# Synchronization

Chester Rebeiro
IIT Madras

# Motivating Scenario

shared variable

```
int counter=5;
```

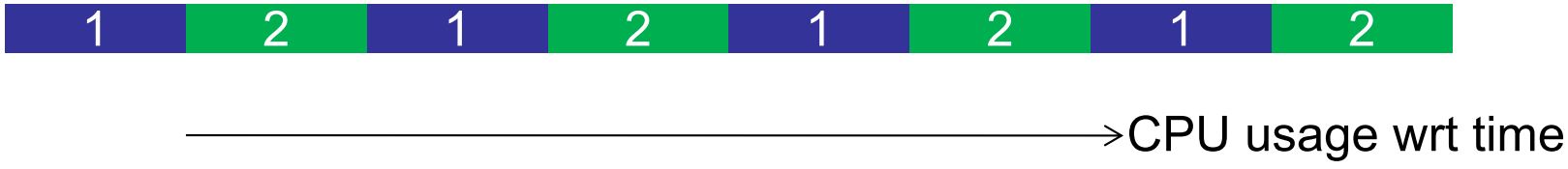
program 0

```
{
   *
   *
counter++
   *
}
```

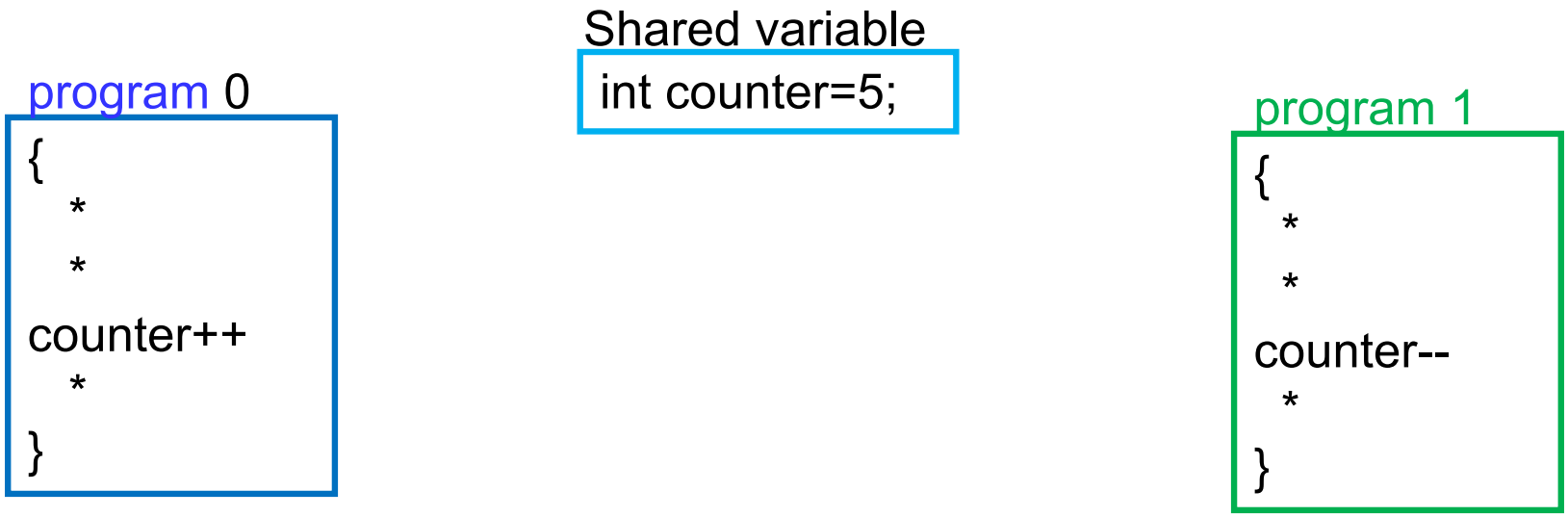
program 1

```
{
  *
  *
counter--
  *
}
```

- Single core
  - Program 1 and program 2 are executing at the same time but sharing a single core

# Motivating Scenario




- What is the value of counter?
  - expected to be 5
  - but could also be 4 and 6

# Motivating Scenario

Shared variable

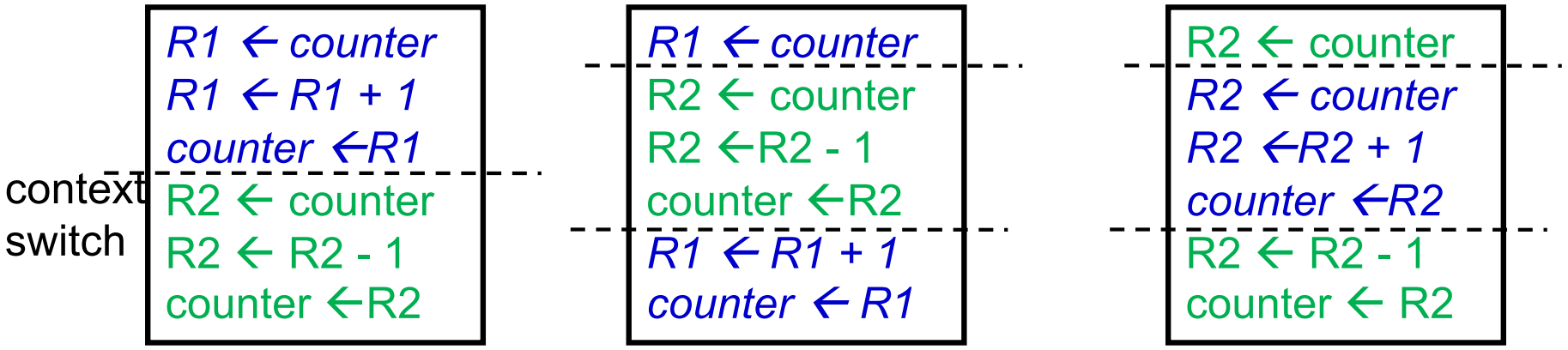
```
int counter=5;
```

*program 0*

```
{
   *
   *
counter++
   *
}
```

program 1

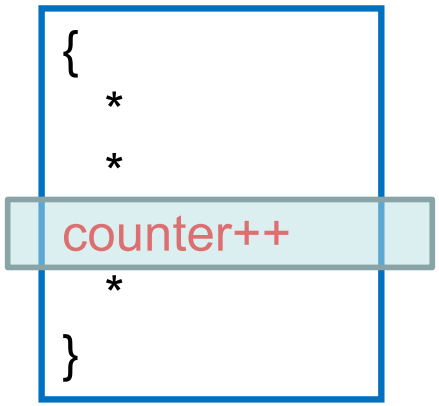
```
{
  *
  *
counter--
  *
}
```




counter = 5 counter = 6 counter = 4

# Race Conditions

- Race conditions
  - A situation where several processes access and manipulate the same data (*critical section*)
  - The outcome depends on the order in which the access take place
  - Prevent race conditions by synchronization
    - Ensure only one process at a time manipulates the critical data




critical section

*No more than one process should execute in critical section at a time*

# Race Conditions in Multicore

shared variable

```
int counter=5;
```

program 0

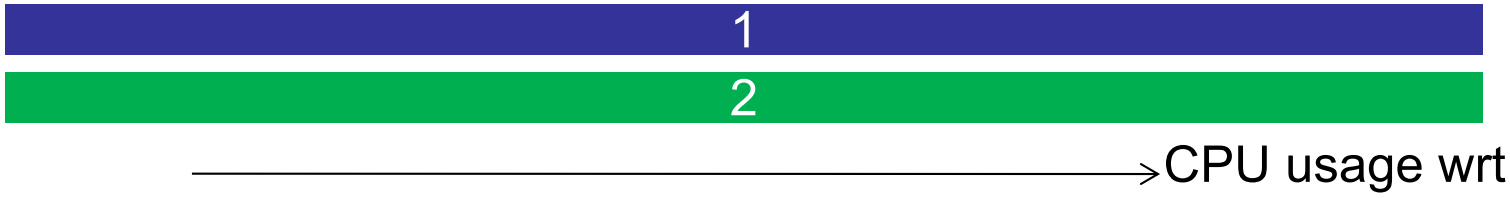
```
{
   *
   *
counter++
   *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

- Multi core
  - Program 1 and program 2 are executing at the same time on different cores





# Critical Section

- Any solution should satisfy the following requirements
  - **Mutual Exclusion :** No more than one process in critical section at a given time
  - **Progress :** When no process is in the critical section, any process that requests entry into the critical section must be permitted without any delay
  - **No starvation (bounded wait):** There is an upper bound on the number of times a process enters the critical section, while another is waiting.

# Locks and Unlocks

shared variable

```
int counter=5;
lock_t L;
```

program 0

```
{
  *
  *
lock(L)
counter++
unlock(L)
  *
}
```

program 1

```
{
  *
  *
lock(L)
counter--
unlock(L)
  *
}
```

- lock(L) : acquire lock L exclusively
  - Only the process with L can access the critical section
- unlock(L) : release exclusive access to lock L
  - Permitting other processes to access the critical section

# When to have Locking?

- Single instructions by themselves are atomic

  eg. add %eax, %ebx

- Multiple instructions need to be explicitly made atomic
  - Each piece of code in the OS must be checked if they need to be atomic

# How to Implement Locking (Software Solutions)

Chester Rebeiro
IIT Madras

# Using Interrupts

Process 1

```
while(1){
lock ------> disable interrupts ()
        critical section
unlock ------> enable interrupts ()
        other code
}
```

Process 2

```
while(1){
    disable interrupts ()
    critical section
    enable interrupts ()
    other code
}
```

- Simple
  - When interrupts are disabled, context switches won't happen
- Requires privileges
  - User processes generally cannot disable interrupts
- Not suited for multicore systems

# Software Solution (Attempt 1)

Shared

```
int turn=1;
```

Process 1

```
while(1){
    while(turn == 2); // lock
    critical section
    turn = 2;  // unlock
     other code
}
```

Process 2

```
while(1){
    while(turn == 1); // lock
    critical section
    turn = 1; // unlock
    other code
}
```

- Achieves mutual exclusion
- Busy waiting – waste of power and time
- Needs to alternate execution in critical section

  *process1* →*process2* →*process1* →*process2*

# Problem with Attempt 1

Shared

```
int turn=1;
```

Process 1

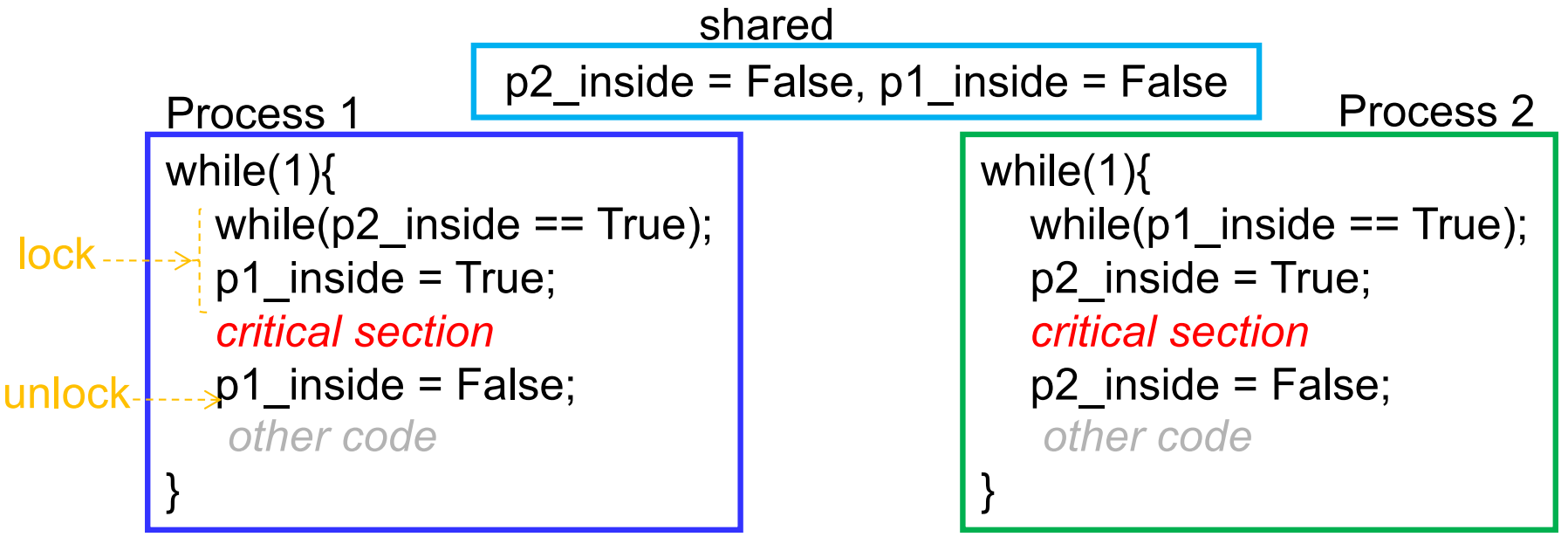
```
while(1){
    while(turn == 2); // lock
    critical section
    turn = 2;  // unlock
     other code
}
```

Process 2

```
while(1){
    while(turn == 1); // lock
    critical section
    turn = 1; // unlock
    other code
}
```

- Had a common turn flag that was modified by both processes
- This required processes to alternate.
- Possible Solution : Have two flags – one for each process

# Software Solution (Attempt 2)




- Need not alternate execution in critical section
- Does not guarantee mutual exclusion

# Attempt 2: No mutual exclusion

time

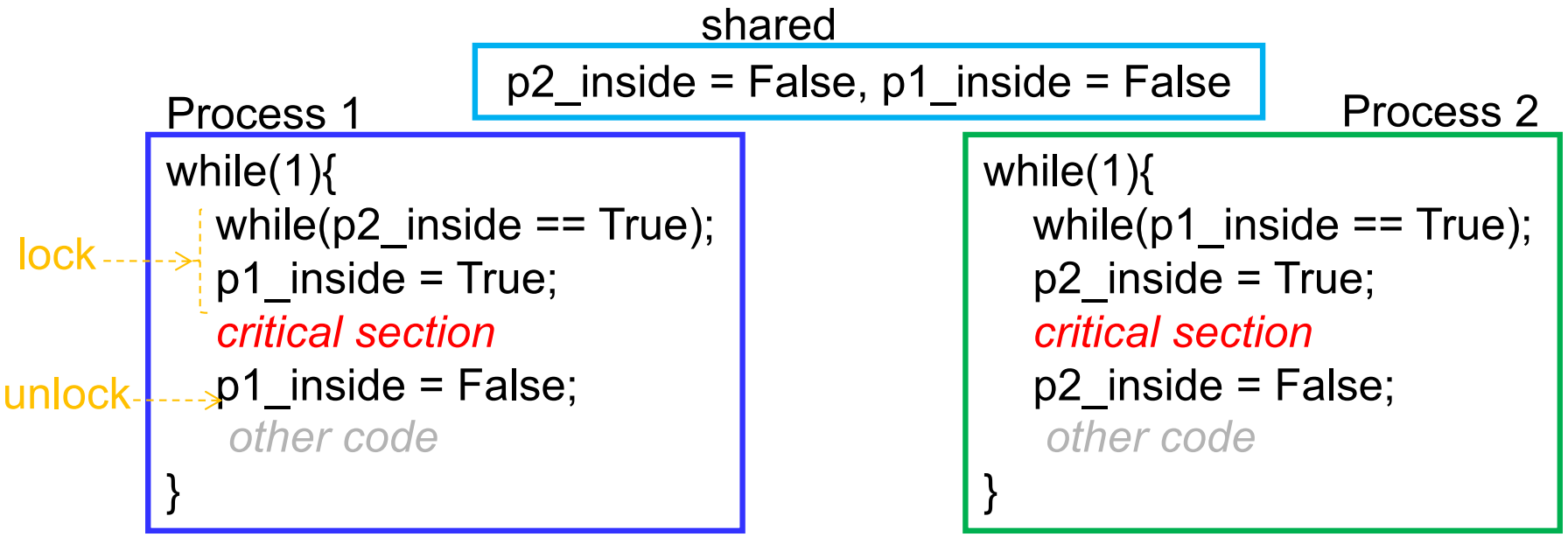
| CPU | p1_inside | p2_inside |
|---|---|---|
| while(p2_inside == True); | False | False |
| context switch | | |
| while(p1_inside == True); | False | False |
| p2_inside = True; | False | True |
| context switch | | |
| p1_inside = True; | True | True |

Both p1 and p2 can enter into the critical section at the same time

```
while(1){
    while(p2_inside == True);
    p1_inside = True;
    critical section
    p1_inside = False;
     other code
}
```

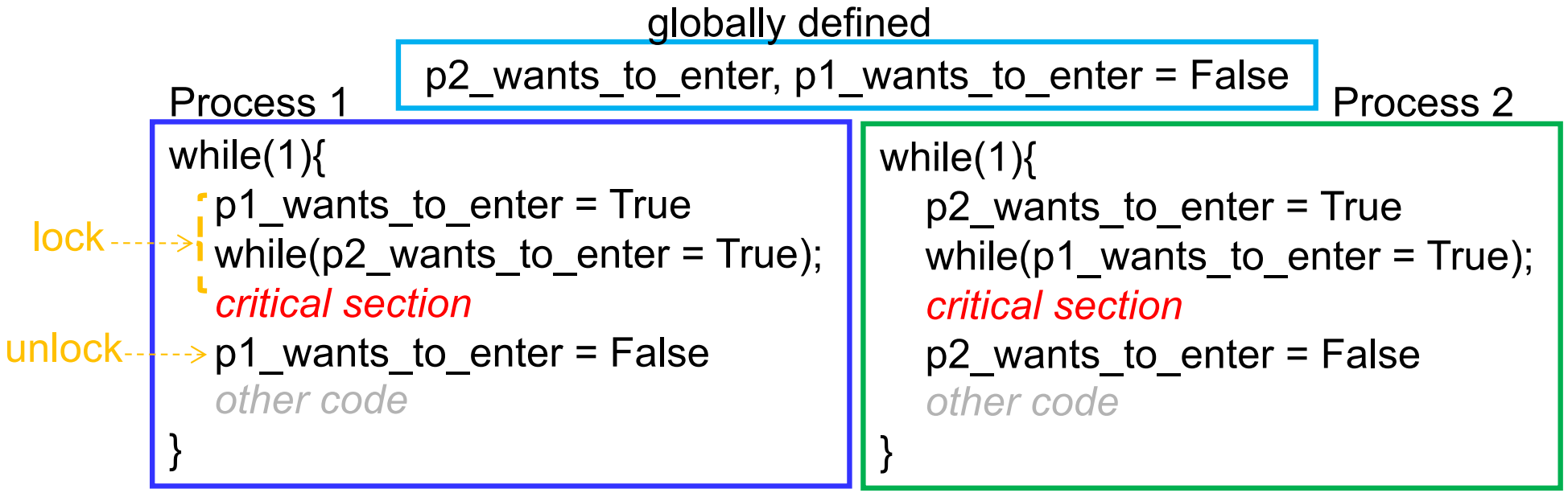
```
while(1){
    while(p1_inside == True);
    p2_inside = True;
    critical section
    p2_inside = False;
     other code
}
```

# Problem with Attempt 2




- The flag (p1_inside, p2_inside), is set after we break from the while loop.

# Software Solution (Attempt 3)




- Achieves mutual exclusion
- Does not achieve progress (could deadlock)

# Attempt 3: No Progress


| CPU | p1_inside | p2_inside |
|---|---|---|
| p1_wants_to_enter = True | False | False |
| context switch | | |
| p2_wants_to_enter = True | False | False |

There is a tie!!!

Both p1 and p2 will loop infinitely

Progress not achieved

Each process is waiting for the other

this is a deadlock

```
while(1){
    p2_wants_to_enter = True
    while(p1_wants_to_enter = True);
    critical section
    p2_wants_to_enter = False
    other code
}
```

# Deadlock

time

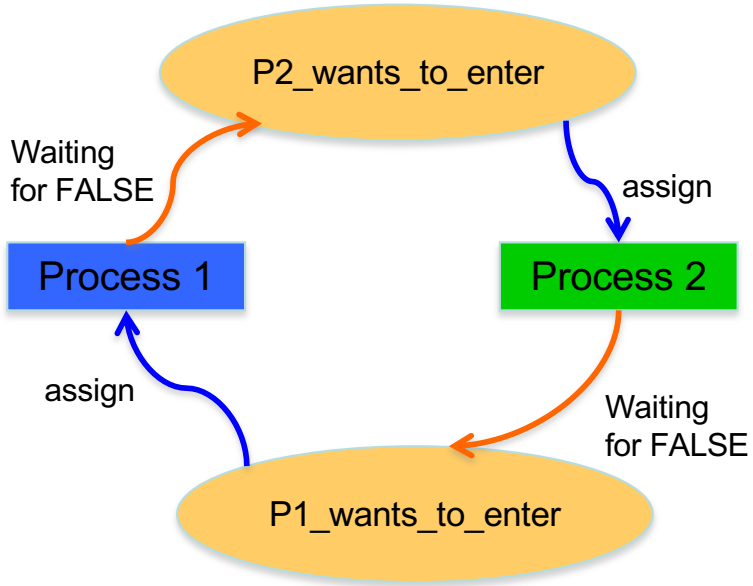
| CPU | p1_inside | p2_inside |
|---|---|---|
| p1_wants_to_enter = True | False | False |
| context switch | | |
| p2_wants_to_enter = True | False | False |

There is a tie!!!

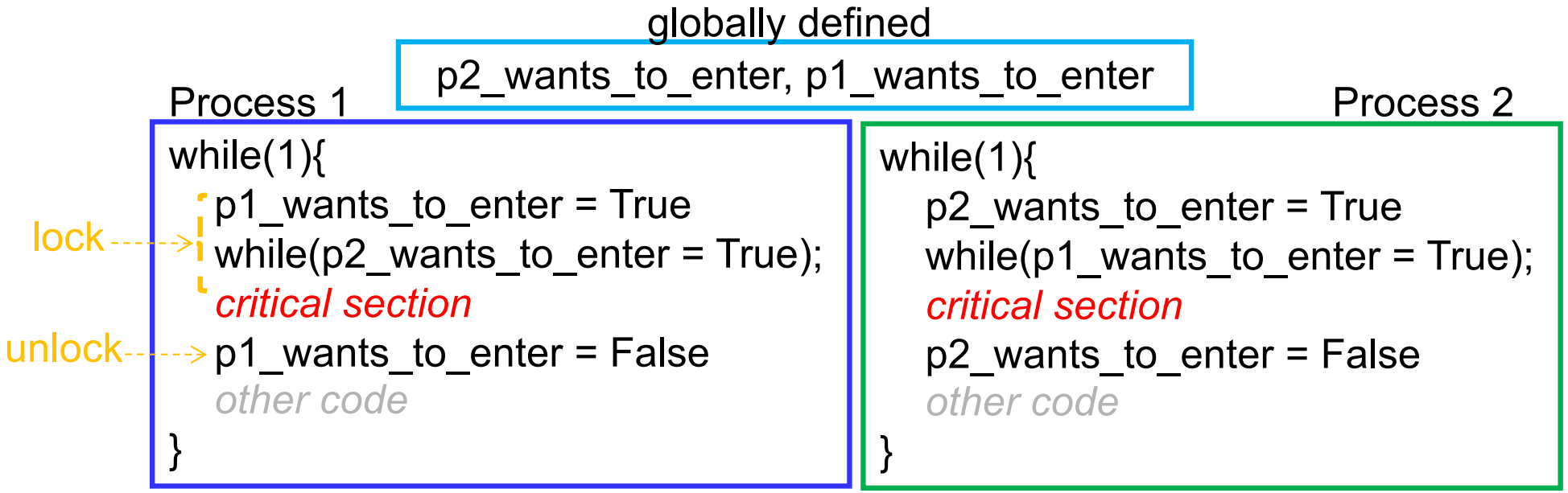
Both p1 and p2 will loop infinitely

Progress not achieved

Each process is waiting for the other this is a deadlock



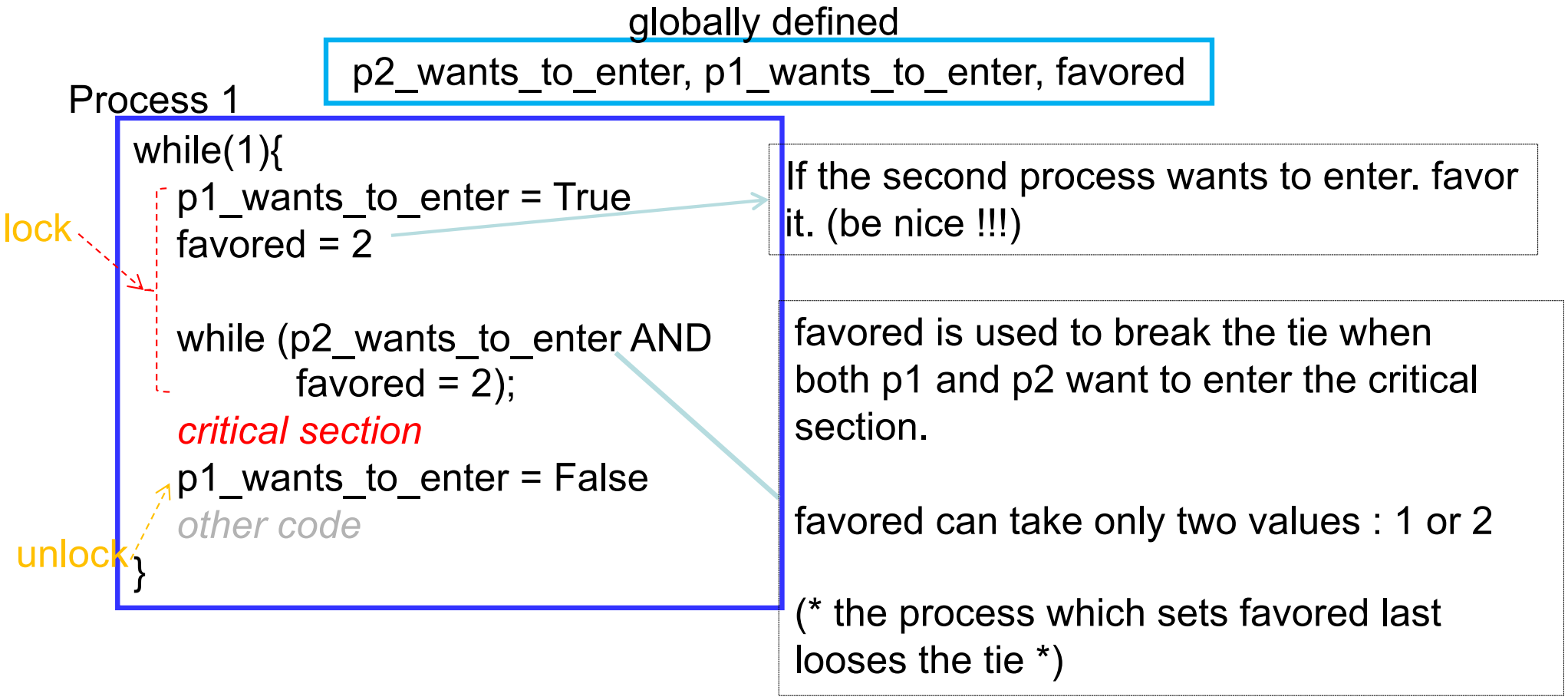
# Problem with Attempt 3




- Deadlock

  Have a way to break the deadlock

# Peterson's Solution




Break the deadlock with a 'favored' process

# Peterson's Solution

globally defined

p2_wants_to_enter, p1_wants_to_enter, favored

Process 1

```
while(1){
    p1_wants_to_enter = True
    favored = 2

    while (p2_wants_to_enter AND
            favored = 2);
    critical section
    p1_wants_to_enter = False
    other code
}
```

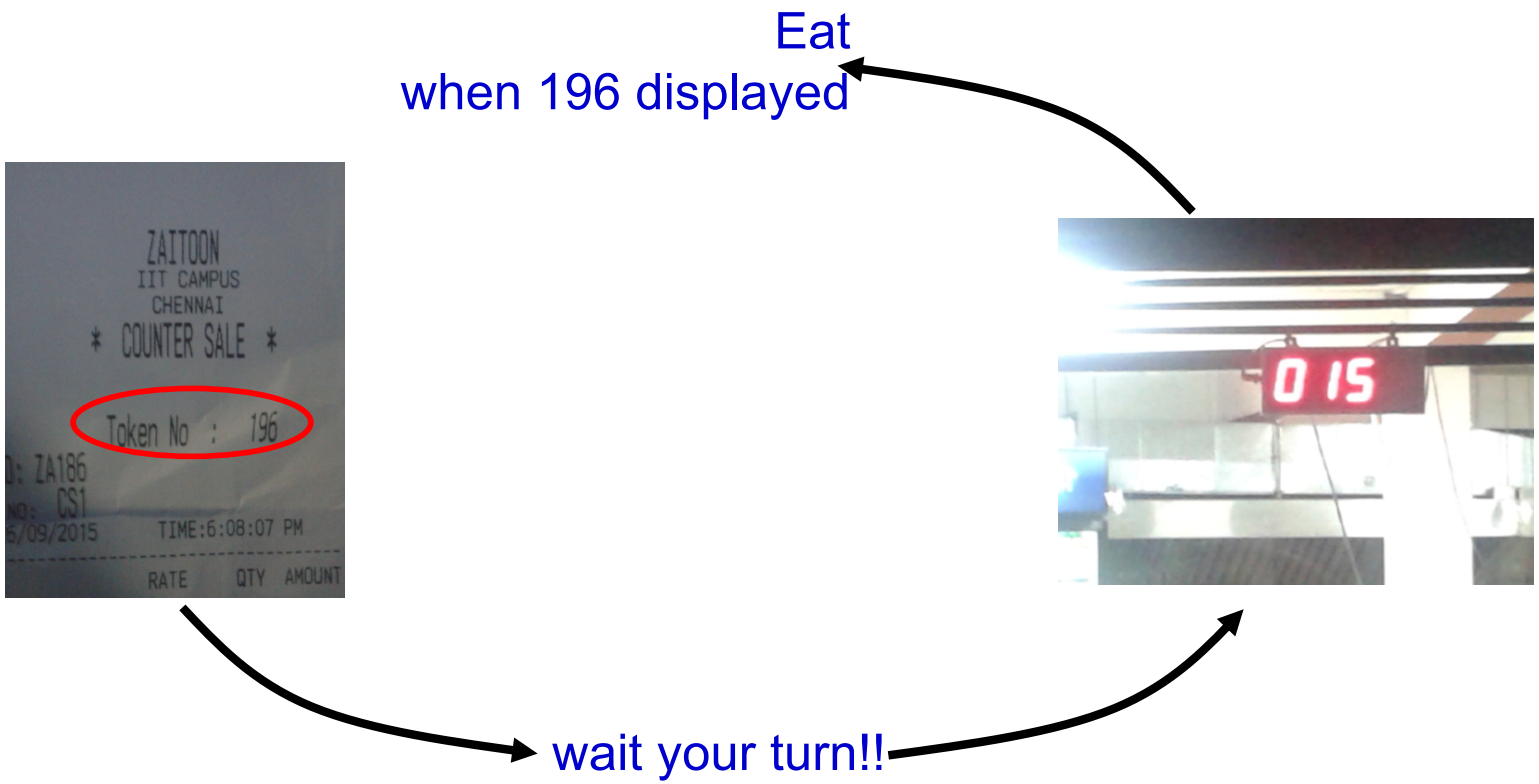
Process 2

```
while(1){
    p2_wants_to_enter = True
    favored = 1

    while (p1_wants_to_enter AND
            favored = 1);
    critical section
    p2_wants_to_enter = False
    other code
}
```

- Deadlock broken because favored can only be 1 or 2.
  - Therefore, tie is broken. Only one process will enter the critical section
- Solves Critical Section problem for two processes

# Bakery Algorithm

- Synchronization between N > 2 processes
- By Leslie Lamport




http://research.microsoft.com/en-us/um/people/lamport/pubs/bakery.pdf

# Simplified Bakery Algorithm

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```
lock(i){
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

critical section

```
unlock(i){
    num[i] = 0;
}
```

This is at the doorway!!!
It has to be atomic
to ensure two processes
do not get the same token

# Simplified Bakery Algorithm (example)

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```
lock(i){
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

critical section

```
unlock(i){
    num[i] = 0;
}
```

| P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|
| 0 | 4 | 1 | 2 | 3 |

# Simplified Bakery Algorithm (example)

Processes numbered 0 to N-1
num is an array N integers (initially 0).
Each entry corresponds to a process

```
lock(i){
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

critical section

| P1 | P2 | P3 | P4 | **P5** |
|---|---|---|---|---|
| 0 | 4 | 1 | 2 | 3 |

```
unlock(i){
    num[i] = 0;
}
```

# Simplified Bakery Algorithm (why atomic doorway?)

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```
lock(i){
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

This is at the doorway!!!
Assume it is not atomic

critical section

```
unlock(i){
    num[i] = 0;
}
```

| P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|
| 0 | 3 | 0 | 2 | 2 |

P4 and P5 can enter the critical section at the same time.

# Original Bakery Algorithm (making MAX atomic)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

critical section

```
unlock(i){
    num[i] = 0;
}
```

Choosing ensures that a process
Is not at the doorway
i.e., the process is not 'choosing'
a value for num

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

# Original Bakery Algorithm (making MAX atomic)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

critical section

```
unlock(i){
    num[i] = 0;
}
```

Favor one process when there is a conflict.
If there are two processes, with the same num value, favor the process with the smaller id (i)

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

# Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

critical section

```
unlock(i){
    num[i] = 0;
}
```

| P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|
| 0 | 3 | 1 | 2 | 2 |

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

# Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

critical section

```
unlock(i){
    num[i] = 0;
}
```

| P1 | P2 | P3 | P4 | **P5** |
|---|---|---|---|---|
| 0 | 3 | 1 | 2 | 2 |

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

# How to Implement Locking
## (Hardware Solutions and Usage)

# Analyze this

- Does this scheme provide mutual exclusion?

Process 1

```
while(1){
    while(lock != 0);
    lock= 1; // lock
    critical section
    lock = 0;  // unlock
    other code
}
```

lock=0

Process 2

```
while(1){
    while(lock != 0);
    lock = 1; // lock
    critical section
    lock = 0; // unlock
    other code
}
```

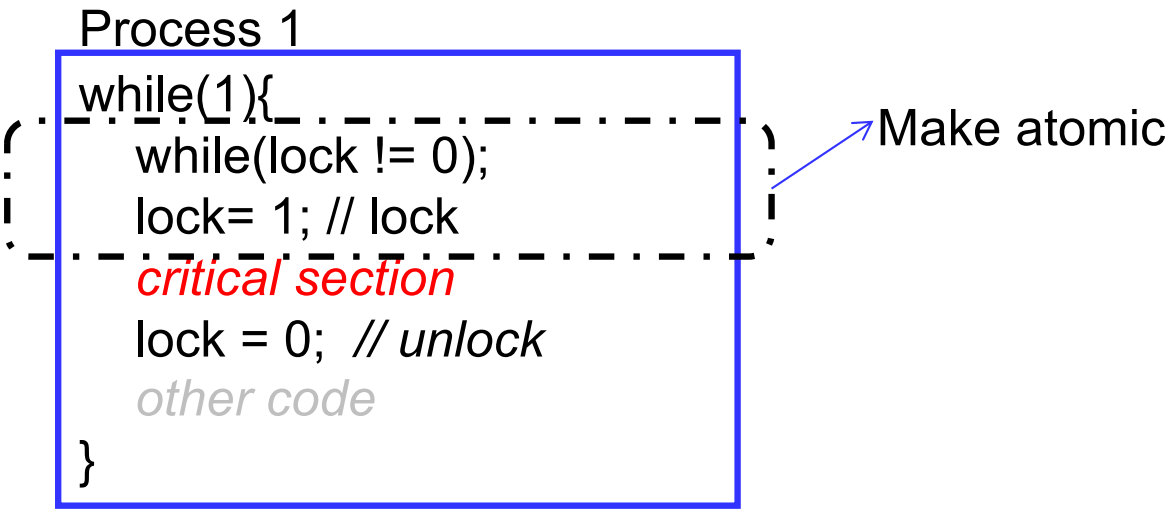
No

```
lock = 0
P1: while(lock != 0);
P2: while(lock != 0);
P2: lock = 1;
P1: lock = 1;
…. Both processes in critical section
```

context switch

# If only…

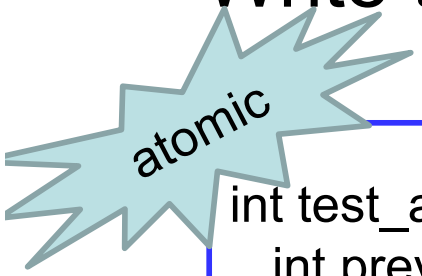
- We could make this operation atomic




Hardware to the rescue….

# Hardware Support (Test & Set Instruction)

- Write to a memory location, return its old value




```
int test_and_set(int  *L){
   int prev = *L;
   *L = 1;
   return prev;
}
```

equivalent software representation
(the entire function is executed atomically)

```
while(1){
   while(test_and_set(&lock) == 1);
   critical section
   lock = 0;  // unlock
   other code
}
```

Usage for locking

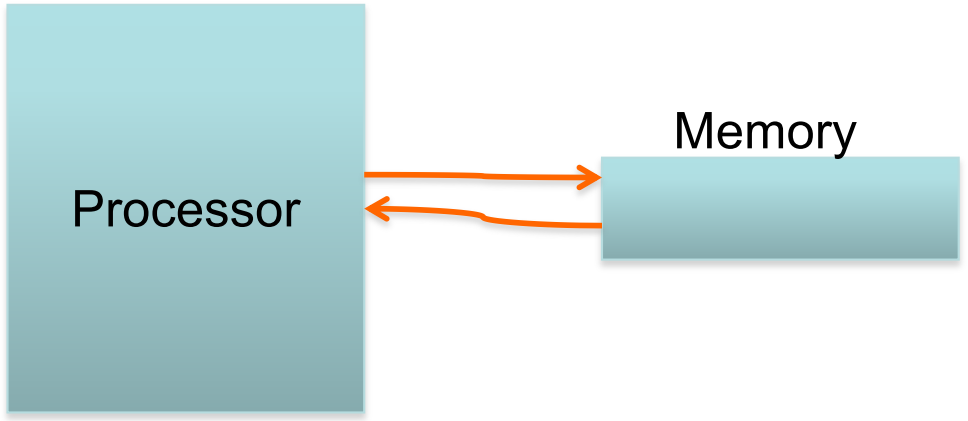
Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.
So the first invocation of test_and_set will read a 0 and set lock to 1 and return. The second test_and_set invocation will then see lock as 1, and will loop continuously until lock becomes 0

# Hardware Support (Test & Set Instruction)

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
   int prev = *L;
   *L = 1;
   return prev;
}
```



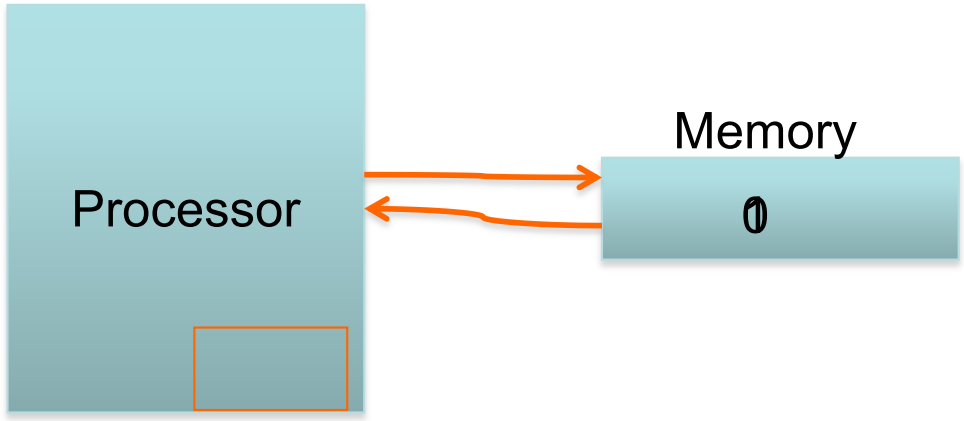

equivalent software representation
(the entire function is executed atomically)

# Hardware Support (Test & Set Instruction)

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
   int prev = *L;
   *L = 1;
   return prev;
}
```




equivalent software representation
(the entire function is executed atomically)

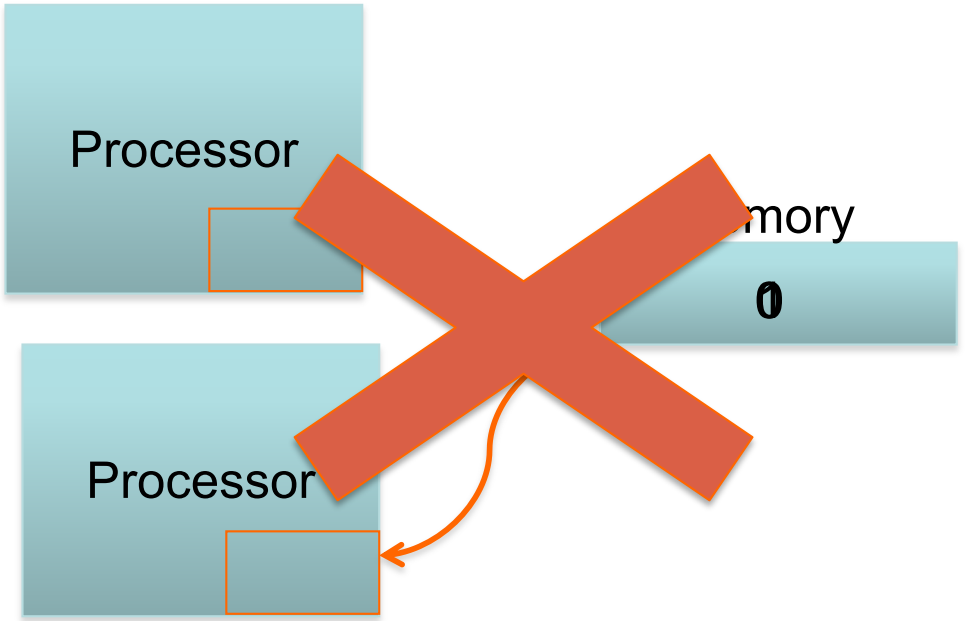
# Hardware Support (Test & Set Instruction)

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
   int prev = *L;
   *L = 1;
   return prev;
}
```

equivalent software representation (the entire function is executed atomically)




Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.

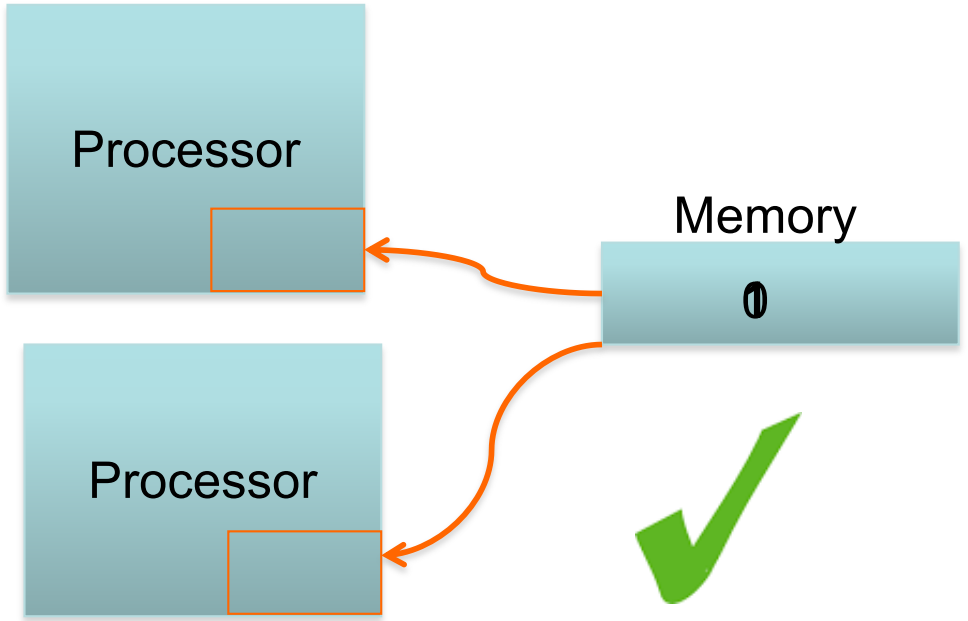
# Hardware Support (Test & Set Instruction)

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
   int prev = *L;
   *L = 1;
   return prev;
}
```

equivalent software representation (the entire function is executed atomically)




Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.

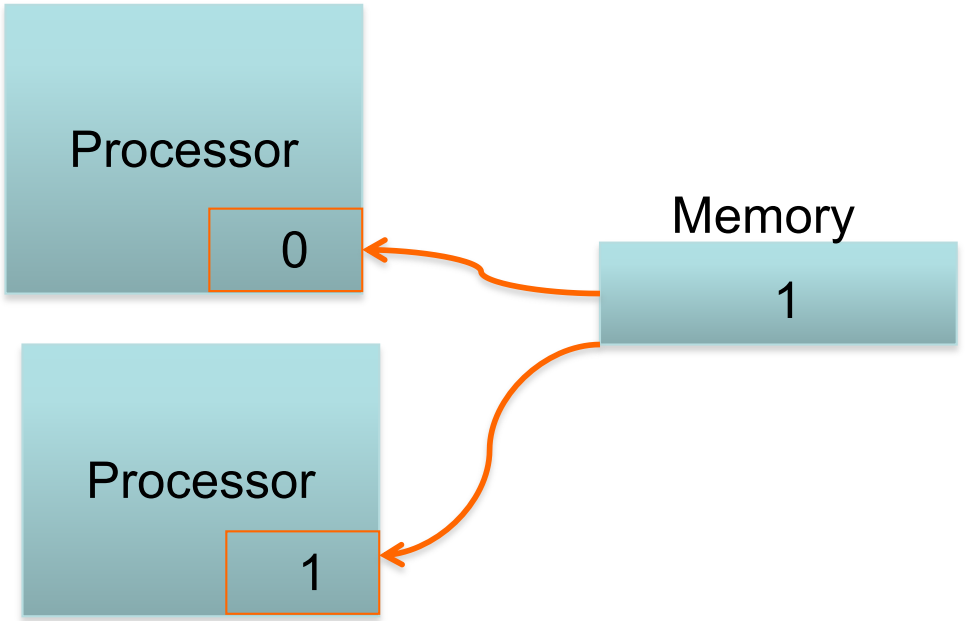
# Hardware Support (Test & Set Instruction)

- Write to a memory location, return its old value

atomic

```
int test_and_set(int  *L){
   int prev = *L;
   *L = 1;
   return prev;
}
```

equivalent software representation (the entire function is executed atomically)




Why does this work? If two CPUs execute test_and_set at the same time, the hardware ensures that one test_and_set does both its steps before the other one starts.

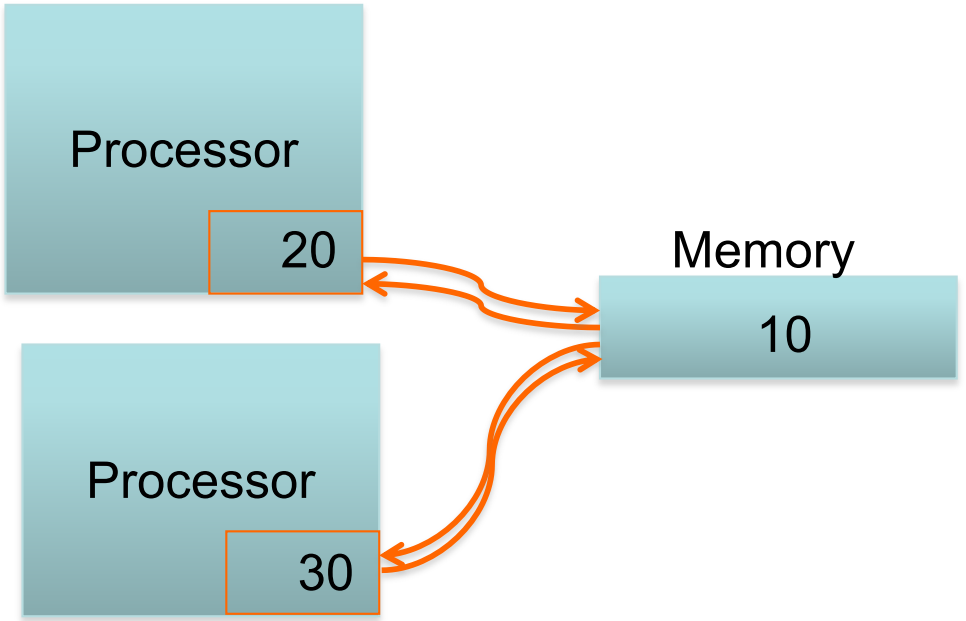
# Intel Hardware Support (xchg Instruction)

- Write to a memory location, return its old value

atomic

```
int xchg(int  *L, int v){
   int prev = *L;
   *L = v;
   return prev;
}
```

equivalent software representation (the entire function is executed atomically)



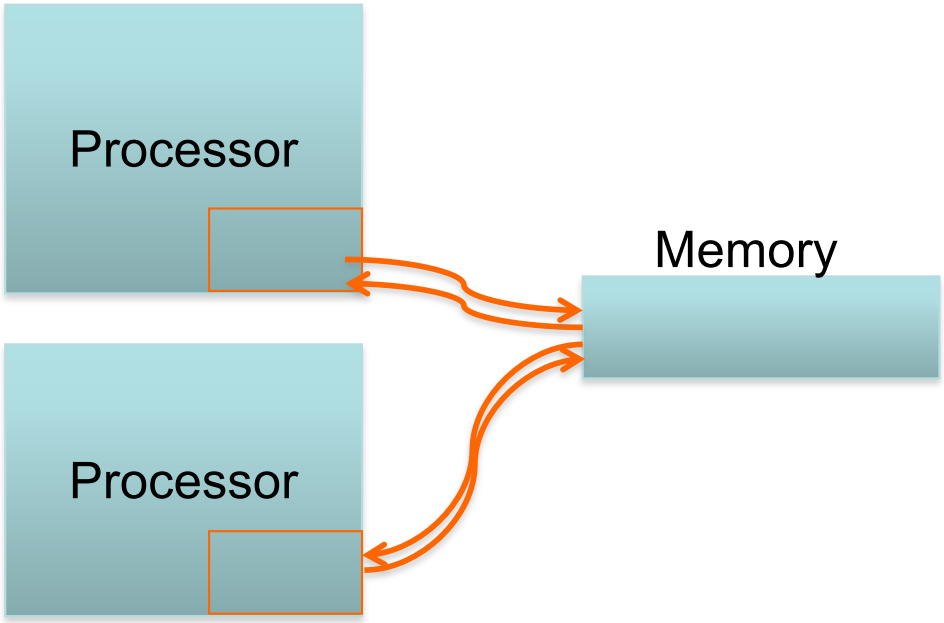

Why does this work? If two CPUs execute xchg at the same time, the hardware ensures that one xchg completes, only then the second xchg starts.

# Intel Hardware Support (using xchg instruction)

Note. %eax is returned

typical usage :

xchg reg, mem



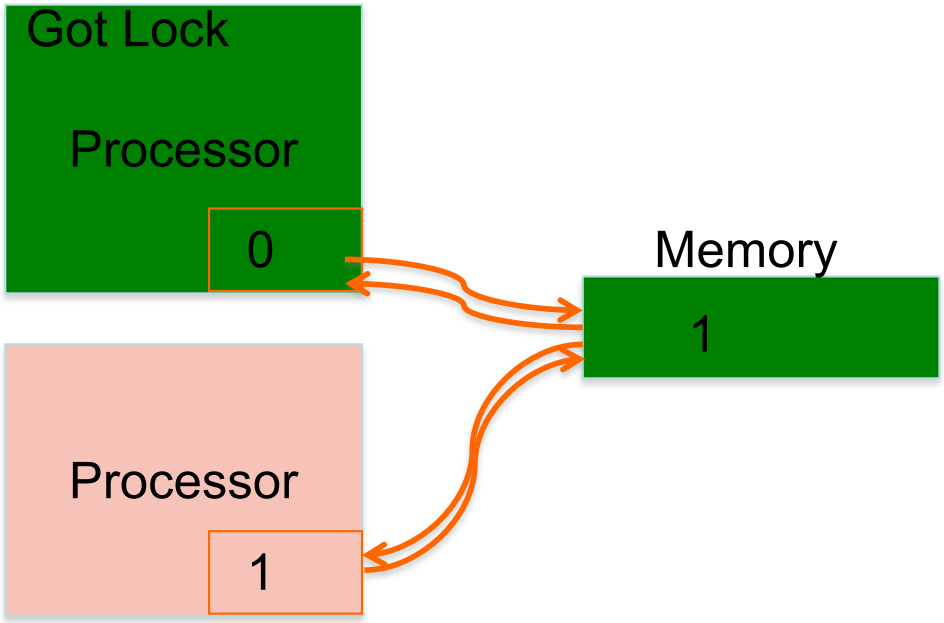

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

# Intel Hardware Support (using xchg instruction)

Note. %eax is returned

typical usage :

xchg reg, mem



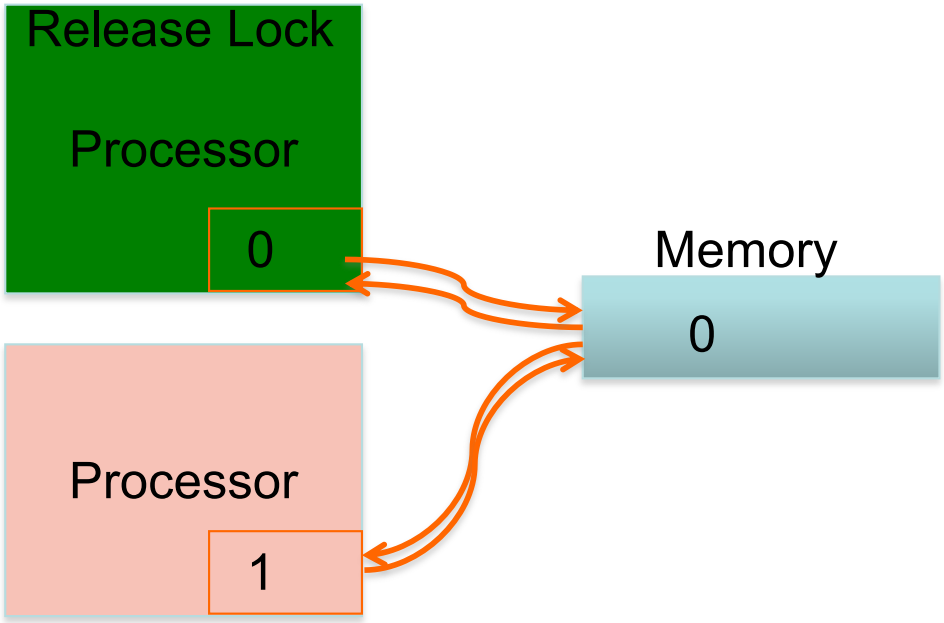

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

# Intel Hardware Support (using xchg instruction)

Note. %eax is returned

typical usage :

xchg reg, mem



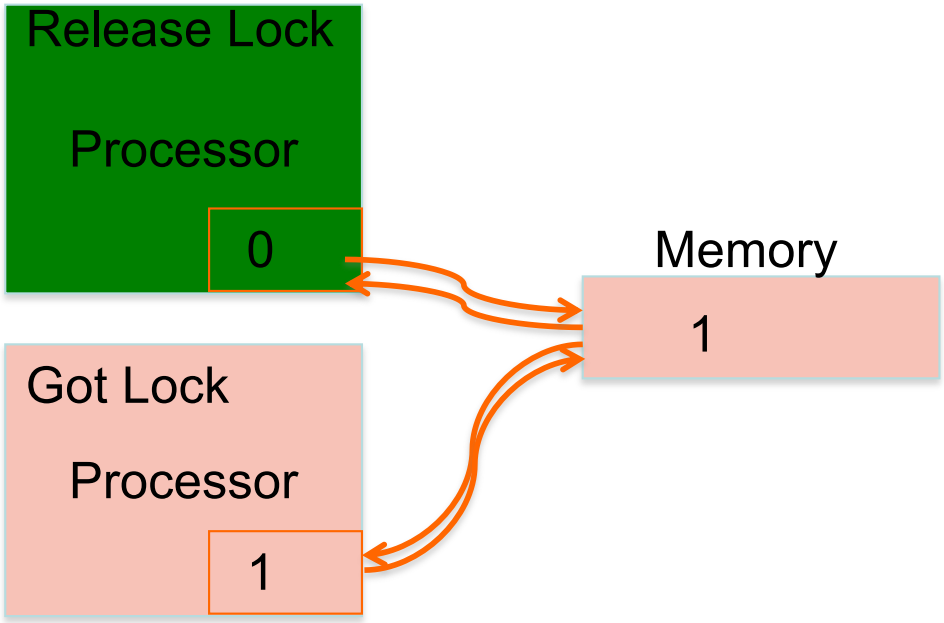

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

# Intel Hardware Support (using xchg instruction)

Note. %eax is returned

typical usage :

xchg reg, mem



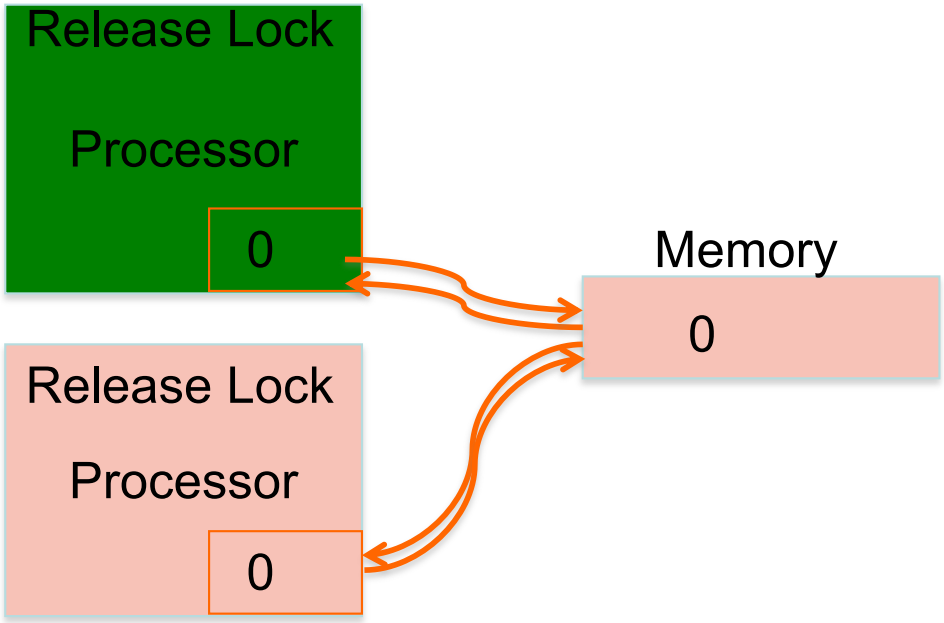

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

# Intel Hardware Support (using xchg instruction)

Note. %eax is returned

typical usage :

xchg reg, mem




```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

# High Level Constructs

- Spinlock
- Mutex
- Semaphore

# Spinlocks Usage

Process 1

```
acquire(&locked)
critical section
release(&locked)
```

Process 2

```
acquire(&locked)
critical section
release(&locked)
```

- One process will acquire the lock
- The other will wait in a loop repeatedly checking if the lock is available
- The lock becomes available when the former process releases it

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void acquire(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
  }
}

void release(int *locked){
    locked = 0;
}
```

See spinlock.c and spinlock.h in xv6 [15]

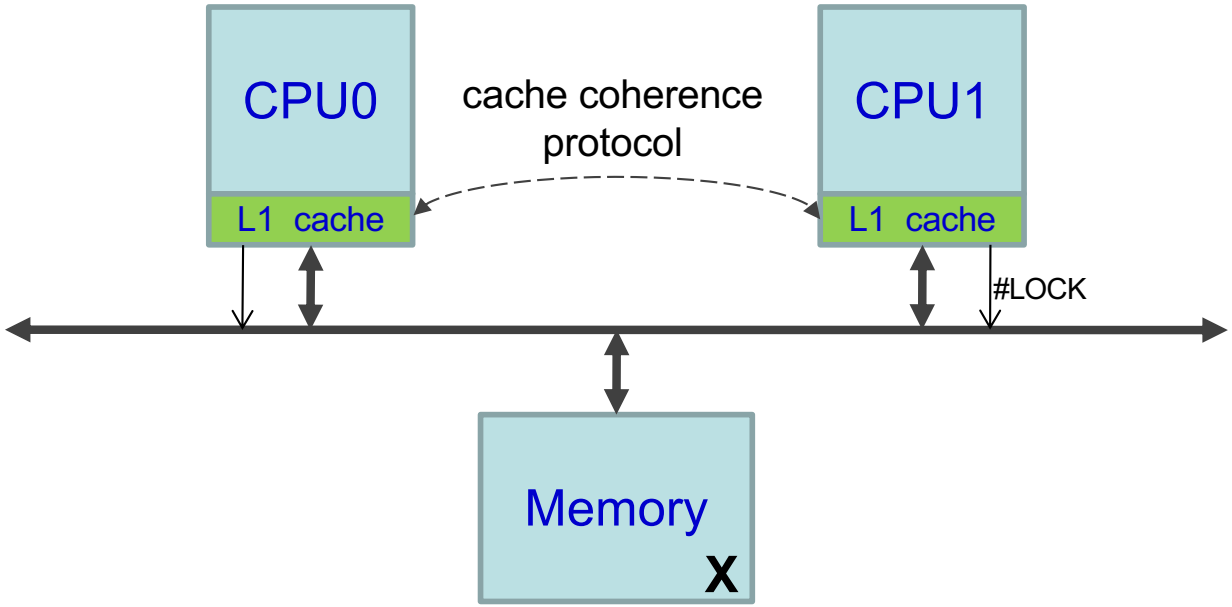
# Issues with Spinlocks

**xchg %eax, X**

- No compiler optimizations should be allowed
  - Should not make X a register variable
    - Write the loop in assembly or use volatile
- Should not reorder **memory** loads and stores
    - Use serialized instructions (which forces instructions not to be reordered)
    - Luckly xchg is already implements serialization

# More issues with Spinlocks

**xchg %eax, X**




- No caching of (X) possible. All xchg operations are bus transactions.
  - CPU asserts the LOCK, to inform that there is a ‘locked ‘ memory access
- acquire function in spinlock invokes xchg in a loop…each operation is a bus transaction …. huge performance hits

# A better acquire

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}
```

```
void acquire(int *locked){
  reg = 1
  while(1)
      if(xchg(locked, reg) == 0)
          break;
}
```

Original.
Loop with xchg.
Bus transactions.
Huge overheads

```
void acquire(int *locked) {
  reg = 1;
  while (xchg(locked, reg) == 1)
        while (*locked == 1);
}
```

Better way
inner loop allows caching of
locked. Access cache instead of memory.

# Spinlocks (when should it be used?)

- Characteristic : busy waiting
  - Useful for short critical sections, where much CPU time is not wasted waiting
    - eg. To increment a counter, access an array element, etc.
  - Not useful, when the period of wait is unpredictable or will take a long time
    - eg. Not good to read page from disk.
    - Use mutex instead (…mutex)

# Spinlock in pthreads

```c
#include <pthread.h>
#include <stdio.h>

int global_counter;
pthread_spinlock_t splk;

void *thread_fn(void *arg){
  long id = (long) arg;
  while(1){
    pthread_spin_lock(&splk);
    if (id == 1) global_counter++;
    else global_counter--;
    pthread_spin_unlock(&splk);
    printf("%d(%d)\n", id, global_counter);
    sleep(1);
  }

  return NULL;
}

int main(){
  pthread_t t1, t2;

  pthread_spin_init(&splk, PTHREAD_PROCESS_PRIVATE);
  pthread_create(&t1, NULL, thread_fn, (void *)1);
  pthread_create(&t2, NULL, thread_fn, (void *)2);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  pthread_spin_destroy(&splk);
  printf("Exiting main\n");
  return 0;
}
```

lock

unlock

create spinlock

destroy spinlock

# Mutexes

- Can we do better than busy waiting?
  - If critical section is locked then yield CPU
    - Go to a SLEEP state
  - While unlocking, wake up sleeping process

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
      sleep();
  }
}

void unlock(int *locked){
    locked = 0;
    wakeup();
}
```

Ref: wakeup(2864), sleep(2803)

# Thundering Herd Problem

- A large number of processes wake up (almost simultaneously) when the event occurs.
  - All waiting processes wake up
  - Leading to several context switches
  - All processes go back to sleep except for one, which gets the critical section
    - Large number of context switches
    - Could lead to starvation

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else
      sleep();
  }
}

void unlock(int *locked){
    locked = 0;
    wakeup();
}
```

# Thundering Herd Problem

- The Solution
  - When entering critical section, push into a queue before blocking
  - When exiting critical section, wake up only the first process in the queue

```
int xchg(addr, value){
  %eax = value
  xchg %eax, (addr)
}

void lock(int *locked){
  while(1){
    if(xchg(locked, 1) == 0)
      break;
    else{
      // add this process to Queue
       sleep();
    }
  }
}

void unlock(int *locked){
    locked = 0;
    // remove process P from queue
    wakeup(P)
}
```

# pthread Mutex

- pthread_mutex_lock
- pthread_mutex_unlock

# Locks and Priorities

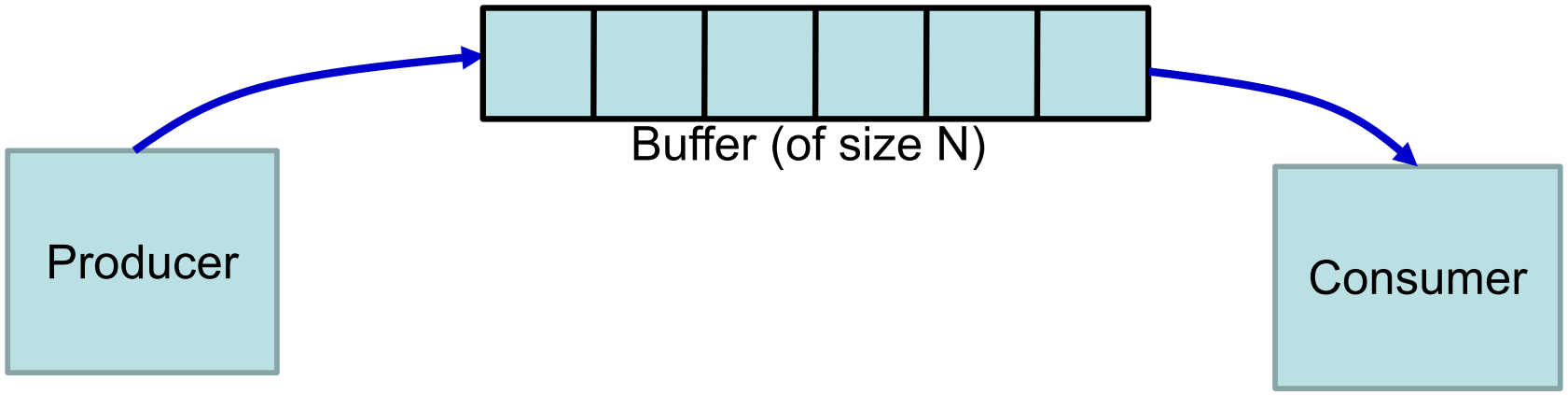
- What happens when a high priority task requests a lock, while a low priority task is in the critical section
  - Priority Inversion
  - Possible solution
    - Priority Inheritance

Interesting Read : Mass Pathfinder
http://research.microsoft.com/en-us/um/people/mbj/mars_pathfinder/mars_pathfinder.html

# Semaphores

# Producer – Consumer Problems

- Also known as *Bounded buffer Problem*
- Producer produces and stores in buffer, Consumer consumes from buffer
- Trouble when
  - Producer produces, but buffer is full
  - Consumer consumes, but buffer is empty



# Producer-Consumer Code

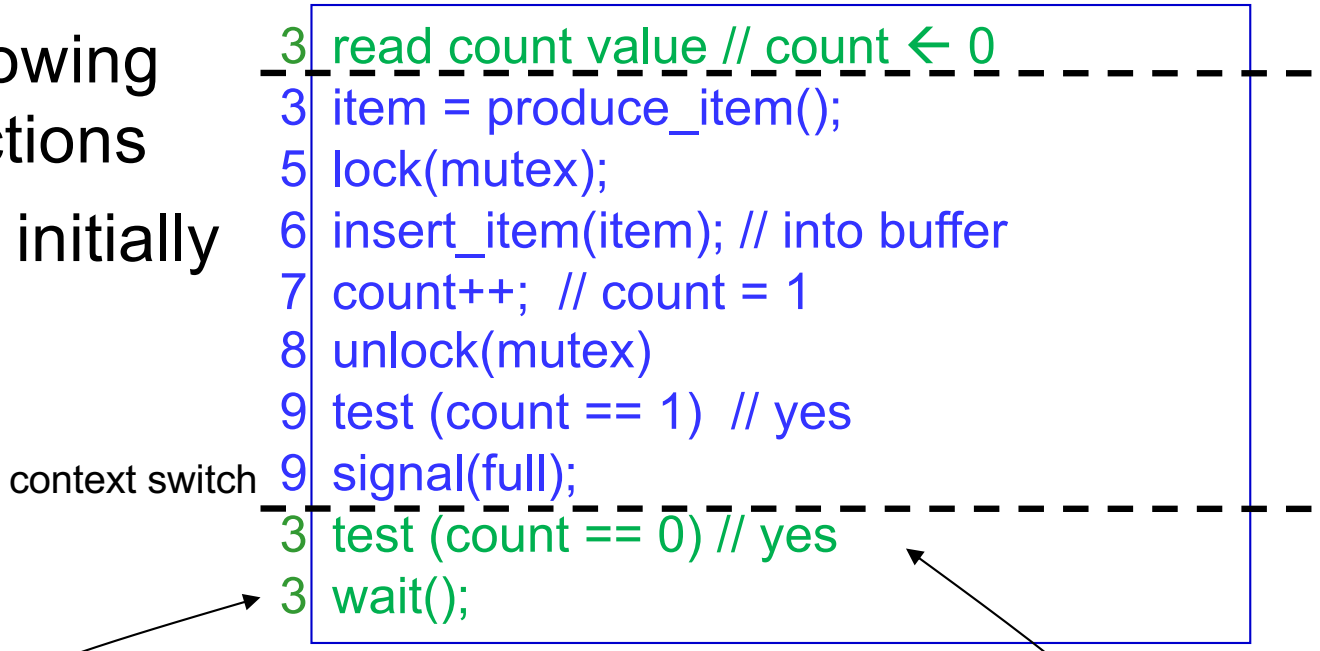
Buffer of size N
int count=0;
Mutex mutex, empty, full;

```
1  void producer(){
2      while(TRUE){
3          item = produce_item();
4          if (count == N) sleep(empty);
5          lock(mutex);
6          insert_item(item); // into buffer
7          count++;
8          unlock(mutex);
9          if (count == 1) wakeup(full);
10     }
   }
```

```
1  void consumer(){
2      while(TRUE){
3          if (count == 0) sleep(full);
4          lock(mutex);
5          item = remove_item(); // from buffer
6          count--;
7          unlock(mutex);
8          if (count == N-1) wakeup(empty);
9          consume_item(item);
10     }
   }
```

# Lost Wakeups

- Consider the following context of instructions
- Assume buffer is initially empty




Note, the wakeup is lost.
Consumer waits even though buffer is not empty.
Eventually producer and consumer will wait infinitely

consumer
still uses the old value of count (ie 0)

# Semaphores

- Proposed by Dijkstra in 1965
- Functions down and up must be atomic
- down also called **P** (Proberen Dutch for try)
- up also called **V** (Verhogen, Dutch form make higher)
- Can have different variants
  - Such as blocking, non-blocking
- If S is initially set to 1,
  - Blocking semaphore similar to a Mutex
  - Non-blocking semaphore similar to a spinlock

```
void down(int *S){
  while( *S <= 0);
  *S--;
}

void up(int *S){
   *S++;
}
```

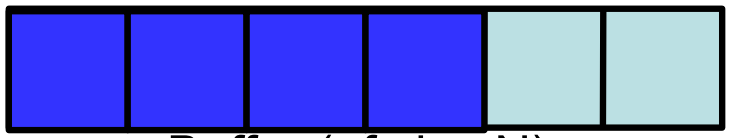
# Producer-Consumer with Semaphores

Buffer of size N
int count;

**full = 0, empty = N**

Number of filled blocks in the buffer

Number of empty blocks in the buffer



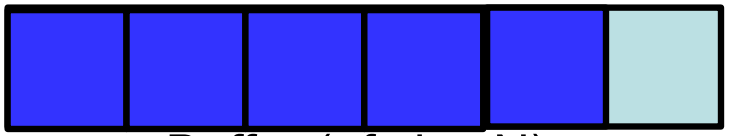
Buffer (of size N)

N = 6
fill = 4
empty = 2

# Producer-Consumer with Semaphores

**full = 0, empty = N**

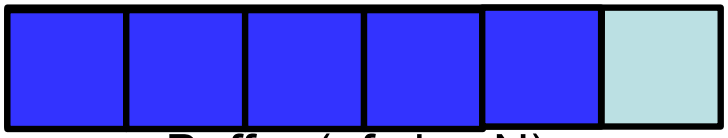
```
void producer(){
    while(TRUE){
        item = produce_item();
        down(empty);

        insert_item(item); // into buffer

        up(full);
    }
}
```



Buffer (of size N)

**N = 6**
**fill = 5 4**
empty = 1 2

# Producer-Consumer with Semaphores




N = 6
fill = 4
empty = 2

```
void consumer(){
    while(TRUE){
        down(full);

        item = remove_item(); // from buffer

        up(empty);
        consume_item(item);
    }
}
```

# Producer-Consumer with Semaphores

**The FULL Buffer**

```
void producer(){
    while(TRUE){
        item = produce_item();
        down(empty);

        insert_item(item); // into buffer

        up(full);
    }
}
```

```
void consumer(){
    while(TRUE){
        down(full);

        item = remove_item(); // from buffer

        up(empty);
        consume_item(item);
    }
}
```


N = 6
fill = 6

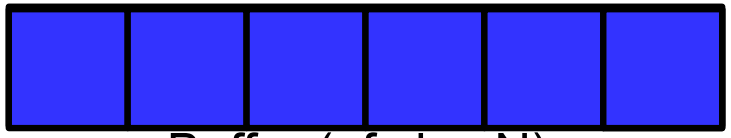
empty = 0

# Producer-Consumer with Semaphores

**The FULL Buffer**

```
void producer(){
    while(TRUE){
        item = produce_item();
        down(empty);

        insert_item(item); // into buffer

        up(full);
    }
}
```

```
void consumer(){
    while(TRUE){
        down(full);

        item = remove_item(); // from buffer

        up(empty);
        consume_item(item);
    }
}
```




N = 6
fill = 6

**empty** = 0

# Producer-Consumer with Semaphores

**The Empty Buffer**

```
void producer(){
    while(TRUE){
        item = produce_item();
        down(empty);

      insert_item(item); // into buffer

        up(full);
    }
}
```

```
void consumer(){
    while(TRUE){
        down(full);

        item = remove_item(); // from buffer

        up(empty);
        consume_item(item);
    }
}
```

Buffer (of size N)

N = 6
fill = 0

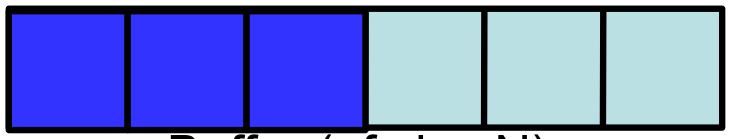
empty = 6

# Producer-Consumer with Semaphores

**Serializing Access to the Buffer**

```
void producer(){
    while(TRUE){
        item = produce_item();
        down(empty);
        lock(mutex)
        insert_item(item); // into buffer
        unlock(mutex)
        up(full);
    }
}
```

```
void consumer(){
    while(TRUE){
        down(full);
        lock(mutex)
        item = remove_item(); // from buffer
        unlock(mutex)
        up(empty);
        consume_item(item);
    }
}
```

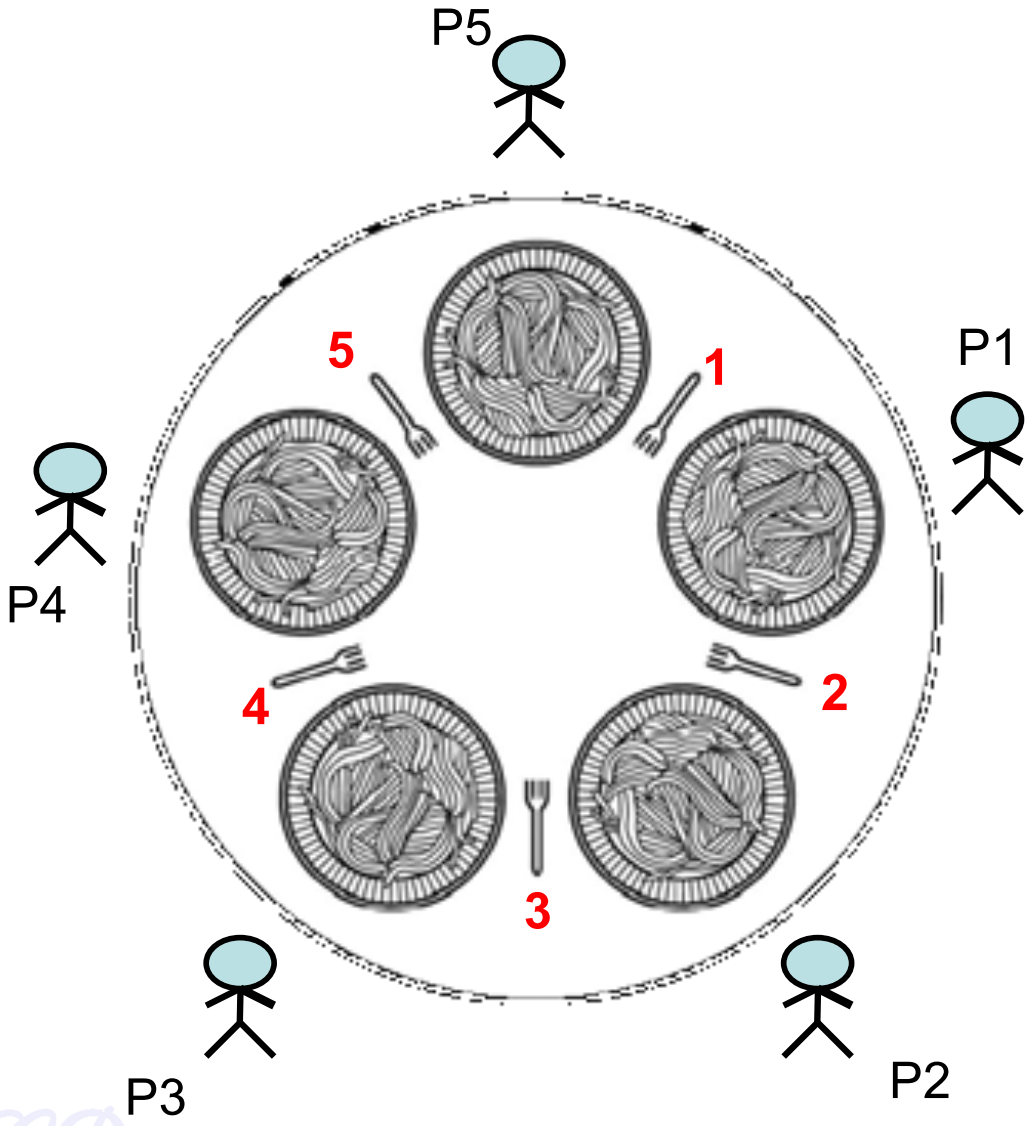



N = 6
fill = 3

empty = 3

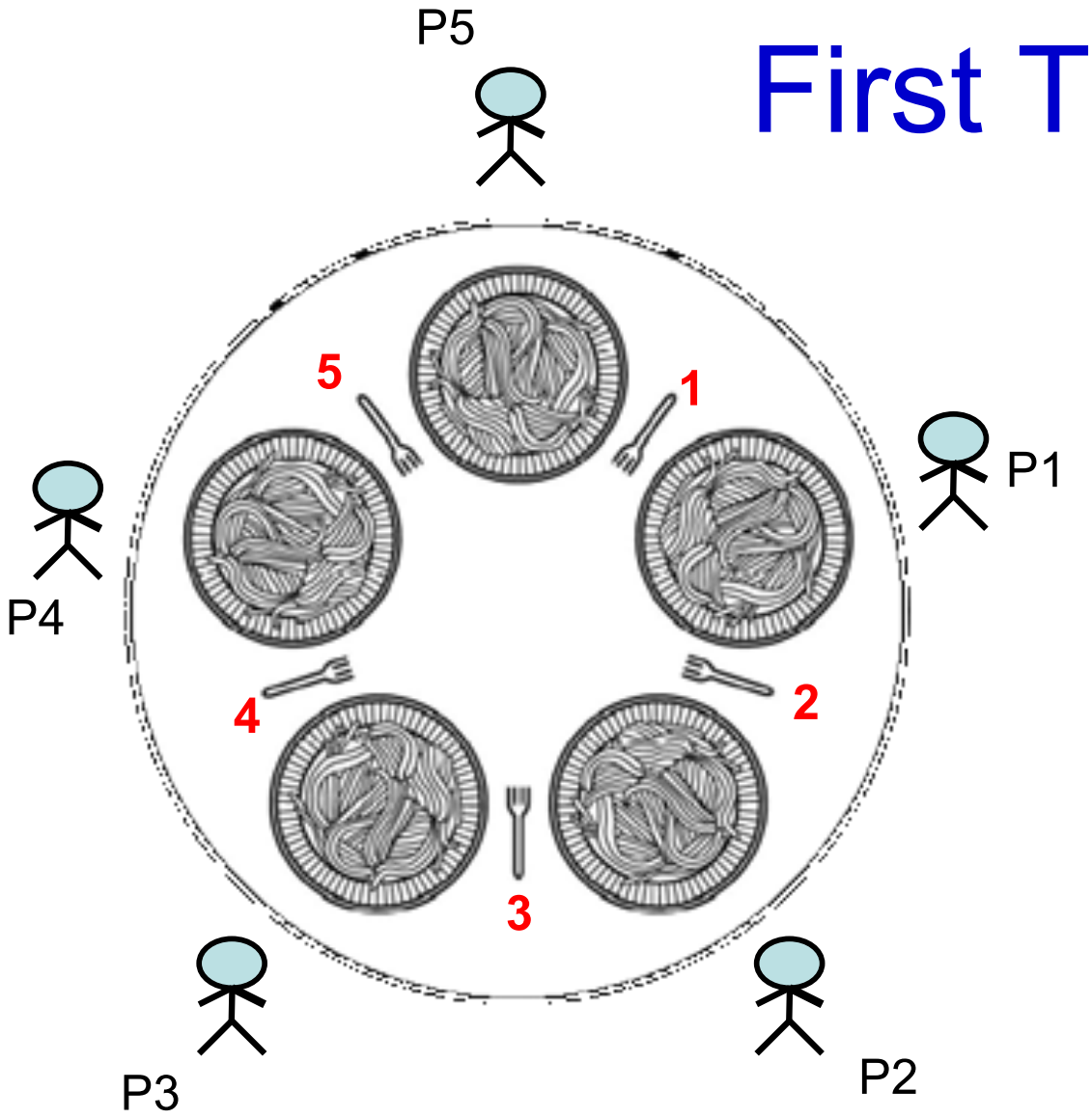
# Dining Philosophers Problem

# Dining Philosophers Problem



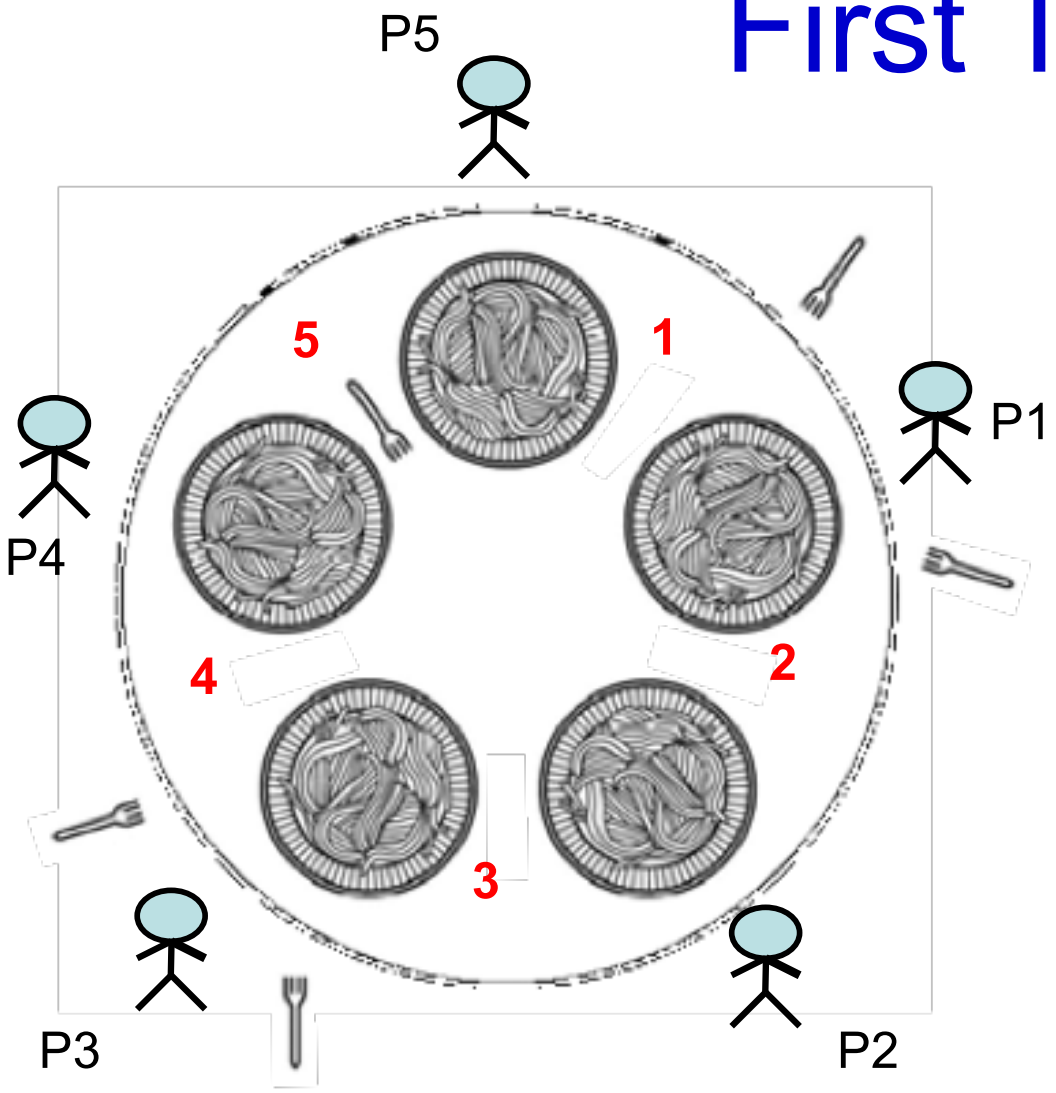

- **Philosophers either think or eat**
- To eat, a philosopher needs to hold both forks (the one on his left and the one on his right)
- If the philosopher is not eating, he is thinking.
- Problem Statement : Develop an algorithm where no philosopher starves.

# First Try




```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      take_fork(Ri);
      take_fork(Li);
      eat();
      put_fork(Ri);
      put_fork(Li);
  }
}
```

# First Try

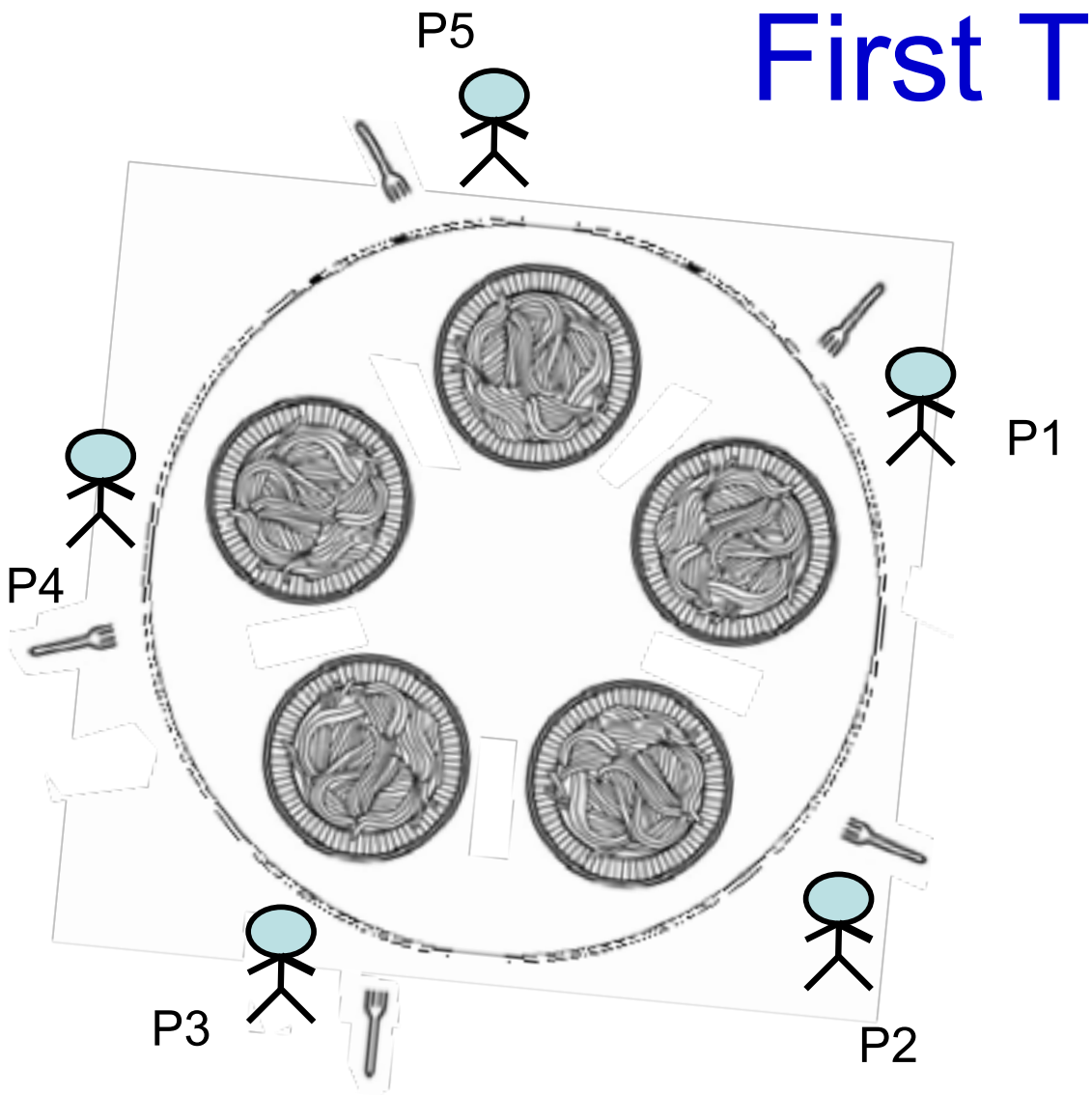



```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      take_fork(Ri);
      take_fork(Li);
      eat();
      put_fork(Ri);
      put_fork(Li);
  }
}
```

What happens if only philosophers P1 and P3 are always given the priority?

P4, P5, and P2 starves… so scheme needs to be fair

# First Try



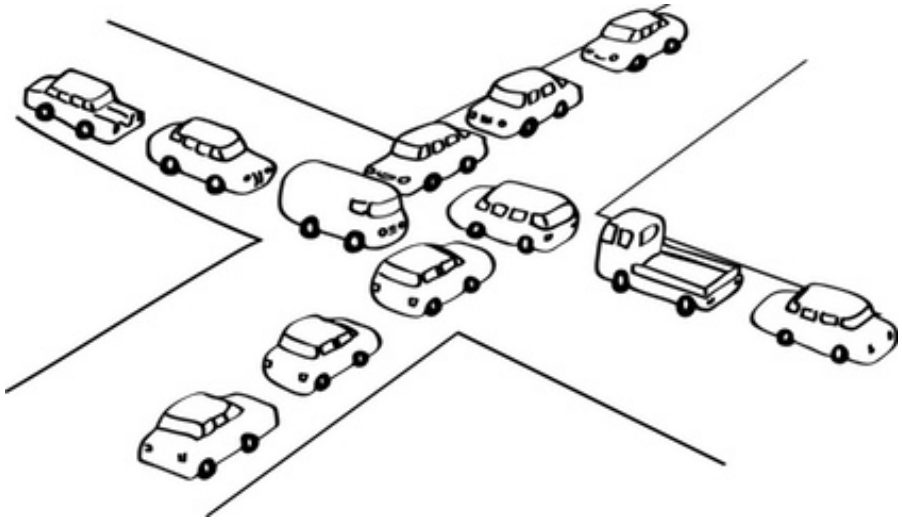

```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      take_fork(Ri);
      take_fork(Li);
      eat();
      put_fork(Ri);
      put_fork(Li);
  }
}
```

What happens if all philosophers decide to pick up their right forks at the same time?

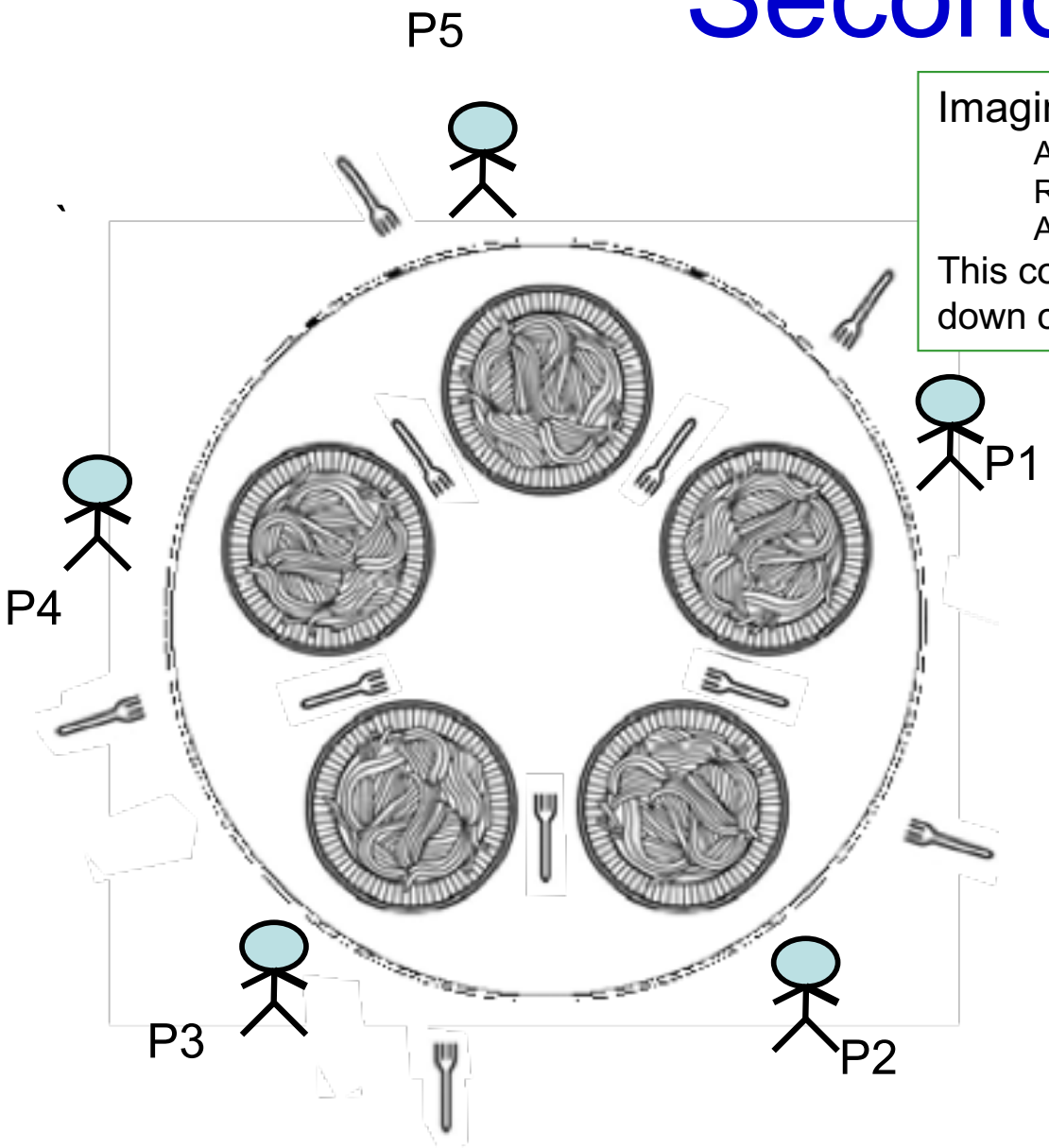
Possible starvation due to deadlock

# Deadlocks

- A situation where programs continue to run indefinitely without making any progress
- Each program is waiting for an event that another process can cause



# Second try



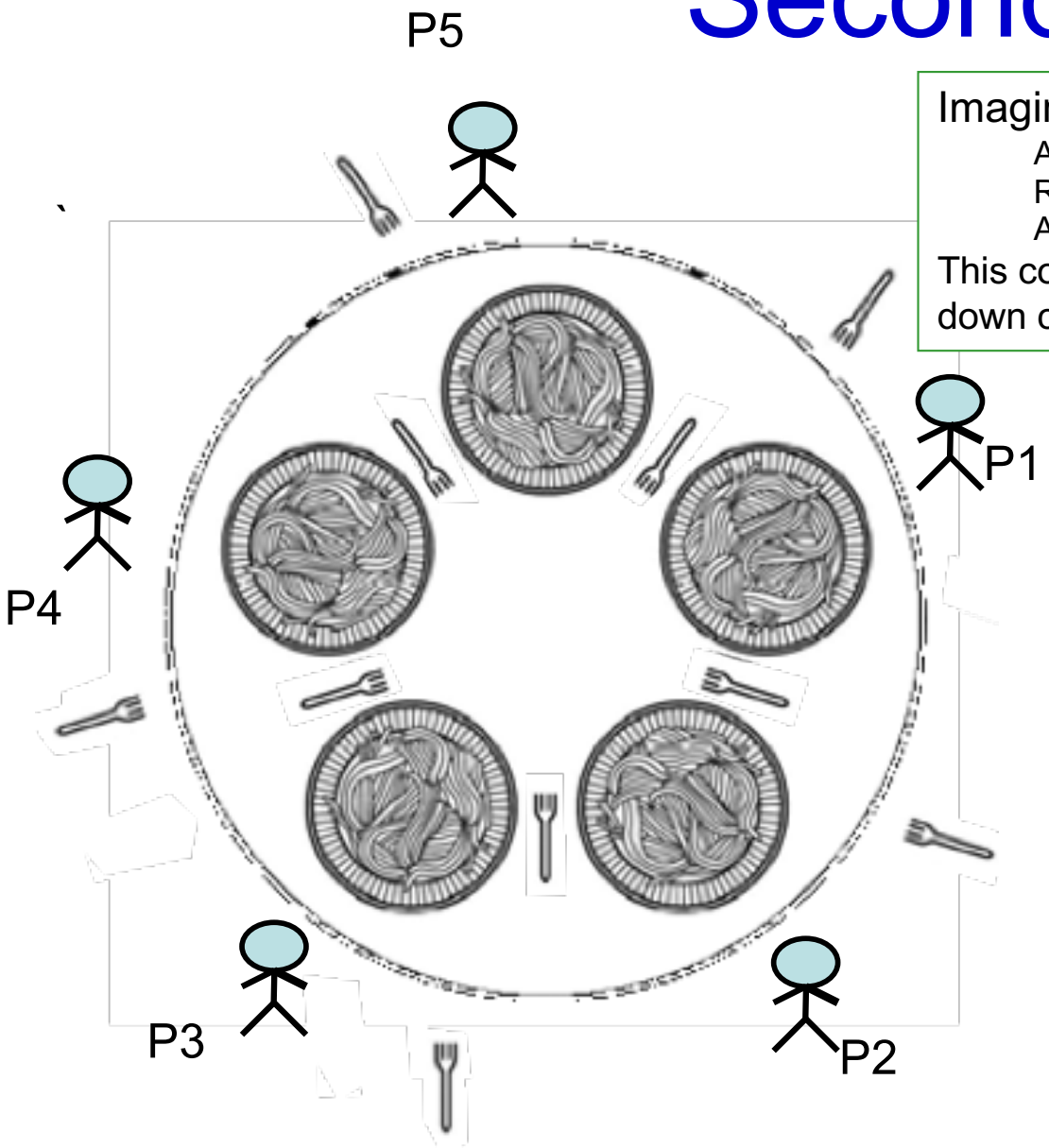

Imagine,

- All philosophers start at the same time
- Run simultaneously
- And think for the same time

This could lead to philosophers taking fork and putting it down continuously. a deadlock.

```
while(TRUE){
    think();
    take_fork(Ri);
    if (available(Li){
      take_fork(Li);
      eat();
    }else{
       put_fork(Ri);
       sleep(T)
    }
}
```

# Second try



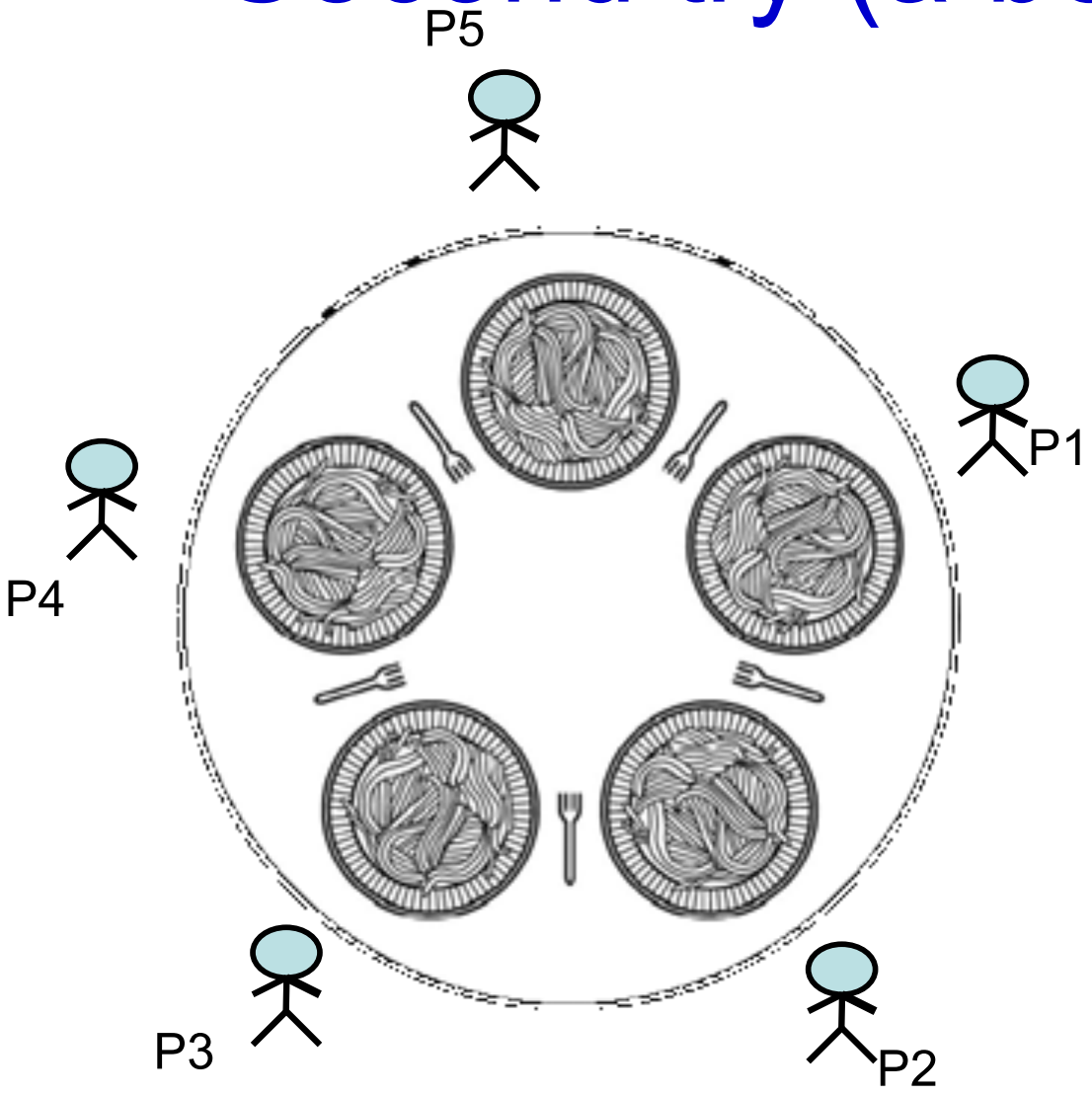

Imagine,

- All philosophers start at the same time
- Run simultaneously
- And think for the same time

This could lead to philosophers taking fork and putting it down continuously. a deadlock.

```
while(TRUE){
    think();
    take_fork(Ri);
    if (available(Li){
      take_fork(Li);
      eat();
    }else{
       put_fork(Ri);
       sleep(T);
    }
}
```

# Second try (a better solution)




```
#define N 5

void philosopher(int i){
  while(TRUE){
      think();
      take_fork(Ri);
      if (available(Li){
         take_fork(Li);
         eat();
      }else{
          put_fork(Ri);
          sleep(random_time);
      }
}
```

# Solution using Mutex

- Protect critical sections with a mutex
- Prevents deadlock
- But has performance issues
  - Only one philosopher can eat at a time

```
#define N 5

void philosopher(int i){
  while(TRUE){
      think(); // for some_time
      wait(mutex);
      take_fork(Ri);
      take_fork(Li);
      eat();
      put_fork(Ri);
      put_fork(Li);
      signal(mutex);
  }
}
```

# Solution with Semaphores

Uses N semaphores (s[1], s[2], …., s[N]) all initialized to 0, and a mutex
Philosopher has 3 states: HUNGRY, EATING, THINKING
*A philosopher can only move to EATING state if neither neighbor is eating*

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

# Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING){
           state[i] = EATING;
           up(s[i]);
    }
}
```

| | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|
| state | T | T | T | T | T |
| semaphore | 0 | 0 | 0 | 0 | 0 |

# Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

s[i] is 1, so down will not block.
The value of s[i] decrements by 1.

```
void test(int i){
    if (state[i] = HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

| | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|
| state | T | T | H | T | T |
| semaphore | 0 | 0 | 0 | 0 | 0 |

# Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

blocked

| | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|
| state | T | T | E | H | T |
| semaphore | 0 | 0 | 0 | 0 | 0 |

# Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

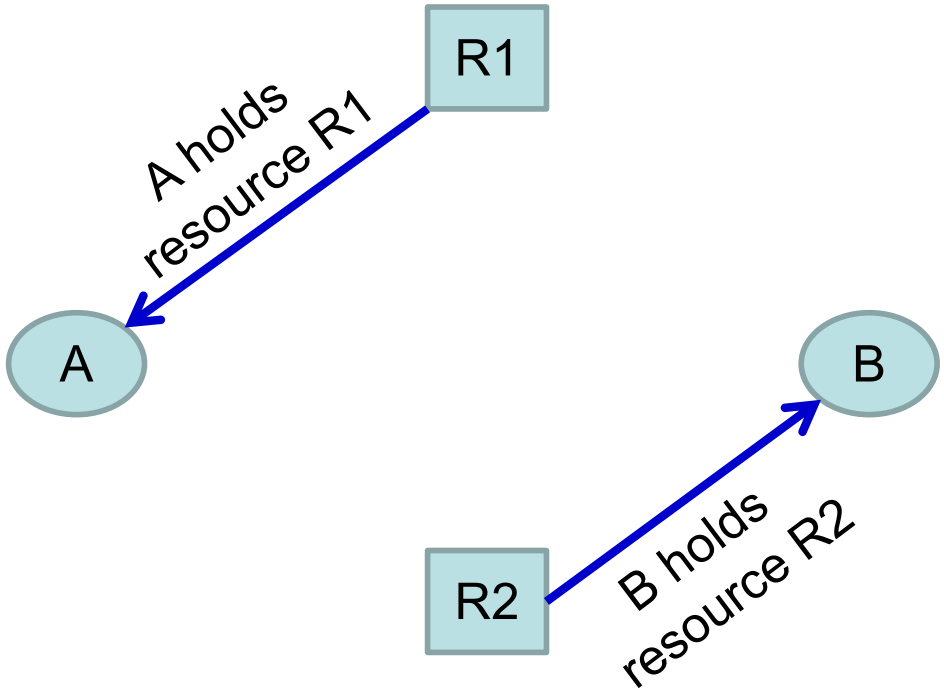
```
void test(int i){
    if (state[i] = HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

blocked

| | P1 | P2 | P3 | P4 | P5 |
|---|---|---|---|---|---|
| state | T | T | ~~E~~ | ~~H~~ | T |
| semaphore | 0 | 0 | 0 | 0 | 0 |

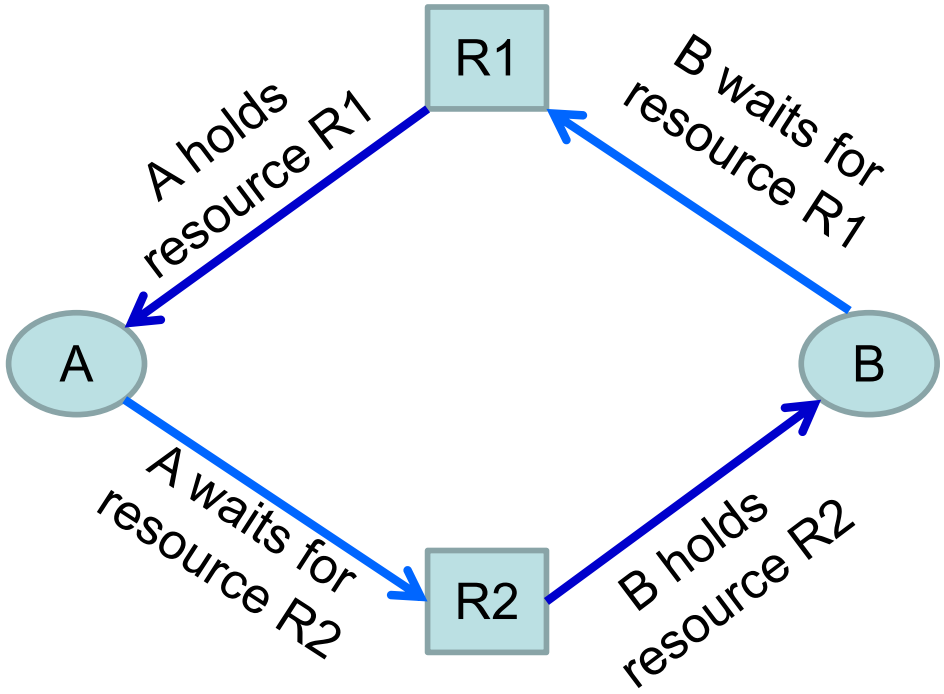
# Deadlocks

# Deadlocks




Consider this situation:

# Deadlocks

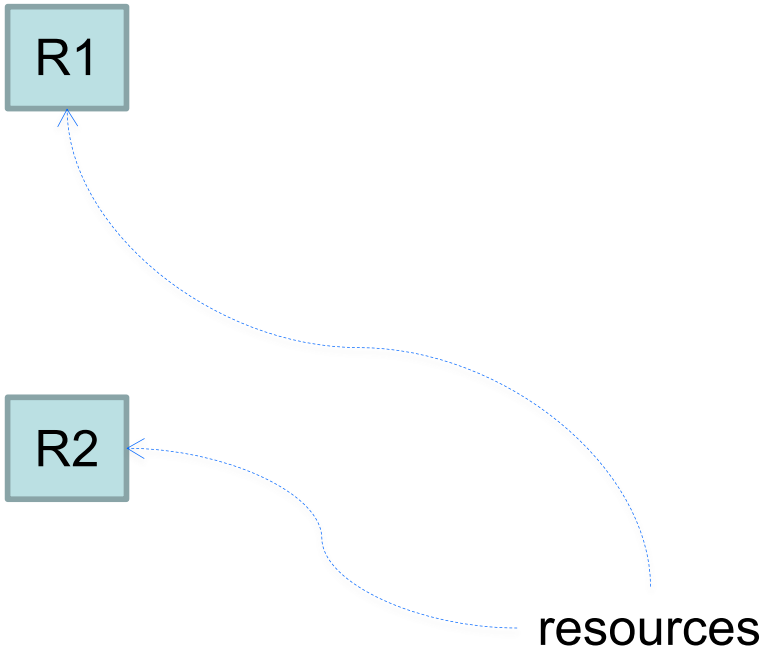



**A Deadlock Arises:**

Deadlock : A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

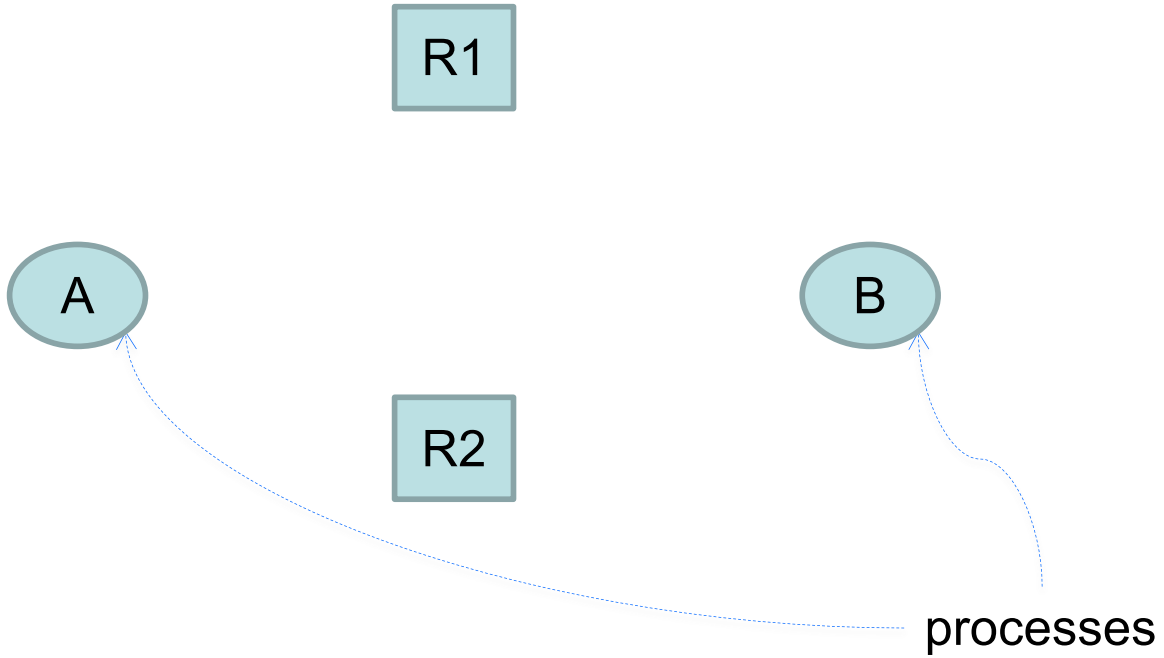
# Resource Allocation Graph

- It is a directed graph
- Used to model resource allocations



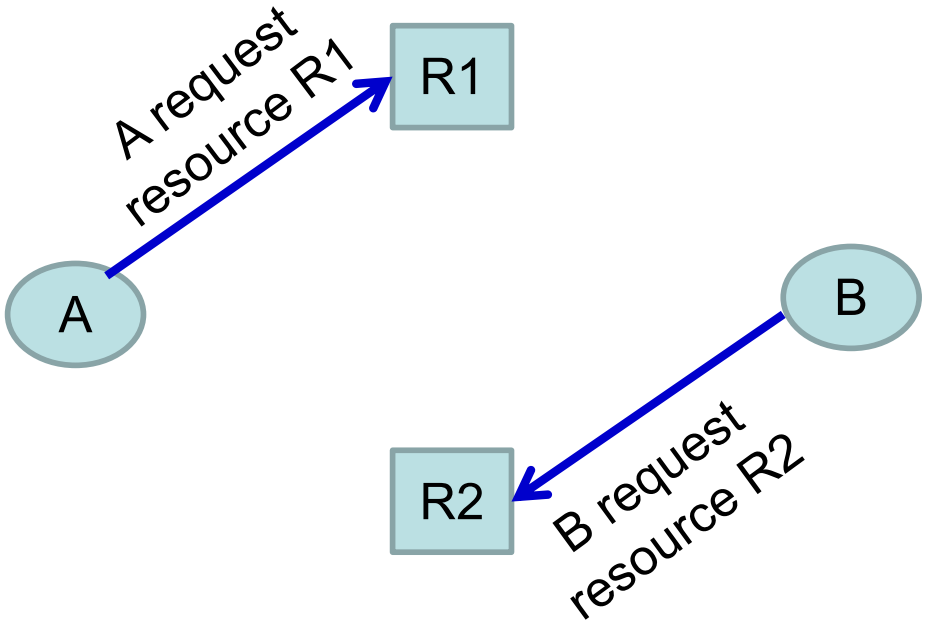
# Resource Allocation Graph

- It is a directed graph
- Used to model resource allocations



# Resource Allocation Graph

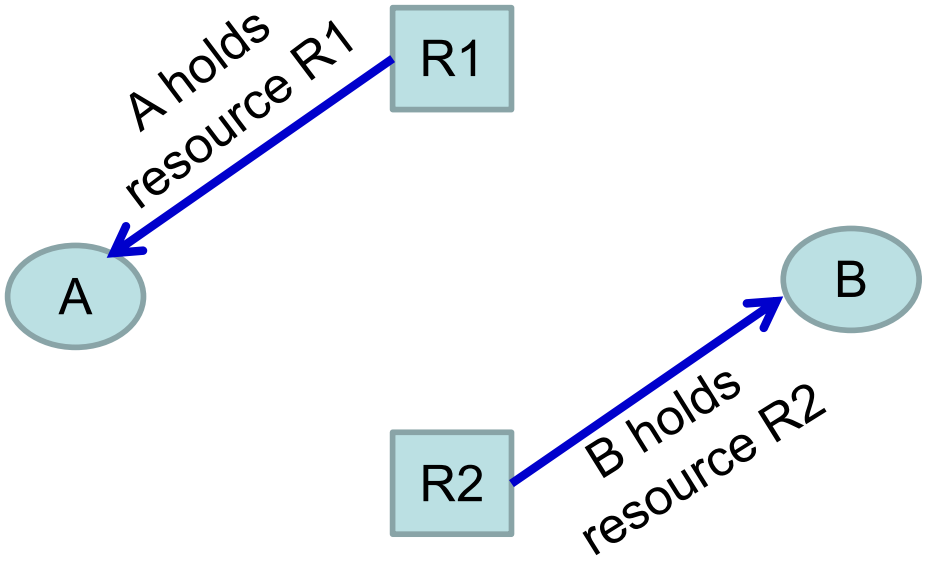
- It is a directed graph
- Used to model resource allocations




Resource requested by process

# Resource Allocation Graph

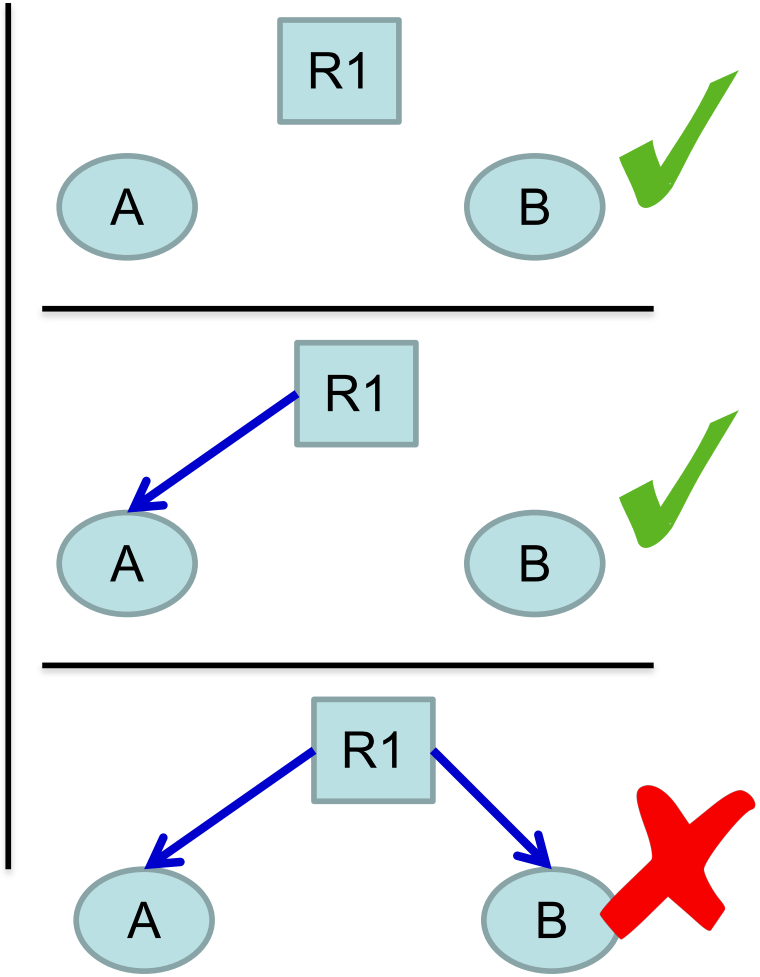
- It is a directed graph
- Used to model resource allocations




Resource held by process

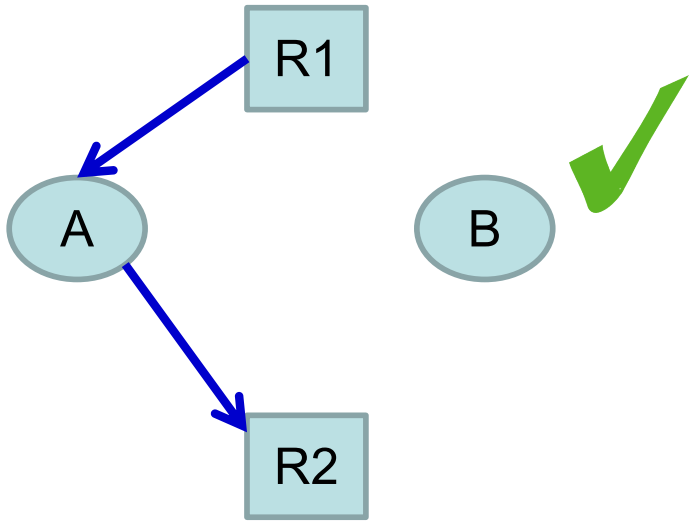
# Conditions for Resource Deadlocks

1. **Mutual Exclusion**
   - Each resource is either available or currently assigned to exactly one process
2. **Hold and wait**
   - A process holding a resource requests another resource
3. **No preemption**
   - Resources previously granted cannot be forcibly taken away from a process
   - They must be explicitly released by the process holding them
4. **Circular wait**
   - There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain



# Conditions for Resource Deadlocks

1. **Mutual Exclusion**
   - Each resource is either available or currently assigned to exactly one process
2. **Hold and wait**
   - A process holding a resource requests another resource
3. **No preemption**
   - Resources previously granted cannot be forcibly taken away from a process
   - They must be explicitly released by the process holding them
4. **Circular wait**
   - There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain

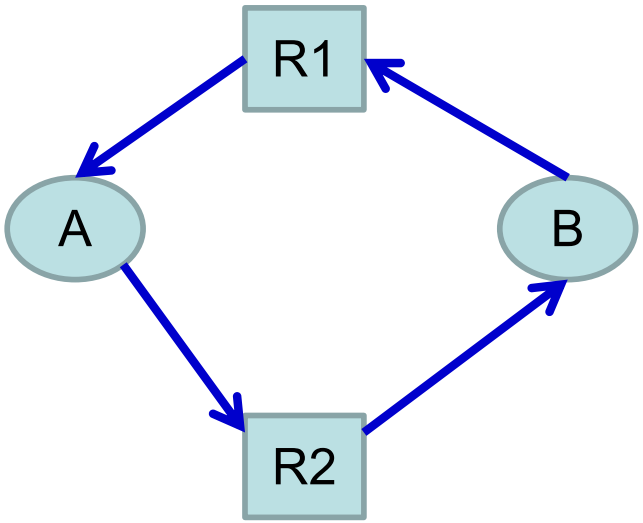


# Conditions for Resource Deadlocks

1. **Mutual Exclusion**
   - Each resource is either available or currently assigned to exactly one process
2. **Hold and wait**
   - A process holding a resource requests another resource
3. **No preemption**
   - Resources previously granted cannot be forcibly taken away from a process
   - They must be explicitly released by the process holding them
4. **Circular wait**
   - There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain

# Conditions for Resource Deadlocks

1. **Mutual Exclusion**
   - Each resource is either available or currently assigned to exactly one process
2. **Hold and wait**
   - A process holding a resource requests another resource
3. **No preemption**
   - Resources previously granted cannot be forcibly taken away from a process
   - They must be explicitly released by the process holding them
4. **Circular wait**
   - There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain



# Conditions for Resource Deadlocks

1. **Mutual Exclusion**
   - Each resource is either available or currently assigned to exactly one process
2. **Hold and wait**
   - A process holding a resource requests another resource
3. **No preemption**
   - Resources previously granted cannot be forcibly taken away from a process
   - They must be explicitly released by the process holding them
4. **Circular wait**
   - There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain

All four of these conditions must be present for a resource deadlock to occur!!

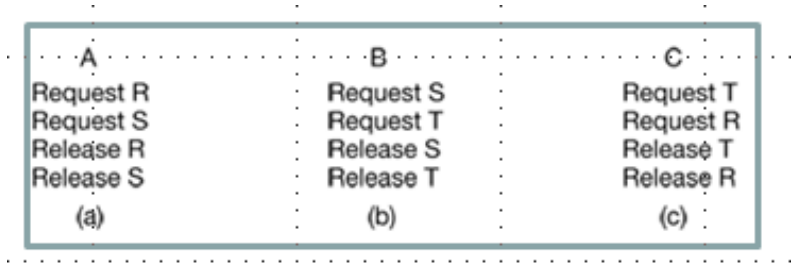
Coffman et al. 1971

# Deadlocks : (A Chanced Event)

- Ordering of resource requests and allocations are probabilistic, thus deadlock occurrence is also probabilistic

| A | B | C |
|---|---|---|
| Request R | Request S | Request T |
| Request S | Request T | Request R |
| Release R | Release S | Release T |
| Release S | Release T | Release R |
| (a) | (b) | (c) |

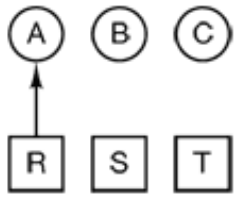
Deadlock occurs



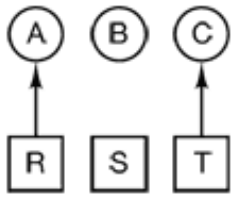

No dead lock occurrence
(B can be granted S
after step q)

1. A requests R
2. C requests T
3. A requests S
4. C requests R
5. A releases R
6. A releases S
   no deadlock

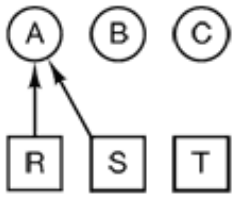
(k)



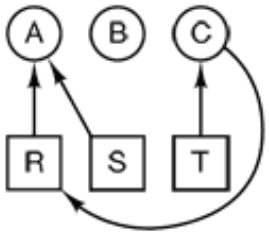
(l)



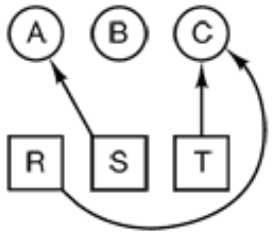
(m)



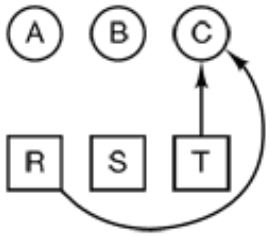
(n)



(o)



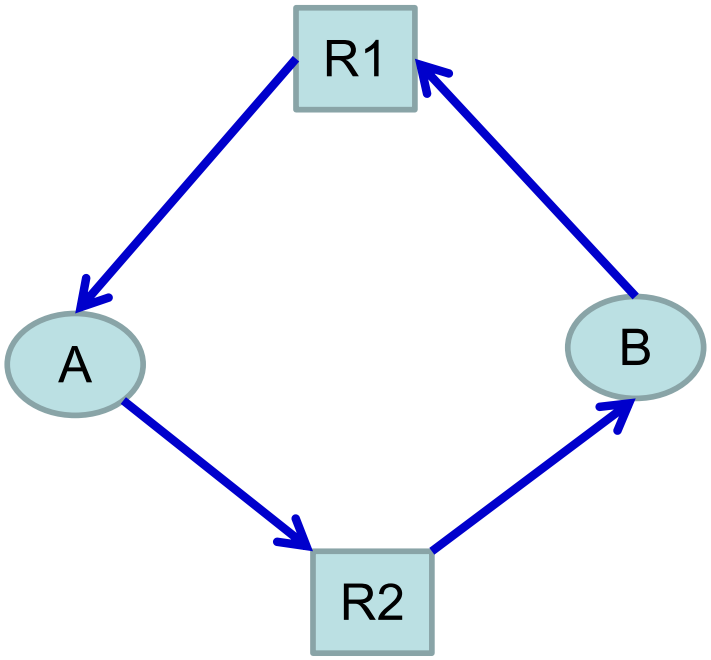
(p)



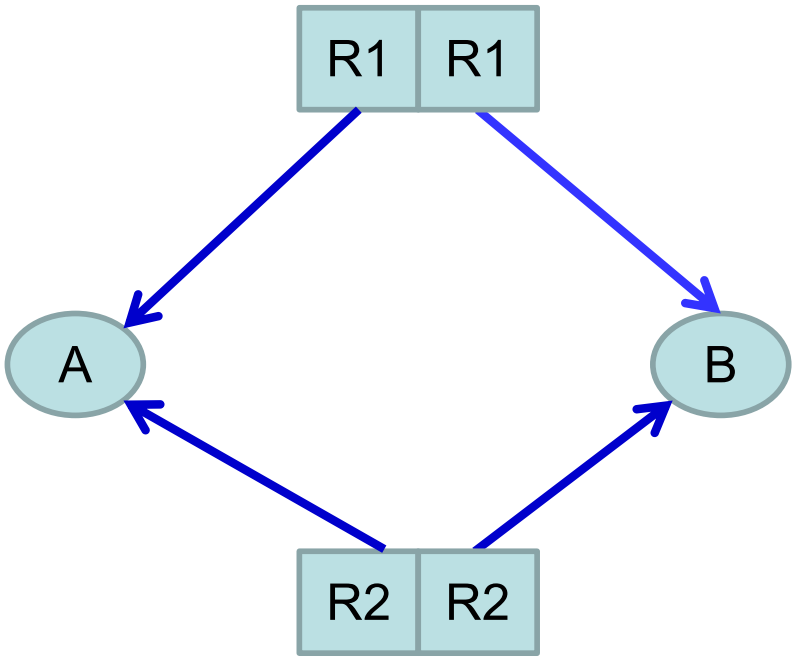
(q)

# Multiple Resources

- Having multiple resources can potentially reduce the chance of having a deadlock




A deadlocked state




No Deadlocks

# Should Deadlocks be handled?

- Preventing / detecting deadlocks could be tedious
- Can we live without detecting / preventing deadlocks?
  - What is the probability of occurrence?
  - What are the consequences of a deadlock? (How critical is a deadlock?)

# Handling Deadlocks

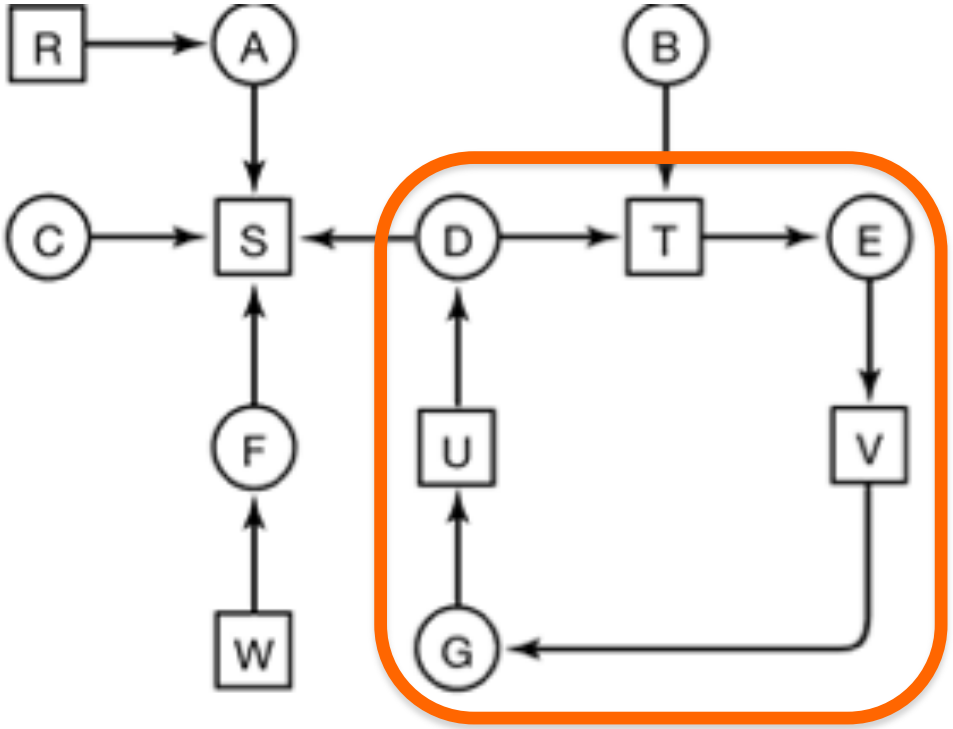
- Detection and Recovery
- Avoidance
- Prevention

# Deadlock Detection

- How can an OS detect when there is a deadlock?
- OS needs to keep track of
  - Current resource allocation
    - Which process has which resource
  - Current request allocation
    - Which process is waiting for which resource
- Use this information to detect deadlocks

# Deadlock Detection

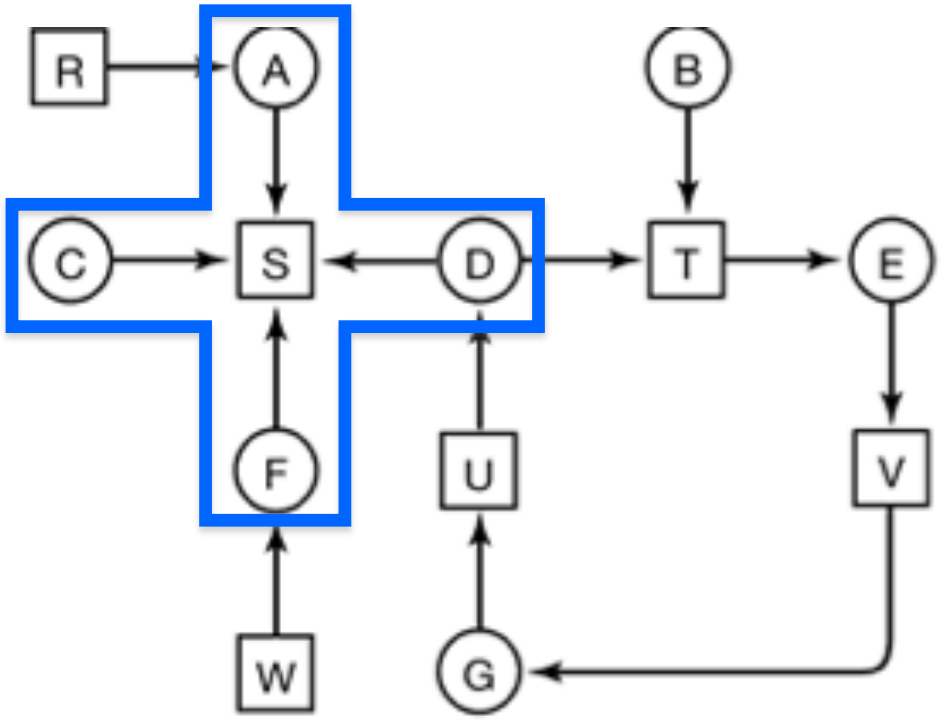
- Deadlock detection with **one resource of each type**
- Find cycles in resource graph




Dead locked

# Deadlock Detection

- Deadlock detection with **one resource of each type**
- There should be atleast one sequence of resource allocation, to avoid a deadlock




Not a Dead lock as an allocation sequence possible

**Sample Allocation Sequence**

S allocated to A, after A completes S allocated to C then, S allocated to F, and finally S allocated to D

# Deadlock Detection

- Deadlock detection with multiple resources of each type

Tape drives Plotters Scanners CD Roms

E = ( 4 2 3 1 )

**Existing Resource Vector**

Tape drives Plotters Scanners CD Roms

A = ( 2 1 0 0 )

**Resources Available**

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

$P_1$
$P_2$
$P_3$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

**Current Allocation Matrix**

*Who has what!!*

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

**Request Matrix**

*Who is waiting for what!!*

Process $P_i$ holds $C_i$ resources and requests $R_i$ resources, where i = 1 to 3
Goal is to check if there is any sequence of allocations by which all current requests can be met. If so, there is no deadlock.

# Deadlock Detection

- Deadlock detection with multiple resources of each type

Tape drives Plotters Scanners CD Roms

E = ( 4 2 3 1 )

**Existing Resource Vector**

Tape drives Plotters Scanners CD Roms

A = ( 2 1 0 0 )

**Resources Available**

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

$P_1$
$P_2$
$P_3$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

**Current Allocation Matrix**

*Who has what!!*

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

**Request Matrix**

*Who is waiting for what!!*

Process $P_i$ holds $C_i$ resources and requests $R_i$ resources, where i = 1 to 3
Goal is to check if there is any sequence of allocations by which all current requests can be met. If so, there is no deadlock.

# Deadlock Detection

- Deadlock detection with multiple resources of each type

Tape drives Plotters Scanners CD Roms

E = ( 4 2 3 1 )

**Existing Resource Vector**

Tape drives Plotters Scanners CD Roms

A = ( 2 1 0 0 )

**Resources Available**

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

$P_1$
$P_2$
$P_3$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

**Current Allocation Matrix**

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

**Request Matrix**

$P_1$ cannot be satisfied

$P_2$ cannot be satisfied

$P_3$ can be satisfied

Process $P_i$ holds $C_i$ resources and requests $R_i$ resources, where i = 1 to 3

# Deadlock Detection

- Deadlock detection with multiple resources of each type

Tape drives Plotters Scanners CD Roms

E = ( 4 2 3 1 )

**Existing Resource Vector**

Tape drives Plotters Scanners CD Roms

A = ( 2 1 0 0 )

**Resources Available**

$P_1$
$P_2$
$P_3$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

**Current Allocation Matrix**

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

**Request Matrix**

$P_3$ runs and its allocation is (2, 2, 2, 0)
On completion it returns the available resources : A = (4 2 2 1)
Either $P_1$ or $P_2$ can now run.
**NO Deadlock!!!**

# Deadlock Detection

- Deadlock detection with multiple resources of each type

Tape drives Plotters Scanners CD Roms

E = ( 4 2 3 1 )

**Existing Resource Vector**

Tape drives Plotters Scanners CD Roms

A = ( 2 1 0 0 )

**Resources Available**

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

$P_1$
$P_2$
$P_3$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

**Current Allocation Matrix**

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 1 & 0 \end{bmatrix}$$

**Request Matrix**

$P_1$ cannot be satisfied

$P_2$ cannot be satisfied

$P_3$ cannot be satisfied

**deadlock**

Process $P_i$ holds $C_i$ resources and requests $R_i$ resources, where i = 1 to 3

Deadlock detected as none of the requests can be satisfied

# Deadlock Detection

- Deadlock detection with multiple resources of each type

Tape drives Plotters Scanners CD Roms

E = ( 4 2 3 1 )

**Existing Resource Vector**

Tape drives Plotters Scanners CD Roms

A = ( 2 1 0 0 )

**Resources Available**

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

$P_1$
$P_2$
$P_3$

Current allocation matrix

$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ 0 & 1 & 2 & 0 \end{bmatrix}$$

**Current Allocation Matrix**

Request matrix

$$R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 2 & 1 & 0 & 0 \end{bmatrix}$$

**Request Matrix**

$P_1$ cannot be satisfied

$P_2$ cannot be satisfied

$P_3$ can be satisfied

**No deadlock**

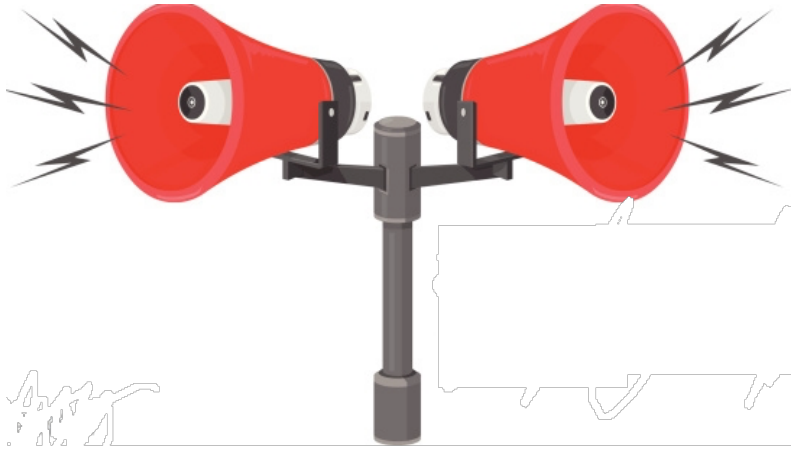
Process $P_i$ holds $C_i$ resources and requests $R_i$ resources, where i = 1 to 3

Deadlock not present since requests can be satisfied

# Deadlock Recovery

What should the OS do when it detects a deadlock?

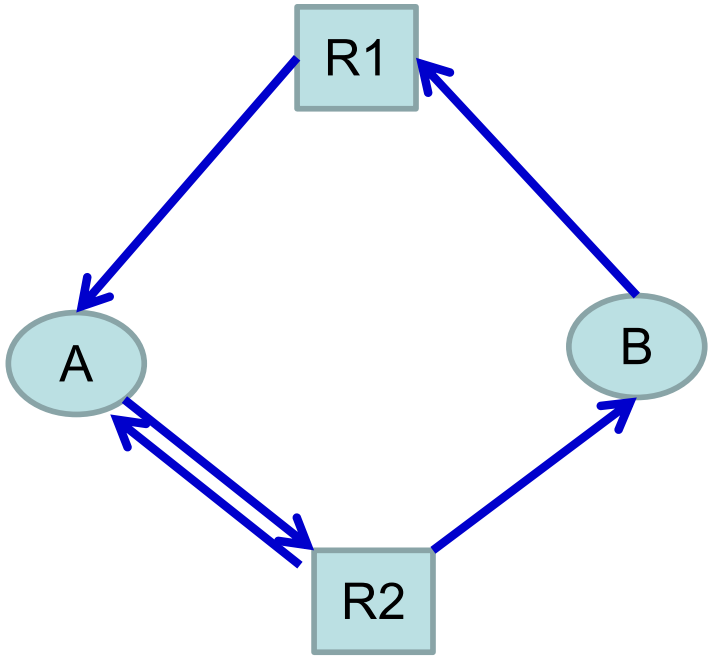
- Raise an alarm
  - Tell users and administrator
- Preemption
  - Take away a resource temporarily (frequently not possible)
- Rollback
  - Checkpoint states and then rollback
- Kill process
  - Keep killing processes until deadlock is broken



# Deadlock Recovery

What should the OS do when it detects a deadlock?

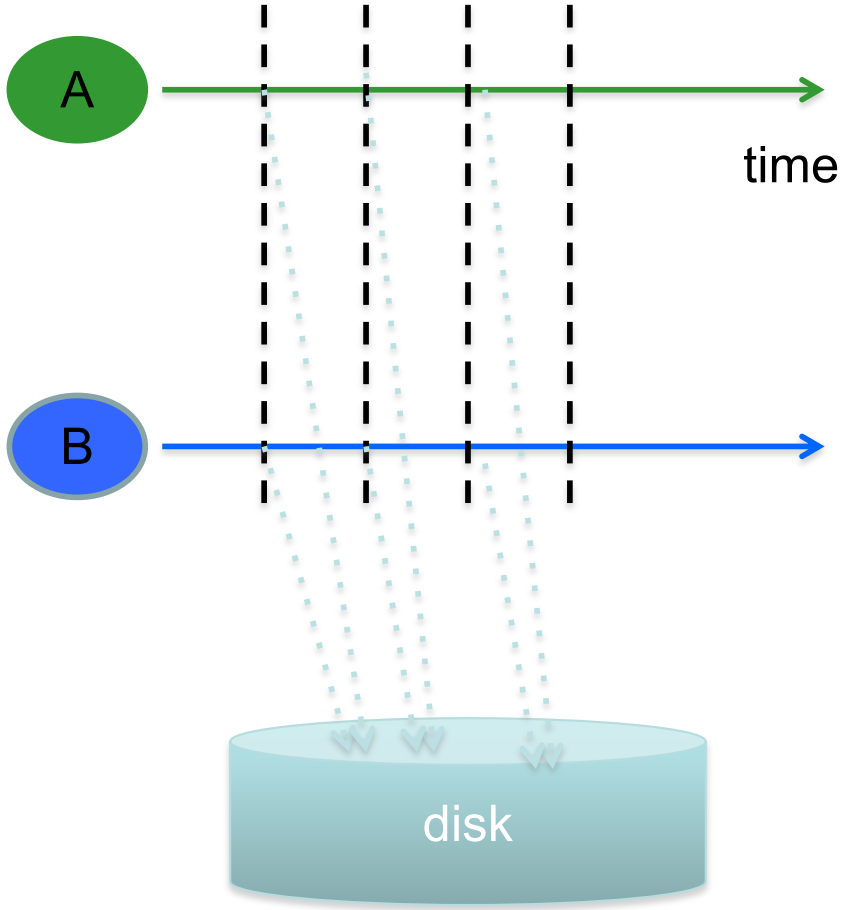
- Raise an alarm
  - Tell users and administrator
- Preemption
  - Take away a resource temporarily (frequently not possible)
- Rollback
  - Checkpoint states and then rollback
- Kill process
  - Keep killing processes until deadlock is broken



# Deadlock Recovery

What should the OS do when it detects a deadlock?

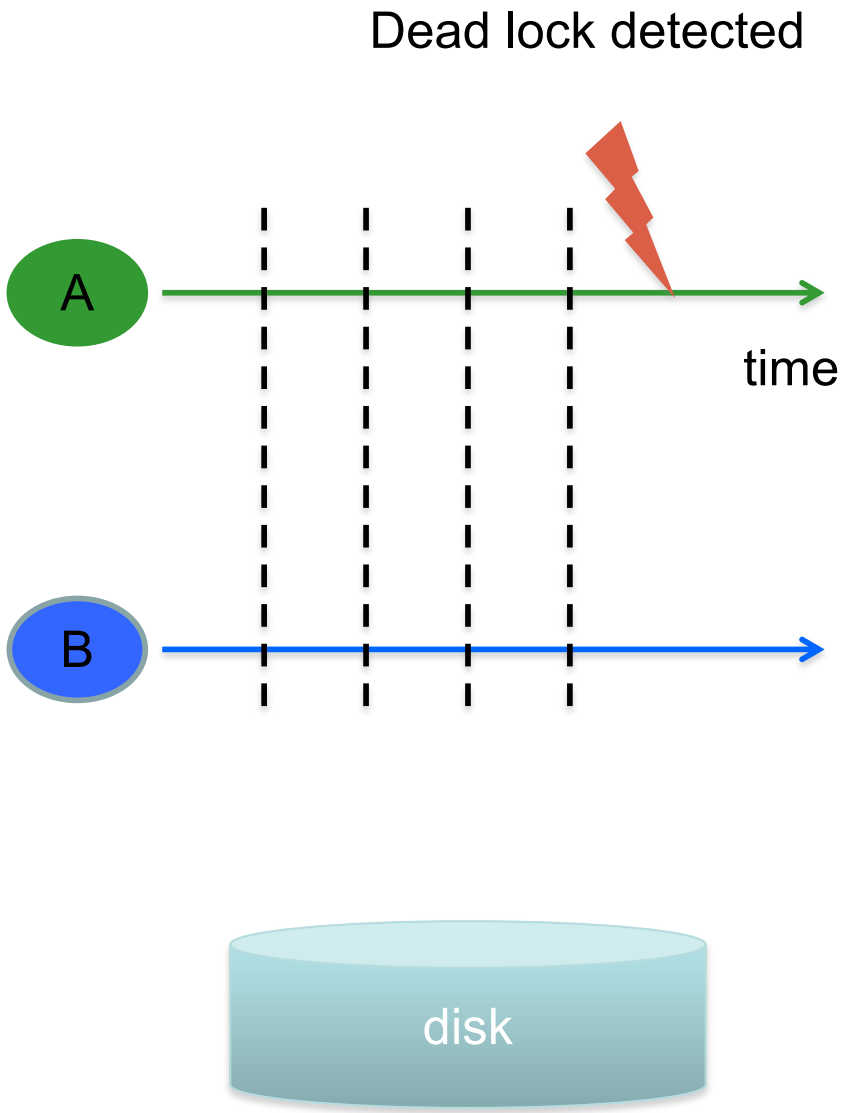
- Raise an alarm
  - Tell users and administrator
- Preemption
  - Take away a resource temporarily (frequently not possible)
- Rollback
  - Checkpoint states and then rollback
- Kill process
  - Keep killing processes until deadlock is broken



# Deadlock Recovery

What should the OS do when it detects a deadlock?

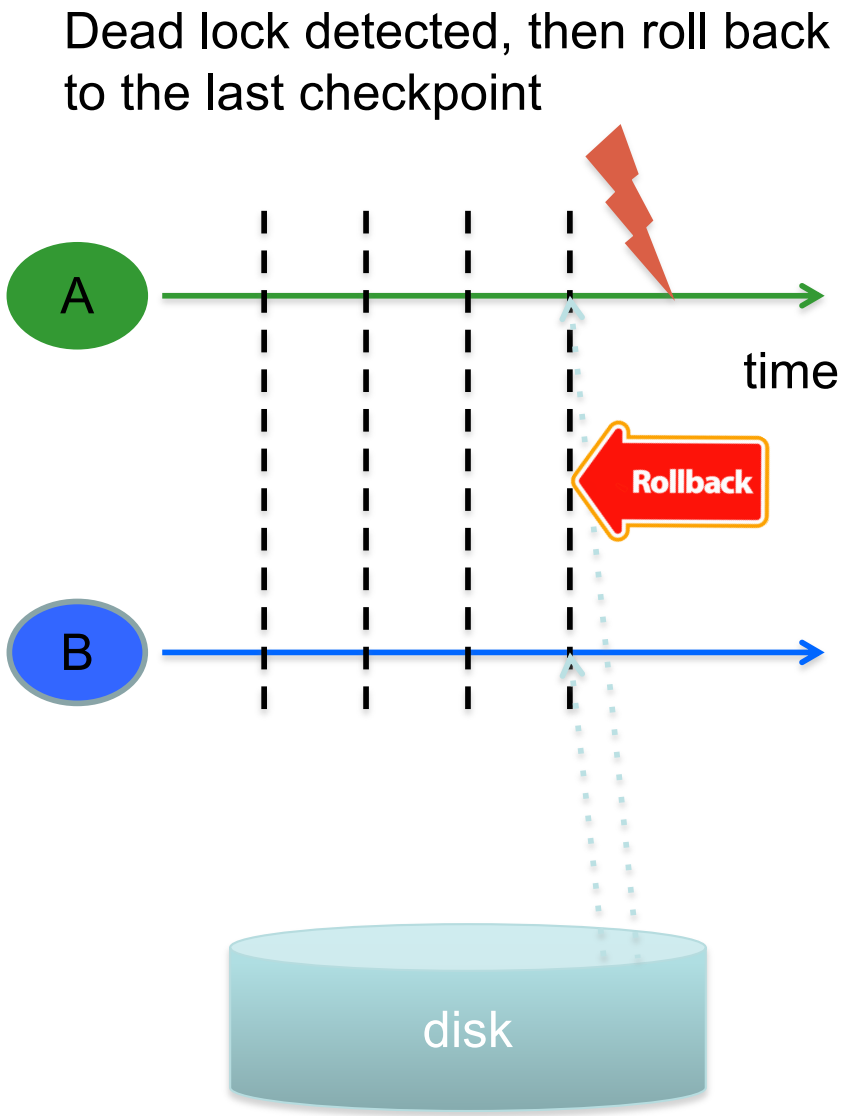
- Raise an alarm
  - Tell users and administrator
- Preemption
  - Take away a resource temporarily (frequently not possible)
- Rollback
  - Checkpoint states and then rollback
- Kill process
  - Keep killing processes until deadlock is broken



# Deadlock Recovery

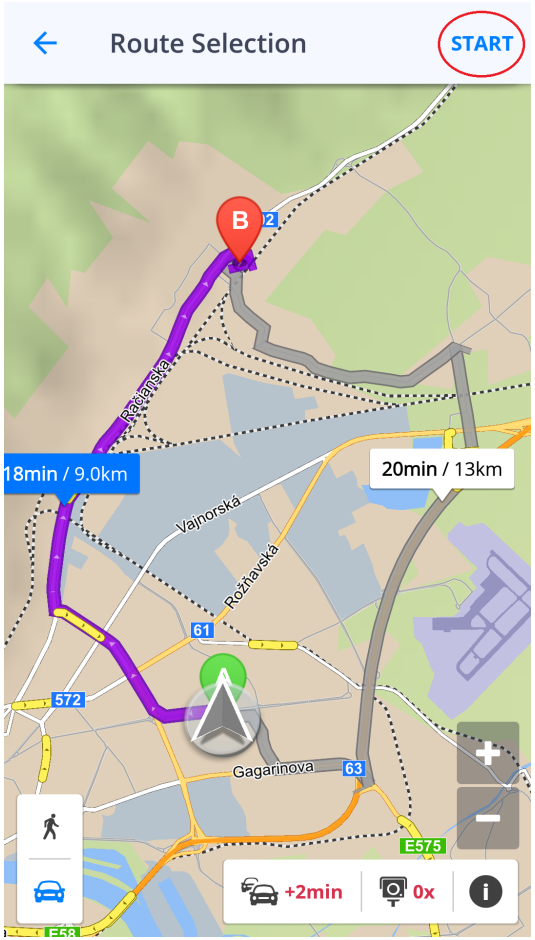
What should the OS do when it detects a deadlock?

- Raise an alarm
  - Tell users and administrator
- Preemption
  - Take away a resource temporarily (frequently not possible)
- Rollback
  - Checkpoint states and then rollback
- Kill process
  - Keep killing processes until deadlock is broken



# Deadlock Avoidance

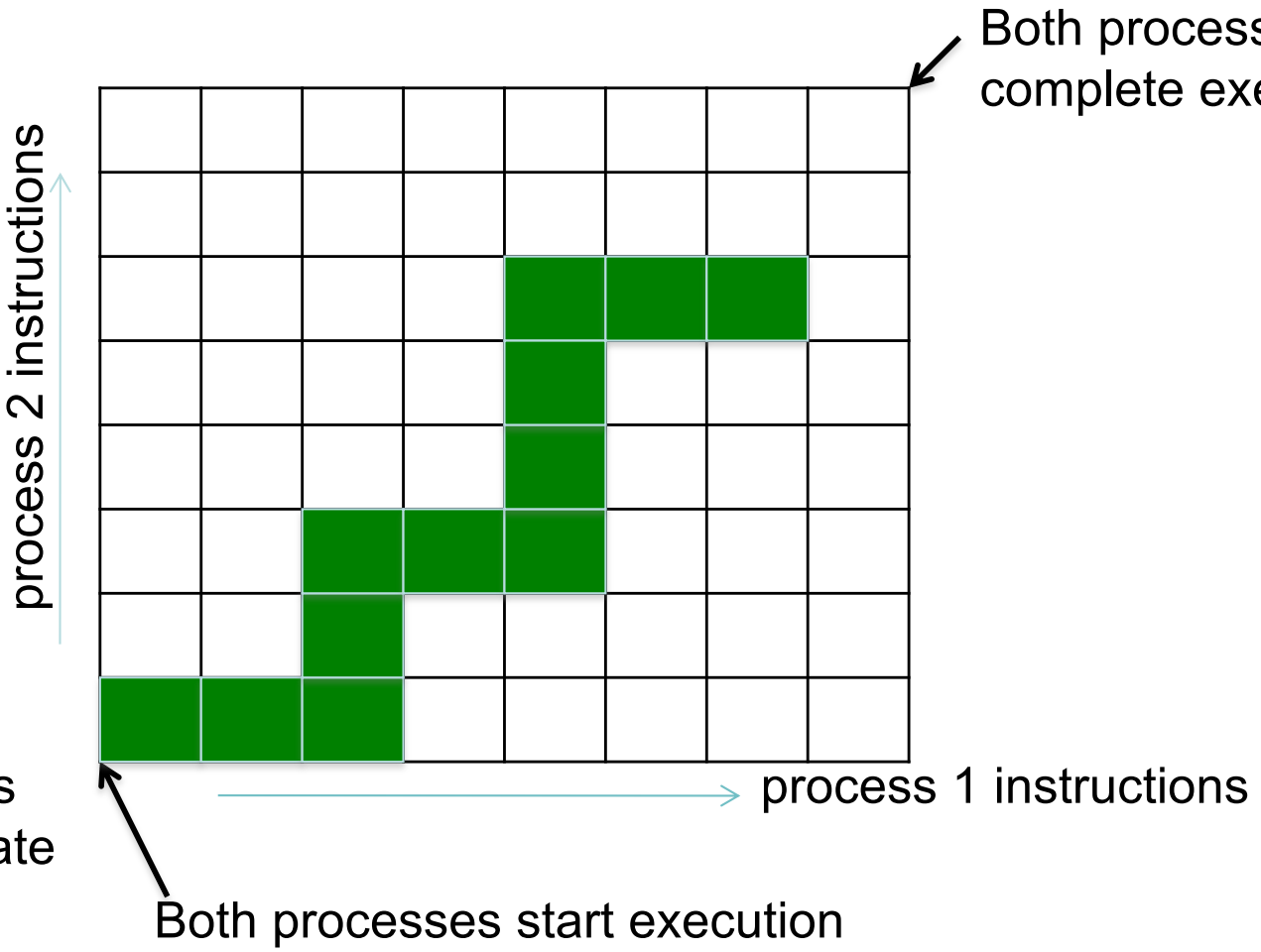
- Basic idea, always make the right choice!



# Deadlock Avoidance

- System decides in advance if allocating a resource to a process will lead to a deadlock

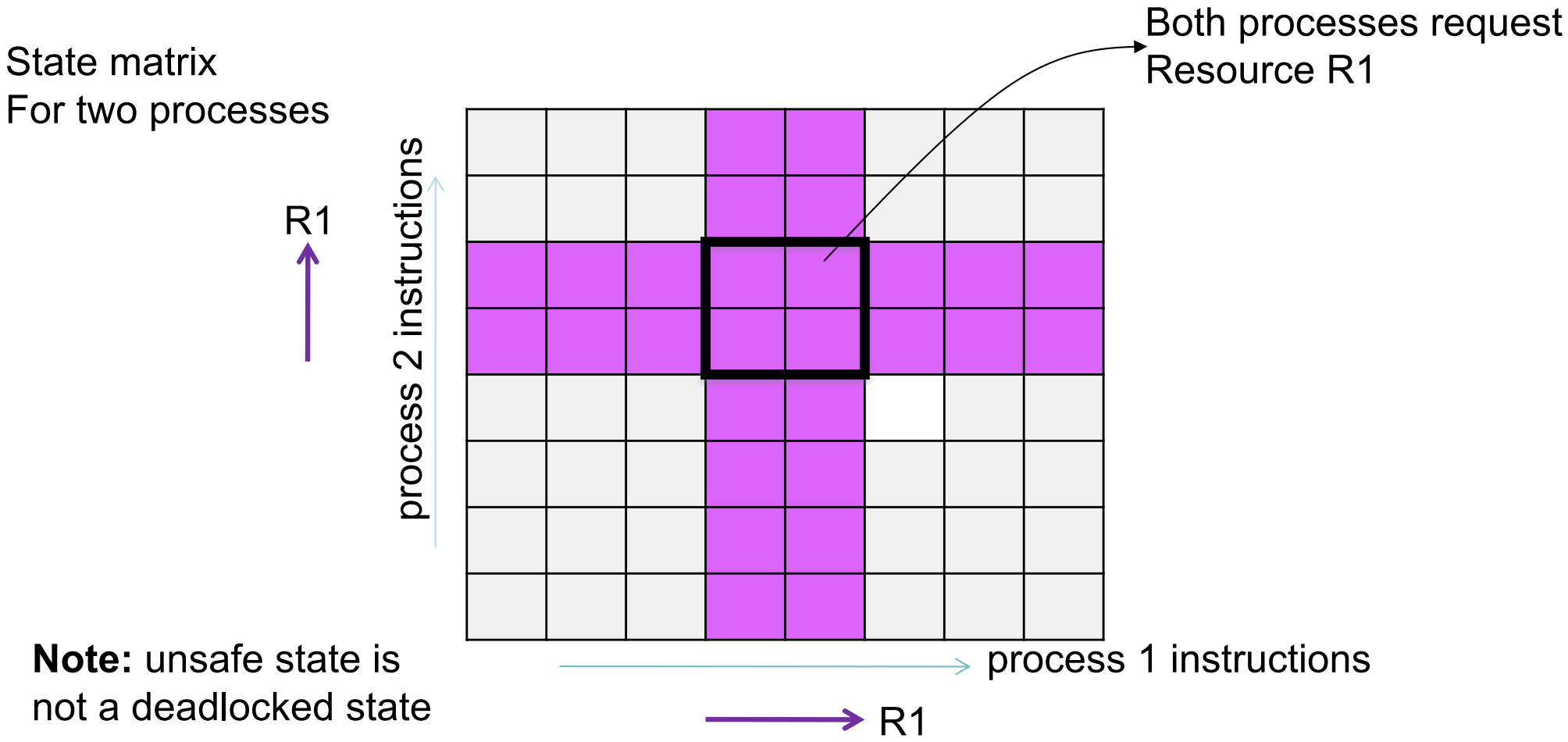
State matrix
For two processes




**Note:** unsafe state is not a deadlocked state

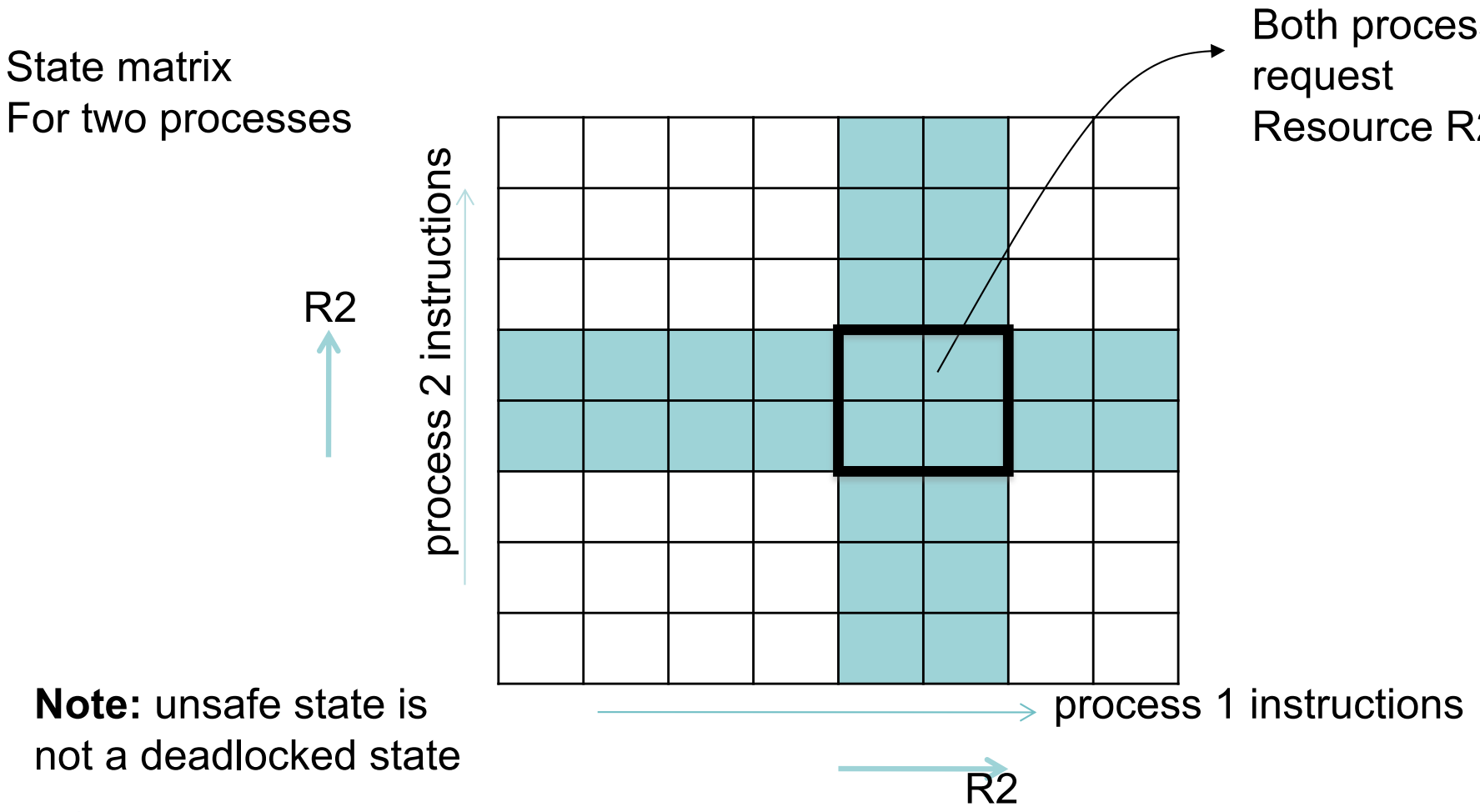
# Deadlock Avoidance

- State matrix for two processes



# Deadlock Avoidance

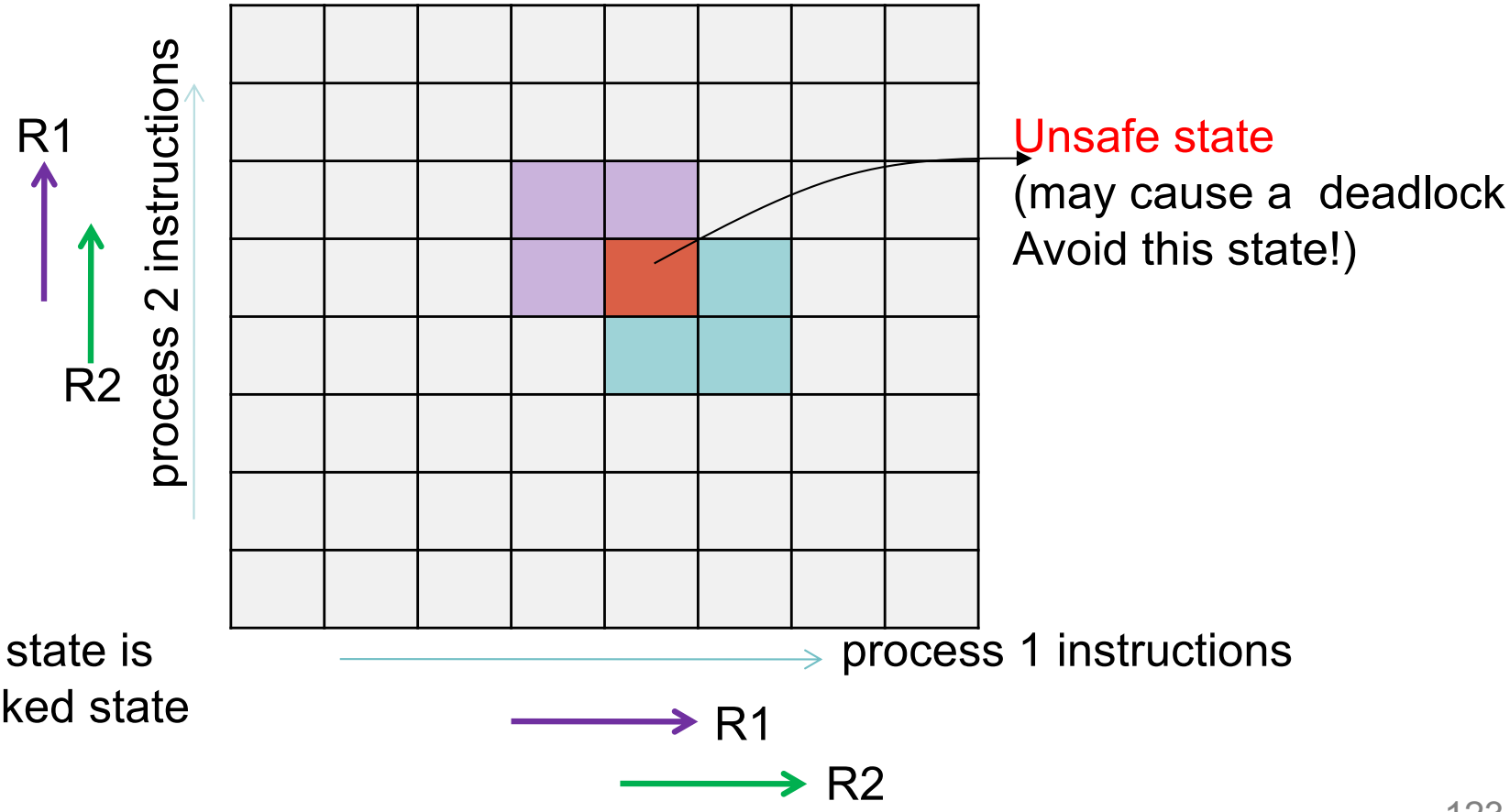
- State matrix for two processes




**Note:** unsafe state is not a deadlocked state

# Deadlock Avoidance

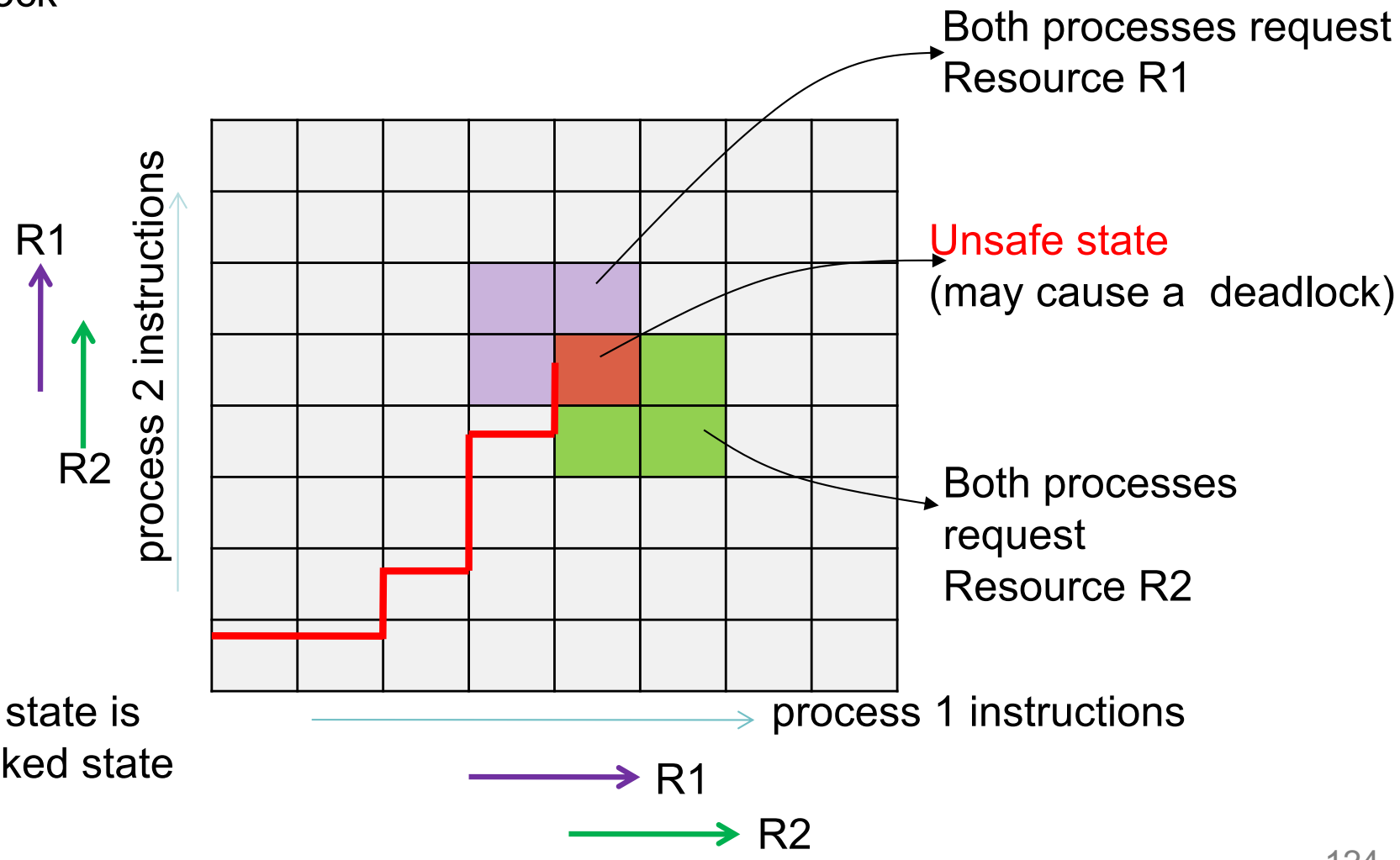
- System decides in advance if allocating a resource to a process will lead to a deadlock




**Note:** unsafe state is not a deadlocked state

# Deadlock Avoidance

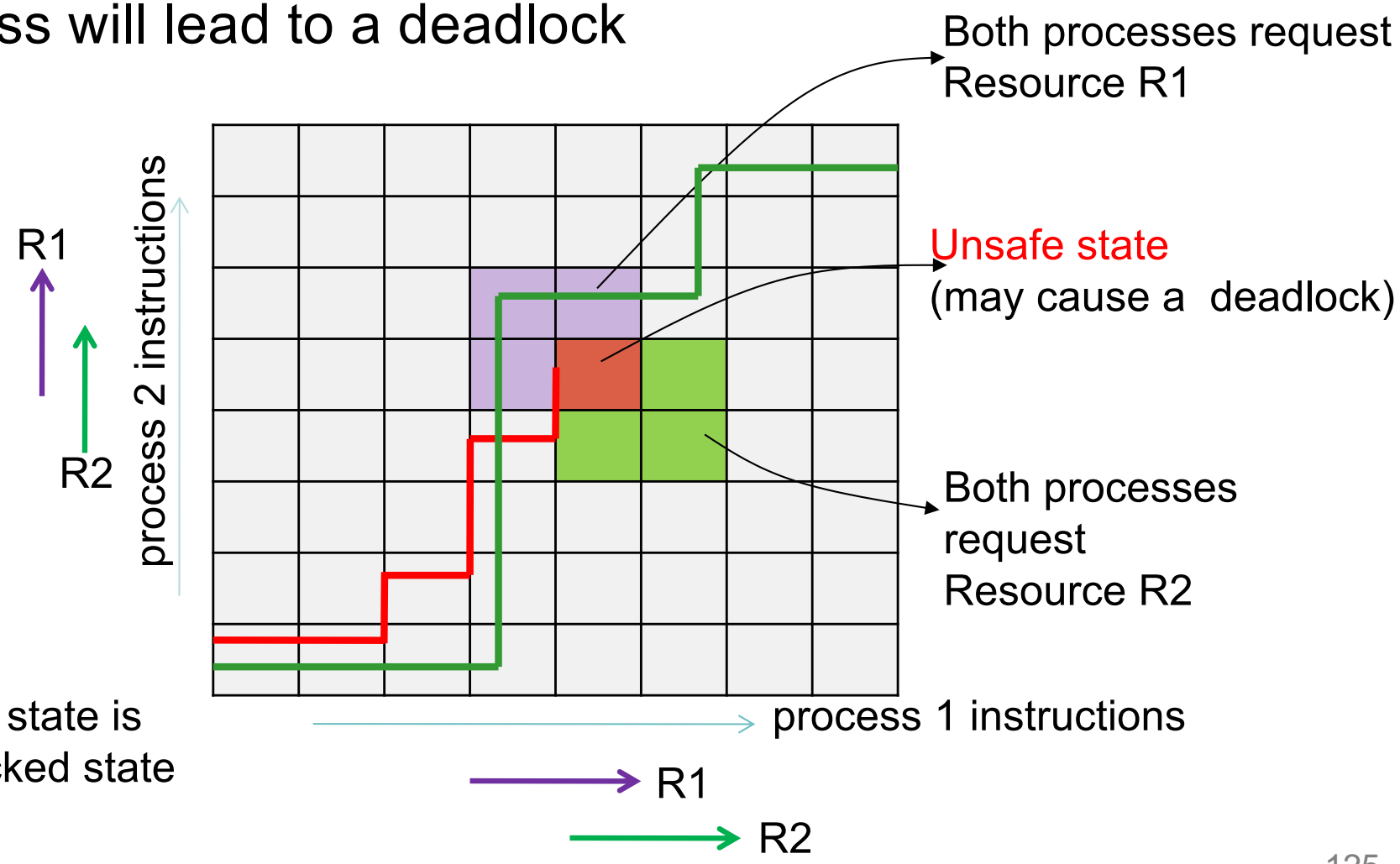
- System decides in advance if allocating a resource to a process will lead to a deadlock




**Note:** unsafe state is not a deadlocked state

# Deadlock Avoidance

- System decides in advance if allocating a resource to a process will lead to a deadlock




**Note:** unsafe state is not a deadlocked state

# Deadlock Avoidance

Is there an algorithm that can always avoid deadlocks by conservatively making the right choice.

- Ensures system never reaches an unsafe state

- Safe state : A state is said to be safe, if there is some scheduling order in which every process can run to completion even if all of them suddenly requests their maximum number of resources immediately
- An unsafe state does not have to lead to a deadlock; *it **could** lead to a deadlock*

# Example with a Banker

- Consider a banker with 4 clients ($P_1$, $P_2$, $P_3$, $P_4$).
  - Each client has certain credit limits (totaling 20 units)
  - The banker knows that max credits will not be used at once, so he keeps only 10 units

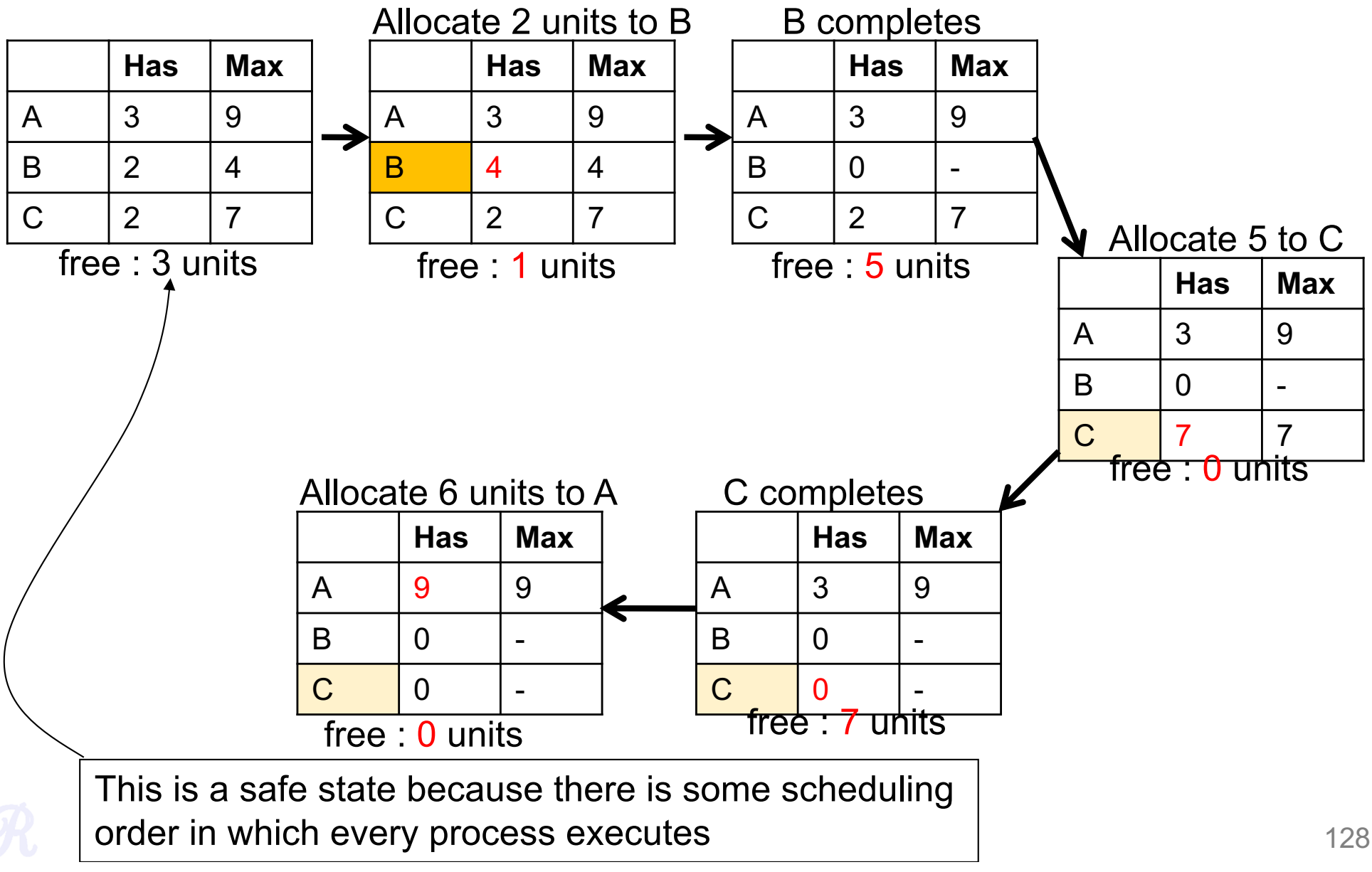
| | Has | Max |
|---|---|---|
| A | 3 | 9 |
| B | 2 | 4 |
| C | 2 | 7 |

free : 3 units

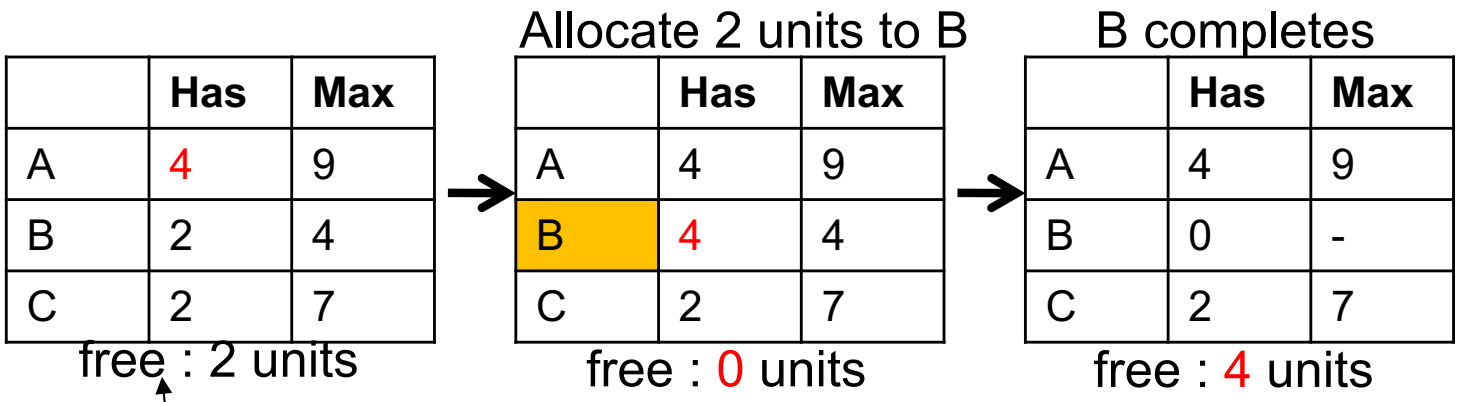
Total : 10 units

  - Clients declare maximum credits in advance. The banker can allocate credits provided no unsafe state is reached.

# Safe State


Allocate 2 units to B
B completes
Allocate 5 to C
Allocate 6 units to A
C completes
Has
Max
A 3 9
B 2 4
C 2 7
free : 3 units
A 3 9
B 4 4
C 2 7
free : 1 units
A 3 9
B 0 -
C 2 7
free : 5 units
A 3 9
B 0 -
C 7 7
free : 0 units
A 3 9
B 0 -
C 0 -
free : 7 units
A 9 9
B 0 -
C 0 -
free : 0 units
This is a safe state because there is some scheduling order in which every process executes

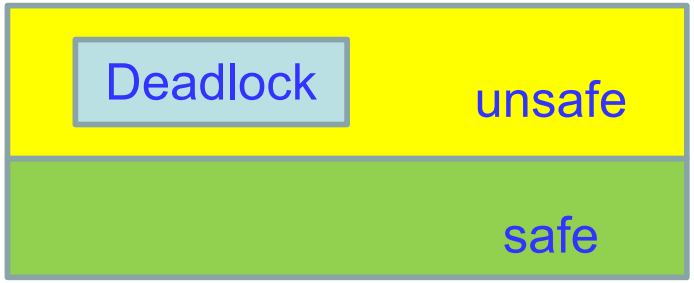
# Unsafe State



# Banker's Algorithm (with a single resource)

When a request occurs

- If(**is_system_in_a_safe_state**)
  - Grant request
- else
  - postpone until later




Please read Banker's Algorithm with multiple resources from Modern Operating Systems, Tanenbaum

# Deadlock Prevention

- Deadlock avoidance not practical, need to know maximum requests of a process
- Deadlock prevention
  - Prevent at-least one of the 4 conditions
    1. **Mutual Exclusion**
    2. **Hold and wait**
    3. **No preemption**
    4. **Circular wait**

# Prevention

## 1. Preventing Mutual Exclusion

- Not feasible in practice
- But OS can ensure that resources are optimally allocated

## 2. Hold and wait

- One way is to achieve this is to require all processes to request resources before starting execution
  - May not lead to optimal usage
  - May not be feasible to know resource requirements

## 3. No preemption

- Pre-empt the resources, such as by virtualization of resources (eg. Printer spools)

## 4. Circular wait

- One way, process holding a resource cannot hold a resource and request for another one
- Ordering requests in a sequential / hierarchical order.

# Hierarchical Ordering of Resources

- Group resources into levels

  (i.e. prioritize resources numerically)
- A process may only request resources at higher levels than any resource it currently holds
- Resource may be released in any order
- eg.
  - Semaphore s1, s2, s3 (with priorities in increasing order)

    down(S1);  down(S2); down(S3) ; → allowed

    down(S1); down(S3); down(S2); →not allowed