

Operating Systems

Eric Lo

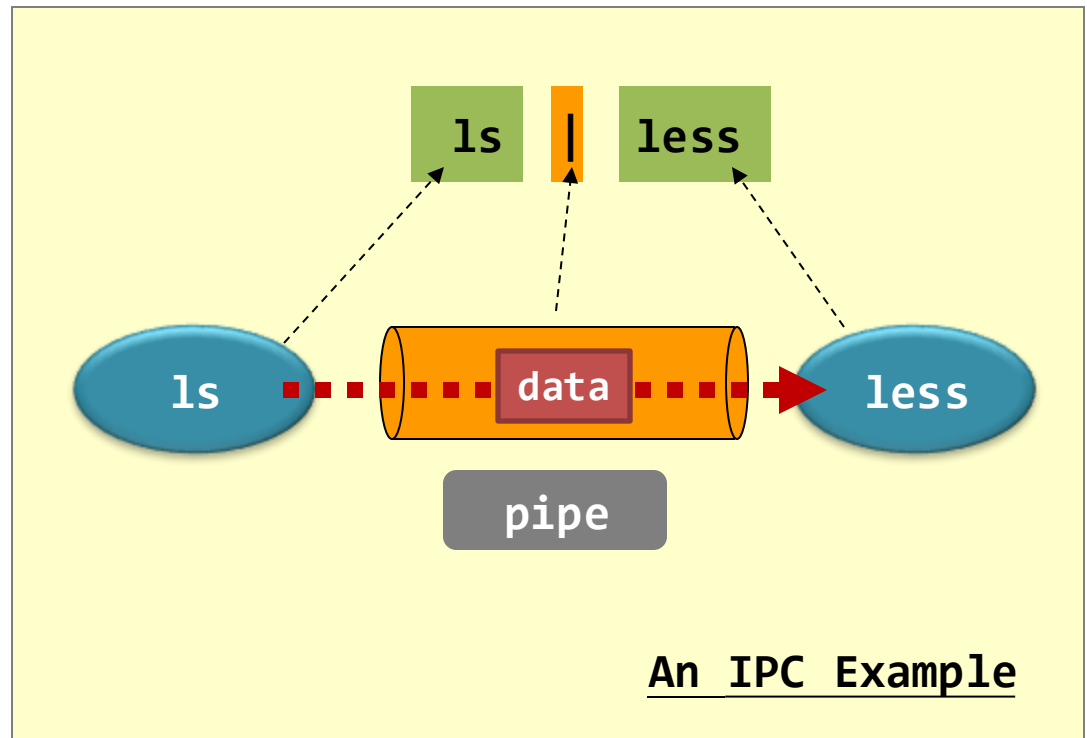
9 – Synchronization I

https://en.wikipedia.org/wiki/Inter-process_communication

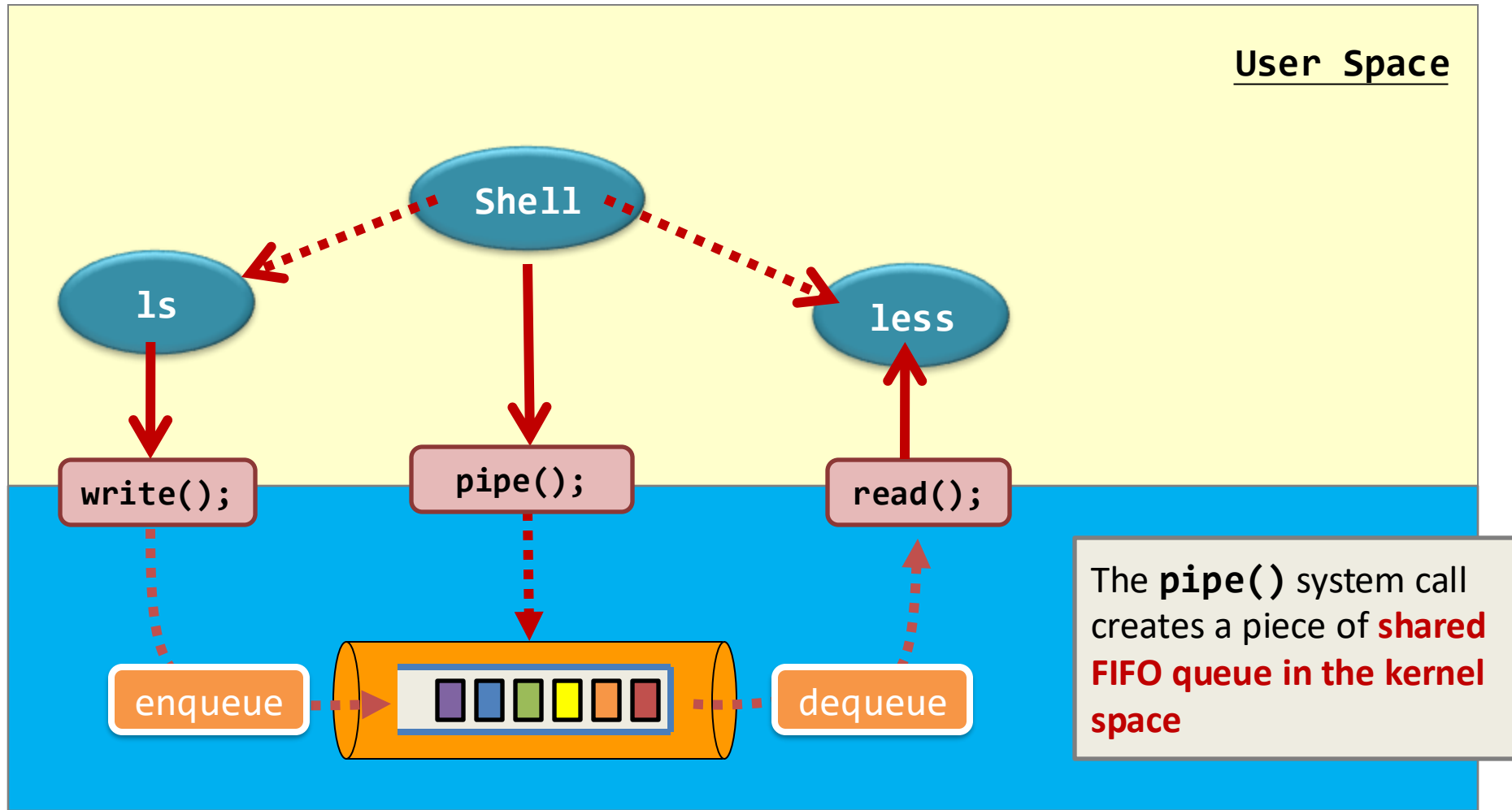
Inter-process communication

- Signal
 - More kernel-level
 - Limited to ~32 signals (SIGCHLD, SIG...)
- Pipe
 - Unidirectional
 - Between processes with a common ancestor (e.g., `ls | less`; ancestor=shell)

Pipe is a **shared object** between two processes.



“ls | less” in kernel

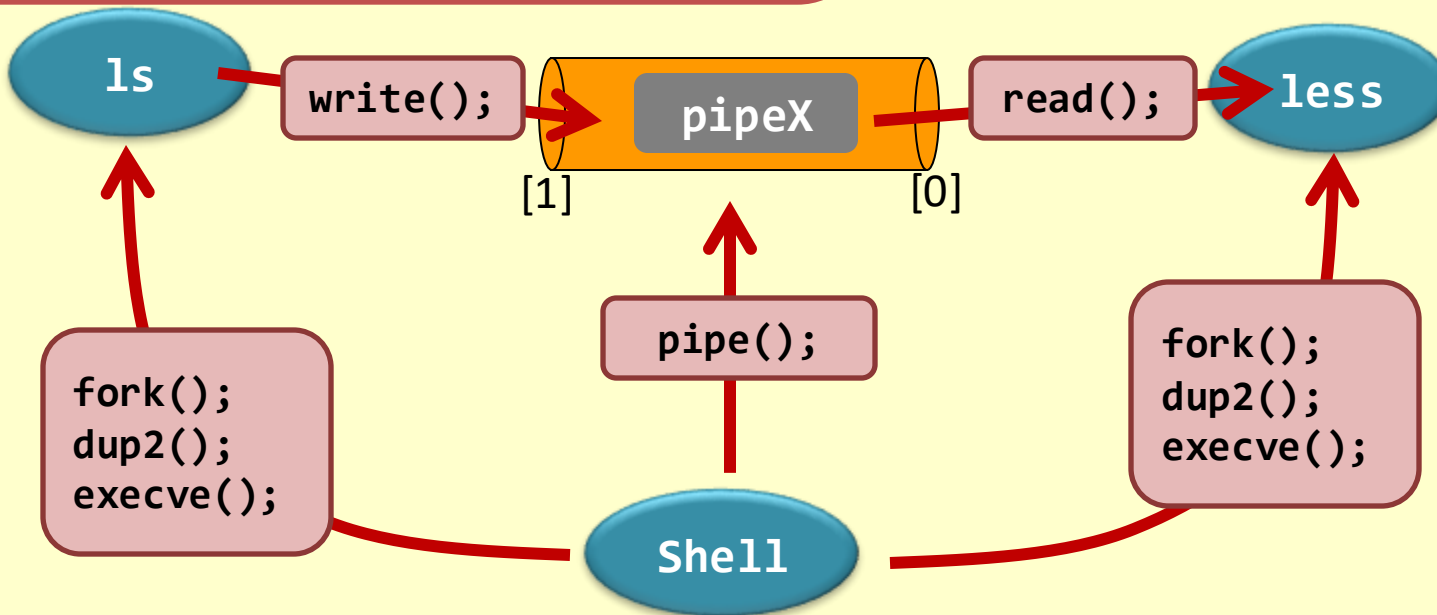


Programming “ls | less”

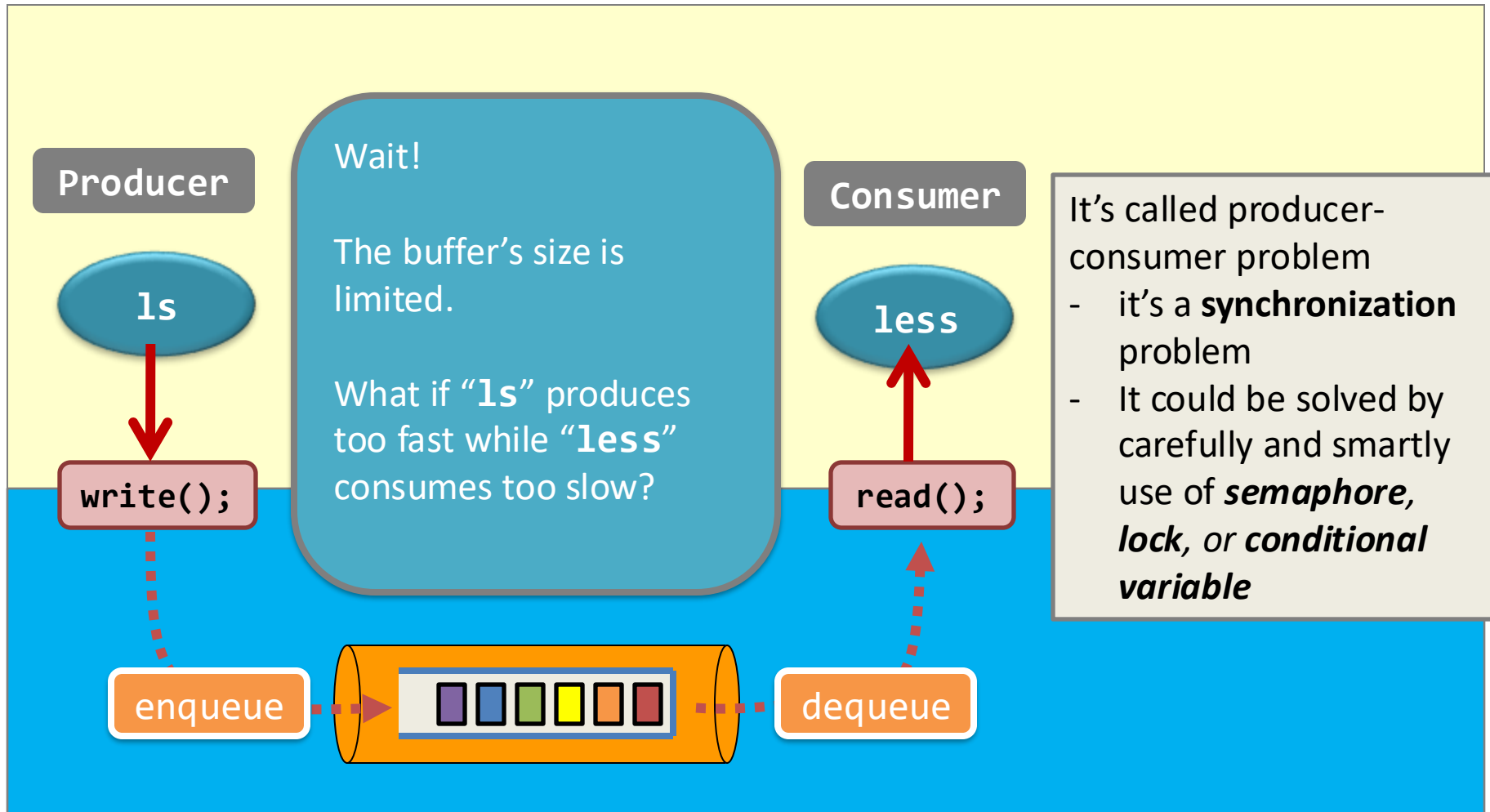
```
fork();  
if (pid==0) { // child; “ls”  
    //dup2: replace child’s stdout  
    //by the write end of the pipe  
    dup2(pipeX[1], STDOUT_FILENO);  
    execlp(“ls”, “ls”, NULL);  
} else ... //parent; “less”
```

In *nix, “everything is a file”

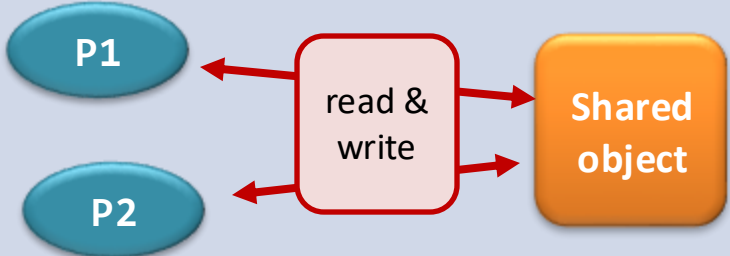
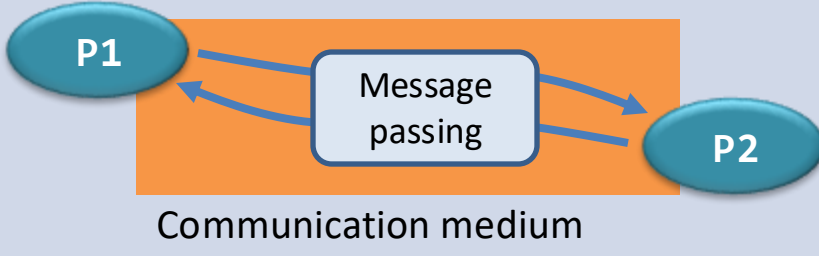
- Every resource that can read/write is represented as a file. E.g.,
 - Network, Disk, Keyboard
- A “file” is indexed by a number called *file descriptor* in the PCB



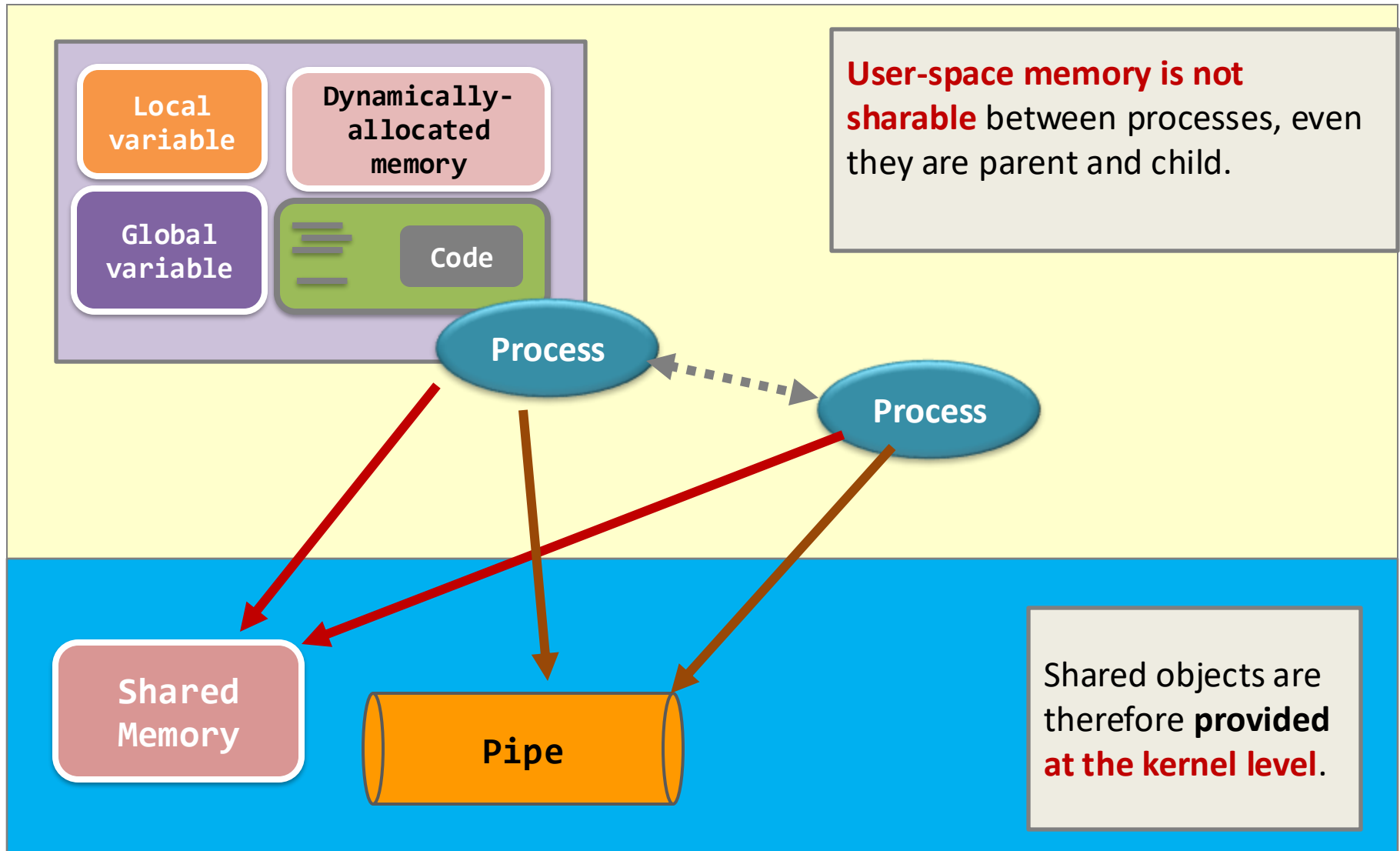
Implementing | is a synchronization problem



Concurrency Programming

Shared Objects	Message Passing
	
<p>E.g.,</p> <ul style="list-style-type: none">• shared files (on disk; slow)• pipes (restricted, but OS takes care of synchronization for you)• shared memory (primitive, general, but synchronization is on you)• shared address space (threading)	<p>E.g.,</p> <ul style="list-style-type: none">• message passing interface (MPI) library for computing clusters (CSCI4160)• GoLang
<ul style="list-style-type: none">- Usually single-node communication- More efficient- Need to take great care of synchronization because of sharing the same object	<ul style="list-style-type: none">- Usually multi-node communication- Less efficient- Less troublesome in synchronization- But need to care of other faults (e.g., what if a network link is broken?)

Evil source: the shared objects.



Evil source: the shared objects.

- Kernel provides you “pipe” to do **one-way** data flow between **2 processes** from the **same ancestor**
 - Super restrictive
- Other IPC problems beyond pipe?
 - Same process
 - Different threads can share through global variables, etc.
 - Different processes?
 - Use shared memory ~~and shared files~~
- **Concurrent access may yield the horrible data race!**
 - Kernel won't take care of that for you
 - You take care of that

Data Race

```
./pthread_race_condition 1000000 5
```

Understanding the problem...

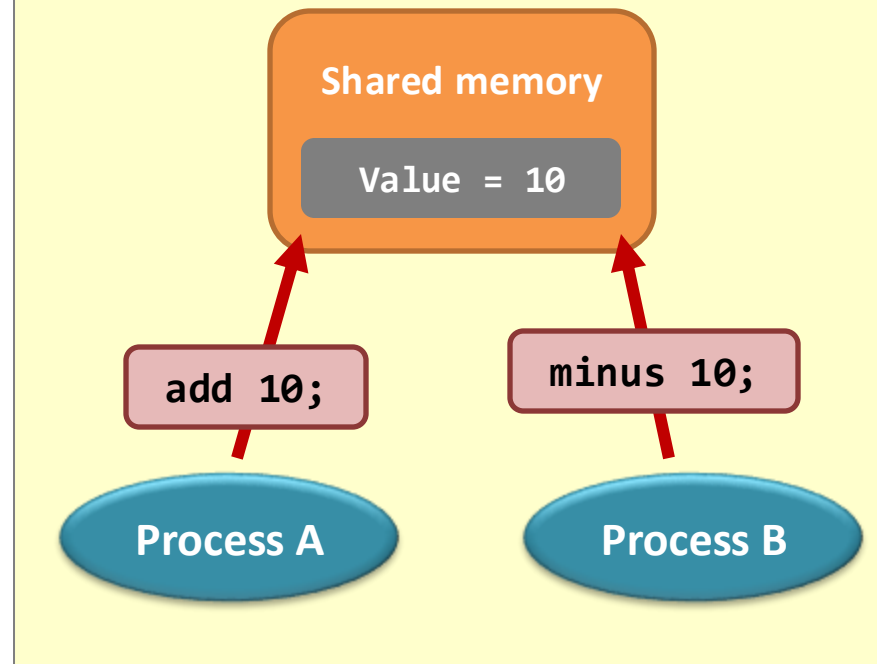
High-level language for Program A

```
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



Shared memory - X

Value = 10

State:
Ready

Register A
Value = 0

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

Process A

Register B
Value = 0

State:
Ready

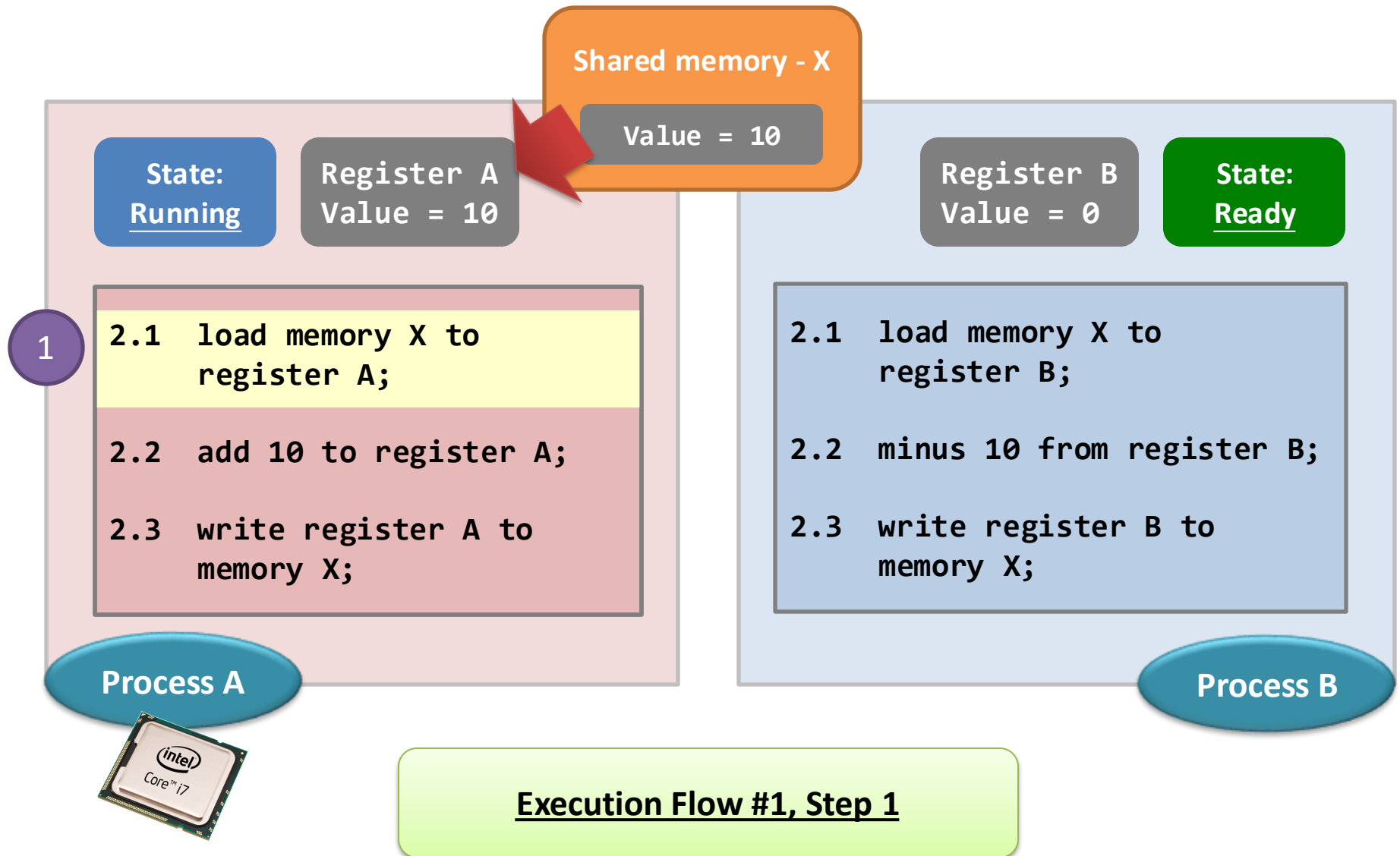
2.1 load memory X to
register B;
2.2 minus 10 from register B;
2.3 write register B to
memory X;

Process B

The initial setting

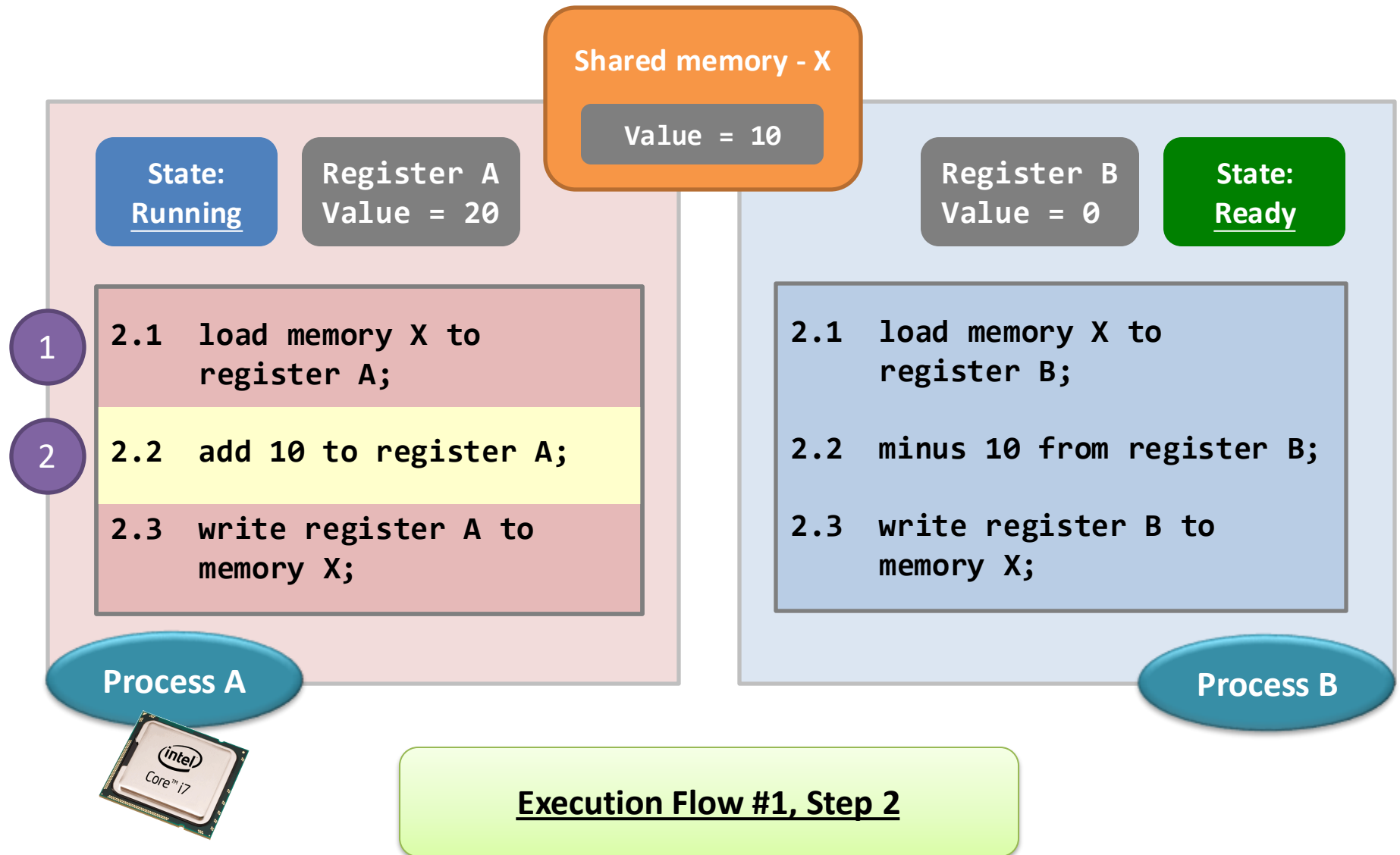
Problem not yet arise...

Don't print



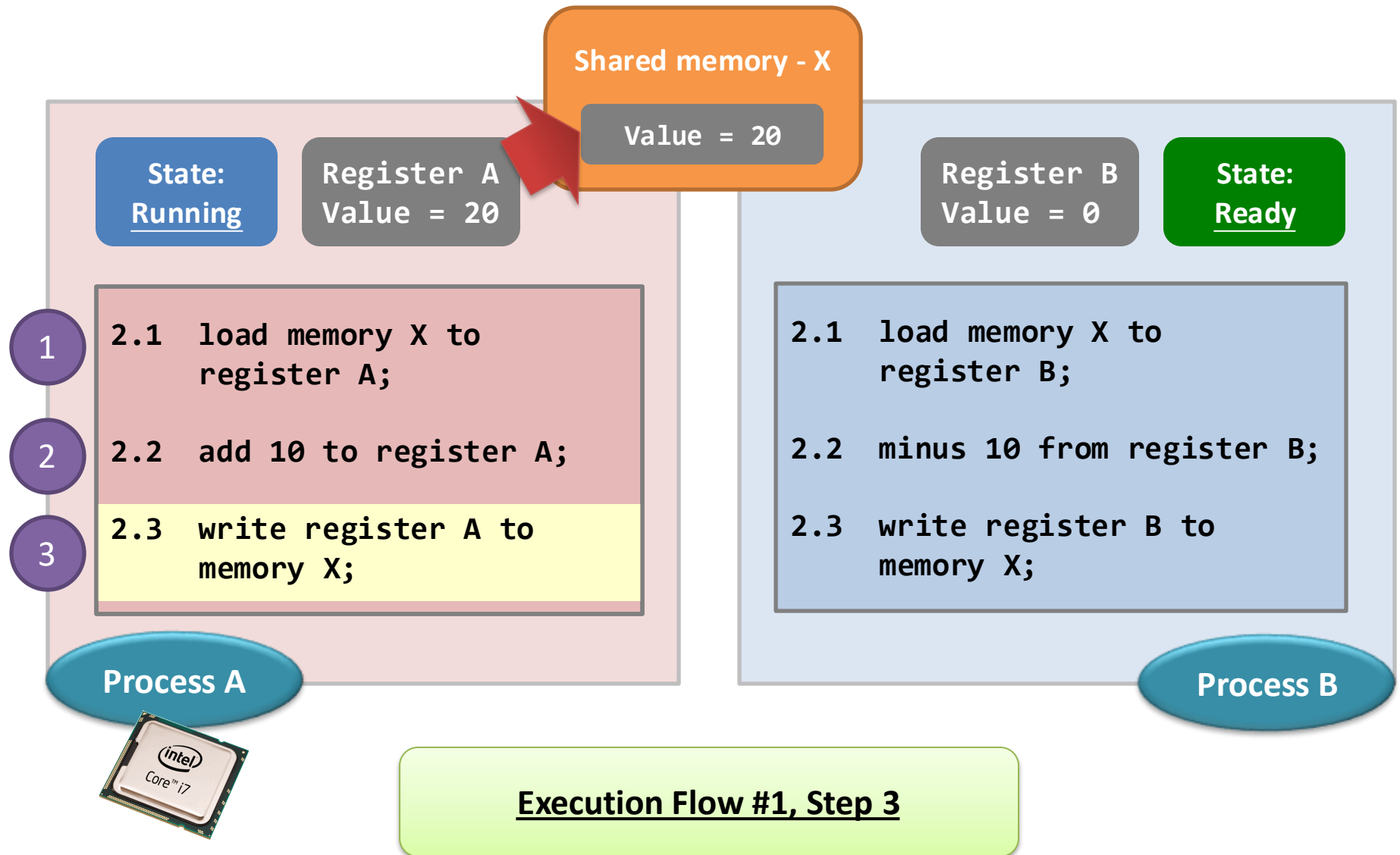
Problem not yet arise...

Don't print



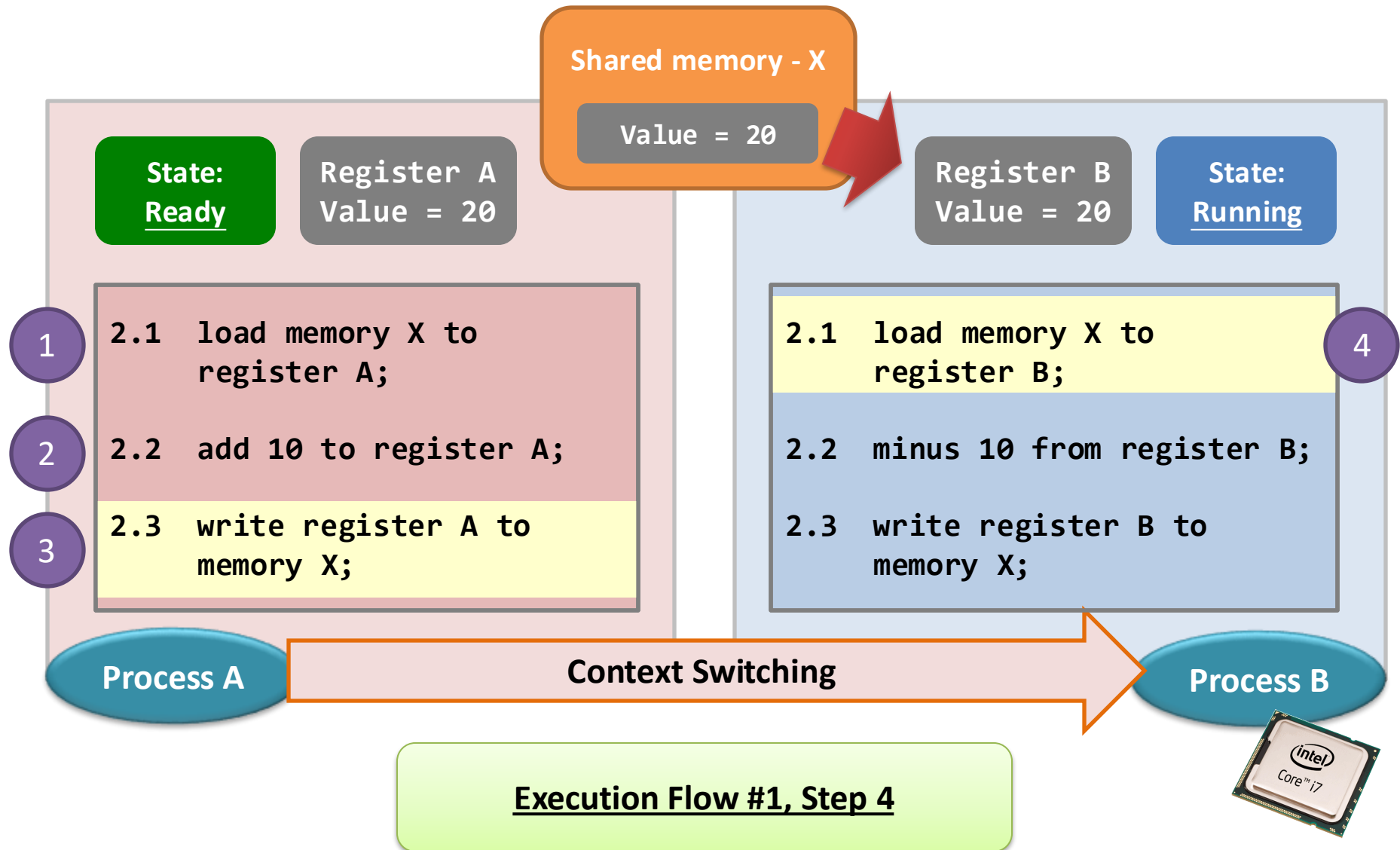
Problem not yet arise...

Don't print



Problem not yet arise...

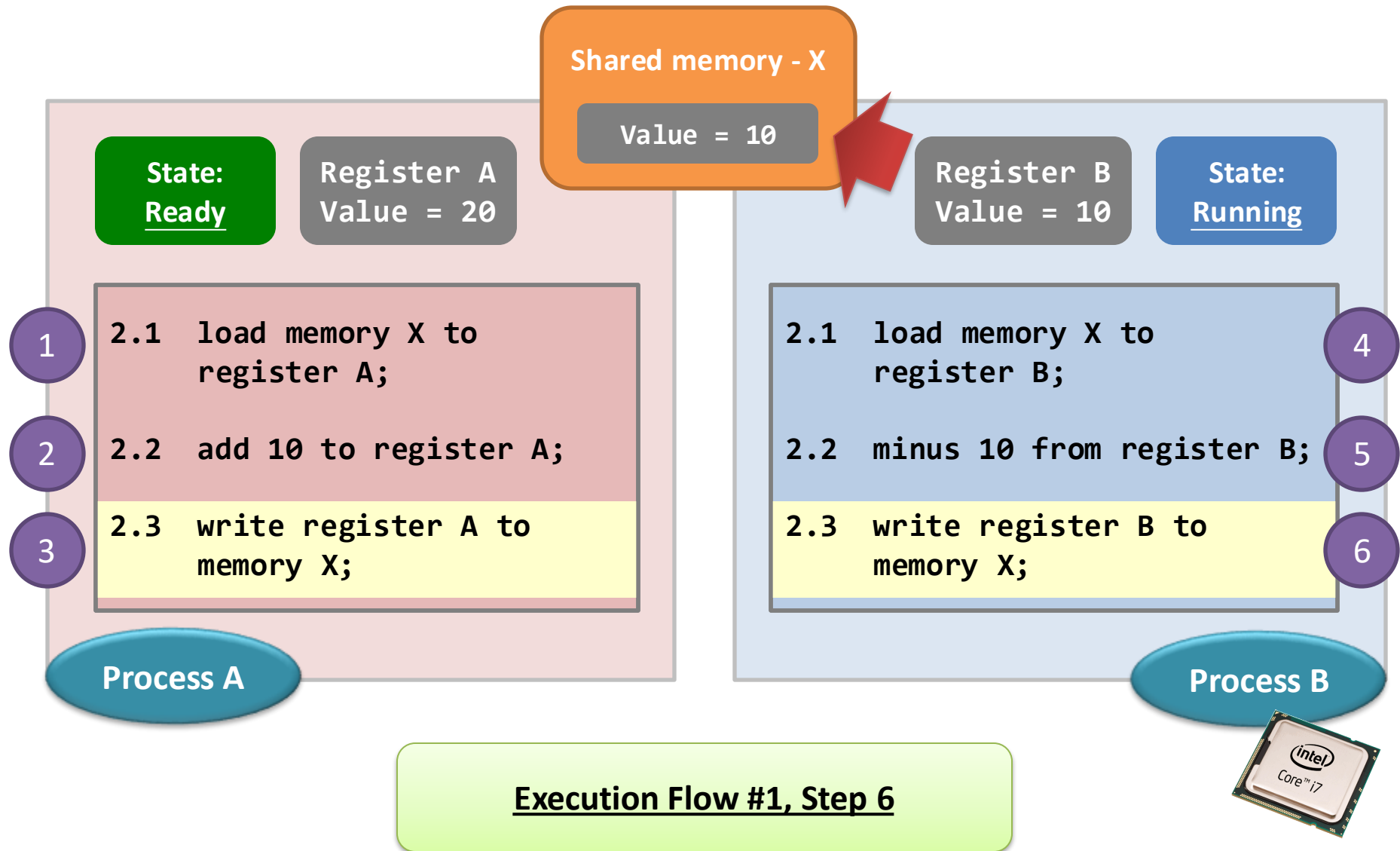
Don't print



Don't print

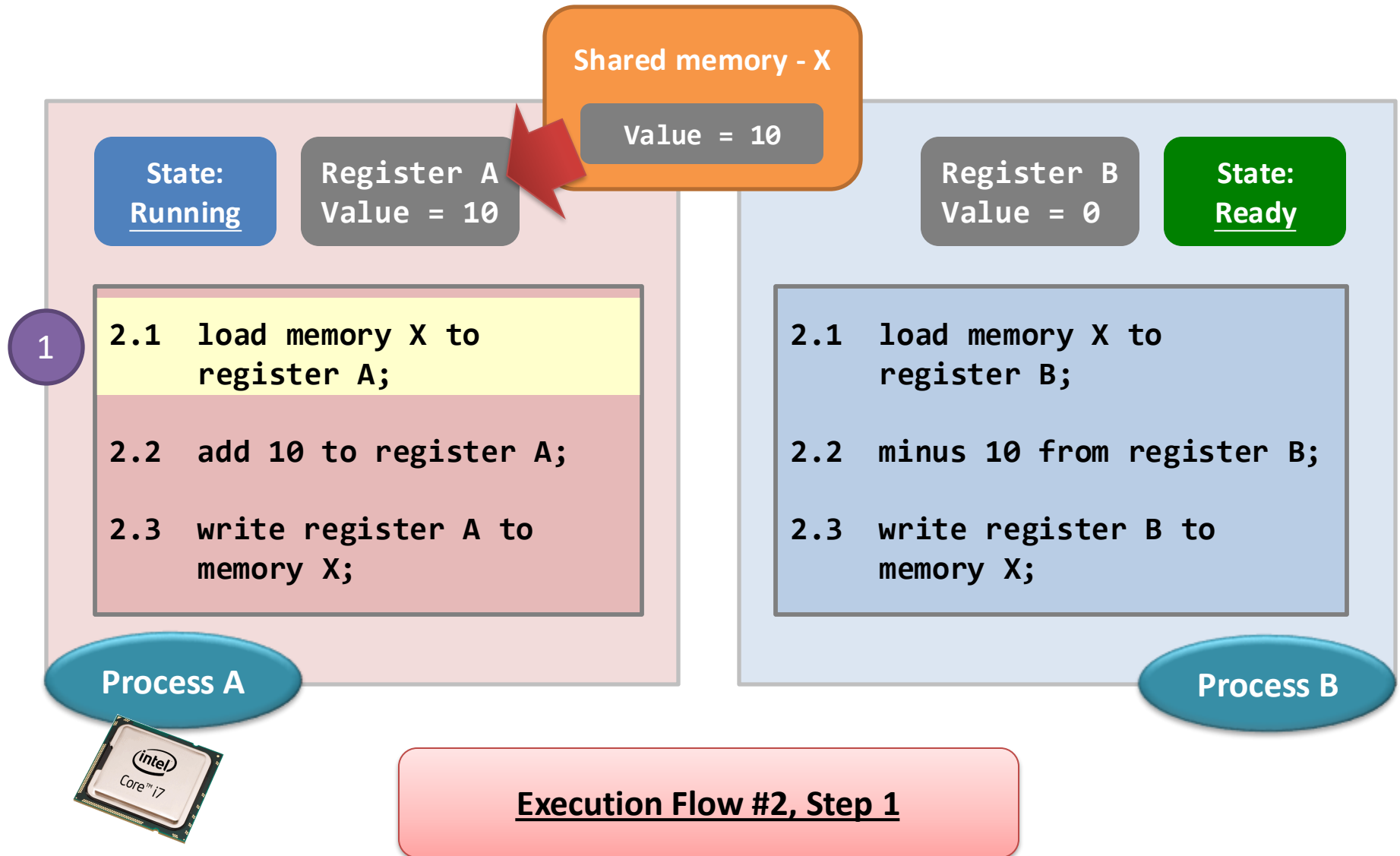


Problem not yet arise...



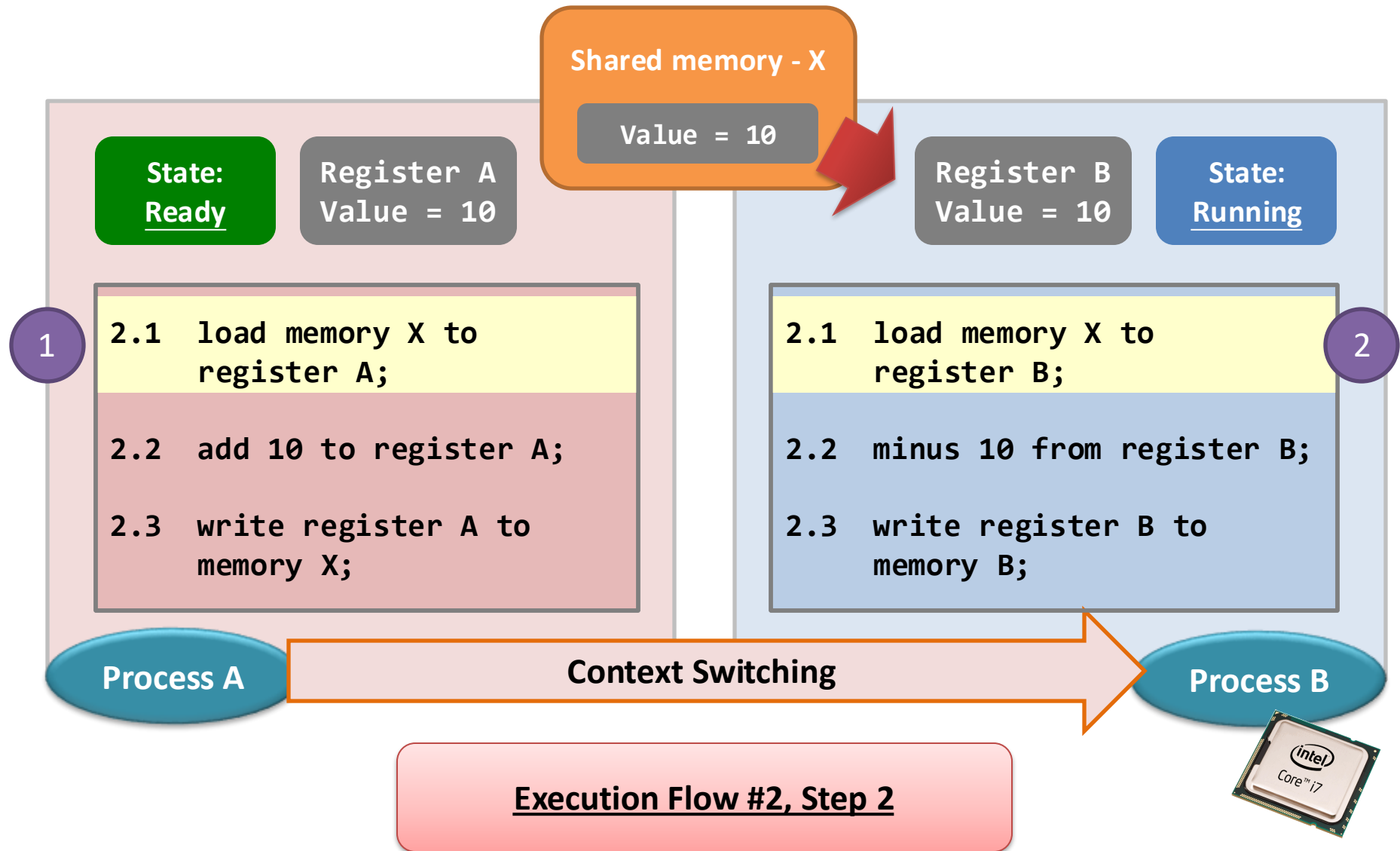
Problem arise...

Don't print



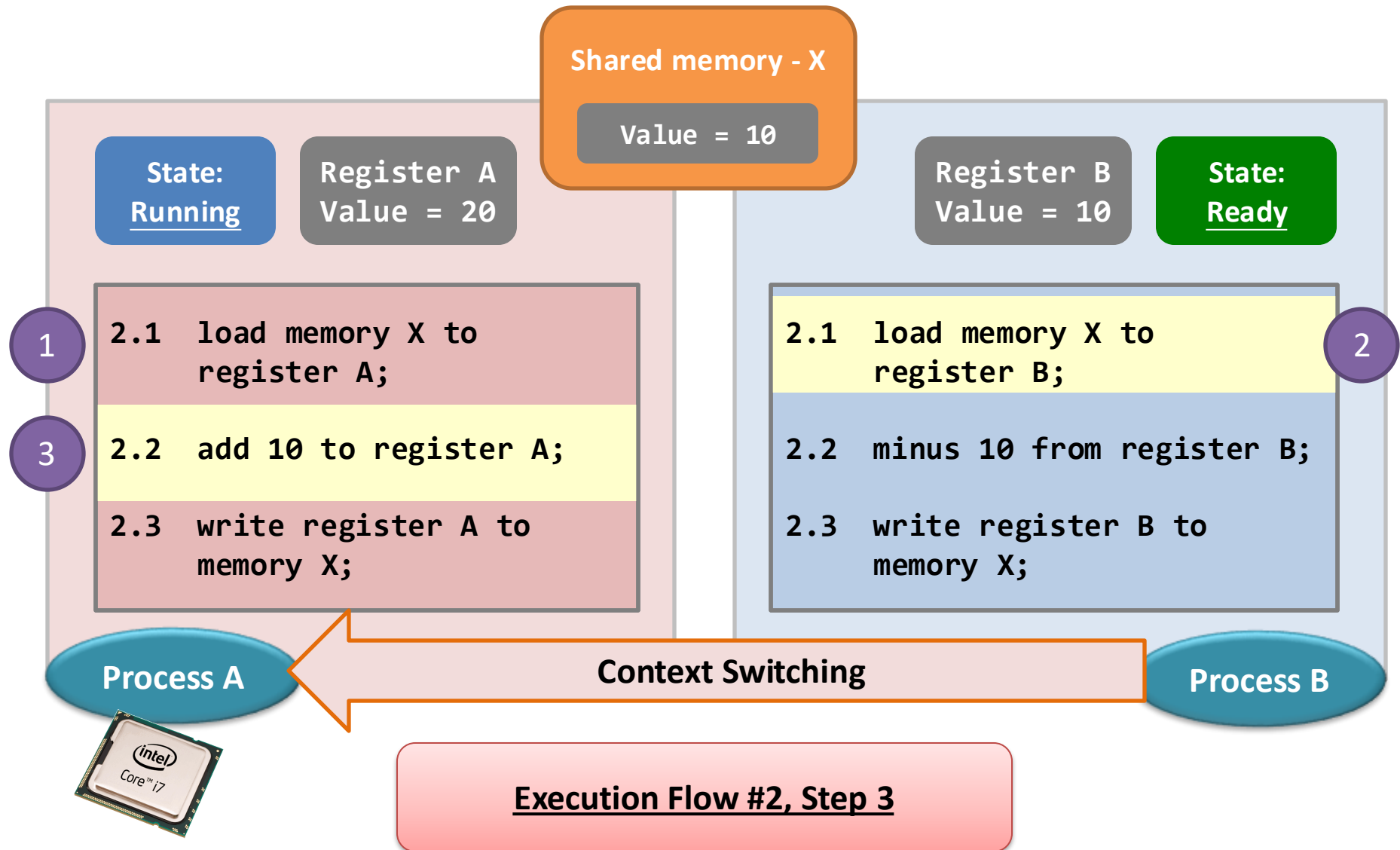
Problem arise...

Don't print



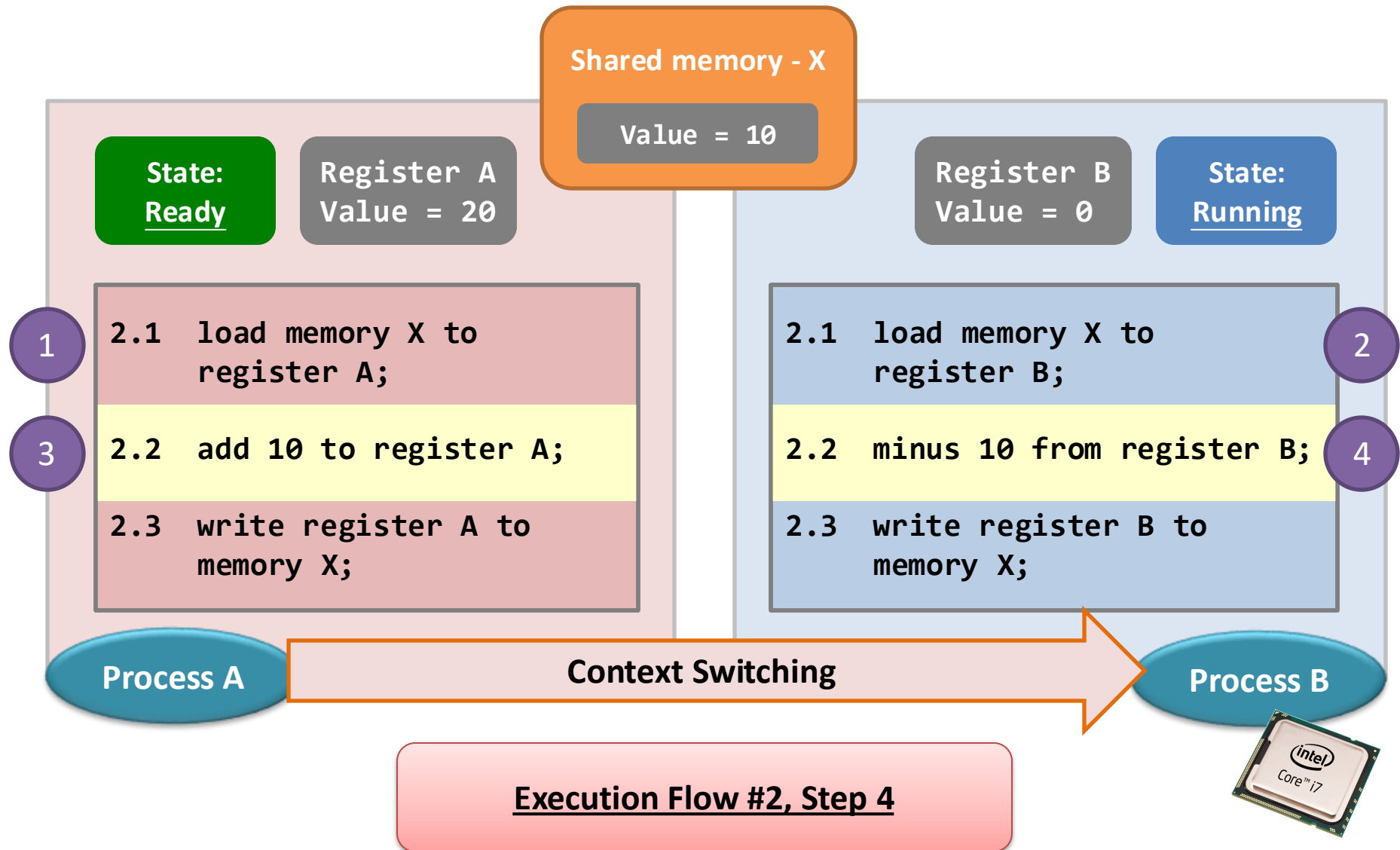
Problem arise...

Don't print

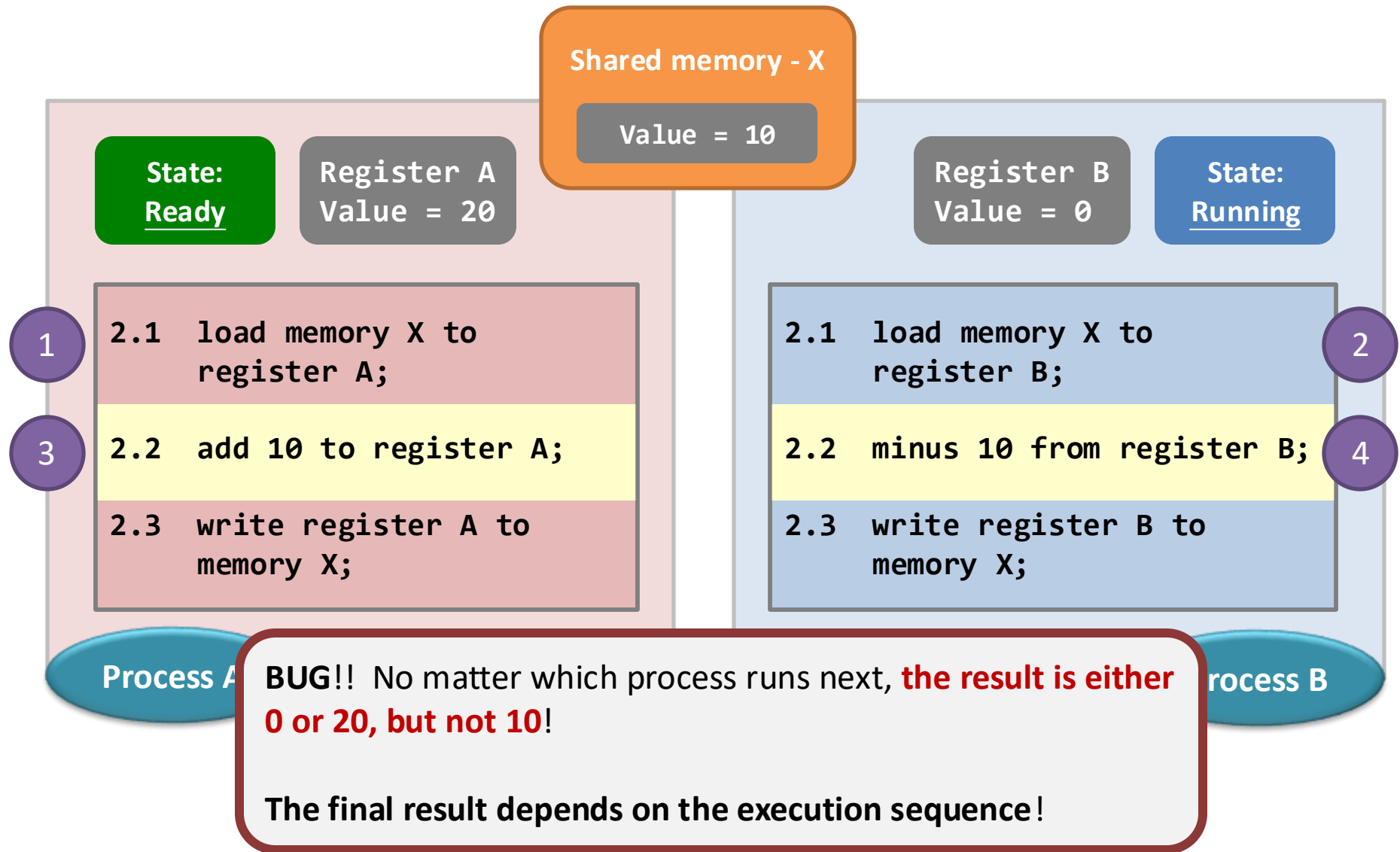


Problem arise...

Don't print



Problem arise...



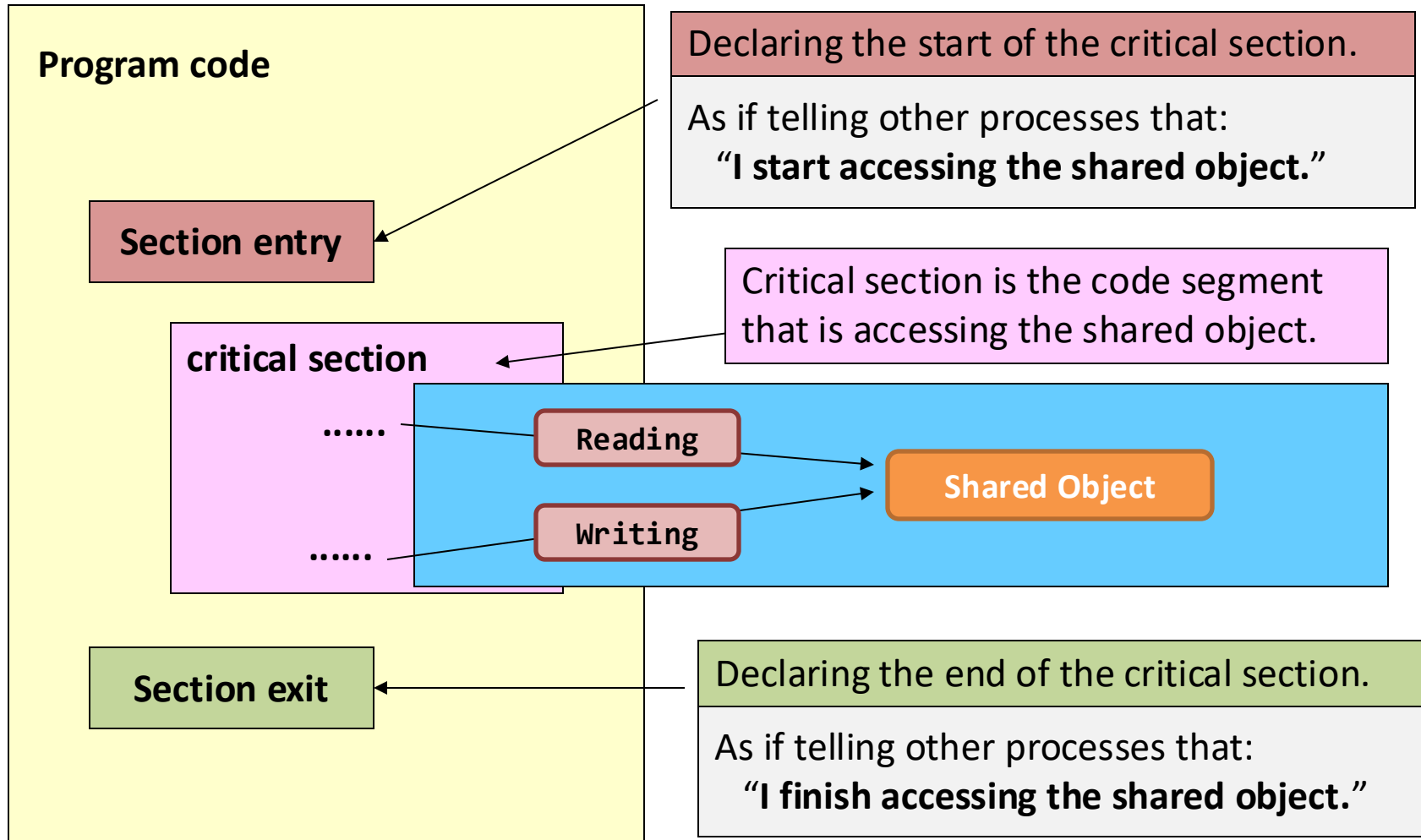
Data Race and race condition

- The above scenario is called the **data race**.
 - May happen whenever “**shared object**” + “**multiple updates**” + “**concurrently**”
- A **race condition** means
 - the outcome of an execution depends on a particular order
- Remember: data race is always a bad thing and debugging it is a **nightmare**!
 - It may end up ...
 - 99% of the executions are fine.
 - 1% of the executions are problematic.

A Classic Solution to Data Race

- Achieving Mutual exclusion
 - Shared object is still sharable, but
 - Not to access the “shared object” at the same time
 - Access the “shared object” one by one

Critical Section – the realization



What's really matter is the section entry/exit



Summary...for the content so far...

- **Data race**
 - happens when programs accessing a shared object
 - The outcome of the computation **totally depends on the execution sequences** of the processes involved.
- **Mutual exclusion**
 - If it could be achieved, then the problem of the race condition would be gone.

Summary...for the content so far...

- **A critical section** is the code segment that accesses 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 **access more than one shared objects**.

Summary...for the content so far...

- **Mutual exclusion hinders the performance of parallel computations.**

Solution to Data Race

- Locking (Classic)
 - Concept:
 - Only one can enter the CS at any point of time
 - = Mutual Exclusion
 - Synchronization Primitive
 - Spin-based lock
 - E.g., `pthread_spin_lock`
 - Sleep-based lock
 - E.g., POSIX semaphore, `pthread_mutex_lock`
- Lock-free (Will see it in Synchronization II)
 - Concept:
 - Everyone can enter their CS and do whatever they like
 - Check if the final result did suffer from race condition before leaving CS,
 - Those violated redo their CS and recheck again

Let's focus on Lock-
based Approach in
Synchronization I

Evaluation of a lock-based solution

- **Mutual Exclusion**

- Basic requirement
- No two processes could be simultaneously go inside their **own** critical sections.

- **Bounded Waiting**

- If a process wants to enter her CS but get to wait
 - E.g., stuck at function `waitForMyTurn()`
 - Such `waitForMyTurn()` will eventually return

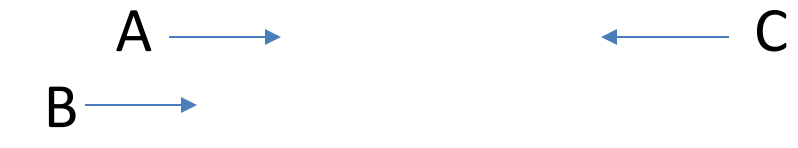
Evaluation of a lock-based solution

- **Progress**
 - Say no process currently in C.S.
 - One of the waiting processes can eventually get in



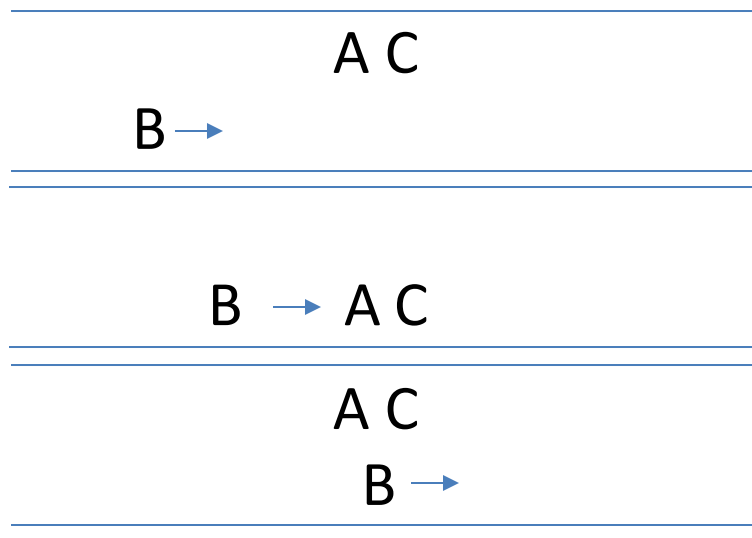
Progress vs. Bounded waiting

- 3 people: A, B, C on an aisle



– Consider a solution of:

- When I crash into someone, I will move to the other side of the aisle and try to continue

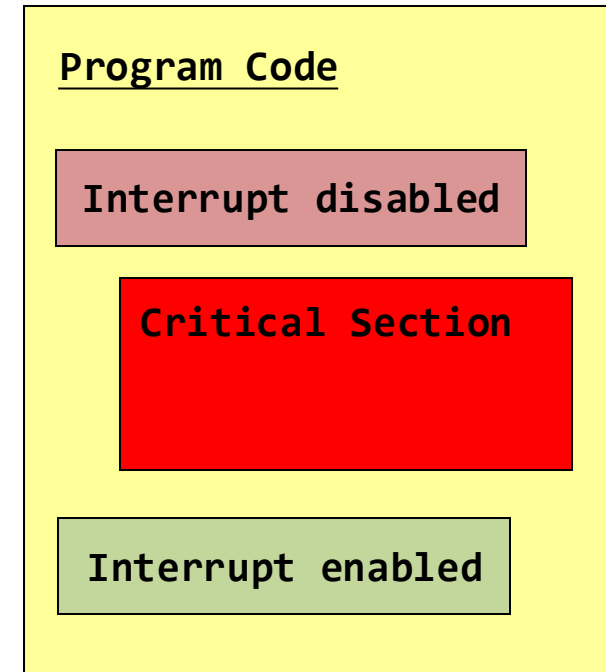


B has progress
B has bounded waiting
(B actually did not need to wait)

AC also have **bounded waiting**. Why?

A simple baseline to implement locking

- **Disabling interrupt** when the process is inside the critical section.
- **Effect**
 - When a process is in its critical section, no context switch
- User space (i.e., to normal user)
 - **Not permissible**
 - What if you write a CS that loops infinitely and the other process (e.g., the shell) never gets the context switch back to kill it?



Implementing a lock

- Locking by Disabling Interrupt
 - For kernel level for old CPUs but now
 - Multi-core complication
 - It is possible that another core modifying the shared object in the memory
- Modern multi-core CPU bundles with **atomic instructions** to make sure one core's modification is atomic
 - All subsequent discussions assume all instructions on shared memory are atomic
 - E.g., `x++` is atomic
 - Details of atomic instruction will be covered in topic "Synchronization II"

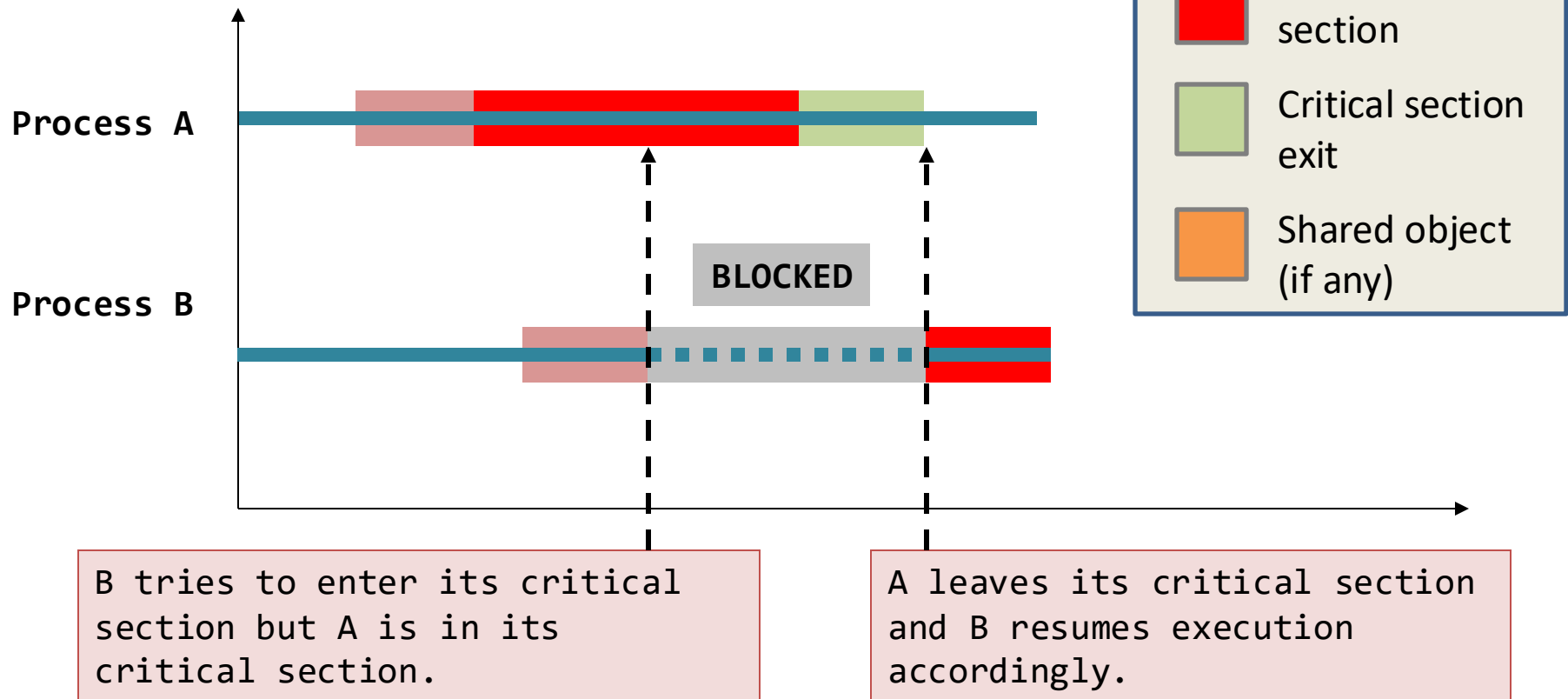
Spin-based Lock

- Implemented by spinning on some **extra shared variables**
- Concept:

```
int shared_x; //shared-variable
int shared_lock; //extra shared variable to be the "lock"
while (shared_lock=0){ //spin on the lock
}
//enter the CS
shared_x++; //atomic instruction
```

A typical mutual exclusion scenario

Remember, it is always the entry blocks other processes, but not the critical section.



Spin-lock

- Basic spinlock
 - Spin on **yet another shared variable**, **turn**, to detect the status of other processes

Extra shared variable "turn" initial value = 0

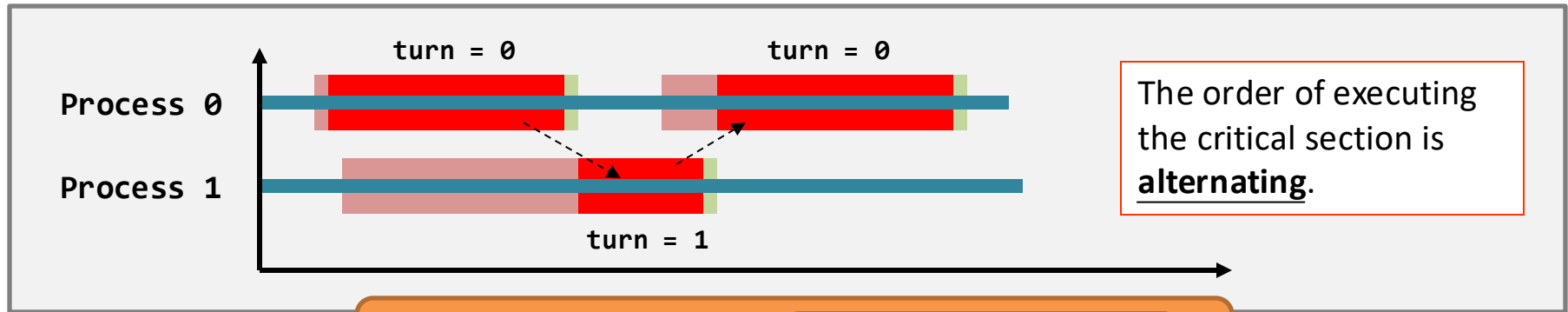
```
1 while (TRUE) {  
2   while( turn != 0 )  
3     ; /* busy waiting */  
4   critical_section  
5   turn = 1;  
6   remainder_section();  
7 }
```

Process 0

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

Process1

Basic Spinlock



Extra shared variable "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

Basic Spinlock

- You can wrap them as lock() and unlock() functions

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

```
1 while (TRUE) {  
2   lock();  
4   critical_section  
5   unlock();  
6   remainder_section();  
7 }
```

Basic Spinlock

- Correct
 - Busy waiting wastes CPU resources
 - But very OK for short critical section
 - Especially these days we have multi-core
 - Won't block other **irrelevant** processes a lot (they can run on other cores)
- But this simple implementation imposes 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

This simple busy waiting implementation violates progress

- 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()

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

Process 0

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

Process 1

Peterson's solution to implement a spin-lock

- Highlight:
 - Use one more extra shared variable: **interested**
 - If I don't show interest
 - I let you go
 - If we both show interest
 - Take turns

Shared variables:

- turn &
- "interested[2]"

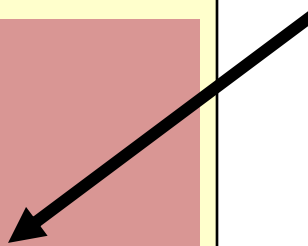
Peterson's solution to implement a spin-lock

```
1  bool turn;                                /* who's last enter cs */
2  bool interested[2] = {FALSE,FALSE}; /* express interest to enter cs*/
3
4  void lock( int process ) { /* process is 0 or 1 */
5      int other;
6      other = 1-process;          /* other is 1 or 0 */
7      interested[process] = TRUE; /* express interest */
8      turn = process;
9      while ( turn == process &&
              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 to implement a spin-lock

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

Express interest to enter CS

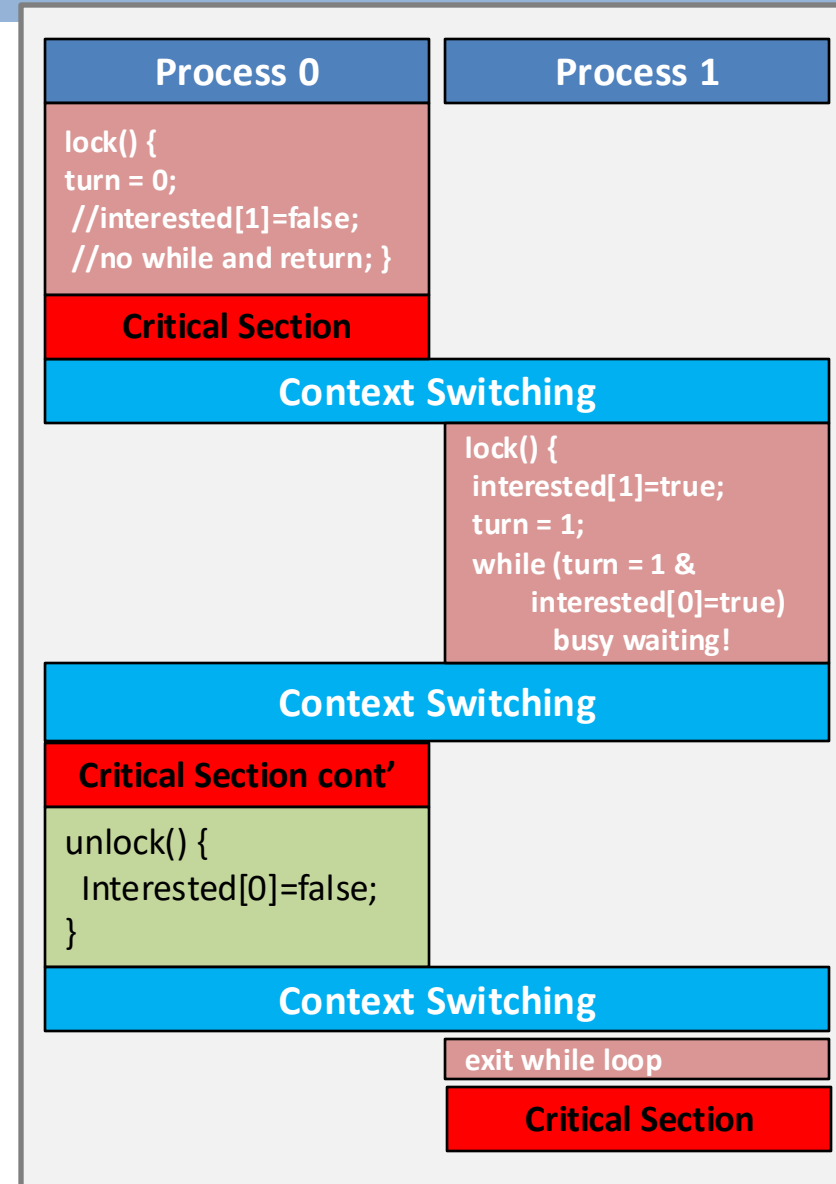


If others not show interest, I
can always go ahead



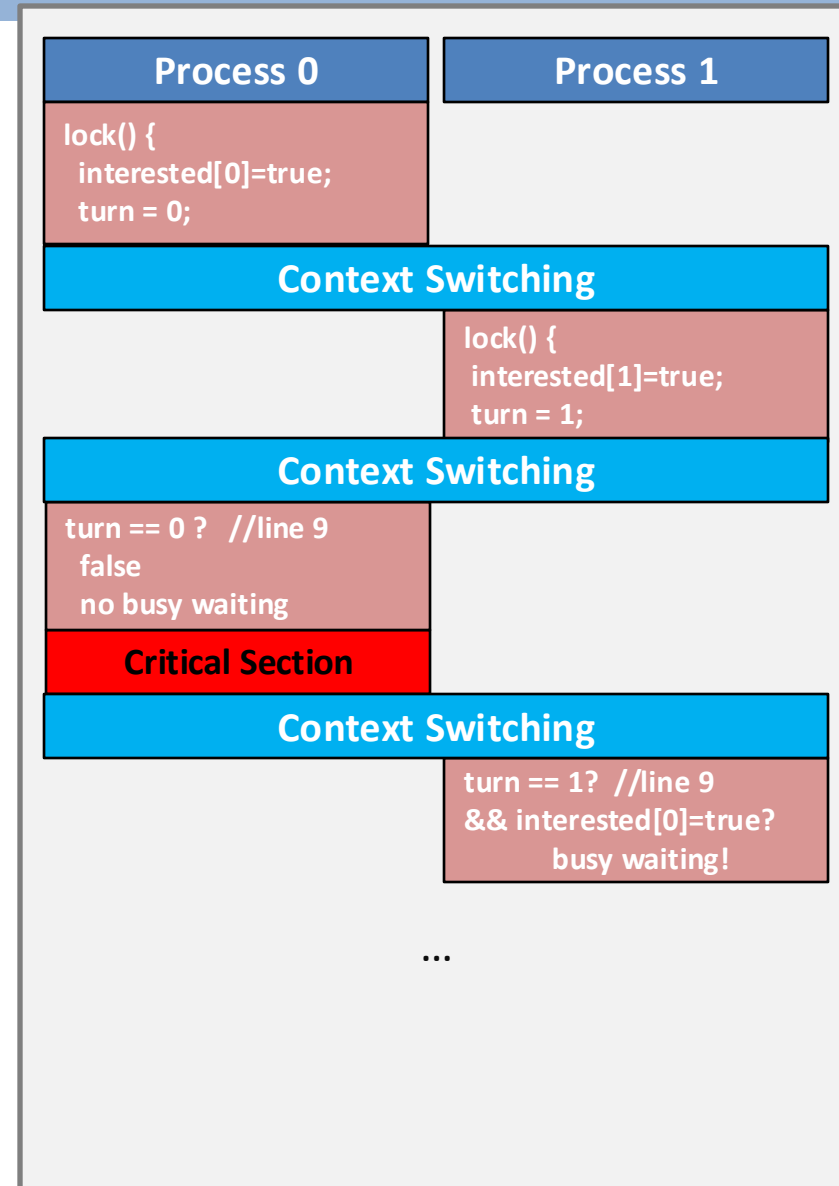
Peterson's solution to implement a spin-lock

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



Another case

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



Peterson's solution to implement a spin-lock

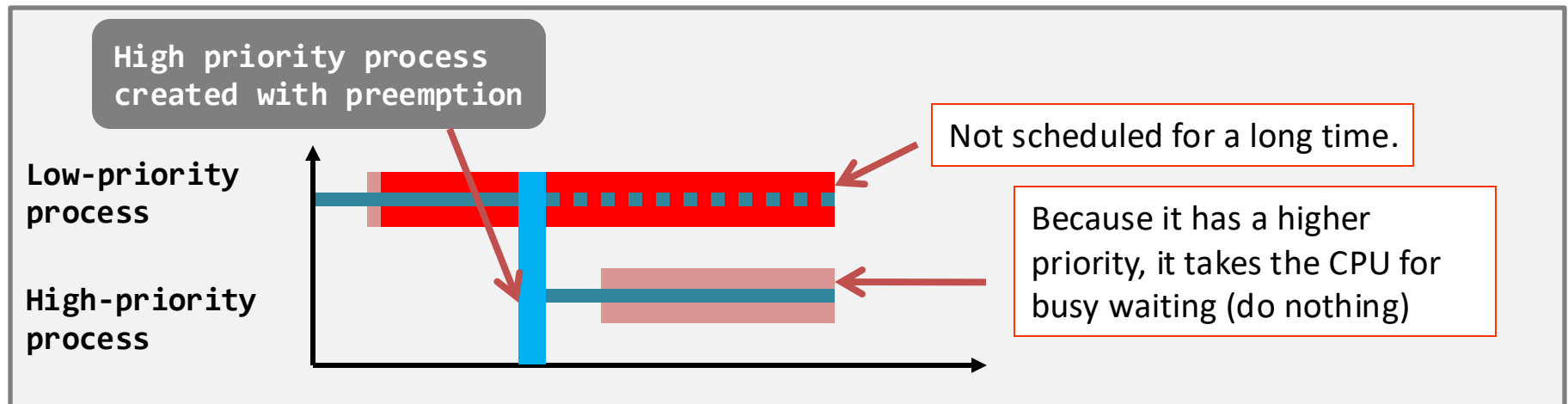
- = Busy waiting
 - + shared variable **turn** for mutual exclusion
 - + shared variable **interest** to resolve strict alternation
- Wikipedia:
 - *“It satisfies the three essential criteria to solve the critical section problem, provided that changes to the variables turn, interest[0], and interest[1] propagate immediately and atomically.”*
- However, suffer from **priority inversion problem**

Does it work for >2 processes?

https://en.wikipedia.org/wiki/Peterson's_algorithm

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



Lock Implementation

- Locking by Spinning on some other shared variables
 - Spinlock
 - Basic Spinlock implementation
 - Who is the extra shared variable? turn
 - Violate progress
 - Peterson's implementation
 - Who are the extra shared variables? turn + interest
 - Suffer from priority inversion
 - pthread_spin_lock is a very efficient implementation
 - Will give you some details in Topic "Synchronization II"
 - Ok for short critical section
 - But what if the process A who gets the lock has a long-running critical section?
 - » The other waiting process B round robin and gets the CPU but just spinning during its entire allocated CPU quantum
 - Sleeplock
 - If a process needs to wait for a process who locks on a long CS, it's better to go to sleep and wake me up when it's my turn
 - Give the CPU to other processes who need the CPU but no access to any shared resources

Sleep-based lock: Semaphore

- Classic
- Semaphore is an extra shared ~~variable~~ struct
 - Include
 - ~~a boolean~~ an integer that counts the # of resources available
 - Can do more than solving mutual exclusion
 - a wait-list



Semaphore **logical** view

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

Section Entry: sem-wait()

```
1 void sem-wait(semaphore *s){  
2     disable_interrupt();  
3     *s = *s - 1; //s.v--  
4     if ( *s < 0 ) {  
5         enable_interrupt();  
6         sleep(s); //add2waitlist  
7         disable_interrupt();  
8     }  
9     enable_interrupt();  
10 }
```

"sem-wait(s)"

- I wait until I get an s
(i.e., only returns when I get an s)
- Implementation:
of **s.v--**;
sleep if # of **s** < 0;

Important 1
s can be a plural

Important 2

A bit similar to wait(child). But that was restrictive, only notified under parent-child state change. Vs 2 independent processes and user designated wake up point

"sem-post(s)"

- I notify the others that one **s** is added
- Implementation:
of **s.v++**;
If someone is waiting **s**, wakeup one of them

Section Exit: sem-post()

```
1 void sem-post(semaphore *s){  
2     disable_interrupt();  
3     *s = *s + 1; //s.v++  
4     if ( *s <= 0 )  
5         wakeup(s);  
6     enable_interrupt();  
7 }
```

Example

Process **1234**

sem-
wait(X)

Section Entry: sem-wait()

```
1 void sem-wait(semaphore *s) {  
2     disable_interrupt();  
3     *s = *s - 1;  
4     if ( *s < 0 ) {  
5         enable_interrupt():  
6         sleep(s);  
7         disable_interrupt();  
8     }  
9     enable_interrupt();  
10 }
```

Assuming someone else (process **1357**) has
already taken the only one resource:

$X = 1$ (initial) $\Rightarrow X = 0$

Now, process **1234** arrives

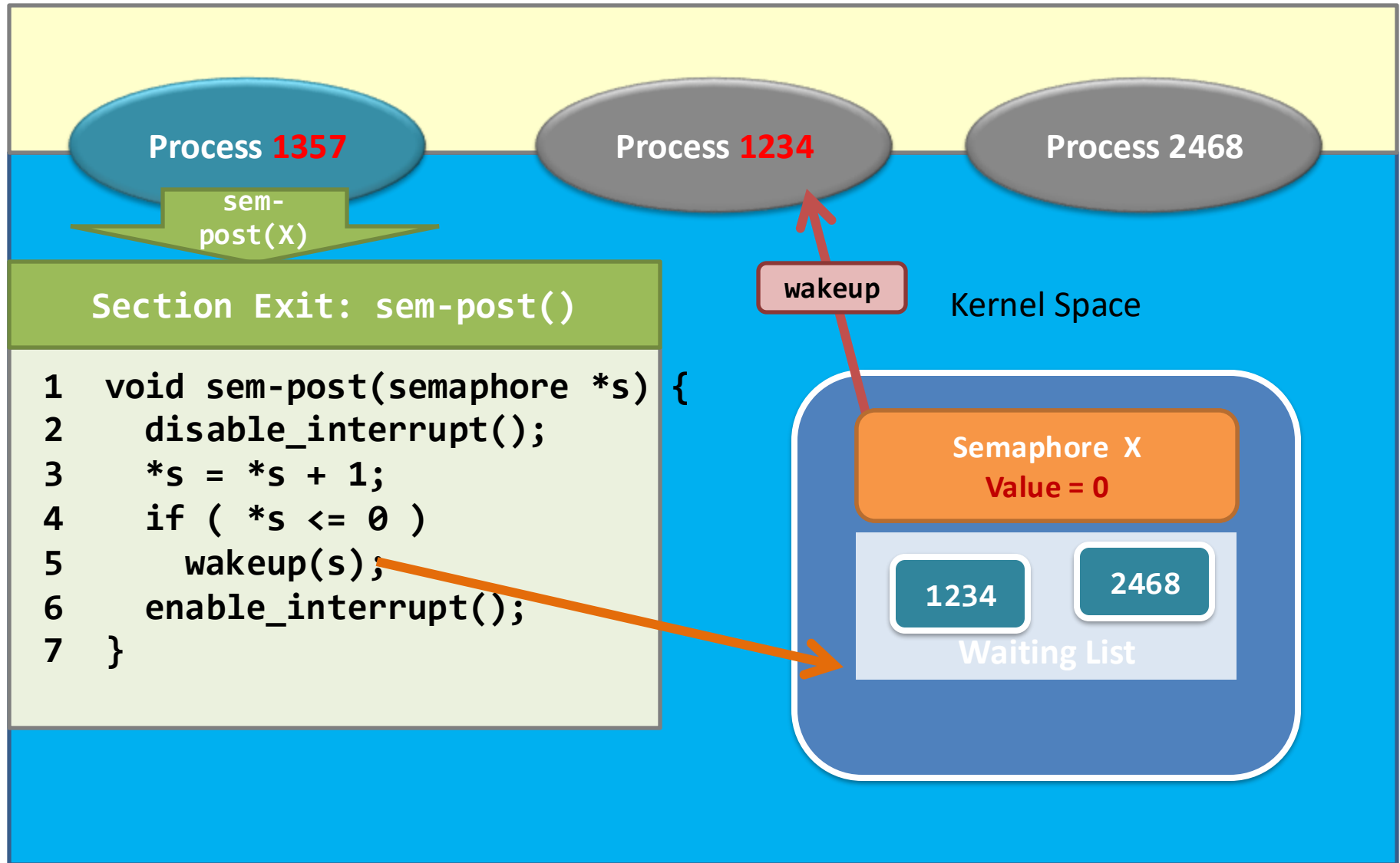
Kernel Space

Semaphore X
Value = -1

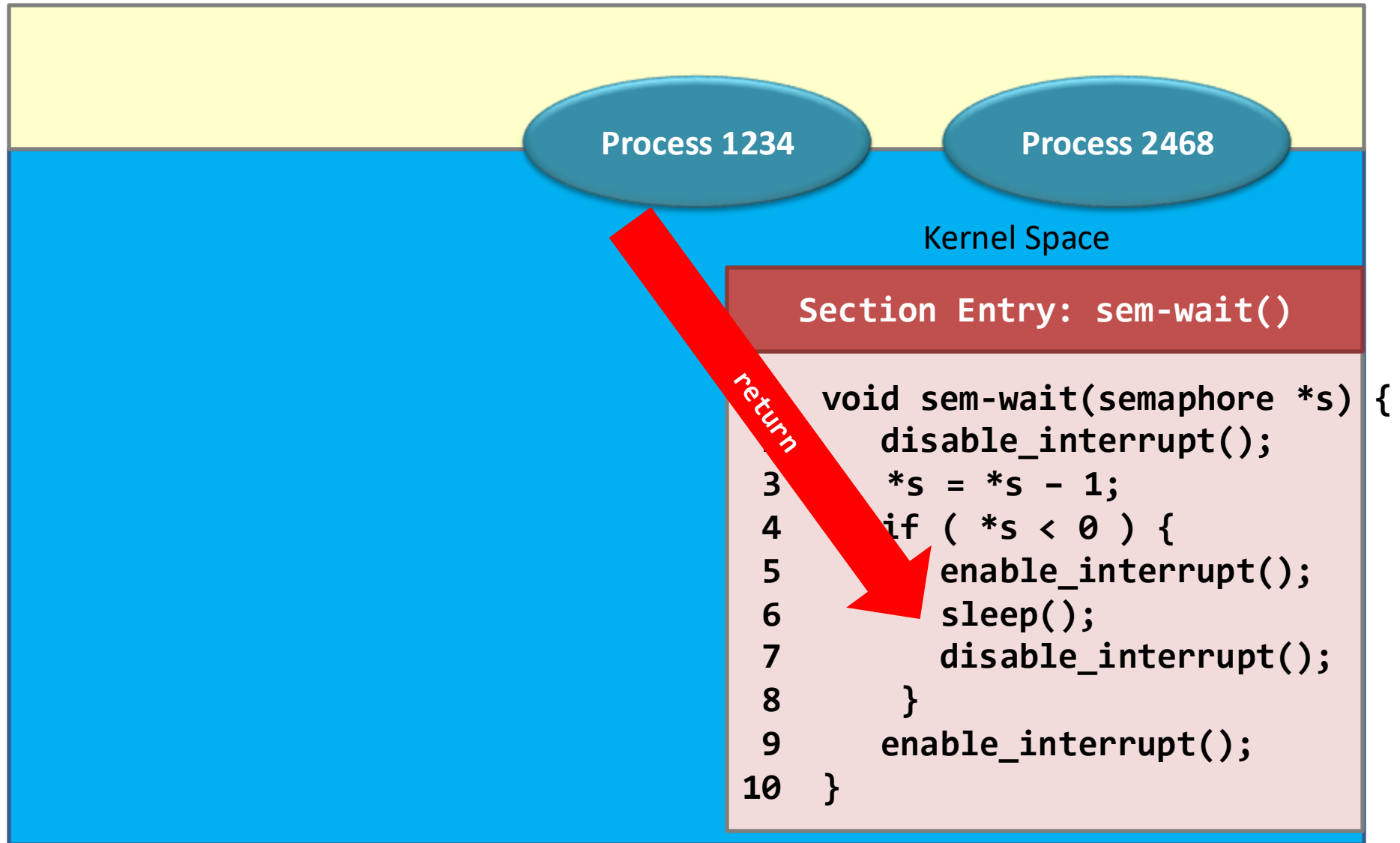
1234

Waiting List

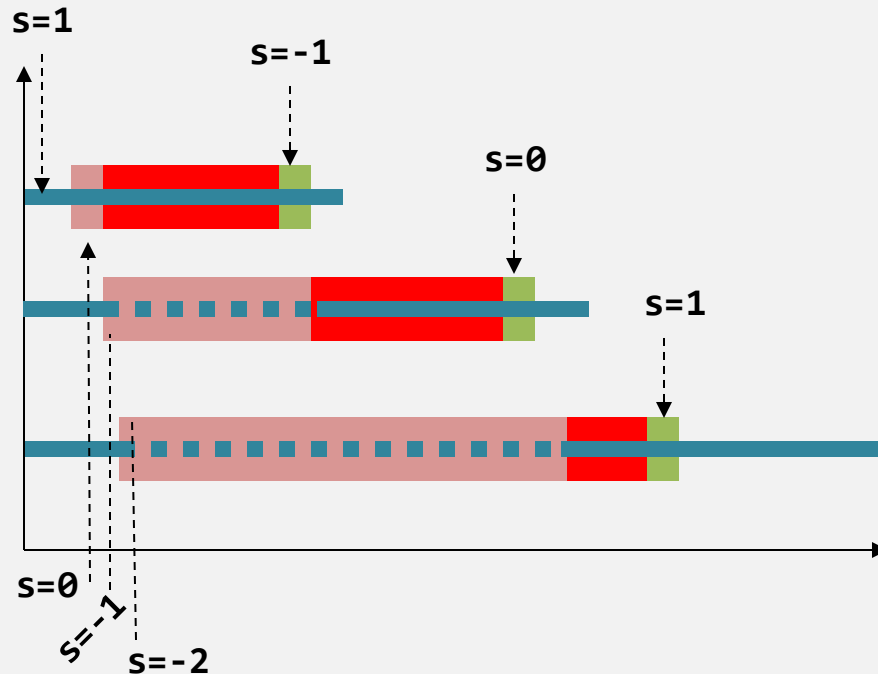
Example



Example



Using Semaphore (user-level)



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

```
1 while(TRUE) {  
2     sem-wait(s);  
3     critical_section();  
4     sem-post(s);  
5 }
```

entry

exit

Why it is named as “Semaphore”?

- $X > 1$ (Counting Semaphore)
 - It can do things beyond locking → coordination
- $X=1$ (Binary Semaphore)
 - Serve as a sleep-based lock → mutual exclusion



Which one is the shared object in this picture?

Achieving Mutual Exclusion is not enough

E.g., My solution:

always let EW to go

→ it satisfies mutual exclusion

→ but NS has starvation

→ We need coordination!

Solution to Race Condition - Summary

- Locking
 - Spin-based lock
 - E.g., use of “pthread_spin_lock”
 - Possible Implementations:
 - » **Basic spinning using 1 shared variable**
 - » **Spin using 2 shared variables + good algorithm (=Peterson’s solution)**
 - » Spin using atomic instructions + smart algorithm (=Ticket, MCS algorithm, etc.)
 - Sleep-based lock
 - E.g., **POSIX semaphore**, pthread_mutex_lock
 - Possible Implementations:
 - » yield(), atomic instructions + smart algorithm
 - Spin-based vs Sleep-based
 - Overhead of Sleep-based: yield() -> context switch
 - Overhead of Spin-based: spin CPU for nothing
 - Lock-free

IPC / Synchronization problems

	Properties	Examples
Producer-Consumer Problem	Two classes of processes: <u>producer</u> and <u>consumer</u> ; At least one producer and one consumer. [Single-Object Synchronization]	Pipe, Named Pipe
Dining Philosopher Problem	They are all running the same program; At least two processes. [Multi-Object Synchronization]	Cross-road traffic control
Reader Writer Problem	Multiple reads, 1 write	...
...		

Named Pipe (a.k.a. FIFO in Linux)

- Like Shared File (so **multiple** producers and consumers; unlike pipe)
- Like Pipe (unidirectional)
- In-memory (unlike file)
- Use like pipe (restrictive interface; unlike shared file/memory)

Inter-process communication (IPC)

- Classic IPC problems.
- Producer-consumer problem.

In the following, we demonstrate how to use semaphore to solve the producer-consumer problem.

Semaphore is not the only primitive to solve IPC problems. You might also use

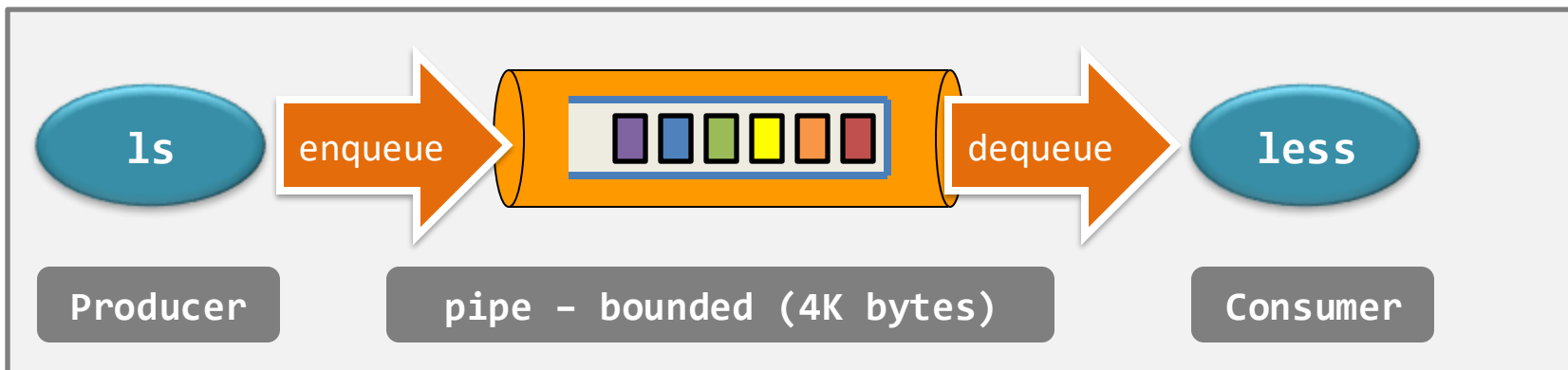
- (i) spinlock + condition variable,
- (ii) directly use 10 more shared variables + atomic instructions,
- (iii) two trylock + ...

...

Producer-consumer problem – introduction

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

A bounded buffer	<ul style="list-style-type: none">-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	<ul style="list-style-type: none">-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	<ul style="list-style-type: none">-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...**

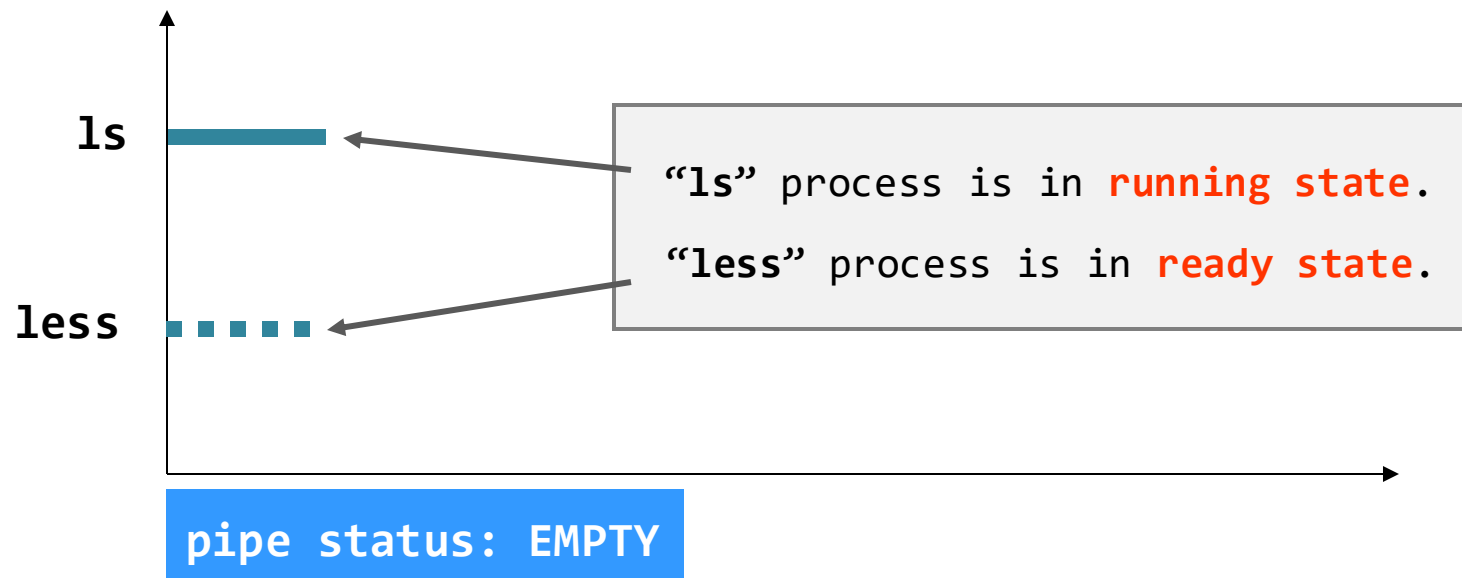
Then, **the consumer should wait.**

The producer should notify the consumer after she has enqueued an item.

Producer-consumer problem – pipe

Assumptions

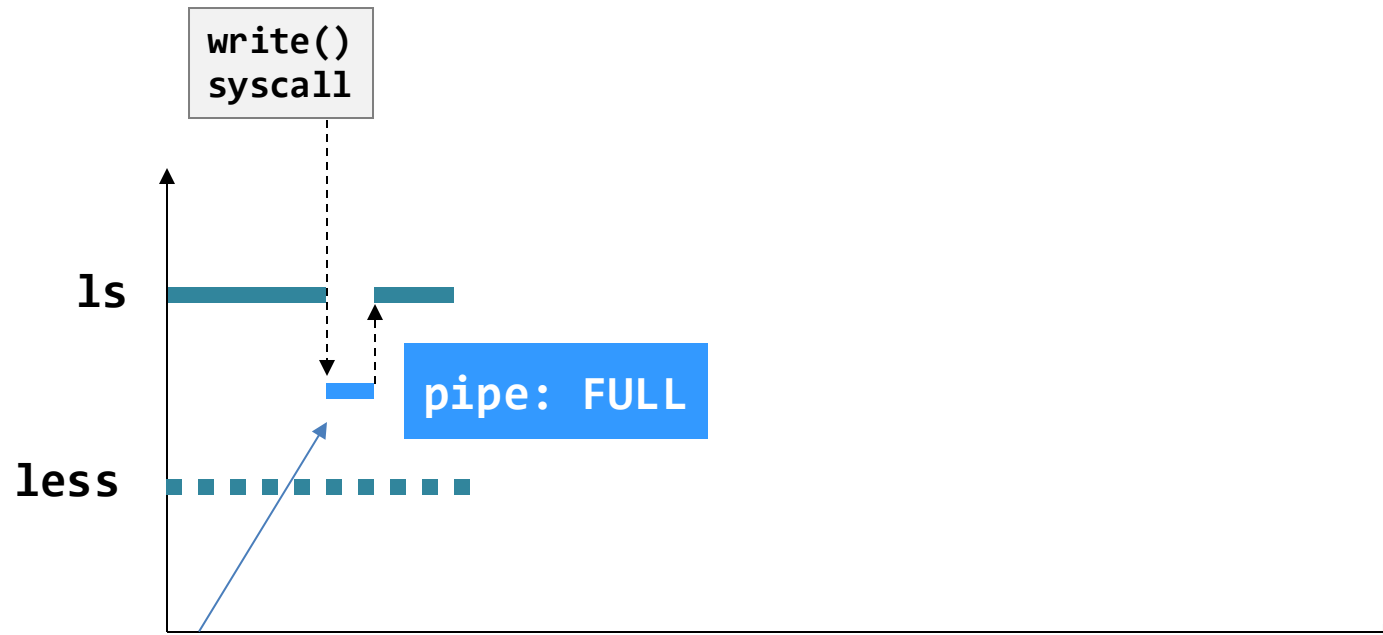
- The pipe is a queue of **1 byte** only!
- Each `write()` system call writes **1 byte** to the pipe.
- Each `read()` system call reads **1 byte** from the pipe.



Producer-consumer problem – pipe

Assumptions

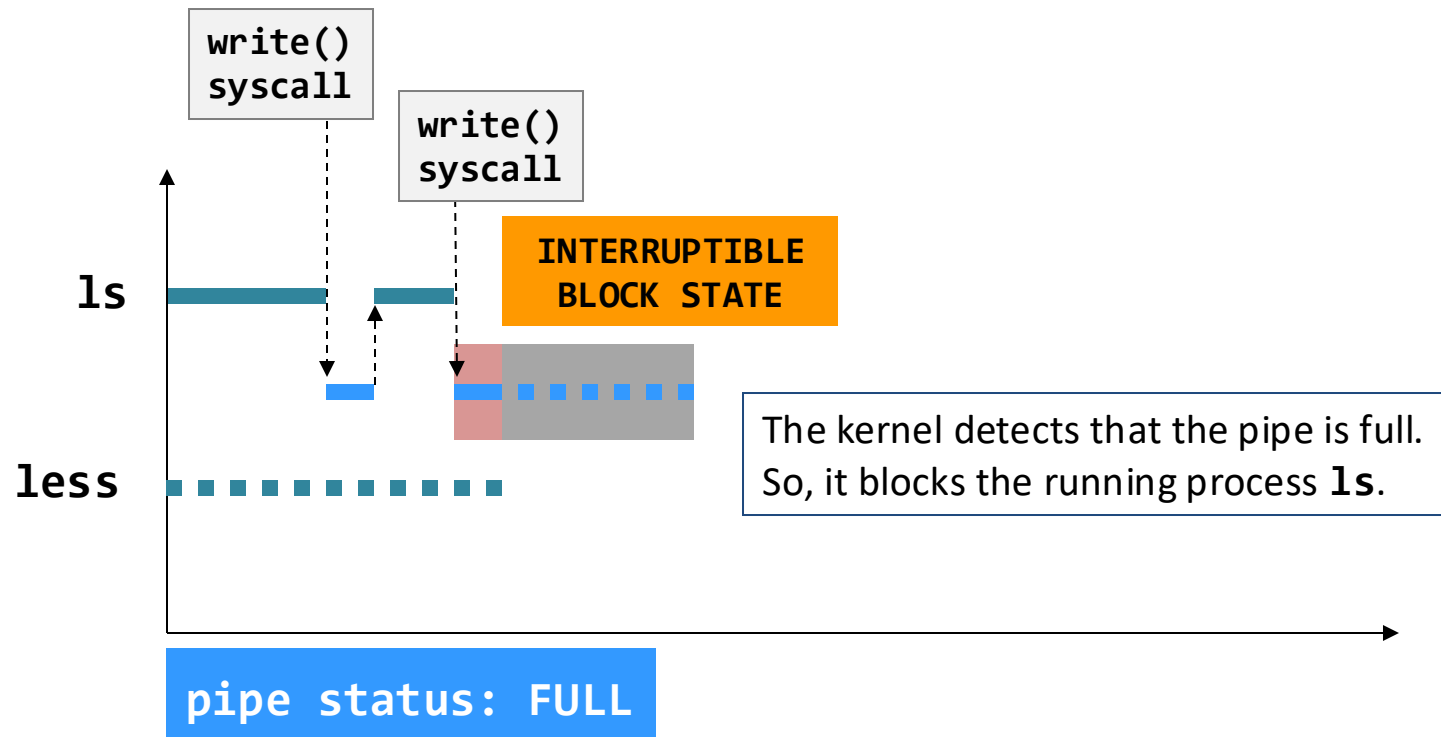
- The pipe is a queue of **1 byte** only!
- Each **write()** system call writes **1 byte** to the pipe.
- Each **read()** system call reads **1 byte** from the pipe.



Producer-consumer problem – pipe

Assumptions

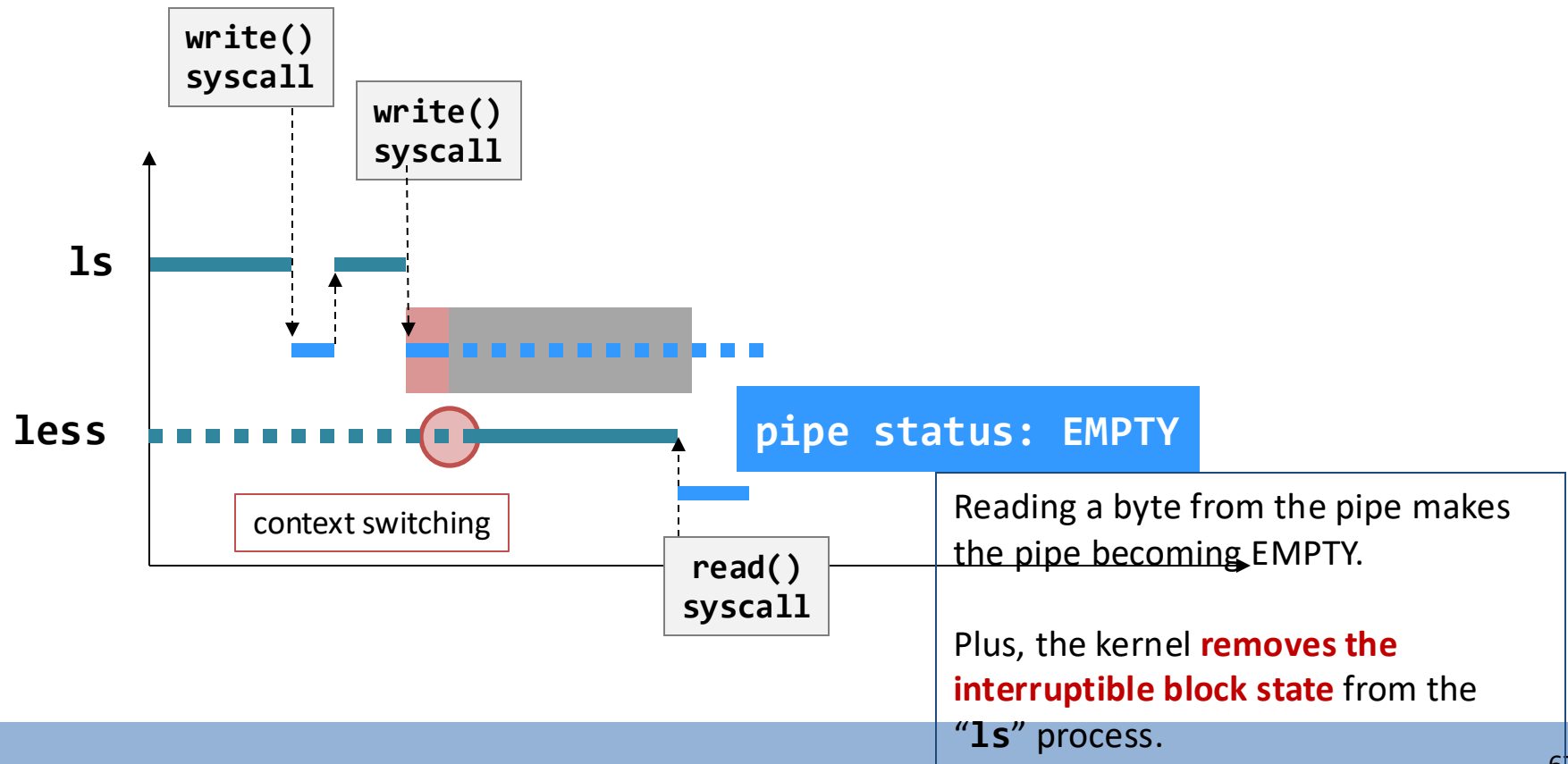
- The pipe is a queue of **1 byte** only!
- Each **write()** system call writes **1 byte** to the pipe.
- Each **read()** system call reads **1 byte** from the pipe.



Producer-consumer problem – pipe

Assumptions

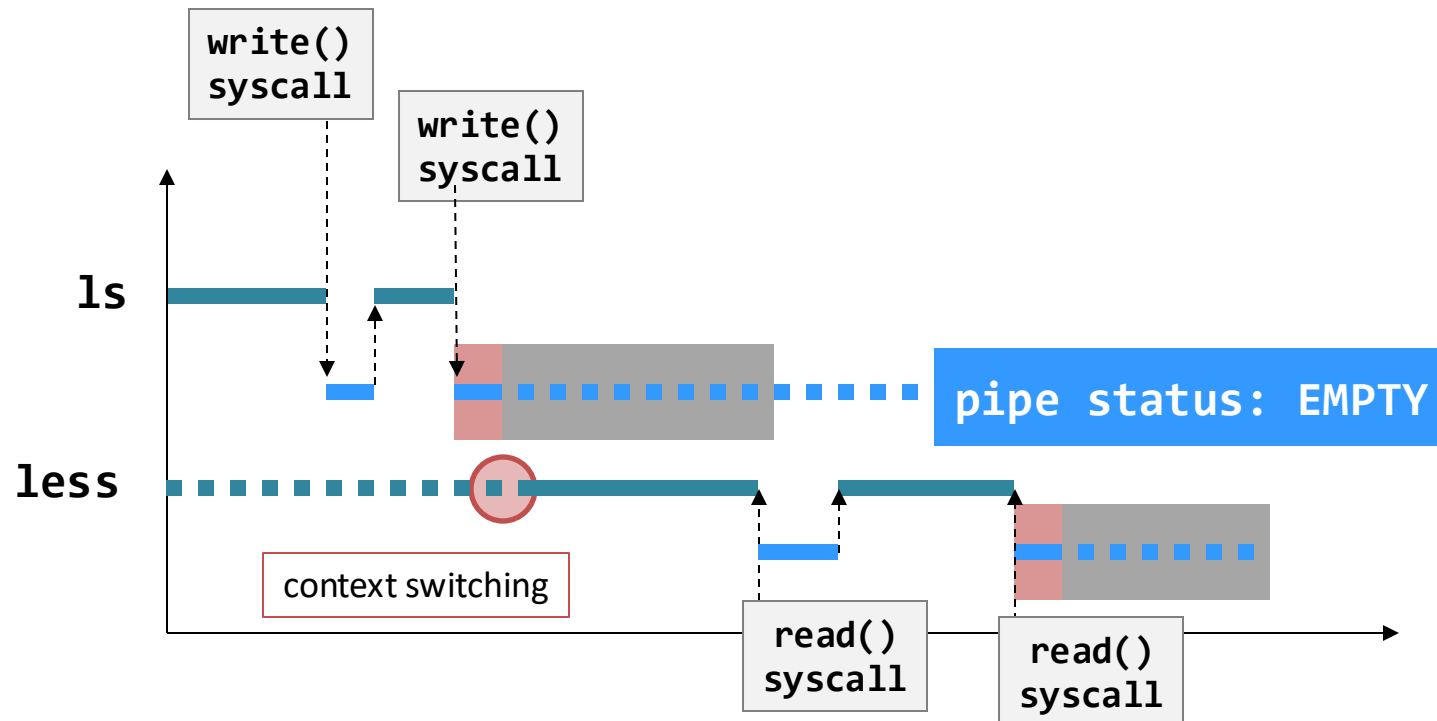
- The pipe is a queue of **1 byte** only!
- Each `write()` system call writes **1 byte** to the pipe.
- Each `read()` system call reads **1 byte** from the pipe.



Producer-consumer problem – pipe

Assumptions

- The pipe is a queue of **1 byte** only!
- Each `write()` system call writes **1 byte** to the pipe.
- Each `read()` system call reads **1 byte** from the pipe.



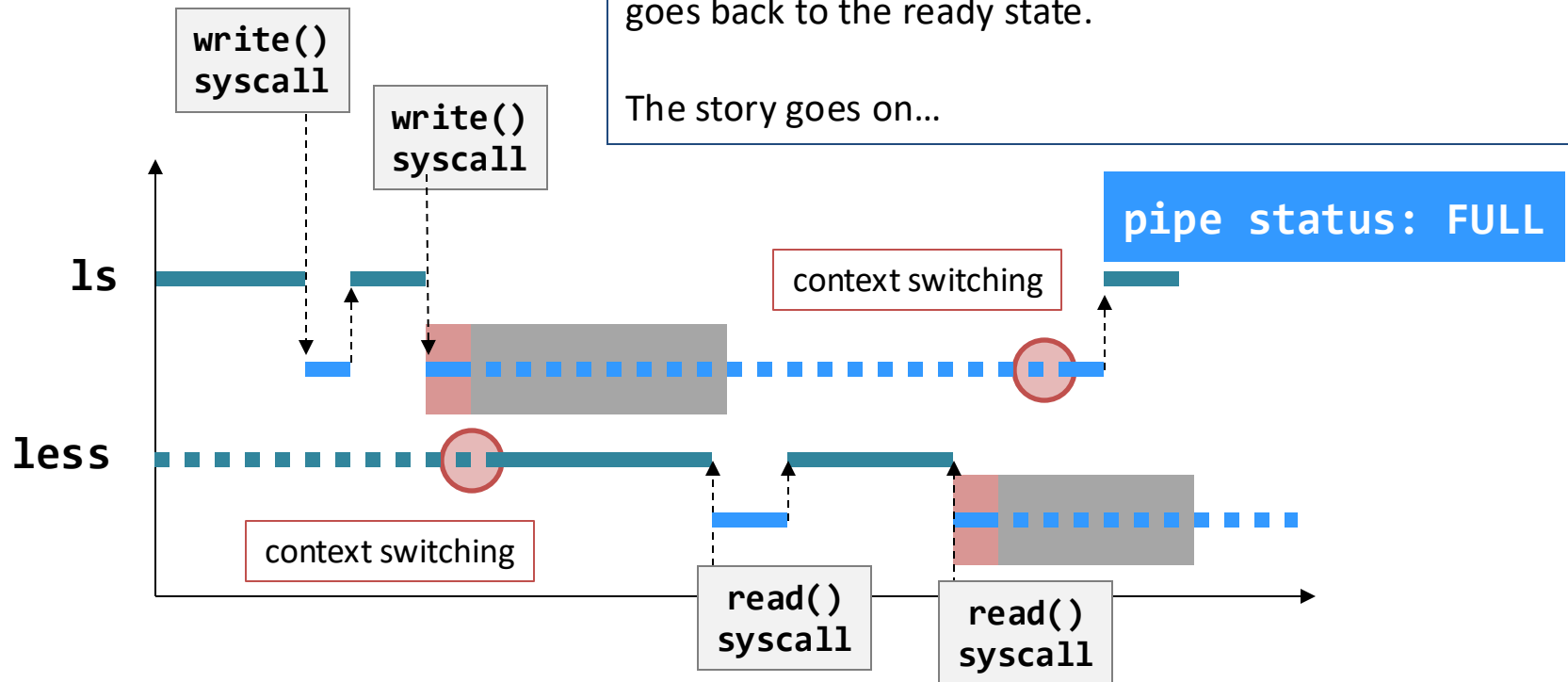
The kernel detects that the pipe is empty.
So, it blocks the running process **less**.

Producer-consumer problem – pipe

When “**ls**” process goes back to the running state, it **immediately writes a byte to the pipe so as to complete the execution of the write() system call.**

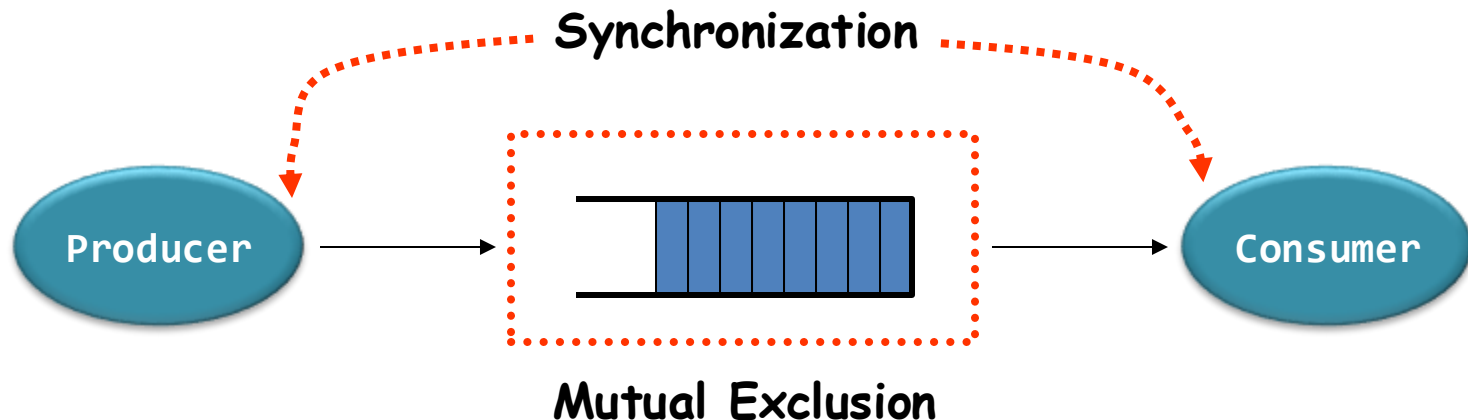
Then, the pipe is no longer empty, and the “**less**” process goes back to the ready state.

The story goes on...



Solving Producer-consumer problem by semaphore

- The Producer-consumer problem is **more general** than the pipe story
 - Pipe can't work with >1 producers/consumers
- 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



Solving Producer-consumer problem by semaphore

Shared object

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

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

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

Consumer Function

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

Solving Producer-consumer problem by semaphore

Note

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

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

Producer function

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


Solving Producer-consumer problem by semaphore

Note

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

10: (Producer) I **notify** the others that I have **filled** 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 **available** slot

Producer function

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

Consumer Function

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

Producer-consumer problem – question #1

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

Let's 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         sem-wait(&avail);
7         sem-wait(&mutex);
8         insert_item(item);
9         sem-post(&mutex);
10 sem-post(&fill);
11     }
12 }
```

Consumer Function

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

Producer-consumer problem – question #1

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

sem-wait s--
sem-post s++

So,

- producer s-- by sem-wait
- consumer s++ by sem-post

Problem solved?

Producer function

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

Consumer Function

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

Producer-consumer problem – question #1

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

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         sem-wait(&avail);
7         sem-wait(&mutex);
8         insert_item(item);
9         sem-post(&mutex);
10        sem-post(&fill);
11    }
12 }
```

Consumer Function

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

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;
```

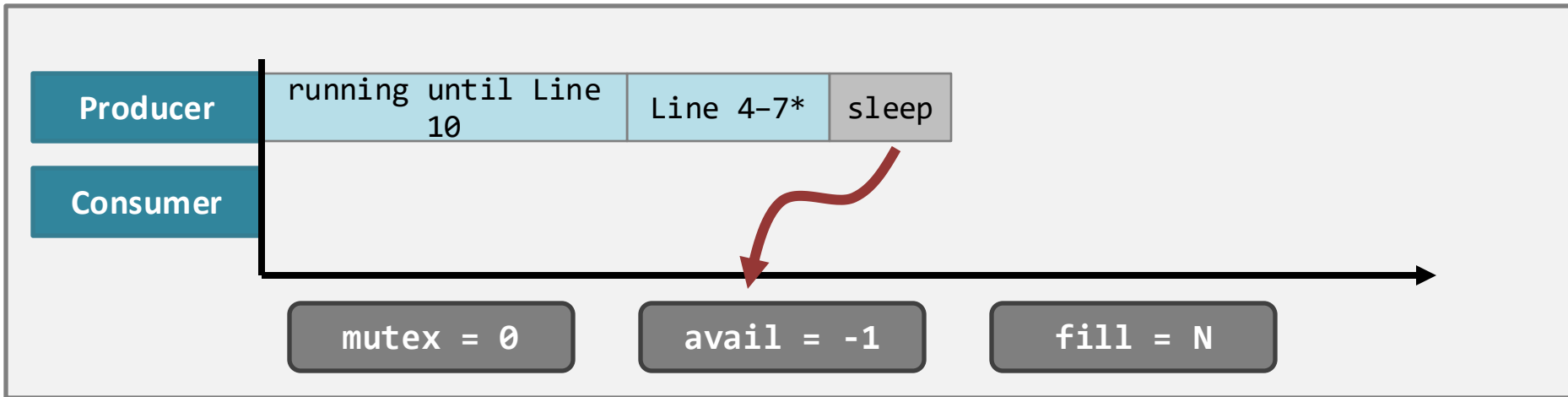
Producer function

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

Consumer Function

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

Producer-consumer problem – question #2



Producer function

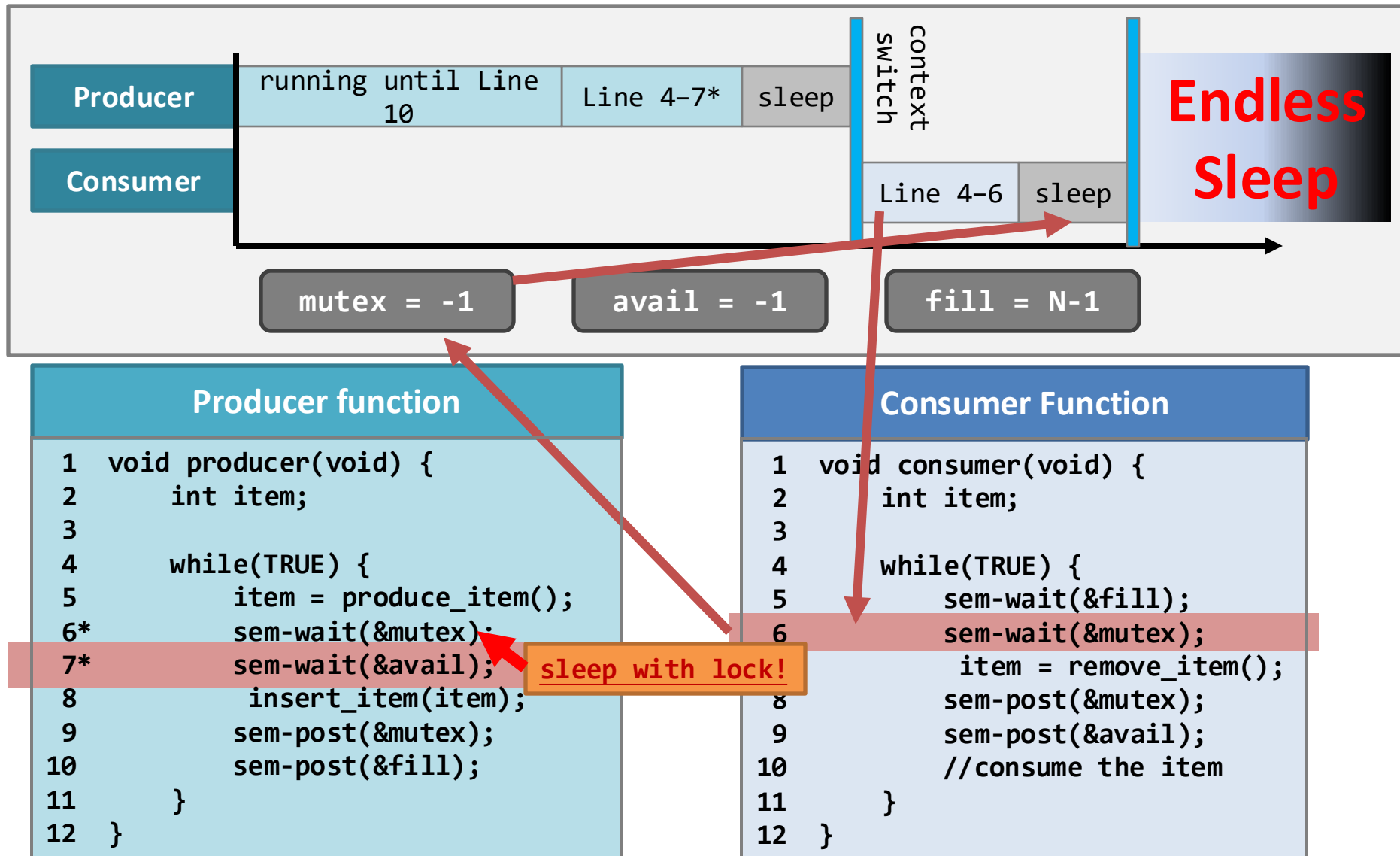
```
1 void producer(void) {
2     int item;
3
4     while(TRUE) {
5         item = produce_item();
6*      sem-wait(&mutex);
7*      sem-wait(&avail);
8         insert_item(item);
9         sem-post(&mutex);
10        sem-post(&fill);
11    }
12 }
```

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

Consumer Function

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

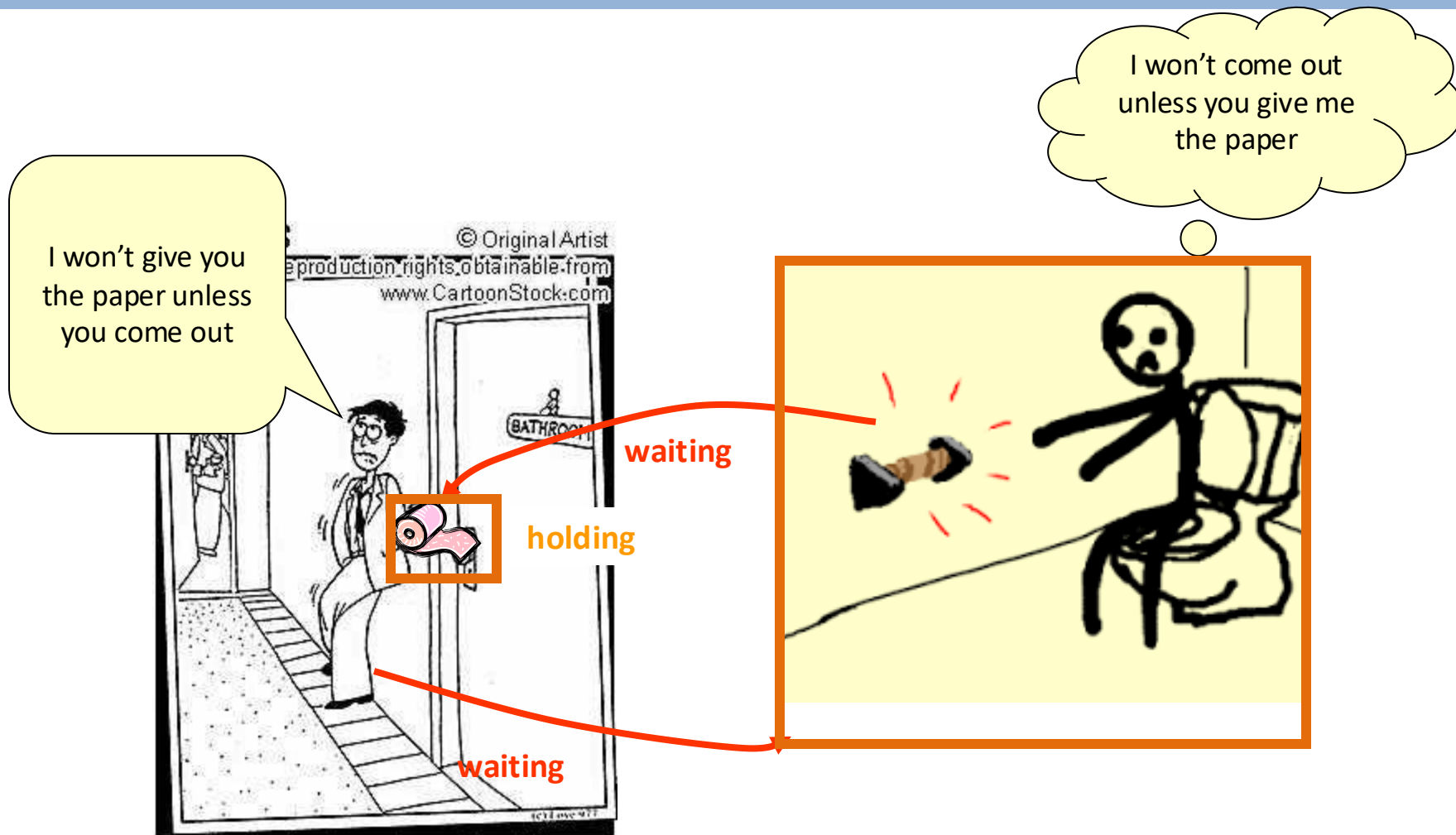
Producer-consumer problem – question #2



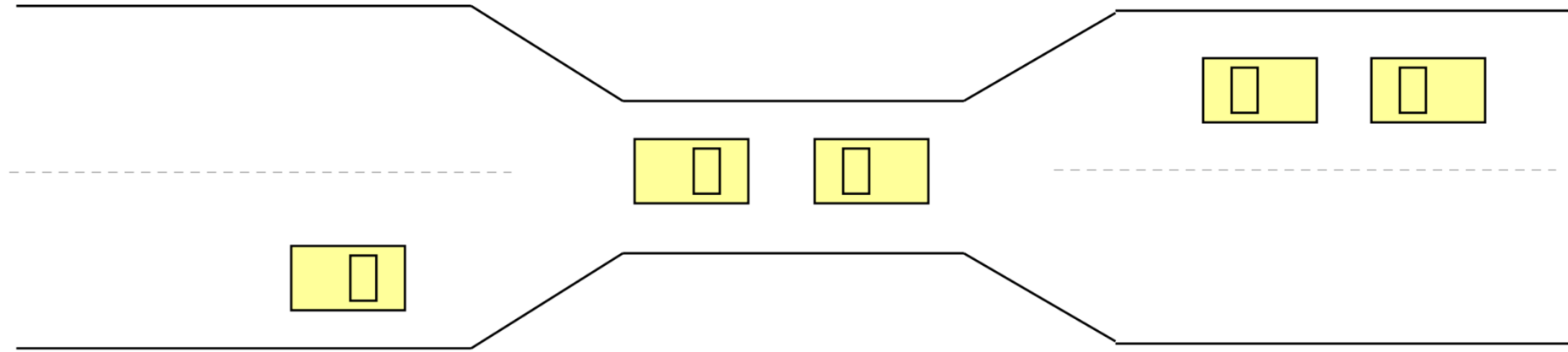
Producer-consumer problem – question #2

- 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



Deadlock



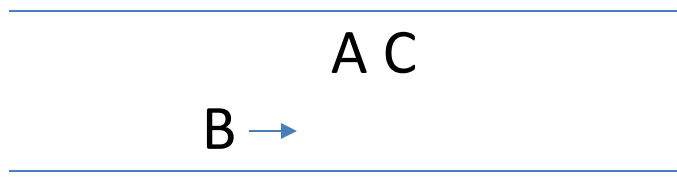
Livelock

- 3 people: A, B, C on an aisle

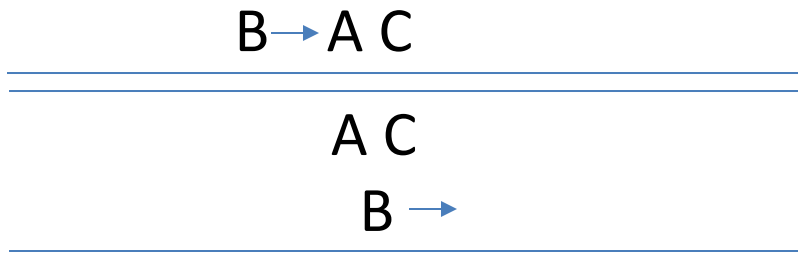


– Consider a solution of:

- When I crash into someone, I will move to the other side of the aisle and try to continue



They are not stuck at the current state
Their 'wake up' time is finite
They do wake up / return from wait()
But they still have **no progress**



Summary on producer-consumer problem

- How to avoid race condition on the shared buffer?
 - E.g., Use a **binary semaphore**.
- How to achieve coordination?
 - E.g., Use two other semaphores: fill and avail

Synchronization Primitives

POSIX **semaphore**

- For locking (mutual exclusion)
 - Binary semaphore
- For coordination
 - Semaphore with proper initialization values
- Semaphore is considered to be powerful but harmful
 - Powerful -> General
 - Harmful -> Buggy

pThread separates locking and coordination explicitly:


- E.g., pthread_**mutex_lock**, pthread_**mutex_trylock**, pthread_**spin_lock**, for locking (mutual exclusion)
- **Condition variable**, for coordination

Pthread's sleeplock

Data type

`pthread_mutex_t`

“mutex” is just a name, you can replace it with “kitty”



Initialization #1

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Declaring a mutex

Initialization #2

```
pthread_mutex_init(&mutex, NULL);
```

Initializing the mutex

Locking

```
pthread_mutex_lock(&mutex);
```

- If the mutex is not locked, lock the mutex.
- If the mutex is locked, block the calling thread.

Unlocking

```
pthread_mutex_unlock(&mutex);
```

- If the mutex is locked, then unlock the mutex. If there are threads blocked because of this mutex, one of those threads will resume.
- If the mutex is unlocked, do nothing.

Mutual exclusion example

Data type

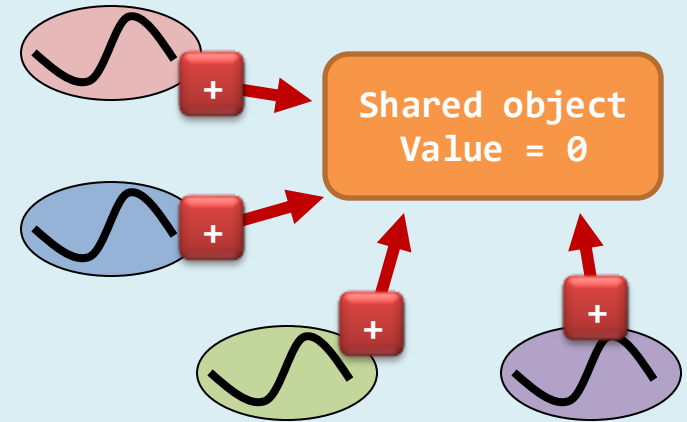
pthread_mutex_t

Thread Function

```
1 while(TRUE) {
2     pthread_mutex_lock(&mutex);
3     if(shared < MAXIMUM)
4         shared++;
5     else {
6         pthread_mutex_unlock(&mutex);
7         break;
8     }
9     pthread_mutex_unlock(&mutex);
10    sleep(rand() % 2);
11 }
```

Shared
object.

Purpose of the program?



[examples@3150] cat pthread_race_condition. Vs pthread_mutex.c

Pthread locks vs POSIX Semaphore

- **POSIX semaphore**

- Owned by kernel
- Can synchronize across process/thread
 - any thread/process can execute `sem_wait()` & `sem-post()` on any semaphores

- **pthread lock**

- Owned by pthread library
- Pthread mutexlock, pthread spinlock, pthread trylock

If protecting shared resources (i.e., requires mutual exclusion)

→ use semaphore or `pthread_mutex`

When doing something more (e.g., IPC problems like producer-consumer)

→ use semaphore or Pthread's condition variables

Condition Variable

- Semaphore
 - The condition of waking up others is **hard-coded inside** the semaphore as “s <= 0”
 - condition hard-coded!
- Condition variable
 - Keep the **atomically wait/notify** part
(i.e., person A: “I wait for Kitty”, person B: “I post the one who is waiting for Kitty”)
 - But wake somebody up once got posted, unconditionally
 - So, **waking up != the condition now holds**
 - Condition checking is left to outside
 - Can then support **any predicate**, e.g., $x \neq y$

Section Entry: sem-wait()

```
1 void sem-wait(semaphore *s) {
2     disable_interrupt();
3     *s = *s - 1;
4     if ( *s < 0 ) {
5         enable_interrupt();
6         sleep();
7         disable_interrupt();
8     }
9     enable_interrupt();
10 }
```

Section Exit: sem-post()

```
1 void sem-post(semaphore *s) {
2     disable_interrupt();
3     *s = *s + 1;
4     if ( *s <= 0 )
5         wakeup();
6     enable_interrupt();
7 }
```

Condition Variable v

- $y=99$; Shared variable $x=0$

Thread 1 Wait until $x == y$	Thread 2 $x++$
<pre>1 mutex_lock(&m); 2 while (x!=y) { 3 cond_wait(&v, &m); 4 } 5 printf(x/y = 1); 6 mutex_unlock(&m);</pre>	<pre>1 While (1){ 2 mutex_lock(&m); 3 x++; 4 cond_signal(&v); 5 mutex_unlock(&m); 6 }</pre>

2) Whenever I wakeup,
Thread 1 (re)-check the
condition

1) Whenever x is updated,
wake up Thread 1

So, condition variable is just a very simple form of semwait & sempost
But it is STATELESS. What's the advantage when something is stateless?

Condition Variable v

- $y=99$; Shared variable $x=0$

Thread 1 Wait until $x == y$	Thread 2 $x++$
<pre>1 mutex_lock(&m); 2 while (x!=y) { 3 cond_wait(&v, &m); 4 } 5 printf(x/y = 1); 6 mutex_unlock(&m);</pre>	<pre>1 mutex_lock(&m); 2 x++; 3 cond_signal(&v); 4 mutex_unlock(&m);</pre>

3) Sleep again if the condition does not hold

Wait... why passing $\&m$ to `cond_wait`?

Condition Variable v

- $y=99$; Shared variable $x=0$

Thread 1 Wait until $x == y$	Thread 2 $x++$
<pre>1 mutex_lock(&m); 2 while (x!=y) { 3 cond_wait(&v, &m); 4 } 5 printf(x/y = 1); 6 mutex_unlock(&m);</pre>	<pre>1 mutex_lock(&m); 2 x++; 3 cond_signal(&v); 4 mutex_unlock(&m);</pre>

If Thread 1 goes to sleep with the lock,
then Thread 2 never be able to
 $x++$;

- So, `cond_wait` atomically
- releases `&m` before sleeps
 - acquires `&m` before wakes up

Wait... why passing `&m` to
`cond_wait`?

PThread condition variable

Data type

`pthread_cond_t`

Condition variables are NOT variables! They have no value!

Vs

Semaphores: each includes a counter of # of resources.

Initialization #1

```
pthread_cond_t hellokitty = PTHREAD_COND_INITIALIZER;
```

Initialization #2

```
pthread_cond_init(&hellokitty, NULL);
```

Waiting #1

```
pthread_cond_wait(&hellokitty, &mutex);
```

Waiting #2

```
pthread_cond_timedwait(&hellokitty, &mutex, &timeout);
```

Wakeup #1

```
pthread_cond_signal(&hellokitty);
```

This function is to wake up a thread which is waiting on the **hellokitty**. If there are more than one thread waiting, then **at least one of those threads will wake up**.

Wakeup #2

```
pthread_cond_broadcast(&hellokitty);
```

This function is to **wake up all threads** which is waiting on the condition variable **hellokitty**.

Summary on synchronization problems

- The problems have the following properties in common:
 - Multiple number of processes (threads);
 - Processes (threads) have to be synchronized in order to generate useful output;
 - Each resource may be shared as well as limited.
- For lock-based approach:
 - It prevents data race by mutual exclusion;
 - It solves the synchronization problem elegantly by using synchronization primitives smartly
 - Elegantly means: Deadlock-free, bounded-waiting, with progress, thread-safe....

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 so that you can never debug it!

What the bug!

