CE 440 Introduction to Operating System

Lecture 9: Deadlock
Fall 2025

Prof. Yigong Hu

BOSTON UNIVERSITY

Slides courtesy of Manuel Egele, Ryan Huang and Baris Kasikci

Deadlock

Synchronization is a live gun

- We can easily shoot ourselves in the foot
- Incorrect use of synchronization can block all processes
- You have likely been intuitively avoiding this situation already

Example: Single-Lane Bridge Crossing

CA 140 to Yosemite National Park

Deadlock

Synchronization is a live gun

- We can easily shoot ourselves in the foot
- Incorrect use of synchronization can block all processes
- You have likely been intuitively avoiding this situation already

If one process tries to access a resource that a second process holds, and vice-versa, they can never make progress

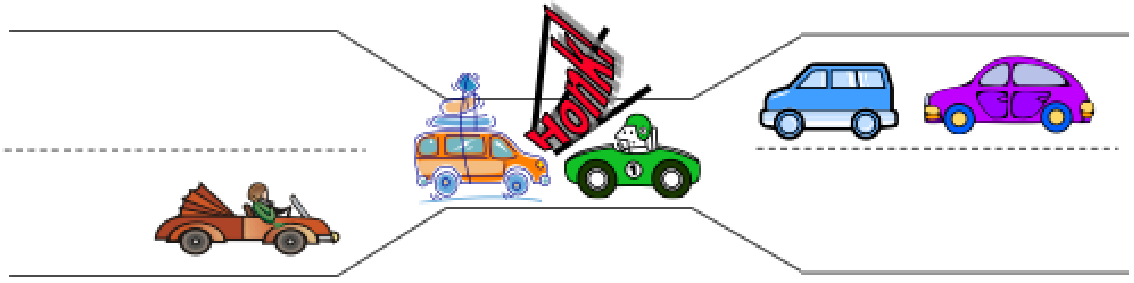
We call this situation deadlock, and we'll look at:

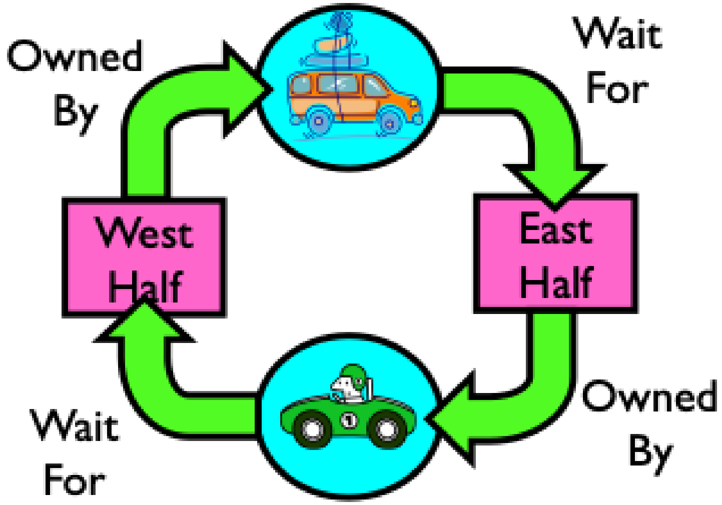
- Definition and conditions necessary for deadlock
- Representation of deadlock conditions
- Approaches to dealing with deadlock

Bridge Crossing Example

Each segment of road can be viewed as a resource

- Car must own the segment under them
- Must acquire segment that they are moving into





Deadlock resolved if one car backs up (preempt resources and rollback)

Starvation: East-going traffic really fast → no one gets to go west

Deadlock Definition

Deadlock is a problem that can arise:

- When processes compete for access to limited resources
- When processes are incorrectly synchronized

Definition:

- Deadlock exists among a set of processes if every process is waiting for an event that can be caused only by another process in the set.

Deadlock Example

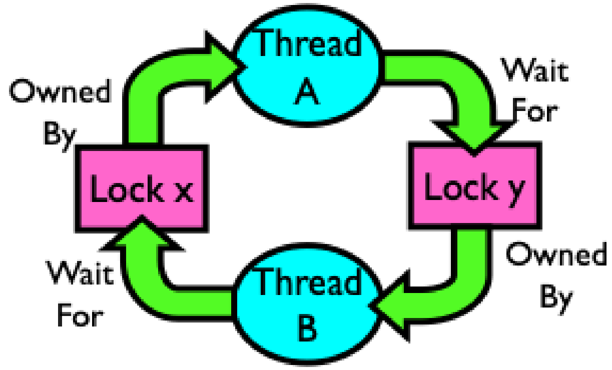
```
mutex_t x, y;
```

Thread A:

```
void p1(void *ignored) {
    lock(x);
    lock(y);
    /* critical section */
    unlock(y);
    unlock(x);
}
```

Thread B:

```
void p2(void *ignored) {
    lock(y);
    lock(x);
    /* critical section */
    unlock(x);
    unlock(y);
}
```



Deadlock Example: “Unlucky” Case

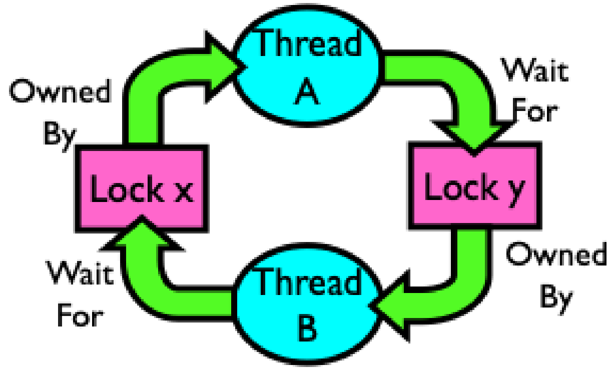
```
mutex_t x, y;
```

Thread A:

```
void p1(void *ignored) {
    lock(x);

    lock(y);  stalled
    <unreachable>
    /* critical section */
    unlock(y);
    unlock(x);
}
```

Thread B:

```
void p2(void *ignored) {

    lock(y);

    lock(x);      stalled
    <unreachable>
    /* critical section */
    unlock(x);
    unlock(y);
}
```



Neither thread will get to run →Deadlock

Deadlock Example: "Lucky" Case

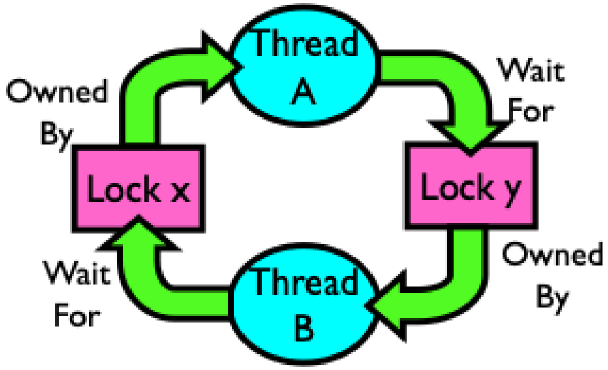
```
mutex_t m1, m2;
```

Thread A:

```
void p1(void *ignored) {
    lock(x);
    lock(y);


    /* critical section */
    unlock(y);
    unlock(x);
}
```

Thread B:

```
void p2(void *ignored) {


    lock(y);
    lock(x);




    /* critical section */
    unlock(x);
    unlock(y);
}
```



Sometimes, schedule won't trigger deadlock!

Deadlock Example

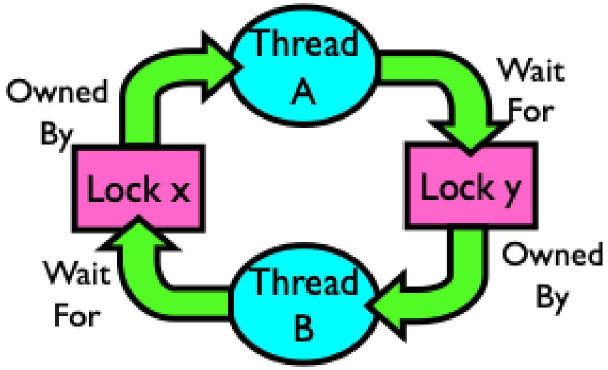
```
mutex_t m1, m2;
```

Thread A:

```
void p1(void *ignored) {
    lock(x);
    lock(y);
    /* critical section */
    unlock(y);
    unlock(x);
}
```

Thread B:

```
void p2(void *ignored) {
    lock(y);
    lock(x);
    /* critical section */
    unlock(x);
    unlock(y);
}
```



This lock pattern exhibits non-deterministic deadlock

- Sometimes it happens, sometimes it doesn’t!

This is really hard to debug!

Deadlock Questions

Can you have deadlock w/o mutexes?

Deadlock Example: Memory Contention

```
mutex_t m1, m2;
```

Thread A:

```
void p1(void *ignored) {
    AllocateOrWait(1 MB)
    AllocateOrWait(1 MB)
    /* do something */
    Free(m2);
    unlock(m1);
}
```

Thread B:

```
void p2(void *ignored) {
    AllocateOrWait(1 MB)
    AllocateOrWait(1 MB)
    /* critical section */
    unlock(m1);
    unlock(m2);
}
```

If only 2 MB of space, we get same deadlock situation

Deadlock Example: Memory Contention

```
mutex_t m1, m2;
```

Thread A:

```
void p1(void *ignored) {
    AllocateOrWait(1 MB)

    AllocateOrWait(1 MB)
    /* do something */
    Free(m2);
    unlock(m1);
}
```

Thread B:

```
void p2(void *ignored) {

    AllocateOrWait(1 MB)

    AllocateOrWait(1 MB)
    /* critical section */
    unlock(m1);
    unlock(m2);
}
```

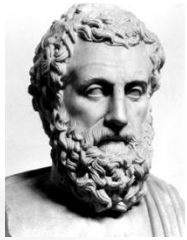
If only 2 MB of space, we get same deadlock situation

Deadlock Questions

Can you have deadlock w/o mutexes?

- Threads often block waiting for resources
  - Locks
  - Terminals
  - Printers
  - CD drives
  - Memory
- Same problem with condition variables
  - Suppose resource 1 managed by *c1*, resource 2 by *c2*
  - A has 1, waits on *c2*, B has 2, waits on *c1*
- Threads often block waiting for other threads
  - Pipes
  - Sockets
- You can deadlock on any of these!

Dining Philosophers Problem

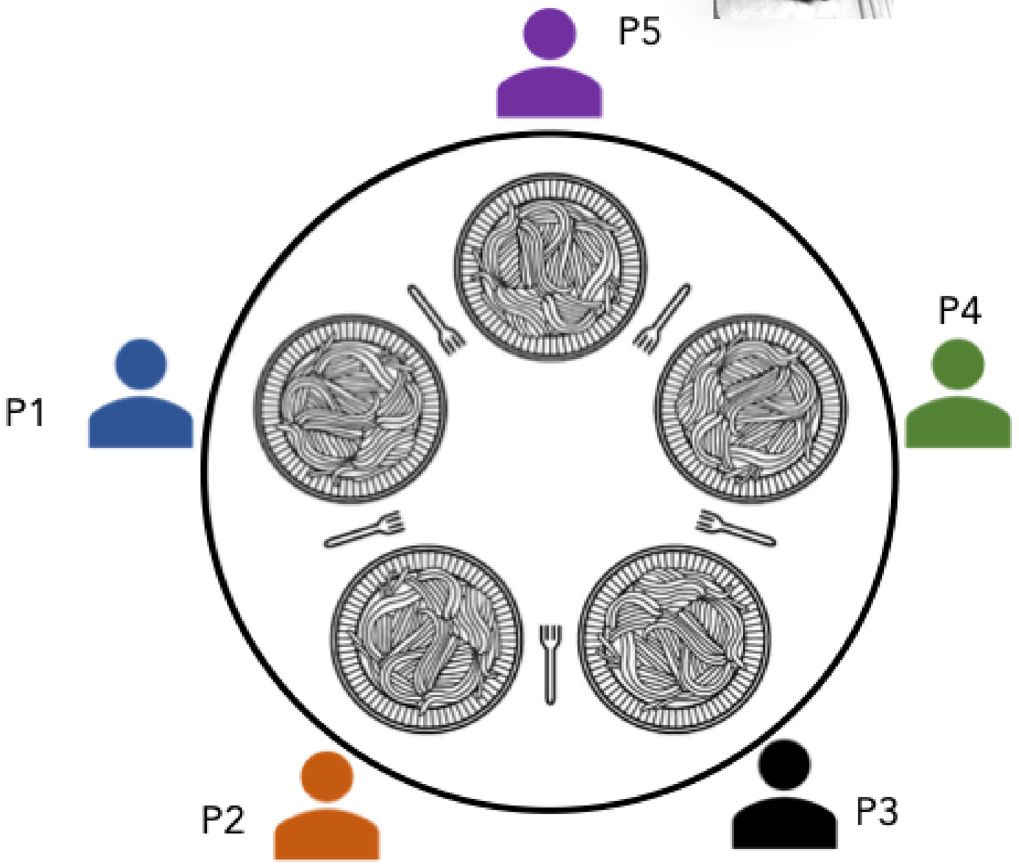


Philosophers spend their lives alternating thinking and eating

Don't interact with neighbors, occasionally eat

- Need 2 chopsticks to eat
- Release both when done

Can only pick up 1 fork at a time



Dining Philosophers in Code

```
#define N 5 /* number of philosophers */
semaphore forks[N]; /* semaphores for each fork, each initialized to 1 (omitted) */
```

```
void philosopher(int i) /* i: philosopher id, 0 to 4
*/
{
  while (true) {
    think(); /* philosopher is thinking */
    take_fork(i); /* take left fork */
    take_fork((i + 1) % N); /* take right fork */
    eat(); /* yum-yum, spaghetti */
    put_fork(i); /*put left fork back on the table*/
    put_fork((i + 1) % N); /* put right fork back on
the table */
  }
}
```

```
void take_fork(int i) {
  forks[i].P();
  /*wait for ith fork's semaphore*/
}

void put_fork(int i) {
  forks[i].V();
  /*signal ith fork's semaphore*/
}
```

What problem with this code?

How to Avoid Deadlock Here?

Multiple solutions exist:

How to Avoid Deadlock Here?

Multiple solutions exist:

Simple one: allow at most 4 philosophers to sit simultaneously at the table

Another solution: define a partial order for resources (forks)

- Number the forks
- Philosopher must always pick up lower-numbered fork first and then higher-numbered fork
- What happens if four philosophers all pick up their lower-numbered fork?
- Disadvantage
  - Not always practical, when the complete list of all resources is not known in advance

Third solution: all or none each time

2nd Attempt to Dining Philosopher Problem

Fix the previous code

```
#define N 5 /* number of philosophers */
semaphore forks[N]; /* semaphores for each fork, each initialized to 1 (omitted) */
```

```
void philosopher(int i) /* i: philosopher id, 0 to 4
*/
{
  while (true) {
    think(); /* philosopher is thinking */
    take_fork(i); /* take left fork */
    take_fork((i + 1) % N); /* take right fork */
    eat(); /* yum-yum, spaghetti */
    put_fork(i); /*put left fork back on the table*/
    put_fork((i + 1) % N); /* put right fork back on
the table */
  }
}
```

```
void take_fork(int i) {
  forks[i].P();
  /*wait for ith fork's semaphore*/
}

void put_fork(int i) {
  forks[i].V();
  /*signal ith fork's semaphore*/
}
```

2^{nd} Attempt to Dining Philosopher Problem

```
#define N 5 /* number of philosophers */
#define LEFT (i+N-1) % N /* i's left neighbor */
#define RIGHT (i+1) % N /* i's right neighbor */
enum State {THINKING, HUNGRY, EATING}; /* a philosopher's status */
enum State states[N]; /* keep track of each philosopher's status */
semaphore mutex = 1; /* mutual exclusion for critical section */
semaphore phis[N]; /* semaphore for each philosopher, init to 0 */

void philosopher(int i) /* i: philosopher id, 0 to N-1 */
{
    while (true) {
        think(); /* philosopher is thinking */
        take_forks(i); /* take both forks */
        eat(); /* yum-yum, spaghetti */
        put_forks(i); /* put both forks */
    }
}
```

2nd Attempt to Dining Philosopher Problem

```
void take_forks(int i) /* i: philosopher id, 0 to N-1 */
{
  mutex.P(); /* enter critical section */
  states[i] = HUNGRY; /* indicate philosopher is hungry */
  test(i); /* try to acquire two forks */
  mutex.V(); /* exit critical section */
  phis[i].P(); /* block if forks not acquired */
}

void put_forks(int i) { /* i: philosopher id, 0 to N-1 */
  mutex.P(); /* enter critical section */
  states[i] = THINKING; /* indicate i finished eating */
  test(LEFT); /* see if left neighbor can eat now */
  test(RIGHT); /* see if right neighbor can eat now */
  mutex.V(); /* exit critical section */
}
```

2nd Attempt to Dining Philosopher Problem

```
void test(int i) /* i: philosopher id, 0 to N-1 */
{
  if (states[i] == HUNGRY &&
    states[LEFT] != EATING &&
    states[RIGHT] != EATING) {
    states[i] = EATING; /* philosopher I can eat now */
    phis[i].V(); /* signal i to proceed */
  }
}
```

Notes for the Solution

What is the purpose of states **array?**

- given that already have the semaphore array?
- A semaphore doesn’t have operations for checking its value!

What if we don’t use the mutex semaphore**?**

Why the semaphore array is for each philosopher?

- Our first attempt uses semaphore array for each fork

What if we put phis[i].P(); **inside the critical section?**

What if we don’t call the two test in put_forks**?**

Conditions for Deadlock

- Mutual exclusion – At least one resource must be held in a non sharable mode
- Hold and wait – There must be one process holding one resource and waiting for another resource
- No preemption – Resources cannot be preempted (critical sections cannot be aborted externally)
- Circular wait – There must exist a set of processes [P1, P2, P3,...,Pn] such that P1 is waiting for P2, P2 for P3, etc.

Questions

How to detect deadlocks?

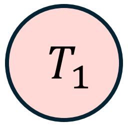
Conditions for Deadlock

View system as graph

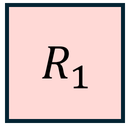
- Processes and Resources are nodes
- Resource Requests and Assignments are edges

Resource-Allocation Graph:

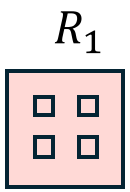
- A set of Threads $T_1, T_2, \ldots, T_n$



- Resource types $R_1, R, \ldots, R_n$
  - CPU cycles, memory space, I/O devices

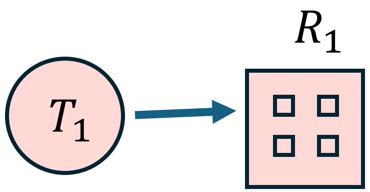


- Each resource type R_i has W_i instances

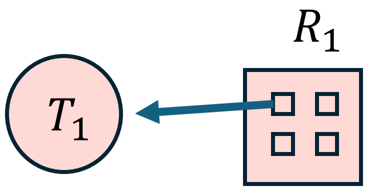


- Each thread utilizes a resource as follows:
  - Request() / Use() / Release()

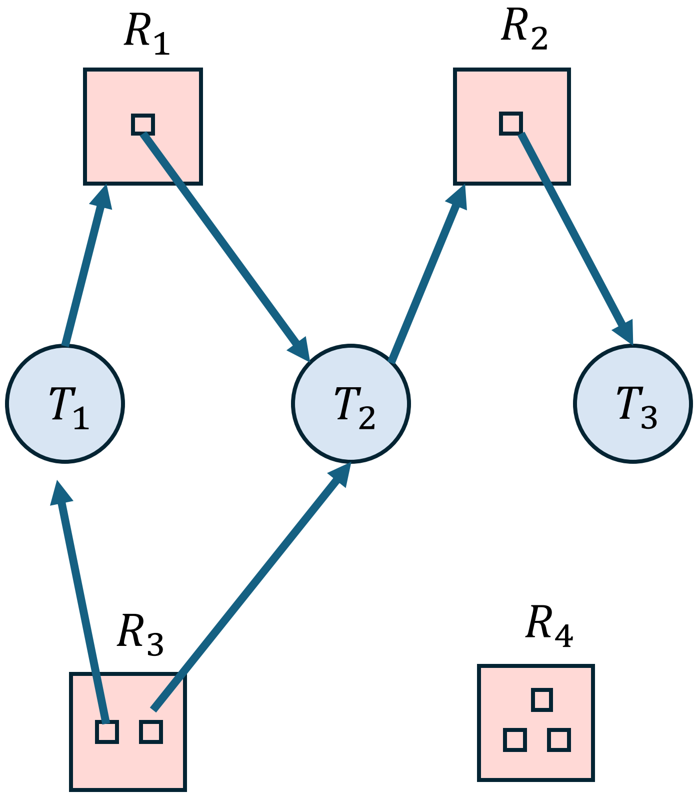
- Thread T_i requesting resource R_j:



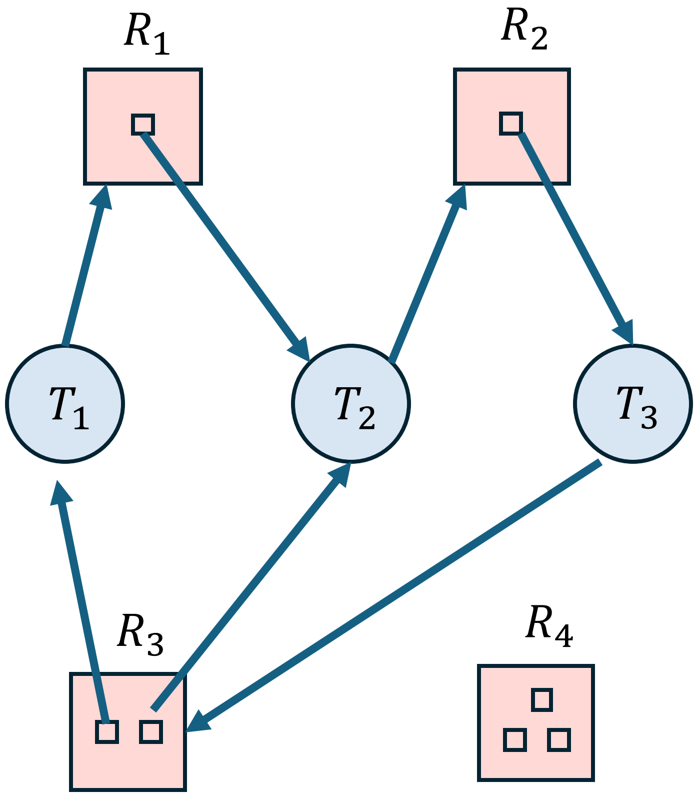
- Thread T_i holding instance of R_j:



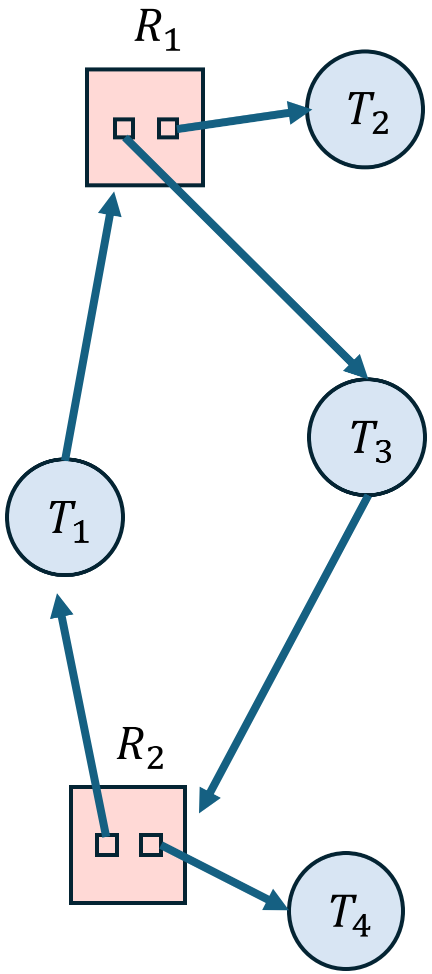
Resource-Allocation Graph Example



Resource-Allocation Graph Example



Is This Deadlock?



Deadlock Detection

If graph has no cycles ⇒ no deadlock

If graph contains a cycle

- Definitely deadlock if only one instance per resource (waits-for graph (WFG))
- Otherwise, maybe deadlock, maybe not

Traverse the resource graph is expensive

- Many processes and resources to traverse

Only invoke detection algorithm periodically

Deal with Deadlock

There are four approaches for dealing with deadlock:

- Ignore it
- Prevention: write your code to make it impossible for deadlock to happen
- Avoidance – control allocation of resources
- Recovery – look for a cycle in dependencies

Prevent by Eliminating One Condition

1. Mutual exclusion
   - Buy more resources, split into pieces, or virtualize to make "infinite" copies
   - Give illusion of infinite resources (e.g. virtual memory)

Virtually Infinite Resources

Thread A:

```
void p1(void *ignored) {
    AllocateOrWait(1 MB)
    AllocateOrWait(1 MB)
    /* do something */
    Free(m2);
    unlock(m1);
}
```

Thread B:

```
void p2(void *ignored) {
    AllocateOrWait(1 MB)
    AllocateOrWait(1 MB)
    /* critical section */
    unlock(m1);
    unlock(m2);
}
```

With virtual memory we have "infinite" space so everything will just succeed, thus above example won't deadlock

Prevent by Eliminating One Condition

1. Mutual exclusion
    - Buy more resources, split into pieces, or virtualize to make "infinite" copies
    - Give illusion of infinite resources (e.g. virtual memory)

2. Hold and wait
    - Wait on all resources at once (must know in advance)

3. No preemption
    - Physical memory: virtualized with VM, can take physical page away and give to any process!

4. Circular wait
    - Partial ordering of resources
        - e.g., always acquire mutex *m1* before *m2*
        - Usually design locking discipline for application this way

Request Resource in Partial Order

```
mutex_t x, y;
```

Thread A:

```
void p1(void *ignored) {
    lock(x);
    lock(y);
    /* critical section */
    unlock(y);
    unlock(x);
}
```

Thread B:

```
void p2(void *ignored) {
    lock(y);
    lock(x);
    /* critical section */
    unlock(x);
    unlock(y);
}
```

Thread A:

```
void p1(void *ignored) {
    lock(x);
    lock(y);
    /* critical section */
    unlock(y);
    unlock(x);
}
```

Thread B:

```
void p2(void *ignored) {
    lock(x);
    lock(y);
    /* critical section */
    unlock(x);
    unlock(y);
}
```

Prevent by Eliminating One Condition

4. Circular wait

- Partial ordering of resources
  - e.g., always acquire mutex *m1* before *m2*
  - Usually design locking discipline for application this way
- Make all threads request everything they'll need at the beginning.
  - Problem: Predicting future is hard, tend to over-estimate resources

Recovering from Deadlock

Terminate processes

- Abort all deadlocked processes
  - Processes need to start over again
- Abort one process at a time until cycle is eliminated
  - System needs to rerun detection after each abort

Preempt resources (force their release)

- Need to select process and resource to preempt
- Need to rollback process to previous state
- Need to prevent starvation

Roll back actions of deadlocked threads

- Common technique in databases (transactions)

Avoid Deadlock

Idea solution: When a process requests a resource, OS only grant it when:

- The process can obtain all resources it needs in future requests
- Information in advance about what resources will be needed by processes to guarantee that deadlock will not happen

Tough

- Hard to determine all resources needed in advance
- Good theoretical problem, not as practical to use

Three States

Safe state

- System can delay resource acquisition to prevent deadlock

Unsafe state

- No deadlock yet...
- But threads can request resources in a pattern that unavoidably leads to deadlock

Deadlock avoidance: prevent system from reaching an *unsafe* state

Deadlocked state

- There exists a deadlock in the system
- Also considered "unsafe"

Banker's Algorithm

1. **Each process must state its maximum resource demand**
   - OS tracks available resource, maximum demand of each process

2. **When a process requests resources:**
   - OS check whether the request would lead to an unsafe state

10 units of resource A

P1: Max = 7, Allocated = 3
P2: Max = 5, Allocated = 2
P3: Max = 3, Allocated = 2

Deadlock Summary

Deadlock occurs when processes are waiting on each other and cannot make progress

- Cycles in Resource Allocation Graph (RAG)

Deadlock requires four conditions

- Mutual exclusion, hold and wait, no resource preemption, circular wait

Four approaches to dealing with deadlock:

- Ignore it – Living life on the edge
- Prevention – Make one of the four conditions impossible
- Avoidance – Banker's Algorithm (control allocation)
- Detection and Recovery – Look for a cycle, preempt or abort

Condition Vars & Locks

C/Vs are also used without monitors in conjunction with locks

- void cond_init (cond_t *, ...);
- void cond_wait (cond_t *c, mutex_t *m);
  - Atomically unlock mand sleep until csignaled
  - Then re-acquire m and resume executing
- void cond_signal (cond_t *c);
- void cond_broadcast (cond_t *c);
  - Wake one/all threads waiting on c

Condition Vars & Locks

C/Vs are also used without monitors in conjunction with locks

A monitor ≈ a module whose state includes a C/V and a lock

- Difference is syntactic; with monitors, compiler adds the code

It is "just as if" each procedure in the module calls acquire() on entry and release() on exit

- But can be done anywhere in procedure, at finer granularity

With condition variables, the module methods may wait and signal on independent conditions

Condition Vars & Locks

C/Vs are also used without monitors in conjunction with locks

A monitor ≈ a module whose state includes a C/V and a lock

- Difference is syntactic; with monitors, compiler adds the code

It is "just as if" each procedure in the module calls acquire() on entry and release() on exit

- But can be done anywhere in procedure, at finer granularity

With condition variables, the module methods may wait and signal on independent conditions

Condition Vars & Locks

Why must cond_wait **both release** mutex_t **& sleep?**

- void cond_wait(cond_t *c, mutex_t *m);

Why not separate mutexes and condition variables?

```
while (count == BUFFER_SIZE) {
    mutex_unlock(&mutex);
    cond_wait(&not_full);
    mutex_lock(&mutex);
}
```

Condition Vars & Locks

Why must cond_wait **both release** mutex_t **& sleep?**

- void cond_wait(cond_t *c, mutex_t *m);

Why not separate mutexes and condition variables?

Producer:

```
while (count == BUFFER_SIZE) {
    mutex_unlock(&mutex);



    cond_wait(&not_full);
    mutex_lock(&mutex);
}
```

Consumer:

```
while (count == BUFFER_SIZE) {

    mutex_unlock(&mutex);
    count --;
    cond_signal(&not_full);
    mutex_lock(&mutex);
}
```

Monitors and Java

A lock and condition variable are in every Java object

- No explicit classes for locks or condition variables

Every object is/has a monitor

- At most one thread can be inside an object's monitor
- A thread enters an object's monitor by
  - Executing a method declared "synchronized"
  - Executing the body of a "synchronized" statement
- The compiler generates code to acquire the object's lock at the start of the method and release it just before returning
  - The lock itself is implicit, programmers do not worry about it

Condition Vars & Locks

Every object can be treated as a condition variable

- Half of Object's methods are for synchronization!

Take a look at the Java Object class:

- Object.wait(*) is Condition::wait()
- Object.notify() is Condition::signal()
- Object.notifyAll() is Condition::broadcast()

Next time…

Read Chapter 15, 16, 18