Lecture 5
Synchronization

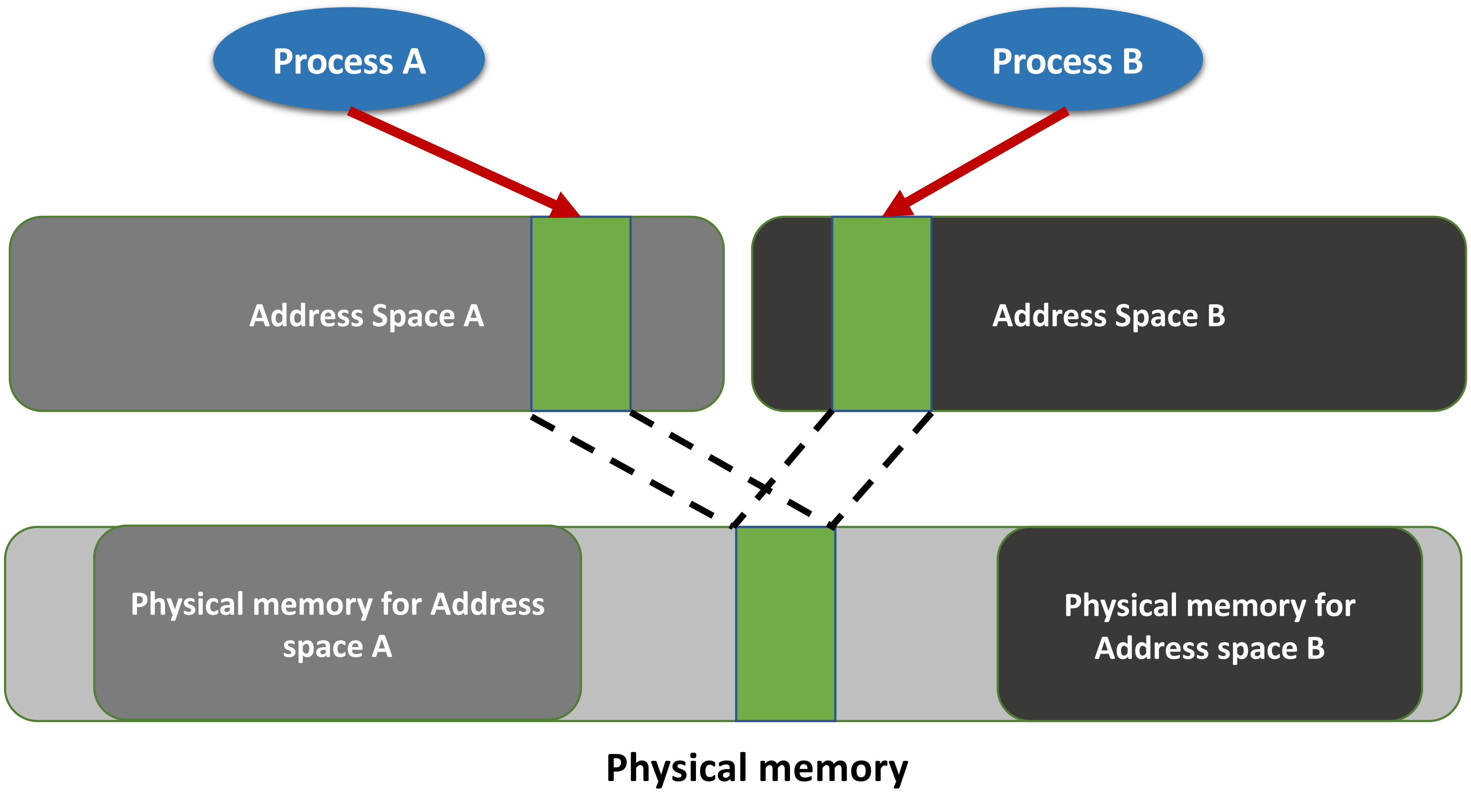
Prof. Yinqian Zhang

Spring 2024

Process Communication

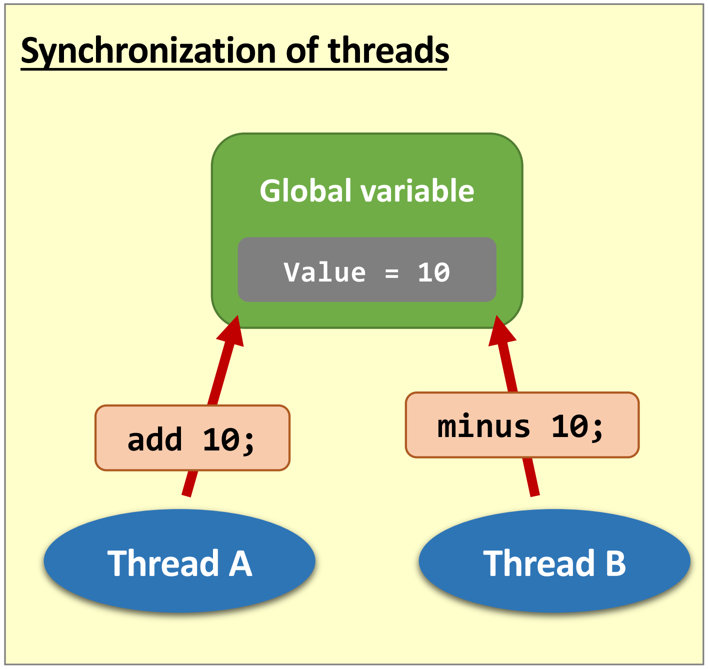
- Threads of the same process share the same address space
  - Global variables are shared by multiple threads
  - Communication between threads made easy
- Process may also need to communicate with each other
  - Information sharing:
    - e.g., sharing between Android apps
  - Computation speedup:
    - e.g., Message Passing Interface (MPI)
  - Modularity and isolation:
    - e.g., Chrome's multi-process architecture

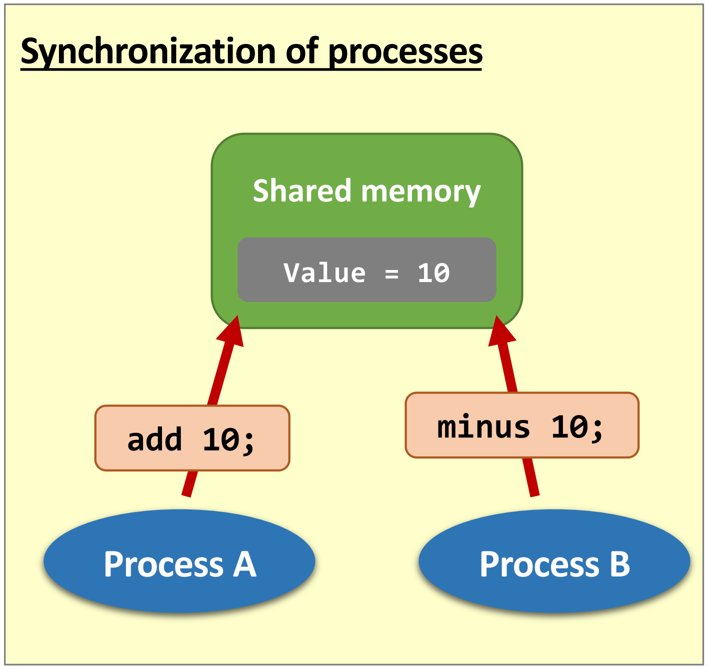
Shared Memory between Processes



Synchronization of Threads/Processes

Process and thread synchronization can be considered in similar way





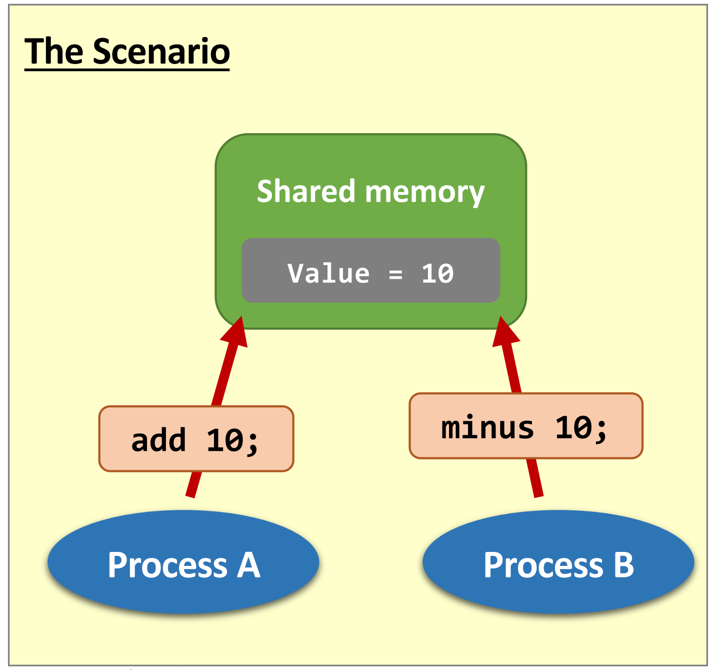
Synchronization of Threads/Processes

High-level language for Program A

```
1   attach to the shared memory X;
2   add 10 to X;
3   exit;
```

Partial low-level language for Program A

```
1     attach to the shared memory X;
......
2.1   load memory X to register A;
2.2   add 10 to register A;
2.3   write register A to memory X;
......
3     exit;
```



Don’t print

Race Condition

Shared memory - X

Value = 10

State: Ready

Register A
Value = 0

```
2.1   load memory X to
      register A;

2.2   add 10 to register A;

2.3   write register A to
      memory X;
```

Process A

Register B
Value = 0

State: Ready

```
2.1   load memory X to
      register B;

2.2   minus 10 from register B;

2.3   write register B to
      memory X;
```

Process B

The initial setting

Don't print

Normal Execution

Shared memory - X

Value = 10

State: Running

Register A
Value = 10

Register B
Value = 0

State: Ready

1

```
2.1   load memory X to
      register A;

2.2   add 10 to register A;

2.3   write register A to
      memory X;
```

```
2.1   load memory X to
      register B;

2.2   minus 10 from register B;

2.3   write register B to
      memory X;
```

Process A

Process B

Execution Flow #1, Step 1

Don't print

Normal Execution

Shared memory - X

Value = 10

Process A

State: Running

Register A Value = 20

1 2.1 load memory X to register A;

2 2.2 add 10 to register A;

2.3 write register A to memory X;

Process B

Register B Value = 0

State: Ready

2.1 load memory X to register B;

2.2 minus 10 from register B;

2.3 write register B to memory X;

Execution Flow #1, Step 2

Don't print

Normal Execution

Shared memory - X

Value = 20

State: Running

Register A Value = 20

1. 2.1 load memory X to register A;
2. 2.2 add 10 to register A;
3. 2.3 write register A to memory X;

Process A

Register B Value = 0

State: Ready

2.1 load memory X to register B;

2.2 minus 10 from register B;

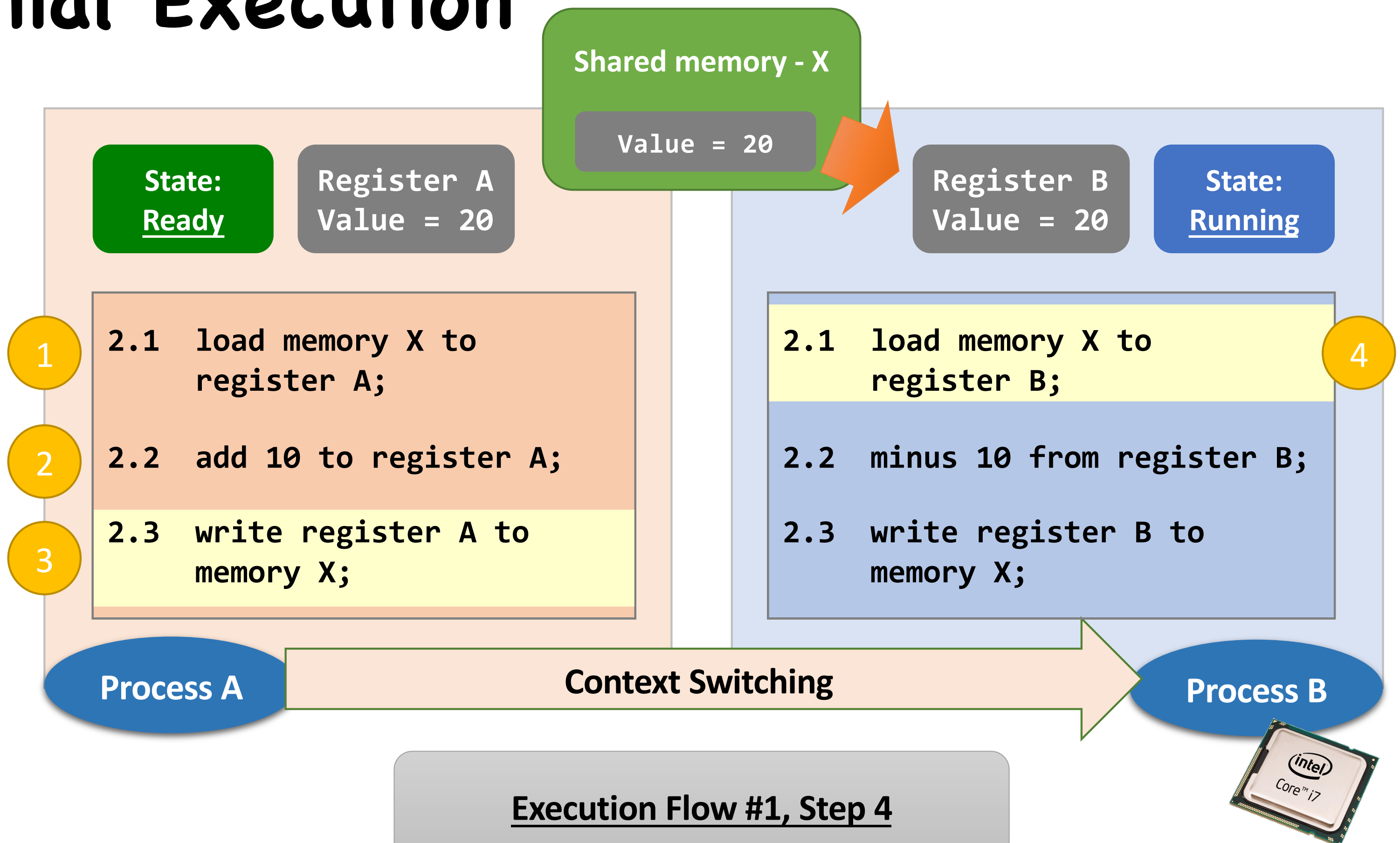
2.3 write register B to memory X;

Process B

Execution Flow #1, Step 3

Don't print

Normal Execution



Execution Flow #1, Step 4

Don't print

Normal Execution

Shared memory - X

Value = 20

Process A

State: Ready

Register A
Value = 20

1. 2.1 load memory X to register A;
2. 2.2 add 10 to register A;
3. 2.3 write register A to memory X;

Process B

Register B
Value = 10

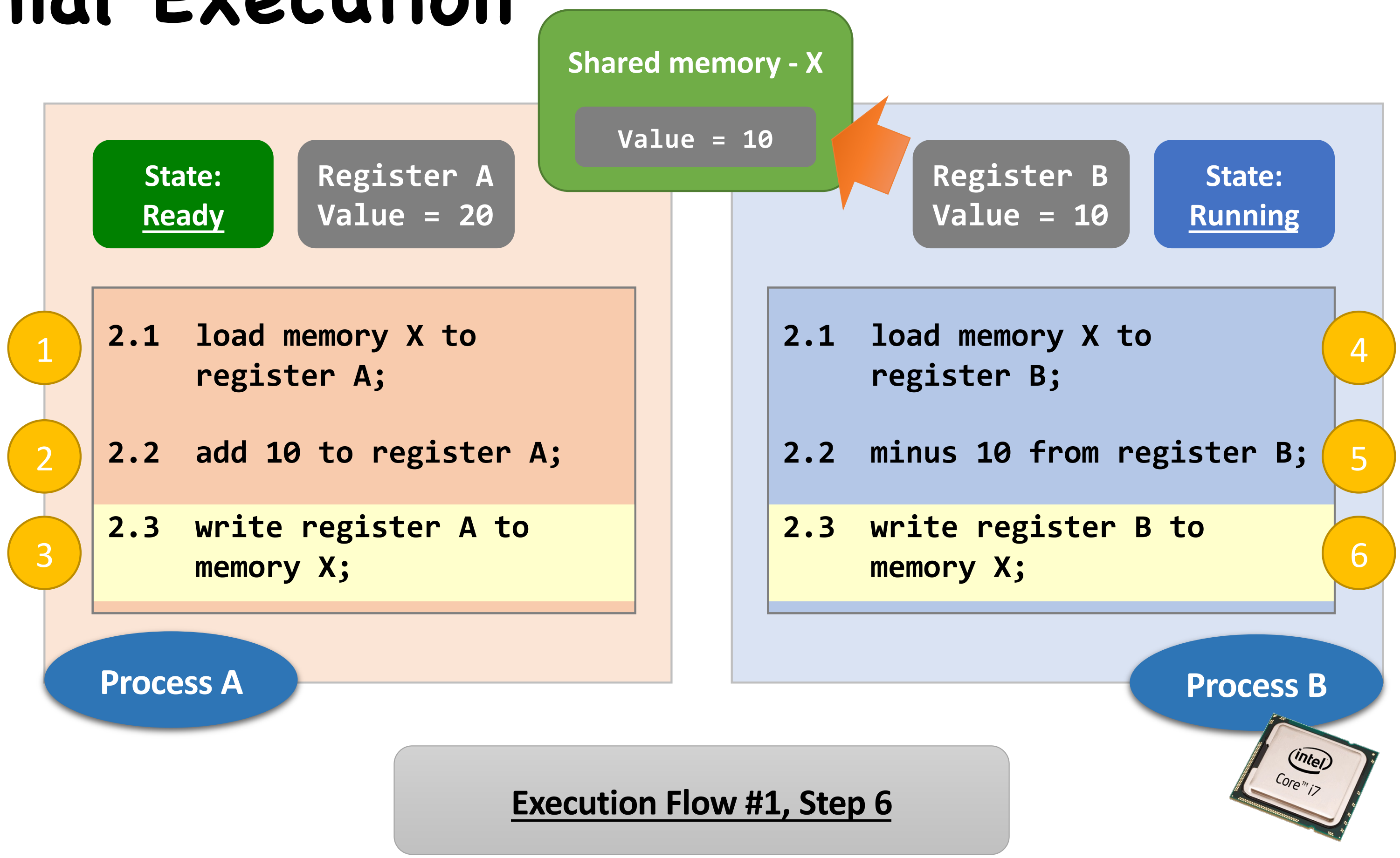
State: Running

4. 2.1 load memory X to register B;
5. 2.2 minus 10 from register B;

2.3 write register B to memory X;

Execution Flow #1, Step 5

Normal Execution



Don't print

Abnormal Execution

Shared memory - X

`Value = 10`

Process A

State: <u>Running</u>

```
Register A
Value = 10
```

1

```
2.1   load memory X to
      register A;

2.2   add 10 to register A;

2.3   write register A to
      memory X;
```

Process B

```
Register B
Value = 0
```

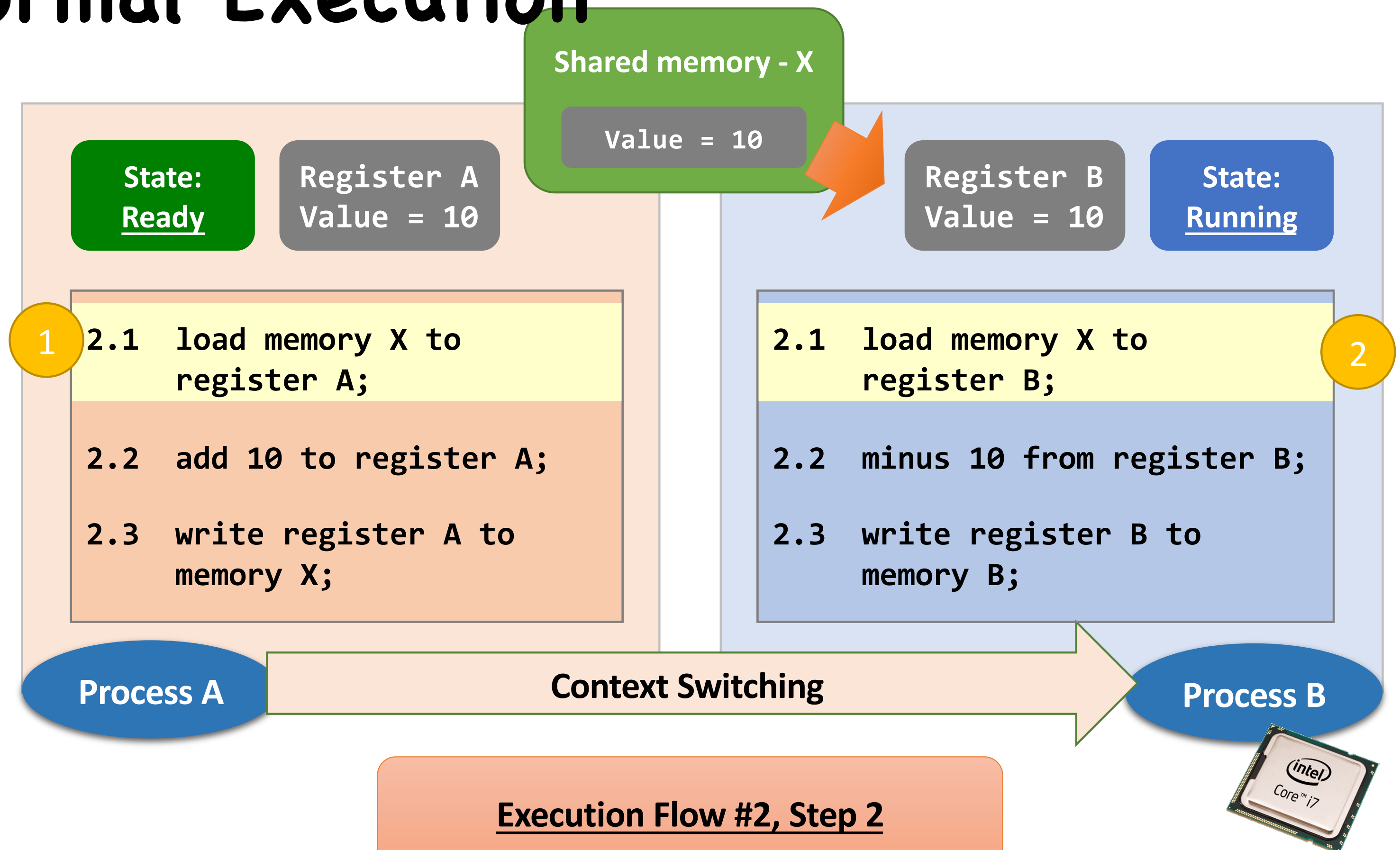
State: <u>Ready</u>

```
2.1   load memory X to
      register B;

2.2   minus 10 from register B;

2.3   write register B to
      memory X;
```

<u>**Execution Flow #2, Step 1**</u>

Don't print

Abnormal Execution



Execution Flow #2, Step 2

Abnormal Execution

Don't print

Shared memory - X

`Value = 10`

Process A

State: Ready

```
Register A
Value = 10
```

1

```
2.1   load memory X to
      register A;

2.2   add 10 to register A;

2.3   write register A to
      memory X;
```

Process B

```
Register B
Value = 0
```

State: Running

```
2.1   load memory X to
      register B;

2.2   minus 10 from register B;

2.3   write register B to
      memory B;
```

2

3

Execution Flow #2, Step 3

Don’t print

Abnormal Execution

Shared memory - X

Value = 0

State: Running

Register A Value = 10

Register B Value = 0

State: Ready

1 2.1 load memory X to register A;

2.2 add 10 to register A;

2.3 write register A to memory X;

2.1 load memory X to register B; 2

2.2 minus 10 from register B; 3

2.3 write register B to memory B; 4

Process A

Context Switching

Process B

Execution Flow #2, Step 4

Don't print

Abnormal Execution

Shared memory - X

Value = 0

Process A

State: Running

Register A
Value = 20

1 2.1 load memory X to register A;

5 2.2 add 10 to register A;

2.3 write register A to memory X;

Process B

Register B
Value = 0

State: Ready

2.1 load memory X to register B; 2

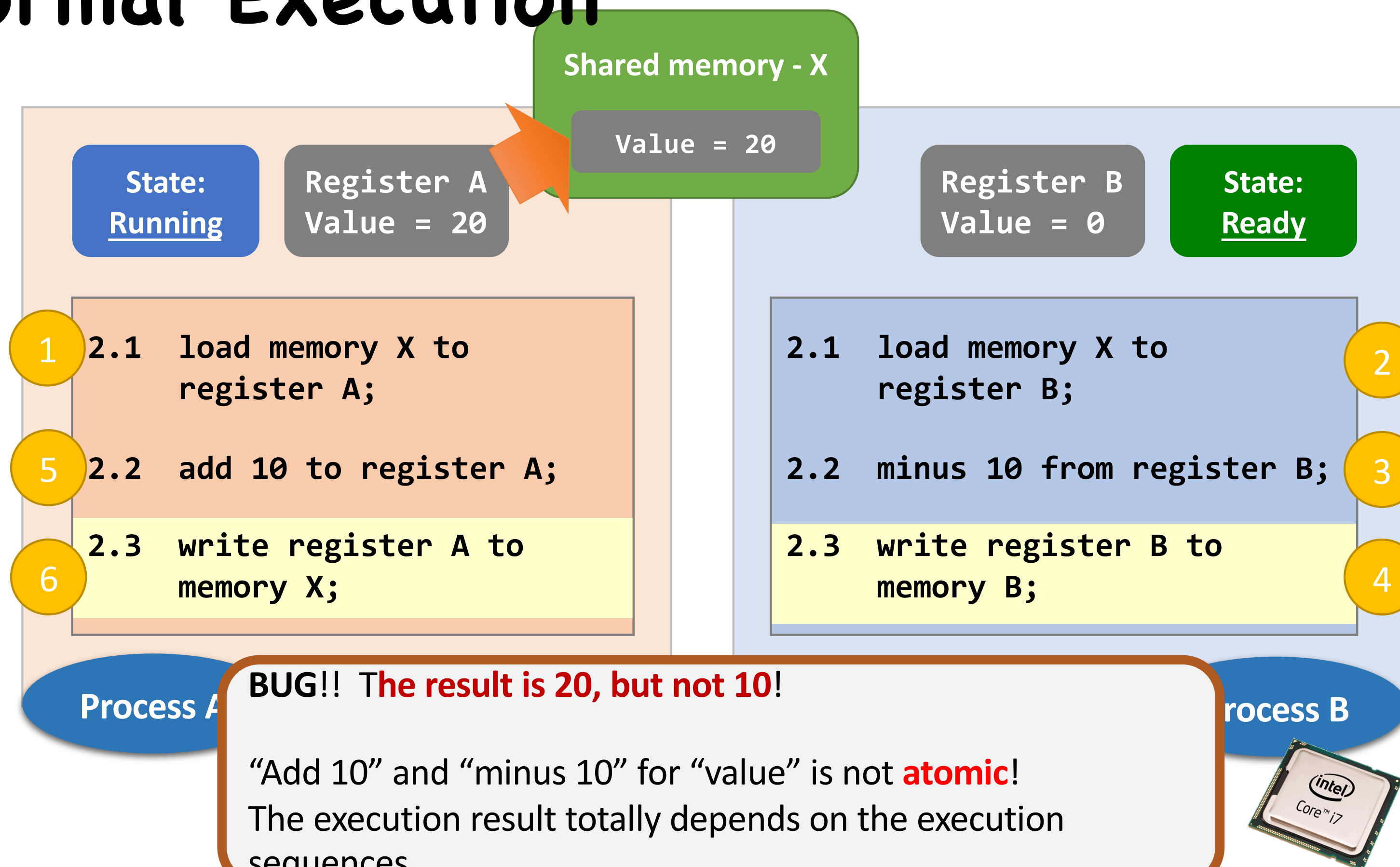
2.2 minus 10 from register B; 3

2.3 write register B to memory B; 4

Execution Flow #2, Step 5

Don’t print

Abnormal Execution



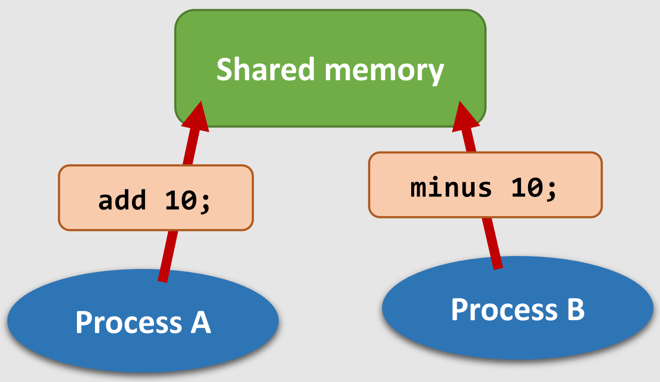
BUG!! **The result is 20, but not 10**!

“Add 10” and “minus 10” for “value” is not **atomic**!
The execution result totally depends on the execution sequences

Race Condition

- The above scenario is called the **race condition**.
  - May happen whenever "**shared object**" + "**multiple processes/threads**" + "**concurrently**"
- A **race condition** means
  - The outcome of an execution depends on a particular order in which the shared resource is accessed.
- Remember: race condition is always a bad thing and debugging race condition is a **nightmare**!
  - It may end up …
    - 99% of the executions are fine.
    - 1% of the executions are problematic.

Solution: Mutual Exclusion

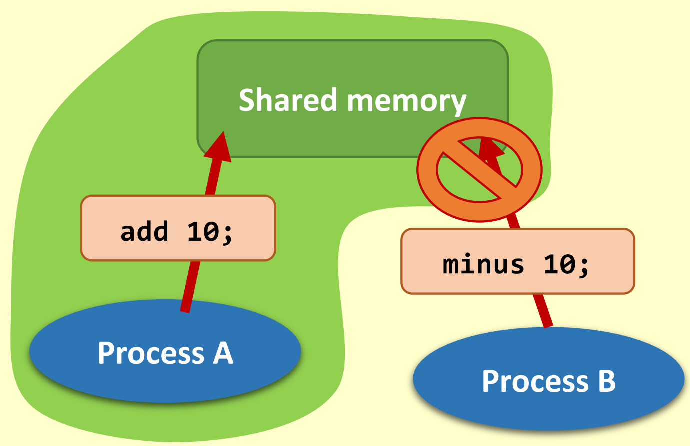


How to resolve race condition?

Solution: **mutual exclusion**

When one process access the shared memory, no one else could access it.

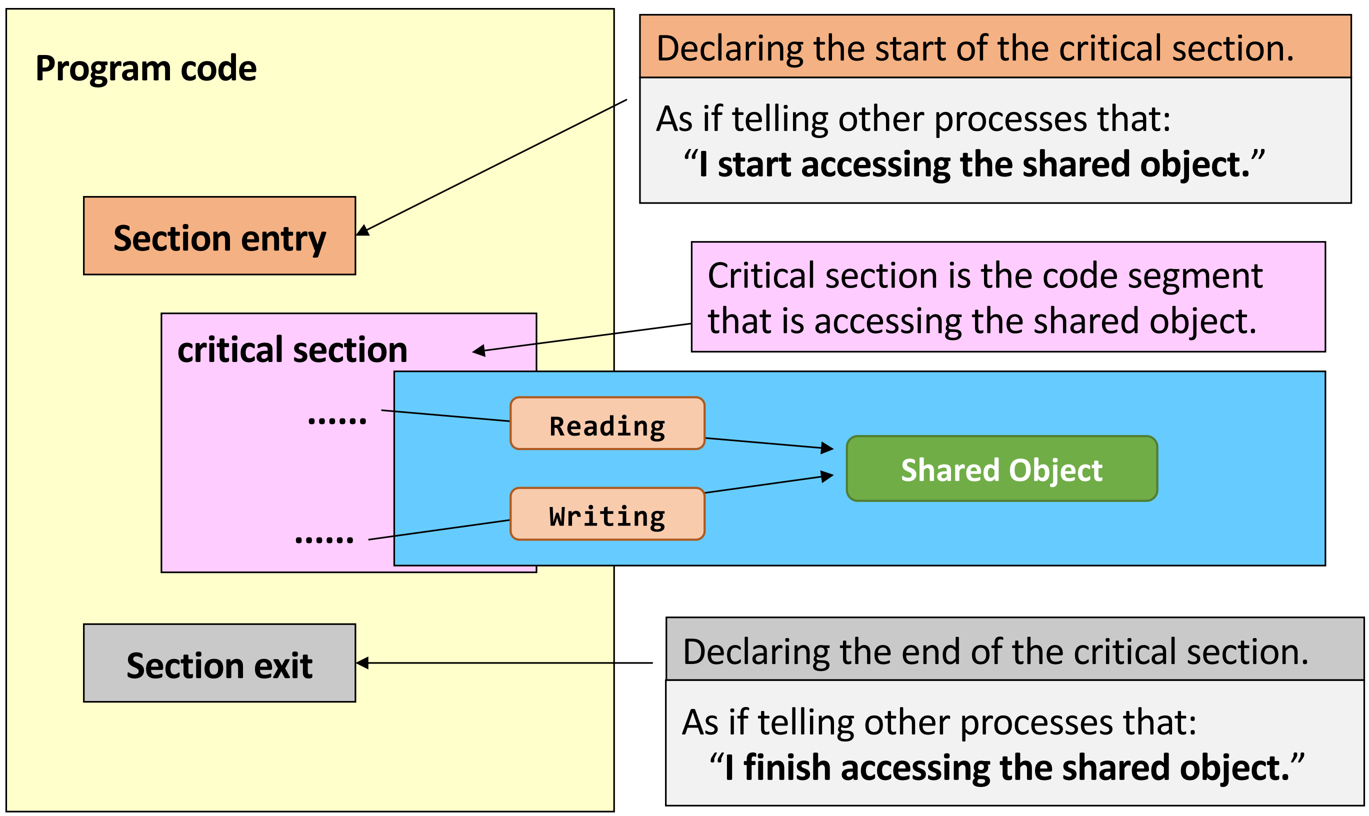
A set of processes would not have the problem of race condition if **mutual exclusion is guaranteed**.



Solution: Mutual Exclusion

- Shared object is still sharable, but
- Do not access the "shared object" **at the same time**
- Access the "shared object" one by one

Critical Section: Realizing Mutual Exclusion



Critical Section: Realizing Mutual Exclusion

Need a section entry here

```
2.1  load memory X to
     register A;

2.2  add 10 to register A;

2.3  write register A to
     memory X;
```

Need a section exit here

Process A

Need a section entry here

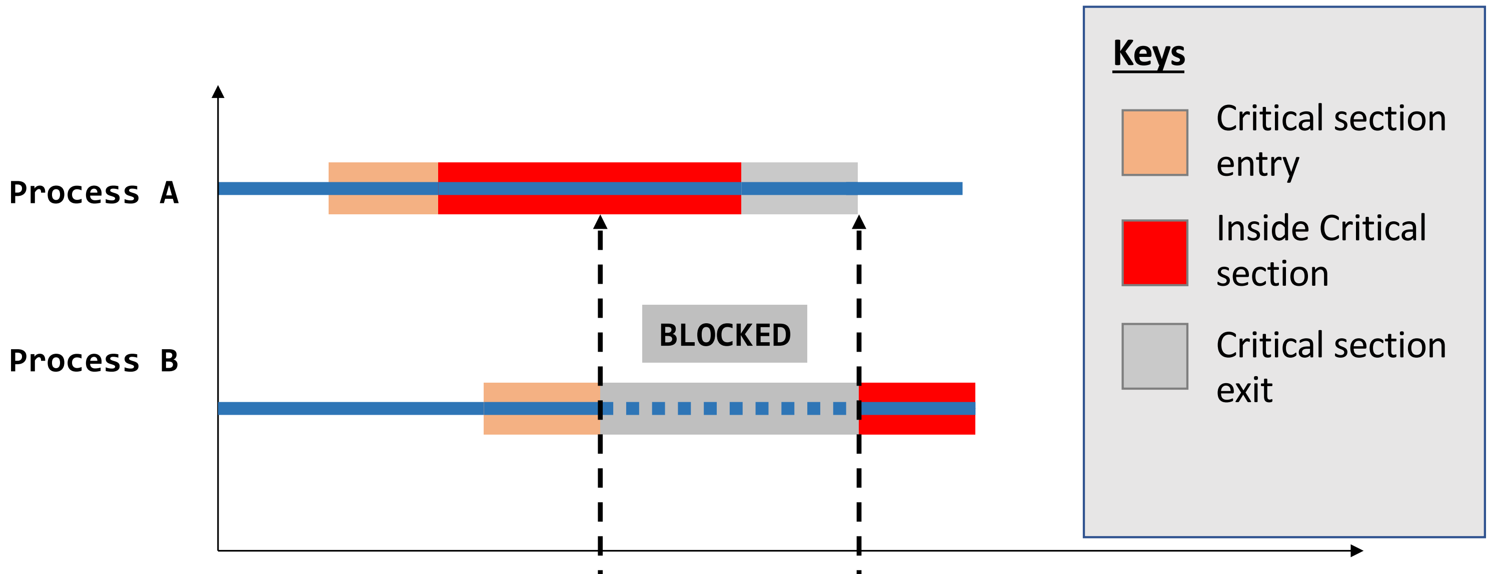
```
2.1  load memory X to
     register B;

2.2  minus 10 from register B;

2.3  write register B to
     memory X;
```

Need a section exit here

Process B

When process A is entering **its critical section**, process B cannot enter **its critical section.**

A Typical Mutual Exclusion Scenario



B tries to enter its critical section but A is in its critical section.

A leaves its critical section and B resumes execution accordingly.

Summary

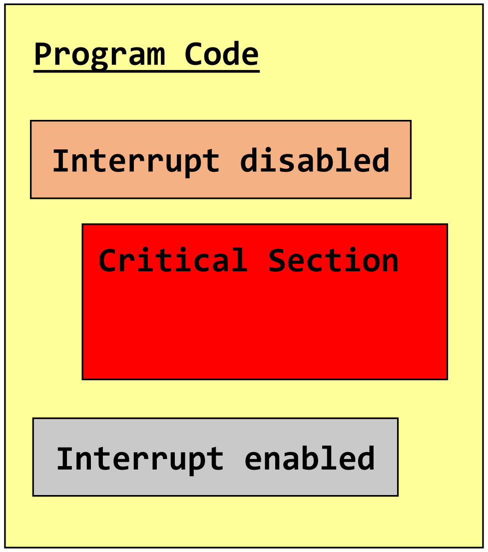
- Race condition
  - happens when programs accessing a shared object
  - The outcome of the computation totally depends on the execution sequences of the processes involved.
- Mutual exclusion is a requirement
  - If it could be achieved, then the problem of the race condition would be gone.
- A critical section is the code segment that access shared objects.
  - Critical section should be as tight as possible.
    - Well, you can set the entire code of a program to be a big critical section.
    - But, the program will have a very high chance to block other processes or to be blocked by other processes.
  - Note that one critical section can be designed for accessing more than one shared objects.

Critical Section Implementation

- Requirement #1. Mutual Exclusion
  - No two processes could be simultaneously go inside their own critical sections.
- Requirement #2. Bounded Waiting
  - Once a process starts trying to enter its critical section, there is a bound on the number of times other processes can enter theirs.
- Requirement #3. Progress
  - Say no process currently in critical section.
  - One of the processes trying to enter will eventually get in

Solution: Disabling Interrupts

- Disabling interrupts when the process is inside the critical section.
- When a process is in its critical section, no other processes could be able to run.
- Uni-core: Correct but not permissible
  - User level: what if one enters a critical section and loops infinitely?
    - OS cannot regain control if interrupt is disabled
  - Kernel level: yes, correct and permissible
- Multi-core: Incorrect
  - if there is another core modifying the shared object in the memory (unless you disable interrupts on all cores!!!!)



Solution: Locks

- Use yet another shared objects: **locks**
  - What about race condition on lock?
  - **Atomic instructions**: instructions that cannot be "interrupted", not even by instructions running on another core
- Spin-based locks
  - Process synchronization
    - Basic spinning using 1 shared variable
    - Peterson's solution: Spin using 2 shared variables
  - Thread synchronization: pthread_spin_lock
- Sleep-based locks
  - Process synchronization: POSIX semaphore
  - Thread synchronization: pthread_mutex_lock

Spin-based Locks

- Loop on a shared object, **turn**, to detect the status of other processes

Shared object “turn”   initial value = 0

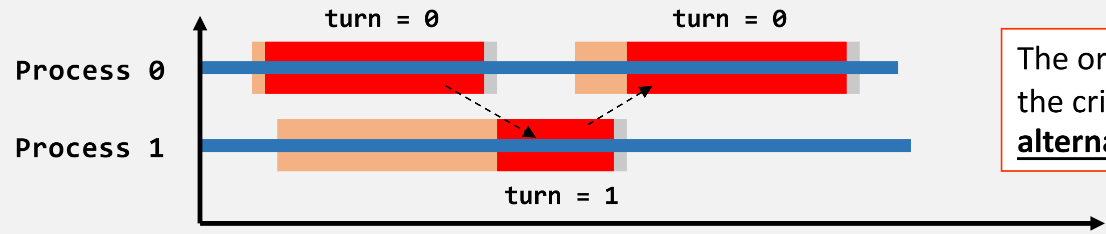
```
while (TRUE) {
  while( turn != 0 )
    ; /* busy waiting */

  critical_section();

  turn = 1;

  remainder_section();
}
```

Process 0

```
while (TRUE) {
  while( turn != 1 )
    ; /* busy waiting */

  critical_section();

  turn = 0;

  remainder_section();
}
```

Process1

Spin-based Locks (Cont'd)



The order of executing the critical section is **alternating**.

Shared object "turn"

initial Value = 0

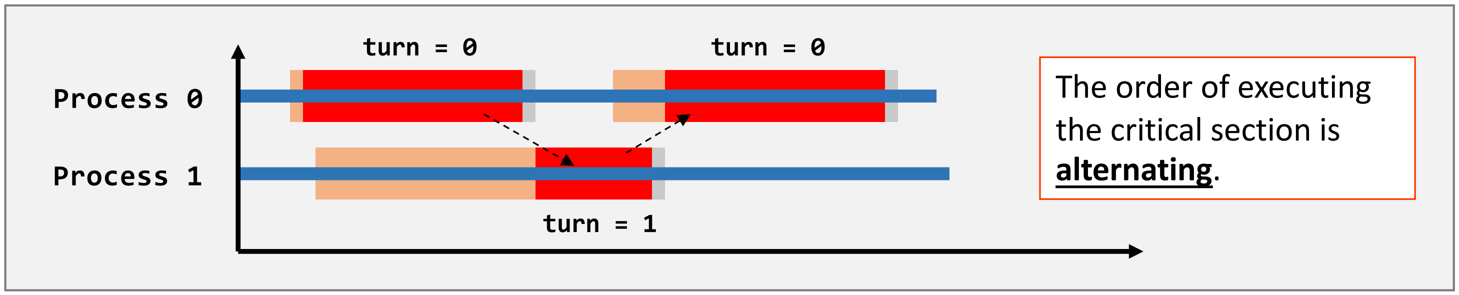
```
while (TRUE) {
  while( turn != 0 )
    ; /* busy waiting */
  critical_section();
  turn = 1;
  remainder_section();
}
```

Process 0

```
while (TRUE) {
  while( turn != 1 )
    ; /* busy waiting */
  critical_section();
  turn = 0;
  remainder_section();
}
```

Process1

Spin-based Locks (Cont'd)



- Correct but waste CPU resources
  - OK for short waiting (spin-time < context-switch-overhead)
    - Especially these days we have multi-core
      - Will not block other irrelevant processes a lot
- Impose a "strict alternating" order
  - Sometimes you give me my turn but I'm not ready to enter critical section yet

Don’t print

Spin-based Locks: Progress Violation

- Consider the following sequence:
  - Process0 leaves cs(), set turn=1
  - Process1 enters cs(), leaves cs(), set turn=0, work on **remainder_section_slow()**
  - Process0 loops back and enters cs() again, leaves cs(), set turn=1
  - Process0 finishes its **remainder_section()**, go back to top of the loop
    - It can’t enter its cs() (as turn=1)
    - That is, process0 gets blocked, but Process1 is outside its cs(), it is at its **remainder_section_slow()**

```
while (TRUE) {
  while( turn != 0 )
    ; /* busy waiting */
  cs();
  turn = 1;
  remainder_section();
}
```

Process 0

```
while (TRUE) {
  while( turn != 1 )
    ; /* busy waiting */
  cs();
  turn = 0;
  remainder_section_slow ();
}
```

Process 1

Turn = 1

Don't print

Spin-based Locks: Progress Violation

- Consider the following sequence:
  - Process0 leaves cs(), set turn=1
  - Process1 enters cs(), leaves cs(), set turn=0, work on **remainder_section-slow()**
  - Process0 loops back and enters cs() again, leaves cs(), set turn=1
  - Process0 finishes its **remainder_section()**, go back to top of the loop
    - It can't enter its cs() (as turn=1)
    - That is, process0 gets blocked, but Process1 is outside its cs(), it is at its **remainder_section-slow()**

Has to wait...

```
while (TRUE) {
  while( turn != 0 )
    ; /* busy waiting */

  cs();

  turn = 1;

  remainder_section();
}
```

Process 0

```
while (TRUE) {
  while( turn != 1 )
    ; /* busy waiting */

  cs();

  turn = 0;

  remainder_section_slow ();
}
```

Process 1

Turn = 1

Peterson's Solution: Improved Spin-based Locks

```
int turn;                             /* whose turn is it next */
int interested[2] = {FALSE,FALSE};  /* express interest to enter cs*/

void lock( int process ) {  /* process is 0 or 1 */
  int other;                          /* number of the other process */
  other = 1-process;                  /* other is 1 or 0 */
  interested[process] = TRUE;         /* express interest */
  turn = other;
  while ( turn == other &&
          interested[other] == TRUE )
    ;    /* busy waiting */
}

void unlock( int process ) {    /* process: who is leaving */
  interested[process] = FALSE;    /* I just left critical region */
}
```

Peterson's Solution: Improved Spin-based Locks

```
int turn;
int interested[2] = {FALSE,FALSE};

void lock( int process ) {
  int other;
  other = 1-process;
  interested[process] = TRUE;
  turn = other;
  while ( turn == other &&
          interested[other] == TRUE )
     ;    /* busy waiting */
}

void unlock( int process ) {
  interested[process] = FALSE;
}
```

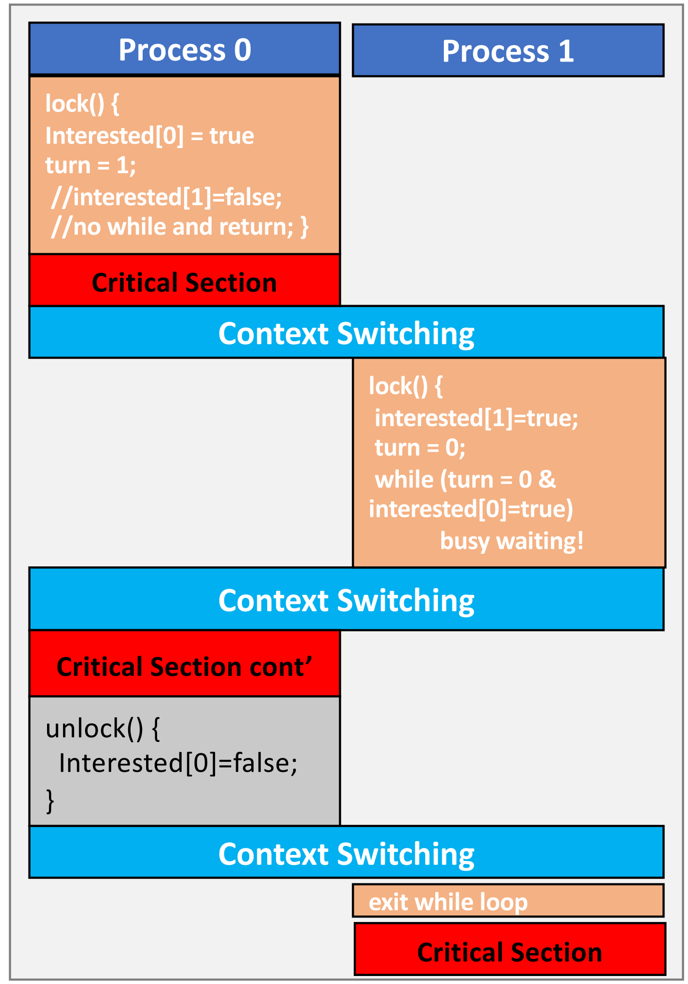
Express interest to enter CS

Being polite and let other go first

If other is not interested, I can always go ahead

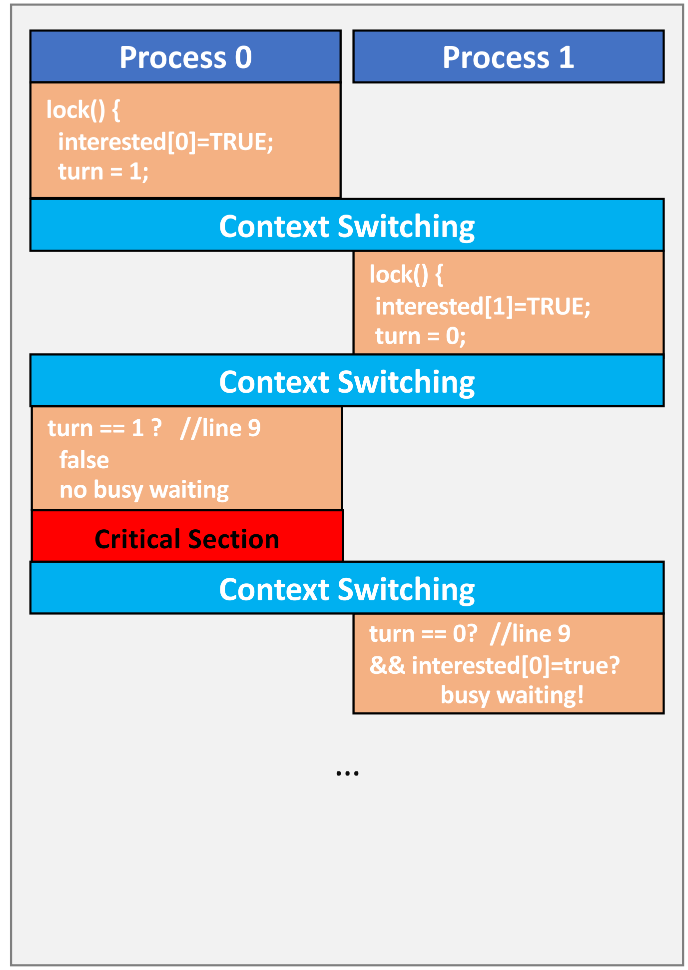
Peterson's Solution

```
int turn;
int interested[2] = {FALSE,FALSE};

void lock( int process ) {
  int other;
  other = 1-process;
  interested[process] = TRUE;
  turn = other;
  while ( turn == other &&
          interested[other] == TRUE )
     ;   /* busy waiting */
}

void unlock( int process ) {
  interested[process] = FALSE;
}
```



Peterson's Solution

```
1   int turn;
2   int interested[2] = {FALSE,FALSE};
3
4   void lock( int process ) {
5     int other;
6     other = 1-process;
7     interested[process] = TRUE;
8     turn = other;
9     while ( turn == other &&
              interested[other] == TRUE )
10      ;    /* busy waiting */
11  }
12
13  void unlock( int process ) {
14    interested[process] = FALSE;
15  }
```



Peterson's Solution Summary

- Mutual exclusion
  - interested[0] == interested[1] == true
  - turn == 0 or turn == 1, not both
- Progress
  - If only P_0 to enter critical section
    - interested[1] == false, thus P_0 enters critical section
  - If both P_0 and P_1 to enter critical section
    - interested[0] == interested[1] == true and (turn == 0 or turn == 1)
    - One of P_0 and P_1 will be selected
- Bounded-waiting
  - If both P_0 and P_1 to enter critical section, and P_0 selected first
  - When P_0 exit, interested[0] = false
    - If P_1 runs fast: interested[0] == false, P_1 enters critical section
    - If P_0 runs fast: interested[0] = true, but turn = 0, P_1 enters critical section

Multi-Process Mutual Exclusion

```
do {
        waiting[i] = TRUE;
        key = TRUE;
        while (waiting[i] && key)
                key = test_and_set(&lock);
        waiting[i] = FALSE;
                // critical section
        j = (i + 1) % n;
        while ((j != i) && !waiting[j])
                j = (j + 1) % n;
        if (j == i)
                lock = FALSE;
        else
                waiting[j] = FALSE;
                // remainder section
} while (TRUE);
```

Multi-Process Mutual Exclusion (Cont'd)

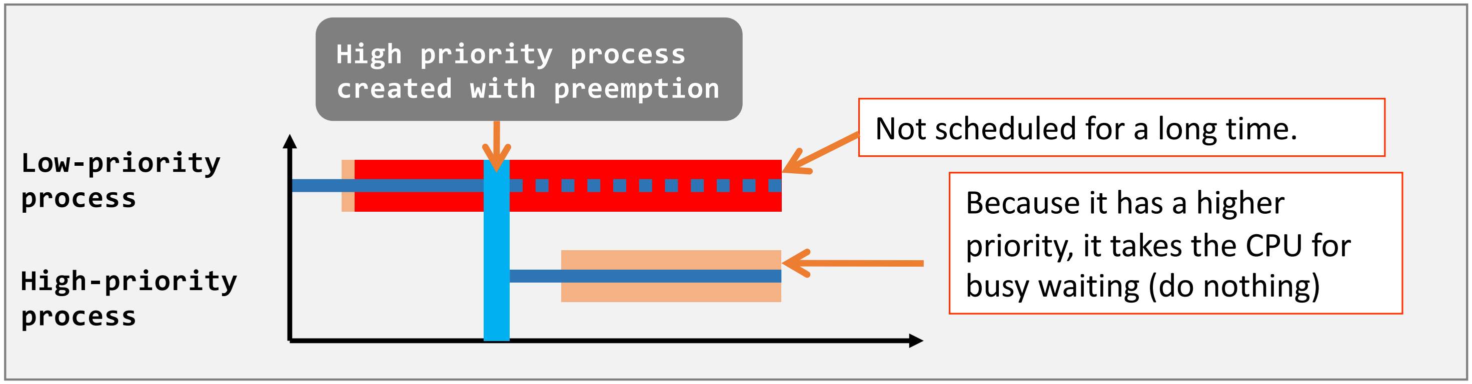
- Support n processes
  - boolean waiting[n]
  - boolean lock
    - initially FALSE
- A process can enter the critical section if either **waiting[i] == FALSE** or **key == FALSE**
  - key is local variable
  - All process must execute test_and_set() at least once
  - The first one call test_and_set() with lock==FALSE wins
    - key = FALSE
  - lock == TRUE after the first process executes test_and_set()
    - key = TRUE
- *Mutual exclusion* and *progress* are satisfied

Multi-Process Mutual Exclusion (Cont'd)

- When a process leaves the critical section
- It scans the array waiting[n] in a cyclic order (i+1, i+2, … n-1, 0, 1, …, i-1)
- The first process with waiting[j] == TRUE enters the critical section next
- Bounded-waiting: Any process waiting to enter its critical section will do so within n-1 turns.
- If no other process to enter critical section: i==j
  - lock = FALSE

Priority Inversion

- Priority/Preemptive Scheduling (Linux, Windows… all OS…)
  - A low priority process **L** is inside the critical region, but …
  - A high priority process **H** gets the CPU and wants to enter the critical region.
    - But **H** cannot **lock** (because **L** has not **unlock**)
    - So, **H** gets the CPU to do nothing but spinning



Sleep-based Lock: Semaphore

- Semaphore is just a struct, which includes
  - an integer that counts the # of resources available
    - Can do more than solving mutual exclusion
  - a wait-list
- The trick is still the section entry/exit function implementation
  - Must involve kernel (for sleep)
  - Implement uninterruptable **section entry/exit**
    - Disable interrupts (on single core)
    - Atomic instructions (on multiple cores)

Semaphore

```
typedef struct {
   int value;
   list process_id;
} semaphore;
```

Section Entry: sem_wait()

```
void sem_wait(semaphore *s) {

   s->value = s->value – 1;
   if ( s->value < 0 ) {

      sleep();

   }

}
```

Initialize **s->value** = 1

"**sem_wait(s)**"

- I wait until s->value>=0 (i.e., **sem_wait(s)** only returns when s->value>=0)

<u>Important</u>

`This wait is different from parent's folk wait(child). When programming, it is sem_wait()`

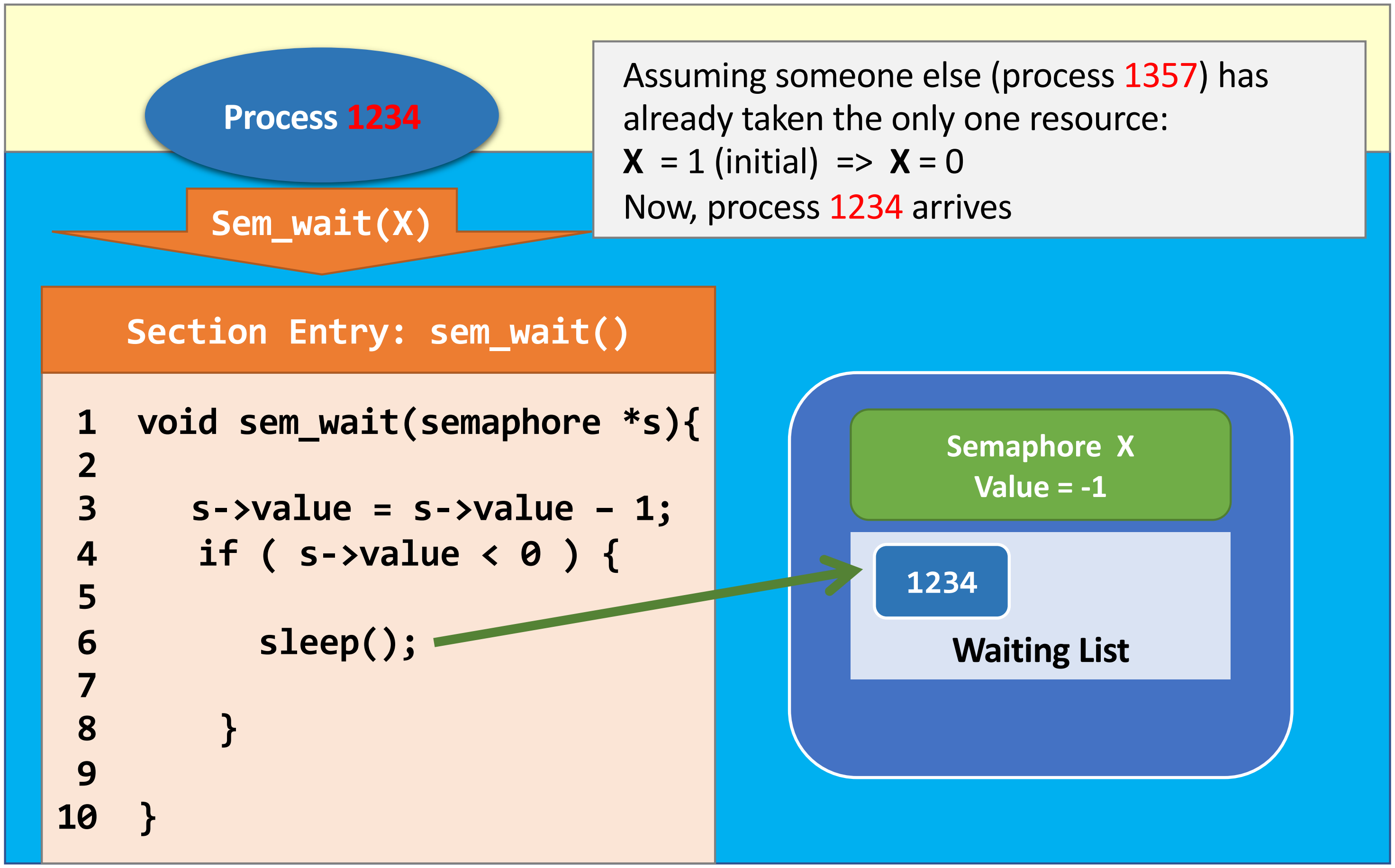
"**sem_post(s)**"

- I notify the others (if anyone waiting) that **s->value <= 0**

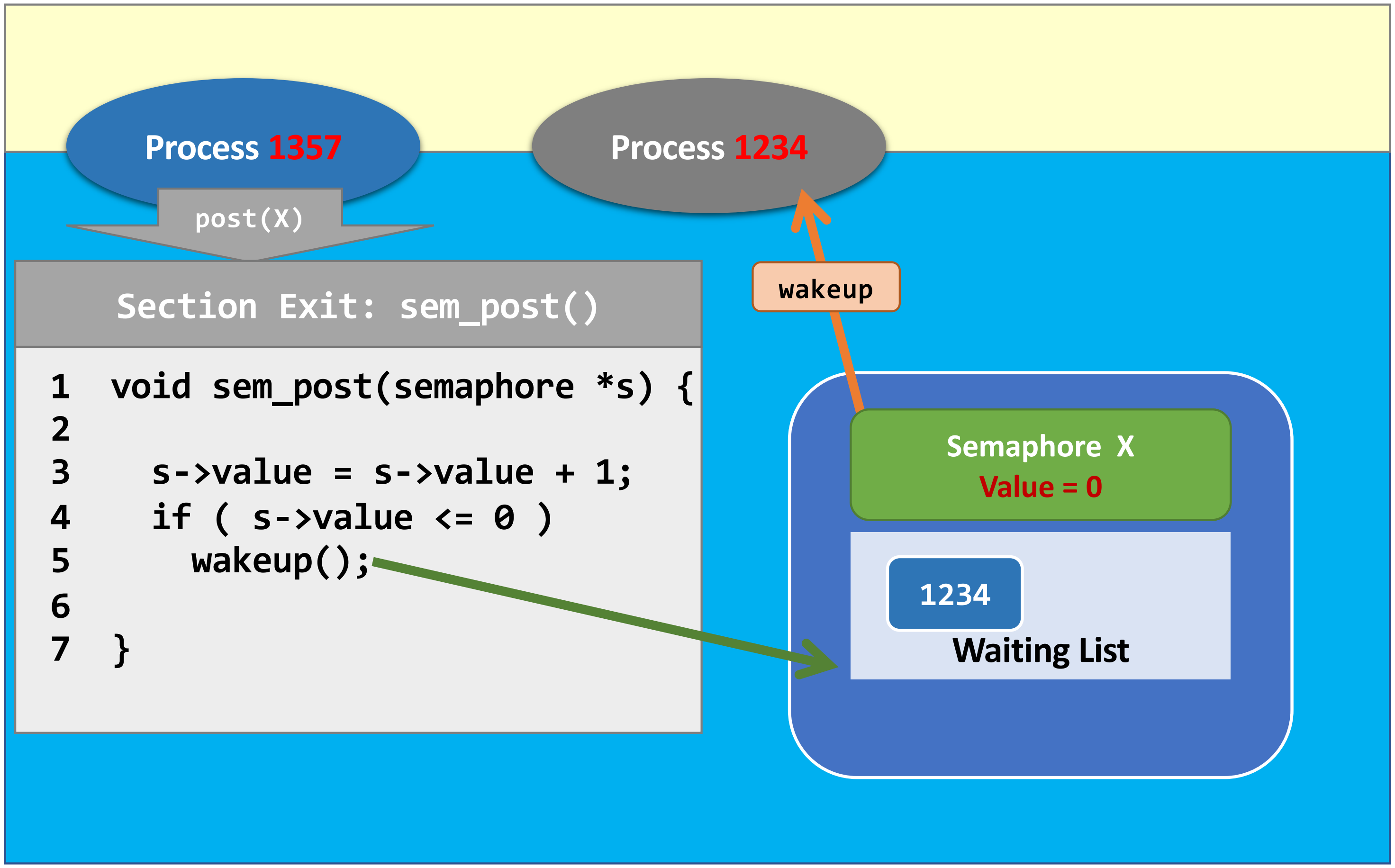
Section Exit: sem_post()

```
void sem_post(semaphore *s) {

   s->value = s->value + 1;
   if ( s->value <= 0 )
      wakeup();

}
```

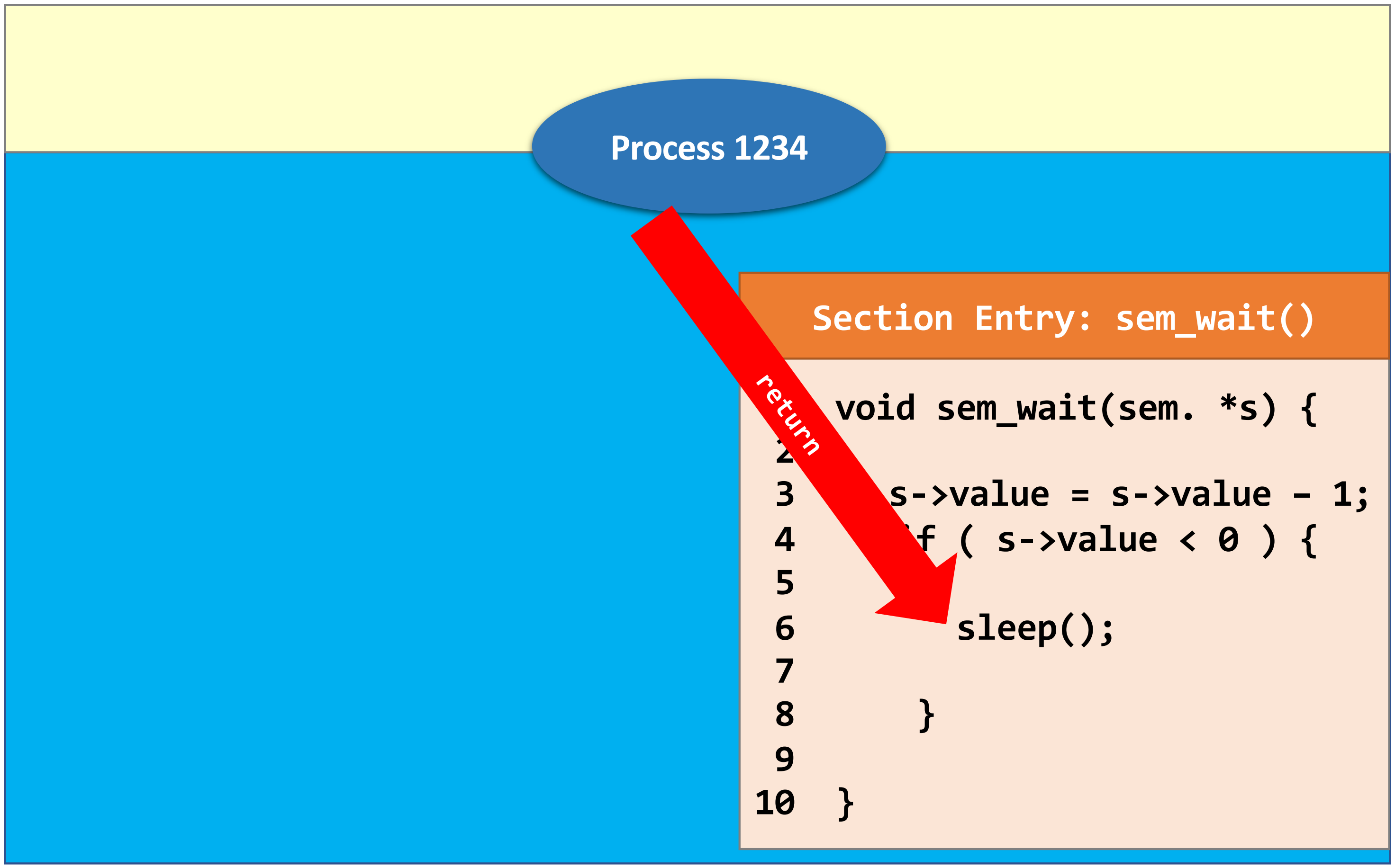
Semaphore Example



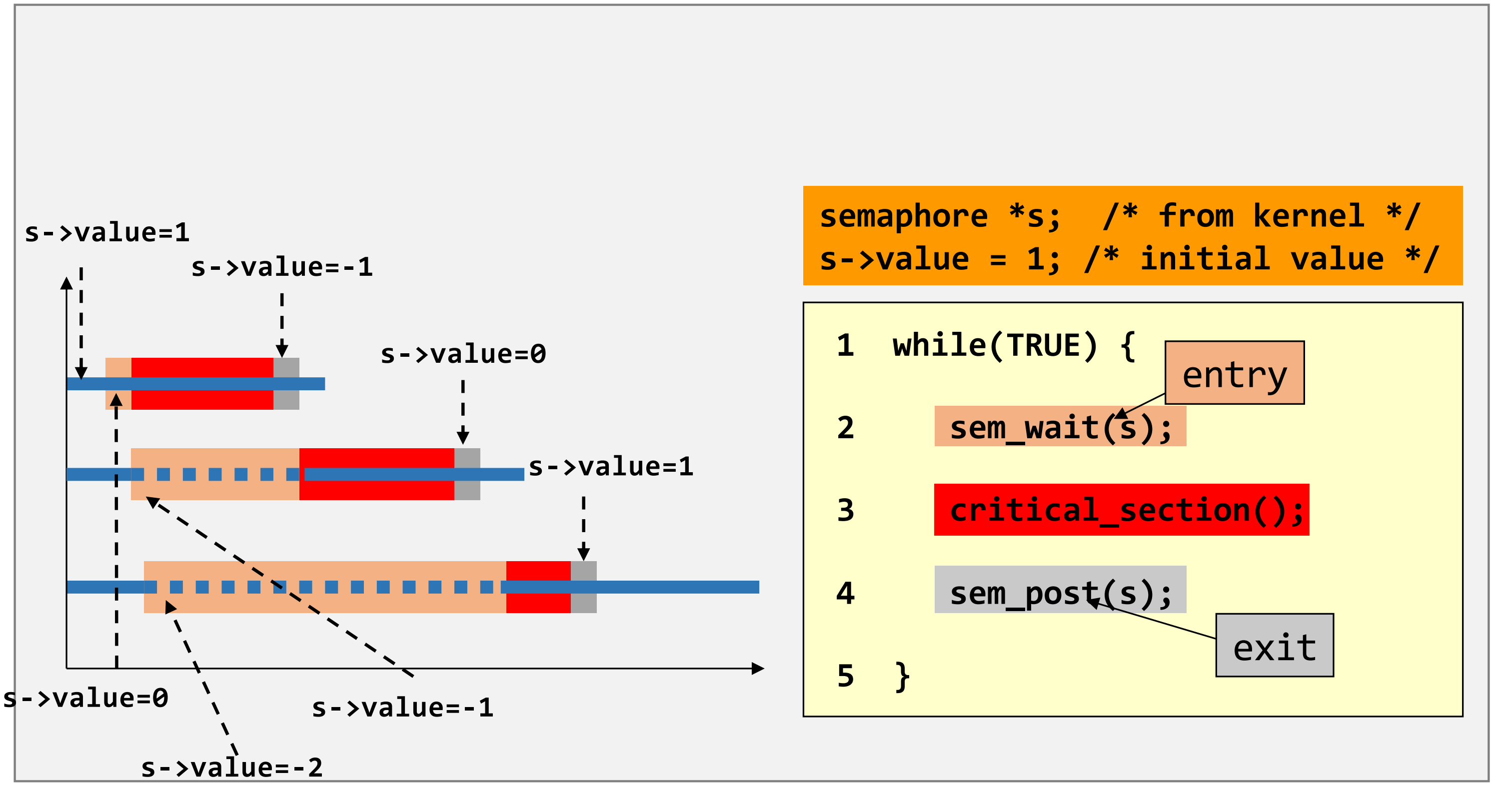
Semaphore Example



Semaphore Example



Using Semaphore in User Process



Semaphore Implementation

- Must guarantee that no two processes can execute sem_wait() and sem_post() on the same semaphore at the same time
- Thus, implementation becomes the critical section problem where the wait and signal code are placed in the critical section.
  - Need to disable interrupt on single-processor machine
  - use atomic instruction cmp_xchg() on multi-core architecture

```
Example: Atomic increment: atomic_inc(addr)
/////////////// implemented as /////////
do {
  int old = *addr;
  int new = old + 1;
} while (cmp_xchg(addr, old, new) != old);
```

```
///one single instruction with the following
///semantics
void cmp_xchg(int *addr, int expected_value,
int new_value)
        {
              int temp = *addr;
              if(*addr == expected_value)
                          *addr = new_value;
              return temp;
        }
```

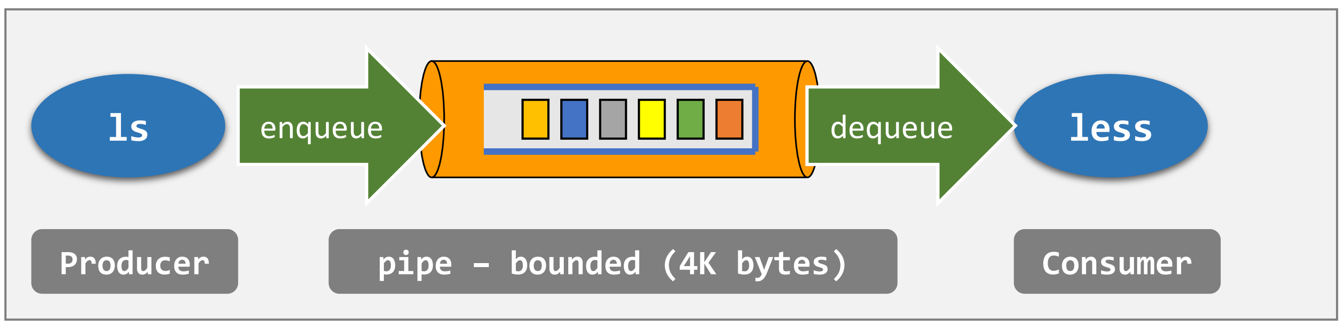
Using Semaphore beyond Mutual Exclusion

- Producer-Consumer Problem
  - Two types of processes: **producer** and **consumer**;
    - At least one producer and one consumer.
- Dining Philosopher Problem
  - Only one type of process
    - At least two processes.
- Reader Writer Problem
  - Multiple readers, one writer

Producer-consumer Problem

- Also known as the bounded-buffer problem.
- Single-object synchronization

A bounded buffer	-It is a shared object; -Its size is bounded, say N slots. -It is a queue (imagine that it is an array implementation of queue).
A producer process	-It produces a unit of data, and -writes a piece of data to the tail of the buffer at one time.
A consumer process	-It removes a unit of data from the head of the bounded buffer at one time.

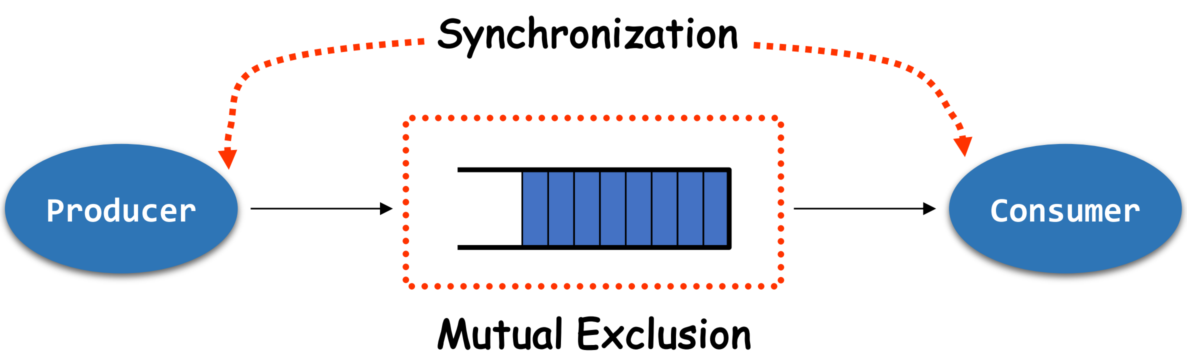


Producer-consumer Problem

Requirement #1	When the **producer** wants to (a) put a new item in the buffer, but **(b) the buffer is already full**… Then, **the producer should wait**. **The consumer should notify the producer** after she has dequeued an item.
Requirement #2	When the **consumer** wants to (a) consumes an item from the buffer, but (b) **the buffer is empty**… Then, **the consumer should wait**. **The producer should notify the consumer** after she has enqueued an item.

Solving Producer-consumer Problem with Semaphore

- The problem can be divided into two sub-problems.
    - **Mutual exclusion** with one **binary semaphore**
        - The buffer is a shared object.
    - **Synchronization** with two **counting semaphores**
        - **Notify** the producer to stop producing when the buffer is full
            - In other words, notify the producer to produce when the buffer is NOT full
        - **Notify** the consumer to stop eating when the buffer is empty
            - In other words, notify the consumer to consume when the buffer is NOT empty



Solving Producer-consumer Problem with Semaphore

Shared object

```
#define N 100
semaphore mutex = 1;
semaphore avail = N;
semaphore fill  = 0;
```

Abstraction of semaphore as integer!

Note

The size of the bounded buffer is "N".

fill : number of occupied slots in buffer
avail: number of empty slots in buffer

Producer function

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        wait(&avail);
        wait(&mutex);
        insert_item(item);
        post(&mutex);
        post(&fill);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        wait(&fill);
        wait(&mutex);
        item = remove_item();
        post(&mutex);
        post(&avail);
        //consume the item;
    }
}
```

Solving Producer-consumer Problem with Semaphore

Note

6: (Producer) I wait for an **avail**able slot and acquire it if I can

10: (Producer) I notify the others that I have **fill**ed the buffer

Producer function

```
1   void producer(void) {
2       int item;
3
4       while(TRUE) {
5           item = produce_item();
6           wait(&avail);
7           wait(&mutex);
8           insert_item(item);
9           post(&mutex);
10          post(&fill);
11      }
12  }
```

Solving Producer-consumer Problem with Semaphore

Note
6: (Producer) I wait for an **available** slot and acquire it if I can

10: (Producer) I notify the others that I have **fill**ed the buffer

Note
5: (Consumer) I wait for someone to **fill** up the buffer and proceed if I can

9: (Consumer) I notify the others that I have made the buffer with a new **avail**able slot

Producer function

```
1   void producer(void) {
2       int item;
3
4       while(TRUE) {
5           item = produce_item();
6           wait(&avail);
7           wait(&mutex);
8           insert_item(item);
9           post(&mutex);
10          post(&fill);
11      }
12  }
```

Consumer Function

```
1   void consumer(void) {
2       int item;
3
4       while(TRUE) {
5           wait(&fill);
6           wait(&mutex);
7           item = remove_item();
8           post(&mutex);
9           post(&avail);
10          //consume the item;
11      }
12  }
```

Question 1

Necessary to use both “avail” and “fill”?

Let us try to remove semaphore fill?

Shared object

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore avail = N;
semaphore fill  = 0;
```

Producer function

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        wait(&avail);
        wait(&mutex);
        insert_item(item);
        post(&mutex);
        post(&fill);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        wait(&fill);
        wait(&mutex);
        item = remove_item();
        post(&mutex);
        post(&avail);
    //consume the item;
    }
}
```

Question 1

Just view wait(avail) as -- resource?
Just view post(avail) as ++ resource?

so
- producer avail-- by wait
- consumer avail++ by post

Problem solved?

Producer function

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        wait(&avail);
        wait(&mutex);
        insert_item(item);
        post(&mutex);
        post(&fill);
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        wait(&fill);
        wait(&mutex);
        item = remove_item();
        post(&mutex);
        post(&avail);
    //consume the item;
    }
}
```

Question 1

```
Just view wait(avail) as -- resource?
Just view post(avail) as ++ resource?
```

```
so
• producer avail-- by wait
• consumer avail++ by post
```

If consumer gets CPU first, it removes item from NULL

ERROR

Producer function

```
void producer(void) {
    int item;

    while(TRUE) {
        item = produce_item();
        wait(&avail);
        wait(&mutex);
        insert_item(item);
        post(&mutex);
        ~~post(&fill);~~
    }
}
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        ~~wait(&fill);~~
        wait(&mutex);
        item = remove_item();
        post(&mutex);
        post(&avail);
    //consume the item;
    }
}
```

Question 2

Question #2.
Can we swap Lines 6 & 7 of the producer?

Let us simulate what will happen with the modified code!

Shared object

```
#define N 100
semaphore mutex = 1;
semaphore avail = N;
semaphore fill  = 0;
```

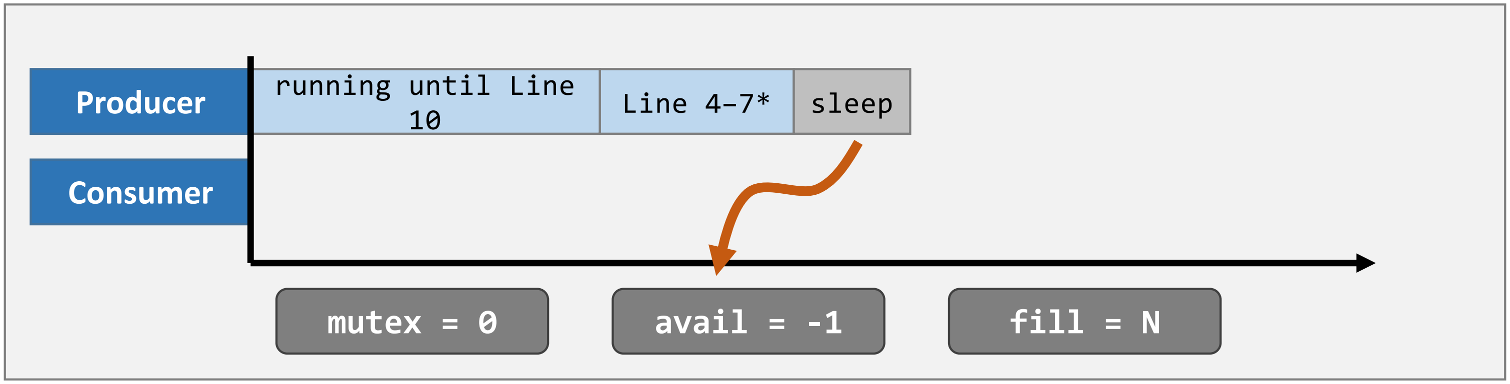
Producer function

```
1   void producer(void) {
2       int item;
3
4       while(TRUE) {
5           item = produce_item();
6*          wait(&mutex);
7*          wait(&avail);
8           insert_item(item);
9           post(&mutex);
10          post(&fill);
11      }
12  }
```

Consumer Function

```
void consumer(void) {
    int item;

    while(TRUE) {
        wait(&fill);
        wait(&mutex);
        item = remove_item();
        post(&mutex);
        post(&avail);
        //consume the item
    }
}
```

Question 2



Producer function

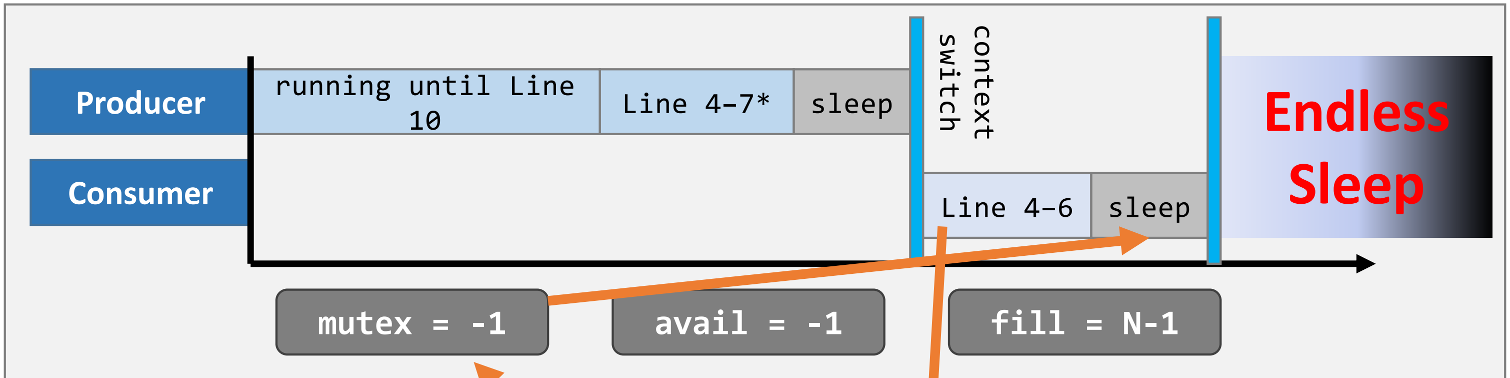
```
1   void producer(void) {
2       int item;
3
4       while(TRUE) {
5           item = produce_item();
6*          wait(&mutex);
7*          wait(&avail);
8           insert_item(item);
9           post(&mutex);
10          post(&fill);
11      }
12  }
```

Consumer Function

```
1   void consumer(void) {
2       int item;
3
4       while(TRUE) {
5           wait(&fill);
6           wait(&mutex);
7           item = remove_item();
8           post(&mutex);
9           post(&avail);
10          //consume the item
11      }
12  }
```

Consider: producer gets the CPU to keep producing until the buffer is full

Question 2



Producer function

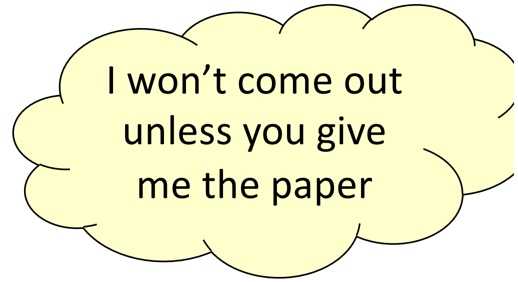
```
1   void producer(void) {
2       int item;
3
4       while(TRUE) {
5           item = produce_item();
6*          wait(&mutex);
7*          wait(&avail);  !sleep with lock!
8           insert_item(item);
9           post(&mutex);
10          post(&fill);
11      }
12  }
```

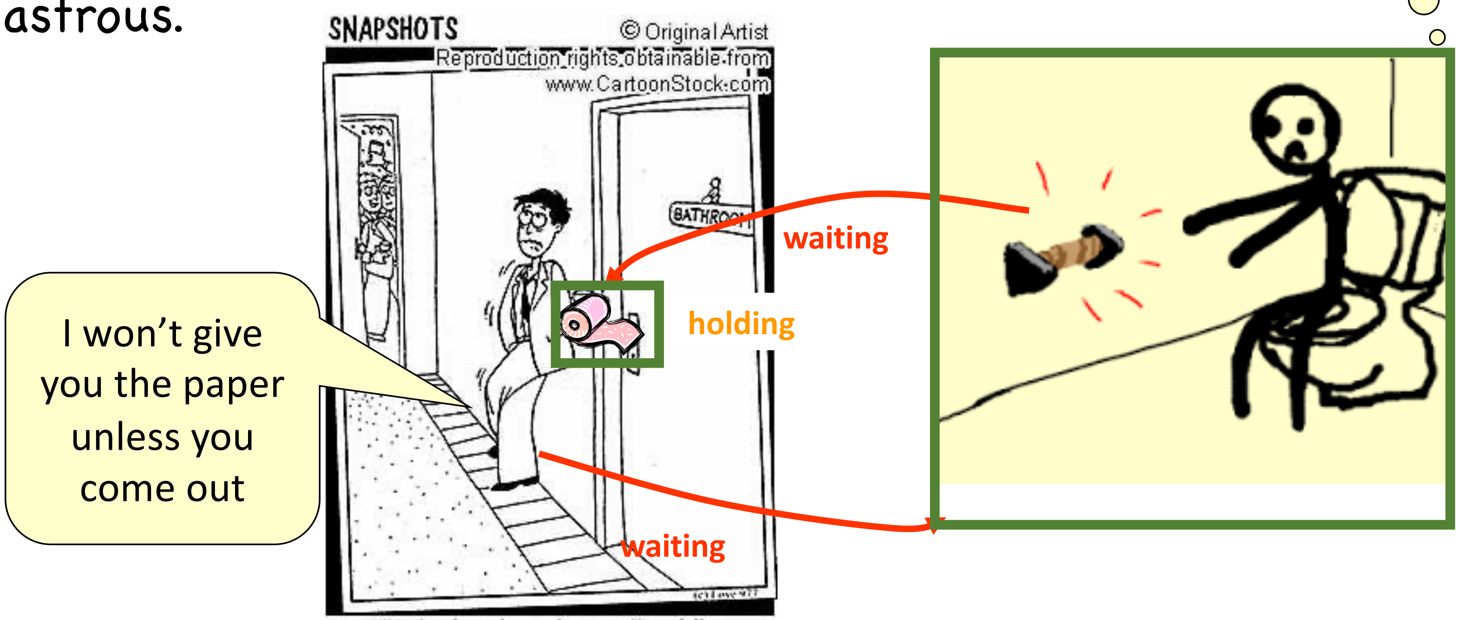
Consumer Function

```
1   void consumer(void) {
2       int item;
3
4       while(TRUE) {
5           wait(&fill);
6           wait(&mutex);
7           item = remove_item();
8           post(&mutex);
9           post(&avail);
10          //consume the item
11      }
12  }
```

Question 2

- This scenario is called a **deadlock**
  - Consumer waits for Producer's **mutex** at line 6
    - i.e., it waits for Producer (line 9) to unlock the **mutex**
  - Producer waits for Consumer's **avail** at line 7
    - i.e., it waits for Consumer (line 9) to release **avail**
  - **Implication**: careless implementation of the producer-consumer solution can be disastrous.





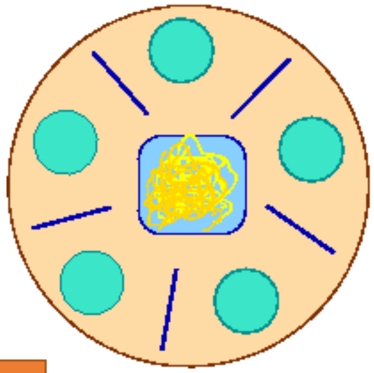
Summary on Producer-consumer Problem

- How to avoid race condition on the shared buffer?
  - E.g., Use a **binary semaphore**.

- How to achieve synchronization?
  - E.g., Use two **counting semaphores**: fill and avail

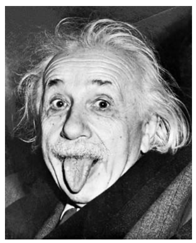
Dining Philosopher Problem

- 5 philosophers, 5 plates of spaghetti, and 5 chopsticks.
- The jobs of each philosopher are to think and to eat
- They **need exactly two chopsticks** in order to eat the spaghetti.
- Question: how to construct a synchronization protocol such that they
  - will not **starve to death**, and
  - will not result in any **deadlock scenarios**?
    - A waits for B's chopstick
    - B waits for C's chopstick
    - C waits for A's chopstick ….



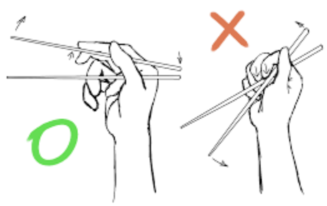
It's a multi-object synchronization problem

Dining Philosopher Problem



Philosophers

Process



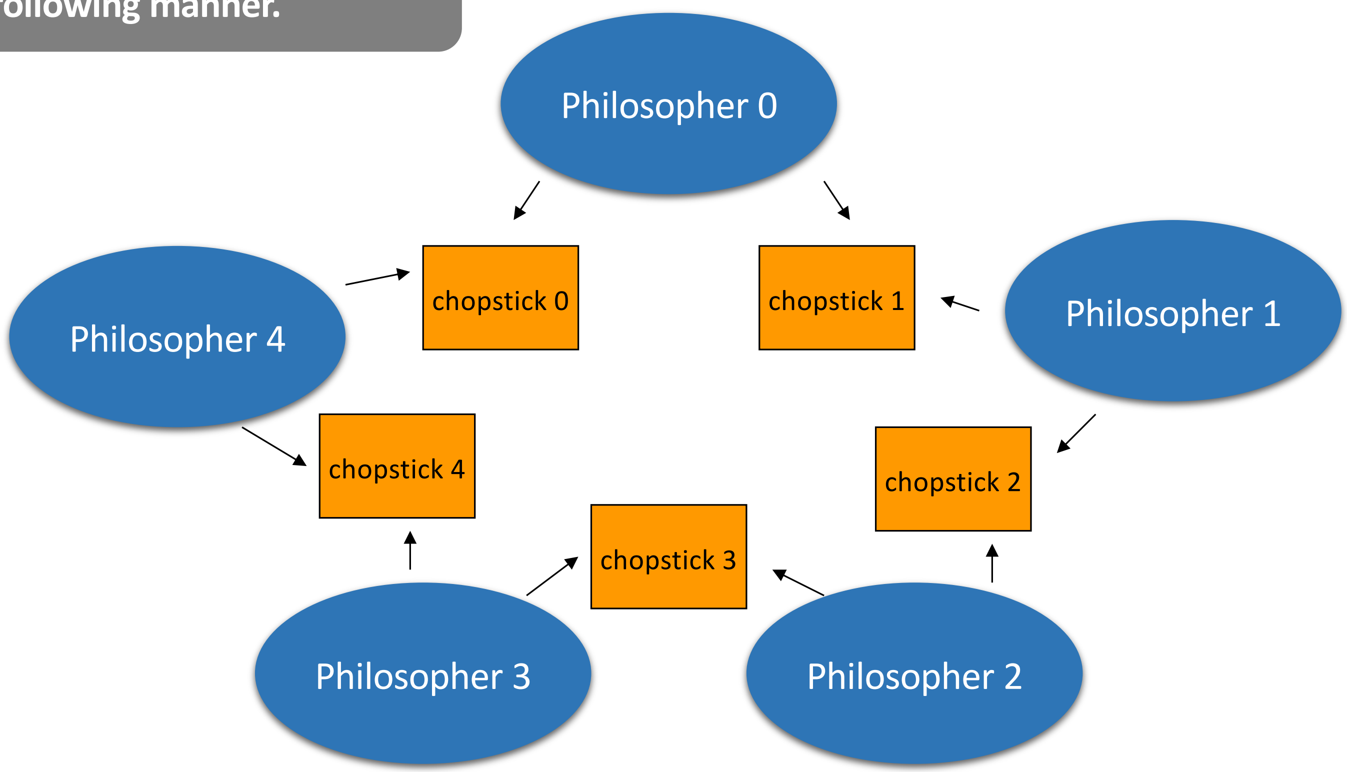
Chopsticks

Shared Object

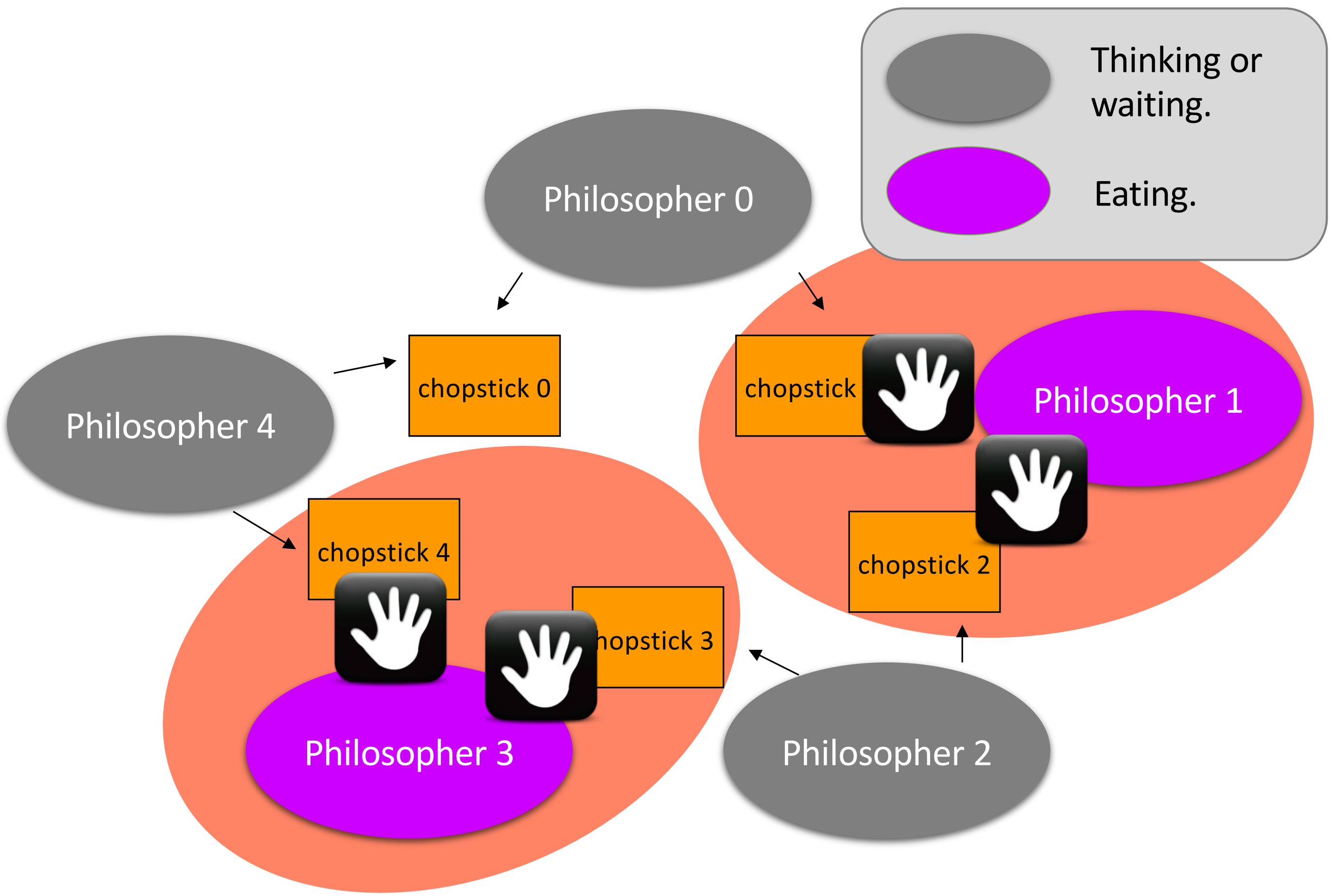
A process needs two shared resources in order to do some work

Dining Philosopher Problem

The chopsticks are arranged in the following manner.



Dining Philosopher Problem



Dining Philosopher – Requirement 1

- **<u>Mutual exclusion</u>**
  - While you are eating, people cannot steal your chopstick
  - Two persons cannot hold the same chopstick

- Let's propose the following solution:
  - When you are hungry, you have to check if anyone is using the chopsticks that you need.
  - If yes, you wait.
  - If no, **seize both chopsticks**.
  - After eating, put down both your chopsticks.

Dining Philosopher – Requirement 1

Shared object

```
#define  N  5
semaphore chopstick[N];
```

Five binary semaphores

Helper Functions

```
void take_chopstick(int i)
{
    wait(&chopstick[i]);
}
```

```
void put_chopstick(int i) {
    post(&chopstick[i]);
}
```

Section Entry

Critical Section

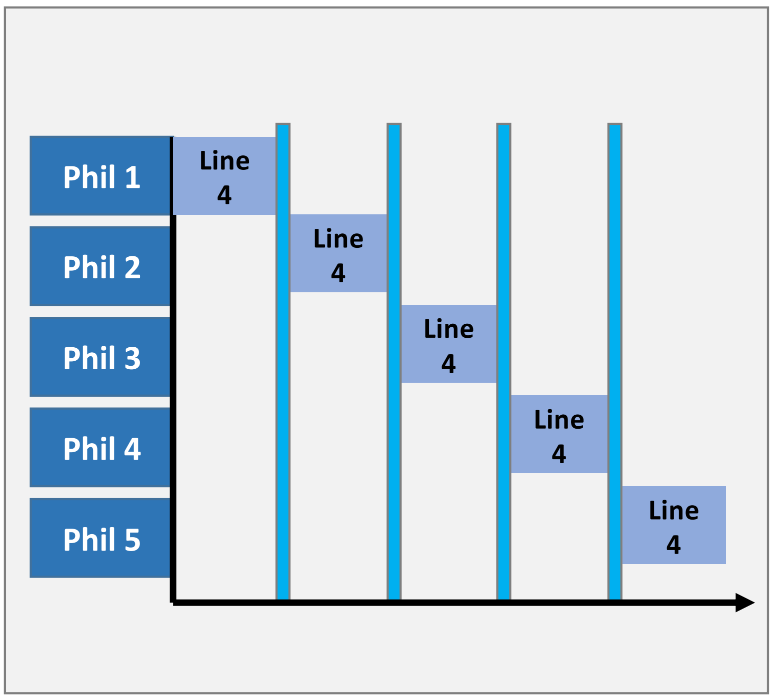
Section Exit

Main Function

```
void philosopher(int i) {
    while (TRUE) {
        think();

        take_chopstick(i);
        take_chopstick((i+1) % N);

        eat();

        put_chopstick(i);
        put_chopstick((i+1) % N);
    }
}
```

Dining Philosopher - Deadlock

- Each philosopher finishes thinking at the same time and each first grabs her left chopstick
- All chopsticks[i]=0
- When executing line 5, all are waiting



Main Function

```
1 void philosopher(int i) {
2     while (TRUE) {
3         think();

4         take_chopstick(i);
5         take_chopstick((i+1) % N);

6         eat();

7         put_chopstick(i);
8         put_chopstick((i+1) % N);
9     }
10 }
```

Dining Philosopher – Requirement 2

- **Synchronization**
  - Should avoid **deadlock.**

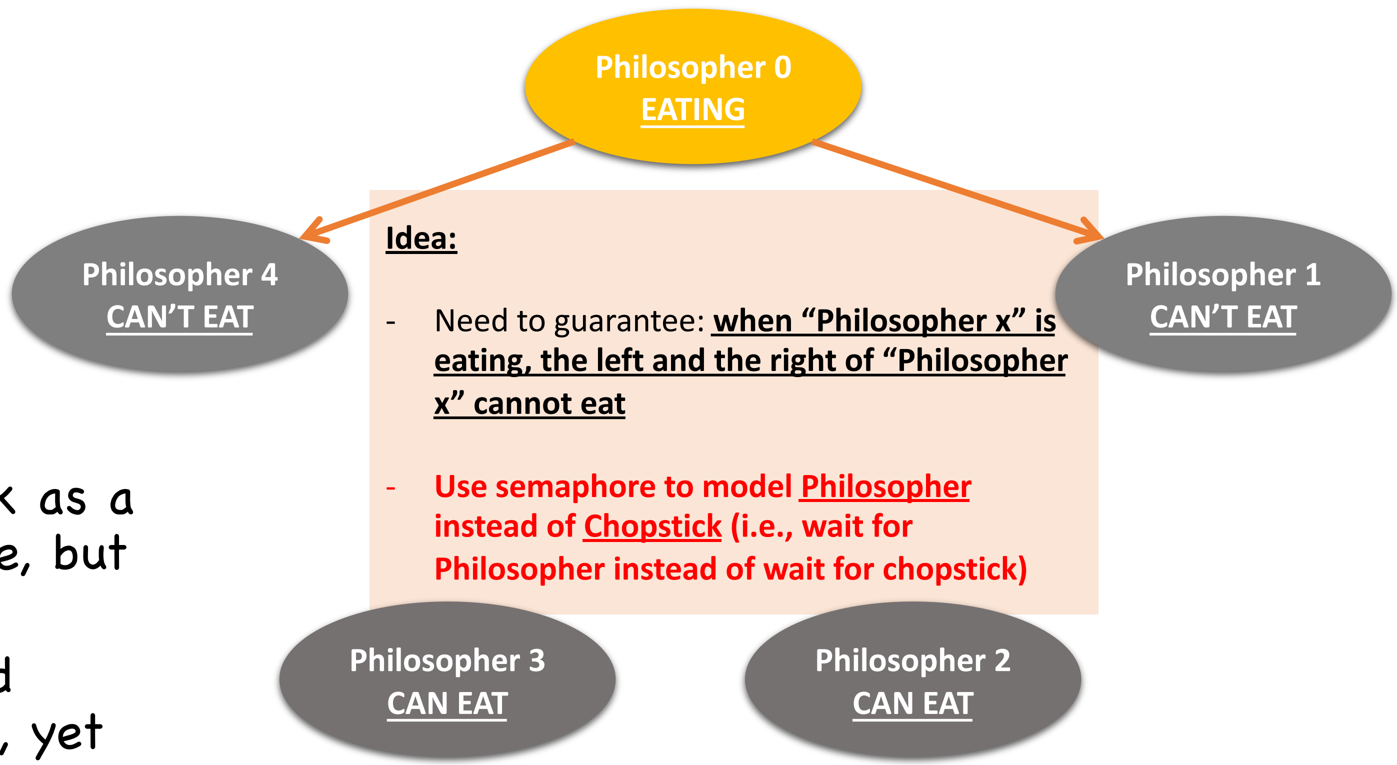
- How about the following suggestions:
  - First, a philosopher **takes a chopstick**.
  - If a philosopher finds that she cannot take the second chopstick, then she should **put it down**.
  - Then, the philosopher **goes to sleep** for a while.
  - When wake up, she retries
  - Loop until both chopsticks are seized.

Dining Philosopher – Requirement 2

- **Potential Problem:**
  - Philosophers are all busy (no deadlock), but no progress (starvation)

- Imagine:
  - all pick up their left chopsticks,
  - seeing their right chopsticks unavailable (because P1's right chopstick is taken by P2 as her left chopstick) and then putting down their left chopsticks,
  - all sleep for a while
  - all pick up their left chopsticks, ...

Dining Philosopher – before the Final Solution



- Two Problems
  - Model each chopstick as a semaphore is intuitive, but may cause deadlock
  - Using sleep() to avoid deadlock is effective, yet creating starvation.

Dining Philosopher – Final Solution

Shared object

```
#define N 5
#define LEFT  ((i+N-1) % N)
#define RIGHT  ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore p[N] = 0;
```

Identifying i's "left" and "right"

The states of the philosophers, including "**EATING**", "**THINKING**", and "**HUNGRY**".

Remember, this is a shared array.

To guarantee mutual exclusive access to the "`state[N]`" array.

To model a philosopher (instead of a chopstick)

Dining Philosopher – Final Solution

Shared object

```
#define N 5
#define LEFT  ((i+N-1) % N)
#define RIGHT  ((i+1) % N)

int state[N];
semaphore mutex = 1;
semaphore p[N] = 0;
```

Main function

```
void philosopher(int i) {
    think();
    take_chopsticks(i);
    eat();
    put_chopsticks(i);
}
```

```
void wait(semaphore *s) {
  *s = *s - 1;
  if ( *s < 0 ) {
    sleep();
  }
}
```

Section entry

```
void take_chopsticks(int i) {
    wait(&mutex);
    state[i] = HUNGRY;
    captain(i);
    post(&mutex);
    wait(&p[i]);
}
```

Section exit

```
void put_chopsticks(int i) {
    wait(&mutex);
    state[i] = THINKING;
    captain(LEFT);
    captain(RIGHT);
    post(&mutex);
}
```

```
void post(semaphore *s) {
  *s = *s + 1;
  if ( *s <= 0 )
    wakeup();
}
```

Extremely important helper function

```
void captain(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        post(&p[i]);
    }
}
```

Dining Philosopher – Final Solution

hungry

Tell the captain that you are hungry

If one of your neighbors is eating, the captain just does <u>nothing</u> for you and returns

Then, you wait for your chopsticks (later, the captain will notify you when chopsticks are available)

<u>Critical Section</u>
The captain is “indivisible”

Section entry

```
void take_chopsticks(int i) {
    wait(&mutex);
    state[i] = HUNGRY;
    captain(i);
    post(&mutex);
    wait(&p[i]);
}
```

Extremely important helper function

```
void captain(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        post(&p[i]);
    }
}
```

Dining Philosopher - Final Solution

Finish eating

Tell the captain
Try to let your **left neighbor** to eat.

Tell the captain
Try to let your right **neighbor** to eat.

Section exit

```
void put_chopsticks(int i) {
    wait(&mutex);
    state[i] = THINKING;
    captain(LEFT);
    captain(RIGHT);
    post(&mutex);
}
```

Extremely important helper function

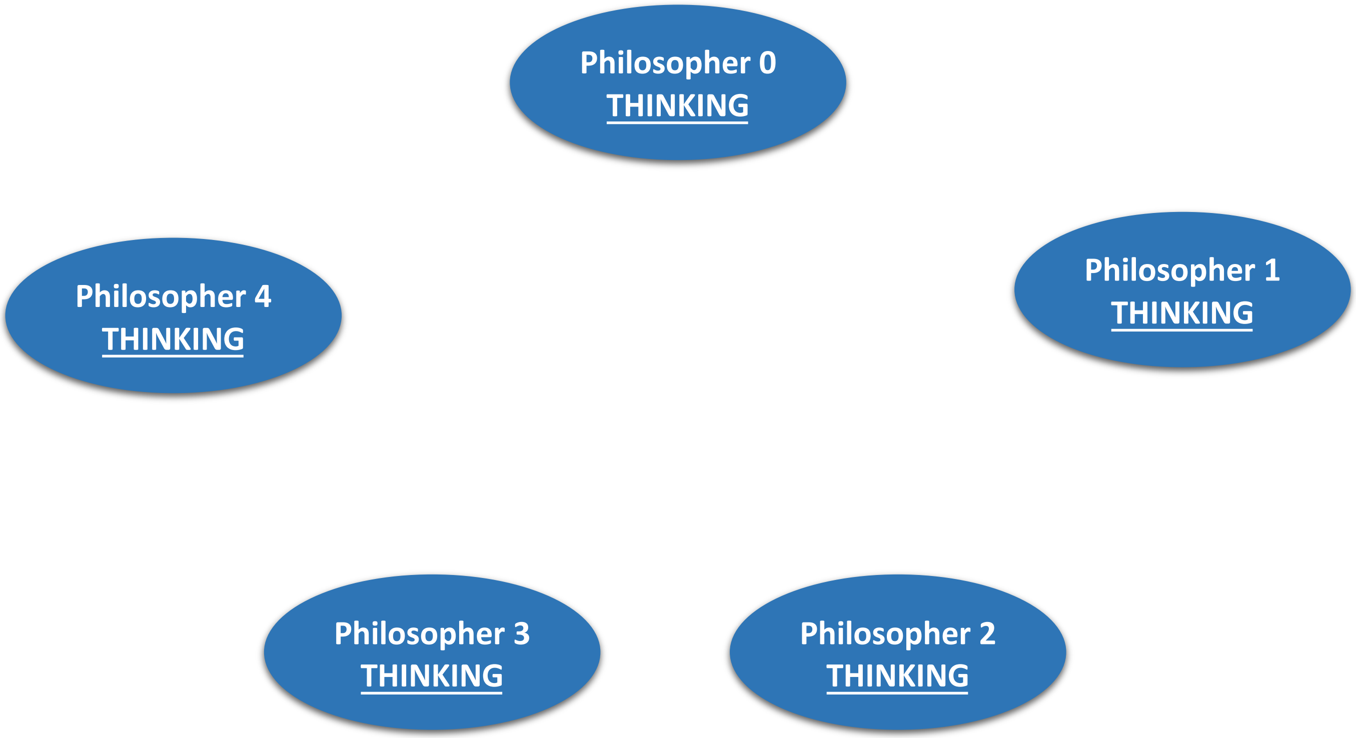
```
void captain(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        post(&p[i]);
    }
}
```

Wake up the one who is sleeping

Dining Philosopher – Final Solution

Don't print

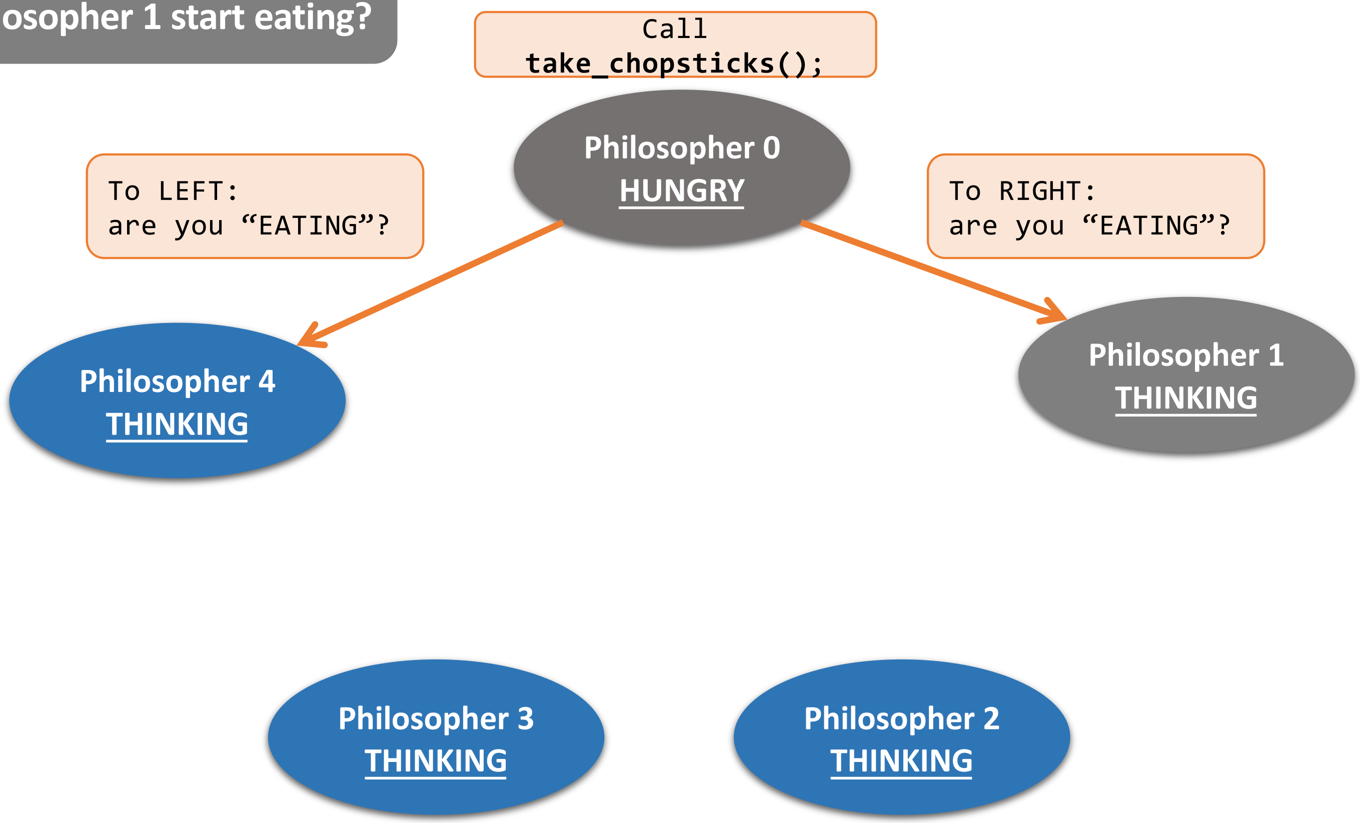
An illustration: How can Philosopher 1 start eating?



Dining Philosopher – Final Solution

Don't print

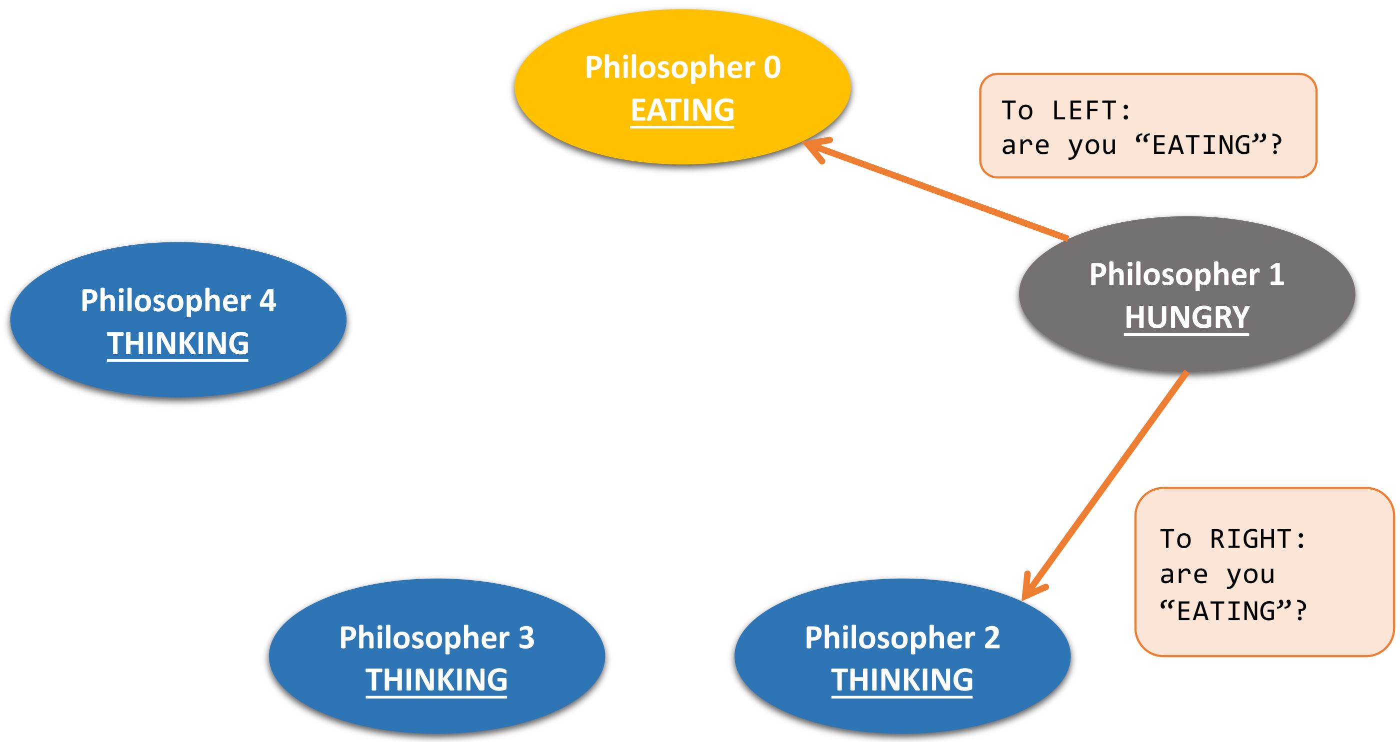
An illustration: How can Philosopher 1 start eating?



Dining Philosopher - Final Solution

Don't print

An illustration: How can Philosopher 1 start eating?



Dining Philosopher – Final Solution

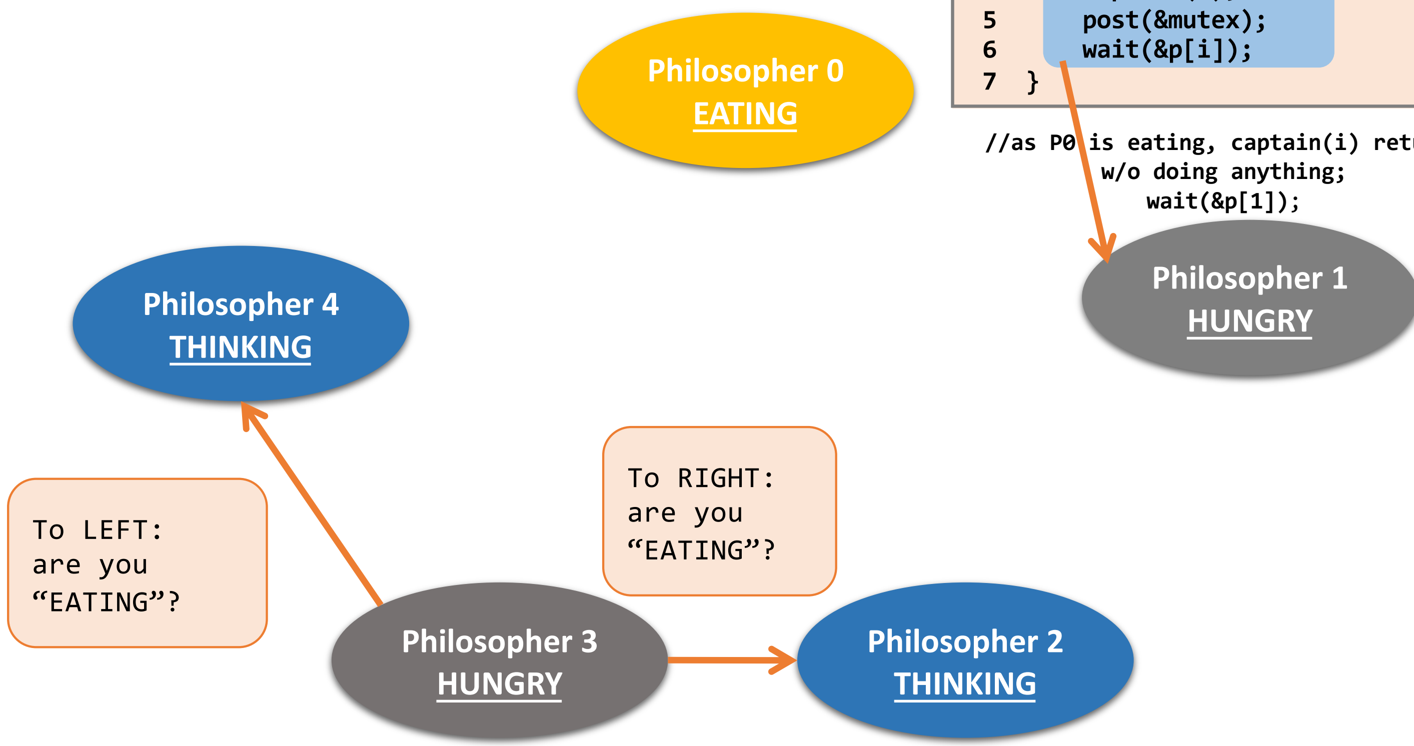
Don't print

An illustration: How can Philosopher 1 start eating?

Section entry

```
void take_chopsticks(int i) {
    wait(&mutex);
    state[i] = HUNGRY;
    captain(i);
    post(&mutex);
    wait(&p[i]);
}
```

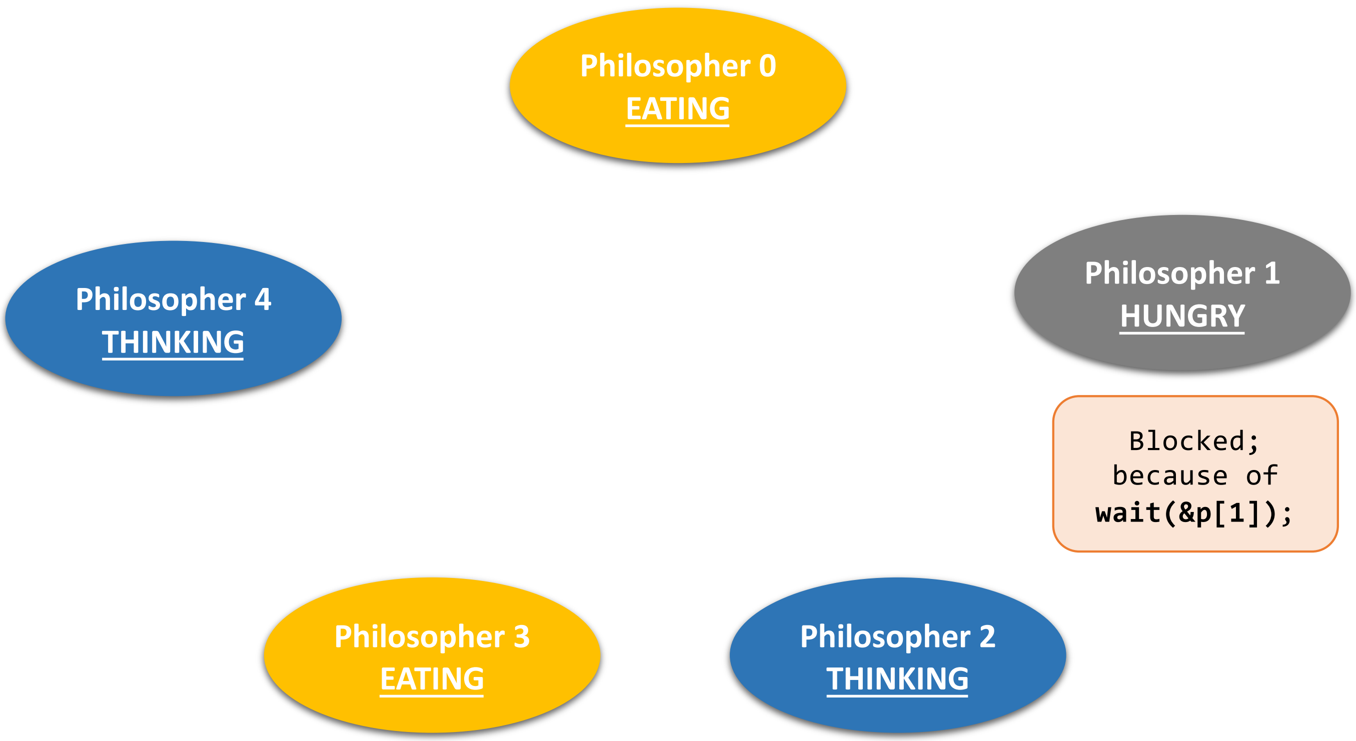
```
//as P0 is eating, captain(i) returns
   w/o doing anything;
      wait(&p[1]);
```



Dining Philosopher – Final Solution

Don't print

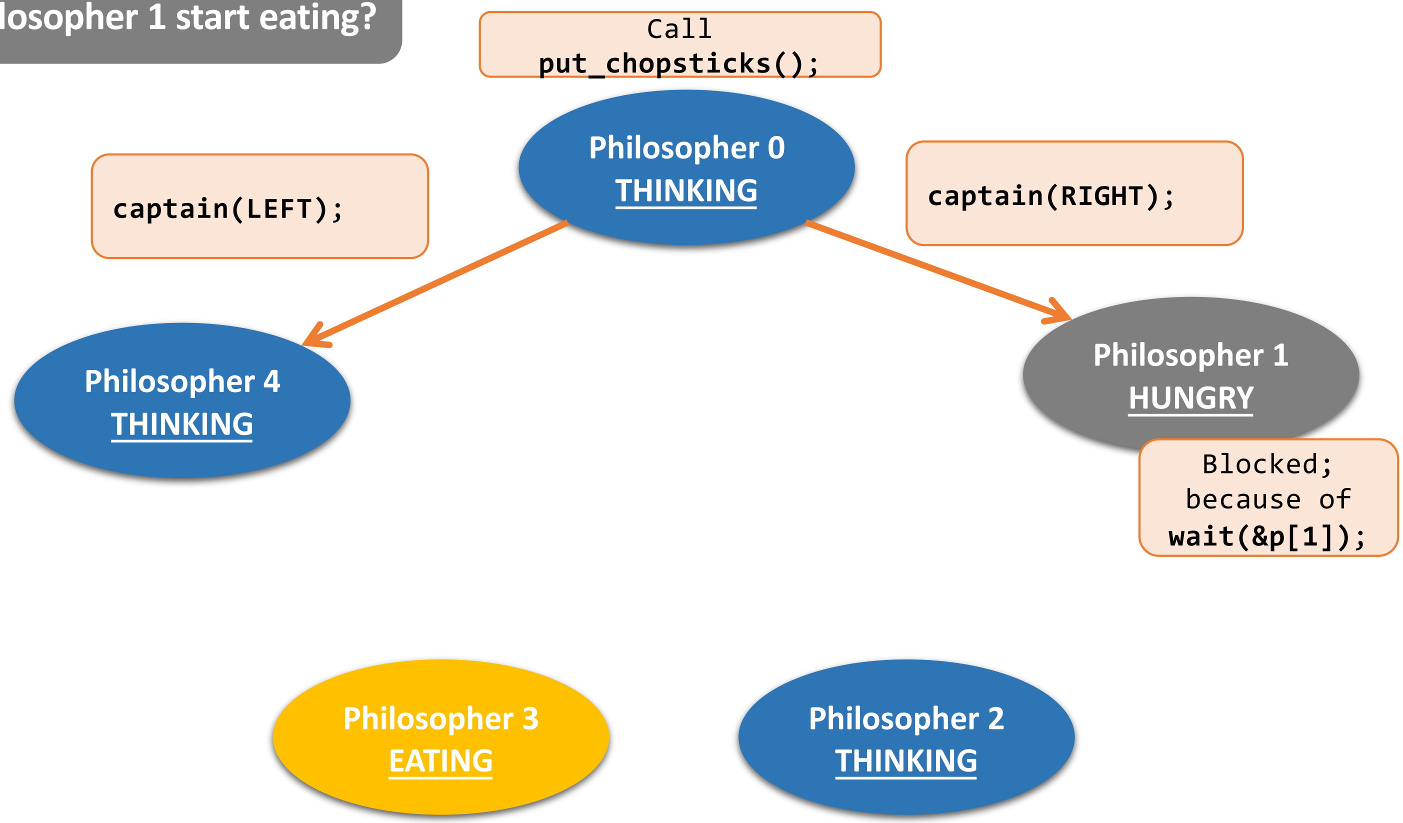
An illustration: How can Philosopher 1 start eating?



Dining Philosopher - Final Solution

Don't print

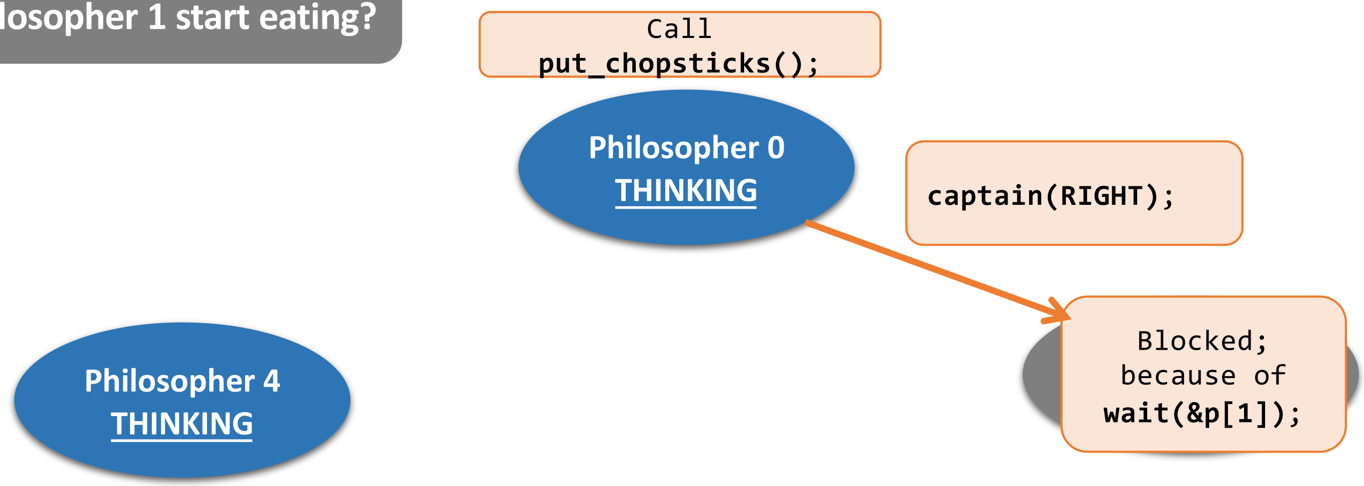
An illustration: How can Philosopher 1 start eating?



Dining Philosopher – Final Solution

Don’t print

An illustration: How can Philosopher 1 start eating?




```
void captain(int i) {
    if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        post(&p[i]);
    }
}
```

Wake up !

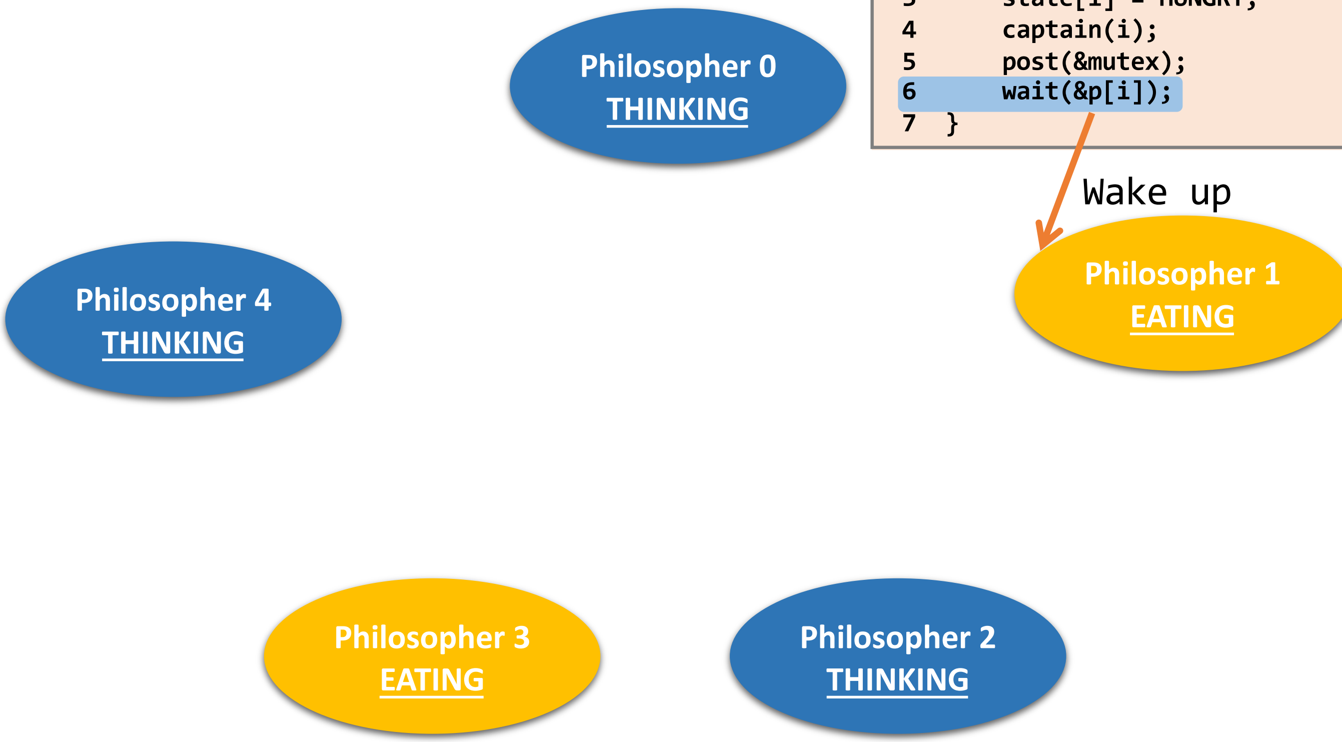
Dining Philosopher – Final Solution

Don't print

An illustration: How can Philosopher 1 start eating?

Section entry

```
void take_chopsticks(int i) {
    wait(&mutex);
    state[i] = HUNGRY;
    captain(i);
    post(&mutex);
    wait(&p[i]);
}
```



Thank you!