CSc 360
Operating Systems
Deadlocks

Jianping Pan

Summer 2006

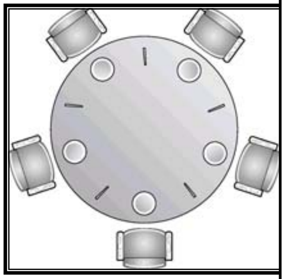
Review

- Ways to process synchronization
  - – hardware-assisted solutions
  - – semaphores
  - – monitors
- Required properties
  - – mutual exclusion
  - – making progress (i.e., no deadlock)
  - – bounded waiting (i.e., no live-lock)

Dining philosophers: semaphores

- Shared data
  - Initially all values are 1

```
semaphore chopstick[5];
```



- Using semaphores, for Philosopher *i*:

```
do {
  wait(chopstick[i])
  wait(chopstick[(i+1) % 5])
     …
   eat
     …
  signal(chopstick[i]);
  signal(chopstick[(i+1) % 5]);
     …
   think
     …
  } while (1);
```

Dining philosophers: monitors

```
monitor DP {
   …

   void test (int i) {
          if ( (state[(i + 4) % 5] != EATING) &&
          (state[i] == HUNGRY) &&
          (state[(i + 1) % 5] != EATING) ) {
               state[i] = EATING ;
               self[i].signal () ; // no effect if not blocked
           }
   }

    initialization_code() {
         for (int i = 0; i < 5; i++)
           state[i] = THINKING;
    }
}
```

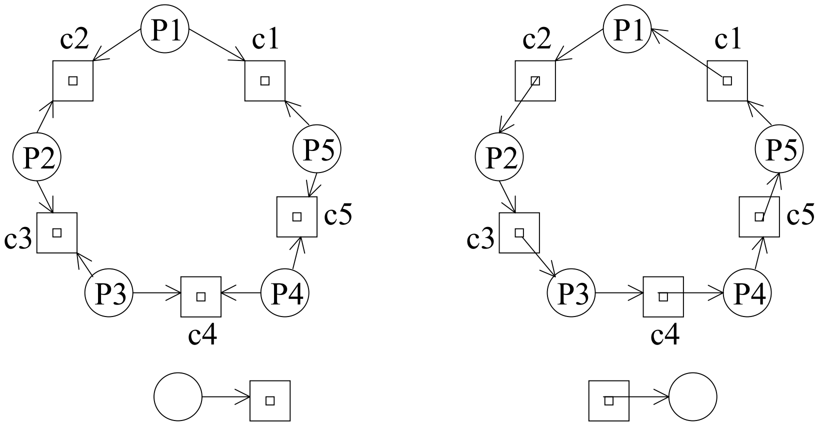
- Using monitors

```
dp.pickup (i)
     ...
   EAT
     ...
dp.putdown (i)
```

Deadlocks

- Deadlock can occur if all are true
  - mutual exclusion
    - wait(chopstick[i]);
  - hold-and-wait
    - **wait(chopstick[i]);** wait(chopstick[(i+1)%5]);
  - no-preemption
    - wait();
  - circular-wait
    - chopstick[(i+1)%5]

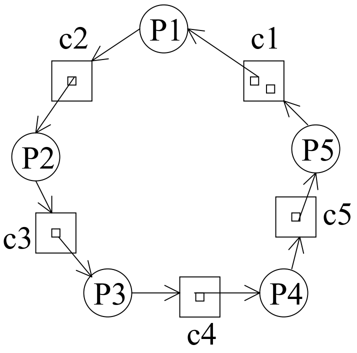
Visualization



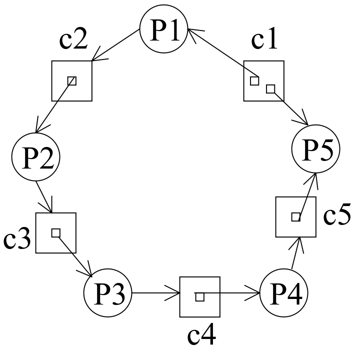
request edge

assignment edge

How about this?



- No directed cycle
  - no deadlock



- Directed cycle
  - one instance per resource type
    - deadlock
  - otherwise
    - *maybe*!

Handling deadlocks

- Prevention
  - mutual exclusion
    - only when mutual exclusion is really necessary
  - hold-and-wait
    - all-or-none
  - non-preemption
    - give up on request
  - circular-wait
    - strictly ordered

```
void test (int i) {
        if ( (state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
                state[i] = EATING ;
                self[i].signal () ;
        }
}

 void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
                self [i].wait;
 }
```

Handling deadlocks: more

- Avoidance
  - – declare maximal resource usage in advance
    - claim edge
  - – check against currently admitted processes
  - – admit if safe (e.g., no circular-wait)
    - single instance resource: resource-allocation graph
    - multi-instance resource: banker’s algorithm
- Detection and recovery

This lecture

- Deadlocks
  - – deadlock characteristics
  - – how to prevent deadlocks
  - – how to avoid deadlocks
- Explore further
  - – CSC 464: Concurrency
  - – NSERC USRA
    - undergraduate student research awards!