CS 537
Lecture 19
Deadlock

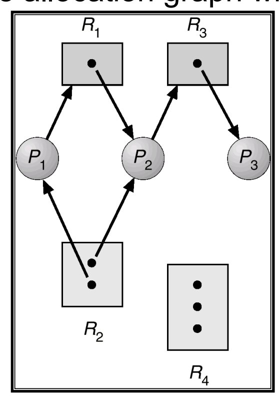
Testing for deadlock

- Steps
  - Collect "process state" and use it to build a graph
    - Ask each process "are you waiting for anything"?
    - Put an edge in the graph if so
  - We need to do this in a single instant of time, not while things might be changing
- Now need a way to test for cycles in our graph

Testing for deadlock

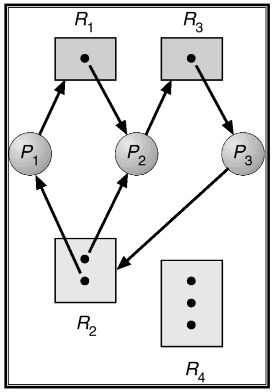
- One way to find cycles
  - Look for a node with no outgoing edges
  - Erase this node, and also erase any edges coming into it
    - Idea: This was a process people might have been waiting for, but it wasn't waiting for anything else
  - If (and only if) the graph has no cycles, we'll eventually be able to erase the whole graph!
- This is called a graph reduction algorithm

Resource allocation graph with no cycle

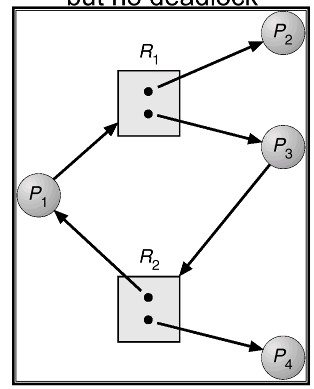


What would cause a deadlock?

Resource allocation graph with a deadlock



Resource allocation graph with a cycle but no deadlock



Some questions you might ask

- If a system is deadlocked, could this go away?
  - No, unless someone kills one of the threads or something causes a process to release a resource
  - Many real systems put time limits on “waiting” precisely for this reason. When a process gets a timeout exception, it gives up waiting and this also can eliminate the deadlock
  - But that process may be forced to terminate itself because often, if a process can’t get what it needs, there are no other options available!

Some questions you might ask

- Suppose a system isn’t deadlocked at time T.
- Can we assume it will still be free of deadlock at time T+1?
  - No, because the very next thing it might do is to run some process that will request a resource…
    - … establishing a cyclic wait
    - … and causing deadlock

Problem 1: can it deadlock?

```
Process 0:               Process 1:

lock1.acquire();         lock1.acquire();
lock2.acquire();         lock2.acquire();
lock1.release();         lock1.release();
lock2.release();         lock2.release();
```


Problem 2: can it deadlock?

```
Process 0:               Process 1:

lock1.acquire();         lock2.acquire();
lock2.acquire();         lock1.acquire();
lock1.release();         lock1.release();
lock2.release();         lock2.release();
```


Problem 3: can it deadlock?

```
Process 0:               Process 1:

lock1.acquire();         lock2.acquire();
lock2.acquire();         lock2.release();
lock1.release();         lock1.acquire();
lock2.release();         lock1.release();
```


Dining Philosophers

- Problem Statement:
  - N Philosophers sitting at a round table
  - Each philosopher shares a fork with neighbor
  - Each philosopher must have both forks to eat
  - Neighbors can't eat simultaneously
  - Philosophers alternate between thinking and eating
- Each philosopher/thread i runs following code:

```
while (1) {
  think();
  take_forks(i);
  eat();
  put_forks(i);
}
```

Dining Philosophers: Attempt #1

- Two neighbors can't use fork at same time
- Must test if fork is there and grab it atomically
  - Represent each fork with a semaphore
  - Grab right fork then left fork
- Code for 5 philosophers:

```
sem_t fork[5]; // Initialize each to 1
take_forks(int i) {
   wait(&fork[i]);
   wait(&fork[(i+1)%5]);
}
put_forks(int i) {
   signal(&fork[i]);
   signal(&fork[(i+1)%5]);
}
```

- What is wrong with this solution???

Dining Philosophers: Attempt #2

- Approach
  - Grab lower-numbered fork first, then higher-numbered
- Code for 5 philosophers:
- `sem_t fork[5]; // Initialize to 1`

```
 take_forks(int i) {
   if (i < 4) {
     wait(&fork[i]);
     wait(&fork[i+1]);
   } else {
     wait(&fork[0]);
     wait(&fork[4]);
 }
```

- What is wrong with this solution???

Dining Philosophers: How to Approach

- Guarantee two goals
  - Safety: Ensure nothing bad happens (don't violate constraints of problem)
  - Liveness: Ensure something good happens when it can (make as much progress as possible)
- Introduce state variable for each philosopher i
  - `state[i] = THINKING, HUNGRY, or EATING`
- Safety: No two adjacent philosophers eat simultaneously
  - `for all i: !(state[i]==EATING && state[i+1%5]==EATING)`
- Liveness: Not the case that a philosopher is hungry and his neighbors are not eating
  - `for all i: !(state[i]==HUNGRY && (state[i+4%5]!=EATING && state[i+1%5]!=EATING))`

Dining Philosophers: Solution

```
sem_t mayEat[5]; // how to initialize?
sem_t mutex; // how to init?
int state[5] = {THINKING};
take_forks(int i) {
   wait(&mutex); // enter critical section
   state[i] = HUNGRY;
   testSafetyAndLiveness(i); // check if I can run
   signal(&mutex); // exit critical section
   wait(&mayEat[i]);
}
put_forks(int i) {
   wait(&mutex); // enter critical section
   state[i] = THINKING;
   test(i+1 %5); // check if neighbor can run now
   test(i+4 %5);
   signal(&mutex); // exit critical section
}
testSafetyAndLiveness(int i) {
   if (state[i]==HUNGRY && state[i+4%5]!=EATING&&state[i+1%5]!=EATING) {
        state[i] = EATING;
        signal(&mayEat[i]);
   }
}
```