Lecture 5
Synchronization

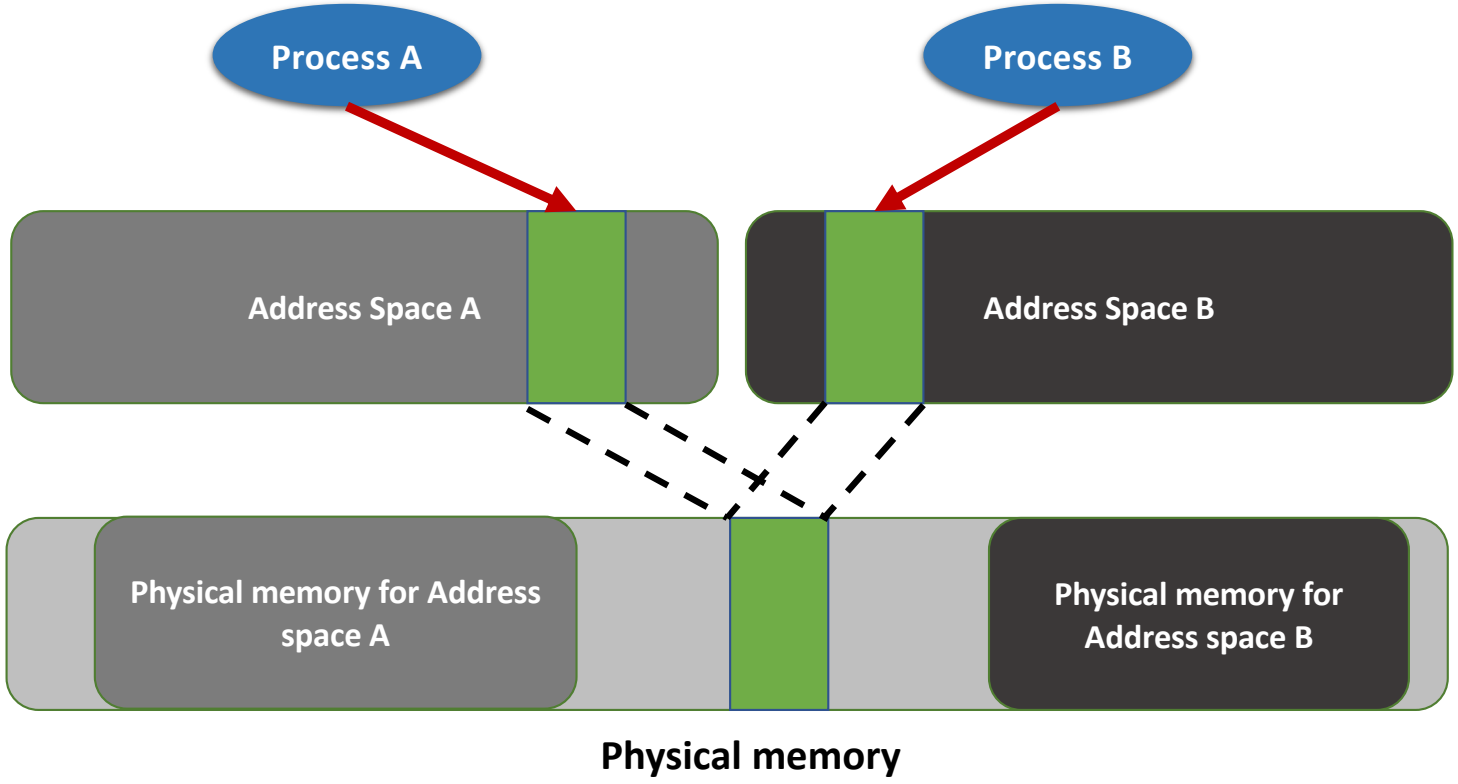
Prof. Yinqian Zhang

Fall 2025

Process Communication

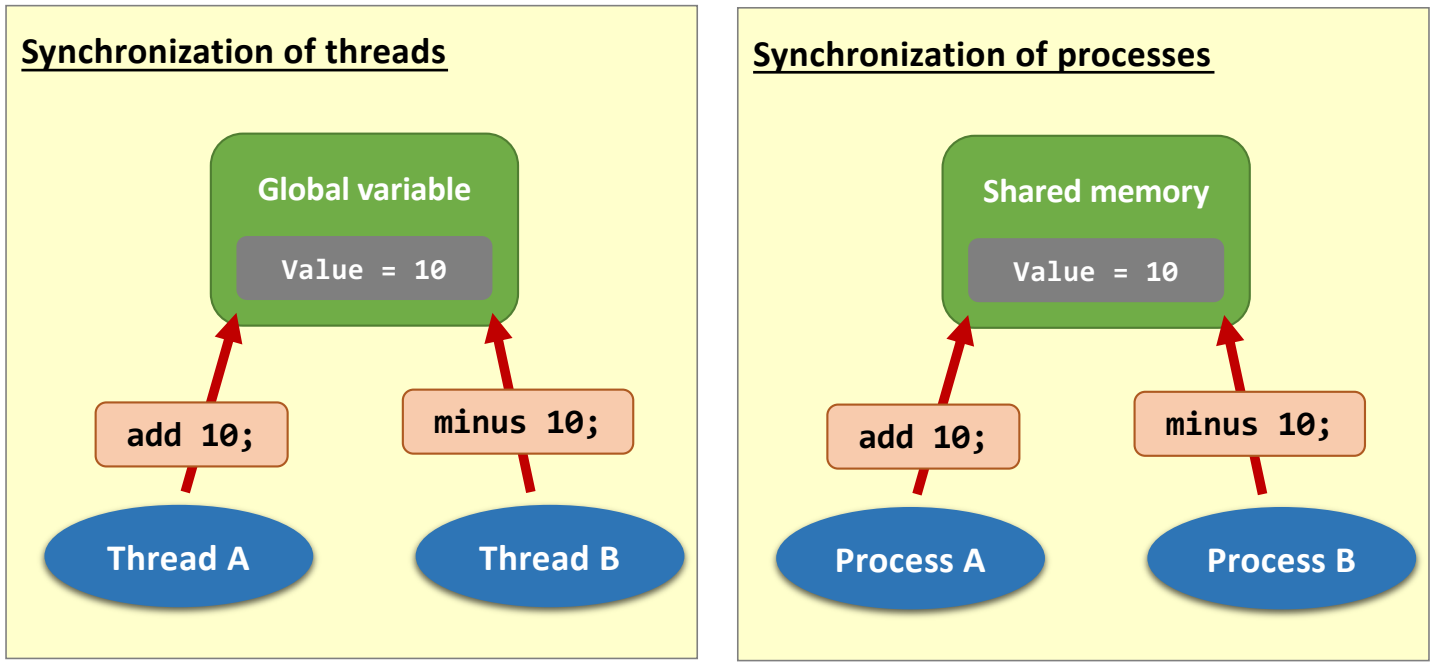
- Threads of the same process share the same address space
  - Global variables are shared by multiple threads
  - Communication between threads made easy
- Process may also need to communicate with each other
  - Information sharing:
    - e.g., sharing between Android apps
  - Computation speedup:
    - 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

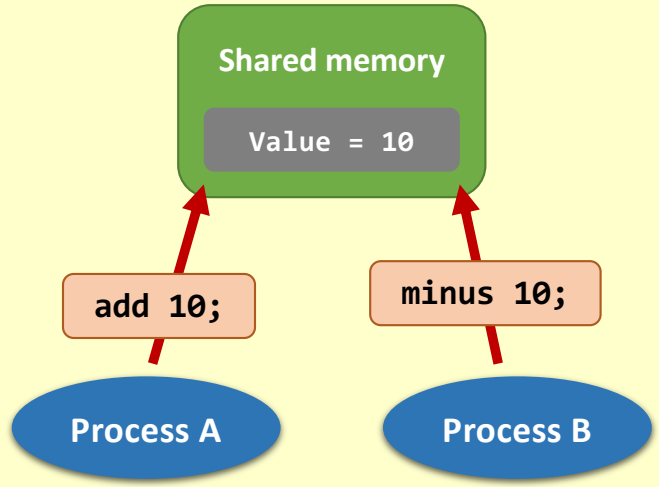
High-level language for Program A

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

Partial low-level language for Program A

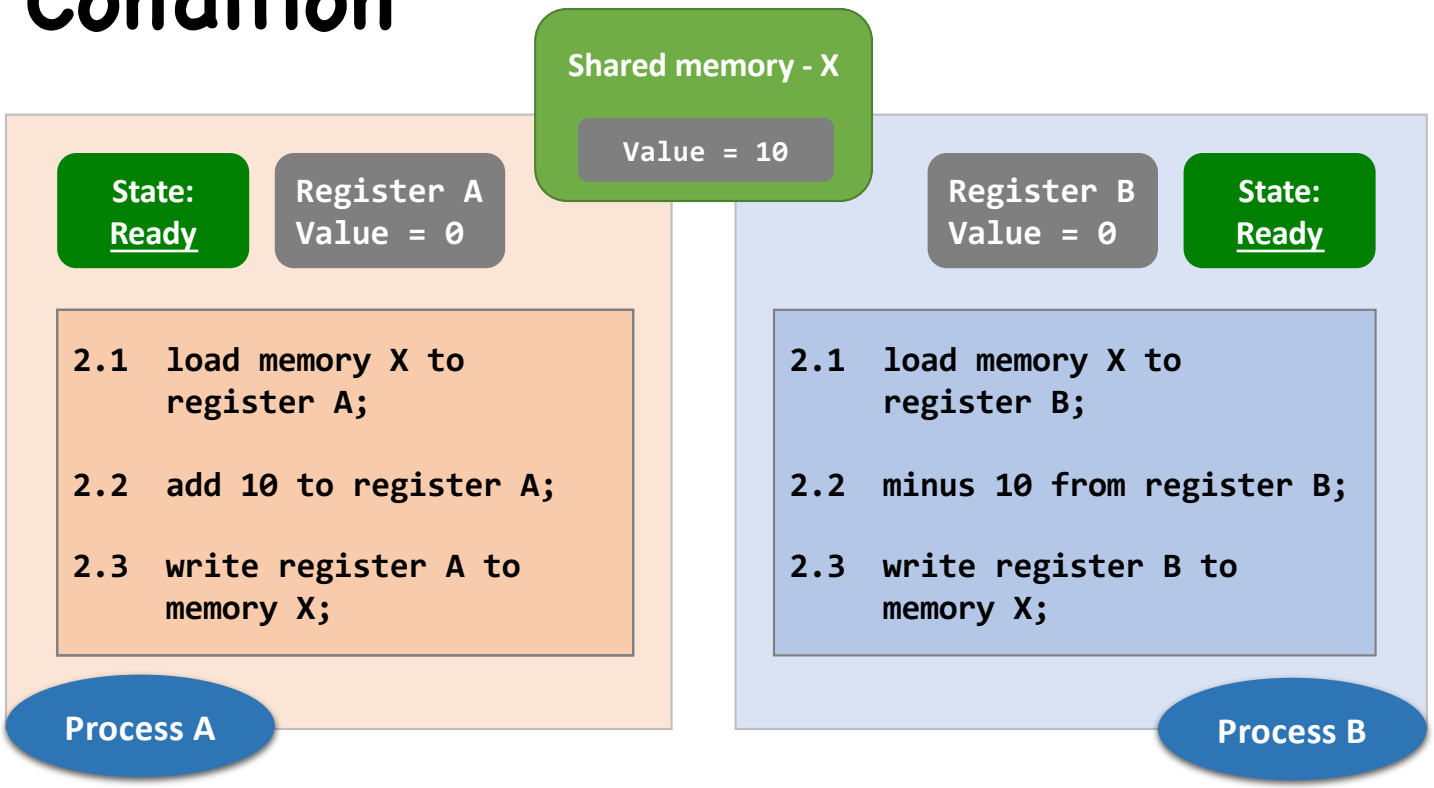
```
1    attach to the shared memory X;
.....
2.1  load memory X to register A;
2.2  add 10 to register A;
2.3  write register A to memory X;
.....
3    exit;
```

The Scenario



Race Condition

Don't print



The initial setting

竞态条件是指当两个或多个进程在没有适当同步的情况下访问共享资源时，可能发生的不确定行为。

这里展示了进程A和进程B同时访问共享内存的情况，导致数据竞争。

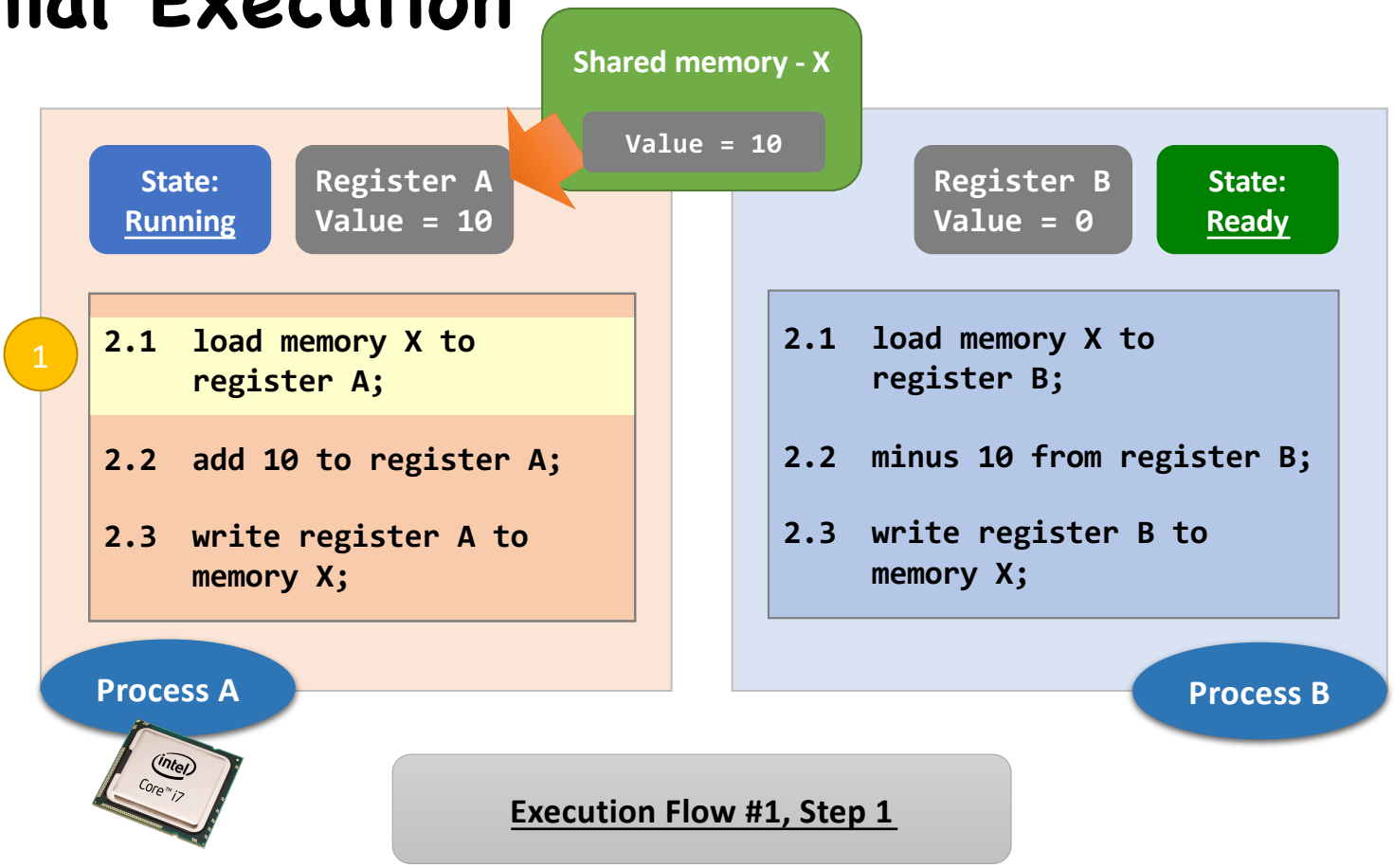
进程A读取共享内存X，将10加到寄存器A，再将A的值写回共享内存X。

进程B读取共享内存X，从寄存器B中减去10，再将B的值写回共享内存X。

在这种情况下，如果没有同步机制，进程A和进程B可能会同时修改共享内存，导致错误的最终结果

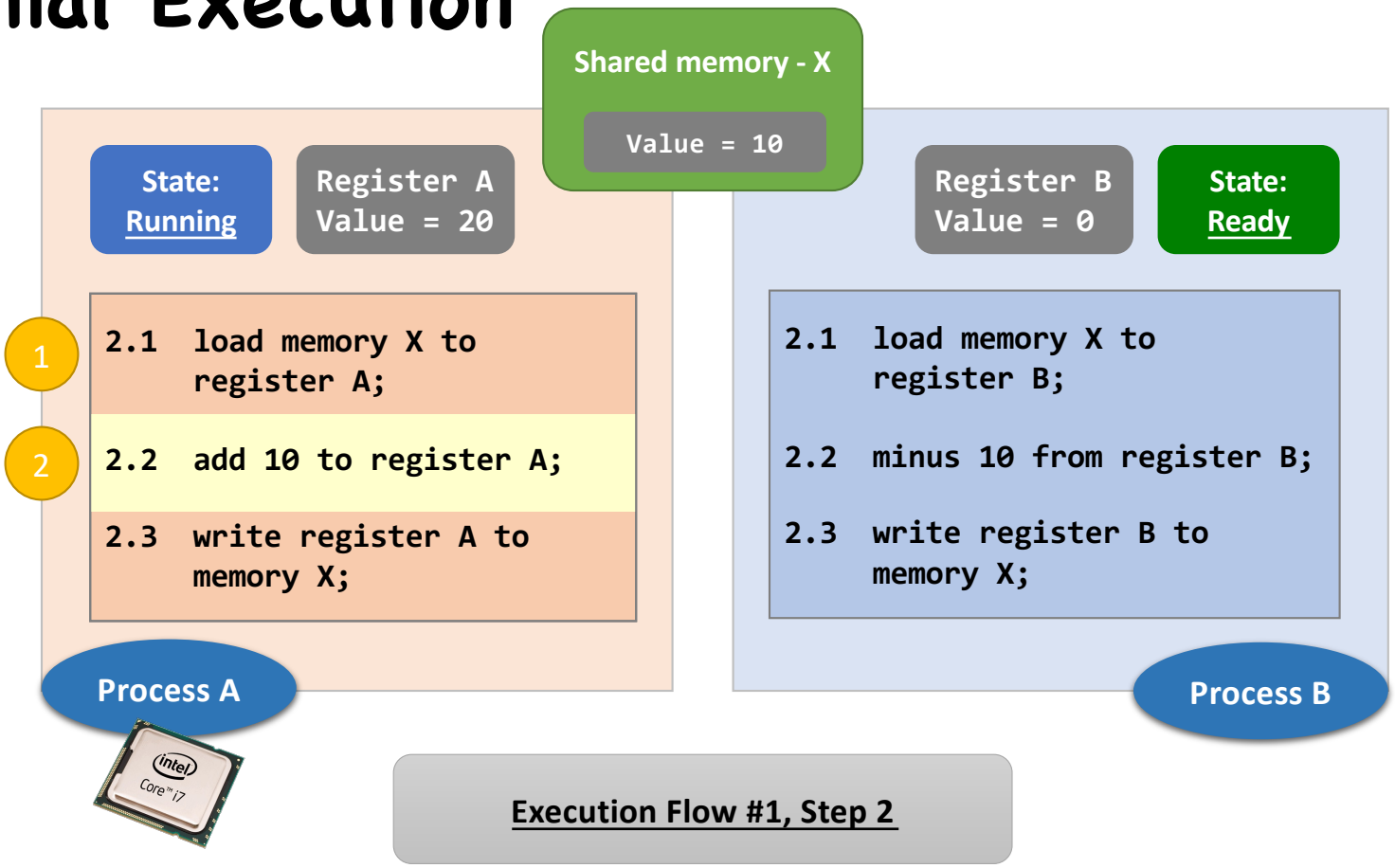
Don't print

Normal Execution



Don't print

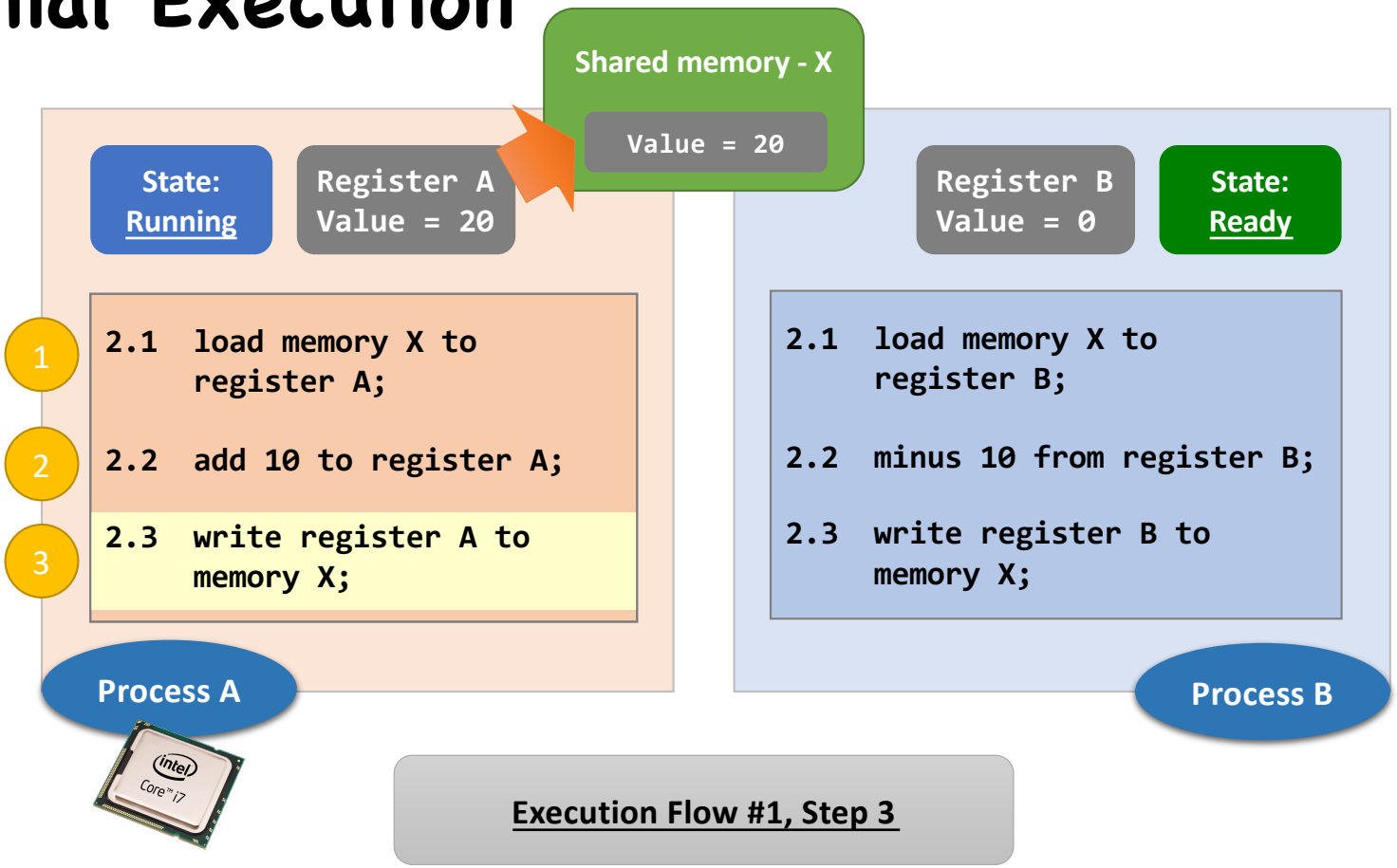
Normal Execution



Execution Flow #1, Step 2

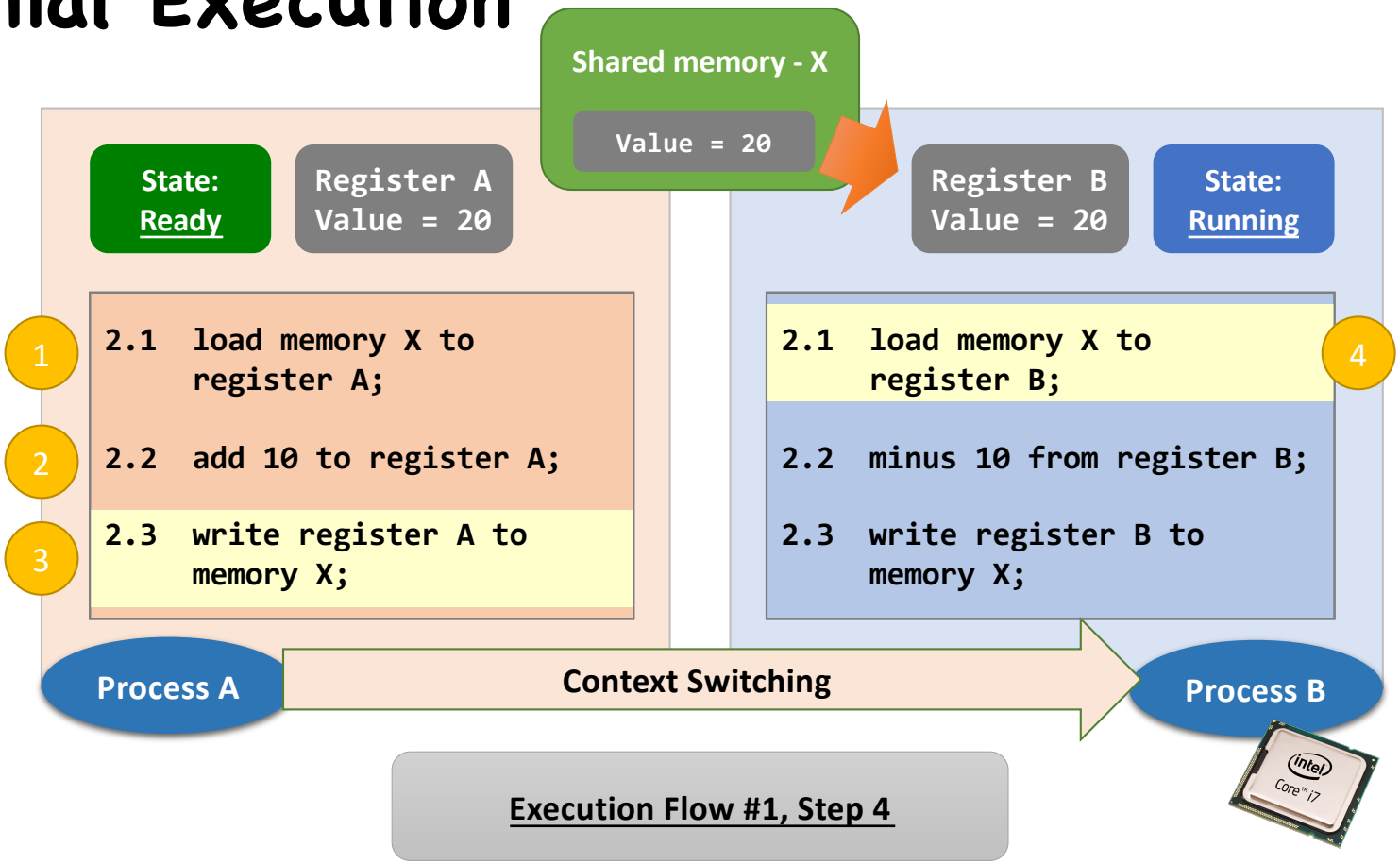
Don't print

Normal Execution



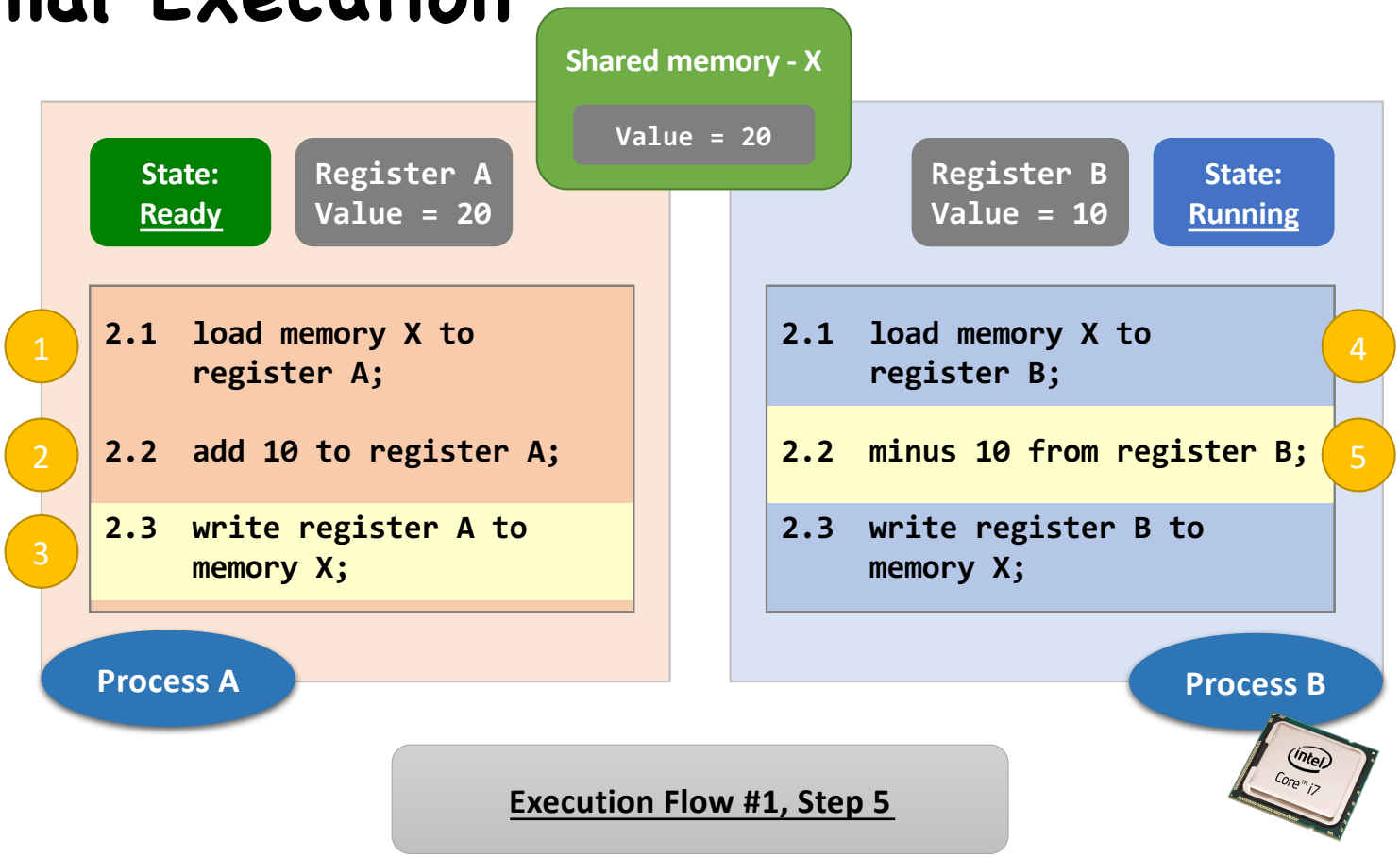
Don't print

Normal Execution

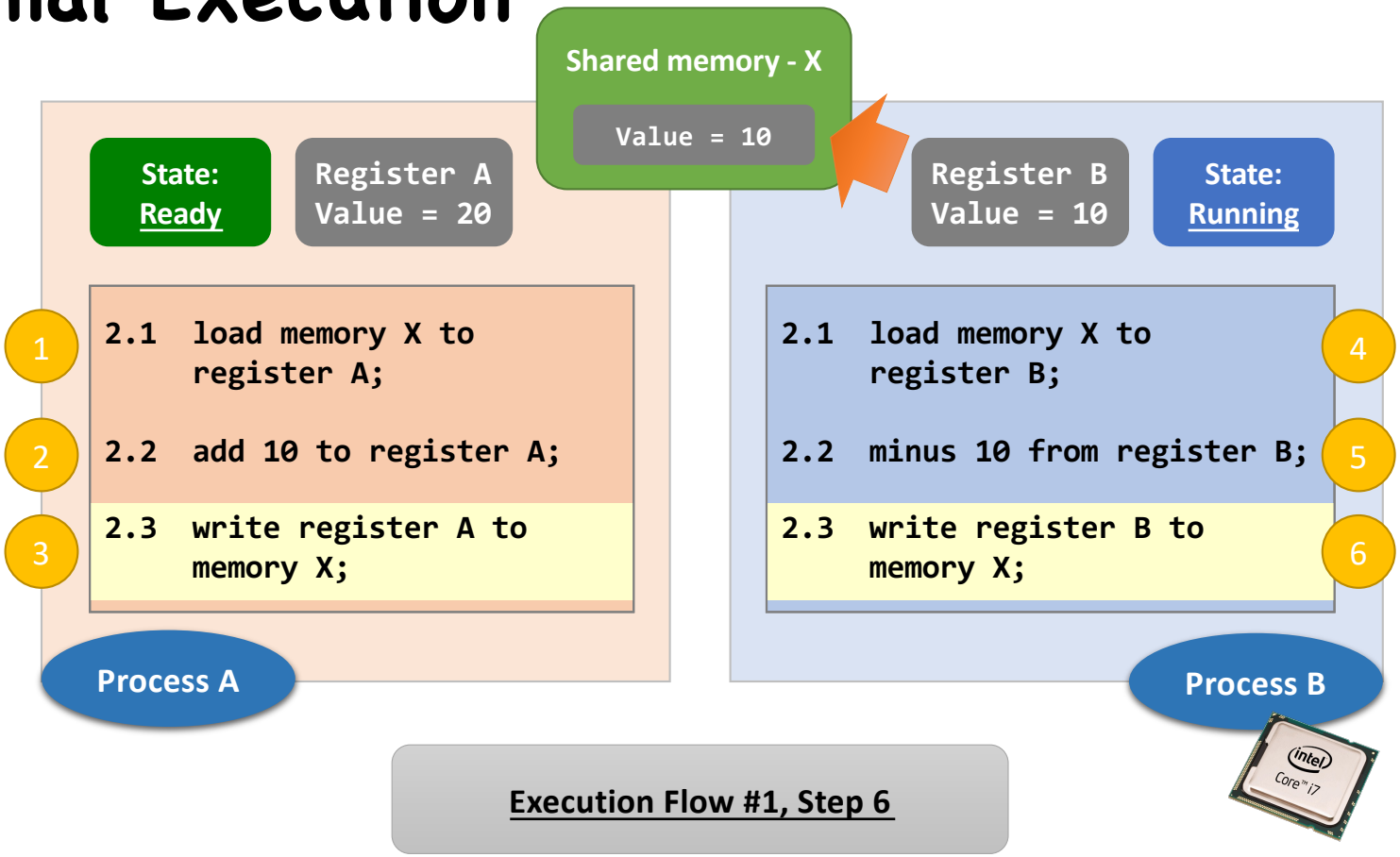


Don't print

Normal Execution

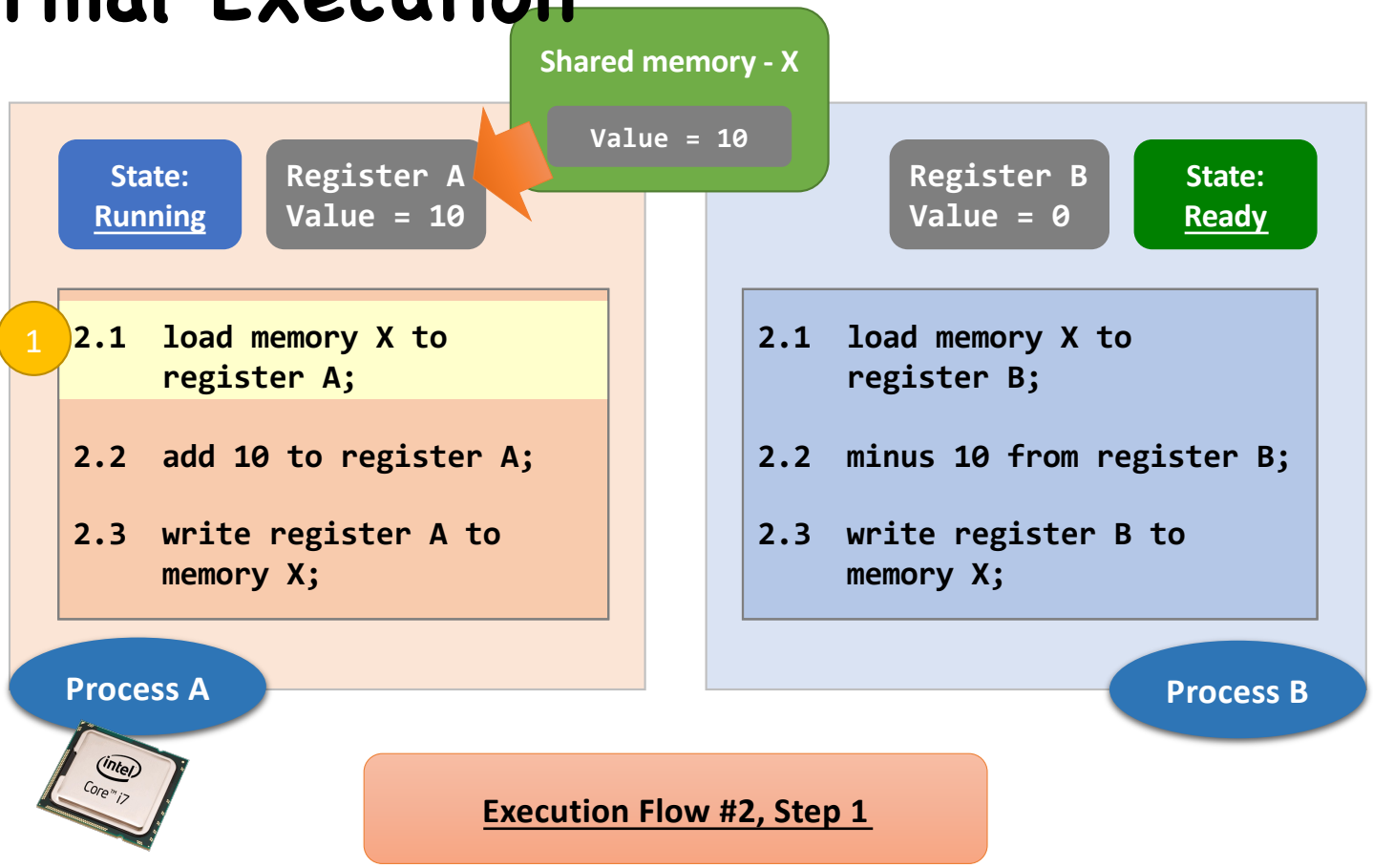


Normal Execution



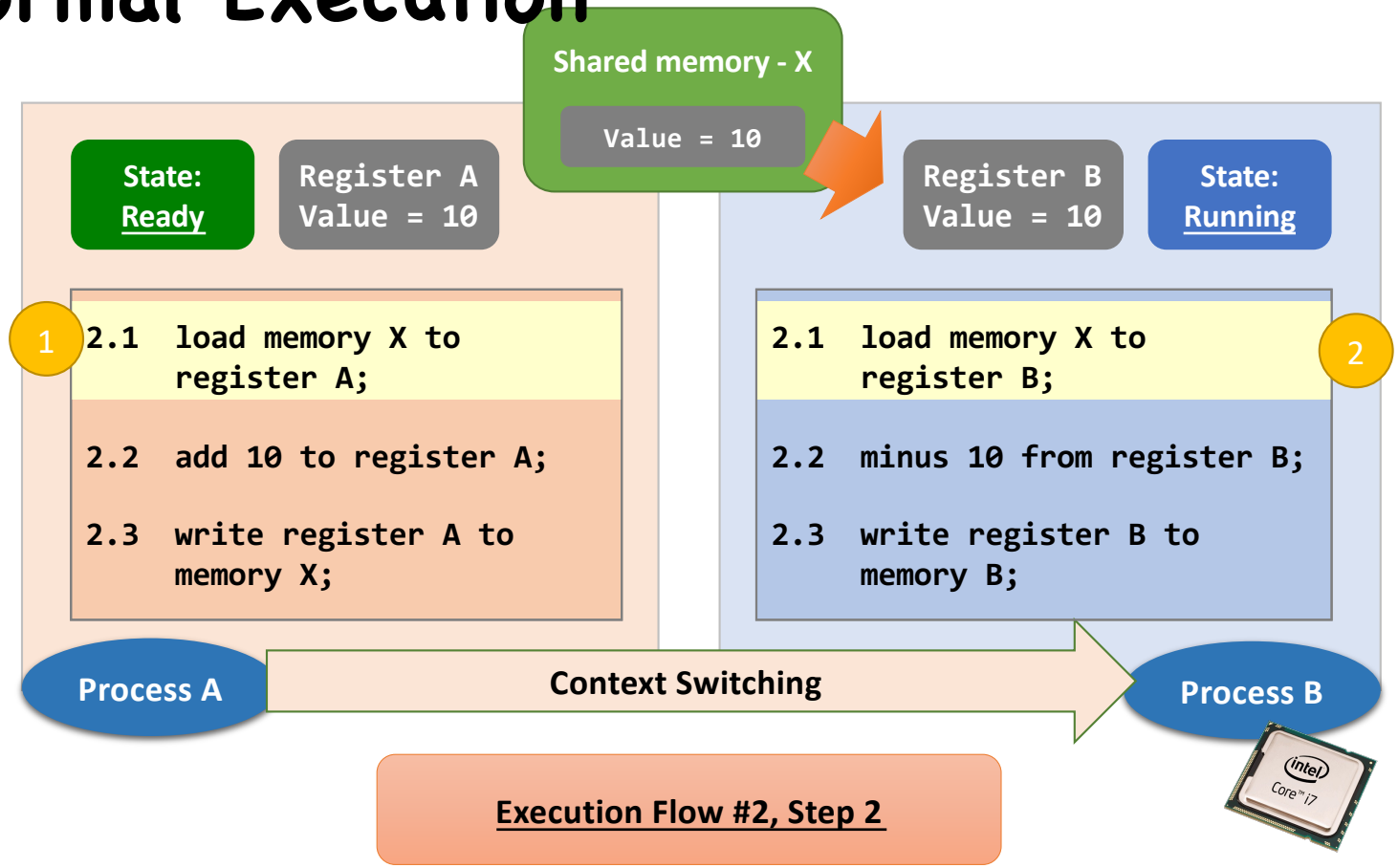
Don't print

Abnormal Execution



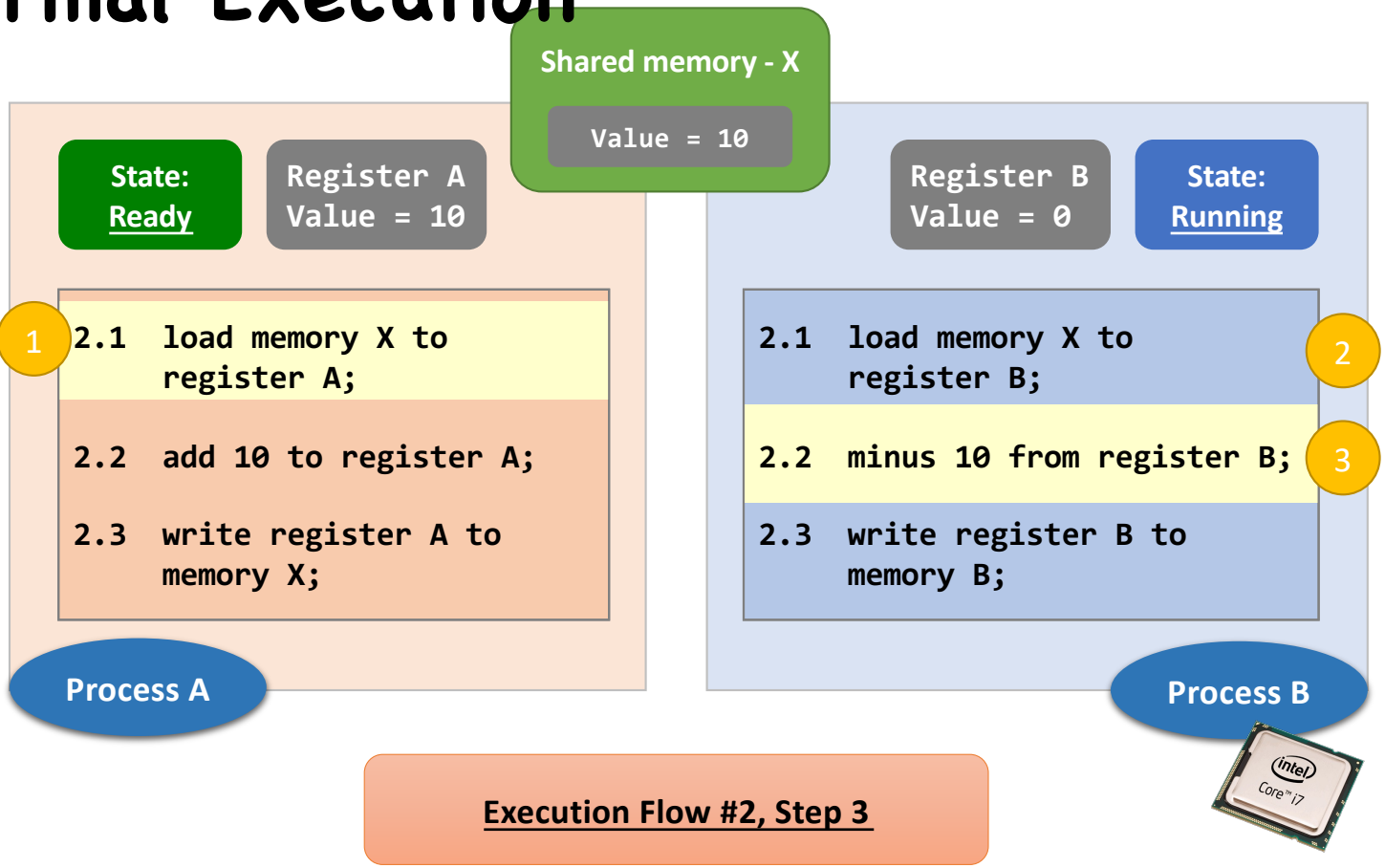
Don't print

Abnormal Execution



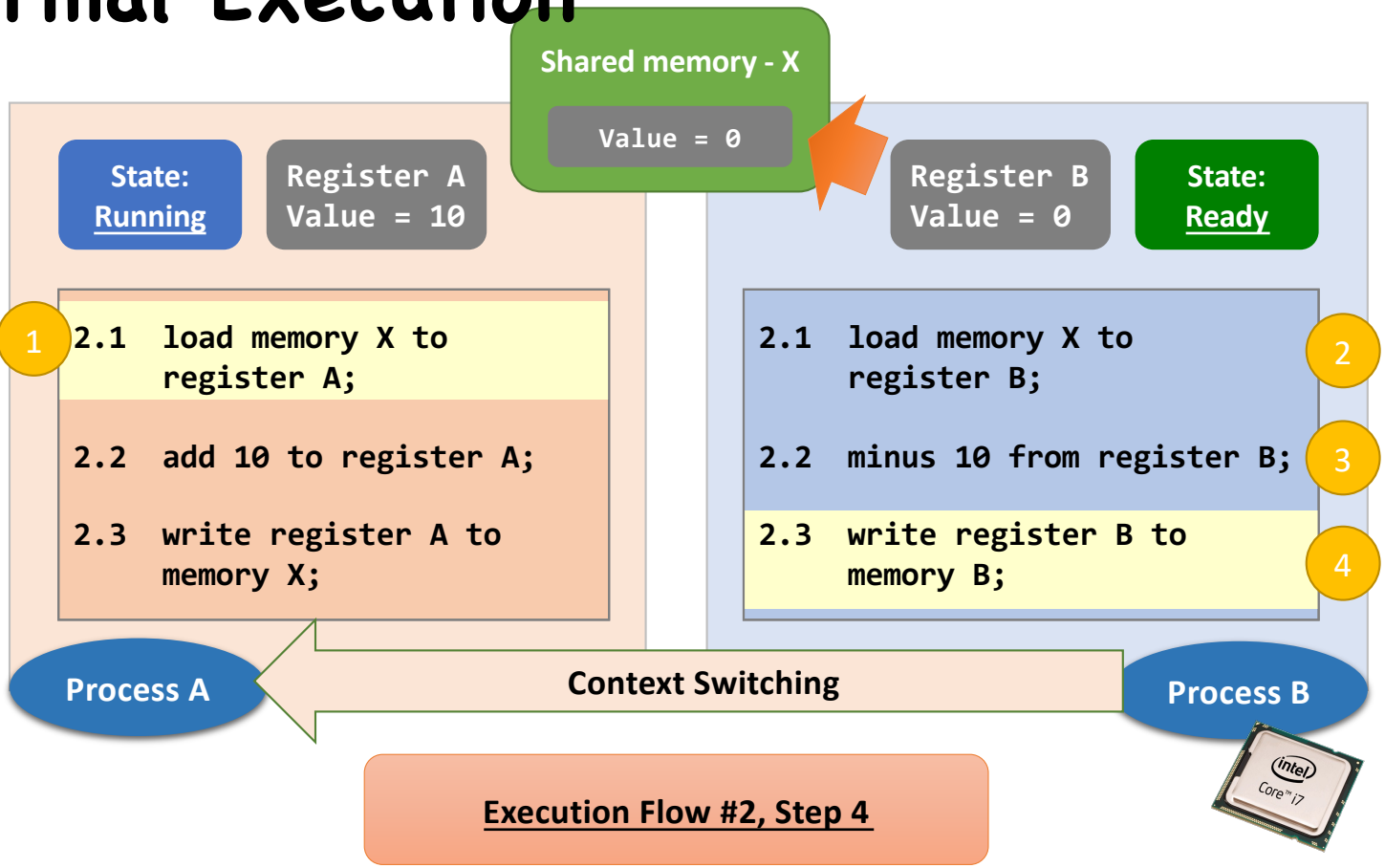
Abnormal Execution

Don't print



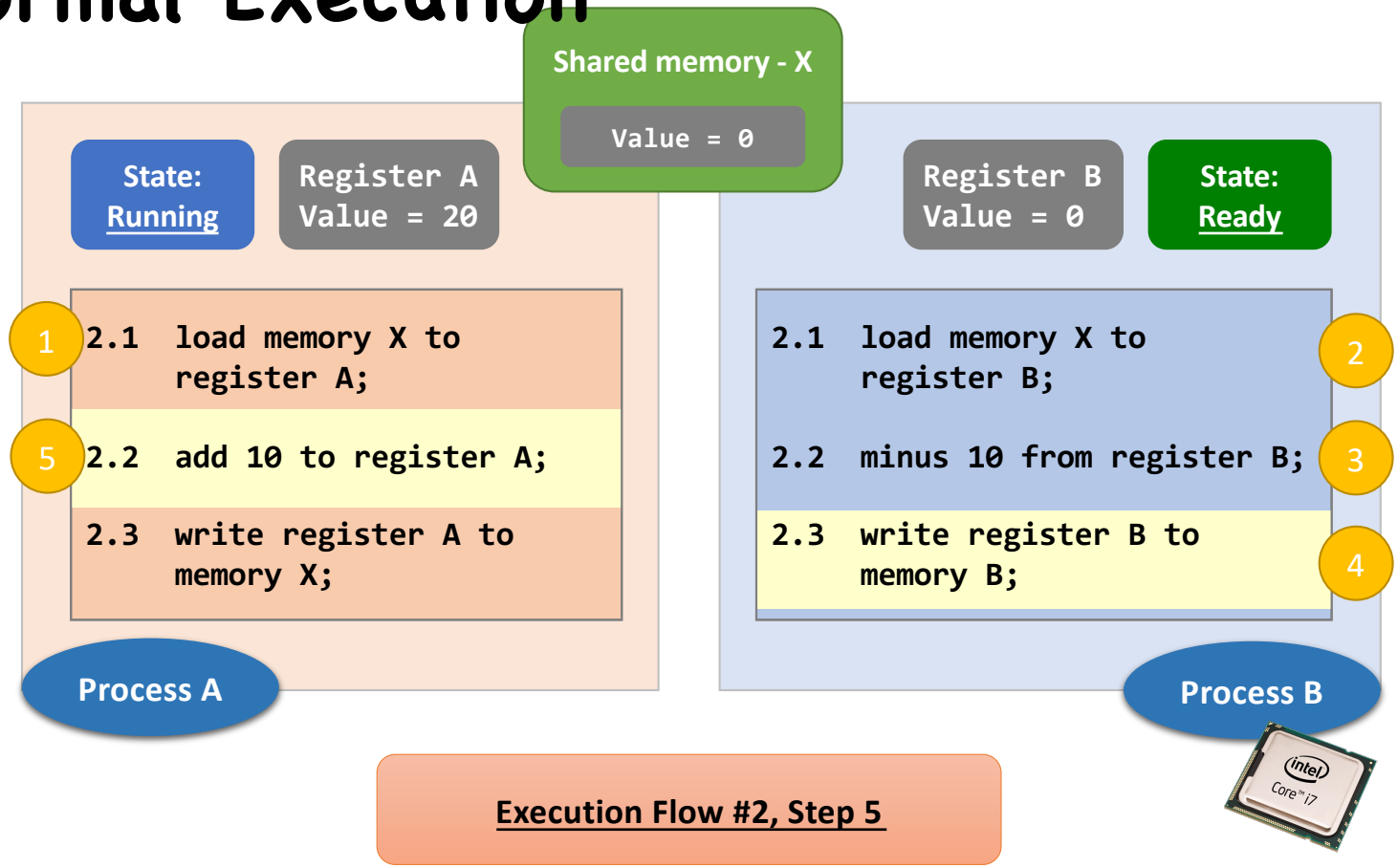
Don't print

Abnormal Execution



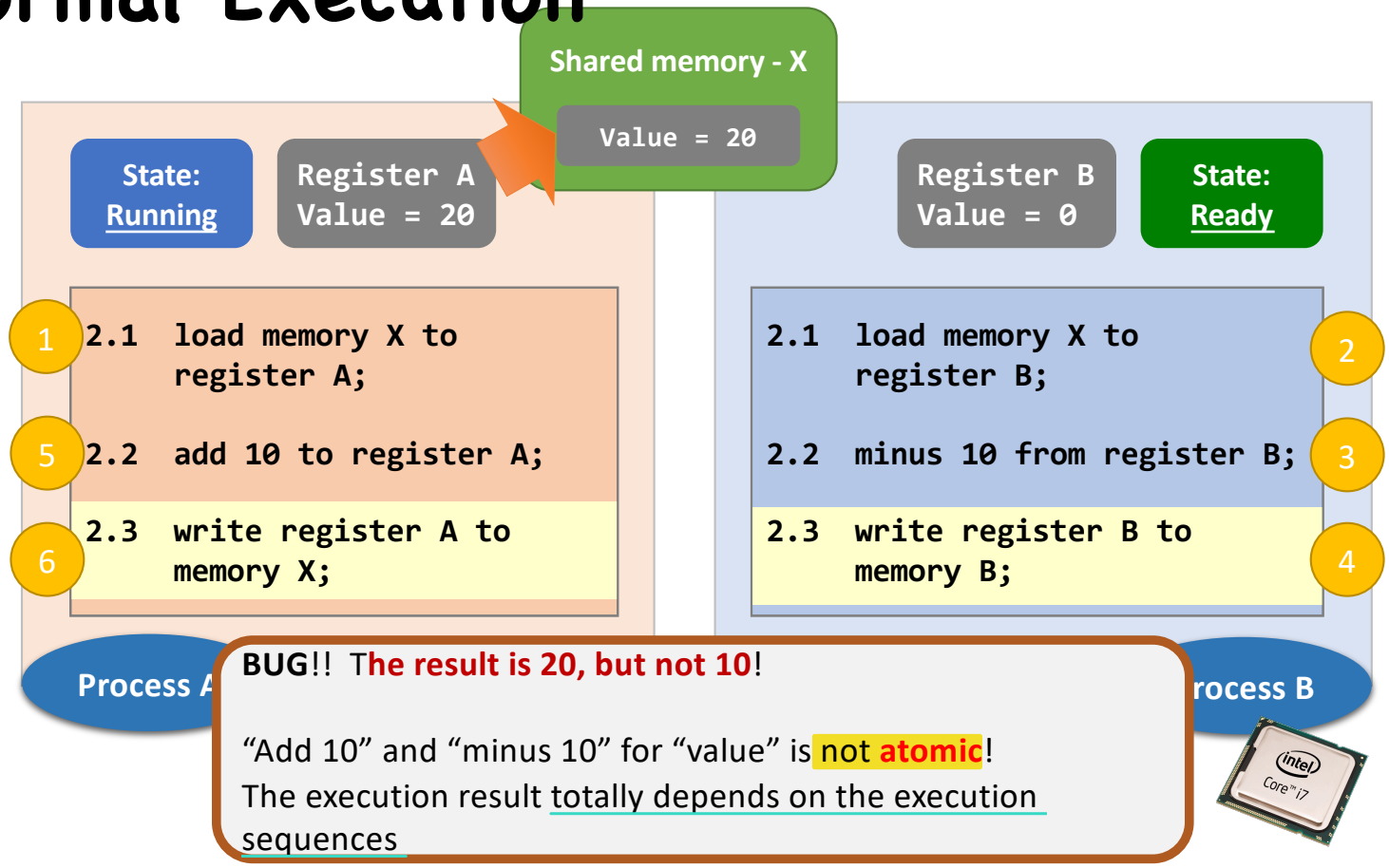
Don't print

Abnormal Execution



Don't print

Abnormal Execution



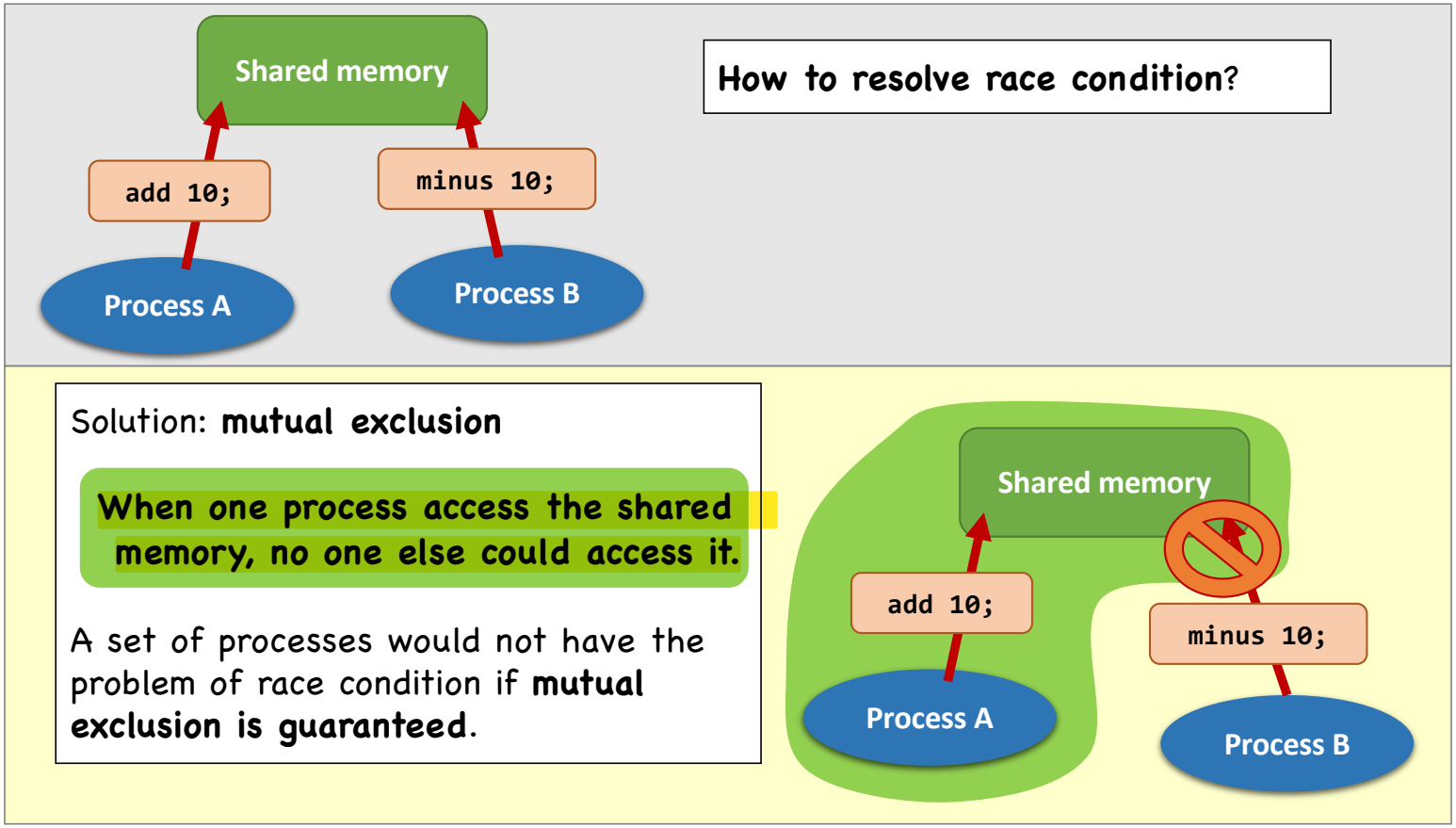
Race Condition

- 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.

竞态条件是指在程序执行时，多个进程或线程访问共享资源并且执行顺序不确定，导致最终结果取决于执行的顺序。

- 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

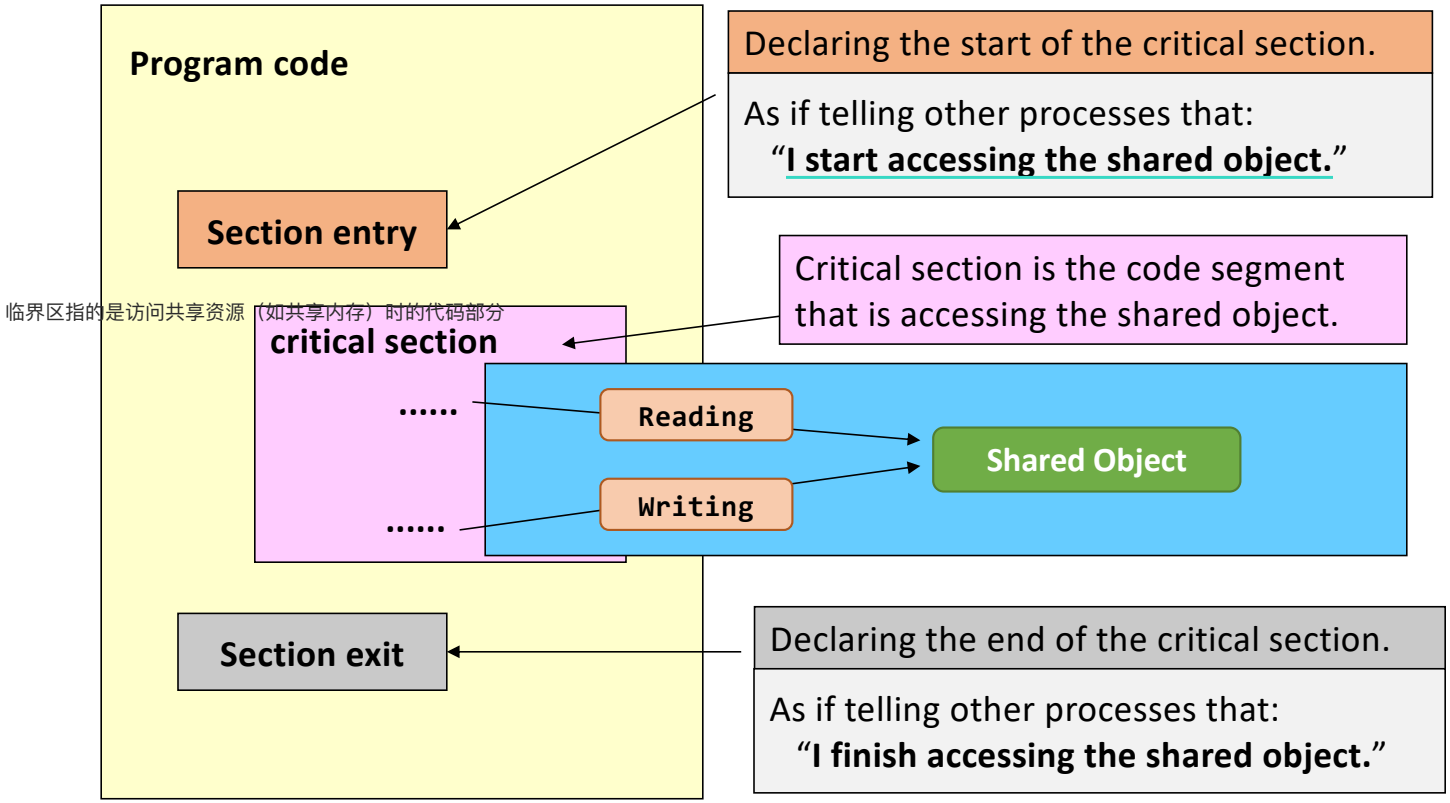


Solution: Mutual Exclusion

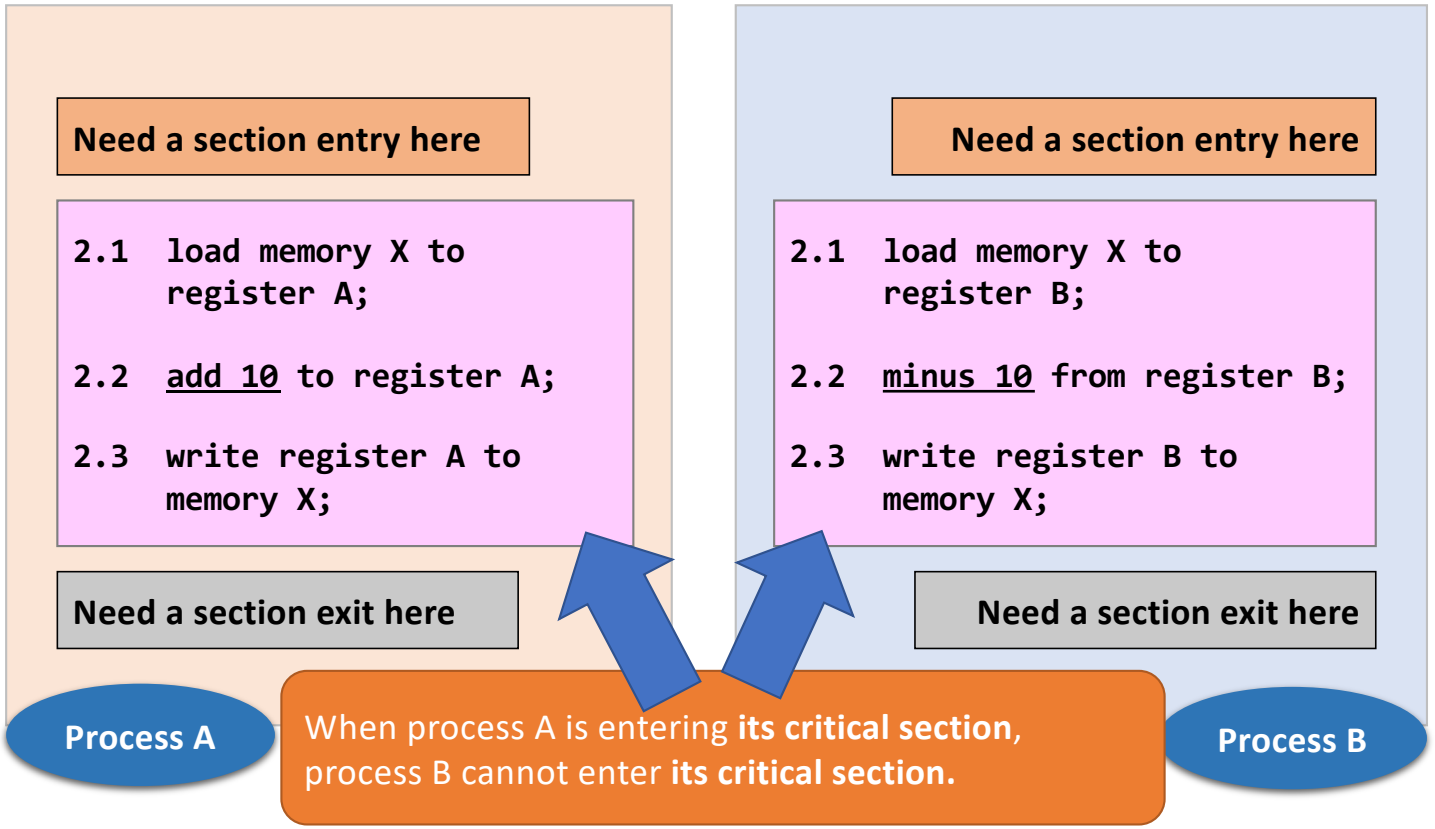
- Shared object is still sharable, but
- Do not access the "shared object" **at the same time**
- Access the "shared object" one by one

如何实现**临界区（Critical Section）**来保证互斥，从而解决竞态条件的问题。

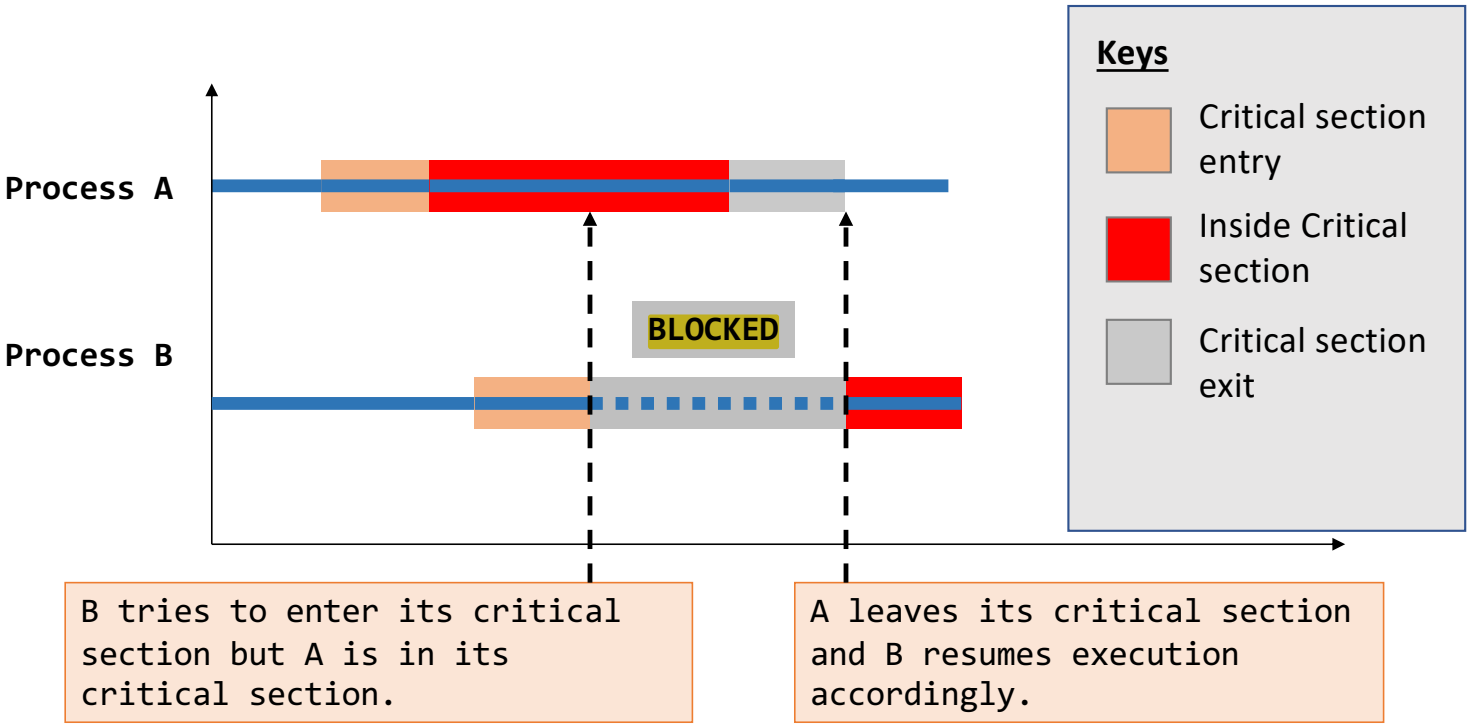
Critical Section: Realizing Mutual Exclusion



Critical Section: Realizing Mutual Exclusion



A Typical Mutual Exclusion Scenario



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. 临界区应该尽量缩小，避免阻塞其他进程
    - 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

一旦一个进程开始尝试进入其临界区，它需要在一定时间内能够进入。即限制其他进程在进入临界区时的等待次数。

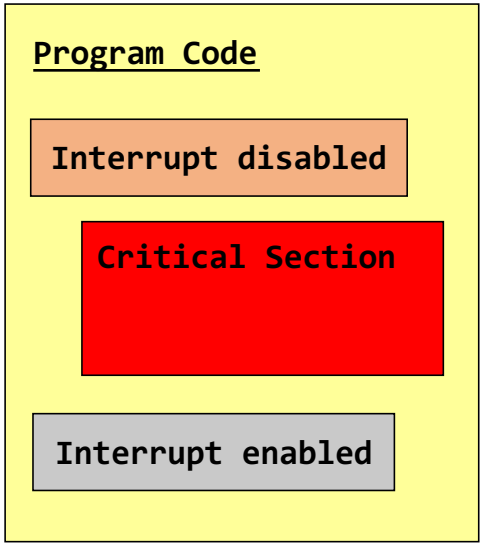
保证没有进程一直停留在临界区内

如果有进程正在尝试进入临界区，它最终会进入

Solution: Disabling Interrupts

在进程进入临界区时，禁用中断，以防止其他进程或线程的中断操作

- 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!!!!)



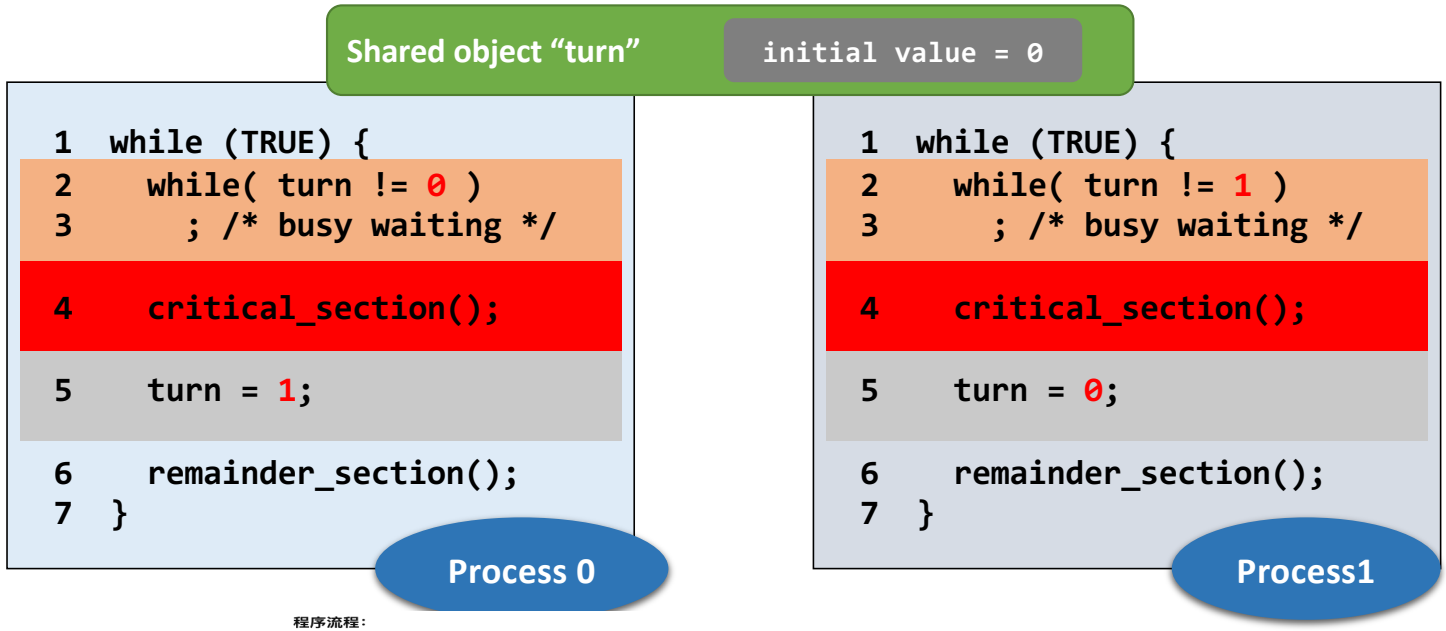
在多个核心上运行时，除非禁用所有核心的中断，否则一个核心修改共享内存时，其他核心可能会干扰。

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 使用一个共享变量进行自旋等待。比如，通过不断检查一个变量的状态来判断是否可以进入临界区。
    - Basic spinning using 1 shared variable
    - Peterson's solution: Spin using 2 shared variables
  - Thread synchronization: pthread_spin_lock 通过pthread_spin_lock来实现线程级别的自旋锁
- Sleep-based locks
  - Process synchronization: POSIX semaphore 使用POSIX信号量进行同
  - Thread synchronization: pthread_mutex_lock

Spin-based Locks

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



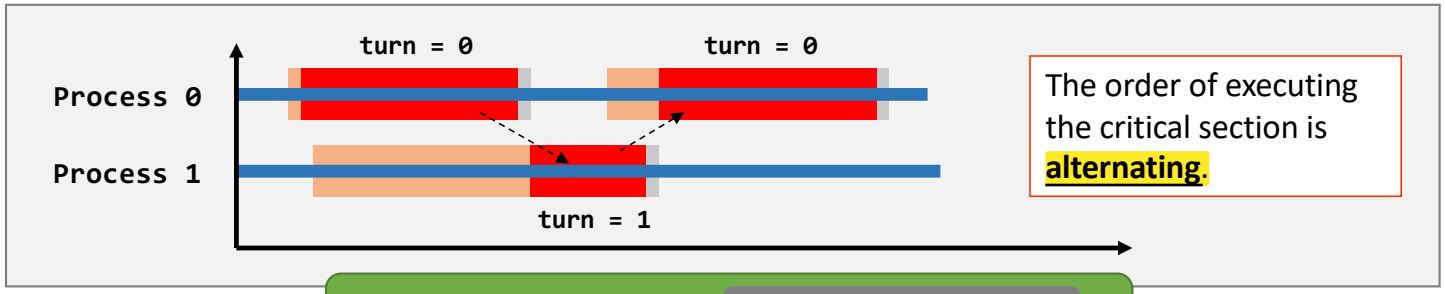
程序流程：

- 进程0（左图）：
  1. 进程0进入一个无限循环，检查 `turn` 的值是否为0。如果是，进程0继续等待（即"忙等待"），直到 `turn` 的值变为1。
  2. 一旦 `turn` 为0，进程0进入临界区执行 `critical_section()`。
  3. 执行完临界区后，进程0将 `turn` 设置为1，通知进程1可以进入其临界区。
- 进程1（右图）：
  1. 进程1也进入一个类似的无限循环，检查 `turn` 是否为1。如果是，进程1会继续等待。
  2. 当 `turn` 为1时，进程1进入临界区执行 `critical_section()`。
  3. 执行完后，进程1将 `turn` 设置为0，通知进程0可以继续执行其临界区代码。

关键点：

- **共享变量 turn**：`turn` 用于控制哪个进程可以进入临界区。进程0和进程1轮流访问临界区，确保互斥。
- **忙等待（busy waiting）**：进程通过持续检查 `turn` 的值来判断是否可以进入临界区，这是自旋锁的一大特点。

Spin-based Locks (Cont'd)



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

Shared object "turn" initial Value = 0

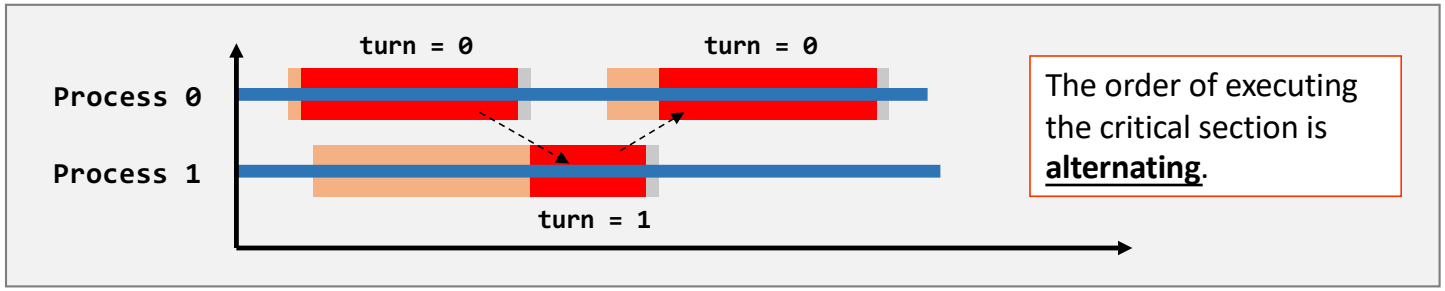
```
1  while (TRUE) {
2    while( turn != 0 )
3      ; /* busy waiting */
4    critical_section();
5    turn = 1;
6    remainder_section();
7  }
```

Process 0

```
1  while (TRUE) {
2    while( turn != 1 )
3      ; /* busy waiting */
4    critical_section();
5    turn = 0;
6    remainder_section();
7  }
```

Process1

Spin-based Locks (Cont'd)



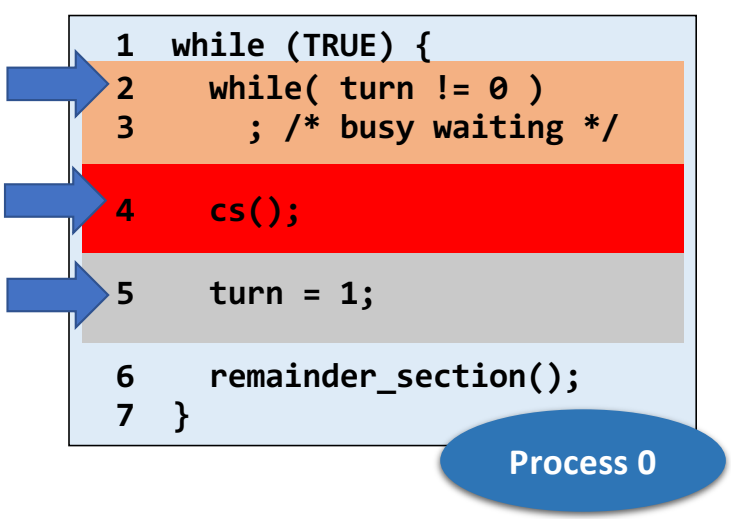
- Correct but waste CPU resources
  进程在等待时会一直占用CPU进行“忙等待”，这可能导致CPU空转，浪费计算资源。
  - OK for short waiting (spin-time < context-switch-overhead)
    就是空转的时间　　上下文切换的时间
    - Especially these days we have multi-core
      - Will not block other irrelevant processes a lot
- Impose a "strict alternating" order
  - Sometimes you give me my turn but I'm not ready to enter critical section yet
    即使一个进程被赋予了进入临界区的顺序（通过turn变量），如果它还没有准备好进入临界区，它也必须等待下一个进程完成其临界区操作。这种严格的交替顺序有时会导致不必要的等待。

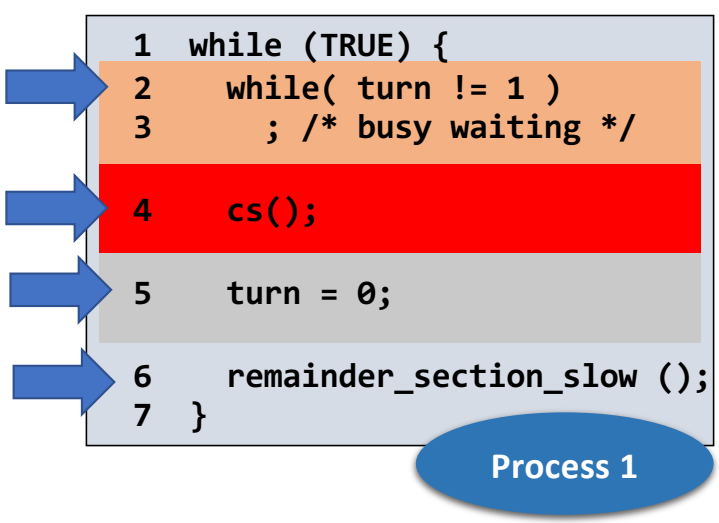
Don't print

进展违规是指在自旋锁的执行过程中，某些进程可能会一直等待，无法继续执行自己的临界区，导致进程被阻塞

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()**





Turn = 1

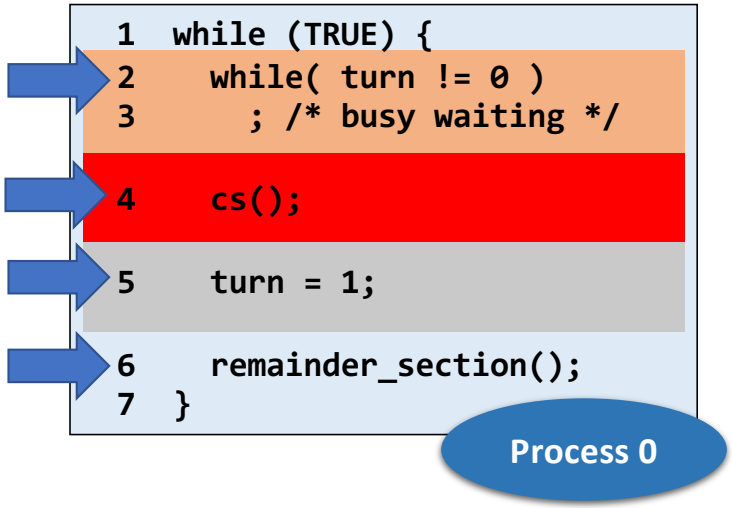
Don't print

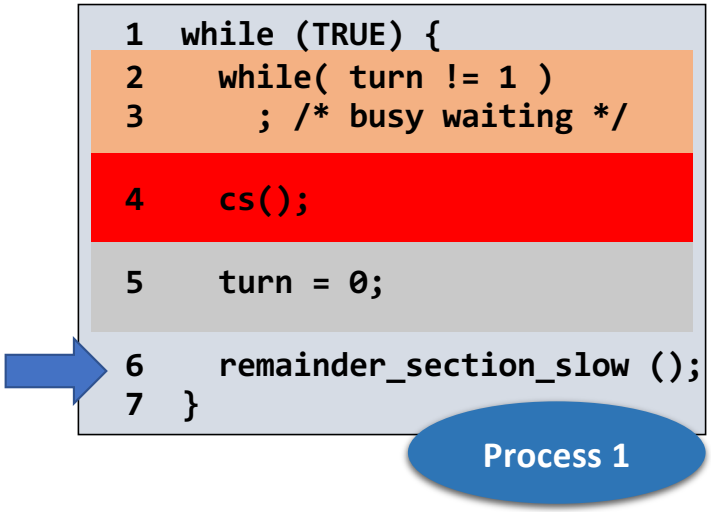
Spin-based Locks: Progress Violation

- 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()**

process1在cs（）外面，实际上process0可以执行critical section但是只能空转等待

Has to wait…





Turn = 1

Peterson's Solution: Improved Spin-based Locks

```
 1  int turn;                          /* whose turn is it next */
 2  int interested[2] = {FALSE,FALSE};  /* express interest to enter cs*/
 3
 4  void lock( int process ) {  /* process is 0 or 1 */
 5    int other;                       /* number of the other process */
 6    other = 1-process;               /* other is 1 or 0 */
 7    interested[process] = TRUE;      /* express interest */
 8    turn = other;
 9    while ( turn == other &&
              interested[other] == TRUE )
10      ;    /* busy waiting */
11  }
12
13  void unlock( int process ) {    /* process: who is leaving */
14    interested[process] = FALSE;    /* I just left critical region */
15  }
```

Peterson's Solution: Improved Spin-based Locks

```
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  }
```

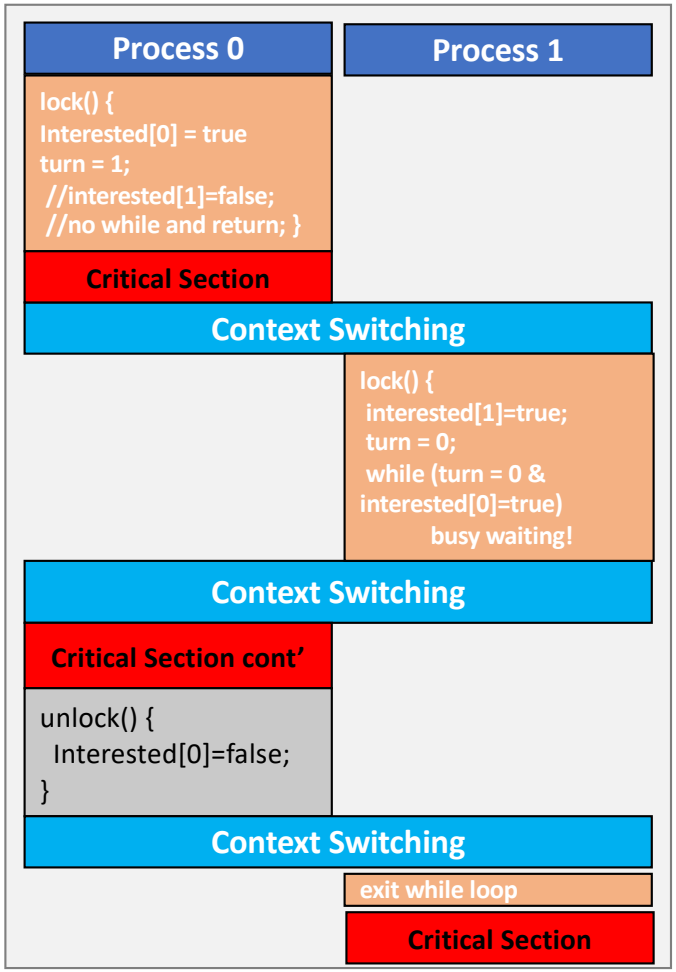
Express interest to enter CS

Being polite and let other go first

If other is not interested, I can always go ahead

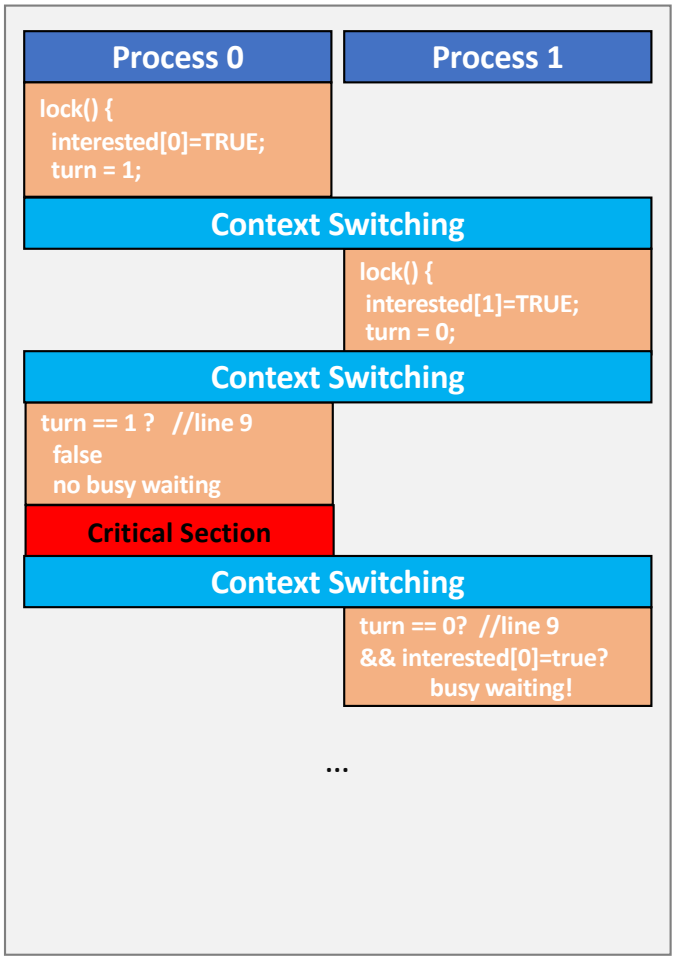
Peterson's Solution

```
 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

```
 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);
```

每个进程对应一个 waiting[i]，表示进程i是否想要进入临界区。waiting[i] = TRUE 表示进程i正在等待进入临界区，FALSE表示它不再等待

lock = TRUE 表示一个进程已经进入临界区，lock = FALSE表示没有进程在临界区。

Multi-Process Mutual Exclusion (Cont'd)

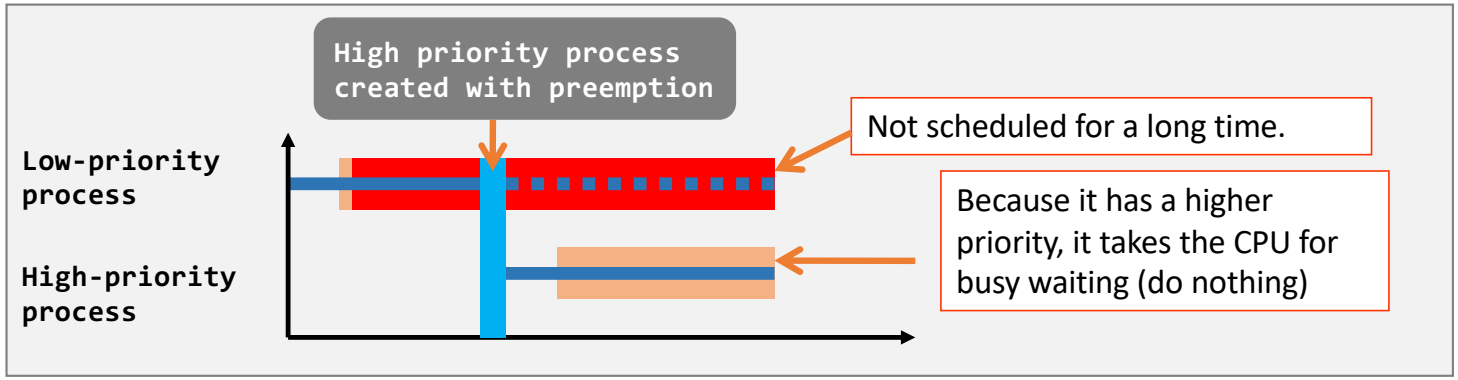
- Support n processes
  - boolean waiting[n]
  - boolean lock
    - initially FALSE
- A process can enter the critical section if either **waiting[i] == FALSE** or **key == FALSE**
  - key is local variable
  - All process must execute test_and_set() at least once
  - The first one call test_and_set() 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



H被赋予较高的优先级，但因为L持有锁，H不能立即进入，这种情况被称为优先级反转。H实际上无法执行有意义的操作，只是在“忙等待”中浪费CPU时间。

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)

H被赋予较高的优先级，但因为L持有锁，H不能立即进入，这种情况被称为优先级反转。H实际上无法执行有意义的操作，只是在“忙等待”中浪费CPU时间。

信号量的实现：需要通过操作系统内核来实现进程的休眠和唤醒。这通常涉及原子操作来保证信号量的更新是安全的，避免竞态条件。

Semaphore

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

Section Entry: sem_wait()

```
1   void sem_wait(semaphore *s) {
2
3       s->value = s->value - 1;
4       if ( s->value < 0 ) {
5
6           sleep();
7
8       }
9
10  }
```

每个调用 sem_wait 的进程都要减去信号量的值，表示它正在请求一个资源。

如果 s->value 小于 0（表示没有资源或其他进程正在使用），进程进入 睡眠状态（sleep()）并等待其他进程释放资源

Initialize **s->value** = 1

通常在初始化时设置为 1，表示有一个资源可以被访

"**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**

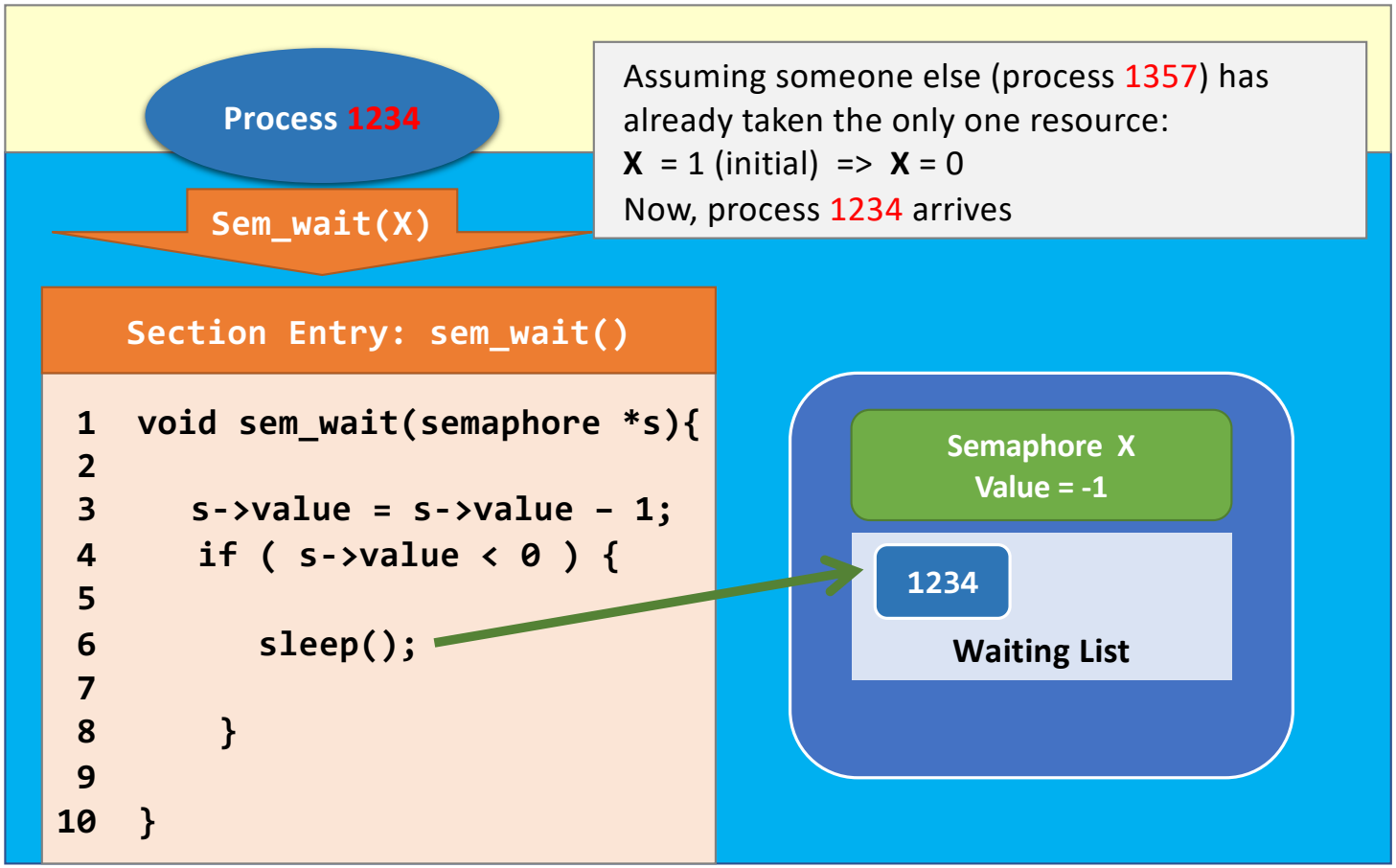
Section Exit: sem_post()

```
1   void sem_post(semaphore *s) {
2
3     s->value = s->value + 1;
4     if ( s->value <= 0 )
5       wakeup();
6
7   }
```

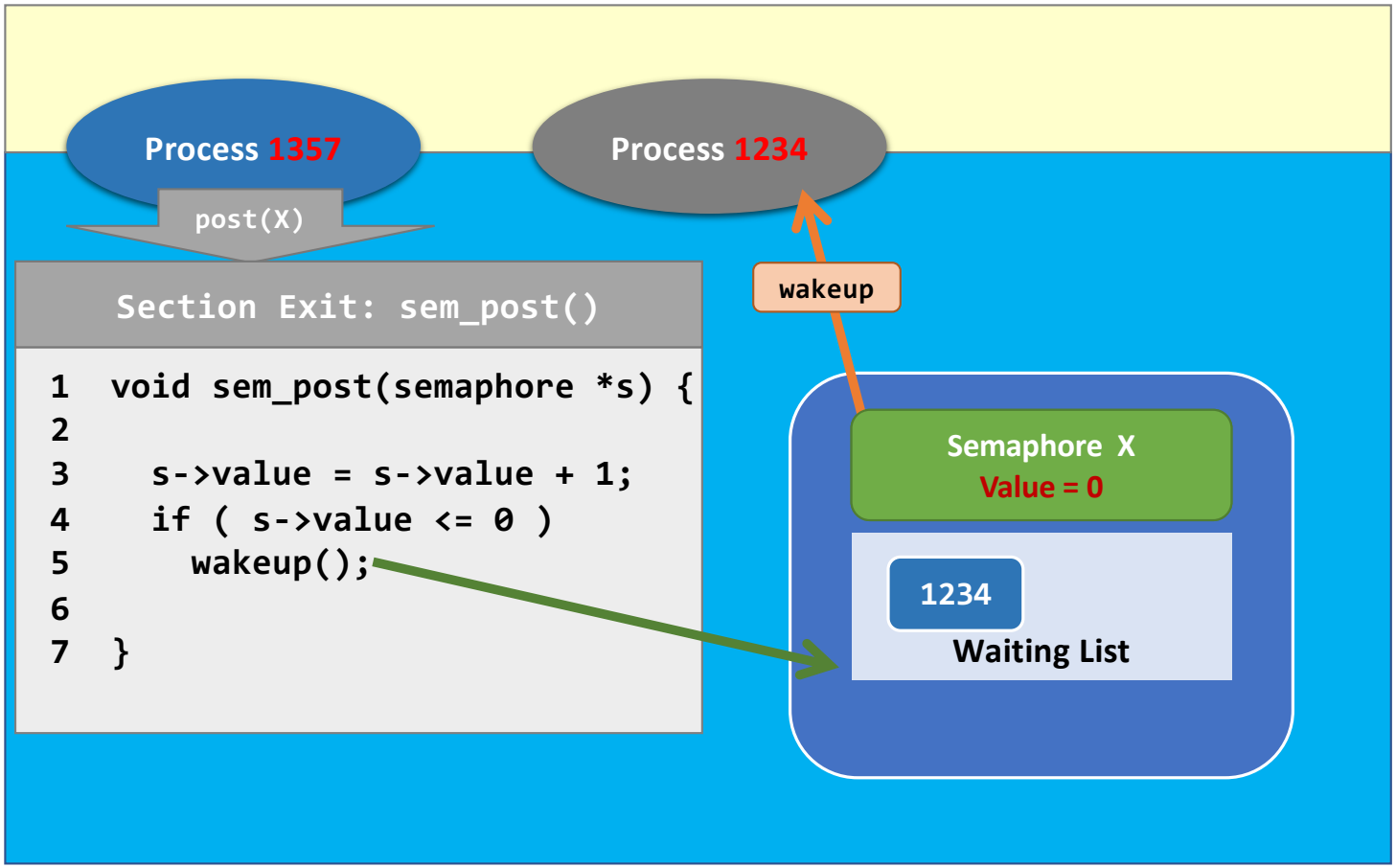
释放资源

如果 s->value 的值小于或等于 0，表示有其他进程在等待资源，于是调用 wakeup() 唤醒一个等待的进程

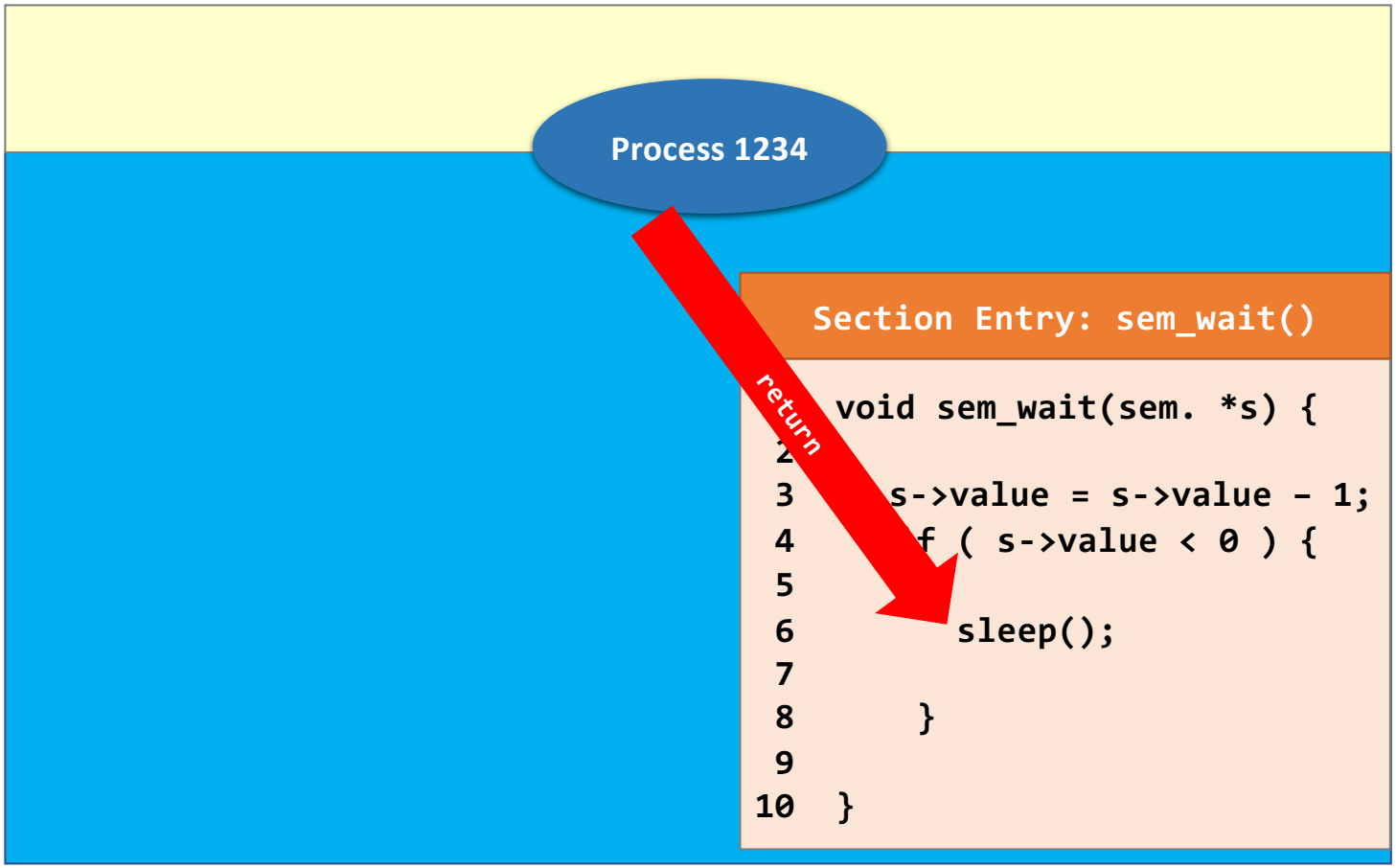
Semaphore Example



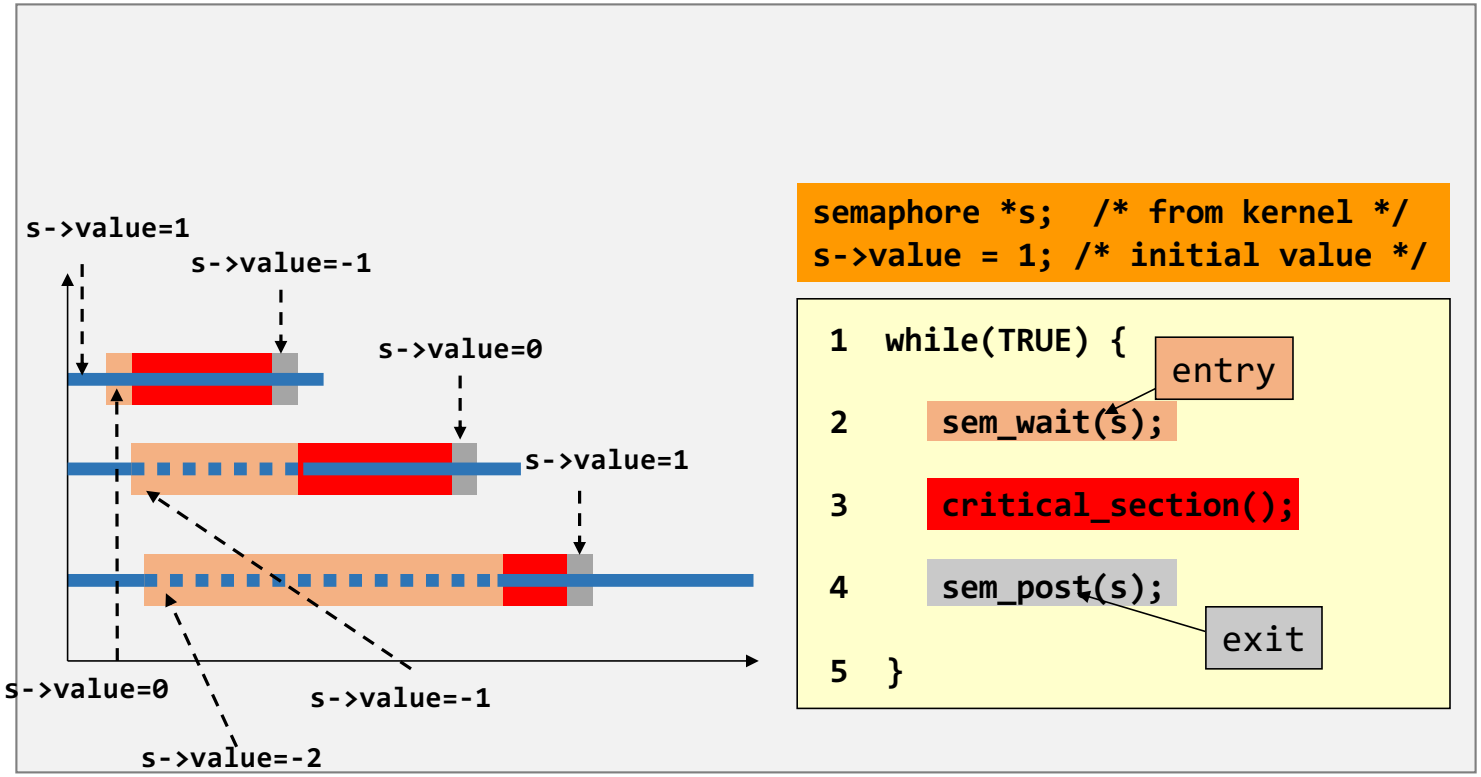
Semaphore Example



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

如果两个进程同时在同一个信号量上执行这两个操作，就可能导致竞态条件，进而破坏同步机制。

- 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

将 sem_wait() 和 sem_post() 代码放入临界区，这样可以确保这些操作是互斥执行的。

```
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;
        }
```

这个函数通过原子操作确保 addr 的值从 old 更改为 new。只有当当前值为 old 时，才会更新为 new，如果有其他进程修改了 addr 的值，当前进程会重新尝试。

Using Semaphore beyond Mutual Exclusion

- Producer-Consumer Problem
  - Two types of processes: **producer** and **consumer**;
    - At least one producer and one consumer.

生产者产生数据并放入缓冲区，消费者从缓冲区中取数据并处理。信号量可以用于控制生产者和消费者之间的同步，确保缓冲区不会溢出或空置。

- Dining Philosopher Problem
  - Only one type of process
    - At least two processes.

包含若干个哲学家，每个哲学家既要思考，也要进餐。他们共享筷子，信号量用于确保哲学家能够独占筷子，从而避免死锁和资源冲突。这个问题是一个经典的多进程同步问题，信号量用来控制每个哲学家的访问顺序。

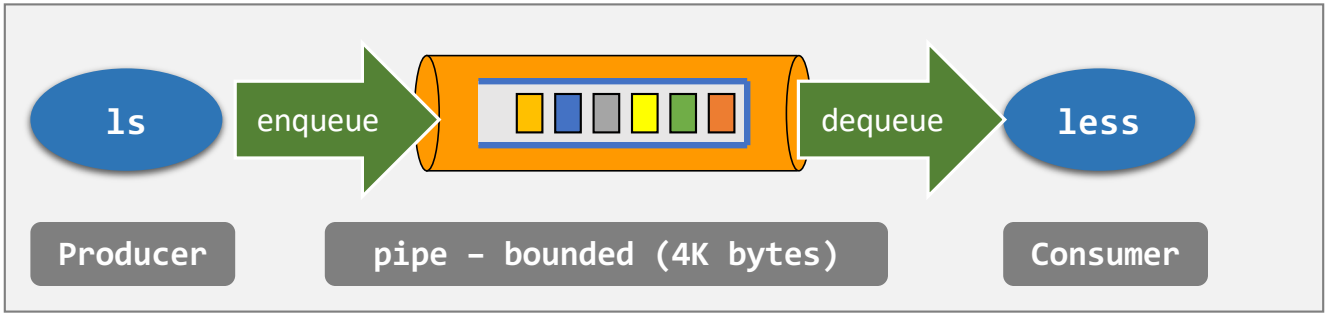
- Reader Writer Problem
  - Multiple readers, one writer

有多个读者和一个写者。读者可以同时读取共享资源，而写者在进行写操作时必须独占资源。信号量用于协调多个读者和写者的访问，确保数据的一致性和避免竞争

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.

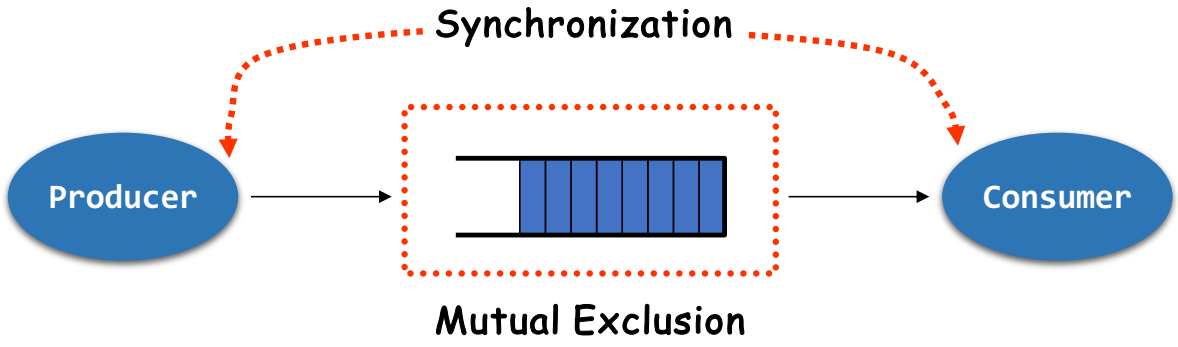


Producer-consumer Problem

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;
```

它确保在同一时刻只有一个进程可以访问缓冲区

表示缓冲区中剩余的空槽位

表示缓冲区中已填充的槽位

Note

The size of the bounded buffer is "N".

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

Abstraction of semaphore as integer!

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 }
```

退出临界区

通知consumer有新的数据项可以消费

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 }
```

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 **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  }
```

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

```
 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

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

```
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

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

```
 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 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;
```

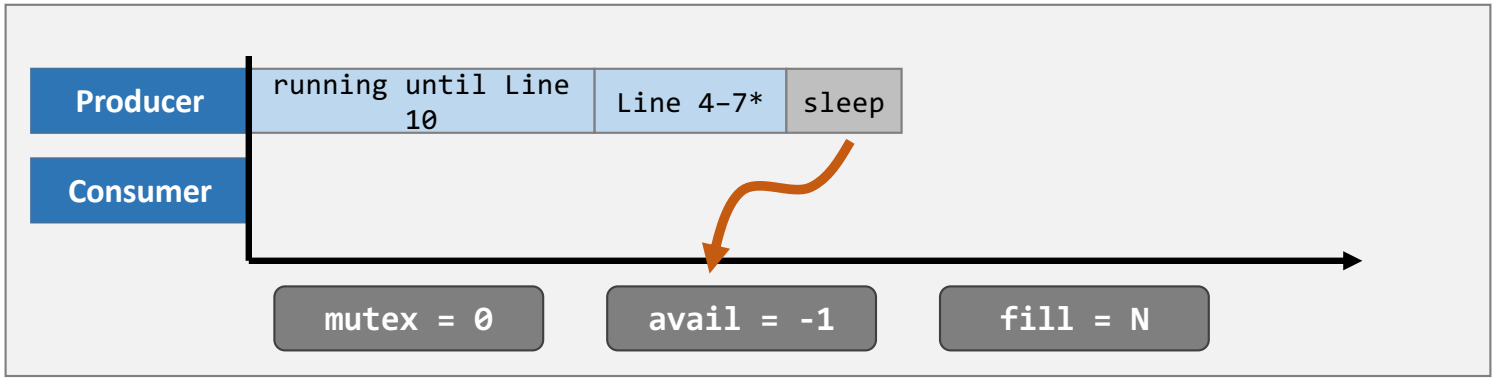
Producer function

```
 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

```
 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



Producer function

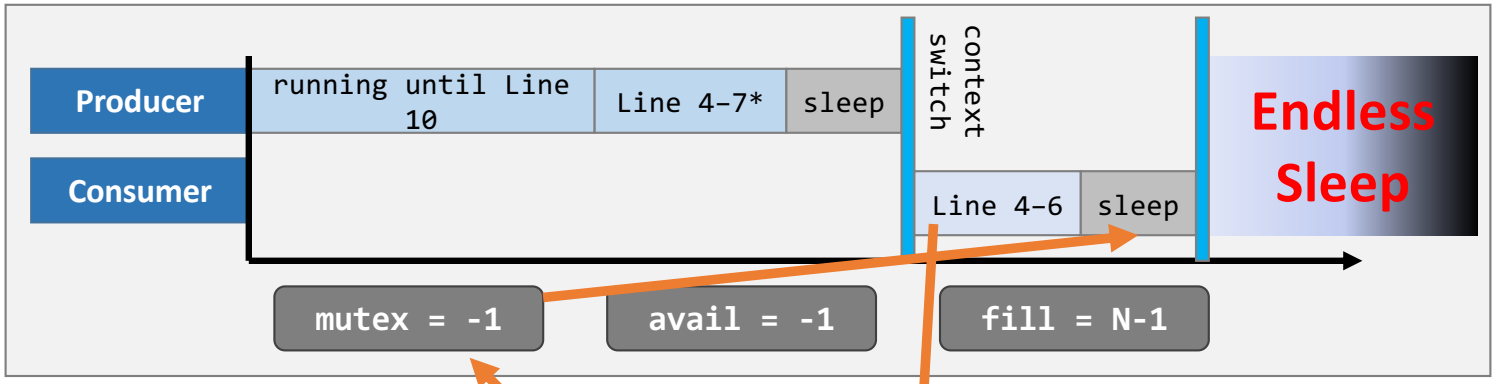
```
1   void producer(void) {
2       int item;
3
4       while(TRUE) {
5           item = produce_item();
6*          wait(&mutex);
7*          wait(&avail);
8           insert_item(item);
9           post(&mutex);
10          post(&fill);
11      }
12  }
```

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

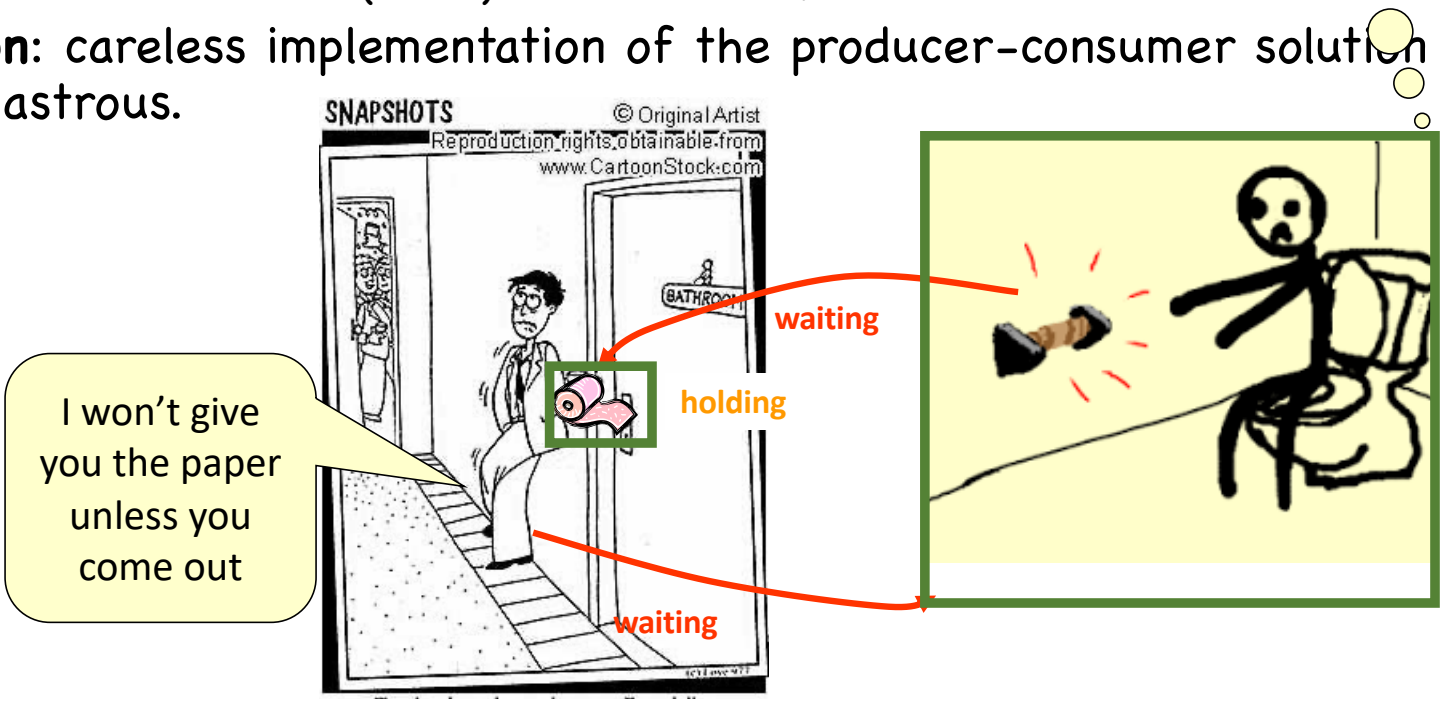
```
1  void consumer(void) {
2      int item;
3
4      while(TRUE) {
5          wait(&fill);
6          wait(&mutex);
7          item = remove_item();
8          post(&mutex);
9          post(&avail);
10         //consume the item
11     }
12 }
```

消费者等着生产者释放 mutex 锁，但生产者又因为 avail 被阻塞无法执行。

生产者等着消费者释放 avail，但消费者又因为生产者没有释放 mutex 锁而无法继续执行。

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.



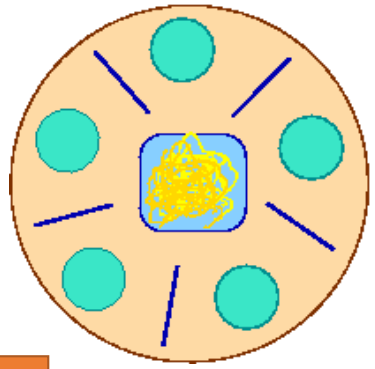
Summary on Producer-consumer Problem

- 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

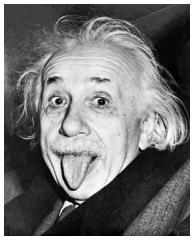
Dining Philosopher Problem

- 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 ….

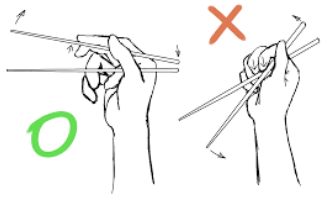


It's a multi-object synchronization problem

Dining Philosopher Problem



Philosophers

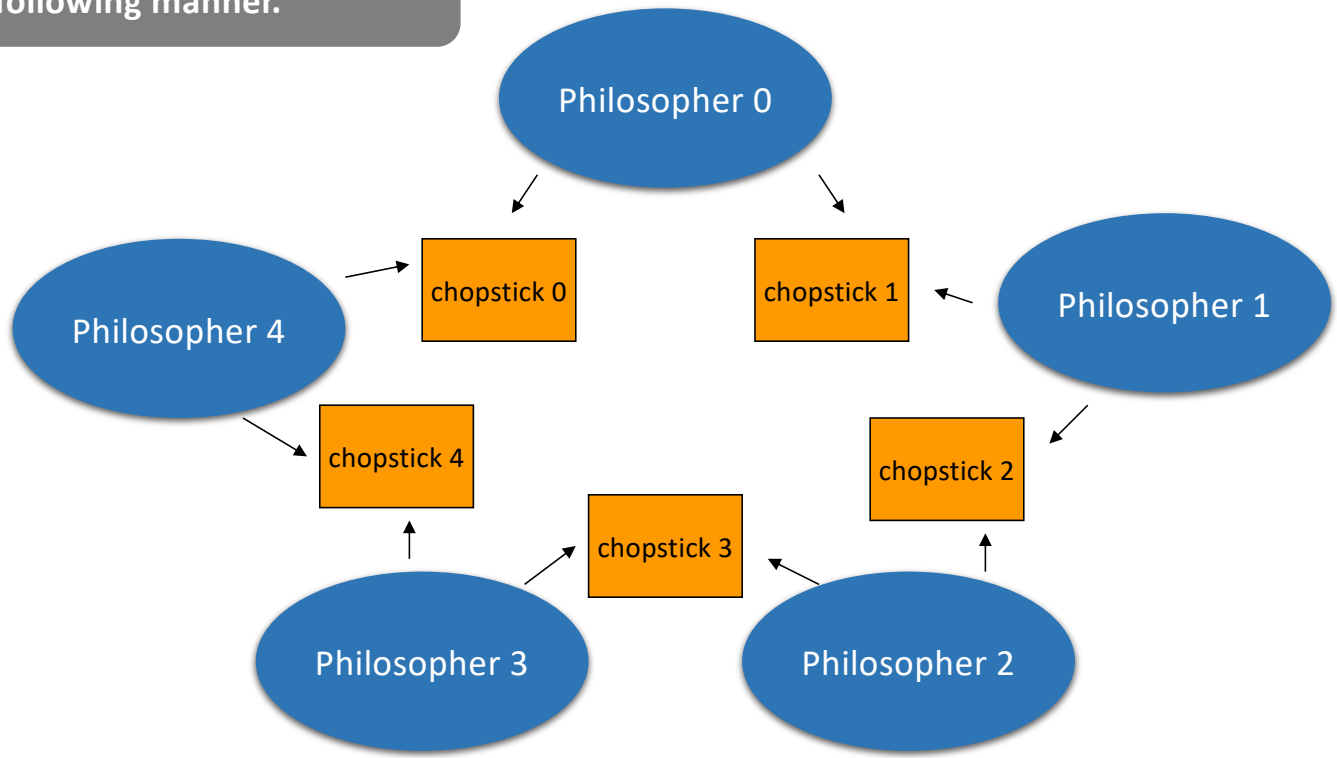


Chopsticks

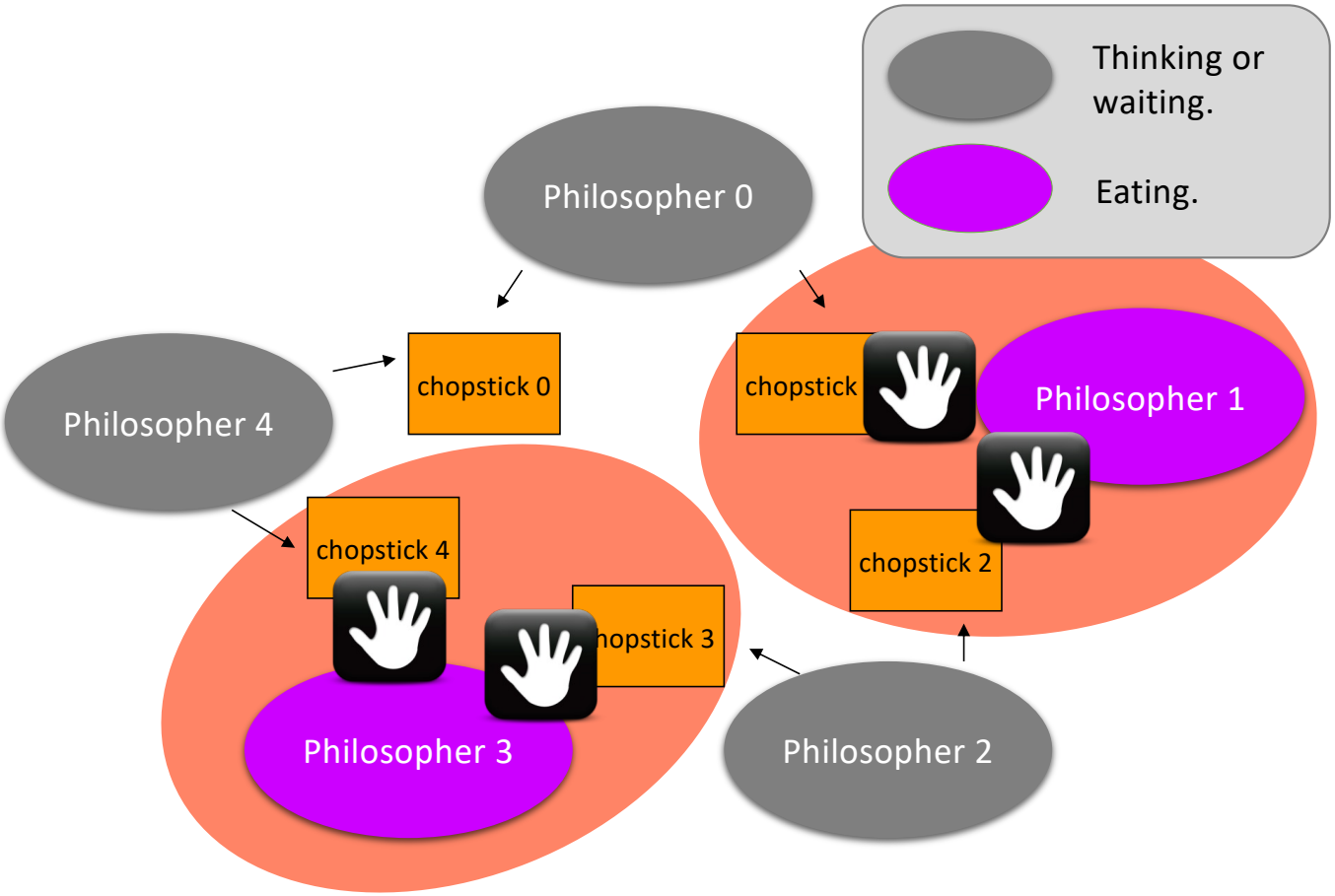
A process needs two shared resources in order to do some work

Dining Philosopher Problem

The chopsticks are arranged in the following manner.



Dining Philosopher Problem



Dining Philosopher – Requirement 1

- **Mutual exclusion**
  - 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

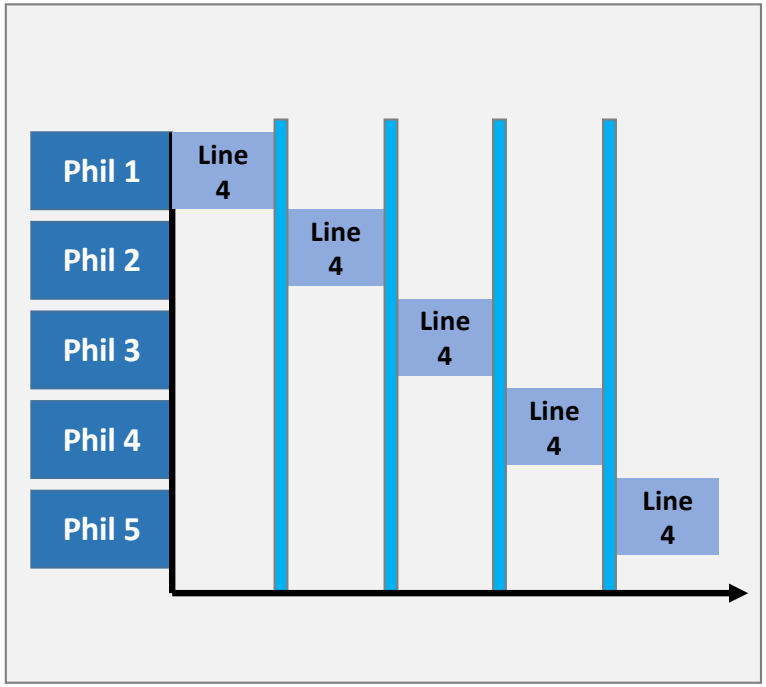
Section Exit

Main Function

```
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 – 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

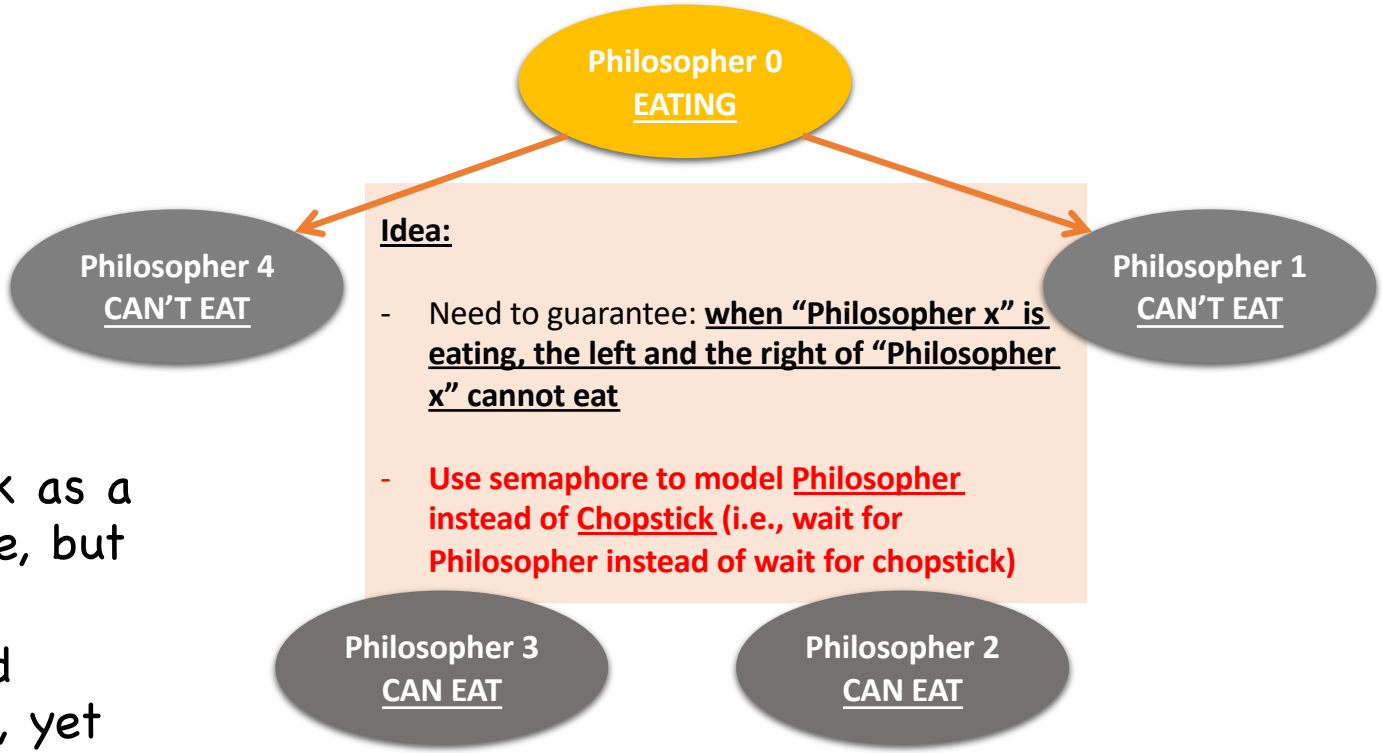
- **Potential Problem:**
  - Philosophers are all busy (no deadlock), but no progress (starvation)

- 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, ...

重新尝试拿起左边的筷子，再次进入等待状态

Dining Philosopher – before the Final Solution

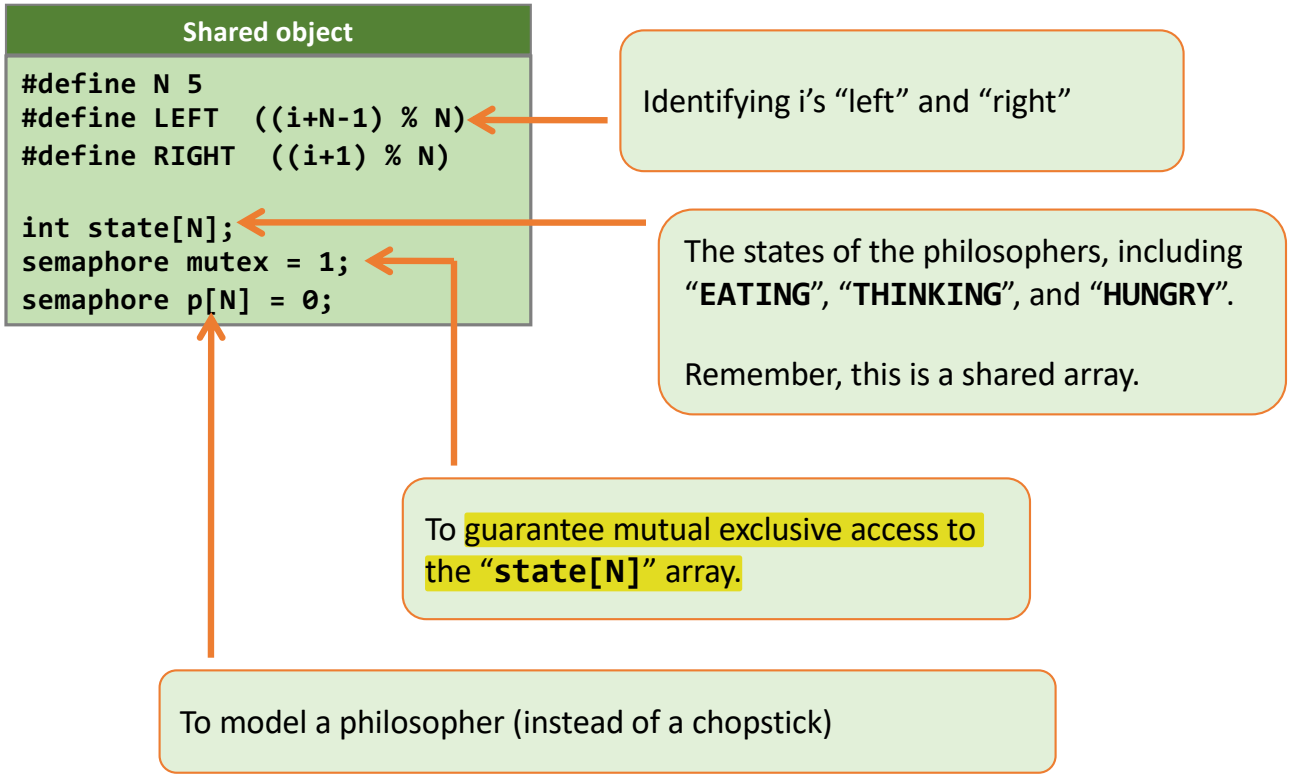
- Two Problems
  - Model each chopstick as a semaphore is intuitive, but may cause deadlock
  - Using sleep() to avoid deadlock is effective, yet creating starvation.



需要保证：当哲学家 x 在吃饭时，x 的左边和右边的哲学家不能进食。这确保了在进餐时不会造成冲突。

改进方法：通过信号量来建模哲学家，而不是建模每根筷子。即，使用信号量来控制哲学家之间的同步，保证当一个哲学家在吃饭时，相邻的哲学家不能同时进食。

Dining Philosopher – Final Solution



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

```
1  void philosopher(int i) {
2      think();
3      take_chopsticks(i);
4      eat();
5      put_chopsticks(i);
6  }
```

Section entry

```
1  void take_chopsticks(int i) {
2      wait(&mutex);
3      state[i] = HUNGRY;
4      captain(i);
5      post(&mutex);
6      wait(&p[i]);
7  }
```

如果哲学家可以开始进餐，它就会等待 p[i] 信号量，确保哲学家开始进餐

Section exit

```
1  void put_chopsticks(int i) {
2      wait(&mutex);
3      state[i] = THINKING;
4      captain(LEFT);
5      captain(RIGHT);
6      post(&mutex);
7  }
```

尝试让左右边的人吃

```
void wait(semaphore *s) {

   *s = *s - 1;
   if ( *s < 0 ) {

      sleep();

    }

}
```

```
void post(semaphore *s) {

  *s = *s + 1;
  if ( *s <= 0 )
    wakeup();

}
```

Extremely important helper function

```
1 void captain(int i) {
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3         state[i] = EATING;
4         post(&p[i]);
5     }
6 }
```

Dining Philosopher – Final Solution

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

```
1  void take_chopsticks(int i) {
2      wait(&mutex);
3      state[i] = HUNGRY;
4      captain(i);
5      post(&mutex);
6      wait(&p[i]);
7  }
```

Critical Section

The captain is "indivisible"

Extremely important helper function

```
1 void captain(int i) {
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3         state[i] = EATING;
4         post(&p[i]);
5     }
6 }
```

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

```
1  void put_chopsticks(int i)
{
2       wait(&mutex);
3       state[i] = THINKING;
4       captain(LEFT);
5       captain(RIGHT);
6       post(&mutex);
7  }
```

Extremely important helper function

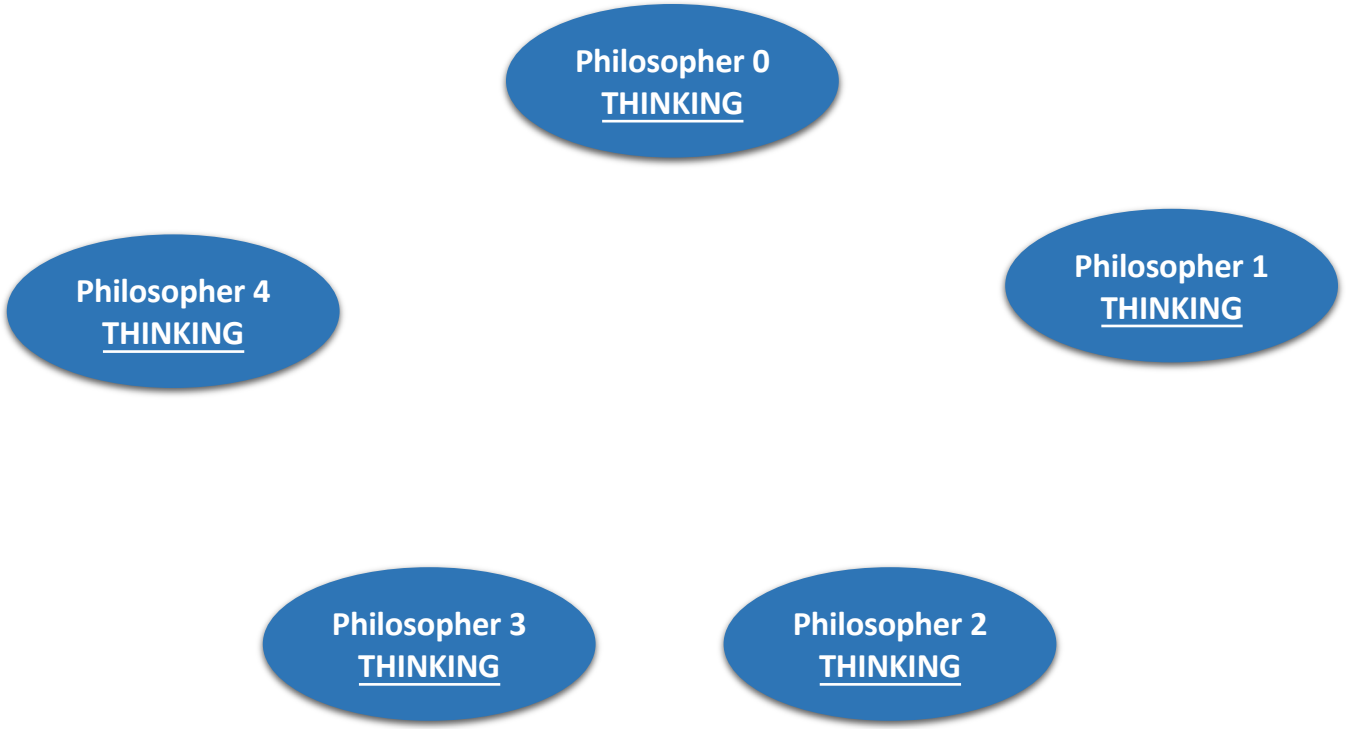
```
1 void captain(int i) {
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3         state[i] = EATING;
4         post(&p[i]);
5     }
6 }
```

Wake up the one who is sleeping

Dining Philosopher – Final Solution

Don't print

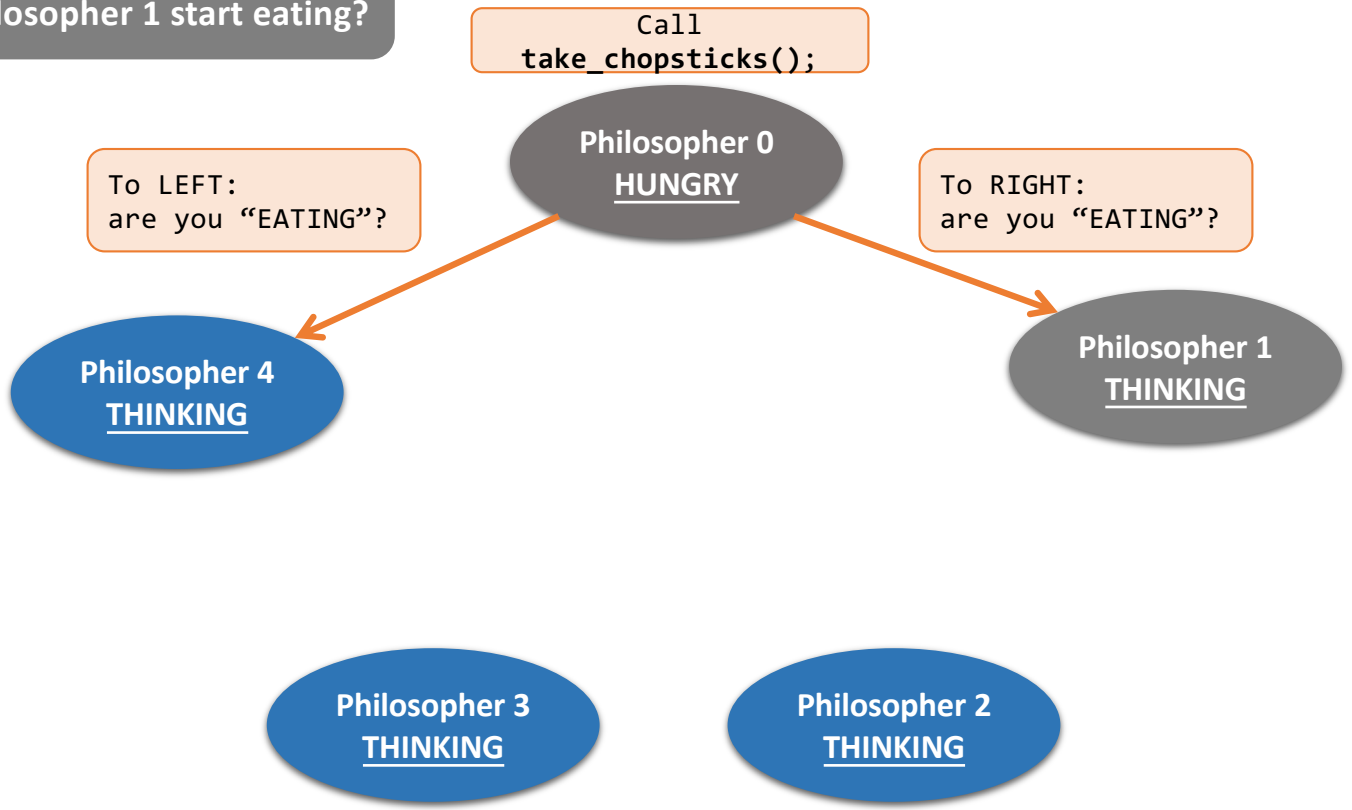
An illustration: How can Philosopher 1 start eating?



Dining Philosopher – Final Solution

Don't print

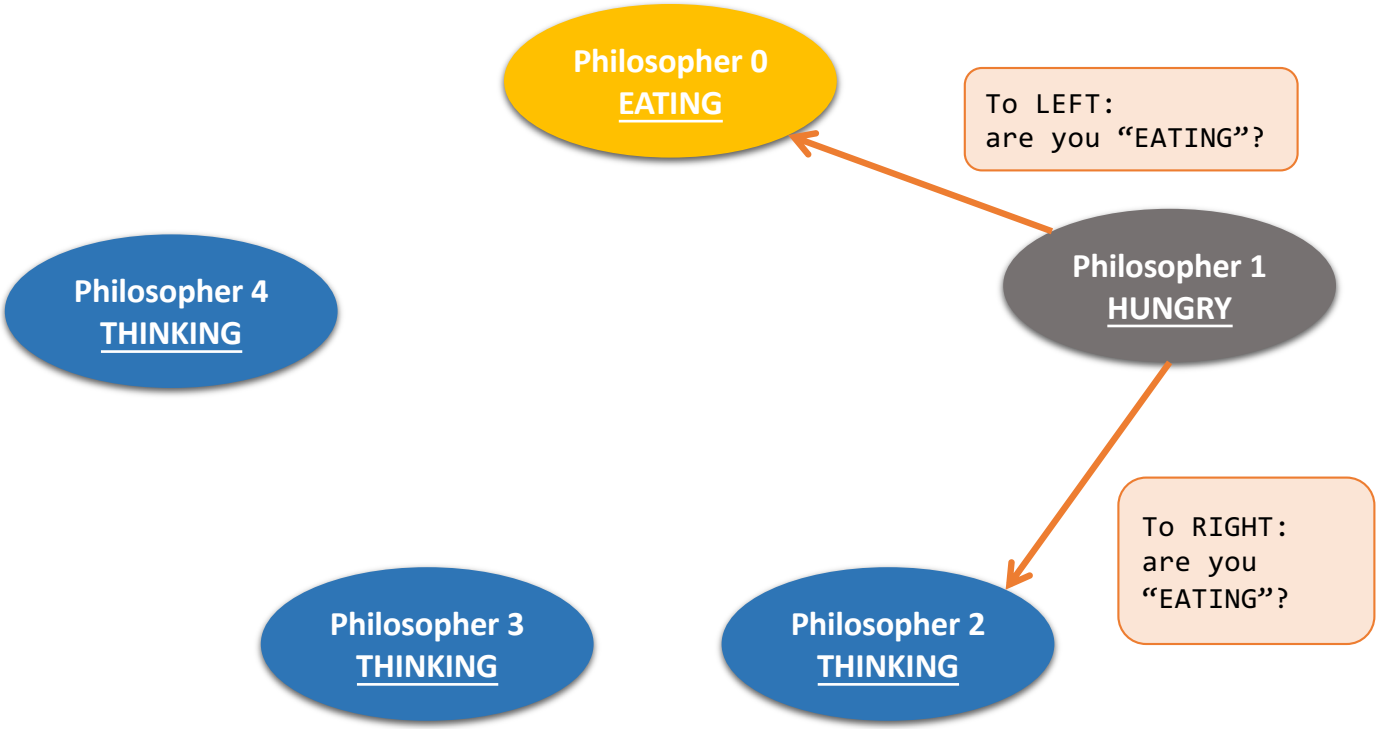
An illustration: How can Philosopher 1 start eating?



Dining Philosopher – Final Solution

Don't print

An illustration: How can Philosopher 1 start eating?



Dining Philosopher – Final Solution

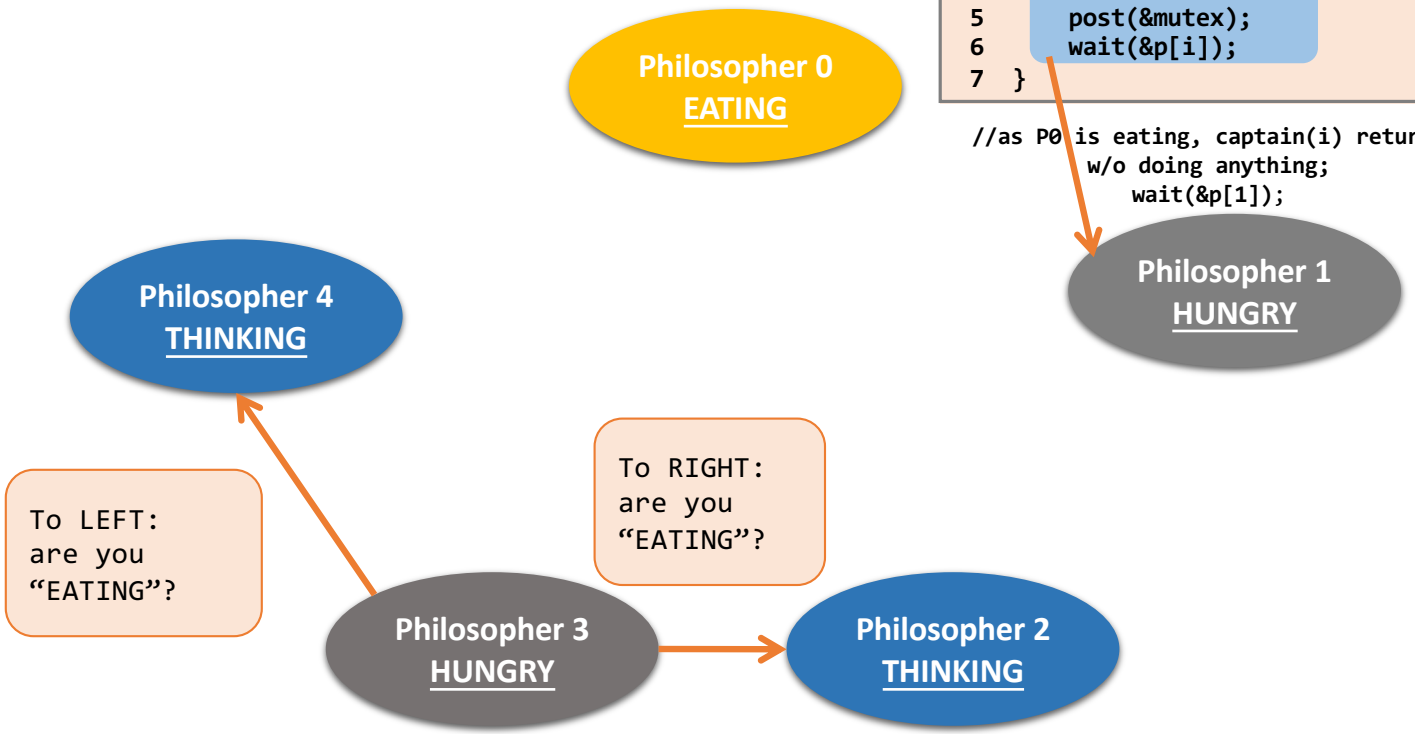
Don't print

An illustration: How can Philosopher 1 start eating?

Section entry

```
1  void take_chopsticks(int i) {
2      wait(&mutex);
3      state[i] = HUNGRY;
4      captain(i);
5      post(&mutex);
6      wait(&p[i]);
7  }
```

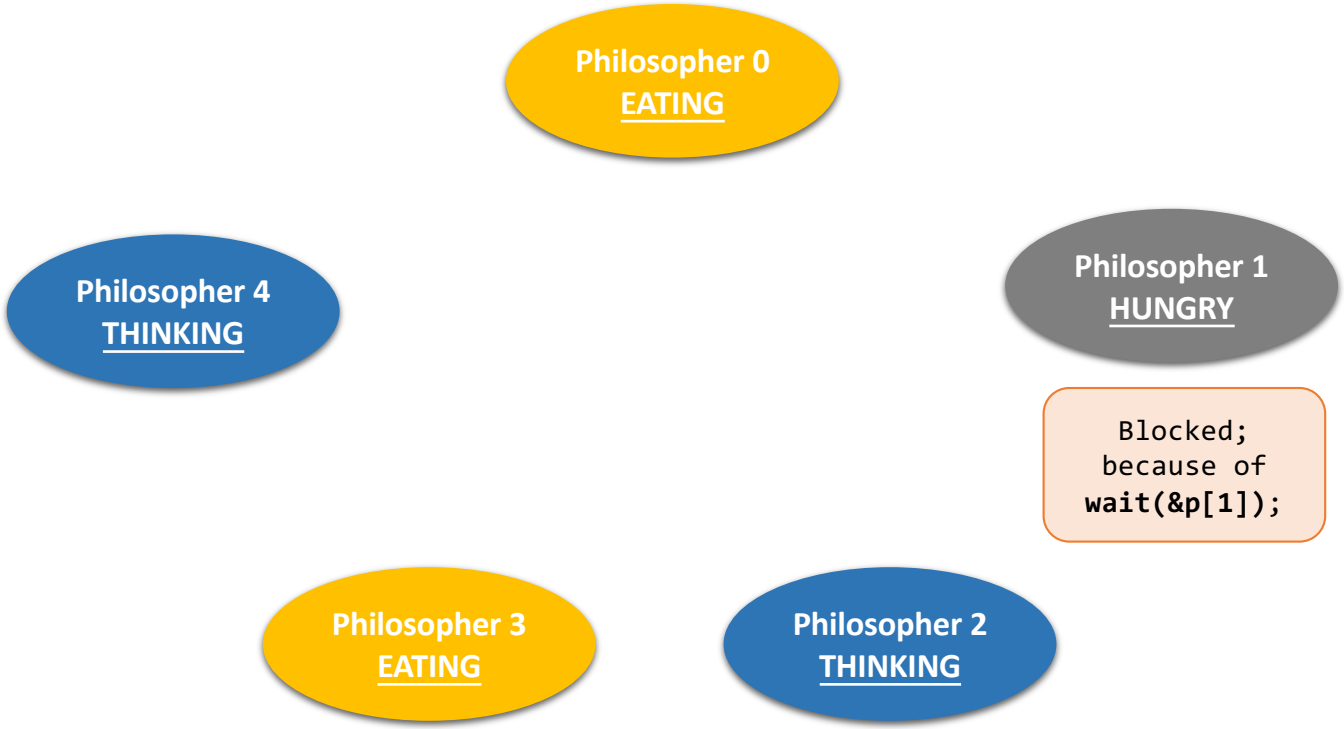
//as P0 is eating, captain(i) returns w/o doing anything;
wait(&p[1]);



Dining Philosopher – Final Solution

Don't print

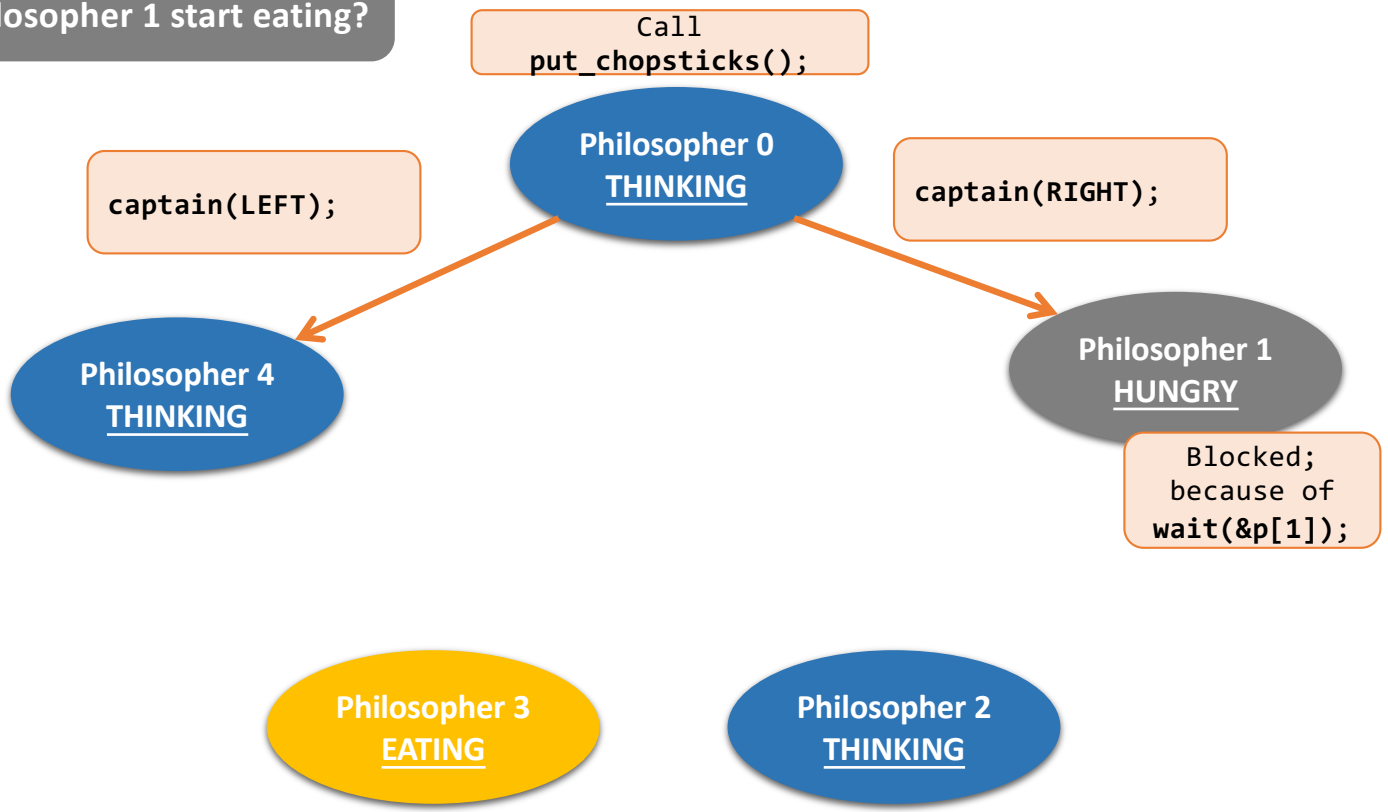
An illustration: How can Philosopher 1 start eating?



Dining Philosopher – Final Solution

Don't print

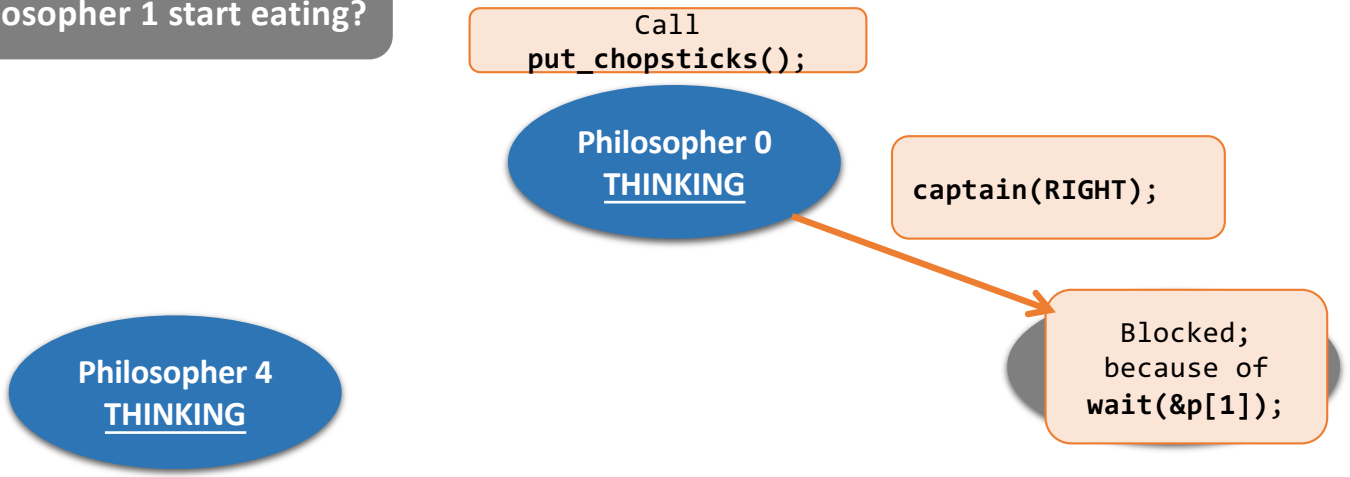
An illustration: How can Philosopher 1 start eating?



Dining Philosopher – Final Solution

Don't print

An illustration: How can Philosopher 1 start eating?




```
1 void captain(int i) {
2     if(state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
3         state[i] = EATING;
4         post(&p[i]);
5     }
6 }
```

Wake up !

Dining Philosopher – Final Solution

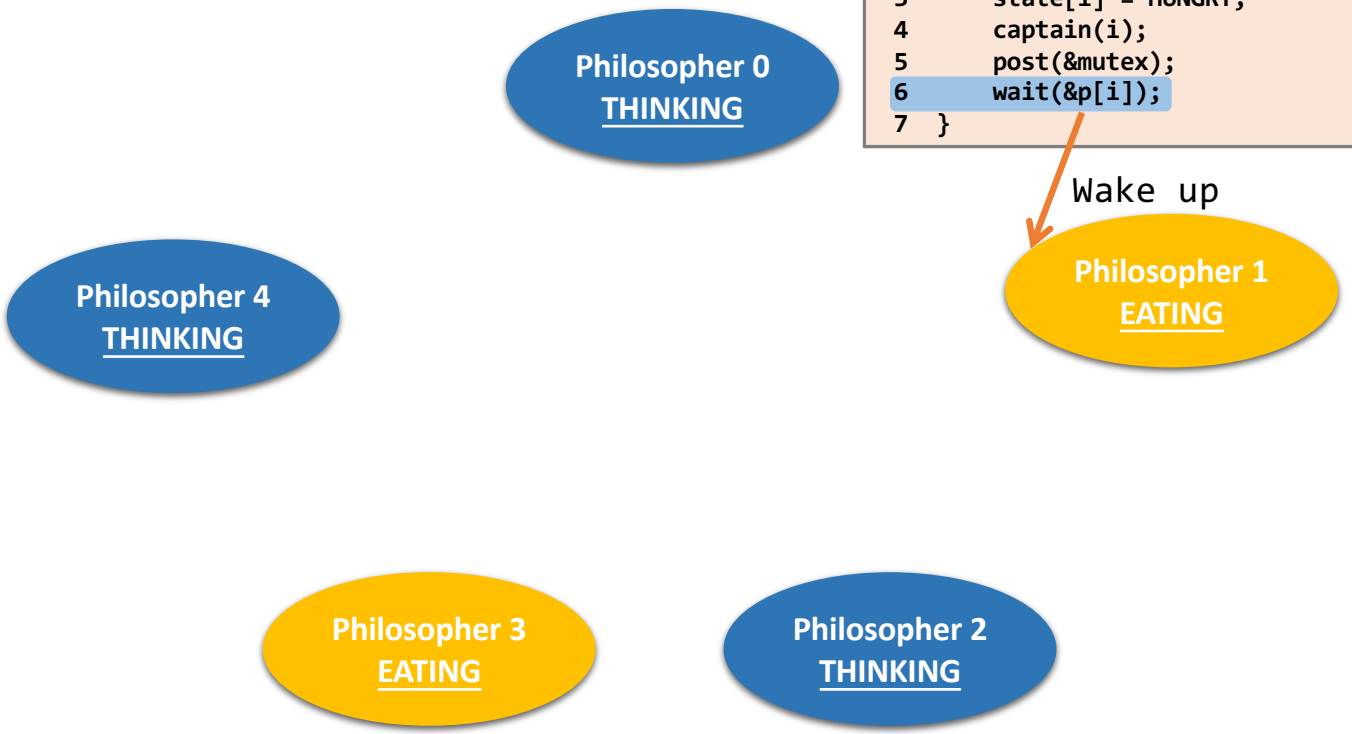
Don't print

An illustration: How can Philosopher 1 start eating?

Section entry

```
1  void take_chopsticks(int i) {
2      wait(&mutex);
3      state[i] = HUNGRY;
4      captain(i);
5      post(&mutex);
6      wait(&p[i]);
7  }
```

Wake up



Thank you!