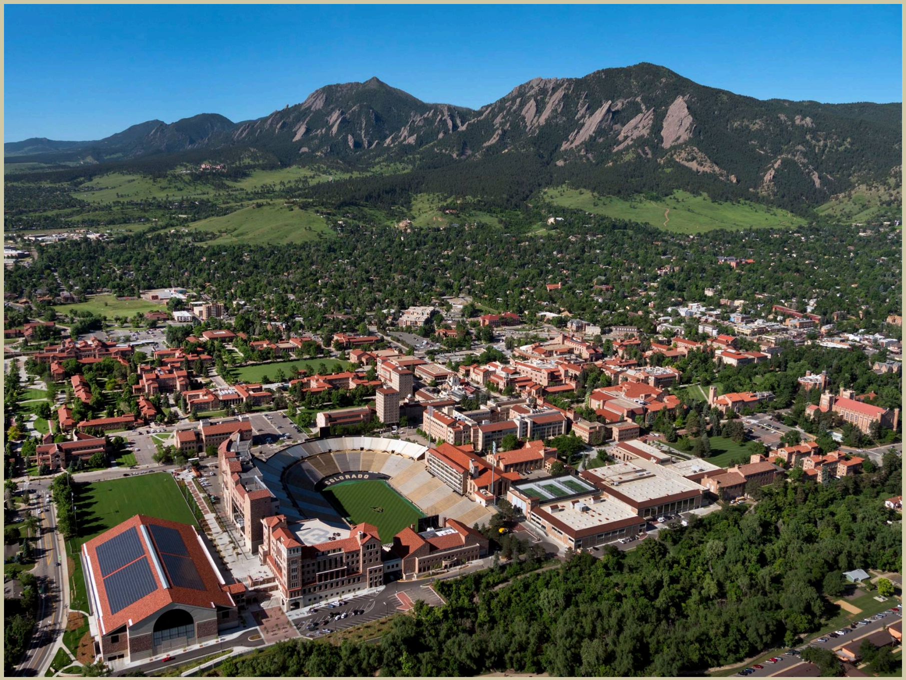
Design and Analysis of Operating Systems
CSCI 3753

Dr. David Knox
University of Colorado Boulder

Dining Philosophers Problem

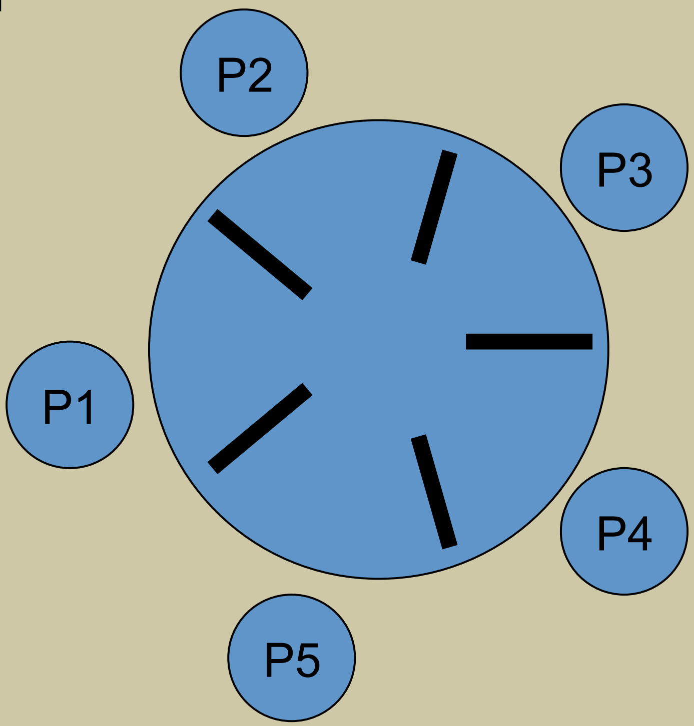


Design and Analysis of
Operating Systems
CSCI 3753

Dr. David Knox
University of Colorado
Boulder

Dining Philosophers Problem

- *N* philosophers seated around a circular table
  - There is one chopstick between each philosopher
  - A philosopher must pick up its two nearest chopsticks in order to eat
  - A philosopher must pick up first one chopstick, then the second one, not both at once
- Devise an algorithm for allocating these limited resources (chopsticks) among several processes (philosophers) in a manner that is
  - deadlock-free, and
  - starvation-free



Dining Philosophers Problem

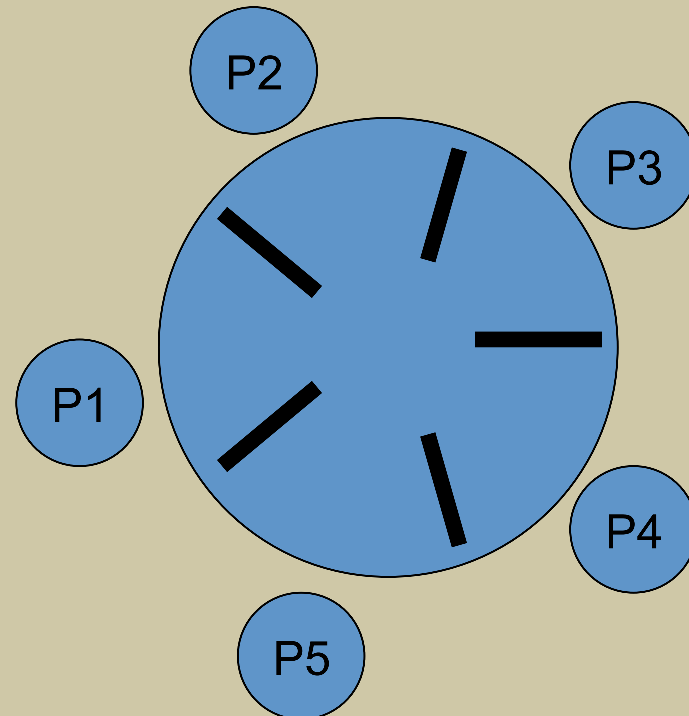
- A simple algorithm for protecting access to chopsticks:
  - Access to each chopstick is protected by a mutual exclusion semaphore
  - prevents any other philosopher from picking up the chopstick when it is already in use by a philosopher

- `semaphore chopstick[5];` // initialized to 1
  - Each philosopher grabs a chopstick *i* by P(chopstick[i])
  - Each philosopher releases a chopstick *i* by V(chopstick[i])

Dining Philosophers Problem

- Pseudo code for Philosopher i:

```
while(1) {
    // obtain 2 chopsticks to my
       immediate right and left
    P(chopstick[i]);
    P(chopstick[(i+1)%N];

    // eat

    // release both chopsticks
    V(chopstick[(i+1)%N];
    V(chopstick[i]);
}
```



Problem?

- Guarantees that no two neighbors eat simultaneously, i.e. a chopstick can only be used by one its two neighboring philosophers

Dining Philosophers Problem

- Unfortunately, the previous "solution" can result in deadlock
  - each philosopher grabs its right chopstick first
    - causes each semaphore's value to decrement to 0
  - each philosopher then tries to grab its left chopstick
    - each semaphore's value is already 0, so each process will block on the left chopstick's semaphore

  - These processes will never be able to resume by themselves - we have deadlock!

Deadlock Can Easily Occur

- Carefully engineered synchronization solutions are required to avoid deadlock
  - The 3 classic synchronization problems like Dining Philosophers, Readers/Writers, and Bounded Buffer P/C

- Semaphores provide mutual exclusion, but can introduce deadlock
  - 2 tasks, each desires a resource locked by the other process
  - Circular dependency
  - can occur easily due to programming errors, e.g. by switching order of P() and V(), etc.

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously:

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

Methods for Handling Deadlocks

- Ensure that the system will ***never*** enter a deadlock state:
  - Deadlock prevention
  - Deadlock avoidance
- Allow the system to enter a deadlock state and then recover
- Ignore the problem and pretend that deadlocks never occur in the system
  - used by most operating systems, including UNIX

Deadlock Prevention

Restrain the ways request can be made

- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources

- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  – Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  – Low resource utilization; starvation possible

Deadlock Prevention (Cont.)

- **No Preemption** –
  - If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - Preempted resources are added to the list of resources for which the process is waiting
  - Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Dining Philosophers Problem

- Deadlock-free solutions?
  - allow at most 4 philosophers at the same table when there are 5 resources
  - odd philosophers pick first left then right, while even philosophers pick first right then left
  - allow a philosopher to pick up chopsticks *only if both are free.*
    - This requires protection of critical sections to test if both chopsticks are free before grabbing them.
    - We'll see this solution next using monitors
- A deadlock-free solution is not necessarily starvation-free
  - for now, we'll focus on breaking deadlock

Dining Philosophers: Monitor-based Solution

- Key insight: pick up 2 forks only if both are free
  - A philosopher can start eating only if both neighbors are not eating
    - Need to define a state for each philosopher
    - Philosopher state: thinking, eating
  - If one of my neighbors is eating, and I'm hungry, ask them to signal() me when they're done
    - Three states of each philosopher: thinking, eating and hungry
    - Need condition variables to signal() waiting hungry philosopher(s)
  - Also, need to Pickup() and Putdown() forks

Dining Philosophers: Monitor-based Solution

```
philosopher (int i)
{
 while (1) {
   //Think
   DiningPhilosophers.pickup(i);
   // pick up forks and eat
   DiningPhilosophers.putdown(i);
 }
}
```

Dining Philosophers: Monitor-based Solution

```
monitor DiningPhilosophers
{
   enum {Thinking, Hungry, Eating} state[5];
   condition self[5]; //to block a philosopher when hungry

   void pickup(int i) {
      //Set state[i] to Hungry
      //If at least one neighbor is eating, block on self[i]
      //Otherwise return
   }

   void putdown(int i) {
      //Change state[i] to Thinking and signal neighbors in
      //case they are waiting to eat
   }
```


... cond. to the next slide

... continuation from the previous slide

```
   void test(int i) {
     //Check if both neighbors of i are not eating and i
     //is hungry
     //If so, set state[i] to Eating, and signal philosopher i
   }

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

```
void pickup(int i) {
    //Set state[i] to Hungry
    //If at least one neighbor is eating, block on self[i]
    //Otherwise return
  }
```

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

```
void test(int i) {
  //Check if both neighbors of i are not eating and i is hungry
  //If so, set state[i] to Eating, and signal philosopher i
}
```

```
void putdown(int i) {
    //change state[i] to Thinking and signal neighbors in
    //case they are waiting to eat
  }
```

```
void putdown(int i) {
  state[i] = Thinking;
  test((i+1)%5);
  test((i-1)%5);
}
```

```
void test(int i) {
 //Check if both neighbors of i are not eating and i is hungry
 //If so, set state[i] to Eating, and signal philosopher i
 }
```

```
void test(int i) {
 //Check if both neighbors of i are not eating and i is hungry
 //If so, set state[i] to Eating, and signal philosopher i
 }
```

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

Dining Philosophers: Monitor-based Solution

- Is deadlock possible in this solution?
  - NO
- Is starvation possible in this solution?
  - YES

Monitor-based Solution to Dining Philosophers (3)

```
monitor DP {
        status state[5];
        condition self[5];
        Pickup(int i);
        Putdown(int i);
        test();
        init();
    }
```

- Each philosopher *i* runs pseudo-code:

```
DP.Pickup(i);
 ... // eat – grab both
          chopsticks
DP.Putdown(i);
```

Monitor-based Solution to Dining Philosophers (4)

```
monitor DP {
    status state[5];
    condition self[5];

    Pickup(int i) {
        state[i] = hungry;
        test(i);
        if(state[i]!=eating)
            self[i].wait;
    }

    test(int i) {
        if (state[(i+1)%5] != eating &&
            state[(i-1)%5] != eating &&
            state[i] == hungry) {

            state[i] = eating;
            self[i].signal();
        }
    }
... monitor code continued next slide ...
```

atomic

atomic

- Pickup chopsticks (atomic)
  - indicate that I'm hungry
  - Atomically test if both my left and right neighbors are not eating. If so, then atomically set my state to eating.
  - if unable to eat, wait to be signaled
- signal() has no effect during Pickup(), but is important to wake up waiting hungry philosophers during Putdown()

Monitor-based Solution to Dining Philosophers (5)

... monitor code continued from previous slide...

...

atomic

```
Putdown(int i) {
        state[i] = thinking;
        test((i+1)%5);
        test((i-1)%5);
    }


    init() {
        for i = 0 to 4
            state[i] = thinking;
    }

}  // end of monitor
```

- Put down chopsticks (atomic)
  - if left neighbor L=(i+1)%5 is hungry and both of L's neighbors are not eating, set L's state to eating and wake it up by signaling L's CV
- Thus, eating philosophers are the ones who (eventually) turn waiting hungry neighbors into active eating philosophers
  - not all eating philosophers trigger the transformation
  - At least one eating philosophers will be the trigger

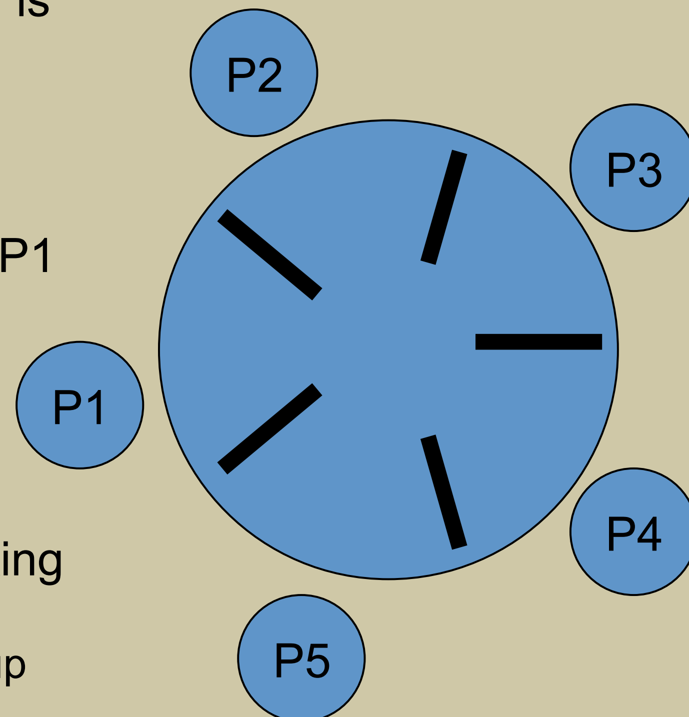
Complete Monitor-based Solution to Dining Philosophers

```
monitor DP {
   status state[5];
   condition self[5];

   Pickup(int i) {
      state[i] = hungry;
      test(i);
      if(state[i]!=eating)
         self[i].wait;
   }

   test(int i) {
      if (state[(i+1)%5] != eating &&
         state[(i-1)%5] != eating &&
         state[i] == hungry) {

         state[i] = eating;
         self[i].signal();
      }
   }

   Putdown(int i) {
         state[i] = thinking;
         test((i+1)%5);
         test((i-1)%5);
   }

   init() {
      for i = 0 to 4
         state[i] = thinking;
   }
}  // end of monitor
```

- Pickup(), Putdown() and test() are all mutually exclusive, i.e. only one at a time can be executing
- Verify that this monitor-based solution is
  - deadlock-free
  - mutually exclusive in that no 2 neighbors can eat simultaneously

DP Monitor Deadlock Analysis

- Try various scenarios to verify for yourself that deadlock does not occur in them
- Start with one philosopher P1
- Now suppose P2 arrives to the left of P1 while P1 is eating
  - What is the perspective from P1?
  - What is the perspective from P2?
- Now supposes P5 arrives to the right of P1 while P1 is eating and P2 is waiting
  - Perspective from P1?
  - Perspective from P5?
  - Perspective from P2?
- Suppose P2 arrives while both P1 and P3 are eating
  - If P1 finishes first, it can't wake up P2
  - But when P3 finishes, its call to test(P2) will wake up P2, so no deadlock
- Suppose there are 6 philosophers and the evens are eating. How do the odds get to eat?



DP Monitor Solution

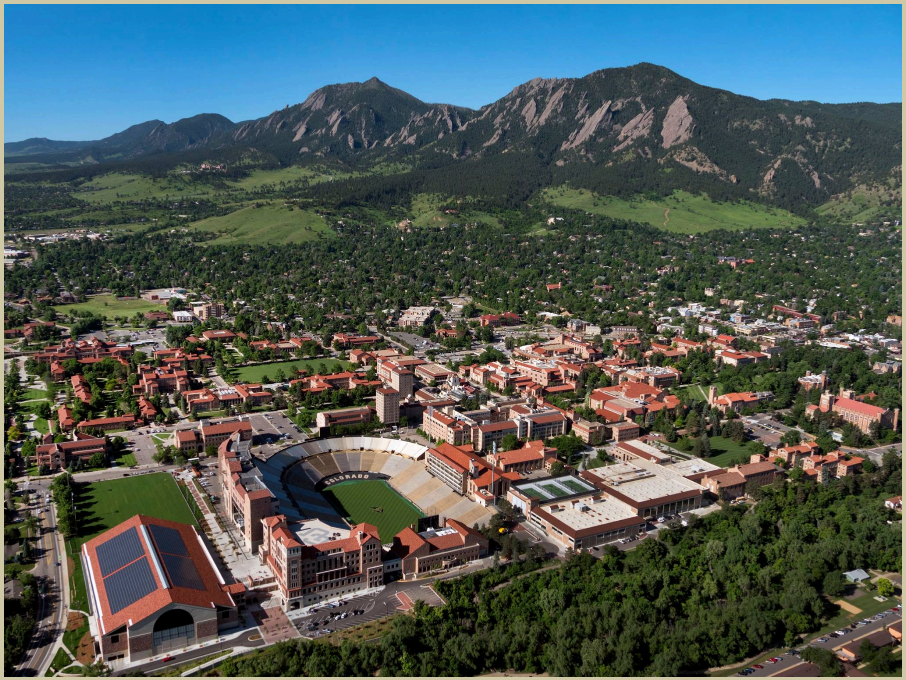
- Note that starvation is still possible in the DP monitor solution
  - Suppose P1 and P3 arrive first, and start eating, then P2 arrives and sets its state to hungry and blocks on its CV
  - When P1 ends eating, it will call test(P2), but nothing will happen, i.e. P2 won't be signaled because the signal only occurs inside the if statement of test, and the if condition is not satisfied
  - Next, P1 can eat again, repeatedly, starving P2

Complete Monitor-based Solution to Dining Philosophers

```
monitor DP {
   status state[5];
   condition self[5];

   Pickup(int i) {
      state[i] = hungry;
      test(i);
      if(state[i]!=eating)
         self[i].wait;
   }

   test(int i) {
      if (state[(i+1)%5] != eating &&
         state[(i-1)%5] != eating &&
         state[i] == hungry) {

         state[i] = eating;
         self[i].signal();
      }
   }
```

```
Putdown(int i) {
         state[i] = thinking;
         test((i+1)%5);
         test((i-1)%5);
   }

   init() {
      for i = 0 to 4
         state[i] = thinking;
   }
}  // end of monitor
```

Design and Analysis of Operating Systems CSCI 3753



Dr. David Knox

University of Colorado
Boulder