Lecture 14: Deadlocks

Professor G. Sandoval

Some slides derived from : Tanenbaum/Bo, John Regehr, and Brendan Dolan-Gavitt
Thanks !!

Today

- XV6 Locks
- Deadlocks

XV6 Locks

- XV6 represents a lock a **`struct spinlock.`**
- The critical field in the struck is **`locked`** {0 = available, non-zero = being held}

Spinlock.h

```
// Mutual exclusion lock.
struct spinlock {
  uint locked;       // Is the lock held?

  // For debugging:
```

XV6 Locks

- To acquire a lock you call acquire, which is implemented as follows:

```
void
acquire(struct spinlock *lk)
{
  for(;;) {
    if(!lk->locked) {
      lk->locked = 1;
      break;
    }
  }
}
```

XV6 Locks

- To acquire a lock you call acquire, which is implemented as follows:

```
void
acquire(struct spinlock *lk)
{
  for(;;) {
    if(!lk->locked) {
      lk->locked = 1;
      break;
    }
  }
}
```

XV6 Locks

- To acquire a lock you call **acquire**, which is **really** implemented as follows:

```c
// Acquire the lock.
// Loops (spins) until the lock is acquired.
// Holding a lock for a long time may cause
// other CPUs to waste time spinning to acquire it.
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  // It also serializes, so that reads after acquire are not
  // reordered before it.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Record info about lock acquisition for debugging.
  lk->cpu = cpu;
  getcallerpcs(&lk, lk->pcs);
}
```

XV6 Locks

- It's counterpart **release** is implemented as follows:

```
// Release the lock.
void
release(struct spinlock *lk)
{
  if(!holding(lk))
    panic("release");

  lk->pcs[0] = 0;
  lk->cpu = 0;

  xchg(&lk->locked, 0);

  popcli();
}
```

XV6 Locks Usage

```
// Kill the process with the given pid.
// Process won't exit until it returns
// to user space (see trap in trap.c).
int
kill(int pid)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->pid == pid){
      p->killed = 1;
      // Wake process from sleep if necessary.
      if(p->state == SLEEPING)
        p->state = RUNNABLE;
      release(&ptable.lock);
      return 0;
    }
  }
  release(&ptable.lock);
  return -1;
}
```

Today

- XV6 Locks
- Deadlocks

Dining Philosophers

```
#define N 5                          /* number of philosophers */

void philosopher(int i)              /* i: philosopher number, from 0 to 4 */
{
      while (TRUE) {
            think( );                /* philosopher is thinking */
            take_fork(i);            /* take left fork */
            take_fork((i+1) % N);    /* take right fork; % is modulo operator */
            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 */
      }
}
```

Solution to the dining philosophers problem?

Deadlock

- Deadlock: two or more threads are waiting on events that only those threads can generate.
    - Computer don't see the big picture needed to break the stalemate.
    - Hard to handle automatically -> lots of theory, little practice.
- Livelock: thread blocked indefinitely by other thread(s) using a resource.
- Livelock naturally goes away when system load decreases
- Dealock does not go away by itself

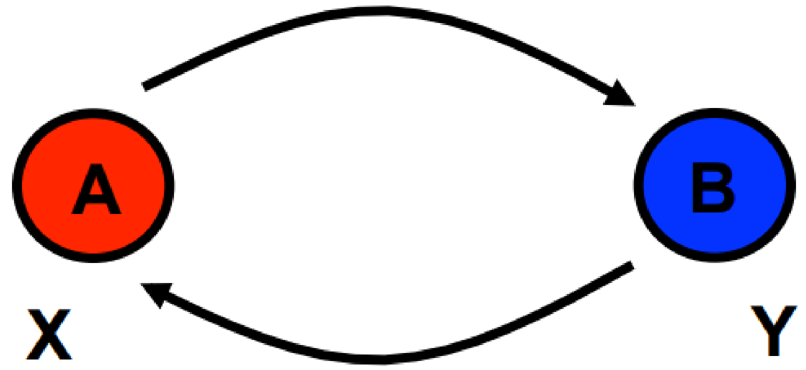
DEADLOCK

Game over, man, game over.

Required Conditions for Deadlock

- Mutual Exclusion: Resources cannot be shared.
- Hold and Wait: A thread is both holding a resource and waiting on another resource to become free.
- No preemption: once are thread gets a resource, it cannot be taken away.
- Circular Wait: There is a cycle in the graph of who has and who wants what.

```
Thread_A:          Thread_B:
   P(X);              P(Y);
   P(Y);              P(X);
```



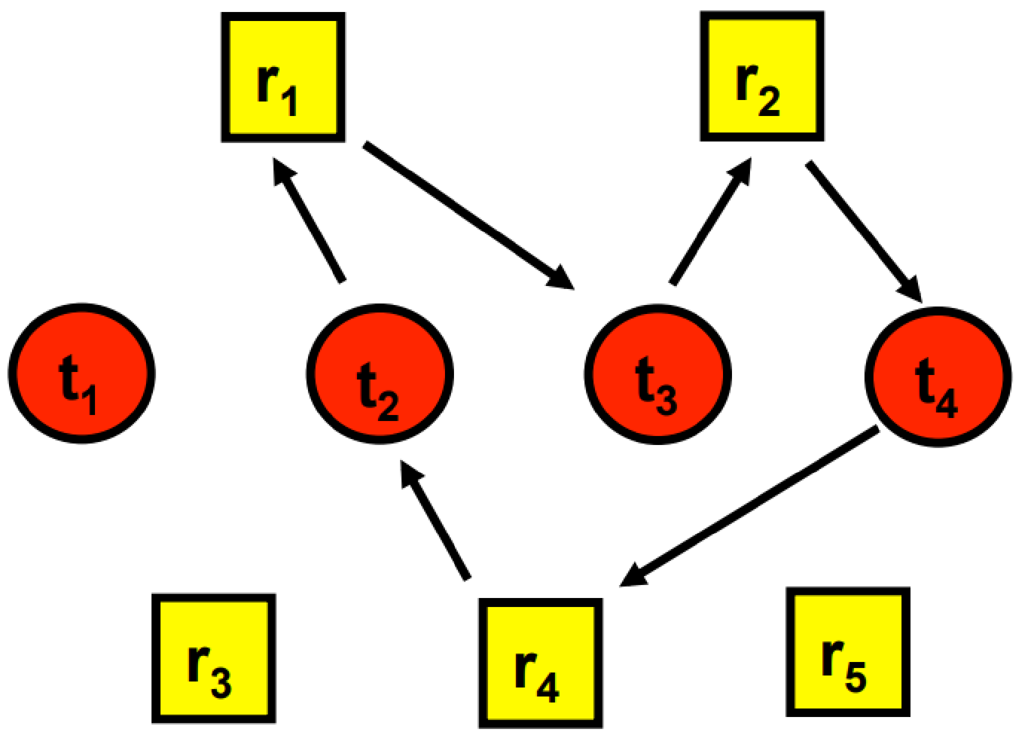
- What’s the smallest number of threads needed to deadlock?

Dealing with Deadlock

- **Deadlock ignorance**
  - Why worry?
- **Deadlock detection**
  - Figure out when deadlock has occurred and "deal with it"
    - » Figuring it out → mildly difficult
    - » Dealing with it → often messy, may need to reboot
- **Deadlock avoidance**
  - Reject resource requests that might lead to deadlock
- **Deadlock prevention**
  - Use "rules" that make it impossible for all four conditions to hold

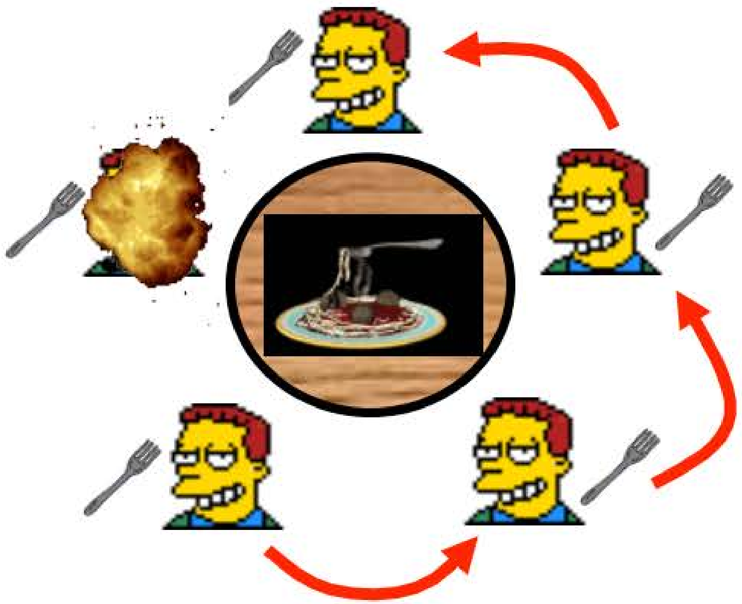
Deadlock Detection

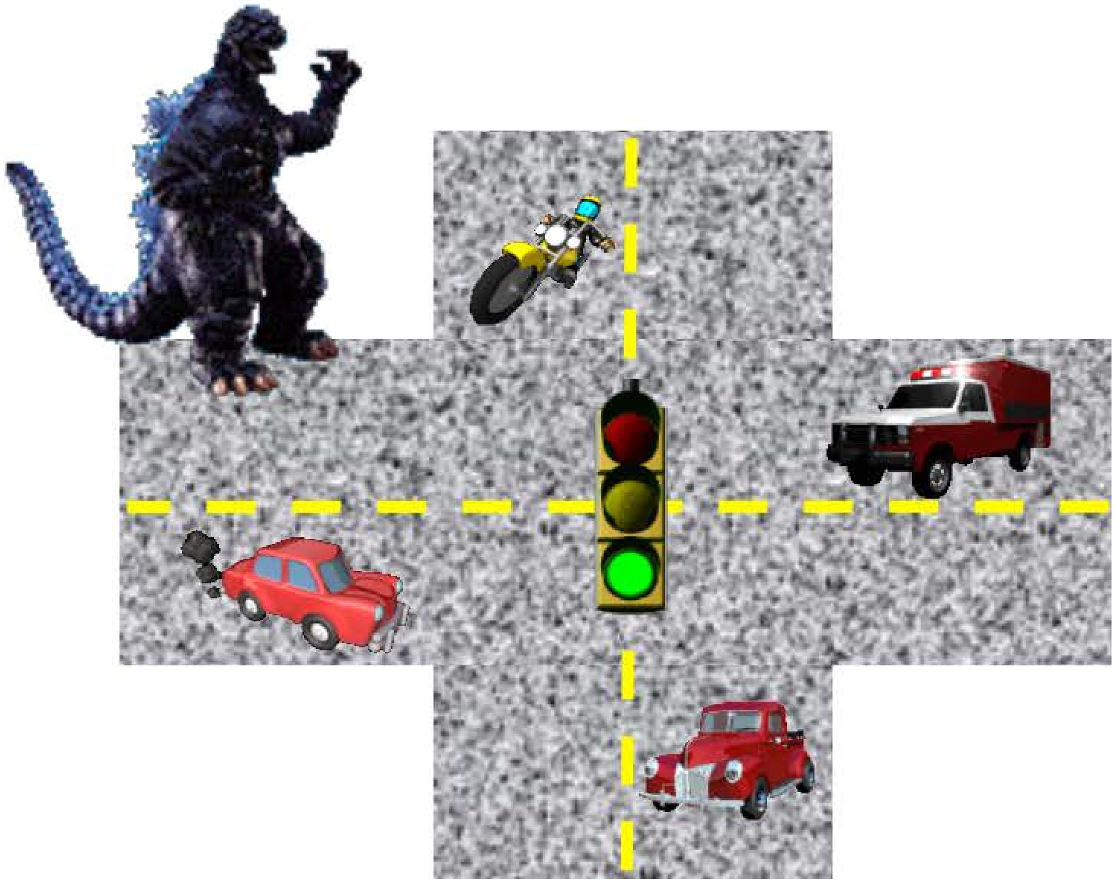
- Resource allocation graph
  - Resources $\{r_1, \ldots, r_m\}$
  - Threads $\{t_1, \ldots, t_m\}$
  - Edges:
    - » $t_i \rightarrow r_j$: t_i wants r_j
    - » $r_j \rightarrow t_i$: r_j allocated to t_i
- Acyclic → no deadlocks
- Cyclic → deadlock
- Cycle detection
  - Several known algorithms
  - $O(n^2)$, $n = |T| + |R|$
- Alternative approach: Wait for someone to hold a resource for way to long, then decide that deadlock has occurred



Deadlock Detection

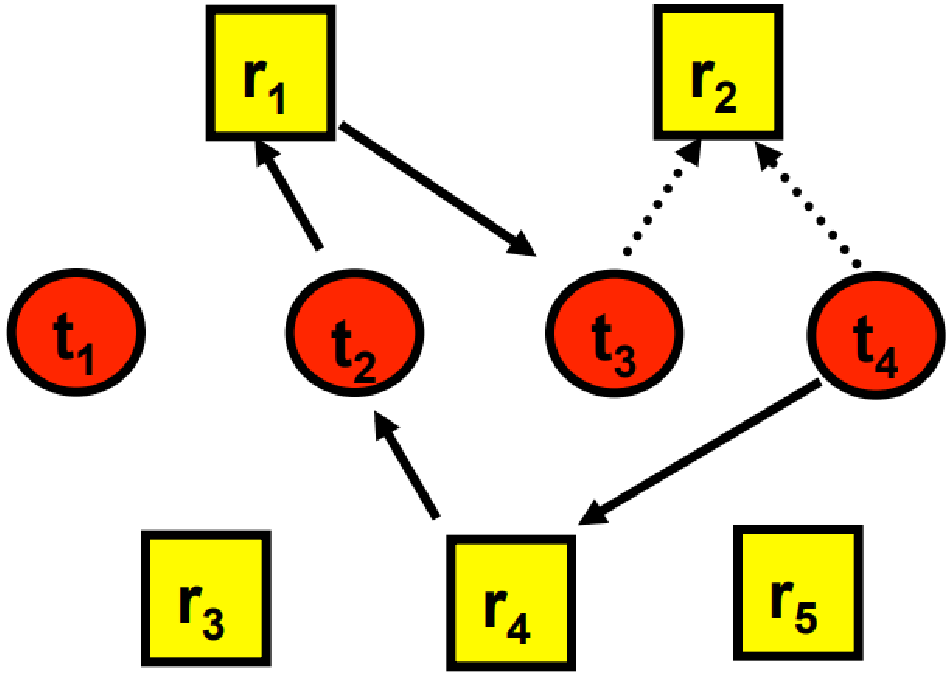
- **Periodically scan for deadlocks**
- **When to scan:**
  - **Just before granting resources?**
  - **Whenever resources unavailable?**
  - **Fixed intervals?**
  - **When CPU utilization drops?**
  - **When thread not runnable for extended period of time?**
- **How to recover?**
  - **Terminate threads**
  - **Break locks**
  - **Invoke exception handlers**
  - **Create more resources**
  - **Reboot**
  - **...**





Deadlock Avoidance

- **Idea:**
  - Run version of deadlock detection algorithm in resource allocator
  - Add "claim" edges to resources threads <u>may</u> request
  - A cycle in extended graph → unsafe state → may lead to deadlock
  - Deny allocations that result in unsafe states
    - » Claim edge converted to request edge
    - » Thread blocks until request can be safely satisfied



Do you think this is used often?

Why or why not?

When to worry?

- When do you have to worry about deadlock?
  - Any time you write code that acquires more than one lock

Approaches in Practice

- User code in Linux
  - Helgrind – a Valgring plugin that checks for circular wait.
- Linux Kernel
  - Lock validator – dynamically checks for circular wait
- Windows thread pool
  - Create more threads if stuck
- Linux CPU Scheduler
  - Avoid starving any thread.

Important from Today

- **Four conditions required for deadlock to occur:**
  - **Mutual exclusion**
  - **Hold and wait**
  - **No resource pre-emption**
  - **Circular wait**
- **Four techniques for handling deadlock**
  - **Prevention: Design rules so one condition cannot occur**
  - **Avoidance: Dynamically deny "unsafe" requests**
  - **Detection and recovery: Let it happen, notice it, and fix it**
  - **Ignore: Do nothing, hope it does not happen, reboot often**