CS 3204 Operating Systems

Lecture 15

Godmar Back

Announcements

- Project 1 is due Feb 27, 11:59pm
  - 7 days left
  - Should have finished alarm clock by now
  - Should have finished basic priority
    - priority-change, -preempt, -fifo, -sema, -condvar
  - Should have started on remaining parts
  - Can now attempt to parallelize some development
    - priority donation
    - fixed-point layer: use at least 14 binary digits after period.
    - advanced scheduler
  - Merge early & often, regression test
- Office hours this week: 3-4 MWR, 4-5 F
- Reading assignments: Stallings Chapter 6 & some of 9

Monitors in Java (revisited)

- Recall: Uses single condition variable implicitly associated with buffer object
- This is correct, though, since:
  - Buffer can have either waiting consumers or waiting producers, but never both – hence notify() will always reach the right thread!
- Possible optimization: only notify() if condition changes

```
class buffer {
  private char buffer[];
  private int head, tail;
  public synchronized produce(item i) {
    while (buffer_full())
       this.wait();
    buffer[head++] = i;
    if (buffer_size() == 1) this.notify();
  }
  public synchronized item consume() {
    while (buffer_empty())
       this.wait();
    buffer[tail++] = i;
    if (buffer_size() == CAPACITY-1)
      this.notify();
  }
```

Q.: Think of scenarios when using notify() & a shared condition variable fails

Optimistic Concurrency Control

- Correction to slide in last lecture: "retry" in lock-free synchronization must repeat actual operation:

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

Deadlock

Continued

Canonical Example (2)

```
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);
```

```
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 << "Transfering $" << amount << " from "
         << this->name << " to " << that->name << endl;
    this->amount -= amount;
    that->amount += amount;
    pthread_mutex_unlock(&that->lock);
    pthread_mutex_unlock(&this->lock);
  }
};
```

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

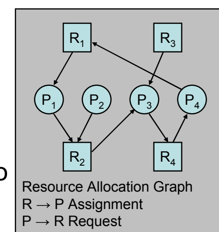
```
void client() {
    for (i = 0; i < 10; i++)
      send(request[i]);
    for (i = 0; i < 10; i++) {
      receive (reply[i]);
      send(ack);
    }
}
```

```
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

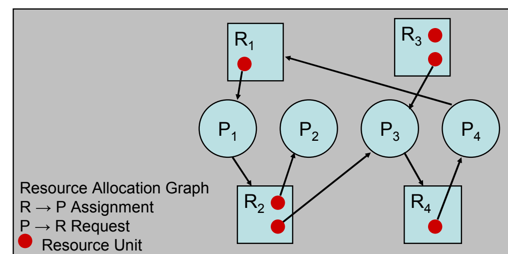
- 4 necessary conditions
  1) Exclusive Access
  2) Hold and Wait
  3) No Preemption
  4) Circular Wait
- Will look at strategies to
  - Prevent
  - Avoid
  - Detect & break deadlocks



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!