IOWA STATE UNIVERSITY
Department of Electrical and Computer Engineering

# Lecture 15: Deadlocks

# Agenda

- Recap
- Deadlocks
  - Deadlock Concepts
  - Deadlock Solutions

# Recap

- Conditional Variables
  - Allows a thread to wait till a condition is satisfied
  - Testing the condition must be done within a mutex
  - A mutex is associated with every condition variable
    - A mutex is passed into wait:
      pthread_cond_wait(cond_var, mutex)

    - **Mutex is unlocked before the thread sleeps**
    - **Mutex is locked again before pthread_cond_wait() returns**
    - Safe to use pthread_cond_wait() in a while loop and check condition again before proceeding

# Recap

- Example

```
int    thread1_done = 0;

pthread_cond_t    cv;
pthread_mutex_t  mutex;
```

Thread 1:

```
printf("hello ");

pthread_mutex_lock(&mutex);
thread1_done = 1;
pthread_cond_signal(&cv);
pthread_mutex_unlock(&mutex);
```

Thread 2:

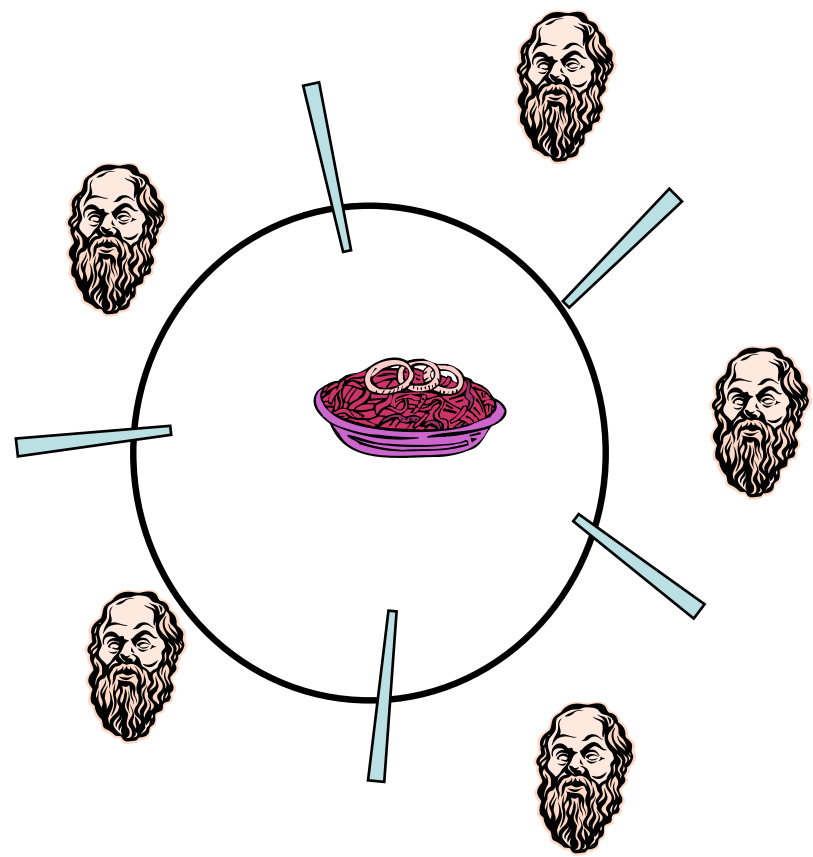
```
pthread_mutex_lock(&mutex);

while (thread1_done == 0) {
    pthread_cond_wait(&cv,
    &mutex);
}

printf(" world\n");
pthread_mutex_unlock(&mutex);
```

# Recap

- Dining Philosophers Problem
  - Philosopher
    - eat, think
    - eat, think
    - ……..
  - Philosopher = Process
  - Eating needs two resources
  - Issues with solutions
    - Deadlock
    - Starvation
    - Poor performance



# Recap

- Readers-Writers Problem
  - Multiple threads reading/writing
  - Many threads can read simultaneously
  - Only one can be writing at any time
    - When a writer is executing, nobody else can read or write
  - One solution idea
    - Readers:
      - First reader locks the database
      - If a reader inside, other readers enter without locking again
      - Checking for readers occurs within a mutex
    - Writer:
      - Always lock database before entering
      - May run into starvation

# Recap

- Example solution

READER:
```
While (1) {
    down(protector);
    rc++;
    if (rc == 1)  //first reader
        down(database);
    up(protector);

    read();

    down(protector);
    rc--;
    If (rc == 0) then // last one
        up(database);
    up(protector);
    ….
}
```

WRITER:
```
While (1) {
    generate_data();
    down(database);
    write();
    up(database);
}
```

Two semaphores:

database

protector

Initial: protector=1, database =1

rc =0

# Agenda

- ~~Recap~~
- Deadlocks
  - Deadlock Concepts
  - Deadlock Solutions

# Deadlock Concepts

- What types of bugs exist?
  - A survey on four major open-source applications
    - MySQL, Apache, Mozilla, OpenOffice.

| Application | What it does | Non-Deadlock | Deadlock |
|---|---|---|---|
| MySQL | Database Server | 14 | 9 |
| Apache | Web Server | 13 | 4 |
| Mozilla | Web Browser | 41 | 16 |
| Open Office | Office Suite | 6 | 2 |
| **Total** | | **74** | **31** |

- Non-deadlock bugs:
  - Atomicity violation, ordering violation, ...

# Deadlock Concepts

- What's deadlock?
  - "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."
    - A Treasury of Railroad Folklore, B.A. Botkin & A.F. Harlow, p. 381

# Deadlock Concepts

- A more formal definition
  - A set of processes/threads is deadlocked if each process/thread in the set is waiting for an event that only another process/thread in the set can cause
- Usually the event is release of a currently held resource
- None of the processes/threads can …
  - run
  - release resources

# Deadlock Concepts

- Why do deadlocks occur?

# Deadlock Concepts

- Why do deadlocks occur?
  - Reason 1:
    - In large code bases, complex dependencies arise between components.
  - Reason 2:
    - Due to the nature of encapsulation
      - Hide details of implementations and make software easier to build in a modular way.
      - Such modularity *does not mesh* well with locking.

# Deadlock Concepts

- Four conditions need to hold for a deadlock to occur

| Condition | Description |
|---|---|
| Mutual Exclusion | Threads claim exclusive control of resources that they require. |
| Hold-and-wait | Threads hold resources allocated to them while waiting for additional resources |
| No preemption | Resources cannot be forcibly removed from threads that are holding them. |
| Circular wait | There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain |

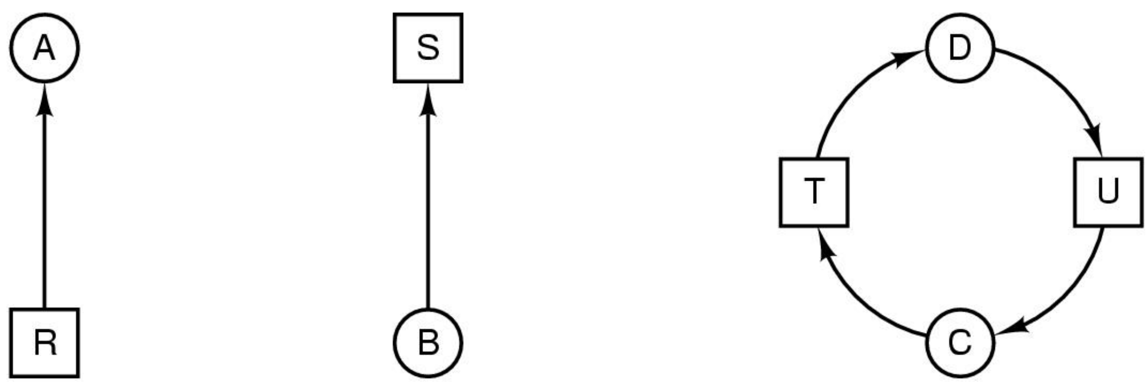
  - If any of these four conditions are not met, **deadlock cannot occur**.

# Agenda

- ~~Recap~~
- Deadlocks
  - ~~Deadlock Concepts~~
  - Deadlock Solutions

# Deadlock Solutions

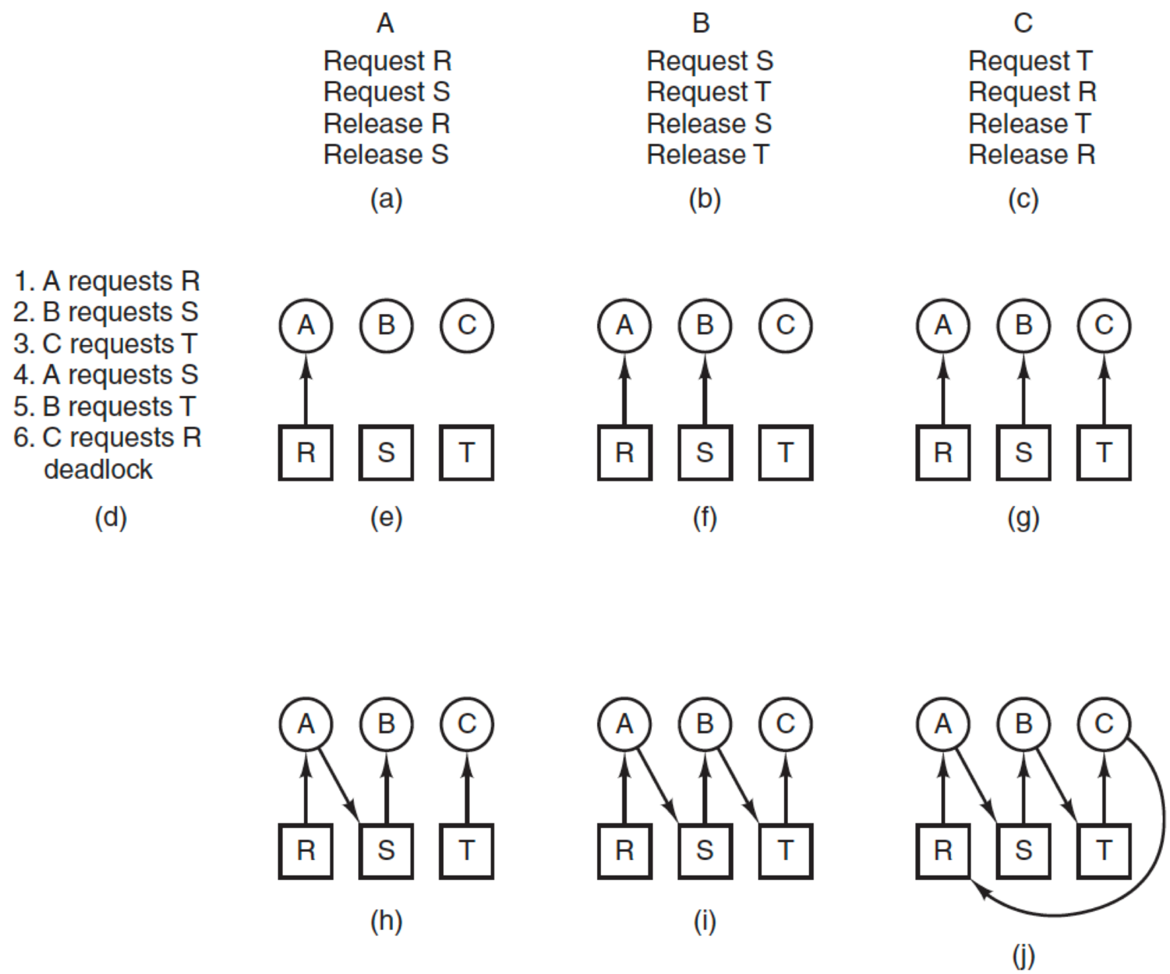
- Deadlock modeling
  - **Resource allocation graph**
    - resource R assigned to process A
    - process B is requesting/waiting for resource S
    - process C and D are in deadlock over resources T and U




(a) (b) (c)

# Deadlock Solutions

- Deadlock modeling



# Deadlock Solutions

- Deadlock detection & recovery
  - **Allow deadlock** to occasionally occur and then *take some action*.
    - E.g.: if an OS froze, you would reboot it.

  - Many database systems employ *deadlock detection* and *recovery technique*
    - A deadlock detector **runs periodically**.
    - Building a **resource allocation graph** and checking it for cycles.
    - In deadlock, the **system performs recovery**
      - Preemption, rollback, killing process, ...

# Deadlock Solutions

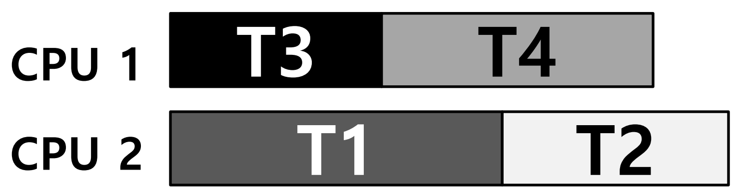
- Deadlock avoidance via scheduling
  - Do not allow deadlock to occur
  - Global knowledge is required:
    - Which locks various threads might grab during their execution.
    - Subsequently schedules said threads in a way as to guarantee no deadlock can occur.

# Deadlock Solutions

- Deadlock avoidance via scheduling
  - E.g., we have two processors and four threads.
    - Lock acquisition demands of the threads:

| | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| L1 | yes | yes | no | no |
| L2 | yes | yes | yes | no |

    - A smart scheduler could compute that as long as T1 and T2 are not run at the same time, no deadlock could ever arise.

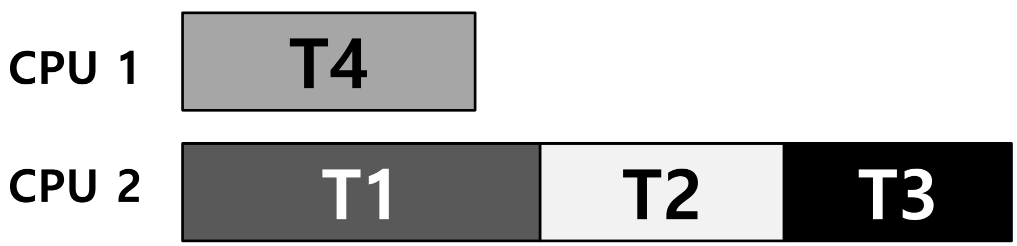


# Deadlock Solutions

- Deadlock avoidance via scheduling
  - E.g., we have two processors and four threads.
    - Lock acquisition demands of the threads:
    - More contention for the same resources

| | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| L1 | yes | yes | yes | no |
| L2 | yes | yes | yes | no |

    - A possible schedule that guarantees that *no deadlock* could ever occur.



# Deadlock Solutions

- Deadlock prevention
  - Lock ordering
    - Provide a total ordering on lock acquisition
      - require *careful design* of global locking strategies.
    - E.g.: there are two locks in the system (L1 and L2); we can prevent deadlock by always acquiring L1 before L2
  - Trylock()
    - Grab the lock (if it is available).
    - Or, return -1: you should try again later.

```
top:
    lock(L1);
    if( tryLock(L2) == -1 ){
            unlock(L1);
            goto top;
    }
```

# Agenda

- ~~Recap~~
- ~~Deadlocks~~
  - ~~Deadlock Concepts~~
  - ~~Deadlock Solutions~~

# Questions?

*acknowledgement: slides include content from “Modern Operating Systems” by A. Tanenbaum, “Operating Systems Concepts” by A. Silberschatz etc., “Operating Systems: Three Easy Pieces” by R. Arpaci-Dusseau etc., and anonymous pictures from internet.