

# Deadlocks

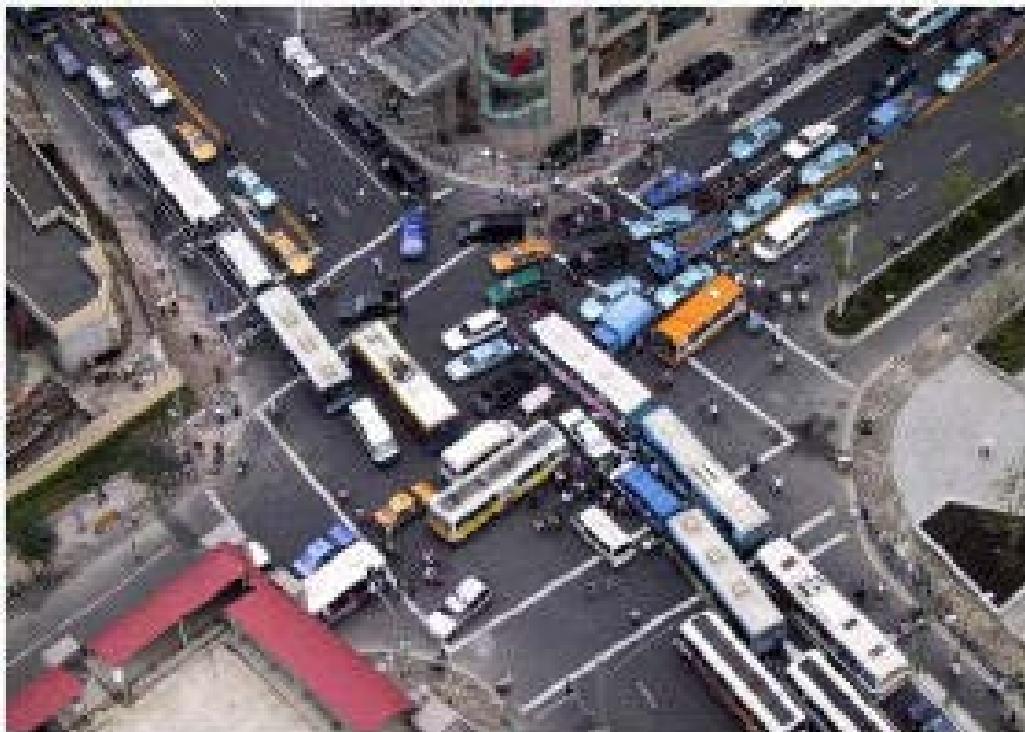
---

Sarah Diesburg  
Operating Systems  
CS 3430



# Example 1

---



# Deadlocks

---

- **Deadlocks**: Occur when threads are waiting for resources with circular dependencies
  - Often involve **nonpreemptable resources**, which cannot be taken away from its current thread without failing the computation (e.g., storage allocated to a file)

# Deadlocks

---

- Deadlocks that involve ***preemptable resources*** (e.g., CPU) can usually be resolved by reallocation of resources
- ***Starvation***: a thread waits indefinitely
- A deadlock implies starvation

# An Example of Deadlocks

---

Thread A

P(x);  
P(y);

Thread B

P(y);  
P(x);

- A deadlock won't always happen with this code, but it might

# 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...

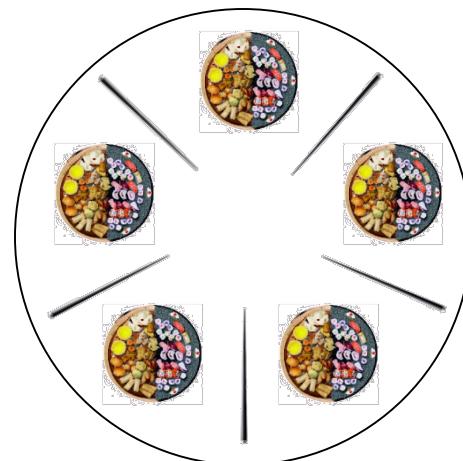
---

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

# A Classic Example of Deadlocks

---

- Dinning Philosophers
- Each needs two chopsticks to eat



# 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 Philosophers Implementation

---

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

# A Dining Philosophers Implementation

---

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

# A Dining Philosophers Implementation

---

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

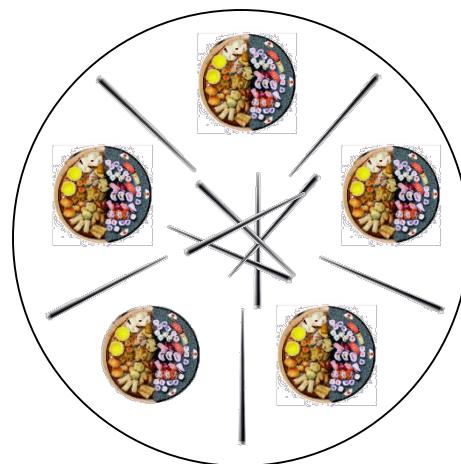
---

- When encountering a deadlock
  - All four conditions must be true
- To prevent deadlocks
  - Remove one of the four conditions

# Deadlock Prevention Techniques

---

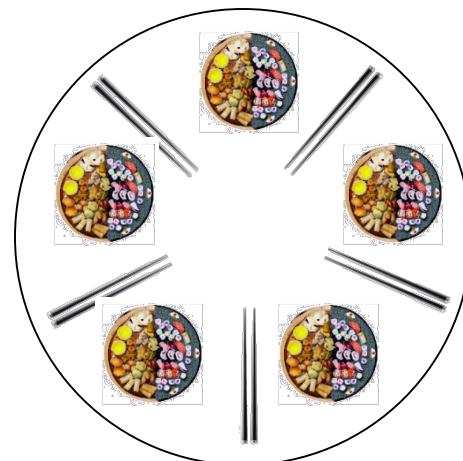
1. Infinite resources (buy a very large disk)



# 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;
person(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

---

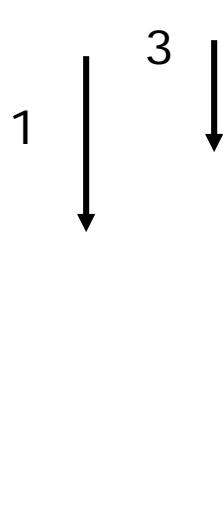
3. Allocate all resources at the beginning (if you need 2 chopsticks, grab both at the same time)

1  
↓

```
semaphore chopstick[5] = {1, 1 → 0, 1 → 0, 1, 1}, s = 1 → 0;  
person(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)



```
// chopstick[5] = {1, 0, 0, 1, 1}, s = 0;
person(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)

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

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

1

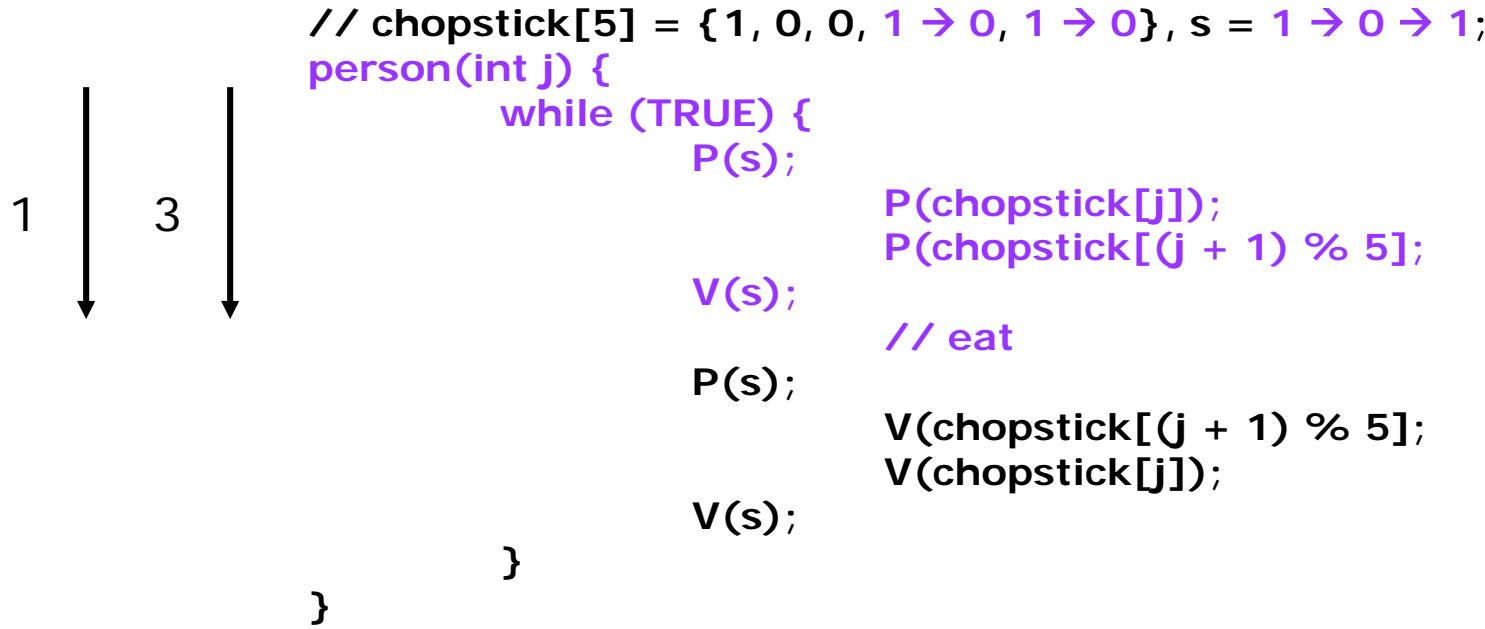


# Deadlock Prevention Techniques

---

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

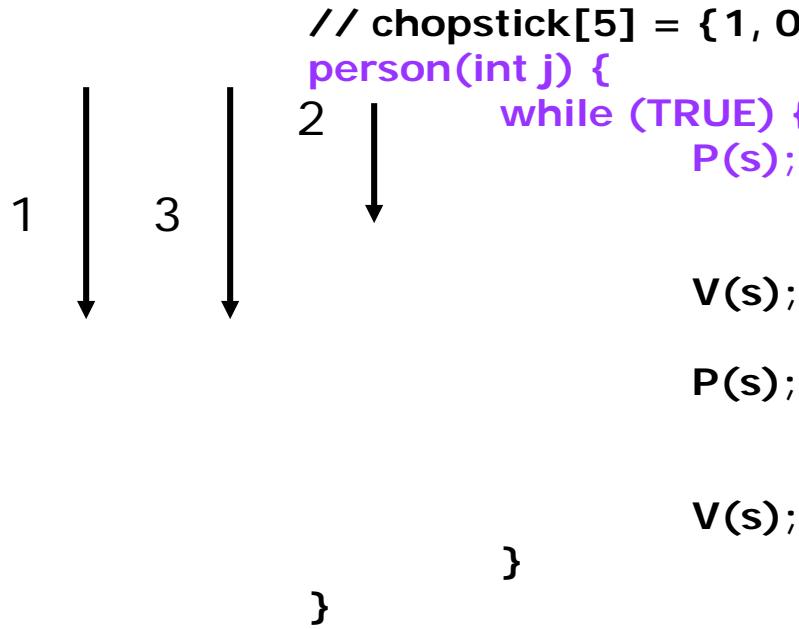
```
// chopstick[5] = {1, 0, 0, 1 → 0, 1 → 0}, s = 1 → 0 → 1;
person(int j) {
    while (TRUE) {
        P(s);
        P(chopstick[j]);
        P(chopstick[(j + 1) % 5]);
        // eat
        V(chopstick[(j + 1) % 5]);
        V(chopstick[j]);
        V(s);
    }
}
```



The diagram consists of two vertical black arrows. The first arrow originates from the number '1' located to the left of the pseudocode and points down to the line 'P(chopstick[j]);'. The second arrow originates from the number '3' located to the left of the pseudocode and points down to the line 'P(chopstick[(j + 1) % 5]);'.

# Deadlock Prevention Techniques

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

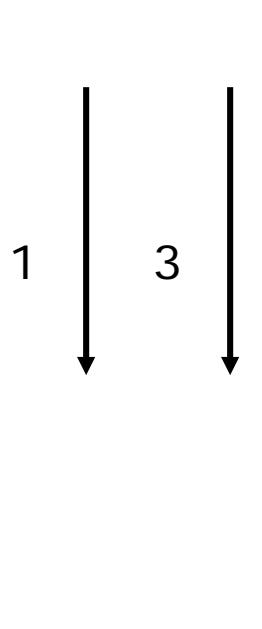


```
// chopstick[5] = {1, 0, 0, 0, 0}, s = 1 → 0;
person(int j) {
    while (TRUE) {
        P(s);
        P(chopstick[j]); // deadlock
        P(chopstick[(j + 1) % 5]);
        // eat
        V(chopstick[(j + 1) % 5]);
        V(chopstick[j]);
        V(s);
    }
}
```

The diagram consists of three vertical arrows pointing downwards from the numbers 1, 2, and 3 to the corresponding lines of code in the pseudocode. Arrow 1 points to the line 'P(s);'. Arrow 2 points to the line 'P(chopstick[j]);'. Arrow 3 points to the line 'V(s);'.

# Deadlock Prevention Techniques

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



```
// chopstick[5] = {1, 0, 0, 0, 0}, s = 0;
person(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);
    }
}
```

The diagram consists of three vertical arrows pointing downwards from the numbers 1, 2, and 3 to the corresponding lines of code in the pseudocode below. The first arrow (number 1) points to the line 'V(s);'. The second arrow (number 2) points to the line 'P(s);'. The third arrow (number 3) points to the line 'P(chopstick[j]);'.

# New Solution

---

- Don't lock around the chopsticks that are being released

# Deadlock Prevention Techniques

---

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

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

# Deadlock Prevention Techniques

---

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

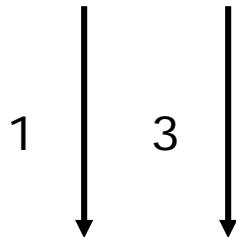
```
// chopstick[5] = {1, 1 → 0, 1 → 0, 1, 1}, s = 1 → 0 → 1;
person(int j) {
    while (TRUE) {
        P(s);
        P(chopstick[j]);
        P(chopstick[(j + 1) % 5]);
        V(s);
        // eat
        V(chopstick[(j + 1) % 5]);
        V(chopstick[j]);
    }
}
```

# Deadlock Prevention Techniques

---

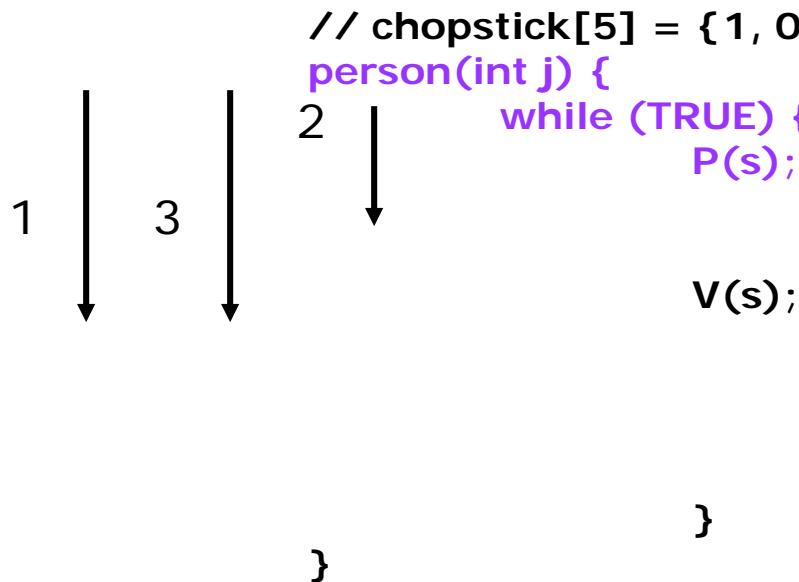
3. Allocate all resources at the beginning (if you need 2 chopsticks, grab both at the same time)

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



# Deadlock Prevention Techniques

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



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

The diagram consists of three vertical arrows pointing downwards from the numbers 1, 2, and 3 to the corresponding lines of code in the pseudocode below. The first arrow (1) points to the line 'P(s);'. The second arrow (2) points to the line 'P(chopstick[j]); // wait'. The third arrow (3) points to the line 'V(chopstick[j]);'.

# Deadlock Prevention Techniques

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

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

# Deadlock Prevention Techniques

---

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

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

# Deadlock Prevention Techniques

---

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

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



## More Elaborate Solution

---

- Allocate some counters
- If someone can't pick up both chopsticks after taking the lock, release the lock

# 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;
person(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;
person(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;
person (int j) {
    ↓
    1
    while (TRUE) {
        ↓
        2
        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);
    }
    ↓
    0
}
```

# 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;
person(int j) {
    ↓   ↓
    1   3
    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

---

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};  
  
person(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))

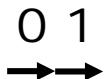
```
// chopstick[5] = {1 → 0, 1, 1, 1, 1};  
  
person(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```

# 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};  
  
person(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};  
  
person(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};  
  
person(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};  
  
person(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)]);  
    }  
}
```



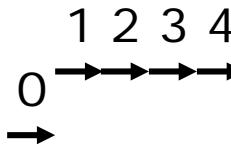
# Deadlock Prevention Techniques

---

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};

person(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)]);
    }
}
```



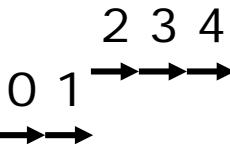
The diagram shows a circular arrangement of five chopsticks, indexed from 0 to 4. An arrow starts at index 0 and points sequentially to indices 1, 2, 3, and 4, indicating a specific ordering for access.

# 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};  
  
person(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)]);  
    }  
}
```

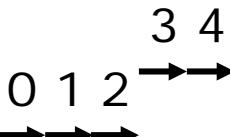


# Deadlock Prevention Techniques

---

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};  
  
person(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)]);  
    }  
}
```

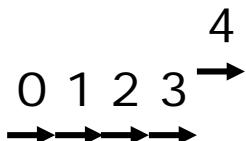


# Deadlock Prevention Techniques

---

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};  
  
person(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)]);  
    }  
}
```

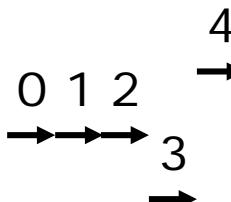


# Deadlock Prevention Techniques

---

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};  
  
person(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};  
  
person(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)]);  
    }  
}
```



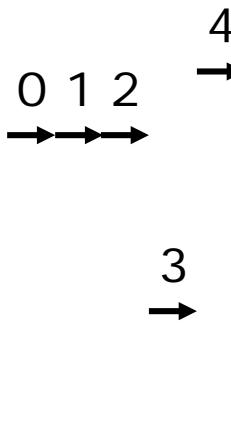
The diagram shows five chopsticks arranged horizontally. Chopstick 0 is held by thread 1, indicated by a double-headed arrow between threads 0 and 1. Chopstick 4 is held by thread 4, indicated by a double-headed arrow between threads 4 and 5.

# 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};  
  
person(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)]);  
    }  
}
```

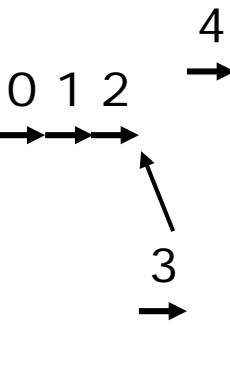


# Deadlock Prevention Techniques

---

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};  
  
person(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)]);  
    }  
}
```



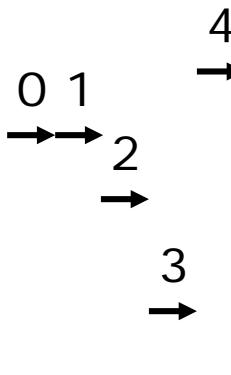
# Deadlock Prevention Techniques

---

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};

person(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)]);
    }
}
```

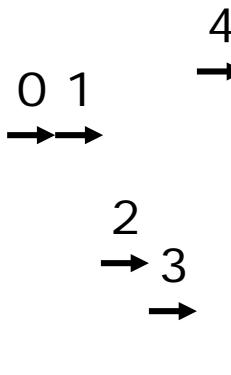


# Deadlock Prevention Techniques

---

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};  
  
person(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)]);  
    }  
}
```



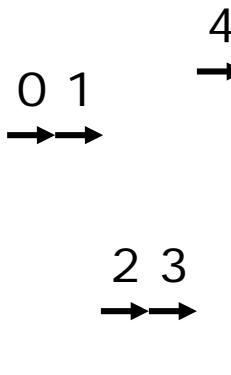
The diagram illustrates the state of five chopsticks (chopstick[5]) represented as an array [0, 0, 0, 0 → 1, 1]. The indices 0 through 4 correspond to the chopsticks. Thread 1 holds chopstick 0. Thread 2 holds chopstick 1. Thread 3 holds chopstick 2. Thread 4 holds chopstick 3. Thread 5 holds chopstick 4. The arrow symbol (→) indicates that thread 4 has acquired chopstick 3 before thread 5 acquired chopstick 4, thus maintaining the required ordering.

# 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};  
  
person(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)]);  
    }  
}
```

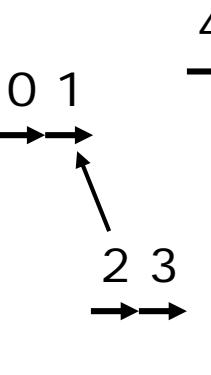


The diagram illustrates the state of five chopsticks for two threads. Thread 0 (represented by a black arrow) holds chopsticks 0 and 1. Thread 1 (represented by a grey arrow) holds chopsticks 2 and 3. An additional arrow labeled '4' points to the position of chopstick 4, which is currently unheld.

# 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};  
  
person(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)]);  
    }  
}
```



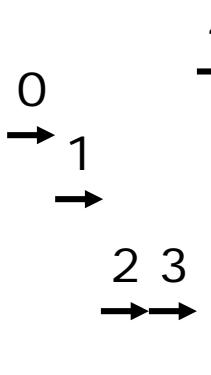
# Deadlock Prevention Techniques

---

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};

person(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)]);
    }
}
```



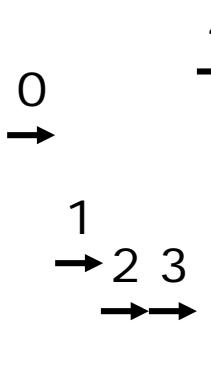
The diagram illustrates the state of five chopsticks (chopstick[5]) represented as an array [0, 0, 0, 1, 1]. Thread 1 holds chopstick 0. Thread 4 holds chopsticks 1, 2, and 3. Thread 4 is shown with two arrows pointing to chopsticks 1 and 3, indicating it has acquired both.

# 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};  
  
person(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)]);  
    }  
}
```

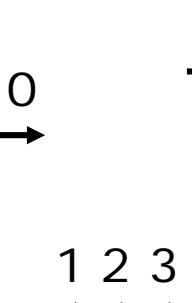


# Deadlock Prevention Techniques

---

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};  
  
person(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)]);  
    }  
}
```

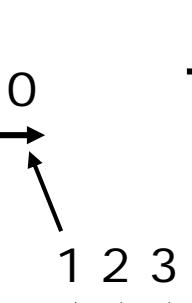


# Deadlock Prevention Techniques

---

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

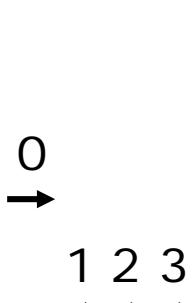
```
// chopstick[5] = {0, 1 → 0, 1, 1, 1};  
  
person(int j) {  
    while (TRUE) {  
        P(chopstick[min(j, (j + 1) % 5)]);  
        P(chopstick[max(j, (j + 1) % 5)]);  
        // eat  
        V(chopstick[max(j, (j + 1) % 5)]);  
        V(chopstick[min(j, (j + 1) % 5)]);  
    }  
}
```



# 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};  
  
person(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)]);  
    }  
}
```

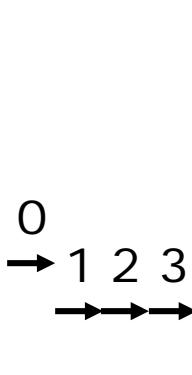


# Deadlock Prevention Techniques

---

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};  
  
person(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)]);  
    }  
}
```

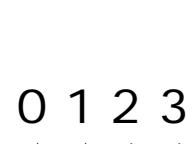


# Deadlock Prevention Techniques

---

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, 1};  
  
person(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)]);  
    }  
}
```



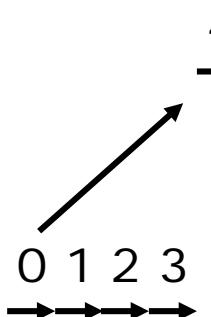
The diagram shows a sequence of five positions labeled 0, 1, 2, 3, and 4. Arrows point from position 0 to 1, from 1 to 2, from 2 to 3, and from 3 to 4, indicating a circular ordering of resources.

# 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};  
  
person(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)]);  
    }  
}
```

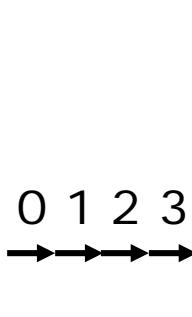


# Deadlock Prevention Techniques

---

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};  
  
person(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)]);  
    }  
}
```





## More Deadlock Prevention Methods

---

5. Banker's algorithm
6. 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

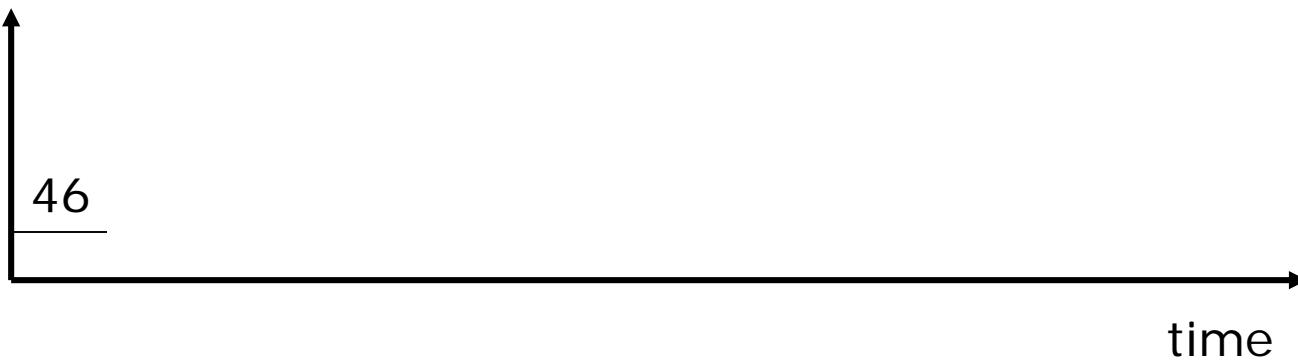
# Example 1

---

- 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

free RAM  
(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

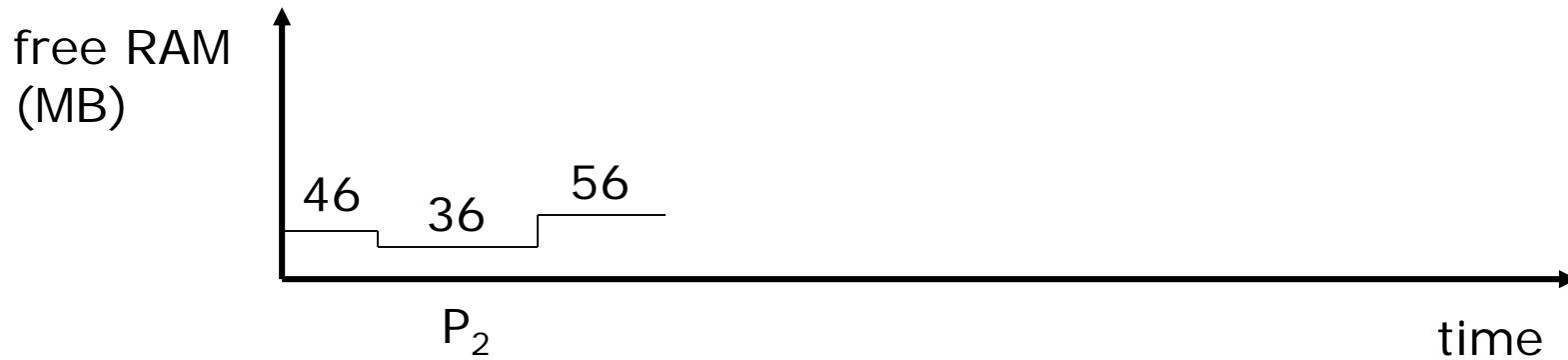


# Example 1

---

- 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

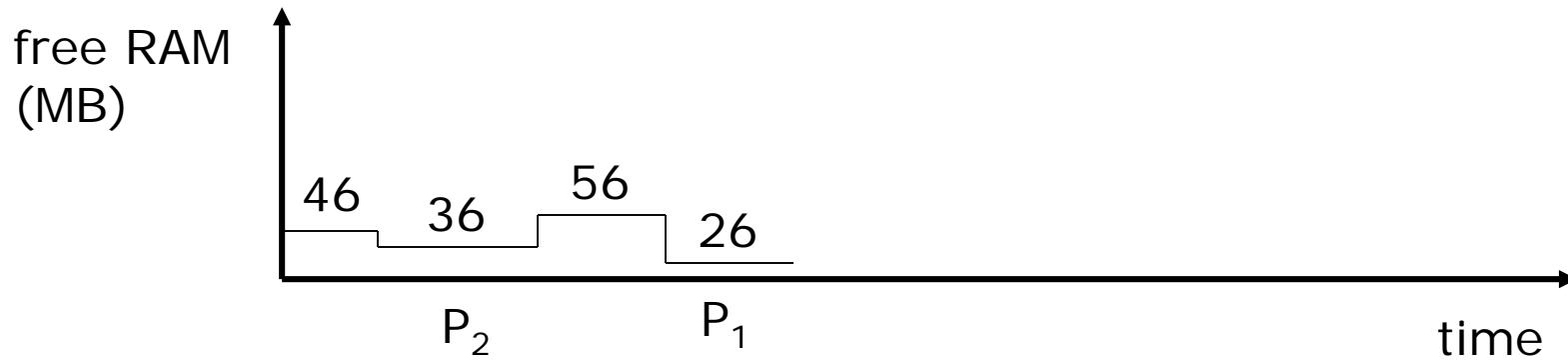


# Example 1

---

- 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

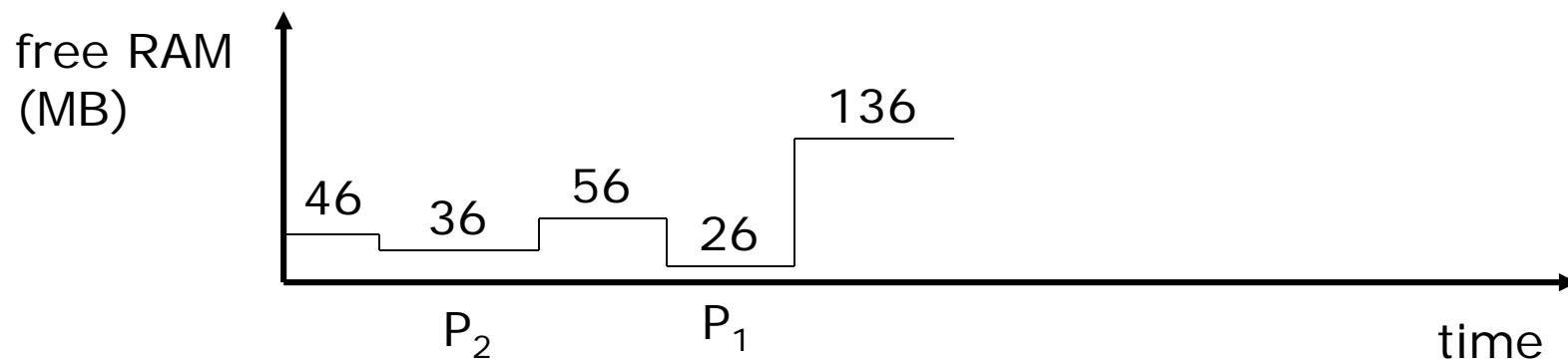


# Example 1

---

- 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

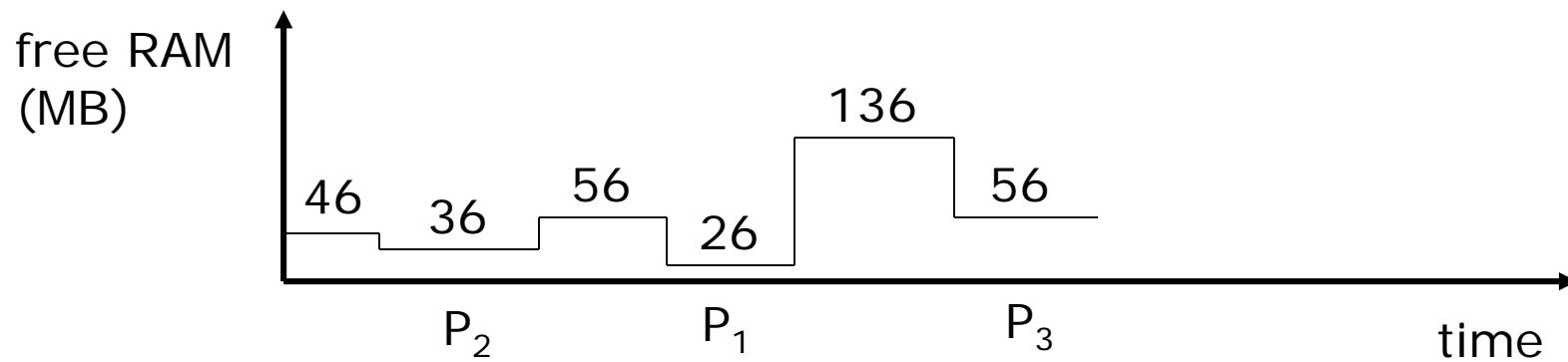


# Example 1

---

- 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

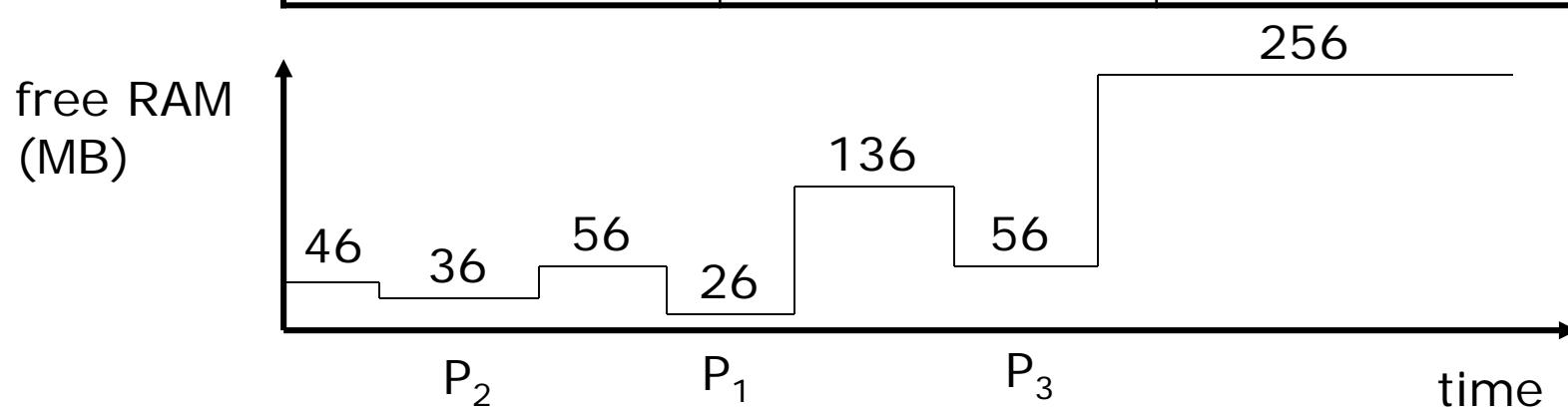


# Example 1

---

- Total RAM 256 MB

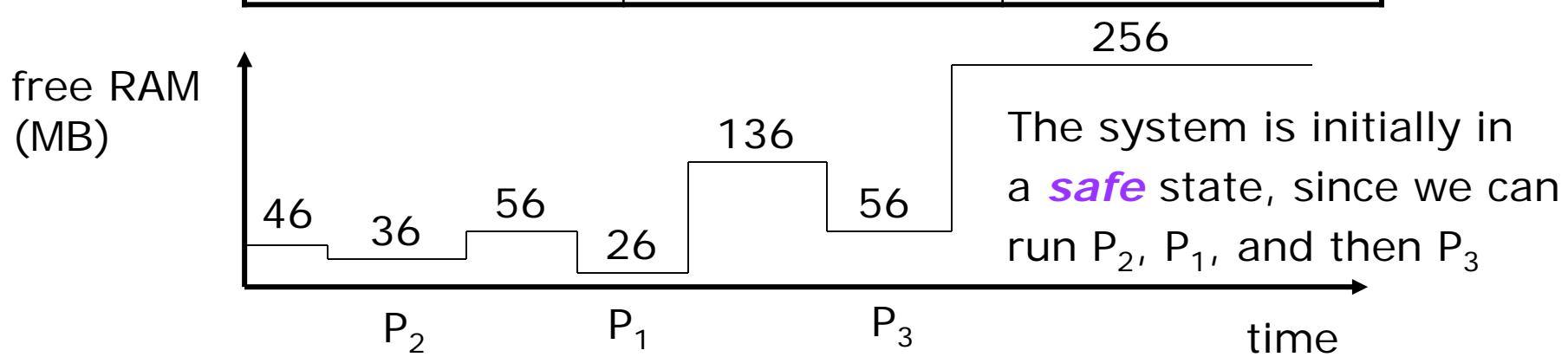
	Allocated	Still needed
$P_1$	0 MB	0 MB
$P_2$	0 MB	0 MB
$P_3$	0 MB	0 MB



# 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



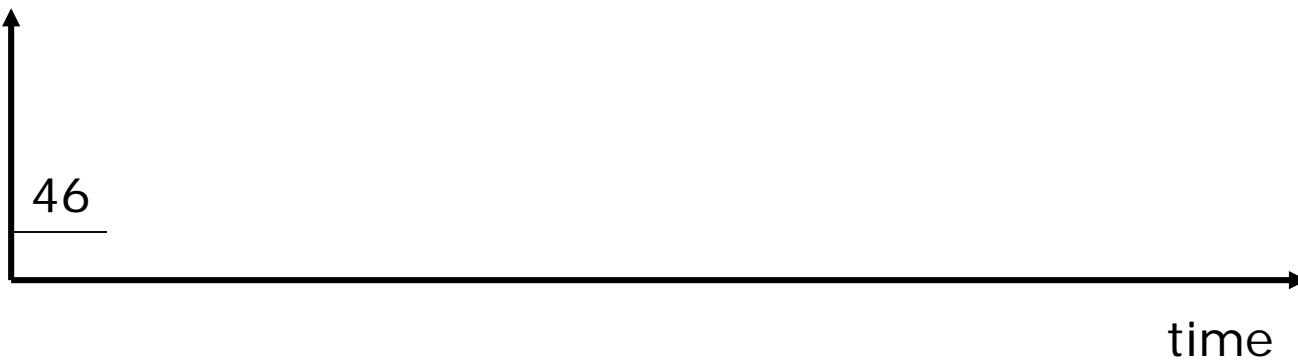
## Example 2

---

- 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

free RAM  
(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

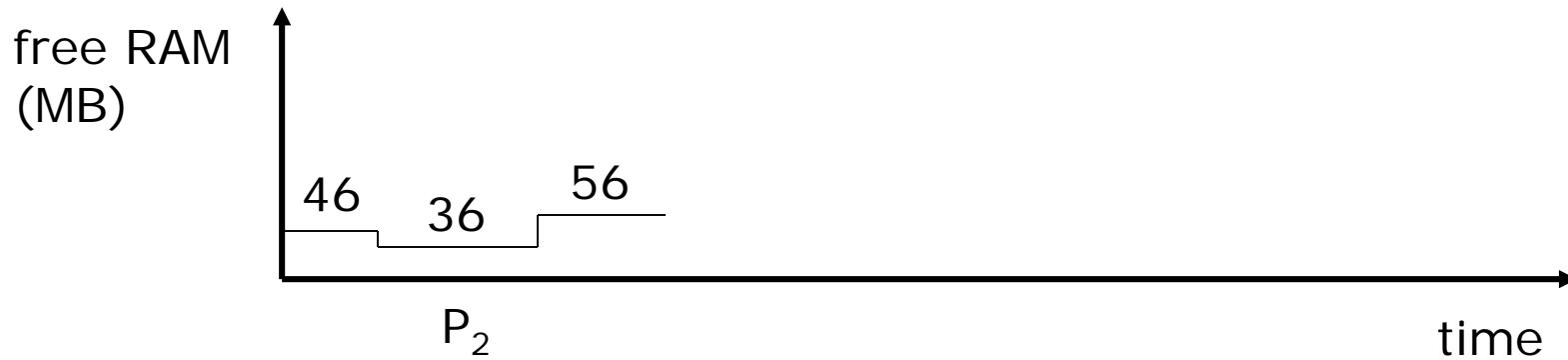


## Example 2

---

- 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

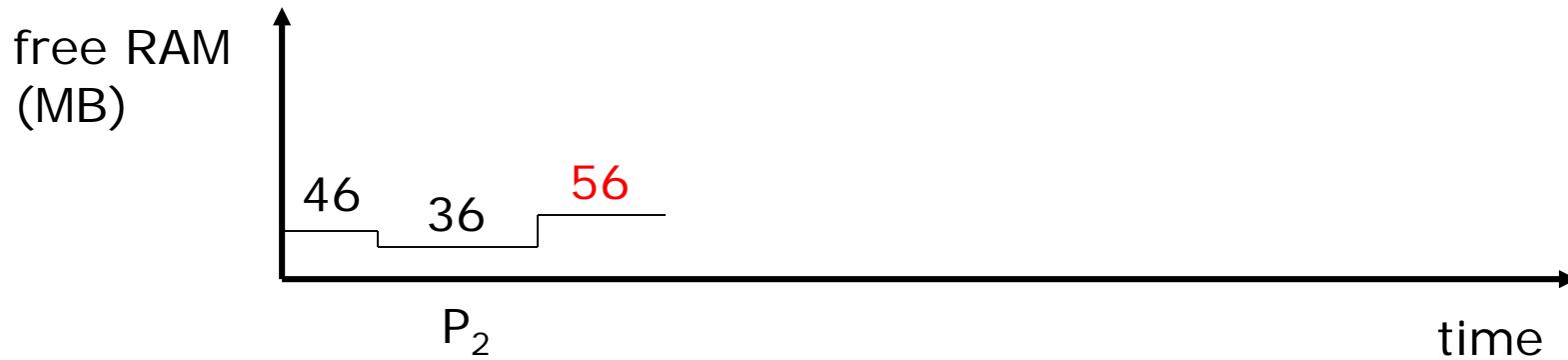


## Example 2

---

- 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

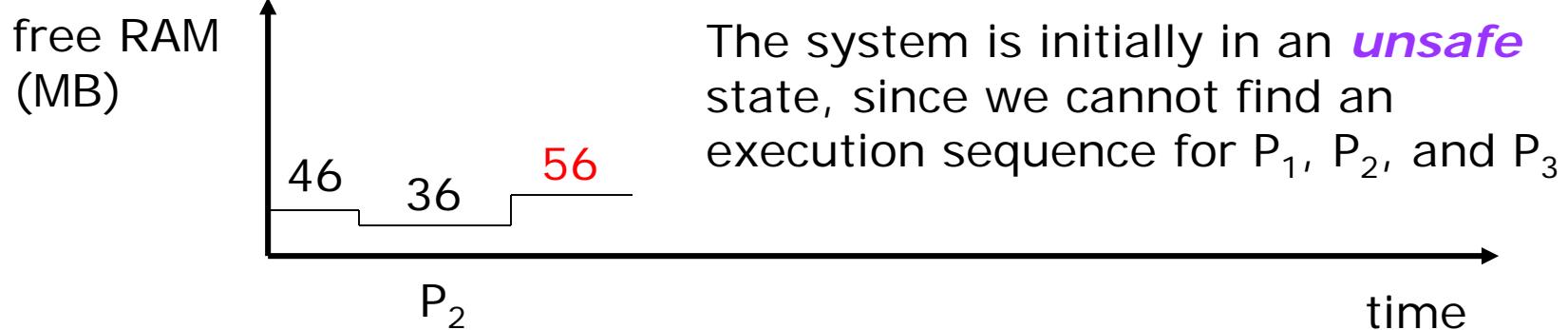


## Example 2

---

- Total RAM 256 MB

	Allocated	Still needed
P <sub>1</sub>	80 MB	60 MB
P <sub>2</sub>	10 MB	10 MB
P <sub>3</sub>	120 MB	80 MB



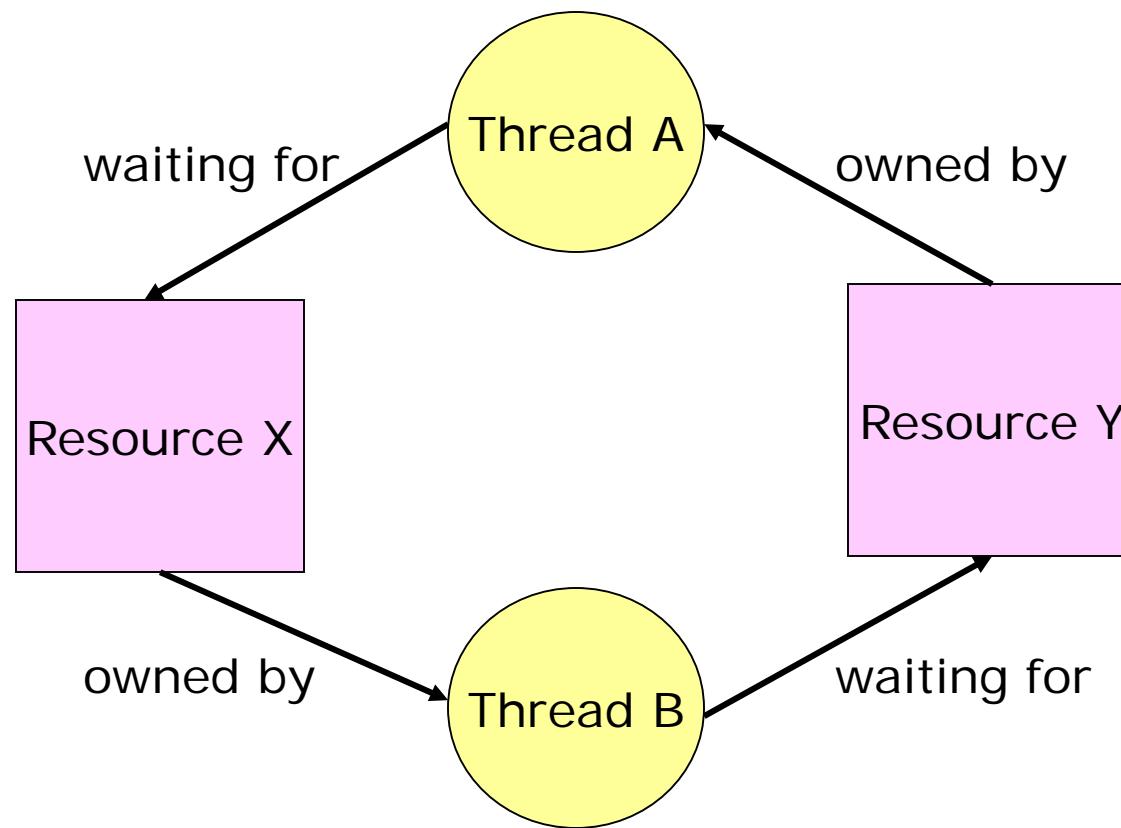
# Deadlock Detection and Recovery

---

- 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