## Review

- Shared File in multiuser system
  - Save i-node index
  - Symbolic link
- Log-Structured File System (extension of i-node + contiguous)
- Disk Space Management
- Free Block Management
  - Linked List
  - Bit Map
- Disk Quota
- File System Backup
  - Physical Backup
  - Logical Backup

## Preview

- What is Deadlock?
- Resource Allocation Graph
  - Deadlock example with Resources Allocation Graph
- Resource Types for a Process
- Sequence for Resource Use
- Implementation of Resource request, use and release
- Deadlock Condition
- Four Strategies for Dealing Deadlock

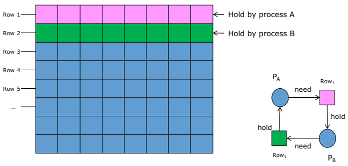
## What is Deadlock

- In a multiprogramming environment, several processes (or threads) may compete for a finite number of resources (racing).
- A process requests resources; if the resources are not available at that time, the process enters a block state.
- Sometimes, a blocked process can never again change state, because the resources it has requested are held by other blocked processes. This situation is called a deadlock
- Deadlocks between processes can be occurred since limitation of resources which must be shared.
- We cannot avoid deadlocks without proper managements by OS.

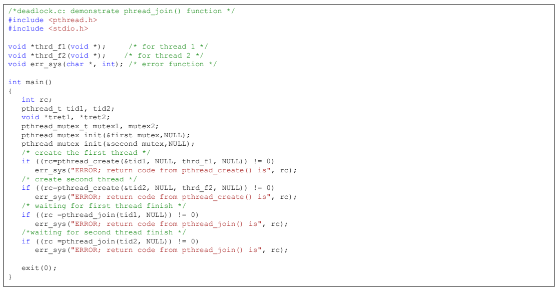
## Deadlocks

- Deadlock is a common problem in multiprocessing systems, parallel computing and distributed systems, where software (semaphore or mutex) and hardware (hardware instructions or hardware component to block) locks are used to handle shared resources and implement process synchronization.
- For example, in a transactional database, a deadlock happens when two processes each within its own transaction updates two rows of information but in the opposite order.
  - Ex) Process$_A$ updates row$_1$ then row$_2$ in the exact timeframe that process$_B$ updates row$_2$ then row$_1$.
  - Process$_A$ can't finish updating row$_2$ until Process$_B$ is finished, but Process$_B$ cannot finish updating row 1 until Process$_A$ is finished.

## Deadlocks



Database transaction



```
/*deadlock.c: demonstrate phread_join() function */
#include <pthread.h>
#include <stdio.h>

void *thrd_f1(void *);     /* for thread 1 */
void *thrd_f2(void *);    /* for thread 2 */
void err_sys(char *, int); /* error function */

int main()
{
   int rc;
   pthread_t tid1, tid2;
   void *tret1, *tret2;
   pthread_mutex_t mutex1, mutex2;
   pthread mutex init(&first mutex,NULL);
   pthread mutex init(&second mutex,NULL);
   /* create the first thread */
   if ((rc=pthread_create(&tid1, NULL, thrd_f1, NULL)) != 0)
      err_sys("ERROR; return code from pthread_create() is", rc);
   /* create second thread */
   if ((rc=pthread_create(&tid2, NULL, thrd_f2, NULL)) != 0)
      err_sys("ERROR; return code from pthread_create() is", rc);
   /* waiting for first thread finish */
   if ((rc =pthread_join(tid1, NULL)) != 0)
      err_sys("ERROR; return code from pthread_join() is", rc);
   /*waiting for second thread finish */
   if ((rc =pthread_join(tid2, NULL)) != 0)
      err_sys("ERROR; return code from pthread_join() is", rc);

   exit(0);
}
```

# Deadlocks

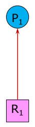
```
/* for thread 1 */
void *thrd_f1(void *arg)
{
   pthread mutex lock(&mutex1);
   pthread mutex lock(&mutex2);
   /* Do some work */
   pthread mutex unlock(&mutex2);
   pthread mutex unlock(&mutex1);
   pthread exit(0);
}
/* for thread 2 */
void * thrd_f2(void *arg)
{
   pthread mutex lock(&mutex2);
   pthread mutex lock(&mutex1);
   /* Do some work */
   pthread mutex unlock(&mutex1);
   pthread mutex unlock(&mutex2);
   pthread exit(0);}

void err_sys(char *str, int msg)
{
   printf ("%s %d\n",str, msg);
   exit (1);
}
```

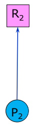
# Resource Allocation Graph

- This graph consists of a set of vertices V and a set of edges E.
- The set of vertices V is partitioned into two types
  - P = {$P_1$, $P_2$, …, $P_n$}= set of processes and
  - R = {$R_1$, $R_2$, …, $R_m$} = set of resources
- Edge from a process to a resource
  - P → R denote process P request resource R and currently waiting
- Edge from a resource to a process
  - R → P denote resource R is currently held by process P

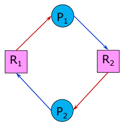
# Resource-Allocation Graph



A process hold resource



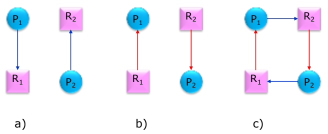
A process request resource



# Deadlocks

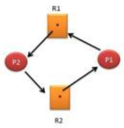
- There are two processes $P_1$, $P_2$ working on their job and , and two resource $R_1$ and $R_2$. Both $P_1$ and $P_2$ need $R_1$ and $R_2$ to finish their job.
  - $T_1$: $P_1$ request $R_1$ and granted
  - $T_1$: $P_2$ request $R_2$ and granted
  - $T_2$: $P_1$ request $R_2$ without releasing $R_1$. But, since $R_2$ is hold by $P_2$, $P_1$ become waiting (block) state waiting for $R_2$ which is granted to $P_2$
  - $T_3$: $P_2$ request $R_1$ but it is not released by $P_1$ yet, $P_2$ go to blocked state. $P_1$ and $P_2$ will be blocked forever.

# Deadlocks

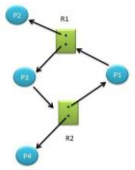


Resource allocation graph

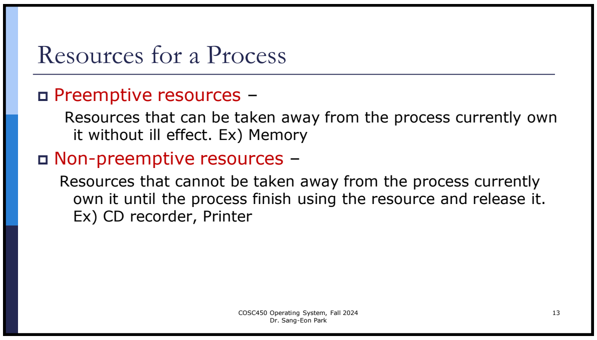
# Deadlocks



Deadlock

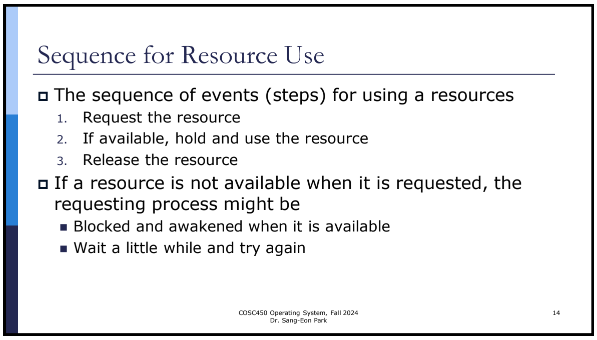


No Deadlock



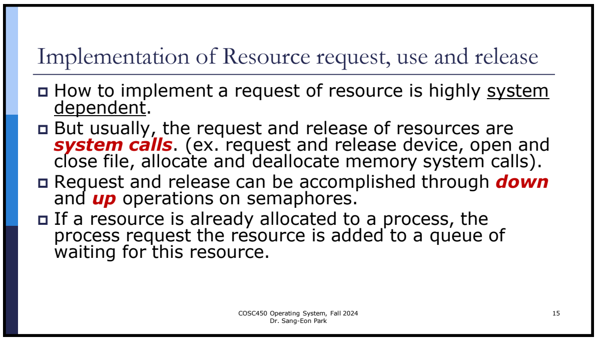
## Resources for a Process

- Preemptive resources –
  Resources that can be taken away from the process currently own it without ill effect. Ex) Memory
- Non-preemptive resources –
  Resources that cannot be taken away from the process currently own it until the process finish using the resource and release it. Ex) CD recorder, Printer



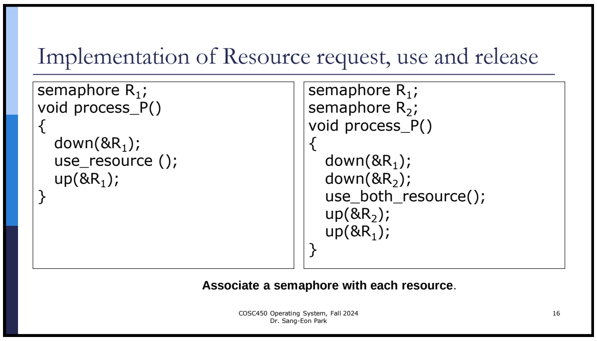
## Sequence for Resource Use

- The sequence of events (steps) for using a resources
  1. Request the resource
  2. If available, hold and use the resource
  3. Release the resource
- If a resource is not available when it is requested, the requesting process might be
  - Blocked and awakened when it is available
  - Wait a little while and try again



## Implementation of Resource request, use and release

- How to implement a request of resource is highly system dependent.
- But usually, the request and release of resources are ***system calls***. (ex. request and release device, open and close file, allocate and deallocate memory system calls).
- Request and release can be accomplished through ***down*** and ***up*** operations on semaphores.
- If a resource is already allocated to a process, the process request the resource is added to a queue of waiting for this resource.

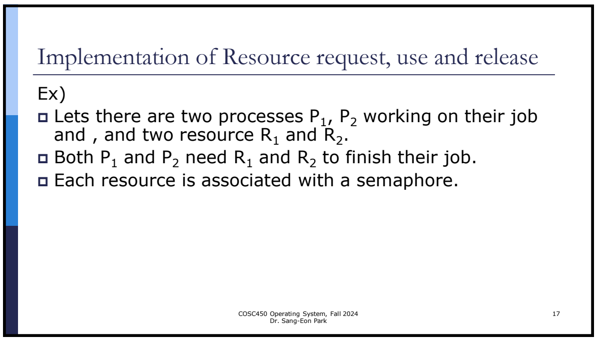


## Implementation of Resource request, use and release

```
semaphore R1;
void process_P()
{
   down(&R1);
   use_resource ();
   up(&R1);
}
```

```
semaphore R1;
semaphore R2;
void process_P()
{
   down(&R1);
   down(&R2);
   use_both_resource();
   up(&R2);
   up(&R1);
}
```

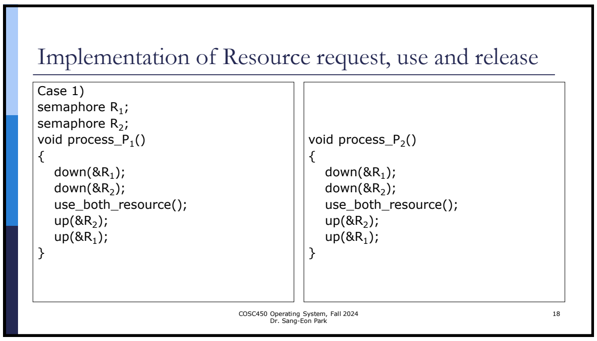
**Associate a semaphore with each resource**.



## Implementation of Resource request, use and release

Ex)

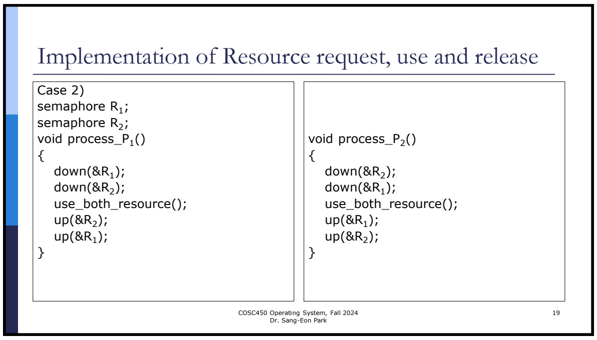
- Lets there are two processes $P_1$, $P_2$ working on their job and , and two resource $R_1$ and $R_2$.
- Both $P_1$ and $P_2$ need $R_1$ and $R_2$ to finish their job.
- Each resource is associated with a semaphore.



## Implementation of Resource request, use and release

```
Case 1)
semaphore R1;
semaphore R2;
void process_P1()
{
   down(&R1);
   down(&R2);
   use_both_resource();
   up(&R2);
   up(&R1);
}
```

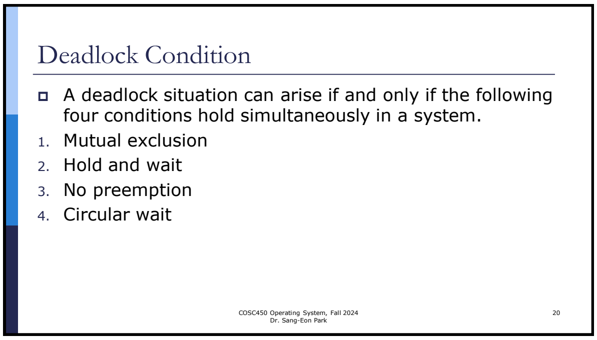
```
void process_P2()
{
   down(&R1);
   down(&R2);
   use_both_resource();
   up(&R2);
   up(&R1);
}
```



## Implementation of Resource request, use and release

```
Case 2)
semaphore R1;
semaphore R2;
void process_P1()
{
    down(&R1);
    down(&R2);
    use_both_resource();
    up(&R2);
    up(&R1);
}
```

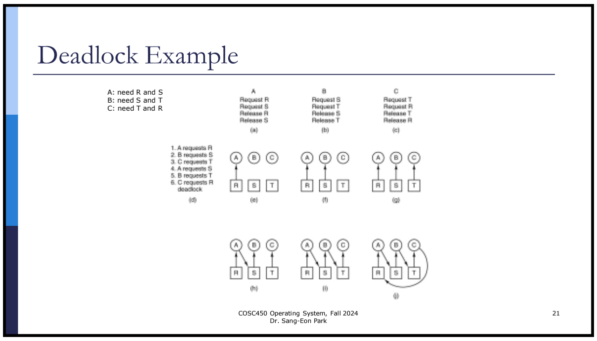
```
void process_P2()
{
    down(&R2);
    down(&R1);
    use_both_resource();
    up(&R1);
    up(&R2);
}
```



## Deadlock Condition

- A deadlock situation can arise if and only if the following four conditions hold simultaneously in a system.

1. Mutual exclusion
2. Hold and wait
3. No preemption
4. Circular wait



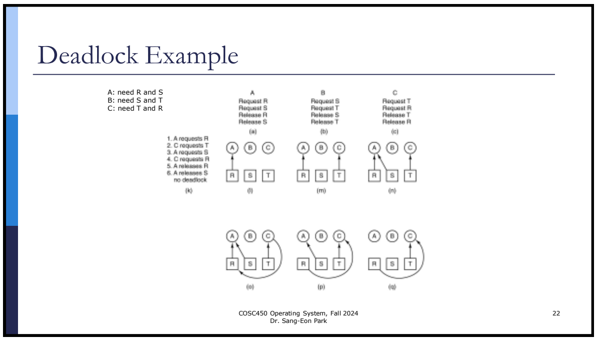
## Deadlock Example

A: need R and S
B: need S and T
C: need T and R

| A | B | C |
|---|---|---|
| Request R<br>Request S<br>Release R<br>Release S | Request S<br>Request T<br>Release S<br>Release T | Request T<br>Request R<br>Release T<br>Release R |
| (a) | (b) | (c) |

1. A requests R
2. B requests S
3. C requests T
4. A requests S
5. B requests T
6. C requests R
deadlock

(d)



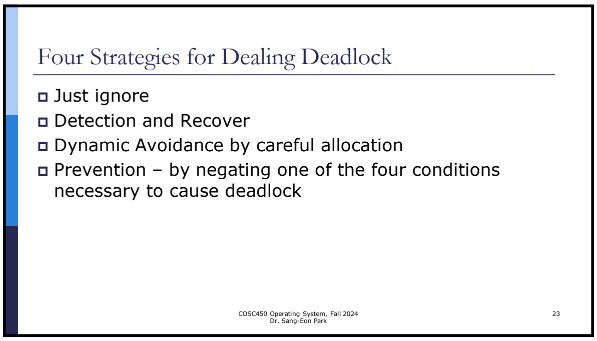
## Deadlock Example

A: need R and S
B: need S and T
C: need T and R

| A | B | C |
|---|---|---|
| Request R<br>Request S<br>Release R<br>Release S | Request S<br>Request T<br>Release S<br>Release T | Request T<br>Request R<br>Release T<br>Release R |
| (a) | (b) | (c) |

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
no deadlock

(k)



## Four Strategies for Dealing Deadlock

- Just ignore
- Detection and Recover
- Dynamic Avoidance by careful allocation
- Prevention – by negating one of the four conditions necessary to cause deadlock