



# Deadlocks

---

Xin Liu

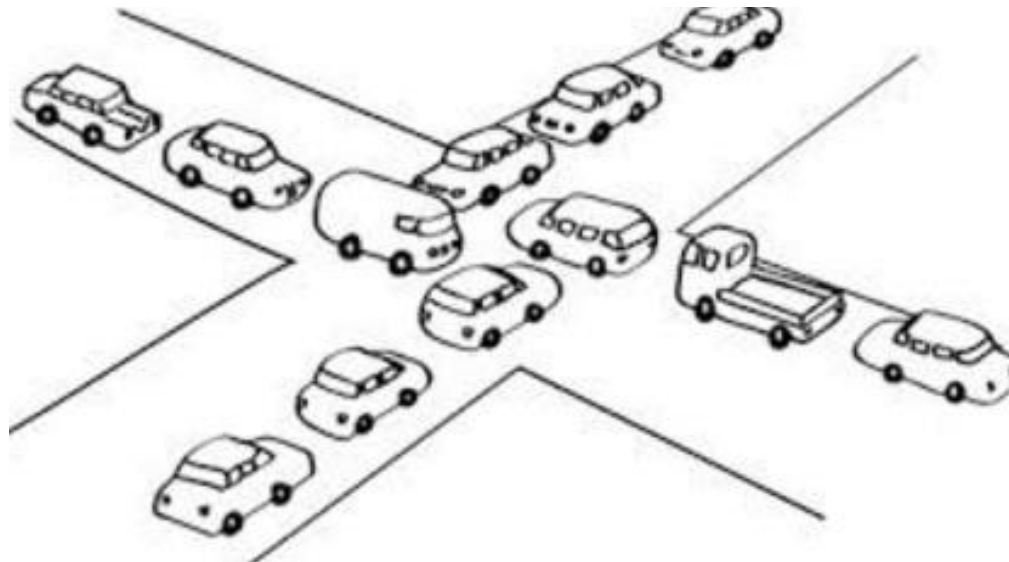
Operating Systems

COP 4610

# Deadlocks

---

- **Deadlocks**: happen when threads are waiting for resources in a circular chain.



# Deadlocks

---

- Deadlocks with ***nonpreemptable resources***
  - ***nonpreemptable resources*** are resources that cannot be forcibly taken away from a thread without causing computation failure
  - E.g., storage allocated to a file
    - If a thread has access to this storage, it cannot be taken away without risking data corruption or computation failure.

# Deadlocks

---

- Deadlocks with *preemptable resources*
  - E.g., the CPU, are resources that can be taken away from a thread without causing failure.
    - If two threads are deadlocked while waiting for the CPU, the system can preempt (take back) the CPU from one thread and give it to the other to resolve the deadlock.

# Starvation

---

## ○ *Starvation:*

- Occurs when a thread is continuously waiting for a resource but never gets it.
- This can happen if other threads keep getting prioritized, leaving the waiting thread indefinitely stuck.
- While starvation can occur without deadlocks, it often implies that the system isn't fairly allocating resources to all threads.
- A deadlock implies starvation

# Example of Deadlocks with 2 Threads

---

Thread A

P(x);

P(y);

Thread B

P(y);

P(x);

- Thread A locks x (P(x)), and Thread B locks y (P(y)) at the same time.
- Then, Thread A tries to lock y, but it can't because Thread B already has y.
- Similarly, Thread B tries to lock x, but it can't because Thread A already has x.

# Example of Deadlock with One Thread

---

```
void os_run() {  
    spin_lock(&list_lock); spin_lock(&xxx);  
    spin_unlock(&xxx);  
}
```

```
void on_interrupt() {  
    spin_lock(&list_lock);  
    spin_unlock(&list_lock);  
}
```

# Example of Deadlock with One Thread

---

- Imagine the thread is running `os_run()` and has already locked `list_lock`.
  - While holding this lock, an interrupt occurs and triggers `on_interrupt()`.
  - Now, `on_interrupt()` tries to lock `list_lock` again, but `list_lock` is already locked by the same thread, causing the thread to wait indefinitely for itself to release the lock, which will never happen because interrupts cannot proceed without the lock being released.
- Self-deadlock (or re-entrance deadlock)
  - The thread holds the `list_lock` in `os_run()` is also the one that tries to lock `list_lock` again in `on_interrupt()`.
  - Since it can't release the lock while it's in the interrupt handler, it gets stuck waiting on itself.





## Deadlocks, Deadlocks, Everywhere...

---

- Deadlocks aren't just about multiple threads
- **Interrupts** or different parts of the same thread can cause deadlock if locks are mismanaged



# Deadlocks, Deadlocks, Everywhere...

---

- Can happen with any kind of resource
  - Among multiple resources
- Cannot be resolved for each resource independently
  - A thread can grab all the memory
  - The other grabs all the disk space
  - Each thread may need to wait for the other to release



## Deadlocks, Deadlocks, Everywhere...

---

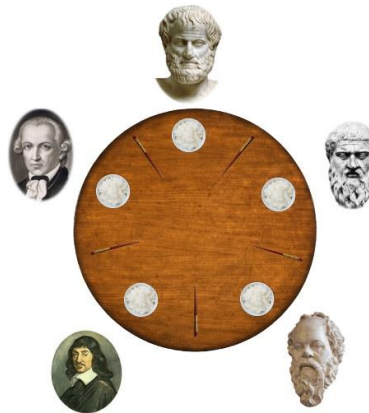
- Round-Robin CPU scheduling cannot prevent deadlocks (or starvation) from happening
- Can occur whenever there is waiting...

# A Classic Example of Deadlocks

---

## ○ Dinning Philosophers (Lawyers)

- You have 5 philosophers sitting around a circular table.
- Between each pair of philosophers, there is one chopstick.
- A philosopher needs two chopsticks (one from their left and one from their right) to eat.
- Philosophers spend their time either thinking or eating.





# Dining Philosophers

---

- If each first grabs the chopstick on their right before the one on their left, and all grab at the same time, we have a deadlock

(Personally, I prefer to starve than share chopsticks...)

# A Dining Philosopher Implementation

---

```
semaphore chopstick[5] = {1, 1, 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[j]);  
        P(chopstick[(j + 1) % 5]);  
        // eat  
        V(chopstick[(j + 1) % 5]);  
        V(chopstick[j]);  
    }  
}
```

# A Dining Philosopher Implementation

---

## ○ Semaphore Initialization:

- The chopstick array is an array of 5 semaphores, each initialized to 1, meaning each chopstick is available.
- The semaphore value 1 indicates that the resource (the chopstick) is available, and 0 means it's in use.

```
semaphore chopstick[5] = {1, 1, 1, 1, 1};
```



# A Dining Philosopher Implementation

---

- P() operation (down or wait operation) is used to lock or take a chopstick (if the value is 1, it is decremented to 0, indicating it is now in use).
- V() operation (up or signal operation) is used to release or put back a chopstick (it increments the semaphore value to 1, making it available).



# A Dining Philosopher Implementation

---

- Philosopher Process:

- Each philosopher first tries to grab the chopstick on their right: `P(chopstick[j]);`
- Then, they try to grab the chopstick on their left: `P(chopstick[(j + 1) % 5]);`
- If successful in grabbing both chopsticks, they can eat (though this part of the code is just commented as `// eat`).
- Once done eating, they release both chopsticks:
  - The left chopstick: `V(chopstick[(j + 1) % 5]);`
  - The right chopstick: `V(chopstick[j]);`



# A Dining Philosopher Implementation

---

- Explanation of  $(j + 1) \% 5$ :
  - This ensures that the philosopher sitting at the 5th position (index 4) wraps around and can access the first chopstick (`chopstick[0]`), as the table is circular.

# Right in Hand, Left on Hold: The Path to Deadlock

---

```
// chopstick[5] = {0, 0, 0, 0, 0};
```

```
philosopher(int j) {  
    while (TRUE) {  
        →→→→→→→→→→ P(chopstick[j]);  
        P(chopstick[(j + 1) % 5]);  
        // eat  
        V(chopstick[(j + 1) % 5]);  
        V(chopstick[j]);  
    }  
}
```

# Right in Hand, Left on Hold: The Path to Deadlock

---

```
// chopstick[5] = {0, 0, 0, 0, 0};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[j]);  
        P(chopstick[(j + 1) % 5]);  
        // eat  
        V(chopstick[(j + 1) % 5]);  
        V(chopstick[j]);  
    }  
}
```





# Conditions for Deadlocks

---

- Four necessary (but not sufficient) conditions
  - Limited access (lock-protected resources)
  - No preemption (if someone has the resource, it cannot be taken away)
  - Wait while holding (holding a resource while requesting and waiting for the next resource)
  - Circular chain of requests

# Deadlock Prevention Techniques

---

- Deadlock  $\Rightarrow$ 
  - All four conditions must be true
  - If deadlock, then all four conditions are true
- All four conditions are true  $\nRightarrow$ 
  - Deadlock
- To prevent deadlocks
  - Remove one of the four conditions



# Deadlock Prevention Techniques

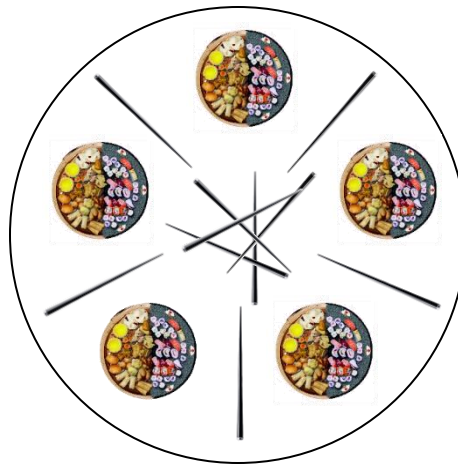
---

- In textbooks:
  - To prevent deadlocks
    - Remove one of the four conditions
- In practice:
  - It is extremely difficult to remove one of the four conditions entirely, making deadlock prevention very challenging in real-world systems

# Deadlock Prevention Techniques

---

1. Infinite resources (buy a very large disk)

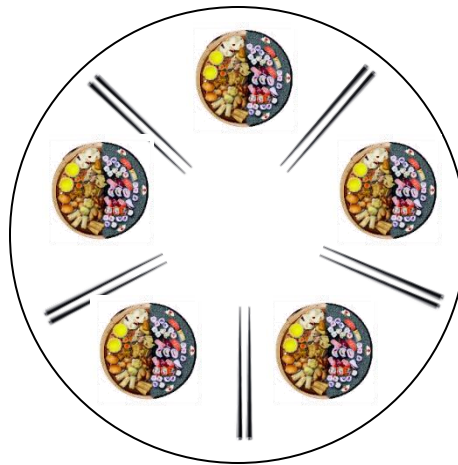




# Deadlock Prevention Techniques

---

## 2. No sharing (independent threads)





# Deadlock Prevention Techniques

---

3. Allocate all resources at the beginning (if you need 2 chopsticks, grab both at the same time)

easier said than done...

# Deadlock Prevention Techniques

---

3. Allocate all resources at the beginning (if you need 2 chopsticks, grab both at the same time)

```
semaphore chopstick[5] = {1, 1, 1, 1, 1}, s = 1; //one more semaphore
philosopher(int j) {
    while (TRUE) {
        P(s);
        P(chopstick[j]);
        P(chopstick[(j + 1) % 5]);
        // eat
        V(chopstick[(j + 1) % 5]);
        V(chopstick[j]);
        V(s);
    }
}
```

# Deadlock Prevention Techniques

3. Allocate all resources at the beginning (if you need 2 chopsticks, grab both at the same time)

**semaphore chopstick[5] = {1, 1 → 0, 1 → 0, 1, 1}, s = 1 → 0;**

**philosopher(int j) {**  
    **while (TRUE) {**  
        **P(s);**

**P(chopstick[j]);**  
        **P(chopstick[(j + 1) % 5]);**  
        **// eat**  
        **V(chopstick[(j + 1) % 5];**  
        **V(chopstick[j]);**

**V(s);**

**}**

**}**

1



# Deadlock Prevention Techniques

3. Allocate all resources at the beginning (if you need 2 chopsticks, grab both at the same time)

1      3  
↓      ↓

```
// chopstick[5] = {1, 0, 0, 1, 1}, s = 0;  
philosopher(int j) {  
    while (TRUE) {  
        P(s);  
        P(chopstick[j]);  
        P(chopstick[(j + 1) % 5]);  
        // eat  
        V(chopstick[(j + 1) % 5]);  
        V(chopstick[j]);  
        V(s);  
    }  
}
```

# Deadlock Prevention Techniques

---

## 3. Adding more semaphores is also a technical skill

```
// chopstick[5] = {1, 1, 1, 1, 1}, s = 1;
philosopher(int j) {
    while (TRUE) {
        P(s);
        P(chopstick[j]);
        P(chopstick[(j + 1) % 5]);
        // eat
        V(chopstick[(j + 1) % 5];
        V(chopstick[j]);
        V(s);
    }
}
```

# Deadlock Prevention Techniques

## 3. Adding more semaphores is also a technical skill

```
// chopstick[5] = {1, 1 → 0, 1 → 0, 1, 1}, s = 1 → 0 → 1;
philosopher(int j) {
    while (TRUE) {
        P(s);

        P(chopstick[j]);
        P(chopstick[(j + 1) % 5];

        // eat

        V(chopstick[(j + 1) % 5];
        V(chopstick[j]);

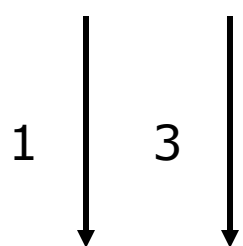
        V(s);
    }
}
```

1



# Deadlock Prevention Techniques

## 3. Adding more semaphores is also a technical skill



```
// chopstick[5] = {1, 0, 0, 1 → 0, 1 → 0}, s = 1 → 0 → 1;
philosopher(int j) {
    while (TRUE) {
        P(s);

        P(chopstick[j]);
        P(chopstick[(j + 1) % 5];

        // eat

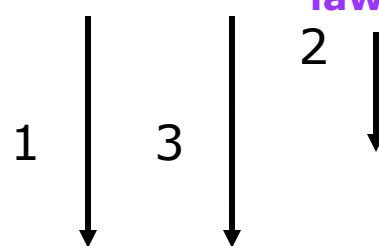
        V(chopstick[(j + 1) % 5];
        V(chopstick[j]);

        V(s);
    }
}
```



# Deadlock Prevention Techniques

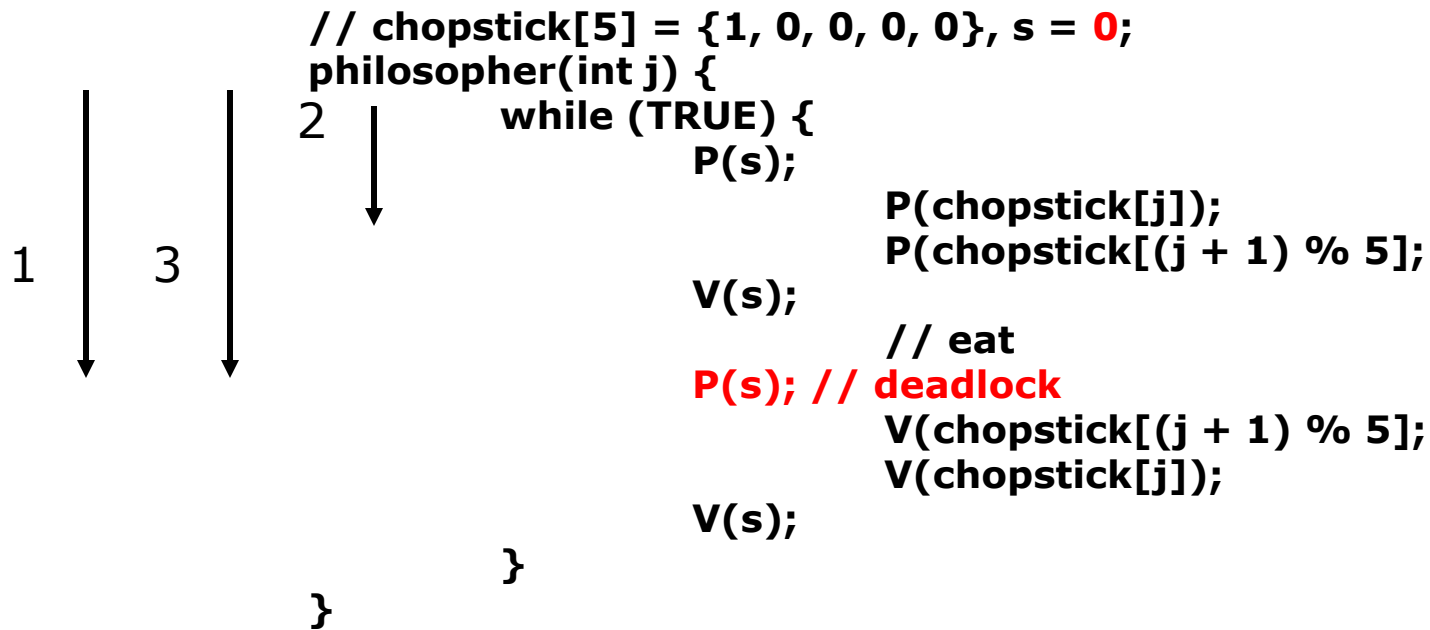
## 3. Adding more semaphores is also a technical skill



```
// chopstick[5] = {1, 0, 0, 0, 0}, s = 1 → 0;  
lawyer(int j) {  
    while (TRUE) {  
        P(s);  
  
        P(chopstick[j]); // deadlock  
        P(chopstick[(j + 1) % 5];  
  
        // eat  
  
        V(chopstick[(j + 1) % 5];  
        V(chopstick[j]);  
  
        V(s);  
    }  
}
```

# Deadlock Prevention Techniques

## 3. Adding more semaphores is also a technical skill



```
// chopstick[5] = {1, 0, 0, 0, 0}, s = 0;
philosopher(int j) {
    while (TRUE) {
        P(s);
        P(chopstick[j]);
        P(chopstick[(j + 1) % 5]);
        V(s);
        // eat
        P(s); // deadlock
        V(chopstick[(j + 1) % 5]);
        V(chopstick[j]);
        V(s);
    }
}
```

# Deadlock Prevention Techniques

---

3. Allocate all resources at the beginning (if you need 2 chopsticks, grab both at the same time)

```
int counter[5] = {1, 1, 1, 1, 1}; semaphore chopstick[5] = {1, 1, 1, 1, 1}, s = 1;
philosopher(int j) {
    while (TRUE) {
        P(s);
        // if both counters j and (j + 1) % 5 > 0, decrement counters
        // and grab chopstick[j] and chopstick[(j + 1) % 5]
        V(s);
        // if holding both chopsticks, eat
        P(s);
        // release chopsticks and increment counters as needed
        V(s);
    }
}
```

# Deadlock Prevention Techniques

3. Allocate all resources at the beginning (if you need 2 chopsticks, grab both at the same time)

```
// counter[5] = {1, 0, 0, 1, 1}; chopstick[5] = {1, 0, 0, 1, 1}, s = 1;
philosopher(int j) {
    while (TRUE) {
        P(s);
        // if both counters j and (j + 1) % 5 > 0, decrement counters
        // and grab chopstick[j] and chopstick[(j + 1) % 5]
        V(s);
        // if holding both chopsticks, eat
        P(s);
        // release chopsticks and increment counters as needed
        V(s);
    }
}
```

1  
↓  
(1, 2)

# Deadlock Prevention Techniques

3. Allocate all resources at the beginning (if you need 2 chopsticks, grab both at the same time)

```
// counter[5] = {1, 0, 0, 1, 1}; chopstick[5] = {1, 0, 0, 1, 1}, s = 1;
philosopher(int j) {
    while (TRUE) {
        P(s);
        // if both counters j and (j + 1) % 5 > 0, decrement counters
        // and grab chopstick[j] and chopstick[(j + 1) % 5]
        V(s);
        // if holding both chopsticks, eat
        P(s);
        // release chopsticks and increment counters as needed
        V(s);
    }
}
```

1  
↓  
(1, 2)  
2  
↓  
( )  
}

# Deadlock Prevention Techniques

3. Allocate all resources at the beginning (if you need 2 chopsticks, grab both at the same time)

```
// counter[5] = {1, 0, 0, 0, 0}; chopstick[5] = {1, 0, 0, 0, 0}, s = 1;
philosopher(int j) {
    while (TRUE) {
        P(s);
        // if both counters j and (j + 1) % 5 > 0, decrement counters
        // and grab chopstick[j] and chopstick[(j + 1) % 5]
        V(s);
        // if holding both chopsticks, eat
        P(s);
        // release chopsticks and increment counters as needed
        V(s);
    }
}
```

1      3  
↓      ↓  
(1, 2)(3, 4)

# Deadlock Prevention Techniques

---

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {1, 1, 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

# Deadlock Prevention Techniques

---

In the code, when philosophers pick up chopsticks, they don't simply follow the order of left hand or right hand.

Instead, they pick up the chopsticks according to  $\min(j, (j + 1) \% 5)$  and  $\max(j, (j + 1) \% 5)$ . This means they always pick up the chopstick with the smaller number first, and then the one with the larger number.

```
// chopstick[5] = {1, 1, 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```



# Deadlock Prevention Techniques

---

4. Make everyone use the same ordering in accessing resource (All threads must call P(x) before P(y))

```
// chopstick[5] = {1 → 0, 1, 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

0  
→

# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 1 → 0, 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

0 1  
→→

# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 1 → 0, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

0 1 2  
→→→

# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 1 → 0, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

0 1 2 3  
→→→→

# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 0, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

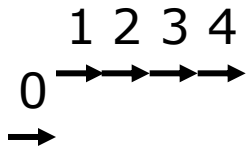
0 1 2 3 4  
→→→→→

# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 0, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

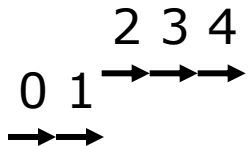


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 0, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

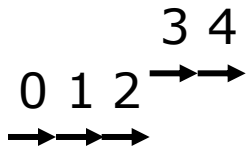


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 0, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```



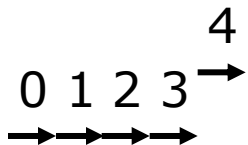


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 0, 1 → 0};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

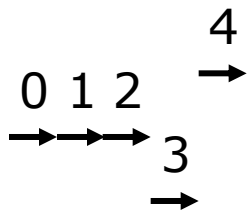


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 0, 0};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

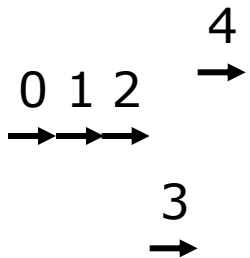


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 0, 0 → 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

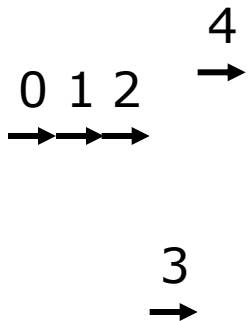


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 0 → 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

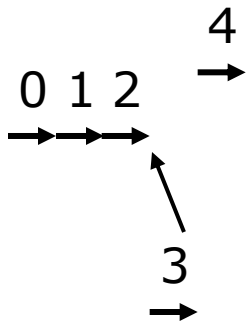


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 1 → 0, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

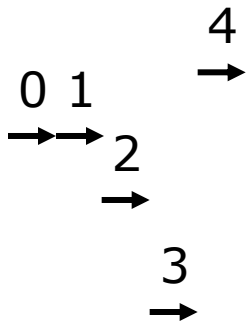


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 0, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

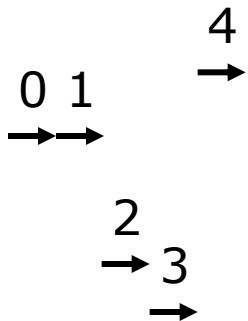


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 0 → 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

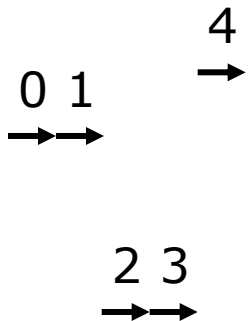


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0 → 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```



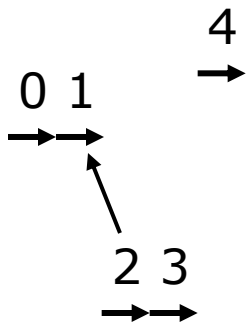


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 1 → 0, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

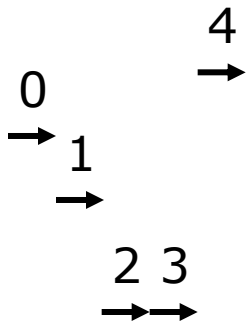


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

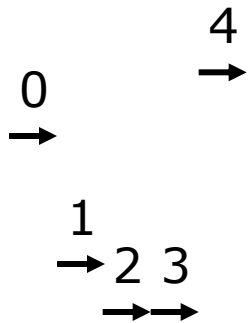


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 0 → 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

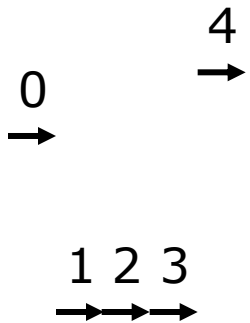


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0 → 1, 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

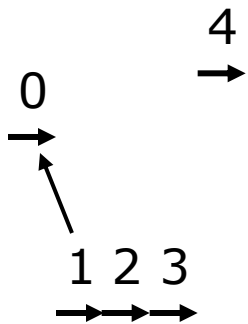


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 1 → 0, 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

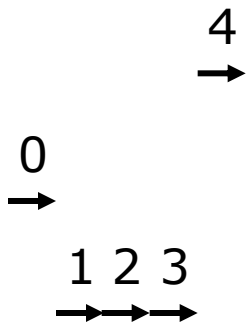


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0, 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```



# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 0 → 1, 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

4  
→

0  
→ 1 2 3  
→→→

# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0 → 1, 1, 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

4  
→

0 1 2 3  
→→→→

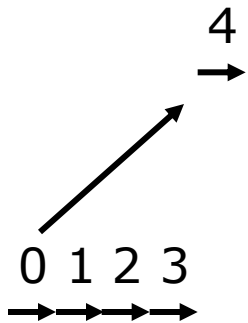


# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {1 → 0, 1, 1, 1, 1};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```



# Deadlock Prevention Techniques

4. Make everyone use the same ordering in accessing resource (All threads must call  $P(x)$  before  $P(y)$ )

```
// chopstick[5] = {0, 1, 1, 1, 1 → 0};
```

```
philosopher(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

4  
→

0 1 2 3  
→→→→

# More Deadlock Prevention Methods

---

## 5. No waiting (phone company)

- In a phone system, if a line is busy, the user typically hears a busy tone and has to call back later instead of waiting for the line to become free.
- Similarly, in the Dining Philosophers Problem,
  - If a philosopher can't pick up both chopsticks immediately, they would put down whatever they have and wait for a better time to try again.
  - This ensures that no philosopher is holding on to a chopstick while waiting for another, which could lead to deadlock.



## More Deadlock Prevention Methods

---

6. Preempt resources (copying memory content to disk)
7. Banker's algorithm
8. A combination of techniques



# Banker's Algorithm

---

- The idea of Banker's algorithm:
  - Allows the sum of requested resources  $>$  total resources
  - As long as, there is some way for all threads to finish without getting into any deadlocks



# Banker's Algorithm

---

- ***Banker's algorithm:***

- A thread states its maximum resource needs in advance
- The OS allocates resource dynamically as needed. A thread waits if granting its request would lead to deadlocks
- A request can be granted if some sequential ordering of threads is deadlock free

# Example 1

---

- Total RAM 256 MB

	Allocated	Still needed
$P_1$	80 MB	30 MB
$P_2$	10 MB	10 MB
$P_3$	120 MB	80 MB



# Example 1

- Total RAM 256 MB

	Allocated	Still needed
$P_1$	80 MB	30 MB
$P_2$	20 MB	0 MB
$P_3$	120 MB	80 MB

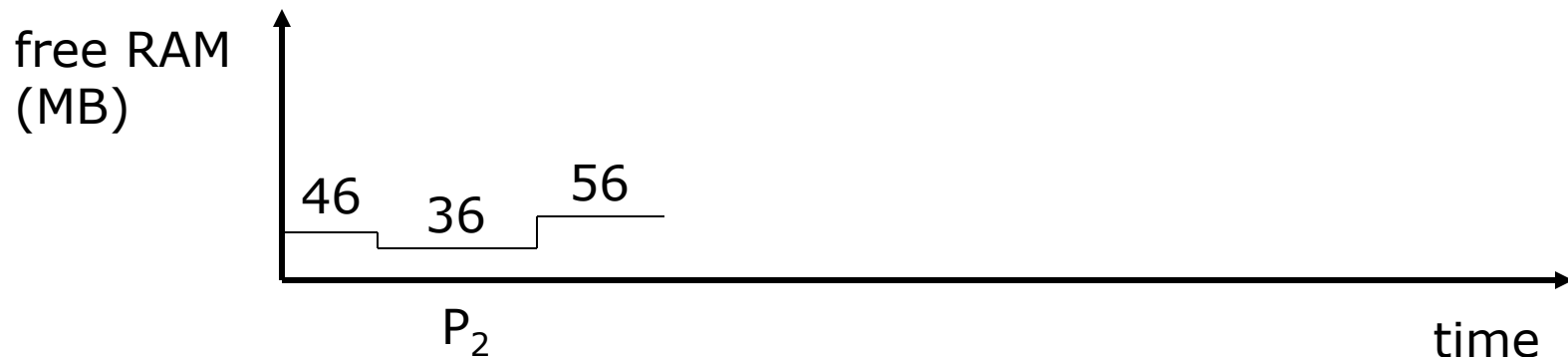




# Example 1

- Total RAM 256 MB

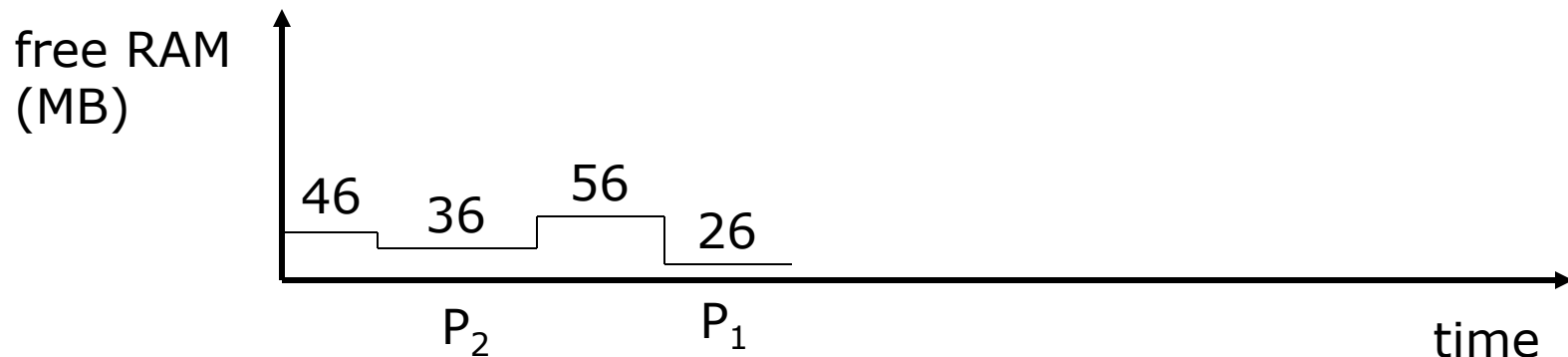
	Allocated	Still needed
$P_1$	80 MB	30 MB
$P_2$	0 MB	0 MB
$P_3$	120 MB	80 MB



# Example 1

- Total RAM 256 MB

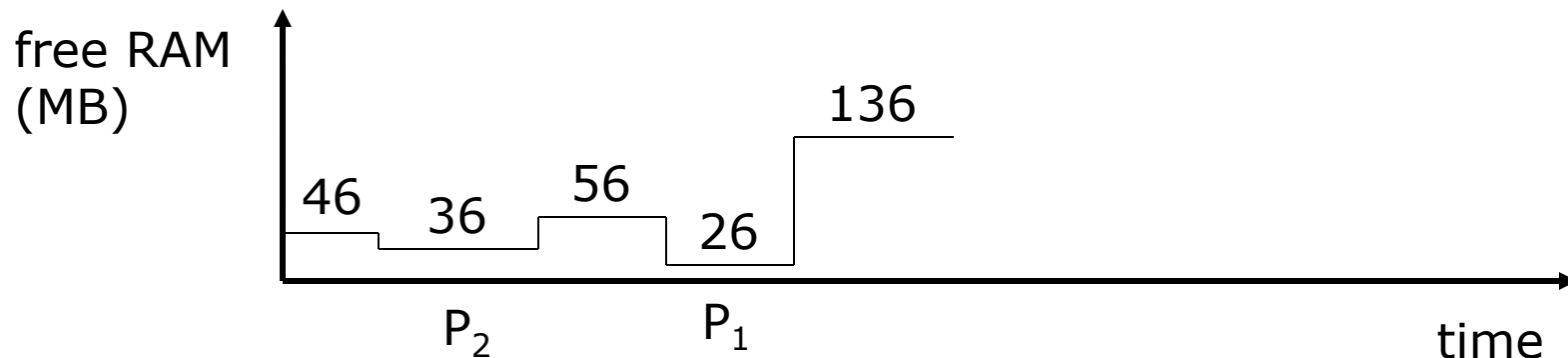
	Allocated	Still needed
$P_1$	110 MB	0 MB
$P_2$	0 MB	0 MB
$P_3$	120 MB	80 MB



# Example 1

- Total RAM 256 MB

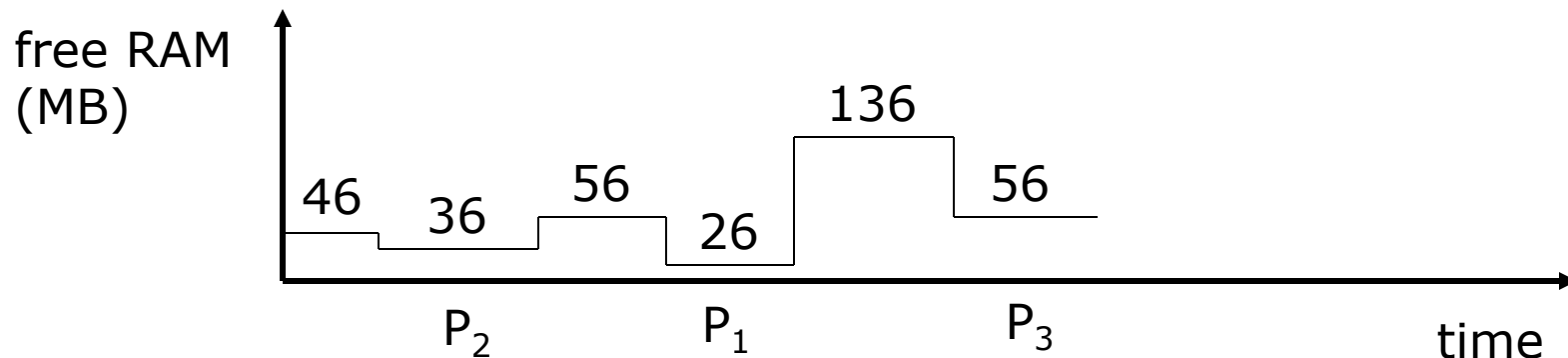
	Allocated	Still needed
$P_1$	0 MB	0 MB
$P_2$	0 MB	0 MB
$P_3$	120 MB	80 MB



# Example 1

- Total RAM 256 MB

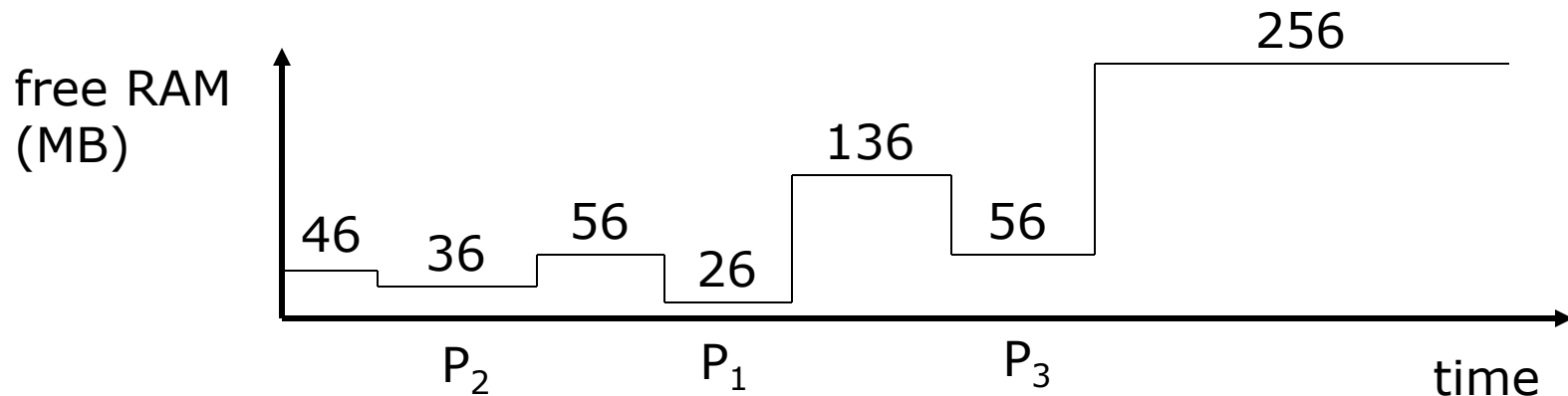
	Allocated	Still needed
$P_1$	0 MB	0 MB
$P_2$	0 MB	0 MB
$P_3$	200 MB	0 MB



# Example 1

- Total RAM 256 MB

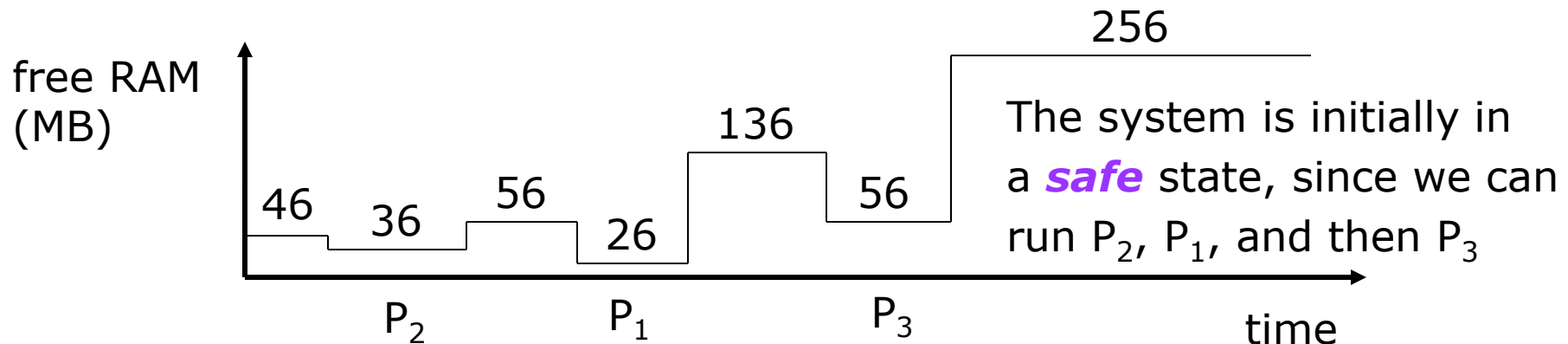
	Allocated	Still needed
$P_1$	0 MB	0 MB
$P_2$	0 MB	0 MB
$P_3$	0 MB	0 MB



# Example 1

- Total RAM 256 MB

	Allocated	Still needed
$P_1$	80 MB	30 MB
$P_2$	10 MB	10 MB
$P_3$	120 MB	80 MB



## Example 2

- Total RAM 256 MB

	Allocated	Still needed
$P_1$	80 MB	60 MB
$P_2$	10 MB	10 MB
$P_3$	120 MB	80 MB



## Example 2

- Total RAM 256 MB

	Allocated	Still needed
$P_1$	80 MB	60 MB
$P_2$	20 MB	0 MB
$P_3$	120 MB	80 MB

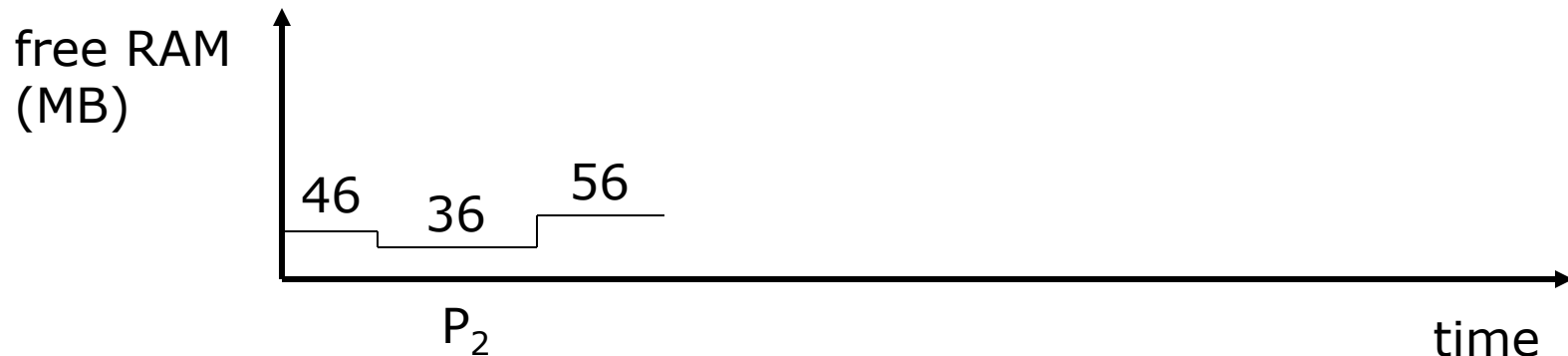




## Example 2

- Total RAM 256 MB

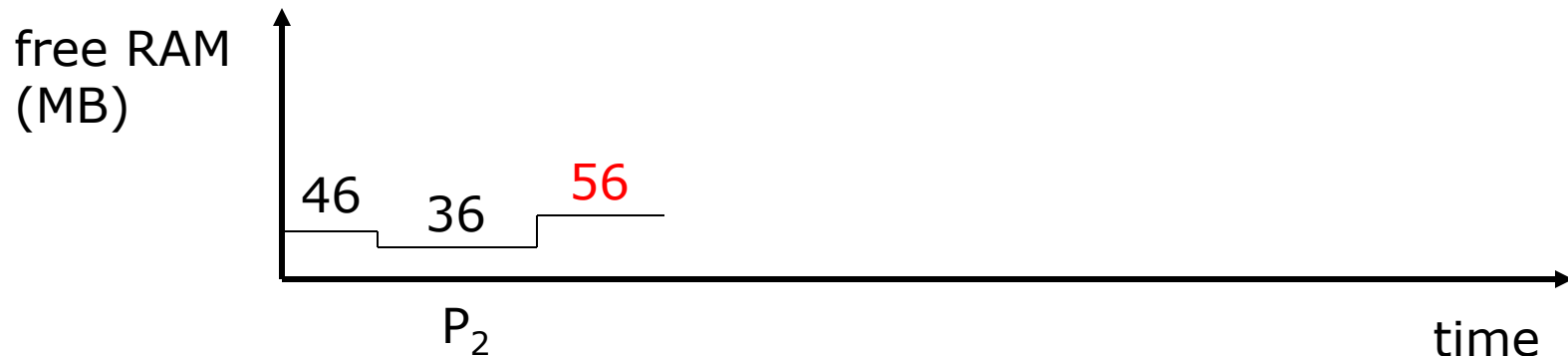
	Allocated	Still needed
$P_1$	80 MB	60 MB
$P_2$	0 MB	0 MB
$P_3$	120 MB	80 MB



## Example 2

- Total RAM 256 MB

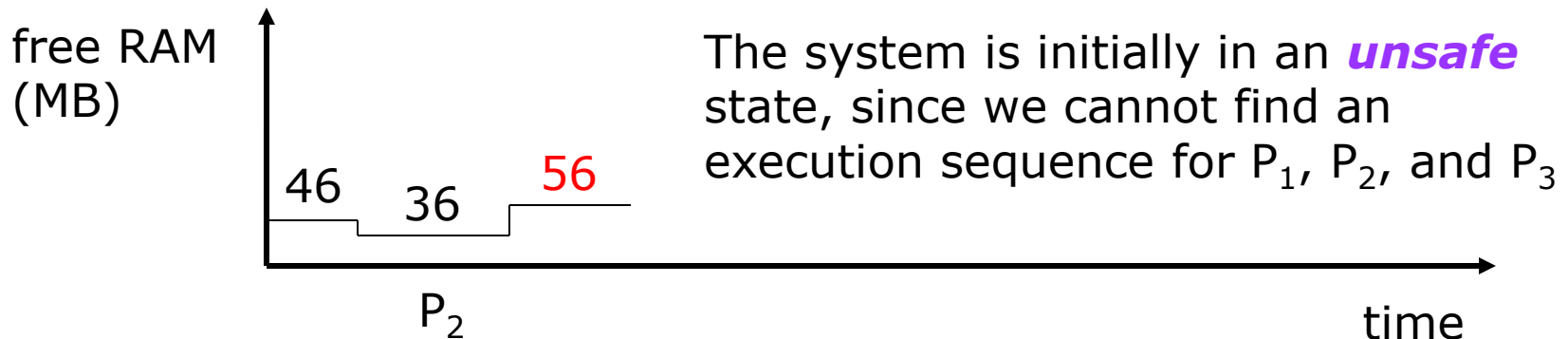
	Allocated	Still needed
$P_1$	80 MB	60 MB
$P_2$	0 MB	0 MB
$P_3$	120 MB	80 MB



## Example 2

- Total RAM 256 MB

	Allocated	Still needed
$P_1$	80 MB	60 MB
$P_2$	10 MB	10 MB
$P_3$	120 MB	80 MB





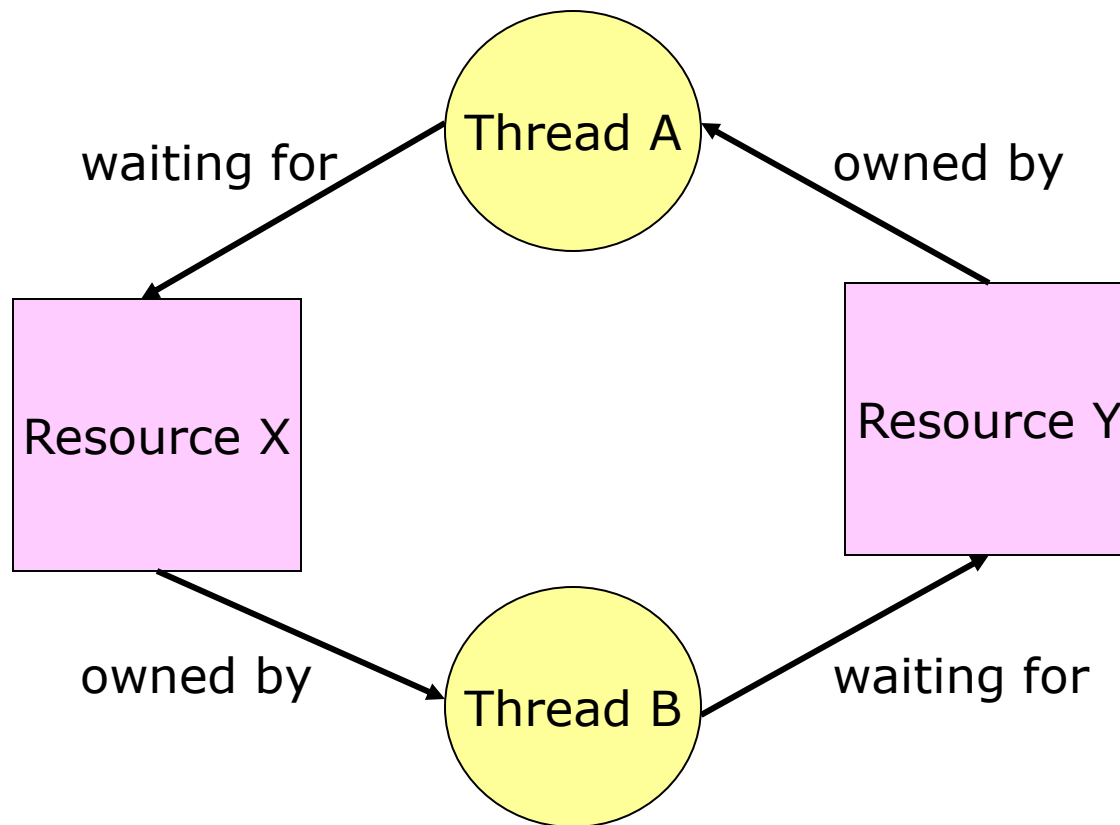
# Deadlock Detection and Recovery

---

- Scan the resource allocation graph
- Detect circular chains of requests
- Recover from the deadlock

# Resource Allocation Graph

---



# Once A Cycle is Detected...

---

- Some possible actions
  - Kill a thread and force it to give up resources
    - Remaining system may be in an inconsistent state
  - Rollback actions of a deadlocked thread
    - Not always possible (a file maybe half-way through modifications)
    - Need **checkpointing**, or taking snapshots of system states from time to time