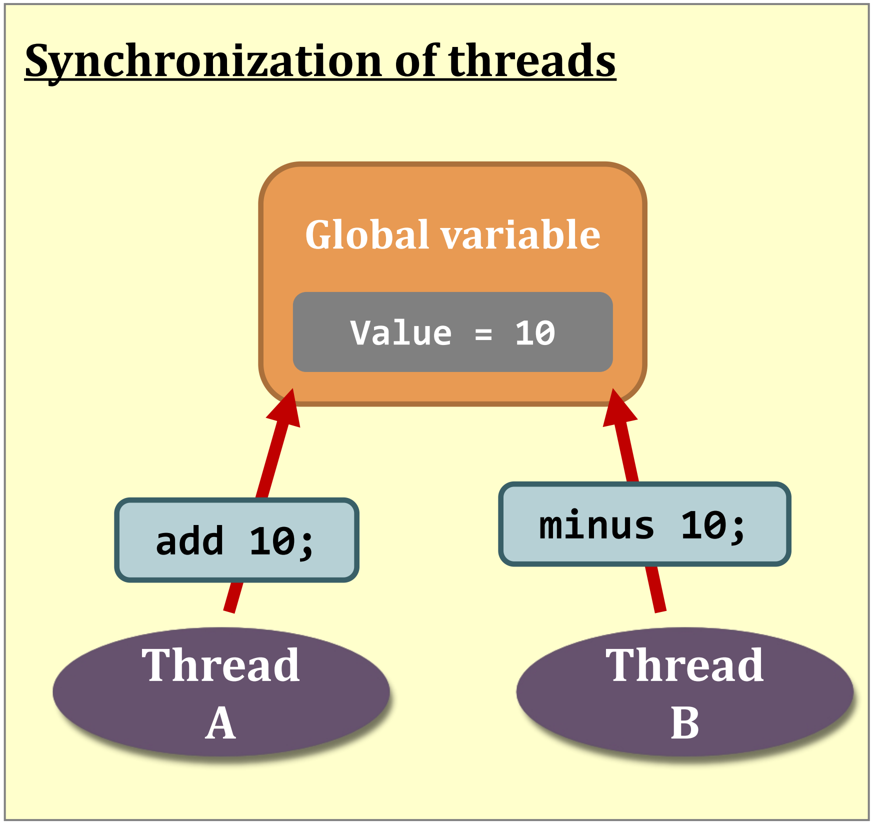
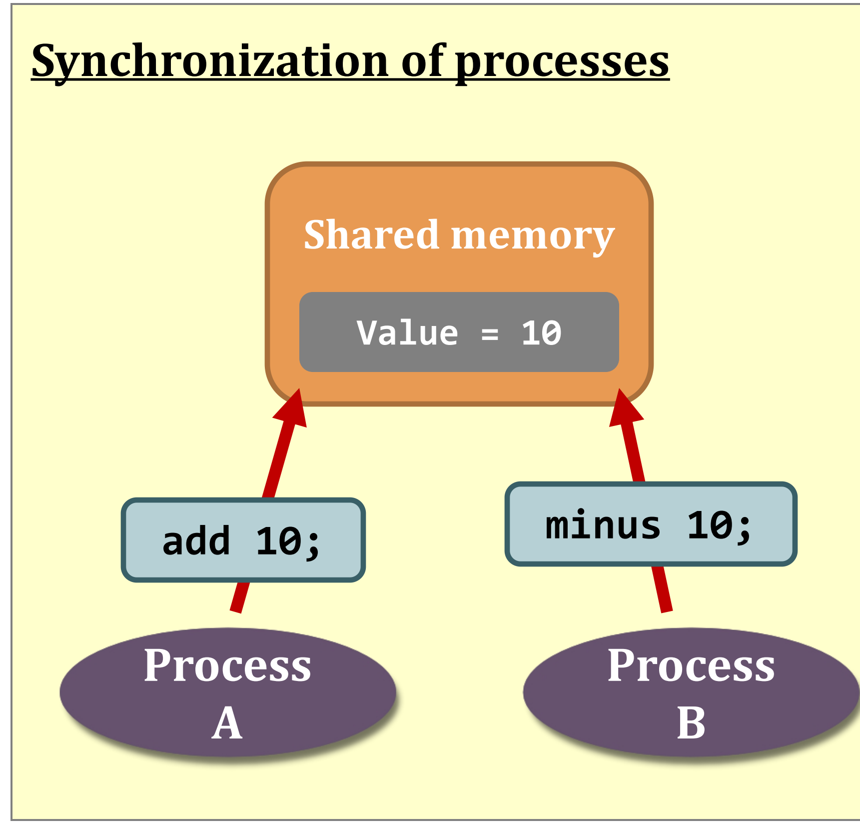
Lecture 6: Synchronization

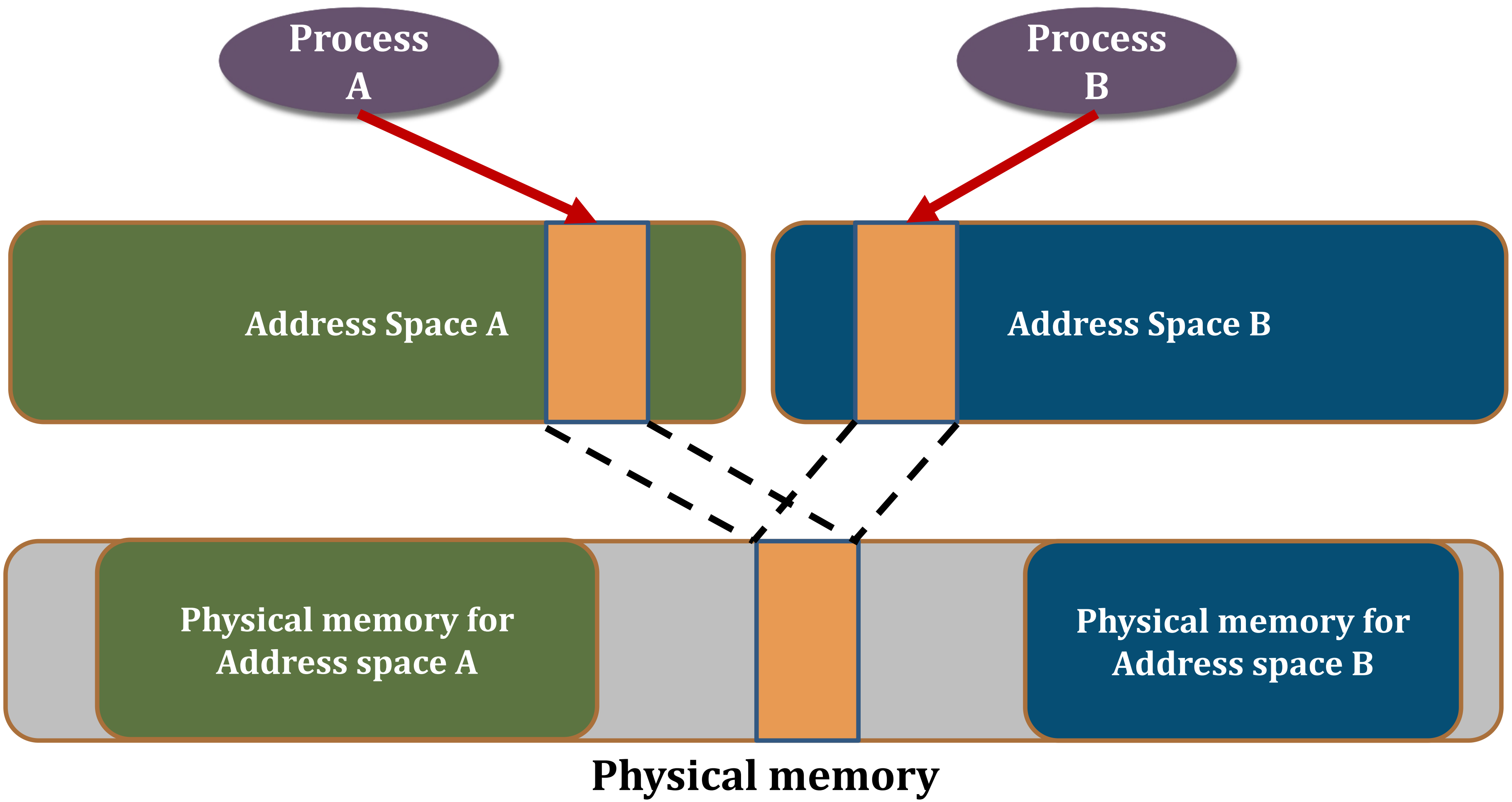
Yinqian Zhang @ 2021, Spring

Synchronization of threads/processes





Shared memory between processes



Race Condition: Understanding the problem

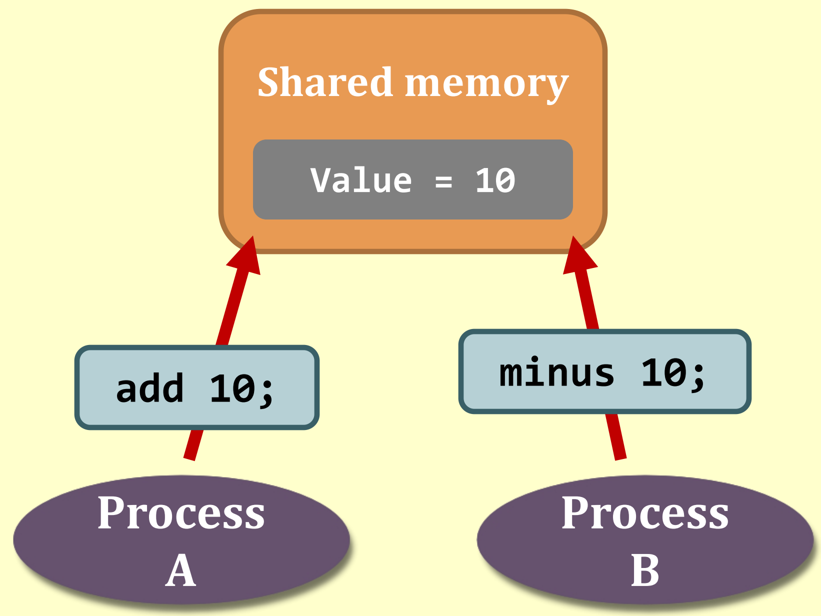
High-level language for Program A

```
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;
```

The Scenario



Race Condition

Don't print

Shared memory - X

`Value = 10`

Process A

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 B

State: Ready

```
Register B
Value = 0
```

```
2.1   load memory X to
      register B;

2.2   minus 10 from register B;

2.3   write register B to
      memory X;
```

The initial setting

Normal execution

Don't print

Shared memory - X

`Value = 10`

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 A


```
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;
```

Process B

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

Don't print

Normal execution

Shared memory - X

`Value = 10`

State: <u>Running</u>

```
Register A
Value = 20
```

```
2.1  load memory X to
     register A;

2.2  add 10 to register A;

2.3  write register A to
     memory X;
```

1

2

Process A


```
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;
```

Process B

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

Normal execution

Don't print

Shared memory - X

Value = 20

State: Running

Register A Value = 20

Register B Value = 0

State: Ready

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;

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 3

Normal execution

Don't print

Shared memory - X

`Value = 20`

State: Ready

`Register A Value = 20`

1
2
3

```
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

Context Switching

`Register B Value = 20`

State: Running

4

```
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 4

Normal execution

Don't print

Shared memory - X

`Value = 20`

Process A

State: <u>Ready</u>

```
Register A
Value = 20
```

1, 2, 3

```
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

State: <u>Running</u>

```
Register B
Value = 10
```

4, 5

```
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 #1, Step 5</u>

Normal execution

Shared memory - X

`Value = 10`

Process A	
State: Ready	`Register A Value = 20`

```
2.1   load memory X to
      register A;
2.2   add 10 to register A;
2.3   write register A to
      memory X;
```

1 2 3

Process A

Process B	
`Register B Value = 10`	**State: Running**

```
2.1   load memory X to
      register B;
2.2   minus 10 from register B;
2.3   write register B to
      memory X;
```

4 5 6

Process B

Execution Flow #1, Step 6

Don't print

Abnormal execution

Shared memory - X

Value = 10

State: Running

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 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 #2, Step 1

Don't print

Abnormal execution

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 = 10	**State: Running**

```
   2.1  load memory X to
        register B;             2

   2.2  minus 10 from register B;

   2.3  write register B to
        memory B;
```

Process A → **Context Switching** → **Process B**

Execution Flow #2, Step 2

Abnormal execution

Don't print

Shared memory - X

`Value = 10`

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 A

`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

Process B

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   minus 10 from register B;

2.3   write register B to
      memory B;
```

2

3

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

State: Ready

`Register B Value = 0`

```
2.1  load memory X to        2
     register B;
2.2  minus 10 from register B;  3
2.3  write register B to      4
     memory B;
```

Execution Flow #2, Step 5

Don't print

Abnormal execution

Shared memory - X

`Value = 20`

State: Running

`Register A Value = 20`

`Register B Value = 0`

State: Ready

Process A

1 `2.1 load memory X to register A;`

5 `2.2 add 10 to register A;`

6 `2.3 write register A to memory X;`

Process B

`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

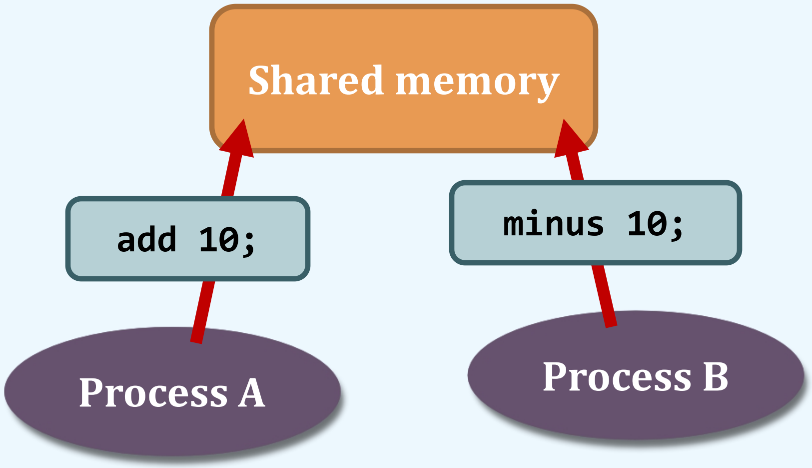
BUG!! T**he result is 20, but not 10**!

"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.

Mutual Exclusion – the cure

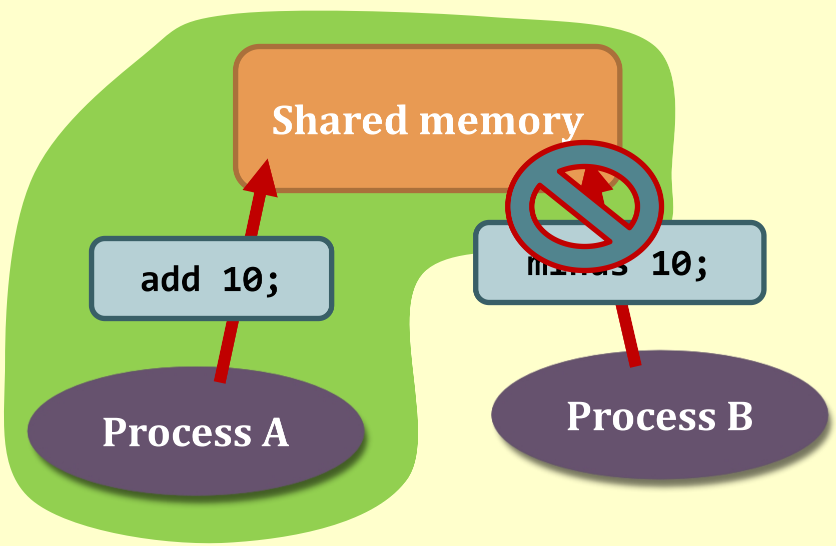


How to resolve race condition?

Solution: **mutual exclusion**

When I'm playing with the shared memory, no one could touch it.

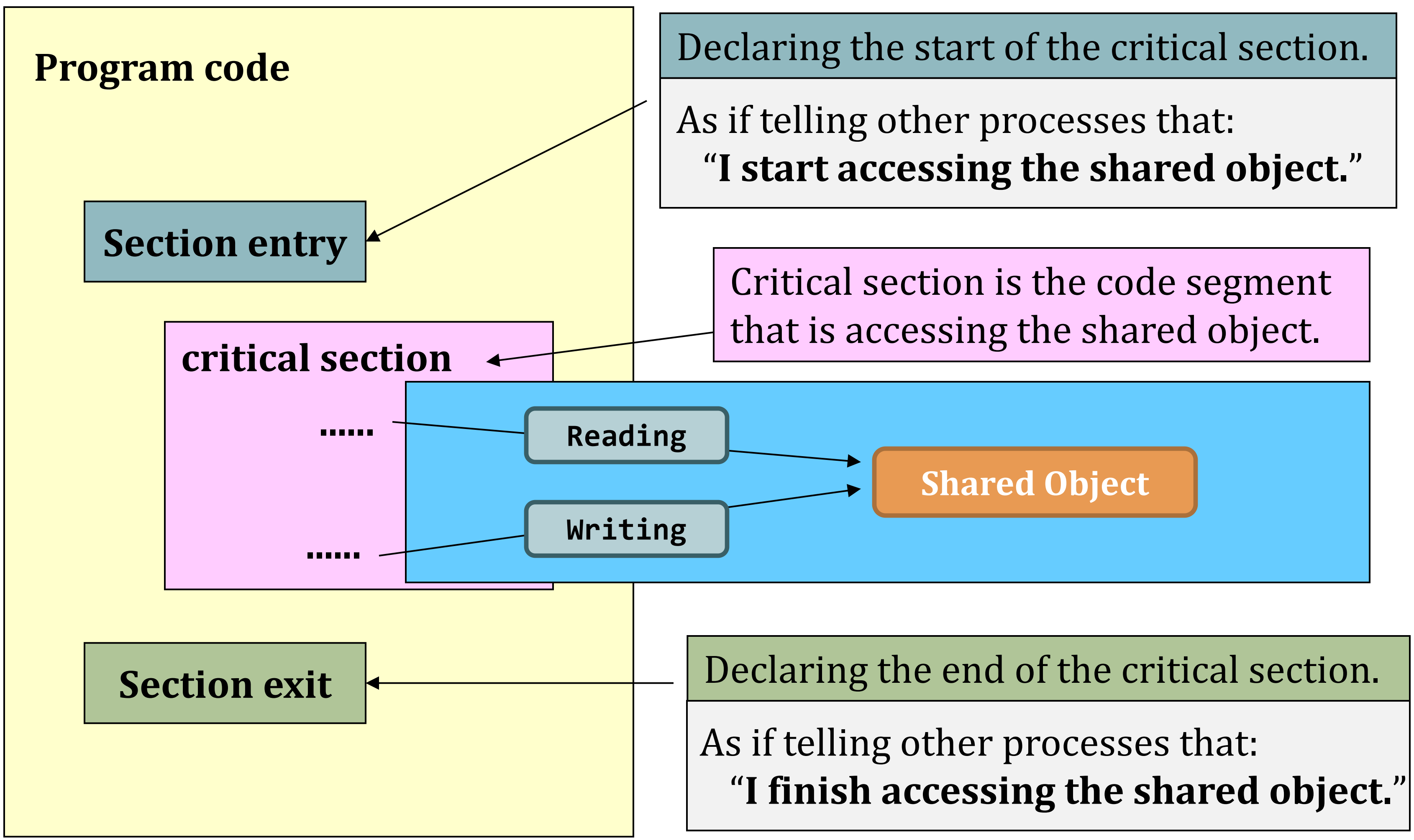
A set of processes would not have the problem of race condition *if* ***mutual exclusion is guaranteed***.



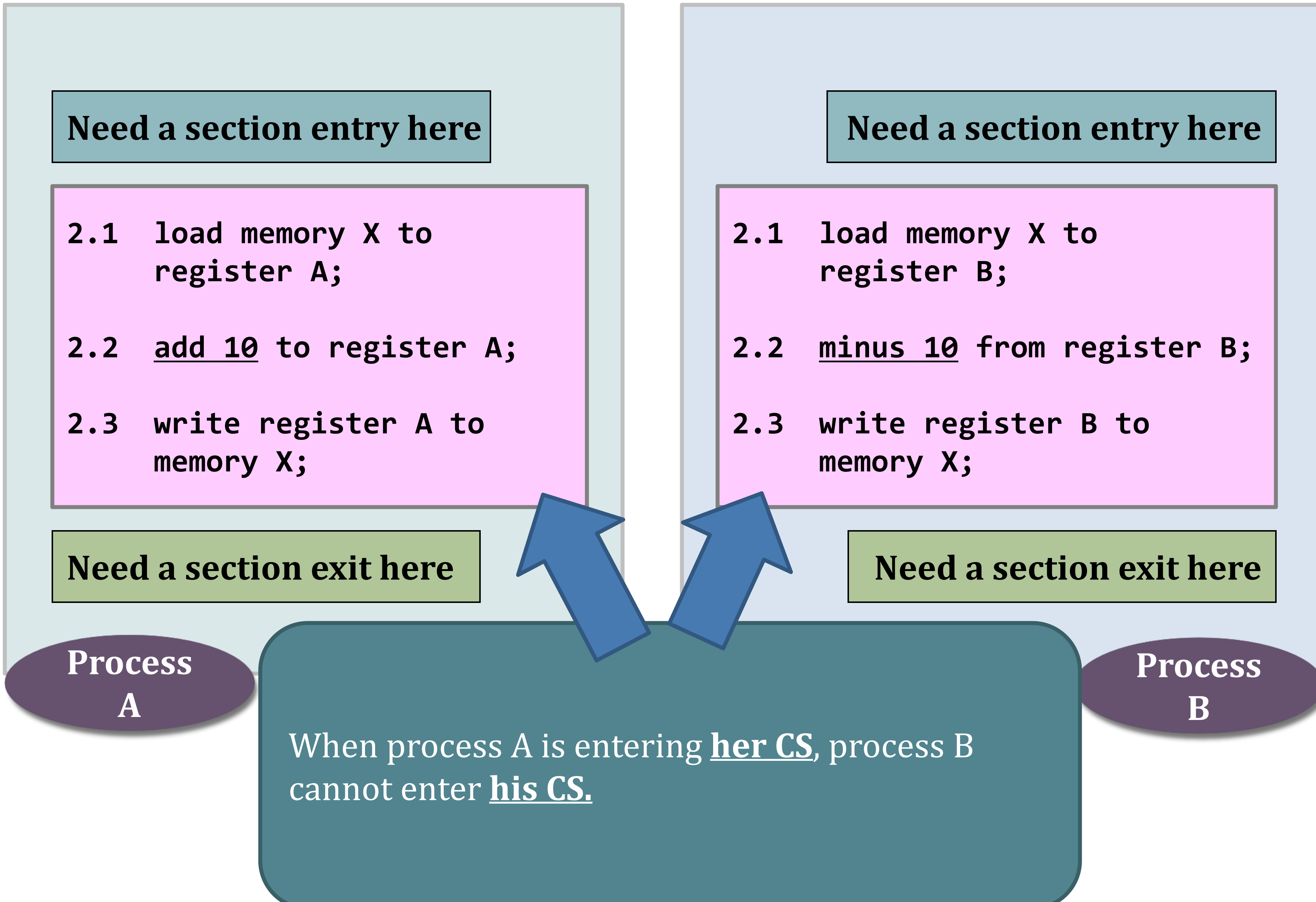
Solution – Mutual exclusion

- Shared object is still sharable, but
- Do not access the “shared object” <u>at the same time</u>
- Access the “shared object” one by one

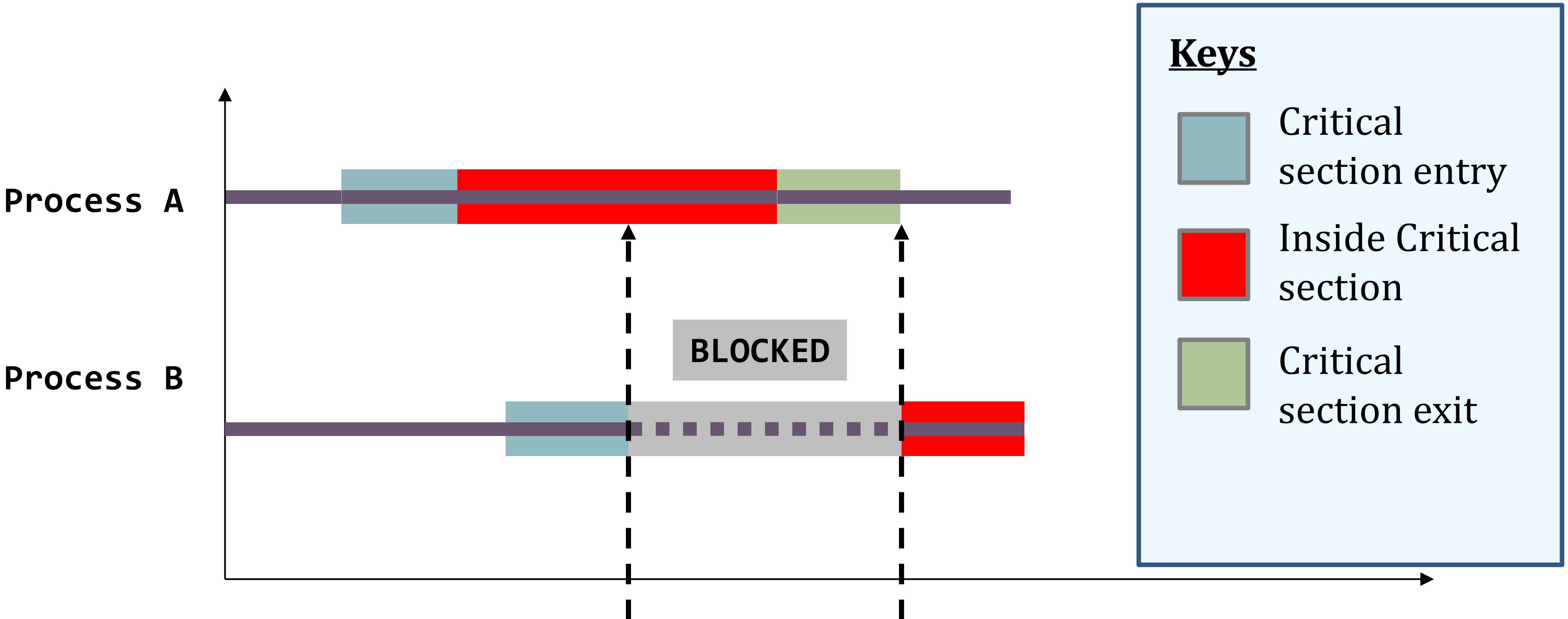
Critical Section – the realization



Critical Section (CS) – the realization



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.

What's really matter is the section entry/exit

Set of blocked processes, waiting to

Shared Object

Certainly, you want mutual exclusion!

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.

Summary

- **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**.

Summary

- **Implementing section entry and exit** is a challenge.
  - The entry and the exit are **the core parts that guarantee mutual exclusion**.
  - Unless they are correctly implemented, race condition would appear.
- **Mutual exclusion hinders the performance of parallel computations.**

Entry and exit implementation - requirements

- **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 her CS, there is a bound on the number of times other processes can enter theirs.

Entry and exit implementation - requirements

- **Requirement #3**. **Progress**
  - Say no process currently in C.S.
  - One of the processes trying to enter will eventually get in

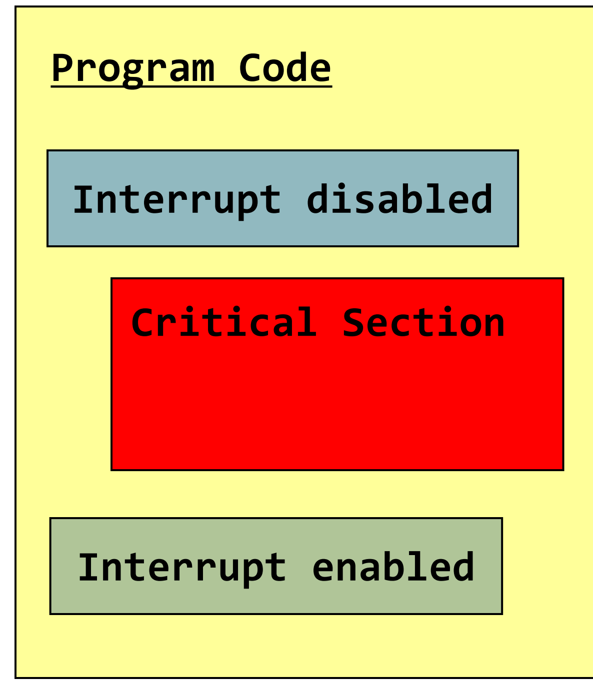
https://en.wikipedia.org/wiki/Non-blocking_algorithm

Progress vs. bounded waiting

- If no process can enter C.S, do not have *progress*
- If A waits to enter its C.S, while B repeated leaves and re-enters its C.S *repeatedly*
- A does not have *bounded waiting (but B is having progress)*

#0 – disabling interrupt for the whole CS

- **Aim**
  - To **disable context switching** when the process is inside the critical section.
- **Effect**
  - When a process is in its critical section, no other processes could be able to run.
- **Correctness?**
  - **Uni-core: Correct but not permissible**
    - at userspace: what if one writes a CS that loops infinitely and the other process (e.g., the shell) never gets the context switch back to kill it?
    - At 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!!!!)



Achieving Mutual Exclusion

- Lock-based
  - Use yet another shared objects: ***locks***
    - ***What about race condition on lock?***
    - Atomic instructions: instructions that cannot be “interrupted”

  - Spin-based lock
    - Process synchronization
      - Basic spinning using 1 shared variable
      - Peterson’s solution: Spin using 2 shared variables
    - Thread synchronization
      - `pthread_spin_lock`

  - Sleep-based lock
    - Process synchronization
      - POSIX semaphore
    - Thread synchronization
      - `pthread_mutex_lock`
- Lock-free

#1: Basic Spin lock (busy waiting)

- **Idea.**
  - Loop on another shared object, **`turn`**, to detect the status of other processes

Shared object "turn" `initial value = 0`

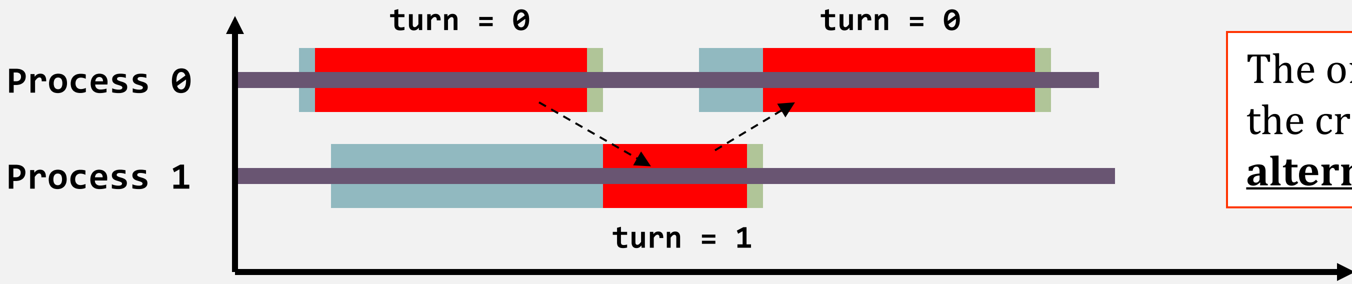
```
while (TRUE) {
    while( turn != 0 )
      ; /* busy waiting */
    critical_section();
    turn = 1;
    remainder_section();
}
```

Process 0

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

Process1

#1: Basic Spin lock (busy waiting)



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

Shared object "turn" `initial Value = 0`

Process 0

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

   critical_section();

   turn = 1;

   remainder_section();
}
```

Process1

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

   critical_section();

   turn = 0;

   remainder_section();
}
```

#1: Basic Spin lock (busy waiting)

- Correct
  - but it wastes CPU resources
  - OK for short waiting
    - Especially these days we have multi-core
      - Will not block other irrelevant processes a lot
    - Ok when spin-time < context-switch-overhead
- Impose a “strict alternation” order
  - Sometimes you give me my turn but I’m not ready to enter CS yet
    - Then you have to wait long

#1: Basic Spin lock violates *progress*

Don't print

- 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()`

Turn = 1

```
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

#1: Basic Spin lock violates *progress*

Don't print

Turn = 1

- 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

#2: Spin Smarter (by Peterson's solution)

- Highlight:
  - Use one more extra shared object: **`interested`**
    - If I am not **interested**
      - I let you go
    - If we are both **interested**
      - Take turns

Shared objects:
- turn &
- "interested[2]"

#2: Spin Smarter (by Peterson's solution)

```
int turn;                                /* who is last enter cs */
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 */
}
```

#2: Spin Smarter (by 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;
}
```

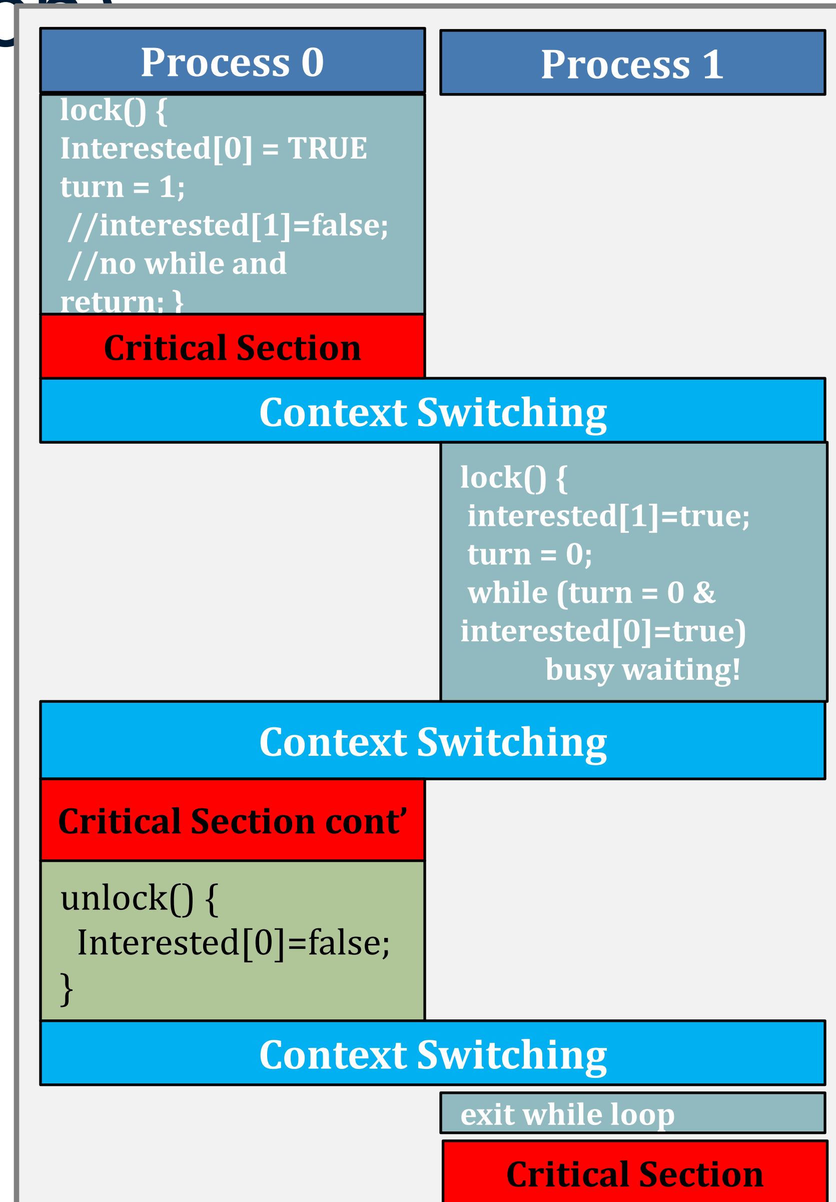
Express interest to enter CS

Being polite and let other go first

If other is not interested, I can always go ahead

#2: Spin Smarter (by 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;
}
```



#2: Spin Smarter (by Peterson's solution) (another case)

```
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  }
```

Process 0	Process 1

lock() {
interested[0]=TRUE;
turn = 1;	

Context Switching	

	lock() {
	interested[1]=TRUE;
	turn = 0;

Context Switching	

turn == 1 ? //line 9
false
no busy waiting	

Critical Section	
Context Switching	

	turn == 0? //line 9
	&& interested[0]=true?
	busy waiting!

…

Spin Smarter (by Peterson's solution)

- = Busy waiting
  + shared variable **`turn`** for mutual exclusion
  + shared variables **`interest`** to resolve strict alternation
- **Peterson's solution satisfies all three criteria! (Why?)**
  - *"It satisfies the three essential criteria to solve the critical section problem, provided that changes to the variables turn, interest[0], and interest[1] propagate immediately and atomically."*---wikipedia

- Suffer from **priority inversion problem**

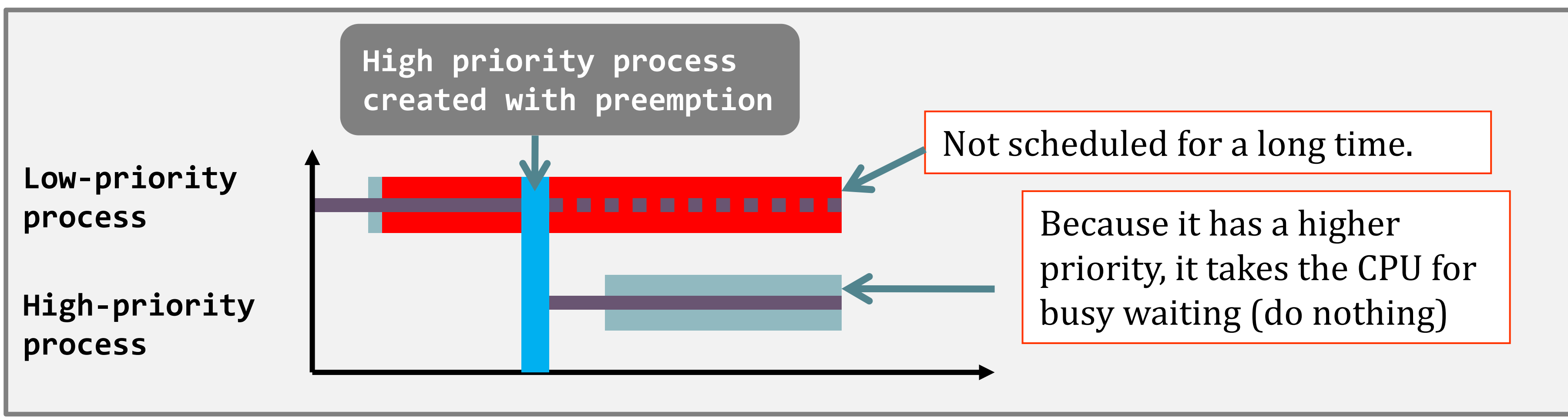
Does it work for >2 processes?
https://en.wikipedia.org/wiki/Peterson's_algorithm

Peterson's solution satisfies three criteria

- 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_1 selected first
  - When P_1 exit, interested[1] = false
    - If P_0 runs fast: interested[1] == false, P_0 enters critical section
    - If P_1 runs fast: interested[1] = true, but turn = 0, P_0 enters critical section

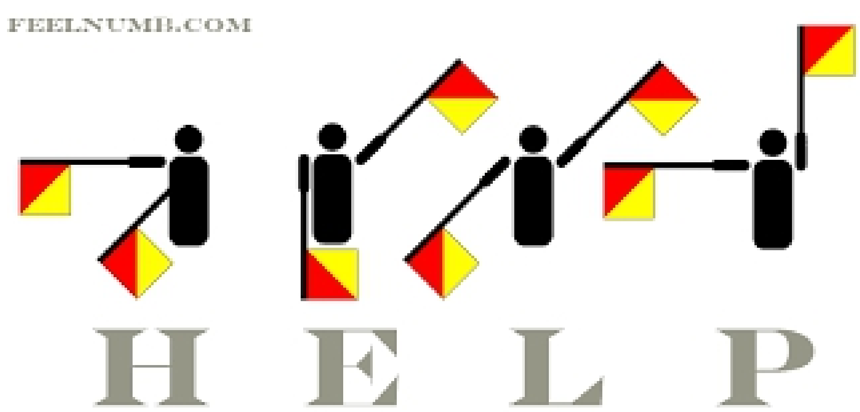
Peterson spinlock suffers from Priority Inversion

- 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** can not **`lock`** (because **L** has not **`unlock`**)
    - So, **H** gets the CPU to do nothing but spinning



#3: 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
  - Need to interact with scheduler (must involve kernel, e.g., syscall)
  - Implement uninterruptable **section entry/exit**
    - Section entry/exit function are **short**
      - **Compared with Implementation #0 (uninterruptable throughout the whole CS)**



https://en.wikipedia.org/wiki/Semaphore_(programming) check out the library analogy

Semaphore **logical** view

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

Section Entry: sem_wait()

```
void sem_wait(semaphore *s) {
   disable_interrupt();
   s->value = s->value – 1;
   if ( s->value < 0 ) {
      enable_interrupt();
      sleep();
      disable_interrupt();
    }
   enable_interrupt();
}
```

Initialize **s** = 1

"**sem_wait(s)**"

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

Important
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) {
   disable_interrupt();
   s->value = s->value + 1;
   if ( s->value <= 0 )
     wakeup();
   enable_interrupt();
}
```

Example

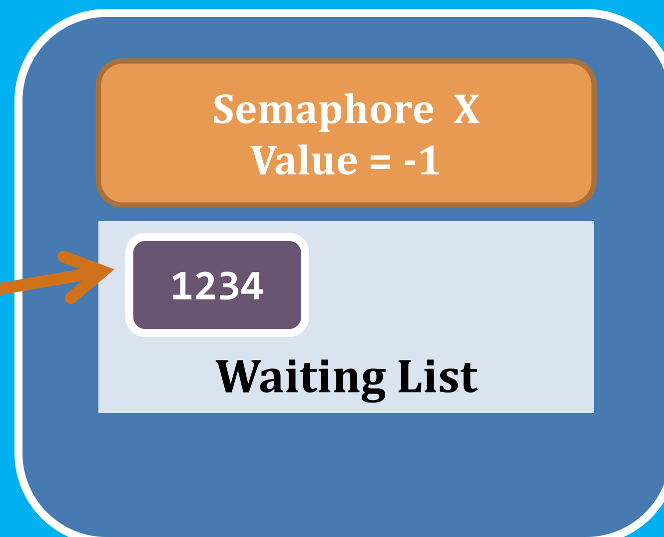
Process 1234

Sem_wait(X)

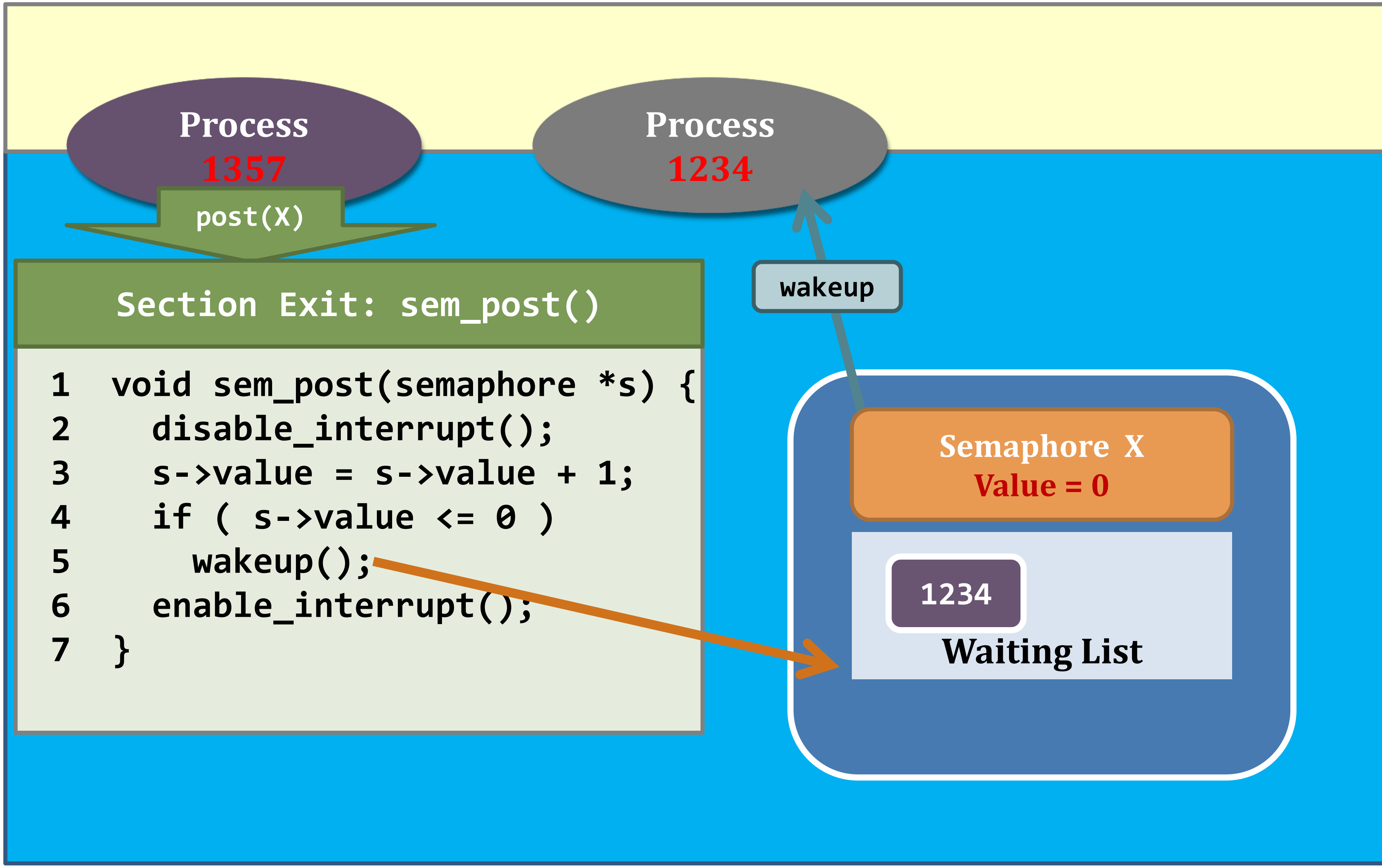
Assuming someone else (process 1357) has already taken the only one resource:
X = 1 (initial) => **X** = 0
Now, process 1234 arrives

Section Entry: sem_wait()

```
void sem_wait(semaphore *s){
    disable_interrupt();
    s->value = s->value – 1;
    if ( s->value < 0 ) {
        enable_interrupt();
        sleep();
        disable_interrupt();
     }
    enable_interrupt();
}
```

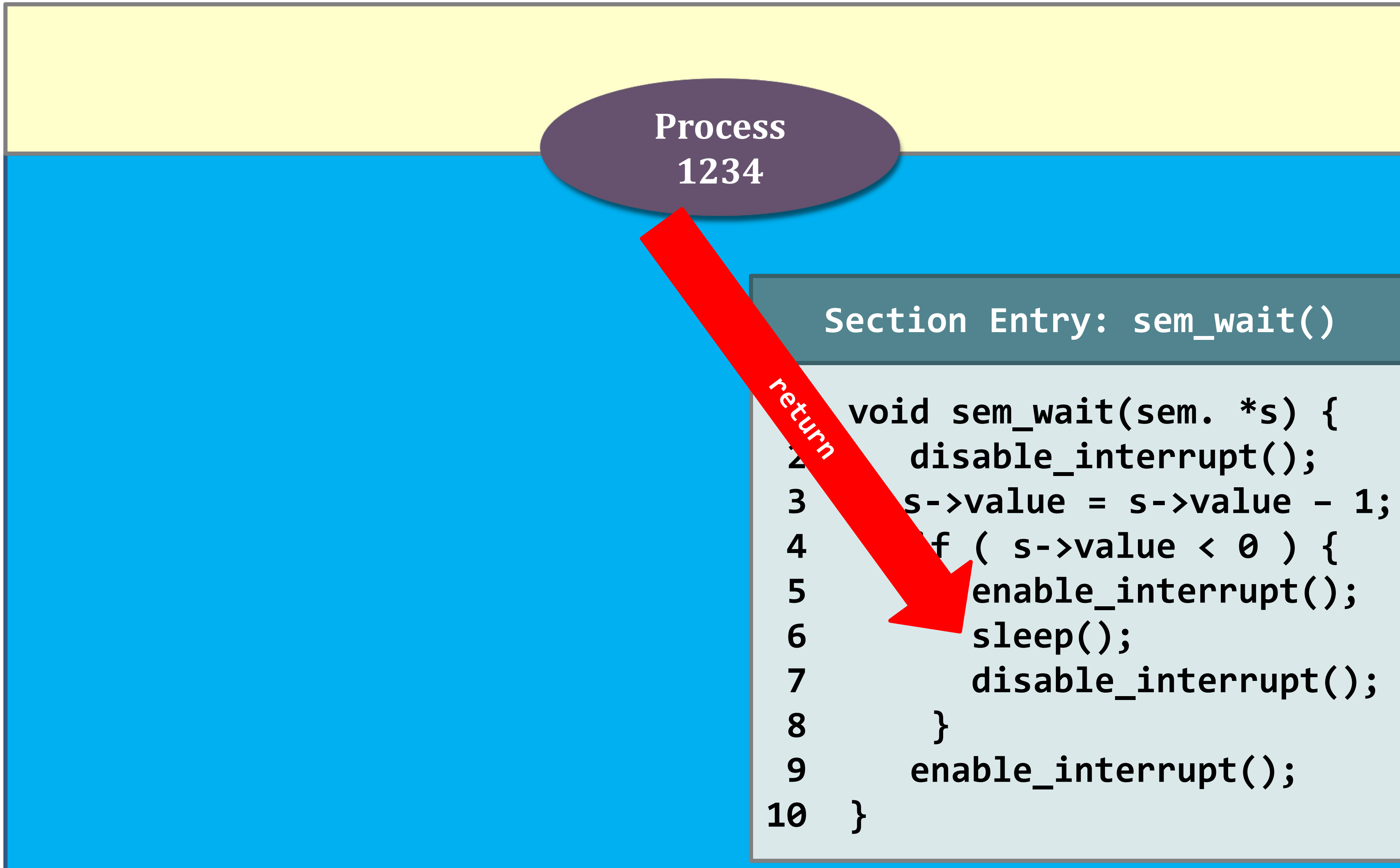


Example



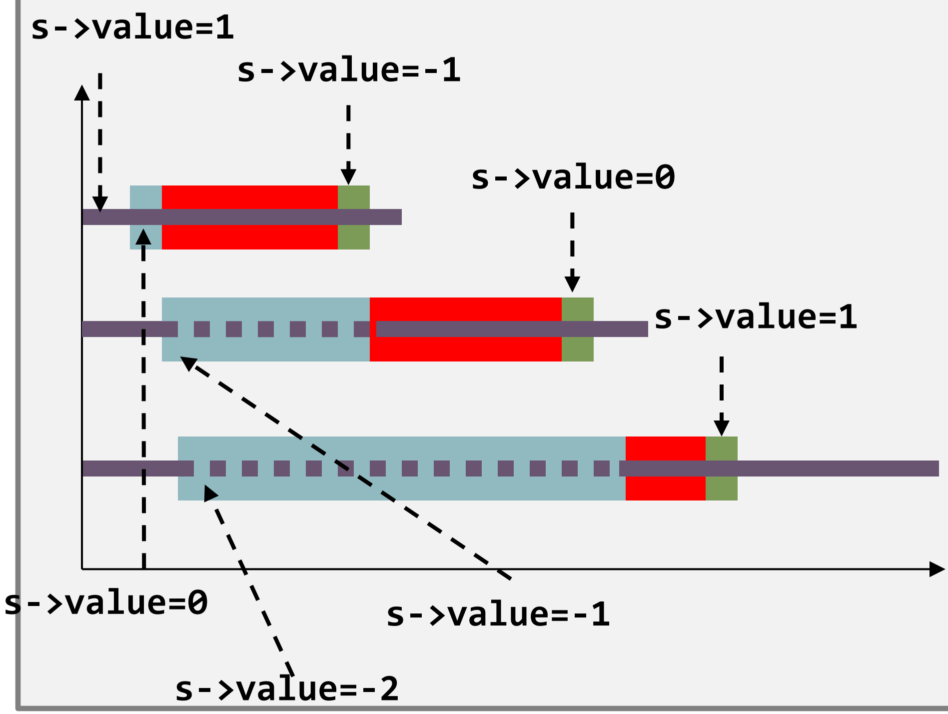
Process
1357
post(X)
Process
1234
wakeup
Section Exit: sem_post()
void sem_post(semaphore *s) {
disable_interrupt();
s->value = s->value + 1;
if (s->value <= 0)
wakeup();
enable_interrupt();
}
Semaphore X
Value = 0
1234
Waiting List

Example




```
void sem_wait(sem. *s) {
    disable_interrupt();
    s->value = s->value – 1;
    if ( s->value < 0 ) {
        enable_interrupt();
        sleep();
        disable_interrupt();
    }
    enable_interrupt();
}
```

Using Semaphore (user-level)




```
semaphore *s;   /* from kernel */
s->value = 1; /* initial value */
```

```
while(TRUE) {
    sem_wait(s);
    critical_section();
    sem_post(s);
}
```

entry

exit

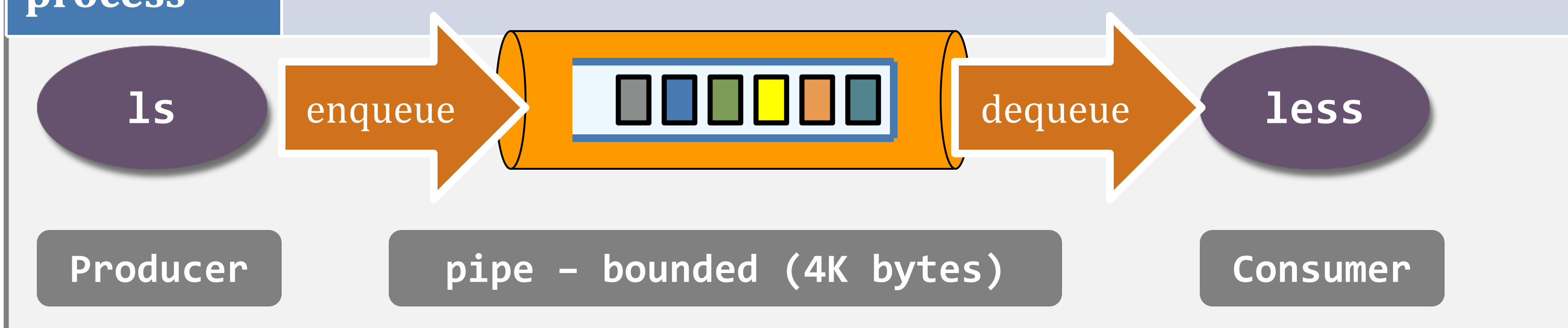
Using Semaphore beyond mutual exclusion

	Properties
Producer-Consumer Problem	Two classes of processes: **producer** and **consumer**; At least one producer and one consumer. [Single-Object Synchronization]
Dining Philosopher Problem	They are all running the same program; At least two processes. [Multi-Object Synchronization]
Reader Writer Problem	Multiple reads, 1 write
...	

Producer-consumer problem – introduction

- 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 that 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.

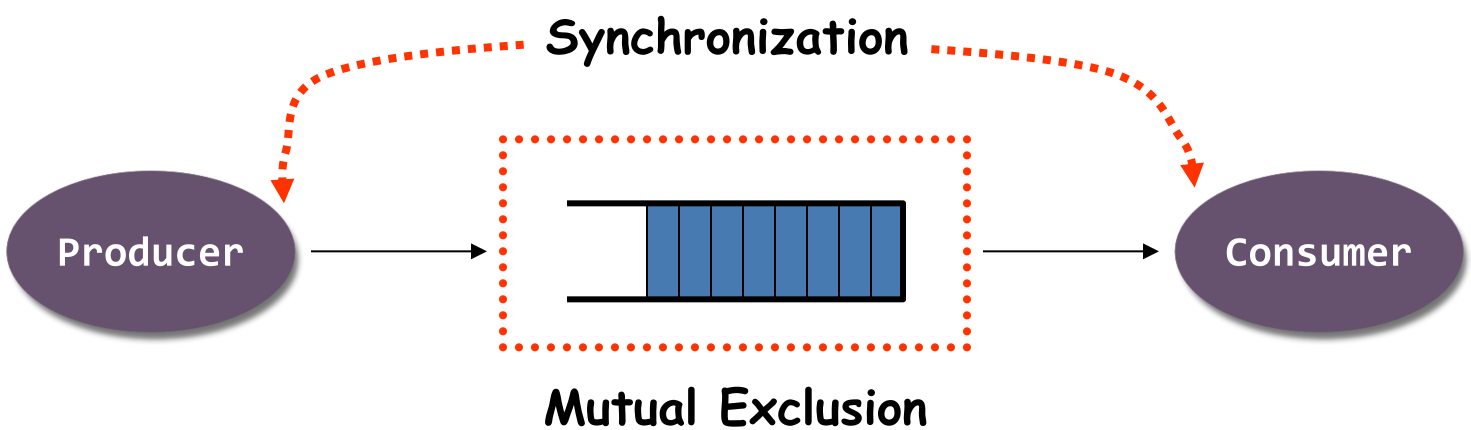


Producer-consumer problem – introduction

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.

Producer-consumer problem: semaphore

- The problem can be divided into two sub-problems.
  - Mutual exclusion.
    - The buffer is a shared object. **Mutual exclusion** is needed. Done by one **binary semaphore**
  - Synchronization.
    - Because the buffer's size is bounded, **coordination** is needed. Done by two 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



Producer-consumer problem: 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;
    }
}
```

Producer-consumer problem: semaphore

<u>Note</u>
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  }
```

Producer-consumer problem: 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

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  }
```

Producer-consumer problem – 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;
     }
}
```

Producer-consumer problem – 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;
    }
}
```

Producer-consumer problem – 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

E R R O R

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;
    }
}
```

Producer-consumer problem – 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;
```

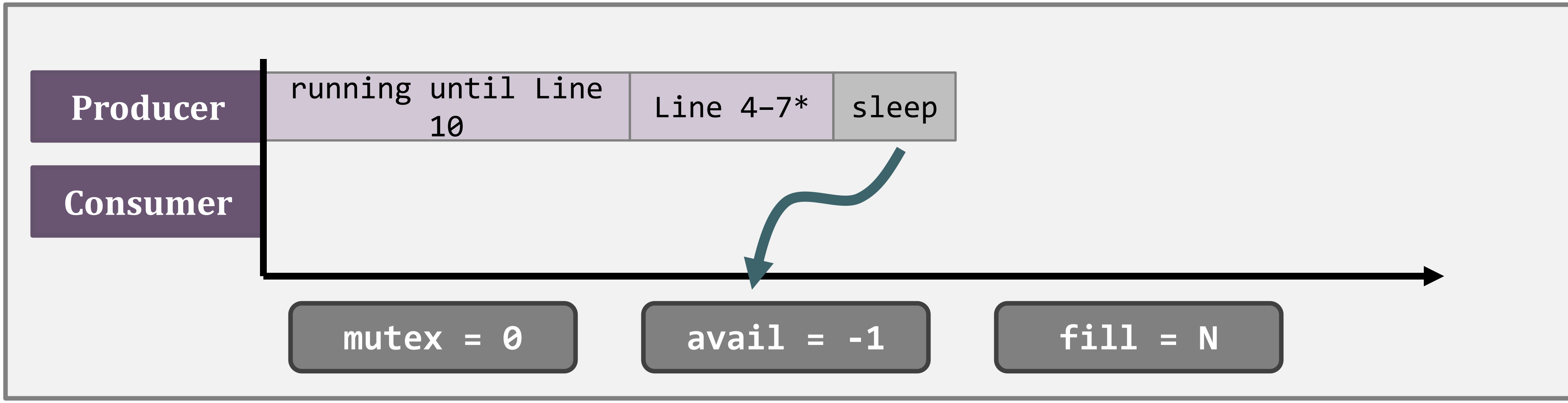
Producer function

```
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
    }
}
```

Producer-consumer problem – question #2



Producer function

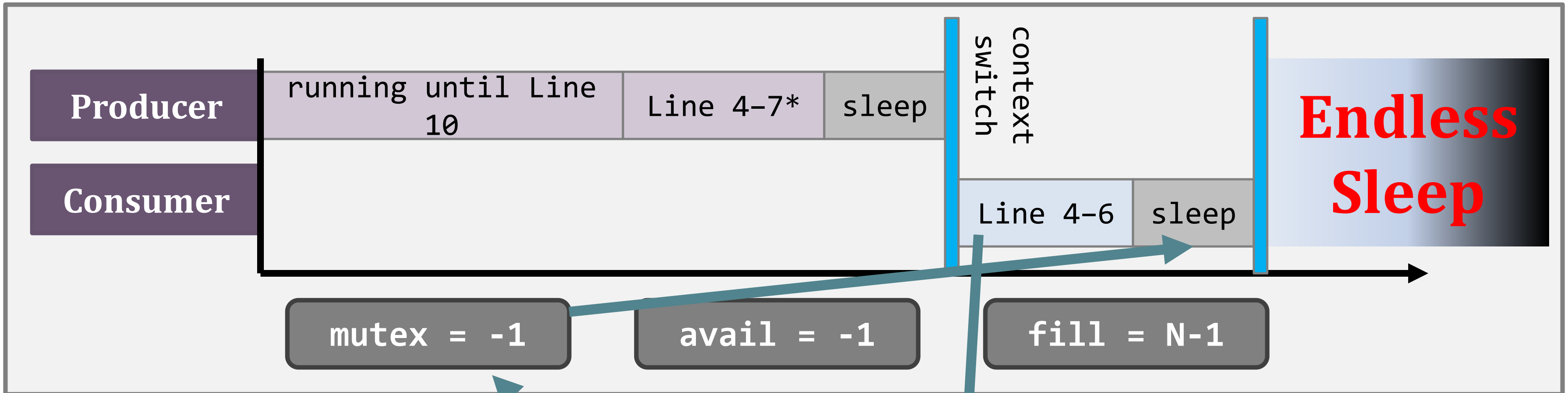
```
1   void producer(void) {
2       int item;
3
4       while(TRUE) {
5           item = produce_item();
6*          wait(&mutex);
7*          wait(&avail);
8           insert_item(item);
9           post(&mutex);
10          post(&fill);
11      }
12  }
```

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

Producer-consumer problem – 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

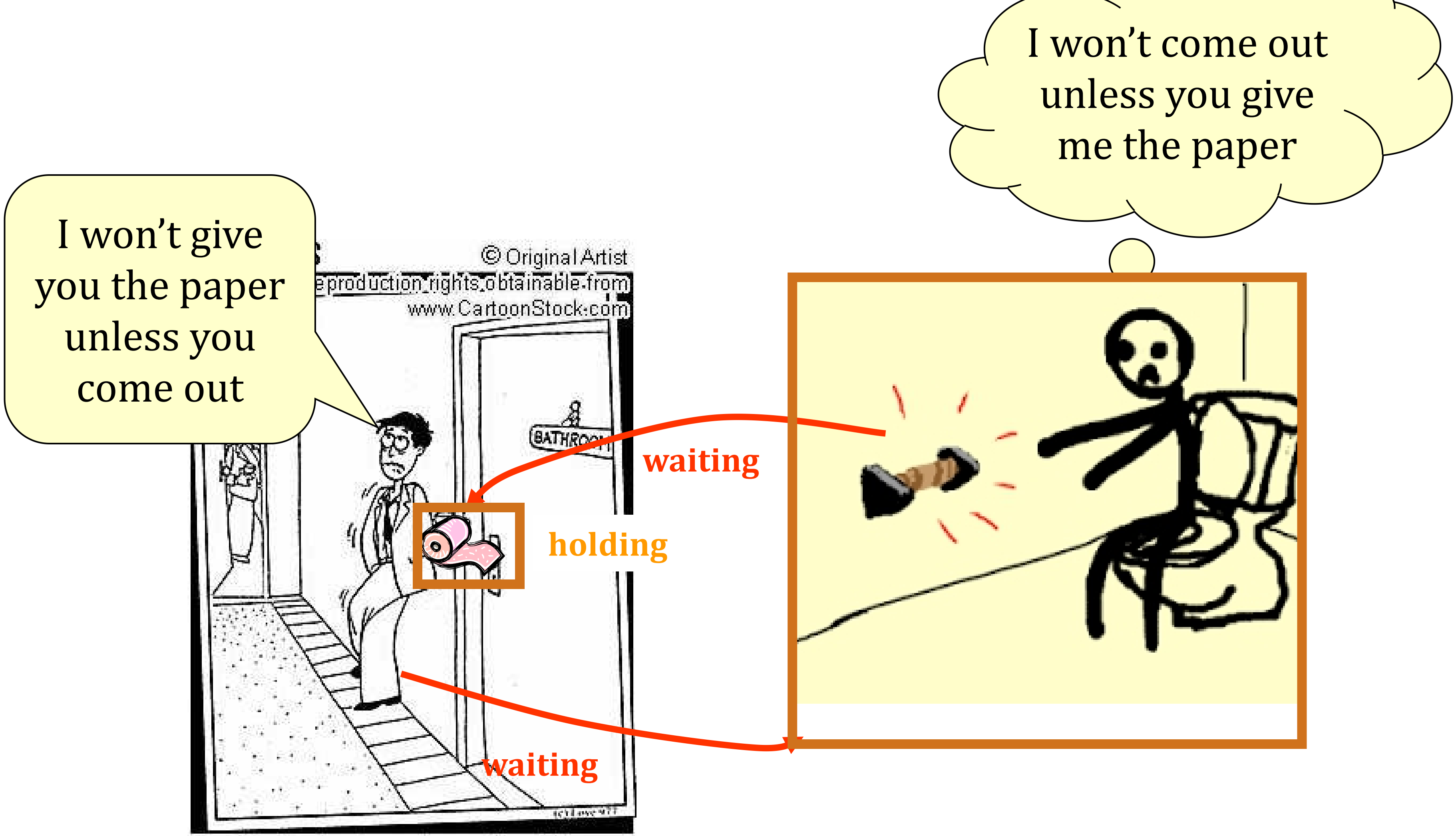
```
1   void consumer(void) {
2       int item;
3
4       while(TRUE) {
5           wait(&fill);
6           wait(&mutex);
7           item = remove_item();
8           post(&mutex);
9           post(&avail);
10          //consume the item
11      }
12  }
```

Producer-consumer problem – 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.

Deadlock


I won't come out unless you give me the paper
I won't give you the paper unless you come out
© Original Artist
eproduction rights obtainable from
www.CartoonStock.com
waiting
holding
waiting

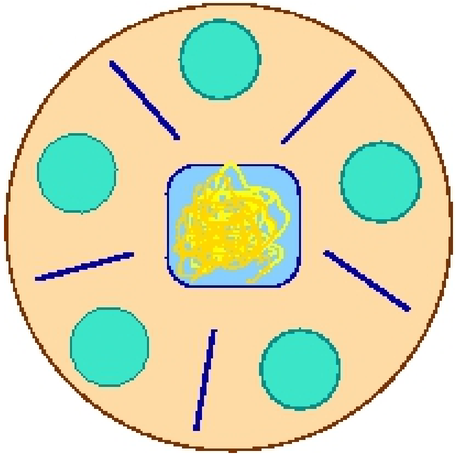
Summary on producer-consumer problem

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

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

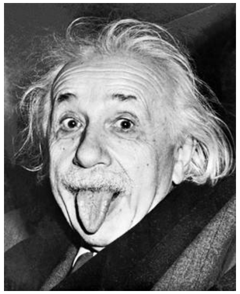
Dining philosopher – introduction

- 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 ....



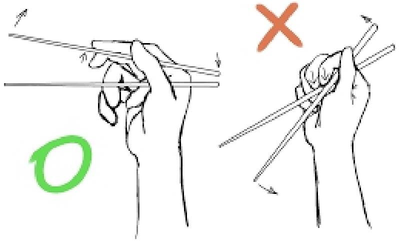
It's a multi-object synchronization problem

Dining philosopher – introduction



Philosophers

Process



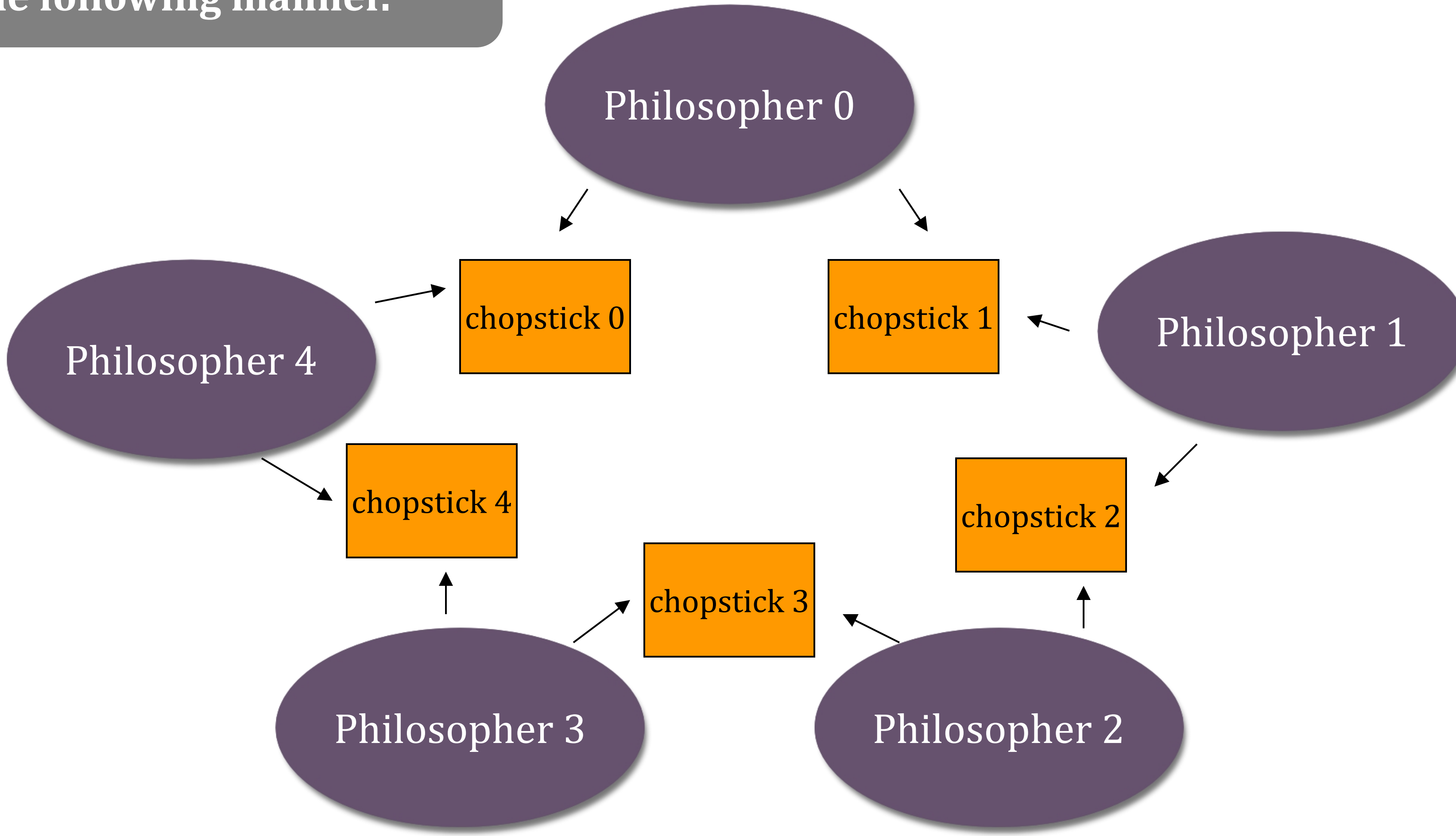
Chopsticks

Shared Object

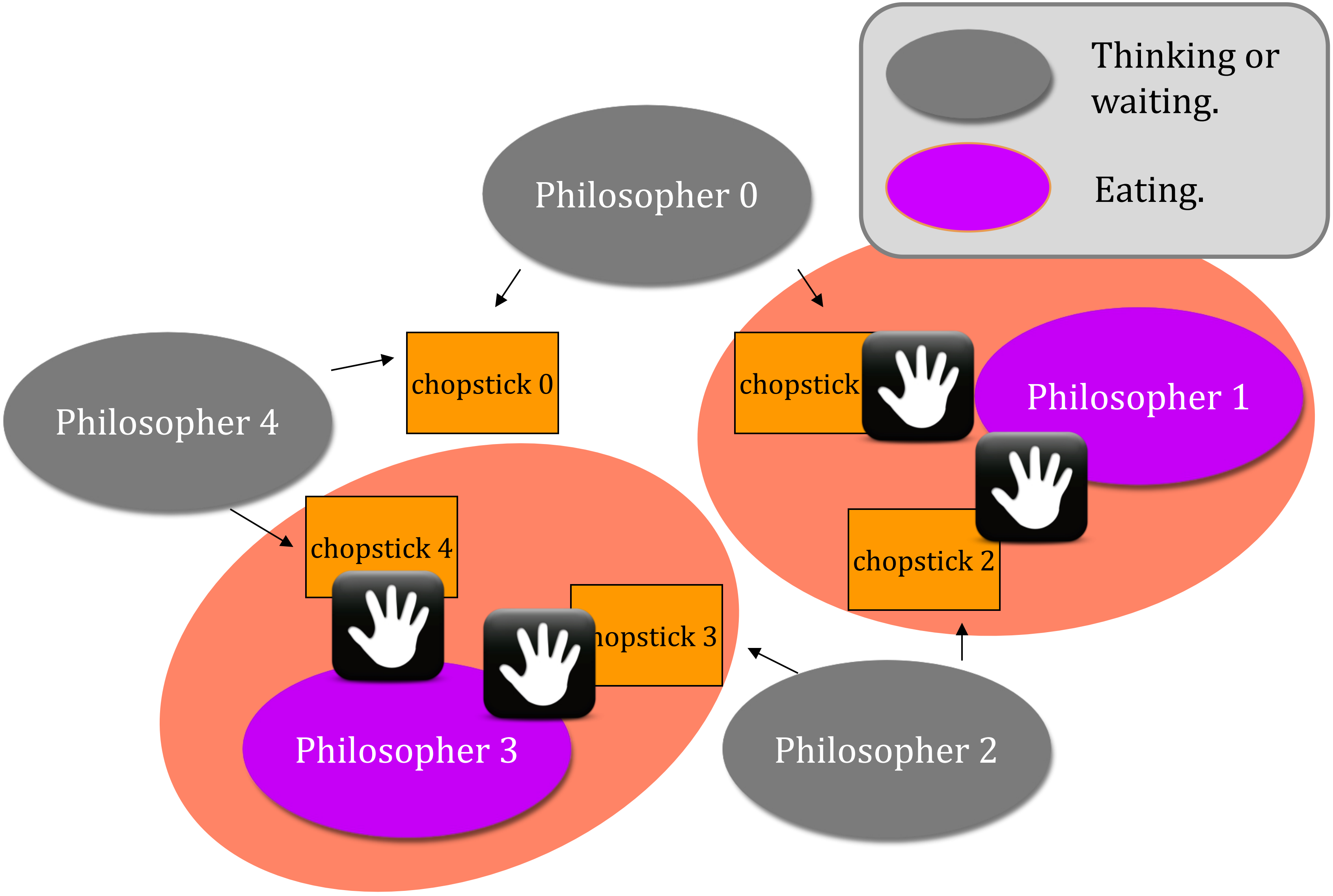
A process needs two shared resources in order to do some work

Dining philosopher – introduction

The chopsticks are arranged in the following manner.



Dining philosopher – introduction



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 – meeting 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

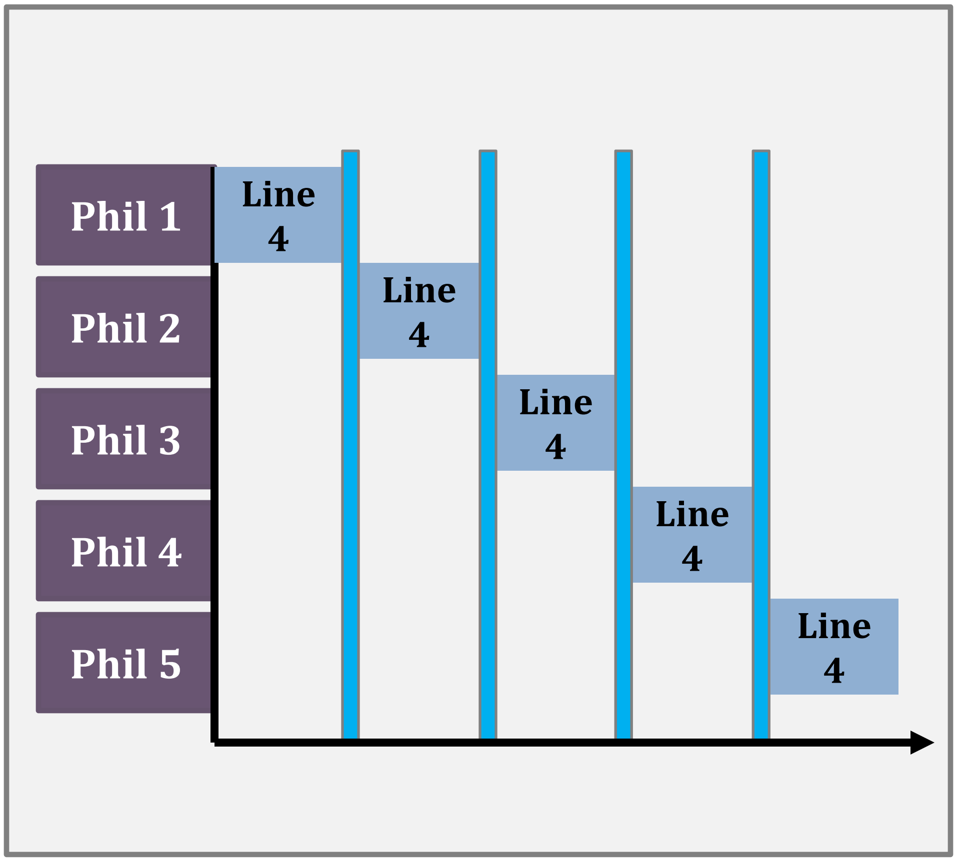
Section Exit

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 – meeting requirement #2?

<u>Potential Problem</u>: Philosophers are all busy (no deadlock), but no progress (<u>starvation</u>)

```
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

- Before we present the final solution, let us see what problems we have.

Problems
<u>Model each chopstick as a semaphore is intuitive, but may cause deadlock</u>
<u>Using `sleep()` to avoid deadlock is effective, yet creating starvation.</u>

Dining philosopher – before the final solution.

Philosopher 0
<u>EATING</u>

Philosopher 4
<u>CAN'T EAT</u>

Philosopher 1
<u>CAN'T EAT</u>

Idea:

- Need to guarantee: **when "Philosopher x" is eating, the left and the right of "Philosopher x" cannot eat**

- **Use semaphore to model Philosopher instead of Chopstick (i.e., wait for Philosopher instead of wait for chopstick)**

Philosopher 3
<u>CAN EAT</u>

Philosopher 2
<u>CAN EAT</u>

Dining philosopher – the 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 – the 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) {
   disable_interrupt();
    *s = *s - 1;
   if ( *s < 0 ) {
      enable_interrupt();
      sleep();
      disable_interrupt();
    }
   enable_interrupt();
}
```

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) {
  disable_interrupt();
  *s = *s + 1;
  if ( *s <= 0 )
    wakeup();
  enable_interrupt();
}
```

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 – Hungry

Tell the captain that you are hungry

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

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

Section entry

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

Critical Section

The captain is “indivisible”

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 – 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 – the final soluti

Don’t print

An illustration: How can Philosopher 1 start eating?

Philosopher 0
THINKING

Philosopher 1
THINKING

Philosopher 4
THINKING

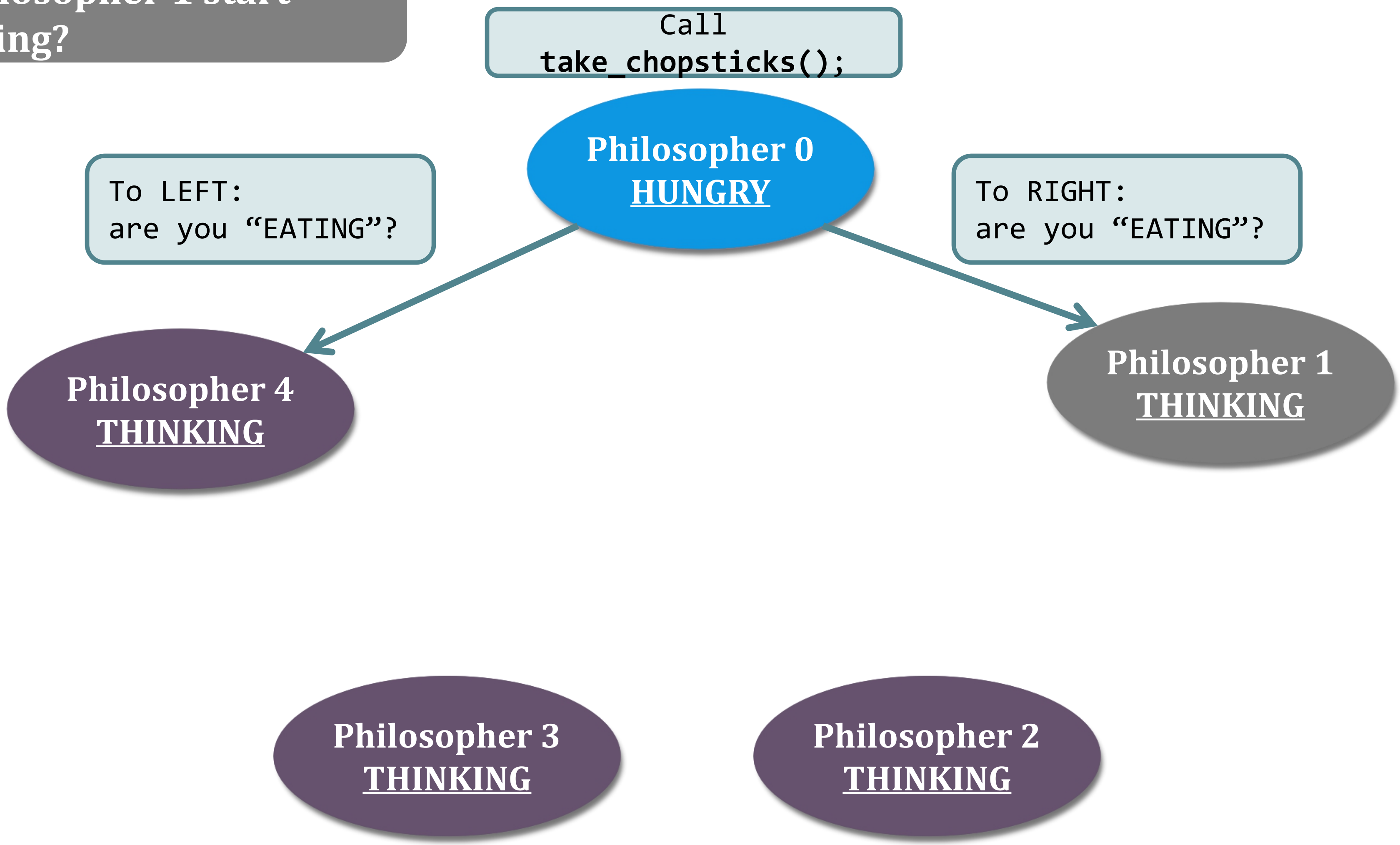
Philosopher 3
THINKING

Philosopher 2
THINKING

Dining philosopher – the final soluti

Don't print

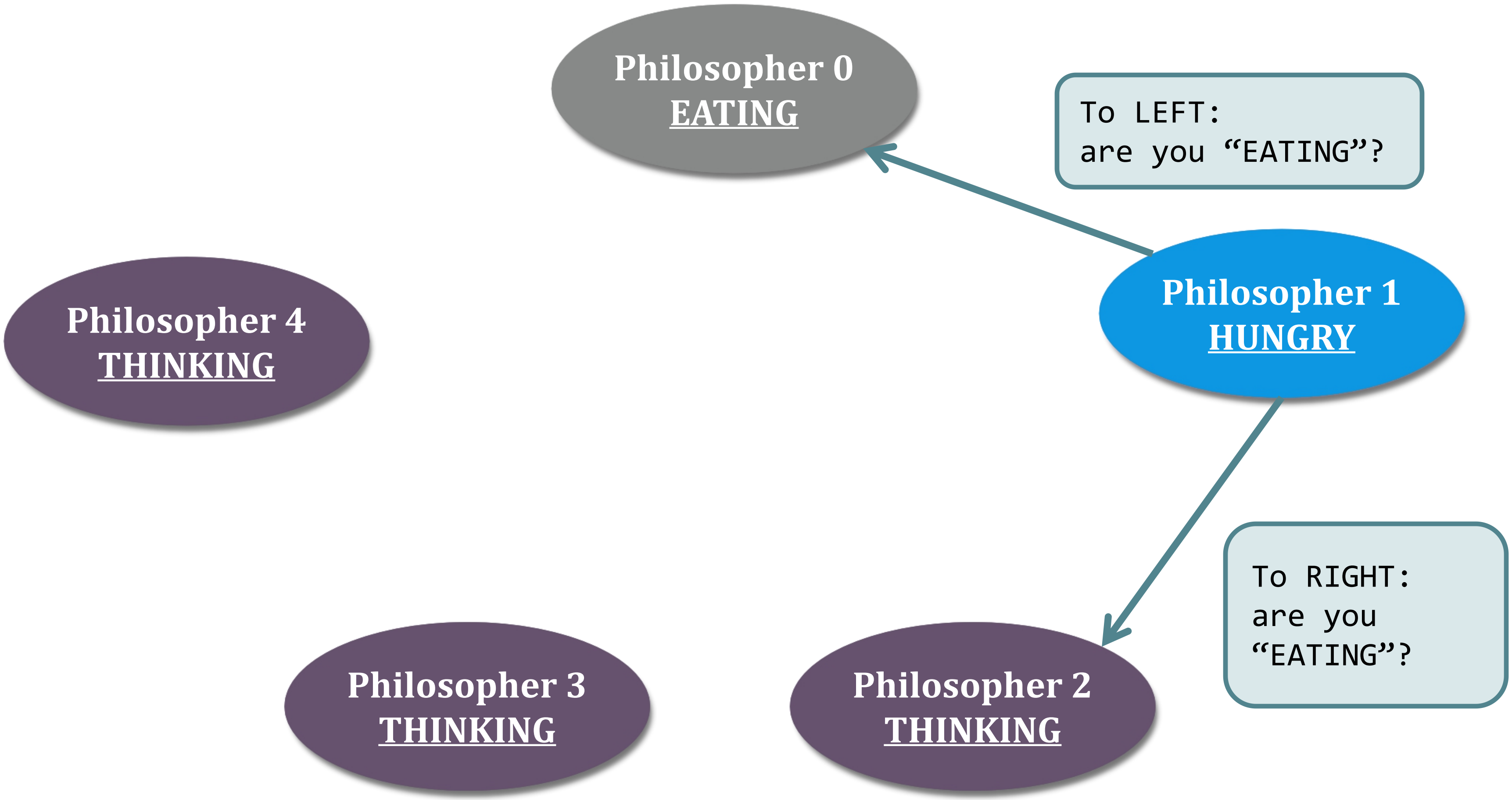
An illustration: How can Philosopher 1 start eating?



Dining philosopher – the final soluti

Don't print

An illustration: How can Philosopher 1 start eating?



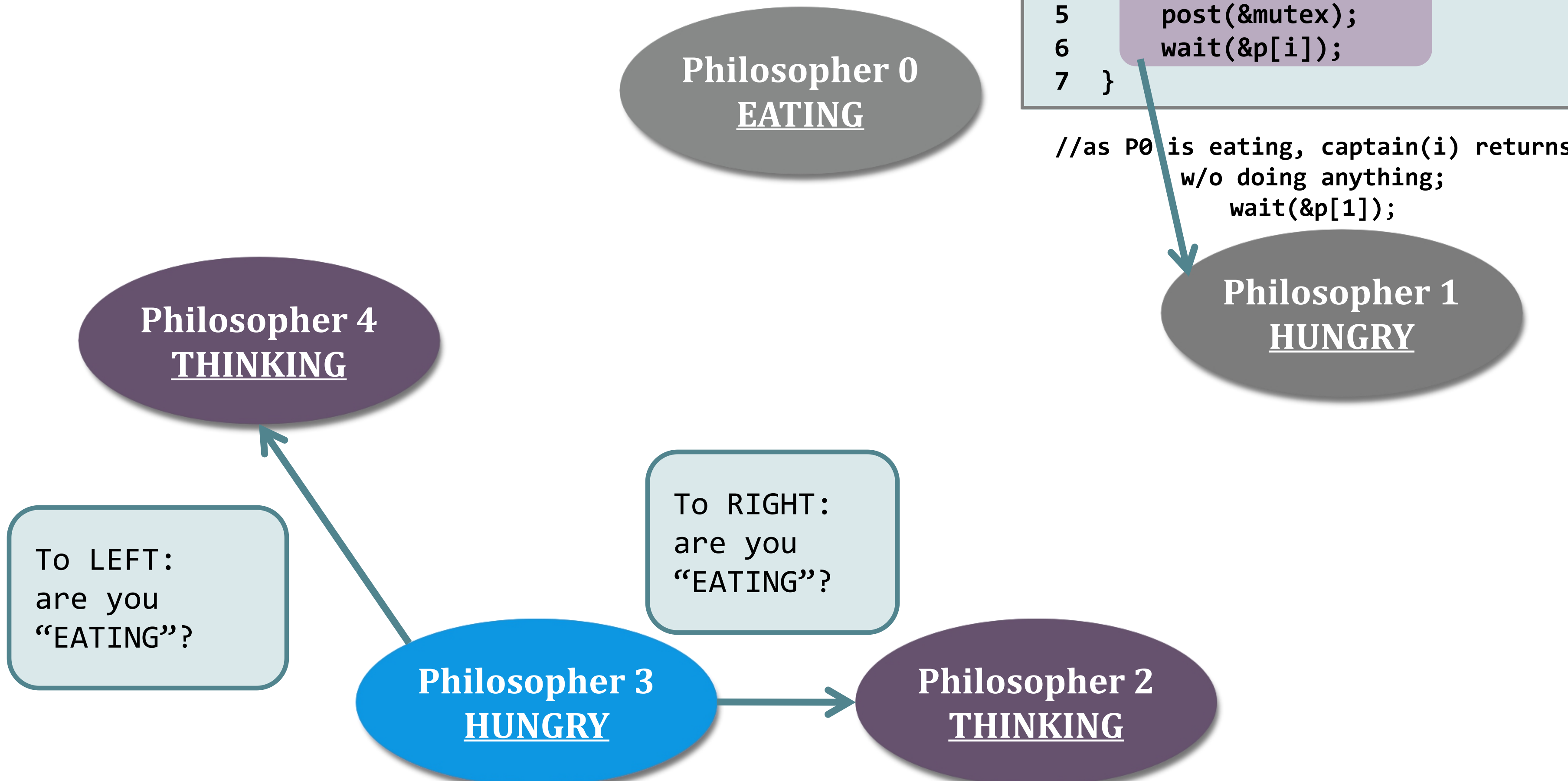
Dining philosopher – the final solution

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]);
}
```



Dining philosopher – the final soluti

Don’t print

An illustration: How can Philosopher 1 start eating?

Philosopher 0
EATING

Philosopher 4
THINKING

Philosopher 1
HUNGRY

```
Blocked;
because of
wait(&p[1]);
```

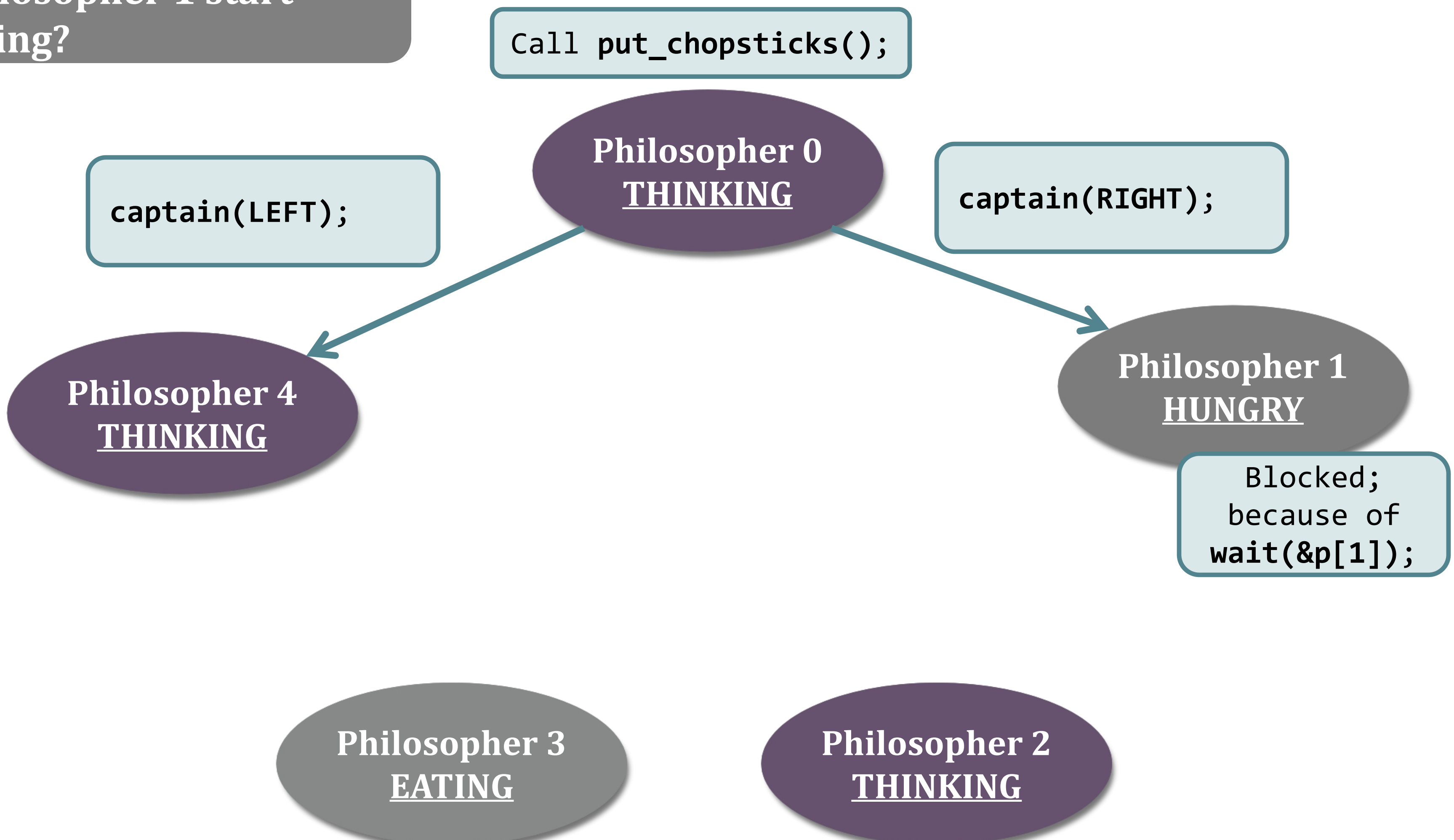
Philosopher 3
EATING

Philosopher 2
THINKING

Dining philosopher – the final soluti

Don't print

An illustration: How can Philosopher 1 start eating?



Dining philosopher – the final soluti

Don't print

An illustration: How can Philosopher 1 start eating?

Call **put_chopsticks();**

Philosopher 0
<u>THINKING</u>

captain(RIGHT);

Blocked; because of **wait(&p[1]);**

Philosopher 4
<u>THINKING</u>

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

Wake up !

Dining philosopher – the 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]);
}
```

Wake up

Philosopher 0
THINKING

Philosopher 1
EATING

Philosopher 4
THINKING

Philosopher 3
EATING

Philosopher 2
THINKING

Dining philosopher – the core

5 philosophers → ideally how many chopsticks?

how many chopsticks do we have now?

Very common in today's cloud computing multi-tenancy model

Summary on IPC problems

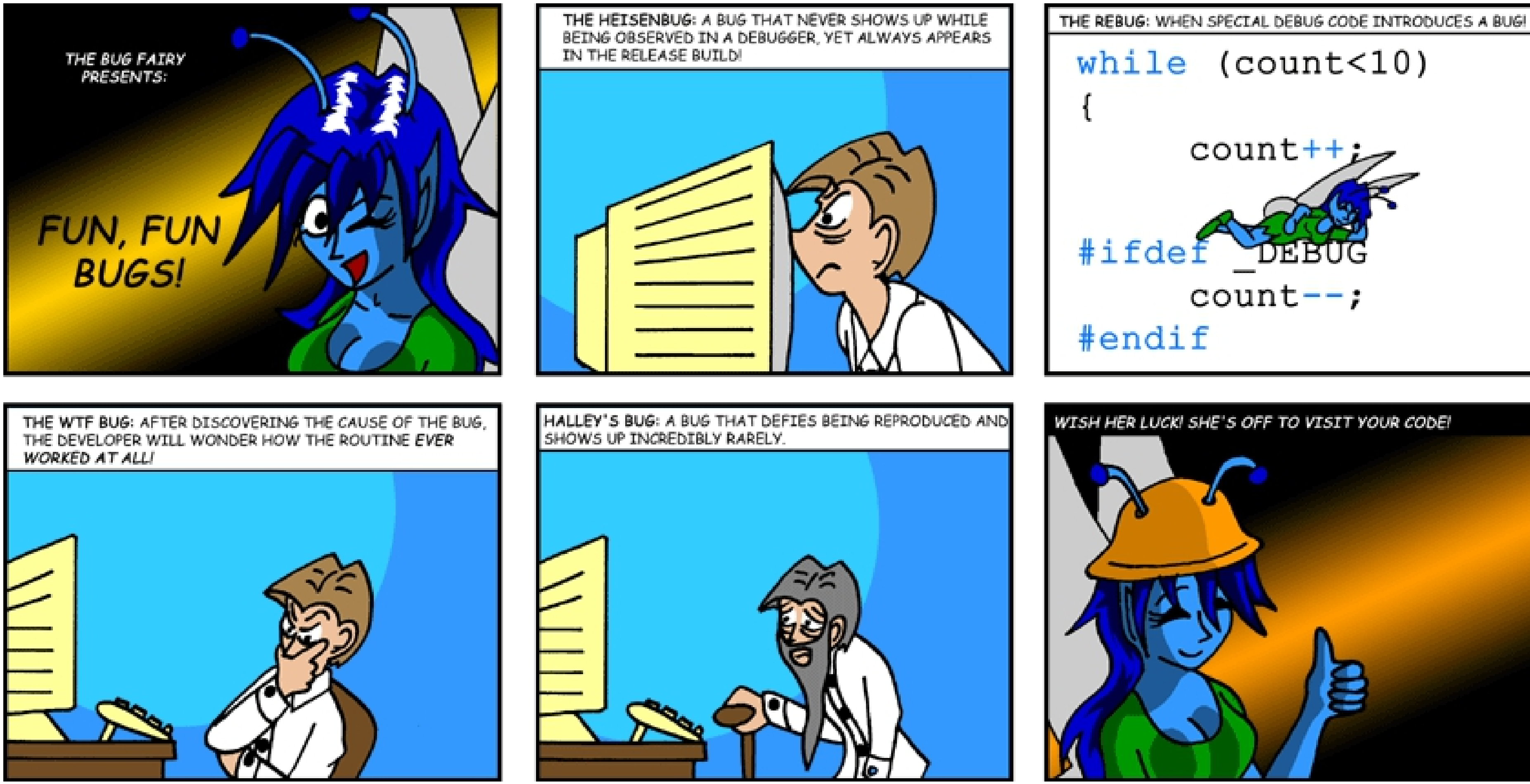
- The problems have the following properties in common:
  - Multiple number of processes;
  - Processes have to be synchronized in order to generate useful output;
  - Each resource may be shared as well as limited, and there may be more than one shared processes.

- The synchronization algorithms have the following requirements in common:
  - Guarantee mutual exclusion;
  - Uphold the correct synchronization among processes; and
  - (must be) Deadlock-free.

Heisenbugs

- Jim Gray, 1998 ACM Turing Award winner, coined that term
- You find your program P has a concurrency bug
- You insert 'printf' statements or GDB to debug P
- Then because of those debugging things added, P behaves normally when you are in debug mode

Heisenbugs


THE BUG FAIRY PRESENTS:
FUN, FUN BUGS!
THE HEISENBUG: A BUG THAT NEVER SHOWS UP WHILE BEING OBSERVED IN A DEBUGGER, YET ALWAYS APPEARS IN THE RELEASE BUILD!
THE REBUG: WHEN SPECIAL DEBUG CODE INTRODUCES A BUG!
while (count<10)
{
count++;
#ifdef _DEBUG
count--;
#endif
THE WTF BUG: AFTER DISCOVERING THE CAUSE OF THE BUG, THE DEVELOPER WILL WONDER HOW THE ROUTINE EVER WORKED AT ALL!
HALLEY'S BUG: A BUG THAT DEFIES BEING REPRODUCED AND SHOWS UP INCREDIBLY RARELY.
WISH HER LUCK! SHE'S OFF TO VISIT YOUR CODE!

Thank You!

Dining philosopher – the 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) {
   disable_interrupt();
    *s = *s - 1;
   if ( *s < 0 ) {
      enable_interrupt();
      sleep();
      disable_interrupt();
    }
   enable_interrupt();
}
```

Section entry

```
void take_chopsticks(int i) {
    wait(&mutex);
    state[i] =____(2)______;
    captain(i);
    post(&mutex);
    wait(___(3)___);
}
```

Section exit

```
void put_chopsticks(int i) {
    wait(&mutex);
    state[i] = ___(4)______;
    captain(LEFT);
    captain(RIGHT);
    post(&mutex);
}
```

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

Extremely important helper function

```
void captain(int i) {
    if(_______(5)_______) {
        state[i] = EATING;
        post(&p[i]);
    }
}
```