

# CS 3204 Operating Systems

Lecture 13  
Godmar Back

## Announcements

- Project 2 due March 20
- Regroup if necessary
  - Read post on how to move CVS forward
- Won't give detailed milestones this time, but:
  - recommend that you reach the point where "write" works (see 3.2 Suggested Order of Implementation) by March 1

## Optimistic Concurrency Control

## Optimistic Concurrency Control

- Alternative to locks: instead of serializing access, detect when bad interleaving occurred, retry if so

```
void increment_counter(int *counter) {
    do {
        int oldvalue = *counter;
        int newvalue = oldvalue + 1;
        [ BEGIN ATOMIC COMPARE-AND-SWAP INSTRUCTION ]
        if (*counter == oldvalue) { *counter = newvalue; success = true; }
        else { success = false; }
        [ END CAS ]
    } while (!success);
}
```

## Optimistic Concurrency Control (2)

- Other names:
  - lock-free synchronization
  - wait-free synchronization
  - non-blocking synchronization
- x86 supports this via cmpxchg instruction
- Advantages:
  - Less overhead for uncontended locks (faster, and need no storage for lock queue)
  - Synchronizes with IRQ handler automatically
  - Easier to clean up when killing a thread
- Disadvantages
  - Can require lots of retries (more inefficient than even a hot lock since no thread might make progress)

## Deadlock

## Deadlock (Definition)

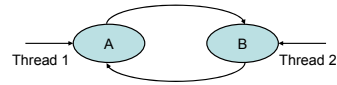
- A situation in which two or more threads or processes are blocked and cannot proceed
- "blocked" either on a resource request that can't be granted, or waiting for an event that won't occur
  - Possible causes: resource-related or communication-related
- Cannot easily back out



## Deadlock Canonical Example (1)

```
pthread_mutex_t A;
pthread_mutex_t B;
...
pthread_mutex_lock(&A);
pthread_mutex_lock(&B);
...
pthread_mutex_unlock(&B);
pthread_mutex_unlock(&A);

pthread_mutex_lock(&B);
pthread_mutex_lock(&A);
...
pthread_mutex_unlock(&A);
pthread_mutex_unlock(&B);
```



## Canonical Example (2)

```
class account {
pthread_mutex_t lock;
int amount; const char *name;
public:
account(int amount, const char *name) :
amount(amount), name(name) { pthread_mutex_init(&this->lock, NULL); }
void transferTo(account *that, int amount) {
pthread_mutex_lock(&this->lock);
pthread_mutex_lock(&that->lock);
cout << "Transferring $" << amount << " from "
<< this->name << " to " << that->name << endl;
this->amount -= amount;
that->amount += amount;
pthread_mutex_unlock(&that->lock);
pthread_mutex_unlock(&this->lock);
}
};
```

```
account acc1(10000, "acc1");
account acc2(10000, "acc2");
// Thread 1:
for (int i = 0; i < 100000; i++)
acc2.transferTo(&acc1, 20);
// Thread 2:
for (int i = 0; i < 100000; i++)
acc1.transferTo(&acc2, 20);
```

Q.: How to fix?

## Canonical Example (2, cont'd)

- Answer: acquire locks in same order

```
void transferTo(account *that, int amount) {
if (this < that) {
pthread_mutex_lock(&this->lock);
pthread_mutex_lock(&that->lock);
} else {
pthread_mutex_lock(&that->lock);
pthread_mutex_lock(&this->lock);
}
/* rest of function */
}
```

## Reusable vs. Consumable Resources

- Distinguish two types of resources when discussing deadlock
- A resource:
  - "anything a process needs to make progress"
- (Serially) **Reusable** resources (*static, concrete, finite*)
  - CPU, memory, locks
  - Can be a single unit (CPU on uniprocessor, lock), or multiple units (e.g. memory, semaphore initialized with N)
- Consumable** resources (*dynamic, abstract, infinite*)
  - Can be created & consumed: messages, signals
- Deadlock may involve reusable resources or consumable resources

## Consumable Resources & Deadlock

```
void client() {
for (i = 0; i < 10; i++)
send(request[i]);
for (i = 0; i < 10; i++) {
receive (reply[i]);
send(ack);
}
}

void server() {
while (true) {
receive(request);
process(request);
send(reply);
receive(ack);
}
}
```

- Assume client & server communicate using 2 bounded buffers (one for each direction)
  - Real-life example: flow-controlled TCP
- Q.: Under what circumstances does this code deadlock?

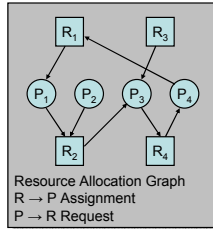
## Deadlocks, more formally

- 4 necessary conditions

- 1) Exclusive Access
- 2) Hold and Wait
- 3) No Preemption
- 4) Circular Wait

- Will look at strategies to

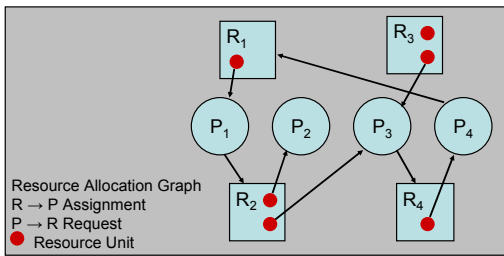
- Prevent
- Avoid
- Detect & break deadlocks



## Deadlock Detection

- Idea: Look for circularity in resource allocation graph
  - Q.: How do you find out if a directed graph has a cycle?
- Can be done eagerly
  - on every resource acquisition/release, resource allocation graph is updated & tested
- or lazily
  - when all threads are blocked & deadlock is suspected, build graph & test
- Windows provides this for its mutexes as an option
- Note: all processes in BLOCKED state is not sufficient to conclude existence of deadlock. (Why?)
- Note: circularity test is only sufficient criteria if there's only a single instance of each resource – see next slide for multi-unit resources

## Multi-Unit Resources



- Note: Cycle, but no deadlock!

## Deadlock Detection

- For reusable resources
  - If each resource has exactly one unit, deadlock iff cycle
  - If each resource has multiple units, existence of cycle may or may not mean deadlock
    - Must use reduction algorithm to determine if deadlock exists (Intuition: remove processes that don't have request edges, return their resource units and remove assignment edges, assign resources to remove request edges, repeat until out of processes without request edges. – if entire graph reduces to empty graph, no deadlock.)
- For consumable resources
  - analog algorithm possible
- Q.: What to do once deadlock is detected?

## Deadlock Recovery

- Preempt resources (if possible)
- Back processes up to a checkpoint
  - Requires checkpointing or transactions (typically expensive)
- Kill processes involved until deadlock is resolved
- Kill all processes involved
- Reboot

Increasing Severity



## Killing Threads or Processes

- Extremely difficult issue:
  - When is it safe to kill a thread?

- Consider:

```
thread_func()
{
    while (!done) {
        lock_acquire(&lock);
        // access shared state
        lock_release(&lock);
    }
}
```

What if thread is killed there?

```
thread_func()
{
    while (!done) {
        lock_acquire(&lock);
        p = queue.get();
        queue.put(p);
        lock_release(&lock);
    }
}
```

- Must guarantee full resource reclamation & consistency of all surviving system data structures

## Deadlock Prevention (1)

- Idea: remove one of the necessary conditions!
- (C1) (Don't require) **Exclusive Access**
  - Duplicate resource or make it shareable (where possible)
- (C2) (Avoid) **Hold and Wait**
  - a) Request all resources at once
    - hard to know in modular system
  - b) Drop all resources if additional request cannot be immediately granted – retry later
    - requires "try\_lock" facility
    - can be inefficient if lots of retries

## Deadlock Prevention (2)

- (C3) (Allow) **Preemption**
  - Take resource away from process
    - Difficult: how should process react?
  - Virtualize resource so it can be taken away
    - Requires saving & restoring resource's state
- (C4) (Avoid) **Circular Wait**
  - Use partial ordering
    - Requires mapping to domain that provides an ordering function (addresses often work!)

## Deadlock Avoidance

- Don't grant resource request if deadlock could occur in future
  - Or don't admit process at all
- Banker's Algorithm (Dijkstra 1965, see book)
  - Avoids "unsafe" states that might lead to deadlock
  - Need to know what future resource demands are ("credit lines" of all customers)
  - Need to capture all dependencies (no additional synchronization requirements – "loans" can be called back if needed)
- Mainly theoretical
  - Impractical assumptions
  - Tends to be overly conservative – inefficient use of resources

## Deadlock in the Real World

- Most common strategy of handling deadlock
  - Test: fix all deadlocks detected during testing
  - Deploy: if deadlock happens, kill and rerun (easy!)
    - If it happens too often, or reproducibly, add deadlock detection code (see next slide for how to do that in Pintos)
- Weigh cost of preventing vs cost of (re-) occurring
- Static analysis tools detects some kinds of deadlocks before they occur
  - Example: Microsoft Driver Verifier
  - Idea: monitor order in which locks are taken, flag if not consistent lock order

## Summary

- Deadlock:
  - 4 necessary conditions: mutual exclusion, hold-and-wait, no preemption, circular wait
- Strategies to deal with:
  - Detect & recover
  - Prevention: remove one of 4 necessary conditions
  - Avoidance: if you can't do that, avoid deadlock by being conservative