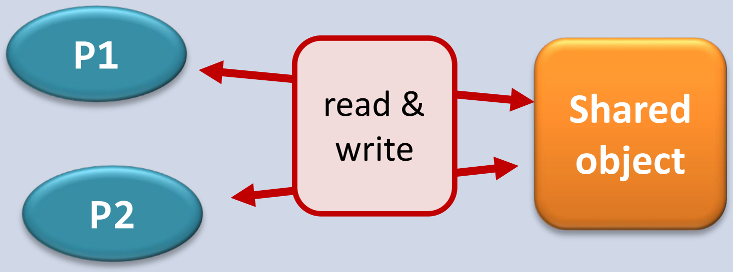
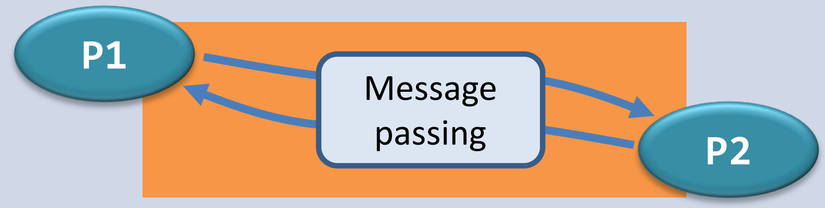
Operating Systems

Associate Prof. Yongkun Li
中科大-计算机学院 副教授
http://staff.ustc.edu.cn/~ykli

Ch5
Process Communication & Synchronization
-Part 2

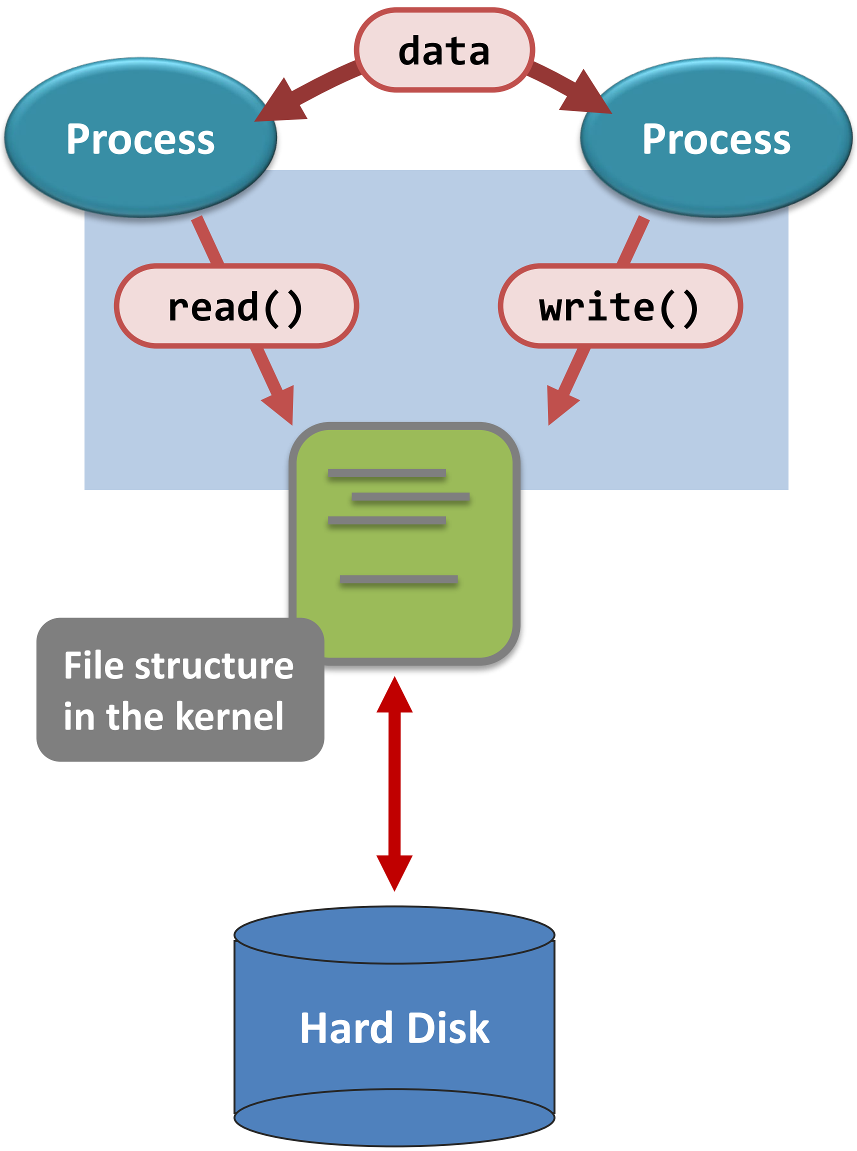
Summary on IPC models – another point of view

Shared Objects	Message Passing
	
<u>Challenge.</u> Coordination can only be done by detecting the status of the shared object. *E.g., is the pipe empty / full?*	**<u>Challenge.</u>** Coordination relies on the reliability and the efficiency of the communication medium (and protocol).
E.g., pipes, shared memory, and regular files.	E.g., socket programming, message passing interface (MPI) library.

IPC problem: Race condition

Evil source: the shared objects

- Pipe is implemented with the thought that **there may be more than one process accessing it "*at the same time*"**
- For shared memory and files, **concurrent access may yield <u>unpredictable outcomes</u>**



Understanding the problem…

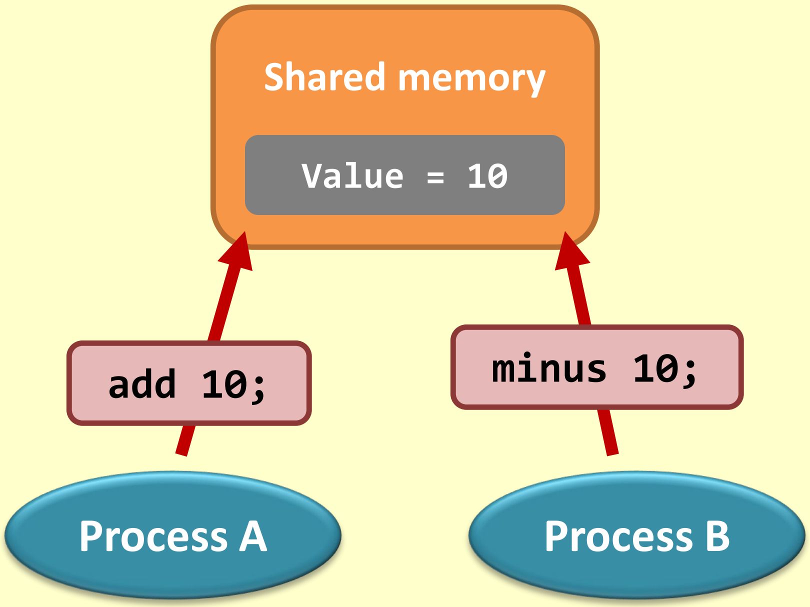
High-level language for Program A

```
attach to the shared memory X;
add 10 to X;
exit;
```

High-level language for Program B

```
attach to the shared memory X;
minus 10 to X;
exit;
```

The Scenario



Guess what the final result should be?

It may be 10, 0 or 20, can you believe it?

Understanding the problem…

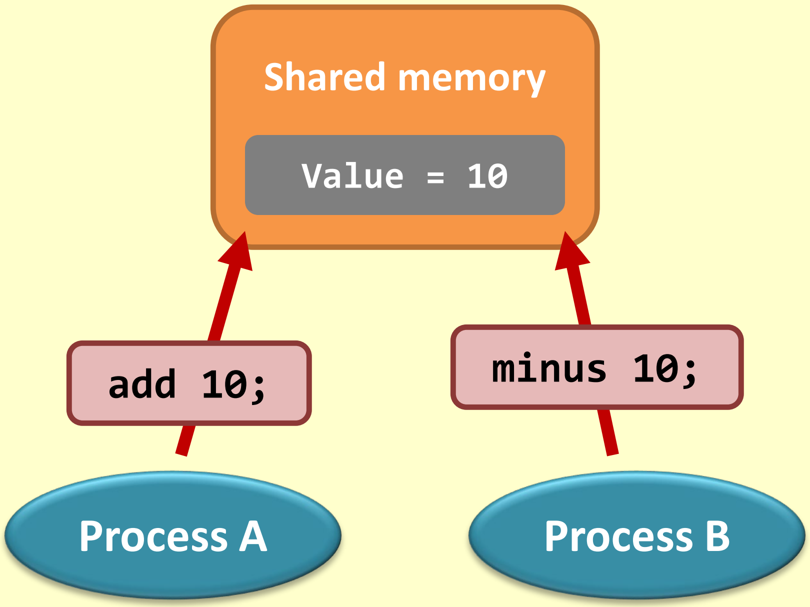
High-level language for Program A

```
attach to the shared memory X;
add 10 to X;
exit;
```

High-level language for Program B

```
attach to the shared memory X;
minus 10 to X;
exit;
```

The Scenario



Remember the flow of executing a program and the system hierarchy?

Understanding the problem…

High-level language for Program A

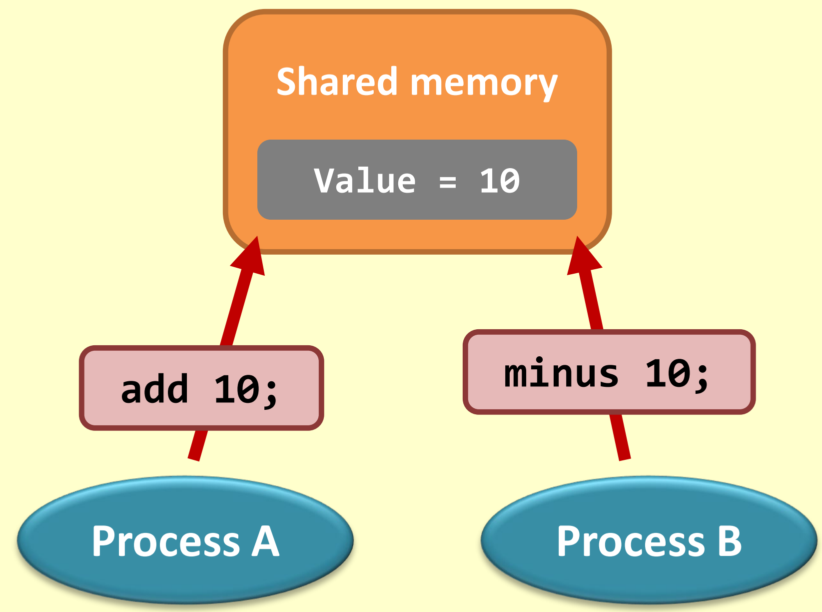
```
1  attach to the shared memory X;
2  add 10 to X;
3  exit;
```

This operation is not atomic

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



Guess what? This code block is evil!

Understanding the problem…

Shared memory - X

`Value = 10`

Process A

State: Ready

```
Register A
Value = 0
```

```
2.1   load memory X to
      register A;

2.2   add 10 to register A;

2.3   write register A to
      memory X;
```

Process B

```
Register A
Value = 0
```

State: Ready

```
2.1   load memory X to
      register A;

2.2   minus 10 from register A;

2.3   write register A to
      memory X;
```

The initial setting

Execution Flow #1

Problem not yet arise…

Shared memory - X

Value = 10

Process A

State: Running

Register A Value = 10

1

```
2.1  load memory X to
     register A;

2.2  add 10 to register A;

2.3  write register A to
     memory X;
```

Process B

Register A Value = 0

State: Ready

```
2.1  load memory X to
     register A;

2.2  minus 10 from register A;

2.3  write register A to
     memory X;
```

Execution Flow #1, Step 1

Problem not yet arise…

Shared memory - X

Value = 10

Process A

State: Running

Register A Value = 20

1 2.1 load memory X to register A;

2 2.2 add 10 to register A;

2.3 write register A to memory X;

Process B

Register A Value = 0

State: Ready

2.1 load memory X to register A;

2.2 minus 10 from register A;

2.3 write register A to memory X;

Execution Flow #1, Step 2

Problem not yet arise…

Shared memory - X

Value = 20

Process A

State: Running

Register A Value = 20

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

Process B

Register A Value = 0

State: Ready

2.1 load memory X to register A;

2.2 minus 10 from register A;

2.3 write register A to memory X;

Execution Flow #1, Step 3

Problem not yet arise...

Shared memory - X

`Value = 20`

Process A

State: Ready

```
Register A
Value = 20
```

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

Context Switching

Process B

State: Running

```
Register A
Value = 20
```

4. `2.1  load memory X to register A;`

`2.2  minus 10 from register A;`

`2.3  write register A to memory X;`

Execution Flow #1, Step 4

Problem not yet arise…

Shared memory - X

Value = 20

State:
Ready

Register A
Value = 20

Register A
Value = 10

State:
Running

Process A

1 2.1 load memory X to register A;

2 2.2 add 10 to register A;

3 2.3 write register A to memory X;

Process B

2.1 load memory X to register A; 4

2.2 minus 10 from register A; 5

2.3 write register A to memory X;

Execution Flow #1, Step 5

Problem not yet arise…

Shared memory - X

Value = 10

Process A	
State:	Ready
Register A	Value = 20

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

Process A

Process B	
Register A	Value = 10
State:	Running

```
2.1   load memory X to        4
      register A;
2.2   minus 10 from register A;  5
2.3   write register A to     6
      memory X;
```

Process B

Execution Flow #1, Step 6

Execution Flow #2

Problem arise…

Shared memory - X

Value = 10

State: Running

Register A
Value = 10

Register A
Value = 0

State: Ready

1

```
2.1   load memory X to
      register A;

2.2   add 10 to register A;

2.3   write register A to
      memory X;
```

```
2.1   load memory X to
      register A;

2.2   minus 10 from register A;

2.3   write register A to
      memory X;
```

Process A

Process B

Execution Flow #2, Step 1

Problem arise...

Shared memory - X

Value = 10

Process A	Process B
State: Ready	State: Running
Register A Value = 10	Register A Value = 10
1	2
2.1 load memory X to register A;	2.1 load memory X to register A;
2.2 add 10 to register A;	2.2 minus 10 from register A;
2.3 write register A to memory X;	2.3 write register A to memory X;

Context Switching

Execution Flow #2, Step 2

Problem arise…

Shared memory - X

Value = 10

State: Running

Register A Value = 20

Register A Value = 10

State: Ready

1

2.1 load memory X to register A;

3

2.2 add 10 to register A;

2.3 write register A to memory X;

2.1 load memory X to register A;

2

2.2 minus 10 from register A;

2.3 write register A to memory X;

Process A

Context Switching

Process B

Execution Flow #2, Step 3

Problem arise...

Shared memory - X

Value = 10

State: <u>Ready</u>	Register A Value = 20

1 — 2.1 load memory X to register A;

3 — 2.2 add 10 to register A;

2.3 write register A to memory X;

Process A

Register A Value = 0	State: <u>Running</u>

2.1 load memory X to register A; — 2

2.2 minus 10 from register A; — 4

2.3 write register A to memory X;

Process B

Context Switching

<u>Execution Flow #2, Step 4</u>

Problem arise…

Shared memory - X

Value = 10

State: Ready

Register A
Value = 20

Register A
Value = 0

State: Running

Process A

1 2.1 load memory X to register A;

3 2.2 add 10 to register A;

2.3 write register A to memory X;

Process B

2.1 load memory X to register A; 2

2.2 minus 10 from register A; 4

2.3 write register A to memory X;

HELP!! No matter which process runs next, **the result is either 0 or 20, but not 10**!

The final result depends on the execution sequence!

Race condition – the curse

- The above scenario is called the **race condition**.

- A **race condition** means
  - the outcome of an execution depends on a particular order in which the shared resource is accessed.

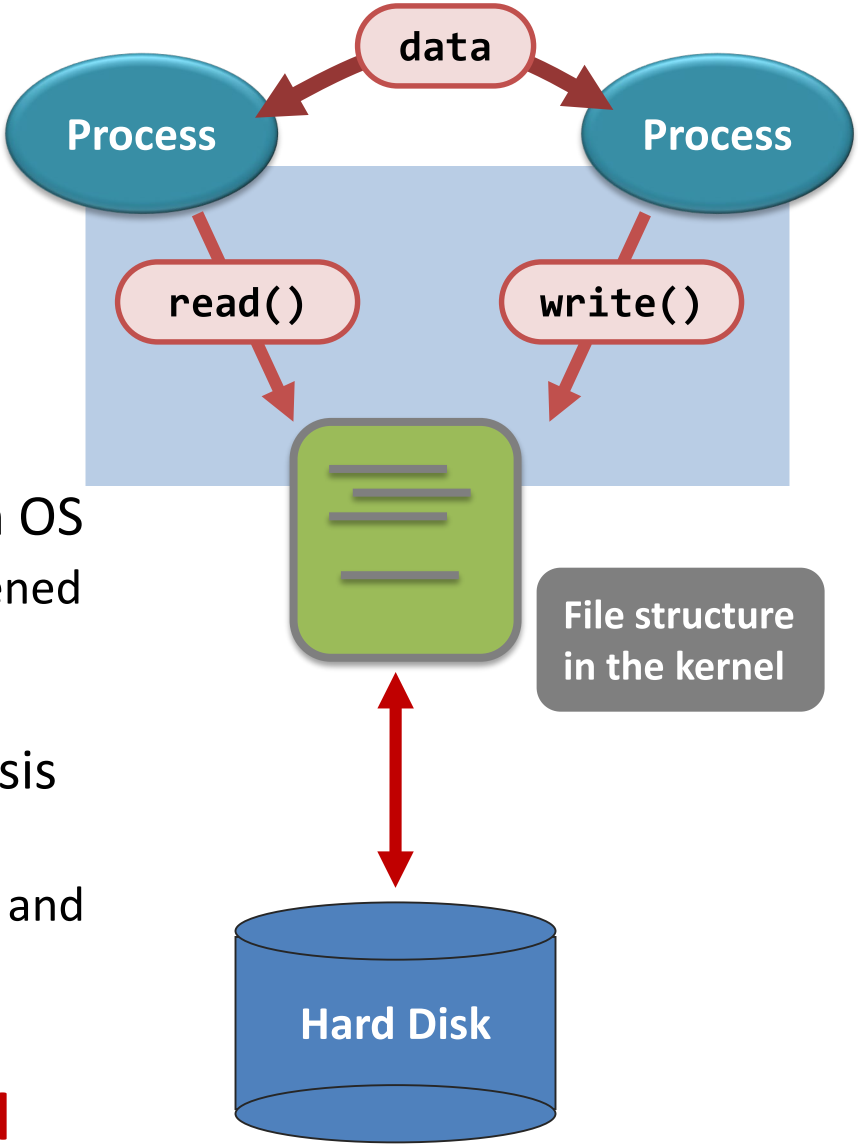
- Remember: race condition is always a bad thing and debugging race condition has no fun at all!
  - It may end up …
    - 99% of the executions are fine.
    - 1% of the executions are problematic.

Race condition – the curse

- For shared memory and files, **concurrent access may yield <u>unpredictable outcomes</u>**
  - **<u>Race condition</u>**

- Common situation
  - Resource sharing occurs frequently in OS
    - EXP: Kernel DS maintaining a list of opened files, maintaining memory allocation, process lists…
  - Multicore brings an increased emphasis on multithreading
    - Multiple threads share global variables and dynamically allocated memory

- **Process synchronization is needed**



Topics in Process Synchronization

Cooperating Processes

concurrent accesses suffer from **race condition**

Solution

Process Sychronization

Guarantee **mutual exclusion**

Application

Semaphore Usage

Avoid **deadlock**

Idea: How to achieve

Define **critical section**

How to implement

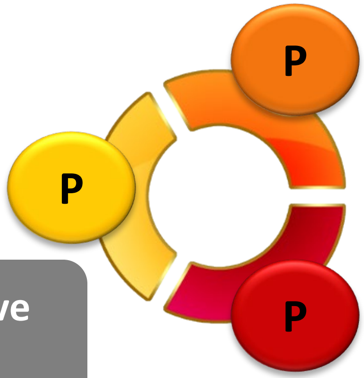
- Four requirements
- Software-based proposals
  - Disabling interrupts
  - strict alternation
  - peterson's solution
  - mutex lock
  - **Semaphore (best choice)**

Classic problems

- Producer-consumer problem
- Dining philosopher problem
- Reader-writer problem

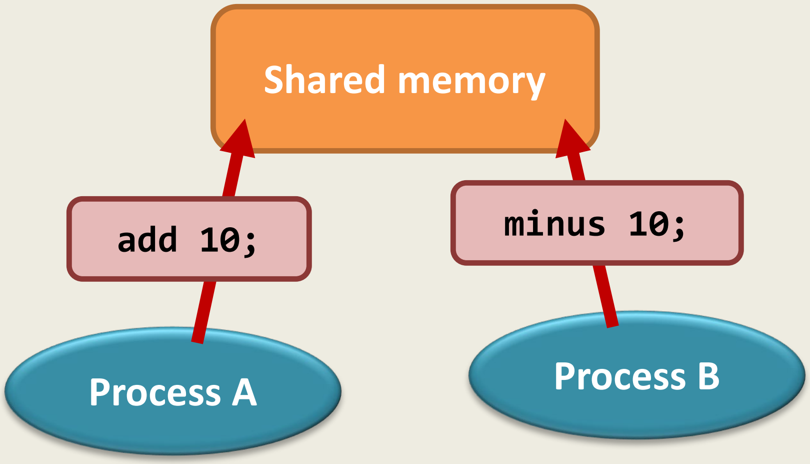
Inter-process communication (IPC)

- Mutual exclusion
  - what & how to achieve?



How to have peace?

Mutual Exclusion



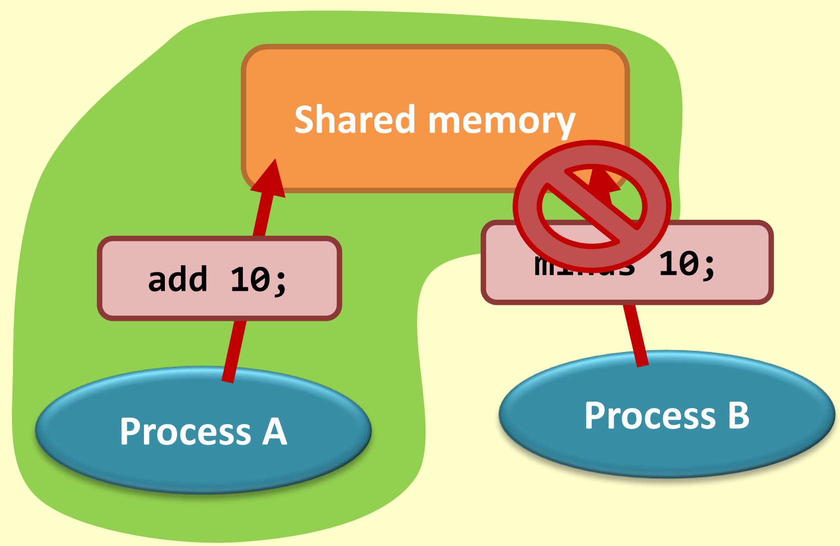
Two processes playing with **the same shared memory** is dangerous.

We will face the curse - **race condition**.

The solution can be simple:

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

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

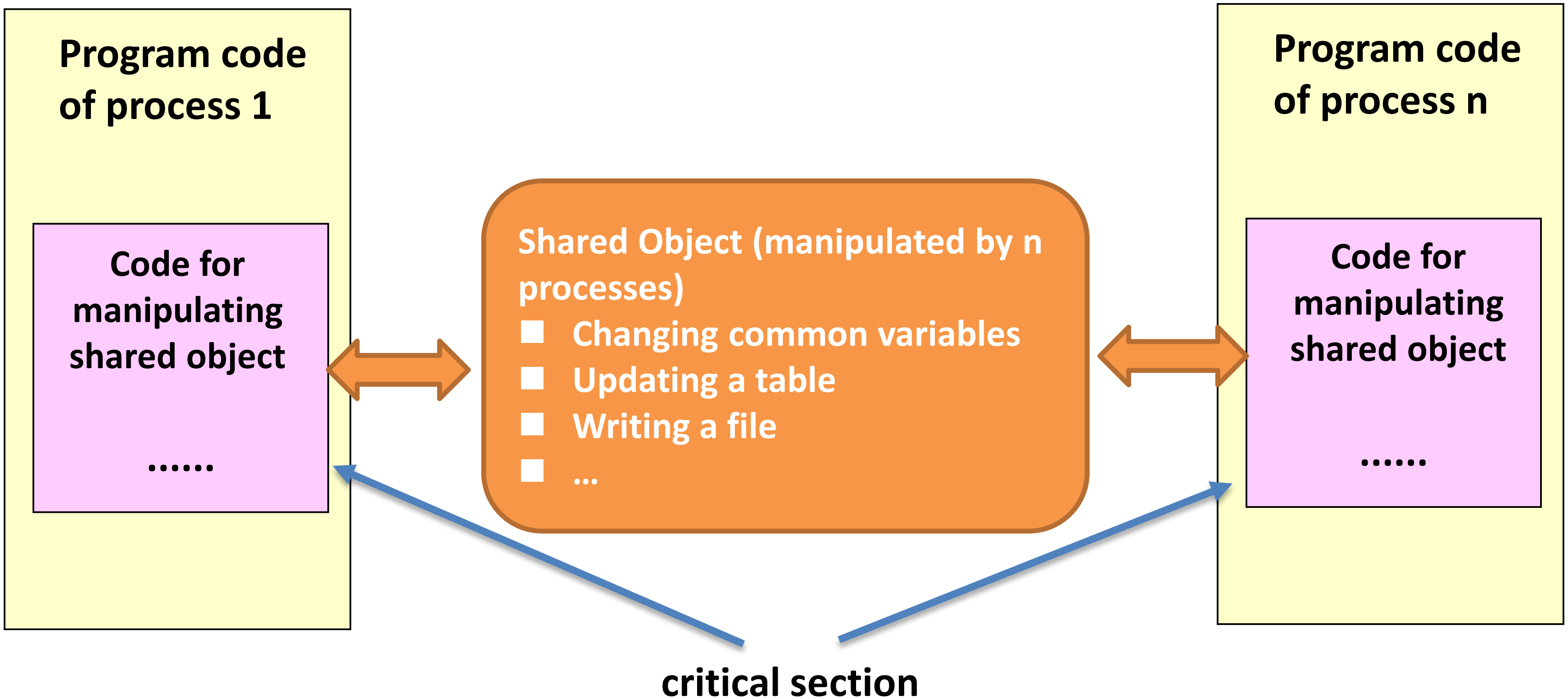


How to realize mutual exclusion?

- Kernel
  - Preemptive kernels and nonpreemptive kernels
    - Allows (not allow) a process to be preempted while it is running in kernel mode
  - A nonpreemptive kernel is essentially free from race conditions on kernel data structures, and also easy to design (especially for SMP architecture)
  - Why would anyone favor a preemptive kernel
    - More responsive
    - More suitable for real-time programming

Mutual Exclusion

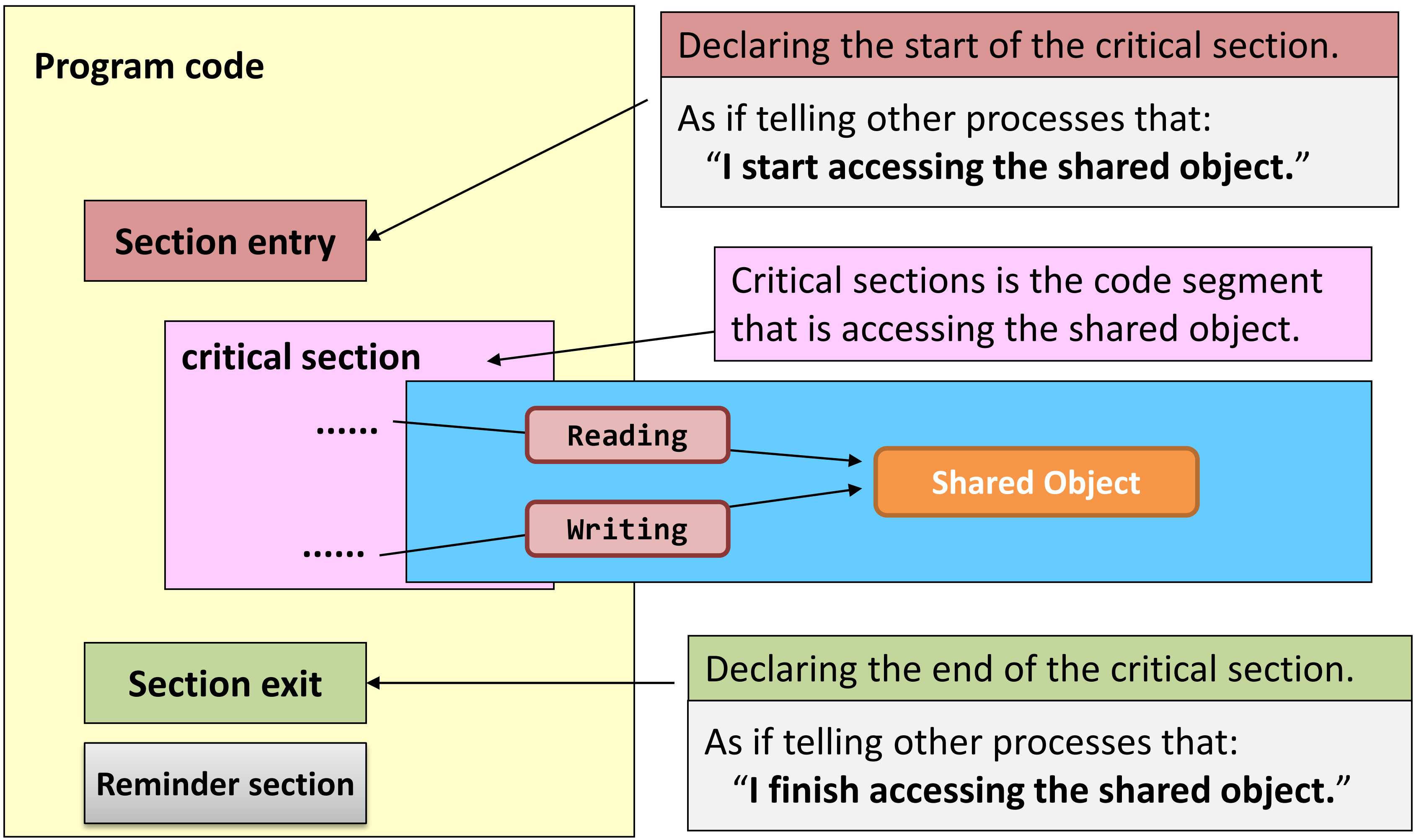
- More generally, how to realize?



Solution: To guarantee that when one process is executing in its critical section, no other process is allowed execute in its critical section.

Critical Section – General Structure

To guarantee that when one process is executing in its critical section, no other process is allowed execute in its critical section.



Critical Section – Example

Process A

Need a section entry here

Critical Section

```
2.1   load memory X to
      register A;

2.2   add 10 to register A;

2.3   write register A to
      memory X;
```

Need a section exit here

Process B

Need a section entry here

```
2.1   load memory X to
      register A;

2.2   minus 10 from register A;

2.3   write register A to
      memory X;
```

Need a section exit here

Important concept here.

Both regions are called critical sections, yet they can be different.

Summary…for the content so far…

- **Race condition** is a problem.
  - It makes a concurrent program producing **unpredictable** results if you are using shared objects as the communication medium.
  - 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.
  - Mutual exclusion hinders the performance of parallel computations.

Summary…for the content so far…

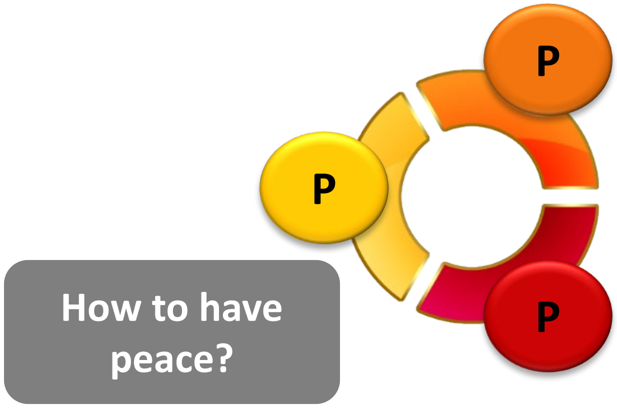
- **Defining critical sections** is a solution.
  - They are code segments that access shared objects.
  - Critical section must be **as tight as possible**.
    - Well, you can declare the entire code of a program to be a big critical section.
    - But, the program will be a very high chance to block other processes or to be blocked by other processes.
  - Note that **one critical section** can be designed for **accessing more than one shared objects**.

Summary…for the content so far…

- **Implementing section entry and exit** is a challenge.
  - The entry and the exit are **the core parts that guarantee mutual exclusion**, but not the critical section.
  - Unless they are correctly implemented, race condition would appear.

Inter-process communication (IPC)

- Mutual exclusion:
  - how to achieve?
  - how to implement?
    (section entry and exit)



Entry and exit implementation - requirements

- **Requirement #1: Mutual Exclusion**. No two processes could be simultaneously inside their critical sections.

Implication: when one process is inside its critical section, any attempts to go inside the critical sections by other processes are not allowed.

- **Requirement #2**. Each process is executing at a nonzero speed, but no assumptions should be made about the relative speed of the processes and the number of CPUs.

Implication: the solution **cannot depend on the time spent inside the critical section**, and the solution cannot assume the number of CPUs in the system.

Entry and exit implementation - requirements

- **Requirement #3: progress**. No process running outside its critical section should block other processes.

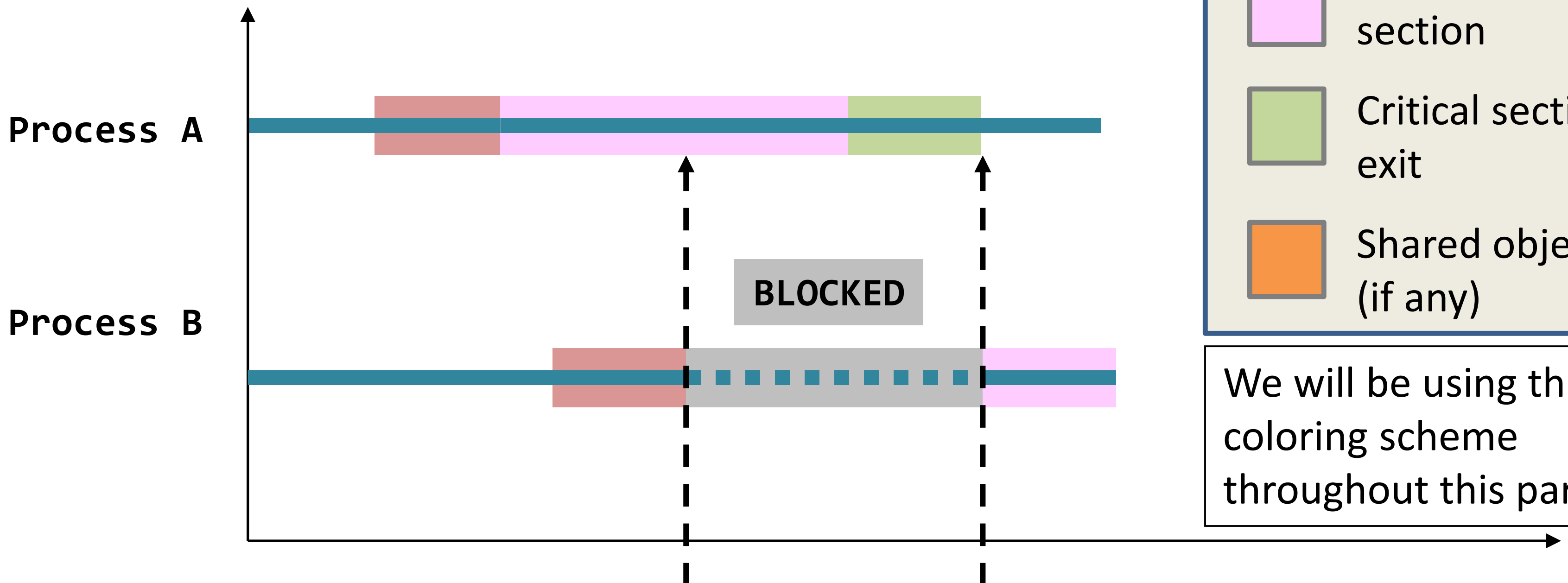
> **Implication**: Only processes that are not executing in their reminder sections can participate in deciding which will enter its critical section.

- **Requirement #4: Bounded waiting**. No process would have to wait forever in order to enter its critical section.

> **Implication**: There exists a bound or limit on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section (no processes should be **starved to death**).

A typical mutual exclusion scenario

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



Keys

- Critical section entry
- Inside Critical section
- Critical section exit
- Shared object (if any)

We will be using this coloring scheme throughout this part.

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

A leaves its critical section and B resumes execution accordingly.

Mutual Exclusion Implementation

- Challenges of Implementing **section entry** & **exit**
  - Both operations must be atomic
  - Also need to satisfy the above requirements
  - Performance consideration

- Hardware solution
  - Rely on atomic instructions
  - test_and_set()
  - compare_and_swap

Example: test_and_set()

- Definition

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

- Mutual exclusion implementation

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Example: compare_and_swap()

- Definition

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;

   if (*value == expected)
      *value = new_value;

   return temp;
}
```

- Mutual exclusion implementation

```
do {
   while (compare_and_swap(&lock, 0, 1) != 0)
      ; /* do nothing */

      /* critical section */

   lock = 0;

      /* remainder section */
} while (true);
```

How to satisfy bounded waiting?

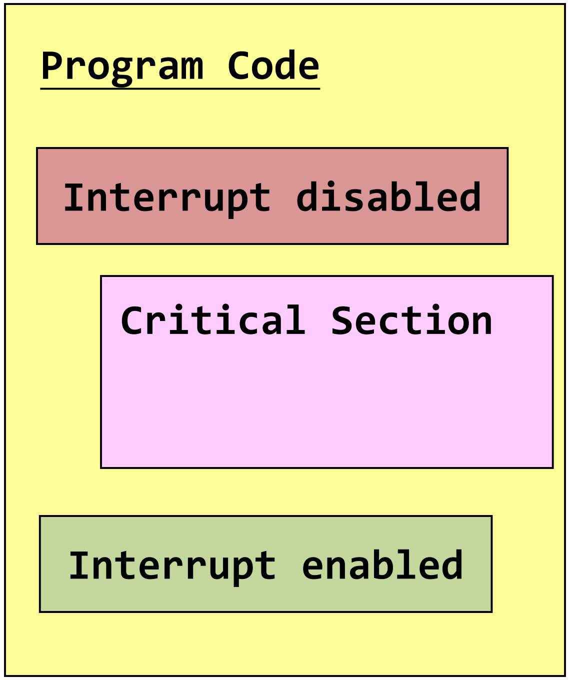
Enhanced version

```
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);
```

lock is initialized as false

Proposal #1 – disabling interrupt.

- **Method**
  - Similar idea as nonpreemptive kernels
  - To **<u>disable context switching</u>** when the process is inside the critical section.
- **Effect**
  - When a process is in its critical section, no other processes could be able to run.
- **Implementation**
  - A new system call should be provided.
- **Correctness?**
  - **<u>Correct,</u>** but it is not an *<u>attractive</u>* solution.
  - Not as feasible in a multiprocessor environment
  - Performance issue (may sacrifice concurrency)



Proposal #2: Mutex Locks

- **Idea**
  - A process must acquire the lock before entering a critical section, and release the lock when it exits the critical section
  - Using a new shared object to detect the status of other processes, and "**lock**" the shared object

Shared object: "available" (lock)

```
acquire(){
   while(!available)
         ; /* busy waiting */
   available = false;
 }
```

```
release(){
   available = true;
 }
```

Proposal #2: Mutex Locks

- **Implementation**
  - Calls to acquire and release locks must be performed **atomically**
  - Often use hardware instructions
- **Issue**
  - Busy waiting: Waste CPU resource
    - **Spinlock**
- **Applications**
  - Multiprocessor system
    - When locks are expected to be held for short times

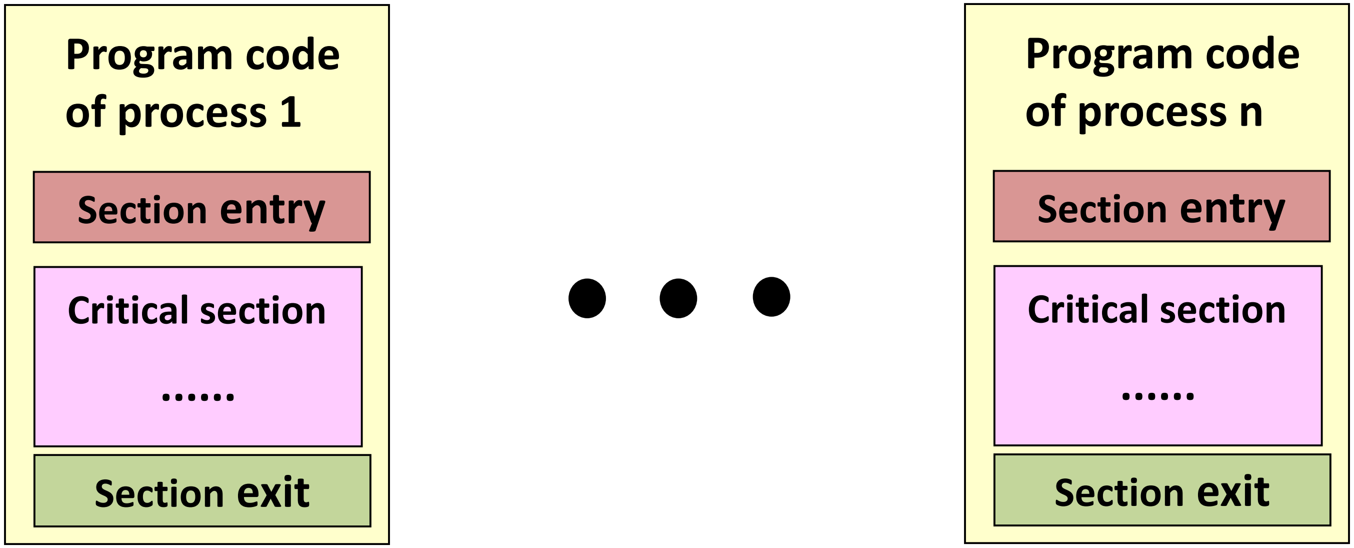
Note that: all processes run the following same code.

Program Code

```
acquire();

    Critical Section

release();
```

Other software-based solutions

- Aim
  - To decide which process could go into its critical section



- Key Issues
  - Detect the status of processes (section entry)
    - Need other shared variables
  - Atomicity of section entry and exit

Proposal #3: Strict alternation

- **Method**
  - Using a new shared object to detect the status of other processes

Shared object “turn” initial Value = 0

Process 0

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

Allow to enter when turn == 0

Entry

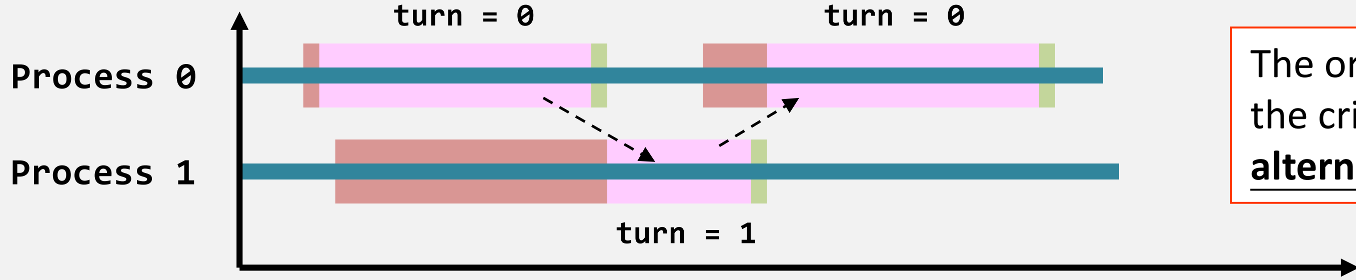
Exit

Process1

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

Allow to enter when turn == 1

Proposal #3: Strict alternation



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

Shared object “turn” initial Value = 0

Process 0

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

Process1

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

Proposal #3: Strict alternation - Cons

- Strict alternation seems good, yet, it is **inefficient**.
  - Busy waiting wastes CPU resources.

- In addition, the alternating order is **too strict**.
  - What if Process 0 wants to enter the critical section **twice in a row**? **NO WAY!**
  - Violate any requirement?

> **Requirement #3**. No process running outside its critical section should block other processes.

Proposal #4: Peterson’s solution

- How to improve the strict alternation proposal?
  - The Peterson’s solution

- Highlights:
  - Share two data items
    - **`int turn;`** //whose turn to enter its critical section
    - **`Boolean interested[2];`** //if a process wants to enter
  - Processes would <u>act as a gentleman</u>: if you want to enter, I’ll let you first
  - No alternation is there

Proposal #4: Peterson’s solution

Shared object: “turn” & “interested[2]”

```
int turn;                          /* who can enter critical section */
int interested[2] = {FALSE,FALSE}; /* wants to enter critical section*/

void enter_region( int process ) { /* process is 0 or 1 */
  int other;                       /* number of the other process */
  other = 1-process;               /* other is 1 or 0 */
  interested[process] = TRUE;      /* want to enter critical section */
  turn = other;
  while ( turn == other &&
           interested[other] == TRUE )
     ;    /* busy waiting */
}

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

Entry

Exit

Proposal #4: Peterson’s solution

```
1   int turn;
2   int interested[2] = {FALSE,FALSE};
3
4   void enter_region( 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 leave_region( int process ) {
14    interested[process] = FALSE;
15  }
```

Line 8 therefore makes the other one the turn to run.

Of course, the process is willing to wait when she wants to enter the critical section.

“I’m a gentleman!”

The process always let another process to enter the critical region first although she wants to enter too.

Proposal #4: Peterson’s solution

```
1   int turn;
2   int interested[2] = {FALSE,FALSE};
3
4   void enter_region( 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 leave_region( int process ) {
14    interested[process] = FALSE;
15  }
```

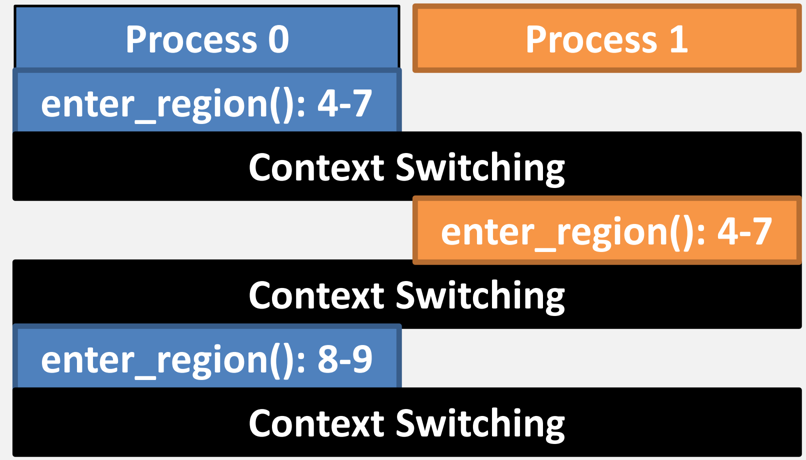
Process 0	Process 1
enter_region(): 4-8	turn = 1;
Context Switching	
turn = 0;	enter_region(): 4-8
Context Switching	
enter_region(): 9 Critical Section	turn = 0; interested[1] = T;
Context Switching	
	Busy waiting
Context Switching	
leave_region()	interested[0] = F;
Context Switching	
	Critical Section

and the story goes on…

Can you show that the requirements are satisfied?

Proposal #4: Peterson’s solution

```
1   int turn;
2   int interested[2] = {FALSE,FALSE};
3
4   void enter_region( 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 leave_region( int process ) {
14    interested[process] = FALSE;
15  }
```

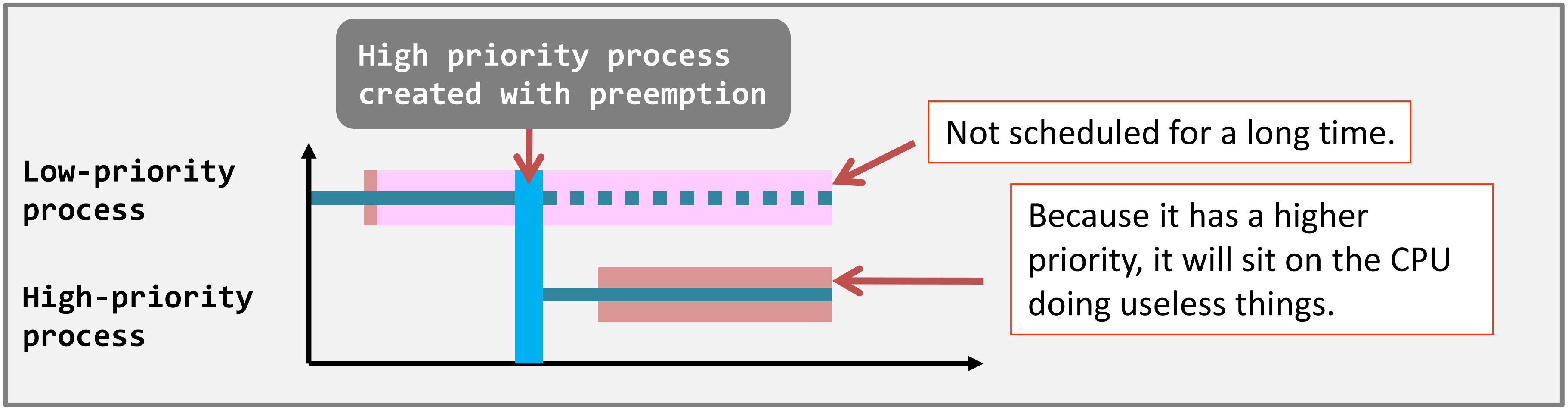


Can you complete the flow?
(what is the difference?)

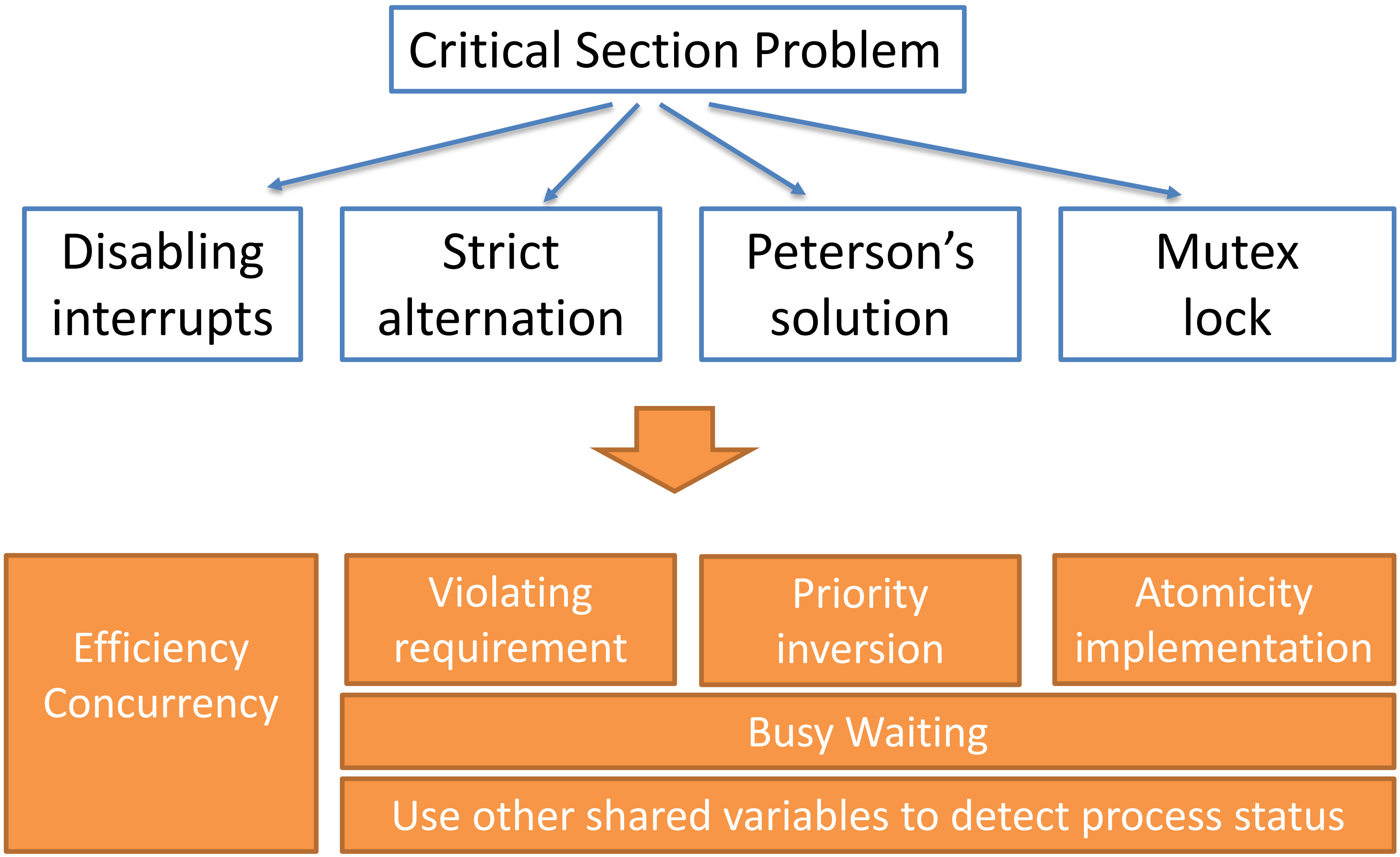
Can both processes progress?

Proposal #4: Peterson's solution – issues

- Busy waiting has its own problem…
  - **An apparent problem**: wasting CPU time.
  - **A hidden, serious problem**: **priority inversion problem.**
    - A low priority process is inside the critical region, but …
    - A high priority process wants to enter the critical region.
    - Then, the high priority process will perform busy waiting for a long time or even forever.

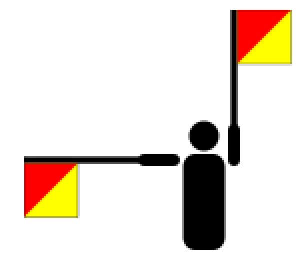


Story so far…


Critical Section Problem
Disabling interrupts
Strict alternation
Peterson's solution
Mutex lock
Efficiency Concurrency
Violating requirement
Priority inversion
Atomicity implementation
Busy Waiting
Use other shared variables to detect process status

Final proposal: Semaphore

- In real life, semaphore is a flag signaling system.
  - It tells a train driver (or a plane pilot) when to stop and when to proceed.



source: wikipedia.

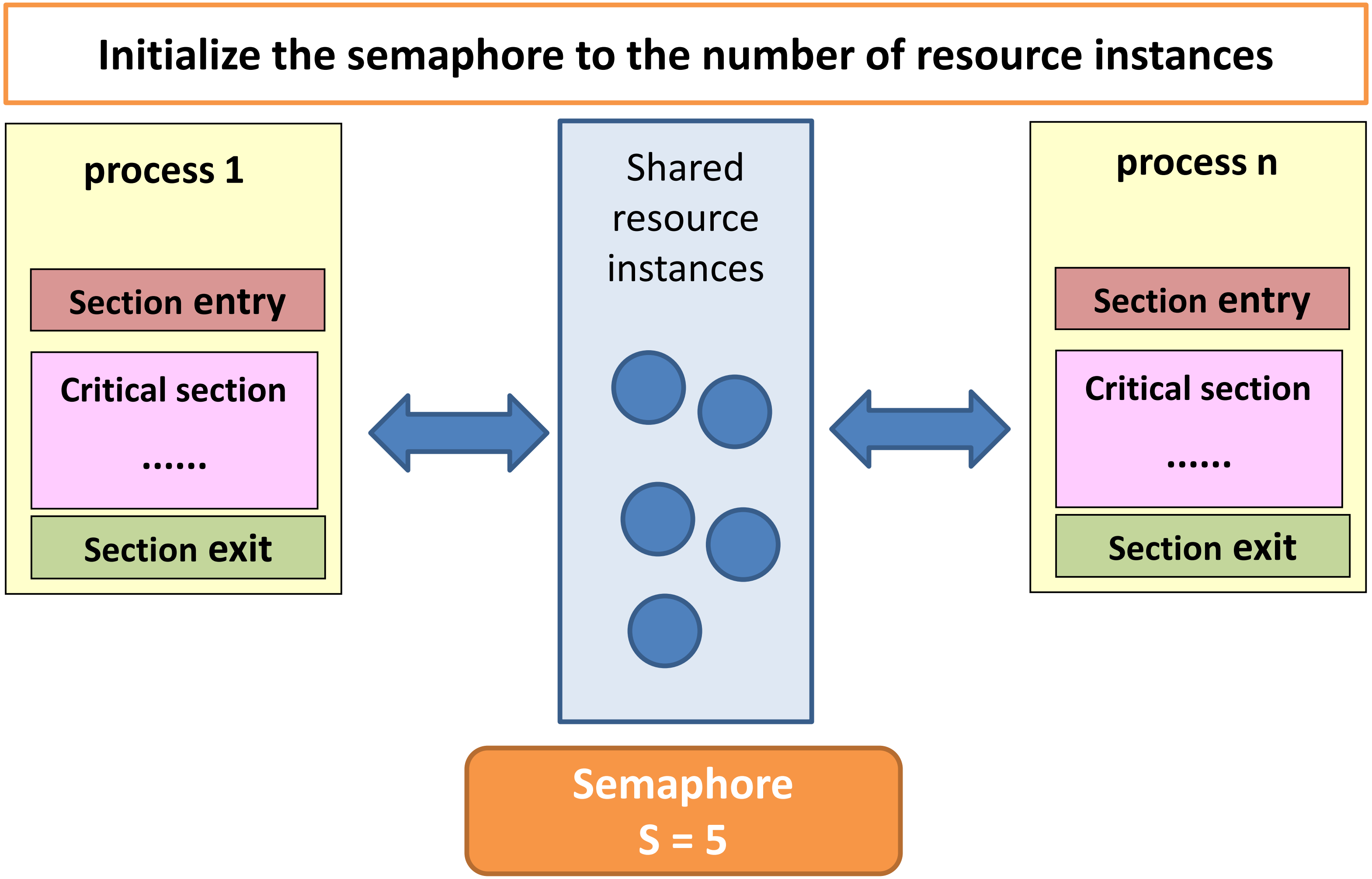
- When it comes to programming…
  - A semaphore is a data type.
  - You can imagine that it is **<u>an integer</u>** (but it is certainly not an integer when it comes to real implementation).

Final proposal: Semaphore

- Semaphore is a data type (**additional shared object**)
  - Accessed only through two standard **atomic** operations
  - **`down():`** originally termed P (from Dutch *proberen*, “to test”), **`wait()`** in textbook
    - Decrementing the count
  - **`up():`** originally termed V (from *verhogen*, “to increment”), **`signal()`** in textbook
    - Incrementing the count
- Two types
  - **Binary semaphore**: 0 or 1 (similar to mutex lock)
  - **Counting semaphore**: control finite number of resources

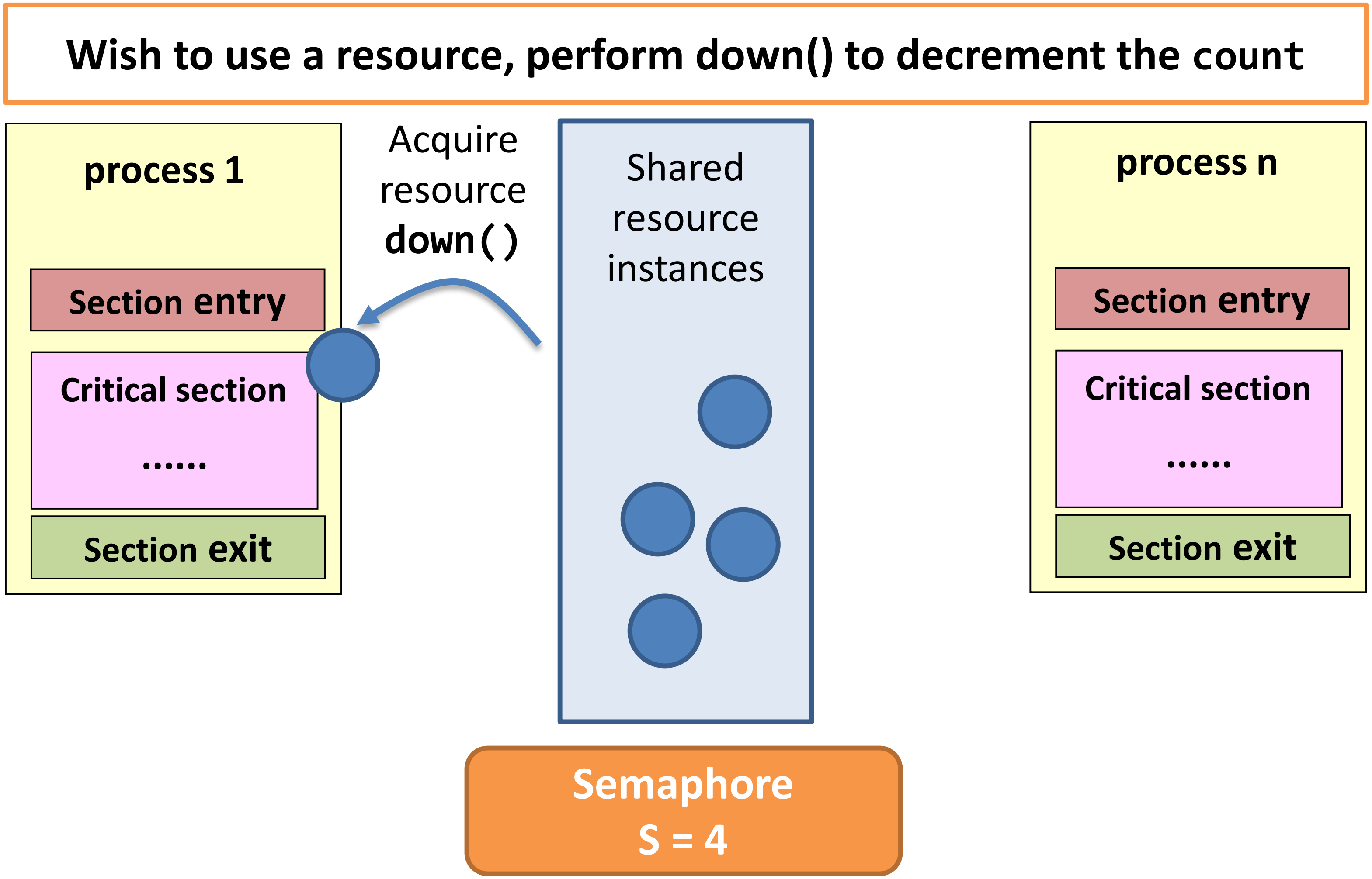
Final proposal: Semaphore

- Idea



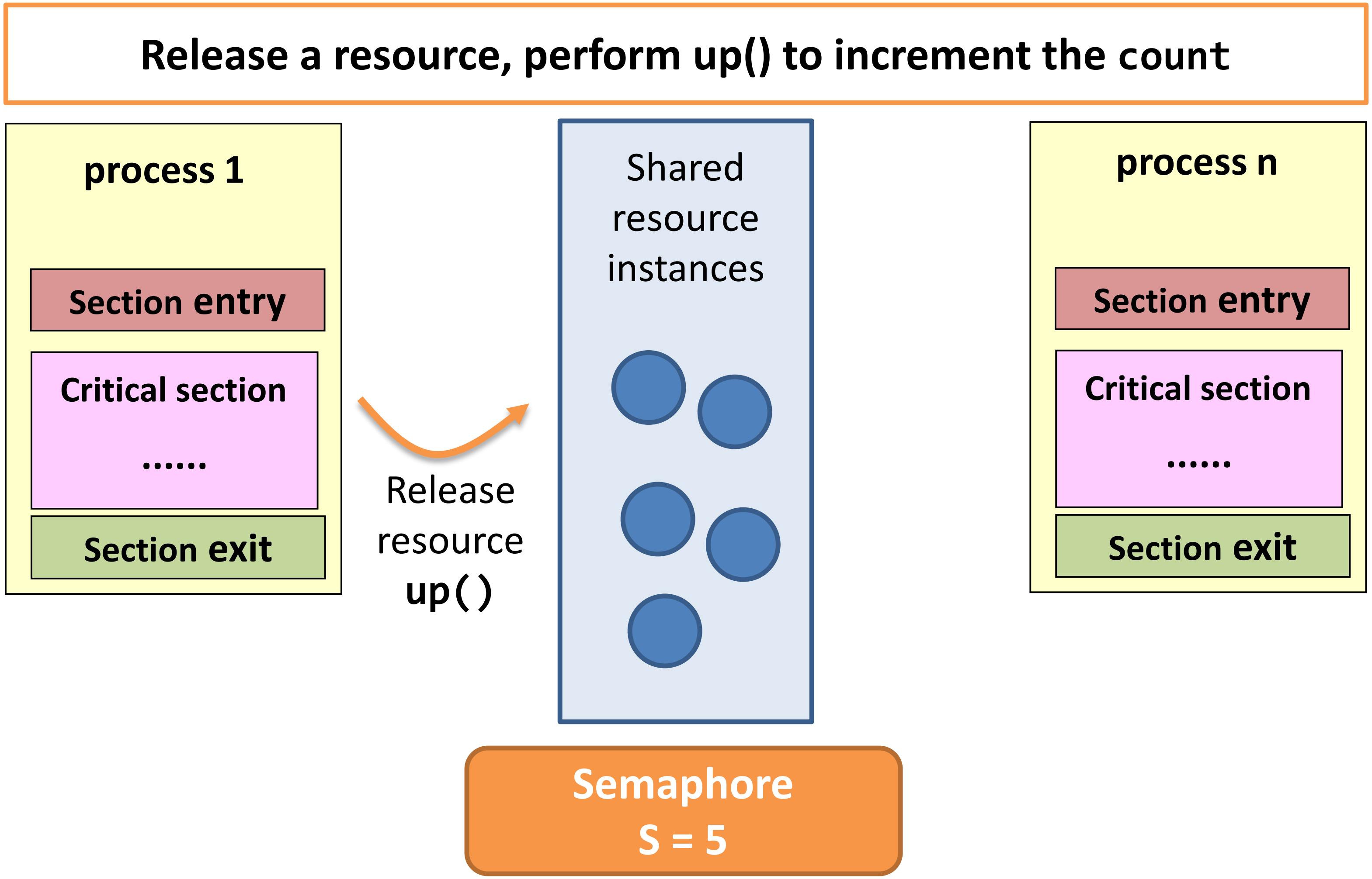
Final proposal: Semaphore

- Idea



Final proposal: Semaphore

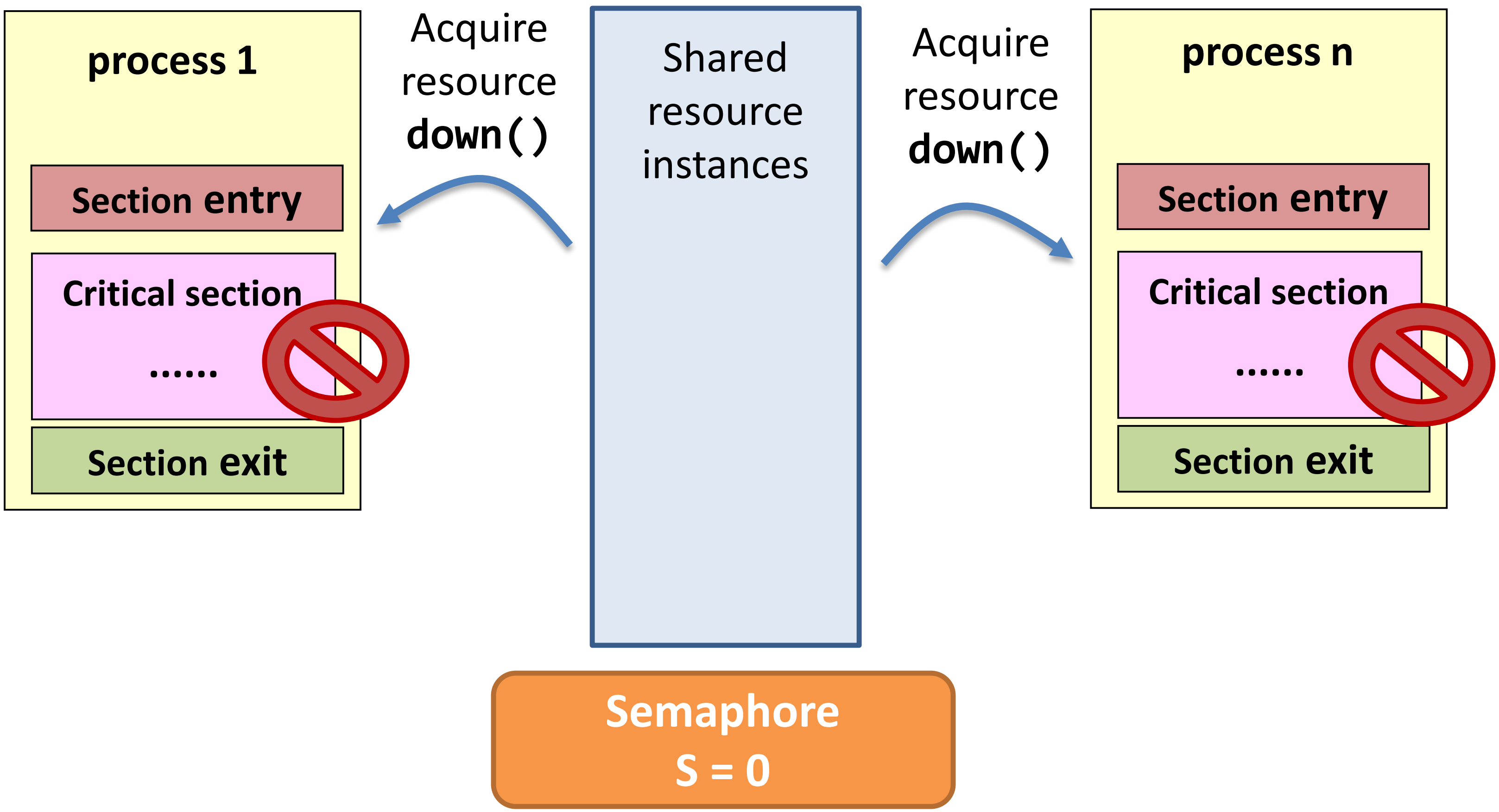
- Idea



Final proposal: Semaphore

- Idea

When the count goes to 0, block the processes that wish to use



Semaphore – Simple Implementation

Data Type definition

```
typedef int semaphore;
```

Section Entry: down()

```
void down(semaphore *s) {

    while ( *s == 0 ) {

          ;//busy wait

     }
    *s = *s - 1;

}
```

Counting Semaphore: initialized to be the number of resources available

Section Exit: up()

```
void up(semaphore *s) {



   *s = *s + 1;

}
```

Semaphore – Address busy waiting

Data Type definition

```
typedef int semaphore;
```

First issue: Busy waiting

Solution: block the process instead of busy waiting (place the process into a waiting queue)

Section Entry: down()

```
void down(semaphore *s) {

    while ( *s == 0 ) {

        special_sleep();

     }
    *s = *s – 1;

}
```

Section Exit: up()

```
void up(semaphore *s) {

  if ( *s == 0 )
    special_wakeup();
  *s = *s + 1;

}
```

Semaphore – Address busy waiting

Data Type definition

```
typedef int semaphore;
```

First issue: Busy waiting

Solution: block the process instead of busy waiting (place the process into a waiting queue)

```
typedef struct{

     int value;
     struct process * list;

}semaphore;
```

Note

Implementation: The waiting queue may be associated with the semaphore, so a semaphore is not just an integer

Semaphore – Atomicity

Data Type definition

```
typedef int semaphore;
```

Section Entry: down()

```
void down(semaphore *s) {
    while ( *s == 0 ) {
        special_sleep();
     }
    *s = *s - 1;
}
```

Second issue: Atomicity (both operations must be atomic)

Solution: Disabling interrupts

Section Exit: up()

```
void up(semaphore *s) {
  if ( *s == 0 )
    special_wakeup();
  *s = *s + 1;
}
```

Semaphore – Atomicity

Data Type definition

```
typedef int semaphore;
```

Section Entry: down()

```
void down(semaphore *s) {
    disable_interrupt();
    while ( *s == 0 ) {
        enable_interrupt();
        special_sleep();
        disable_interrupt();
    }
    *s = *s - 1;
    enable_interrupt();
}
```

Second issue: Atomicity (both operations must be atomic)

Solution: Disabling interrupts

Also, only one process can invoke "**disable_interrupt()**". Later processes would be blocked until "**enable_interrupt()**" is called.

Section Exit: up()

```
void up(semaphore *s) {
  disable_interrupt();
  if ( *s == 0 )
    special_wakeup();
  *s = *s + 1;
  enable_interrupt();
}
```

Semaphore – The code

Data Type definition

```
typedef int semaphore;
```

Section Entry: down()

```
void down(semaphore *s) {
    disable_interrupt();
    while ( *s == 0 ) {
        enable_interrupt();
        special_sleep();
        disable_interrupt();
    }
    *s = *s - 1;
    enable_interrupt();
}
```

Why need these two statements?

Disabling interrupts may sacrifice concurrency, so it is essential to keep the critical section as short as possible

Section Exit: up()

```
void up(semaphore *s) {
    disable_interrupt();
    if ( *s == 0 )
        special_wakeup();
    *s = *s + 1;
    enable_interrupt();
}
```

Semaphore – details

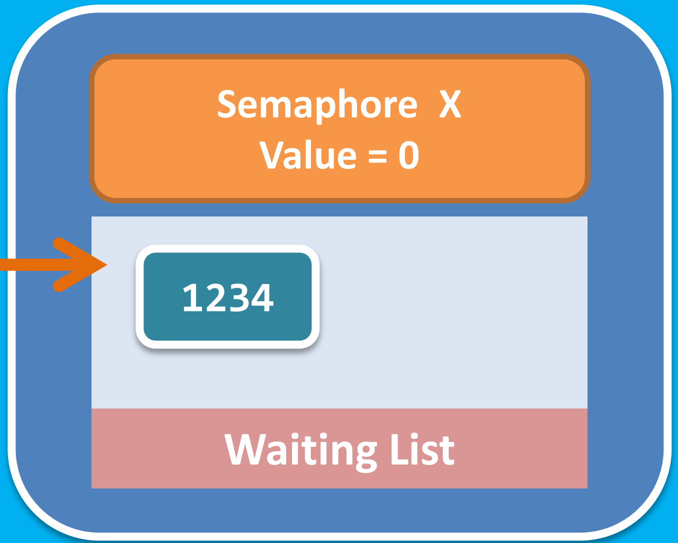
Suppose that process 1234 is willing to access the shared resource (enter its critical section), but no resource is available

Process 1234

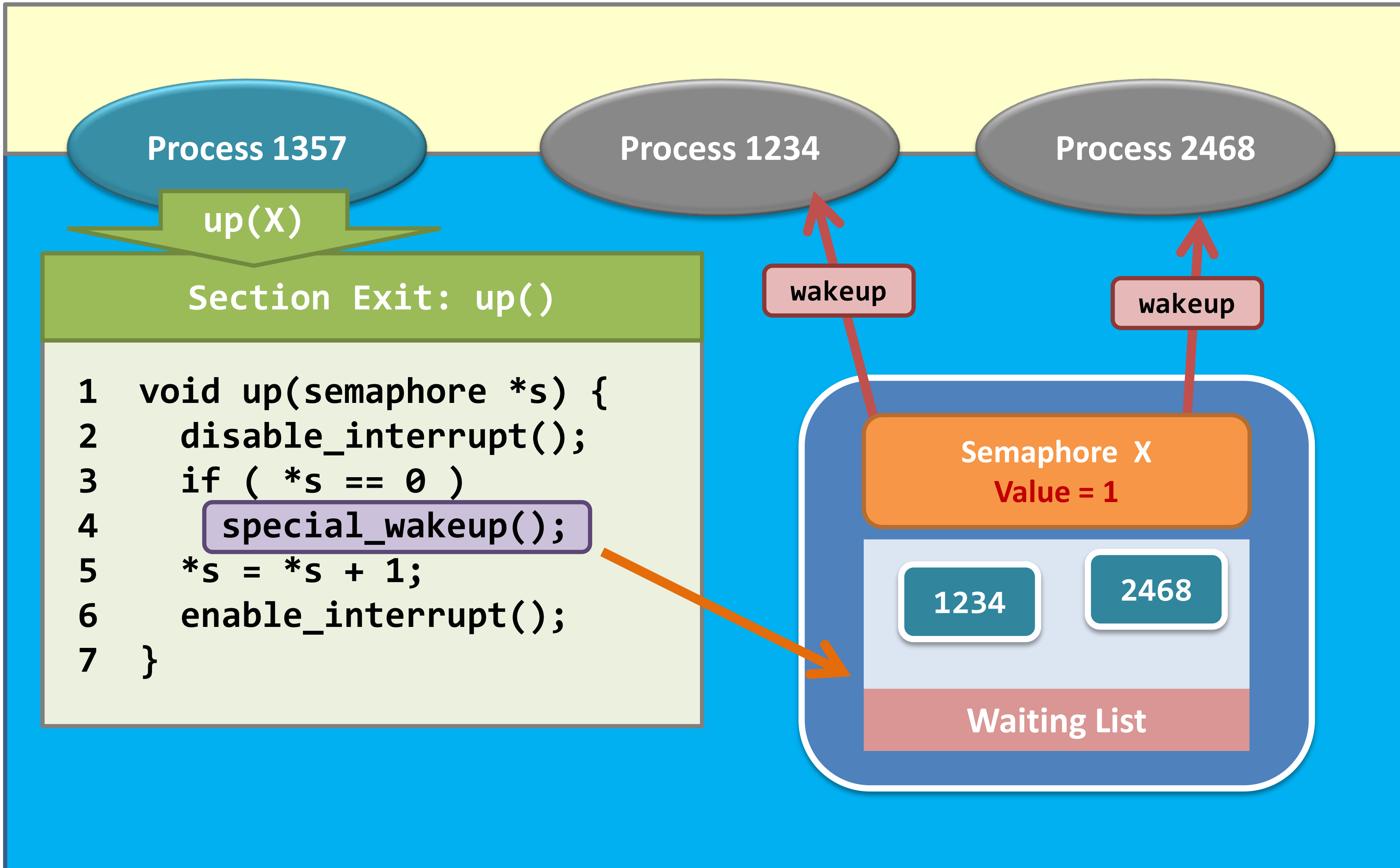
down(X)

Section Entry: down()

```
void down(semaphore *s) {
    disable_interrupt();
    while ( *s == 0 ) {
        enable_interrupt();
        special_sleep();
        disable_interrupt();
     }
    *s = *s - 1;
    enable_interrupt();
}
```



Semaphore – details



Semaphore – details

Process 1234

down(X)

Process 2468

down(X)

Note that it is impossible for **two blocked processes to get out of the down( ) simultaneously.**

Why?

Only one process can invoke **`disable_interrupt()`**

Only one process can manipulate this shared variable

Section Entry: down()

```
void down(semaphore *s) {
    disable_interrupt();
    while ( *s == 0 ) {
        enable_interrupt();
        special_sleep();
        disable_interrupt();
    }
    *s = *s - 1;
    enable_interrupt();
}
```

here

Semaphore – details

Process 1234

Process 2468

down(X)

down(X)

Note that it is impossible for **two processes to get out of the down( ) simultaneously.**

Why?

Whether which process can get out of **down( )** is **the business of the scheduler**.

Section Entry: down()

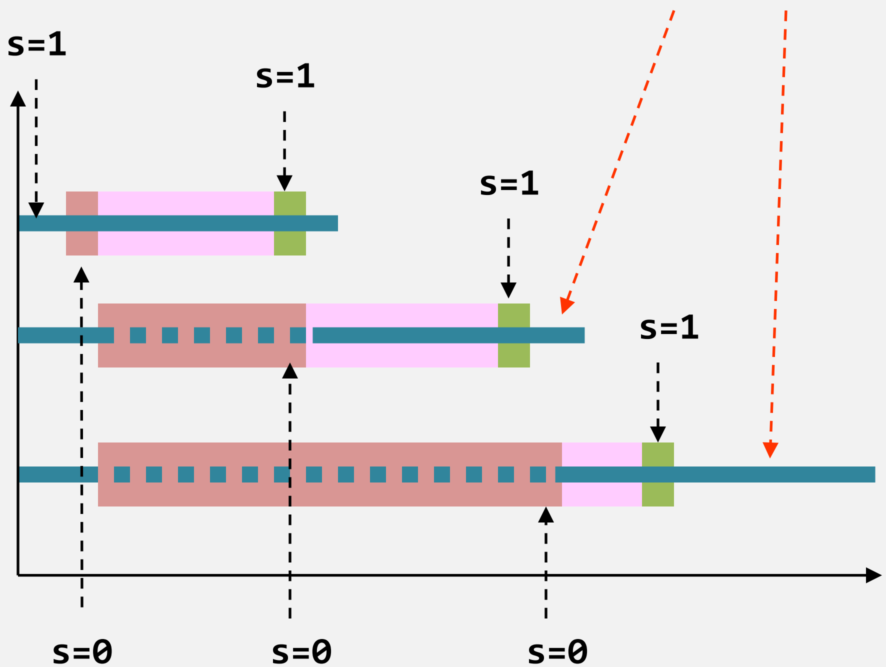
```
void down(semaphore *s) {
    disable_interrupt();
    while ( *s == 0 ) {
        enable_interrupt();
        special_sleep();
        disable_interrupt();
     }
    *s = *s - 1;
    enable_interrupt();
}
```

here

Semaphore – in action

- Add them together…

Either one of the processes can enter the critical section when the first process calls “**up(s)**”.




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

```
while(TRUE) {
    down(s);
    critical_section();
    up(s);
}
```

entry

exit

Summary…on semaphore

- More on semaphore…it demonstrations an important kind of operations – **atomic operations**.

Definition of atomic operation

- Either none of the instructions of an atomic operation were completed, or
- All instructions of an atomic operation are completed.

- In other words, the entire **up()** and **down()** are indivisible.
  - If it returns, the change must have been made;
  - If it is aborted, no change would be made.

Summary…on critical section problem

- What happened is just the implementation of mutual exclusion (section entry and section exit).

	Comments
Disabling interrupts	Time consuming for multiprocessor systems, sacrifices concurrency.
Strict alternation	Not a good one, busy waiting & violating one mutual exclusion requirement.
Peterson's solution	Busy waiting & has a potential "*priority inversion problem*".
Mutex lock	Busy waiting, often relies on hardware instructions.
Semaphore	**BEST CHOICE**.

Story so far…

- Cooperating processes
  - IPC mechanisms (shared memory, pipes, FIFOs, sockets)
  - Race condition
- Synchronization
  - Mutual exclusion
    - Critical section problem
    - Disabling interrupts, strict alternation, Peterson’s solution, mutex lock, semaphore
- What is next?
  - How to use semaphore to solve classic IPC problems
  - Deadlock