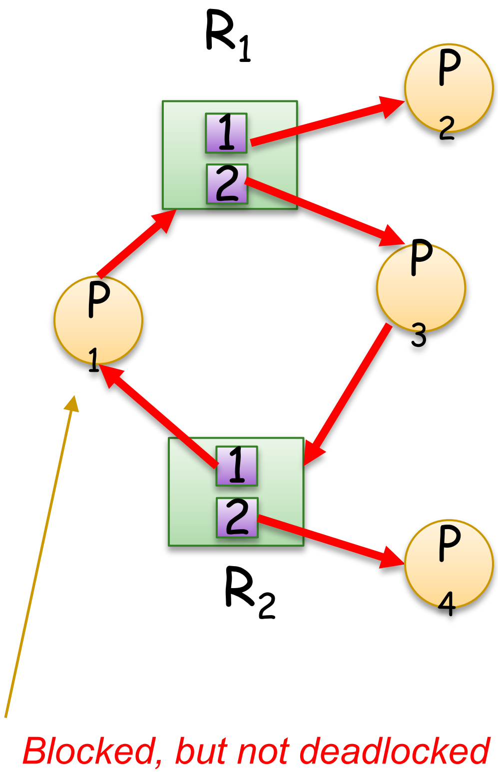
$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$ and

Resource R_2 is in the cycle and has multiple instances = possible deadlock!

Actuality: P_1, P_2, and P_3 are deadlocked:

- P_2 is waiting for the R_3, held by process P_3.
- P_3 is waiting for either P_1 or P_2 to release R_2.
- In addition, P_1 is waiting for P_2 to release R_1.

Resource-Allocation Graph – Another example



Cycle: $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$
and R_1 and R_2 are in the cycle and have multiple instances = possible deadlock!

Actuality: No deadlock.

- Process P_4 may release its instance of R_2.
- R_2 can then be allocated to P_3, breaking the cycle.

Student Participation: Number of processes and resources needed for a deadlock

- Deadlock is possible when
  - A) the number of processes is greater than 1, regardless of the number of resources.
  - B) the number of resources is greater than 1, regardless of the number of processes.
  - C) the number of processes and the number of resources are both greater than 1.

Methods for Handling Deadlocks

- Intuition:
  - Ensure that the system will never enter a deadlocked state.
  - OR allow the system to enter a deadlocked, detect the deadlock, and recover.
  - OR pretend that deadlocks cannot happen (the most popular approach i.e. used in Windows and Unix):
  - Another words, it's up to application developers to write deadlock-free applications.

Methods for Handling Deadlocks

- Specific Approaches:
  - Deadlock Prevention: making ensure that **at least one** of the four conditions necessary for the deadlock does not hold.
  - Deadlock avoidance:
    - Require process provide information about resources it will need in the **future**.
    - Use this information to delegate resources in a way that will avoid deadlocks.
  - Deadlock detection and recovery: allow deadlocks to happen, but have means of **recovering** from them.

Deadlock Prevention

- Must ensure that at least one of the following four conditions necessary for a deadlock does not hold.
- 1. Mutual exclusion: only necessary for non-sharable resources:
  - Example: multiple processes cannot safely share a printer, but can a read-only file.
  - **Problem:** some resource (e.g. mutexes) are inherently non-sharable.

Deadlock Prevention

- Must ensure that at least one of the following four conditions necessary for a deadlock does not hold.
- 2. Hold and wait: must guarantee that whenever a process requests a resource, it holds no other resources:
  - Require process to request and be allocated all its resources before it begins execution (e.g. do not allow system calls from the process until all resources are allocated).
  - **Problem:** low resource utilization, and may cause starvation.

Deadlock Prevention

- Must ensure that at least one of the following four conditions necessary for a deadlock does not hold.
- 3. No preemption: if the process requests a resource that cannot be allocated immediately:
  - 1. Preempt all resources held by the process.
  - 2. Add the released resources to the list of resources requested by the process.
  - 3. Restart the process when all resources are available.
  - **Problem:** only applicable to resources whose state can be easily saved and restored:
    - Example: CPU state, registers, etc. can be easily saved and restored. mutex/sempahore state cannot.

Deadlock Prevention

- Must ensure that at least one of the following four conditions necessary for a deadlock does not hold.
- 4. Circular wait: impose a total ordering of all resource types, and require each process to request resources in an increasing order of enumeration.
  - Can be proved correct by contradiction.
  - Example: if two processes wants to use a tape drive and a printer, all the processes must first request a tape drive and then a printer (and not vice-versa).

Student Participation: Eliminating the conditions for deadlock.

- 1) To guarantee that deadlock is impossible, the hold-and-wait condition must be eliminated for
  - at least 2 processes.
  - all processes.
- 2) To guarantee that deadlock is impossible, the circular-wait condition must be eliminated for
  - at least 1 process.
  - all processes.
- 3) To guarantee that deadlock is impossible
  - eliminating either of the two conditions is sufficient.
  - both conditions must be eliminated.

Deadlock Avoidance

- Next, we study an alternative technique called deadlock avoidance.