# Pthread: Mutex

- Provides synchronization to a shared resource
- A typical sequence in the use of a mutex is as follows:
  - Create and initialize a mutex variable
  - Several threads attempt to lock the mutex
  - Only one succeeds and that thread owns the mutex
  - The owner thread performs some set of actions
  - The owner unlocks the mutex
  - Another thread acquires the mutex
  - … …
  - Finally the mutex is destroyed

# POSIX Thread Library
# # include <pthread.h>

- int **pthread_mutex_init** (pthread_mutex_t  *mutex,
  const pthread_mutexattr_t *attr)
  - initializes a *mutex* with the specified attributes
  - $1^{st}$ argument: the address of the newly created *mutex*
  - $2^{nd}$ argument: the address to the variable containing the mutex attributes object
    - To use the default attributes, use NULL as the second argument

# POSIX Thread Library
# # include <pthread.h>

- int **pthread_mutex_destroy** (pthread_mutex_t  *mutex)
  - Free a mutex object that is no longer needed.

# POSIX Thread Library
# # include <pthread.h>

- int **pthread_mutex_lock** (pthread_mutex_t *mutex)
  - acquires ownership of the *mutex* specified
  - If the specified *mutex* is currently locked, the calling thread is blocked until the *mutex* is available
  - 1st argument: the address of the *mutex* to be locked

- int **pthread_mutex_unlock** (pthread_mutex_t *mutex)
  - unlocks the *mutex* specified
  - 1st argument: the address of the *mutex* to be unlocked

- Important: make sure all threads, that need access to the shared data, use mutex. Otherwise, the shared data could still be corrupted.

# Pthread: Condition variable

- A mutex makes other threads to wait while the thread holding the mutex executes code in a critical section.
- A condition variable is typically used by a thread (that has already acquired a mutex) to make itself wait (and temporarily release the mutex) until signaled by another thread.

# POSIX Thread Library
# # include <pthread.h>

- int **pthread_cond_init** (pthread_cond_t *cond,
  const pthread_condattr_t *attr)
  - initializes a condition variable object with the specified attributes
  - 1st argument: the address of the newly created *condition variable*
  - 2nd argument: the address to the variable containing the condition variable attributes object
    - To use the default attributes, use NULL as the second argument

# POSIX Thread Library
# # include <pthread.h>

- int **pthread_cond_wait** (pthread_cond_t   *cond,
  pthread_mutex_t *mutex)
  - blocks the calling thread on the condition variable *cond*
    - 1st argument: the address of the condition variable to wait on
    - 2nd argument: the address of the mutex associated with the condition variable
  - when **pthread_cond_wait()** is called, the calling thread must have the associated *mutex* locked
  - the **pthread_cond_wait()** function unlocks the associated *mutex* and blocks on the condition variable (waiting for another thread to signal the condition)

# POSIX Thread Library
# # include <pthread.h>

- int **pthread_cond_signal** (pthread_cond_t *cond)
  - wakes up one thread that is waiting on the condition variable
  - 1st argument: the address of the condition variable
  - the thread that is just waked up automatically requests for the mutex that it unlocked when calling *pthread_cond_wait();* it can resume its execution only after it has re-acquired the *mutex*

# Example: Implementing RW-Monitor

**Monitor** Reader-Writer

begin

    readercount: integer

    busy: boolean

    OKtoread, OKtowrite: condition

```
/*Shared variables*/

int readercount;
int busy;
pthread_mutex_t mutex;
pthread_cond_t OKtoread, OKtowrite;
int QL_OKtoread;
```

# Example: Implementing RW-Monitor

```
/*Initialization*/

void init()
{
  pthread_mutex_init(&mutex, NULL);
  pthread_cond_init(&OKtoread, NULL);
  pthread_cond_init(&OKtowrite, NULL);
  readercount=0;
  busy=0;
  QL_OKtoread=0;
}
```

```
begin /* initialization of local
data */
  readercount := 0;
  busy := false;
end
```

# Example: Implementing RW-Monitor

```
/*Procedure startread*/
void startread()
{
 pthread_mutex_lock(&mutex);
   QL_OKtoread++;
   while(busy)
             pthread_cond_wait(
             &OKtoread, &mutex);
   QL_OKtoread--;
   readercount++;
   pthread_cond_signal(&OKtoread);
 pthread_mutex_unlock(&mutex);
}
```

```
procedure startread
begin
  If busy then OKtoread.wait;
  readercount := readercount + 1;
  OKtoread.signal
end startread
```

# Example: Implementing RW-Monitor

```
/*Procedure endread*/
void endread()
{
 pthread_mutex_lock(&mutex);
   readercount--;
   if(readercount==0)
    pthread_cond_signal(&OKtowrite);
 pthread_mutex_unlock(&mutex);
}
```

```
procedure endread;
begin
  readercount := readercount-1;
  If readercount = 0 then
    OKtowrite.signal;
end endread
```

# Example: Implementing RW-Monitor

```
/*Procedure startwrite*/
void startwrite()
{
 pthread_mutex_lock(&mutex);
   while(busy||readercount!=0)
    pthread_cond_wait(&OKtowrite,
    &mutex);
   busy=1;
 pthread_mutex_unlock(&mutex);
}
```

```
procedure startwrite;
begin
   If busy or readercount <>0 then
     OKtowrite.wait;
   busy := true;
end startwrite
```

# Example: Implementing RW-Monitor

```
/*Procedure endwrite*/
void endwrite()
{
 pthread_mutex_lock(&mutex);
   busy=0;
   if(QL_OKtoread>0)
    pthread_cond_signal(&OKtoread);
   else
    pthread_cond_signal(&OKtowrite);
 pthread_mutex_unlock(&mutex);
}
```

```
procedure endwrite;
begin
  busy := false
  if OKtoread.queue then
    OKtoread.signal
  else OKtowrite.signal
end endwrite
```

# Example: Using RW-Monitor

```
/*reader*/
void *reader(void *x)
{
startread();
//code to read
endread();
}
```

```
/*writer*/
void *writer(void *x)
{
startwrite();
//code to write
endwrite();
}
```

# Deadlock (I)

September 25, 2017

# The Deadlock Problem

- A set of blocked processes each holds an instance of resource and waits to acquire an instance of resource held by another process
- Example
  - System has 2 disk drives
  - $P_1$ and $P_2$ each hold one disk drive and each needs another one
- Example
  - semaphores $A$ and $B$, initialized to 1

| $P_0$ | $P_1$ |
| --- | --- |
| wait (A); | wait(B) |
| wait (B); | wait(A) |

# Deadlock Characterization

Deadlock arises → 4 conditions hold simultaneously (necessary conditions of deadlock)

- **Mutual exclusion:** an instance of resource can only be used by a process at a time
- **Hold and wait:** a process holding at least one instance of resource is waiting to acquire additional resources held by other processes
- **No preemption:** resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait:** there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that $P_0$ is waiting for a resource that is held by $P_1$, $P_1$ is waiting for a resource that is held by $P_2$, …, $P_{n-1}$ is waiting for a resource that is held by $P_n$, and $P_n$ is waiting for a resource that is held by $P_0$.

# System Model

- Resource types $R_1, R_2, \ldots, R_m$

  *CPU cycles, memory space, I/O devices*
- Each resource type $R_i$ has $W_i$ instances.
- Each process utilizes a resource in the following order:

  **request → use → release**

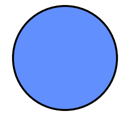
# Resource-Allocation Graph

A set of vertices *V* and a set of edges *E*.

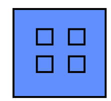
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_i \rightarrow R_j$
- assignment edge – directed edge $R_j \rightarrow P_i$

# Resource-Allocation Graph (Cont.)

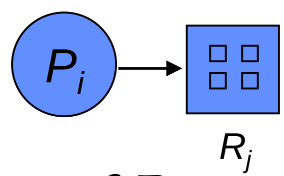
- Process



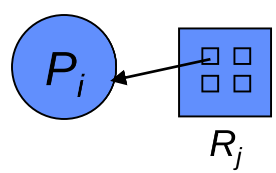
- Resource Type with 4 instances



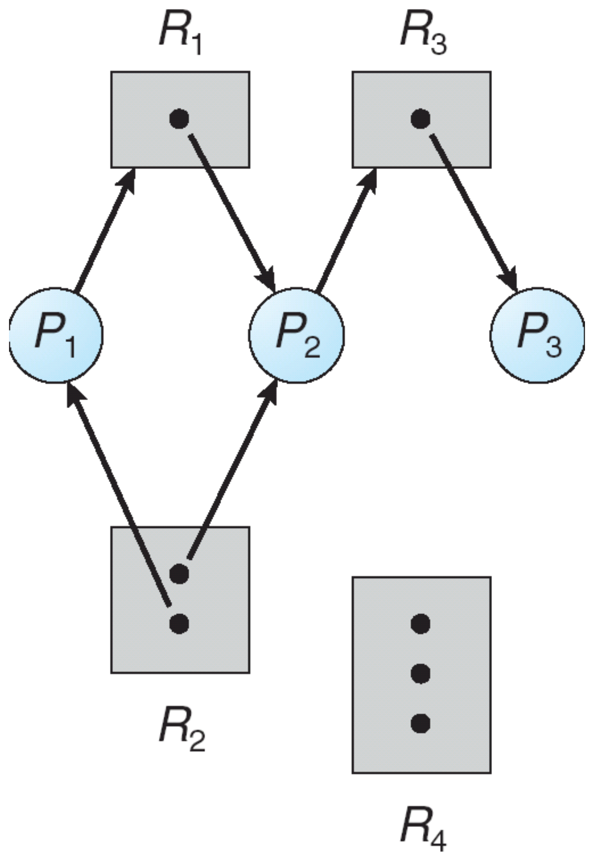
- $P_i$ requests an instance of $R_j$



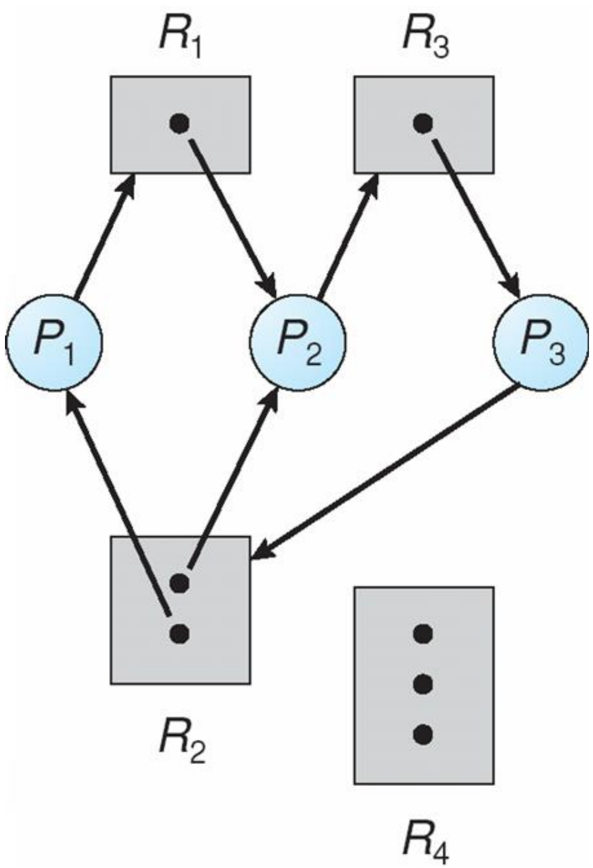

- $P_i$ is holding an instance of $R_j$



# Example of a Resource Allocation Graph

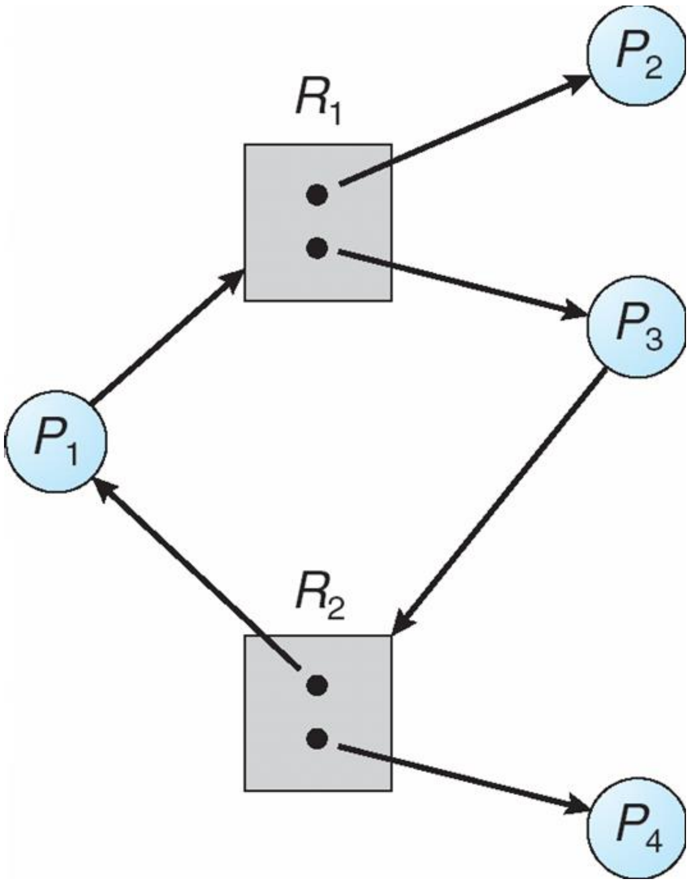


# Resource Allocation Graph With A Deadlock




There is one or more cycles in the graph

# Graph With A Cycle But No Deadlock



# Basic Facts

- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

# Methods for Handling Deadlocks

- Ensure that the system will *never* enter a deadlock state
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

# Deadlock Prevention

Restrain the ways that request can be made such that at least one of the deadlock necessary conditions fails

- **Mutual Exclusion → Make each instance of resource simultaneously sharing** – often impossible

- **Hold and Wait → Request all at once; or, Give up all before request** – must guarantee that whenever a process requests a resource, it does not hold any other resources (i.e., it should release all resources that it currently holds)
  - Low resource utilization; starvation possible

# Deadlock Prevention

- **No Preemption → Allow resource instance to be re-assignable**
  - Often impossible
- **Circular Wait → Prevent Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

# Deadlock due to circular wait

P1 & P2 share semaphores A & B & C

| P1: | P2: |
|---|---|
| wait(A) | wait(B) |
| ...segment 1… | ...segment 4… |
| wait(B) | wait(C) |
| ...segment 2… | ...segment 5… |
| wait(C) | wait(A) |
| …segment 3 … | …segment 6 … |
| signal(C) | signal(A) |
| signal(B) | signal(C) |
| signal(A) | signal(B) |

# Prevent circular wait

P1 & P2 share semaphores A & B & C; impose order: A<B<C

```
P1:
wait(A)
...segment 1…
wait(B)
...segment 2…
wait(C)
…segment 3 …
signal(C)
signal(B)
signal(A)
```

```
P2:
wait(A)
wait(B)
...segment 4…
wait(C)
...segment 5…
…segment 6 …
signal(A)
signal(C)
signal(B)
```