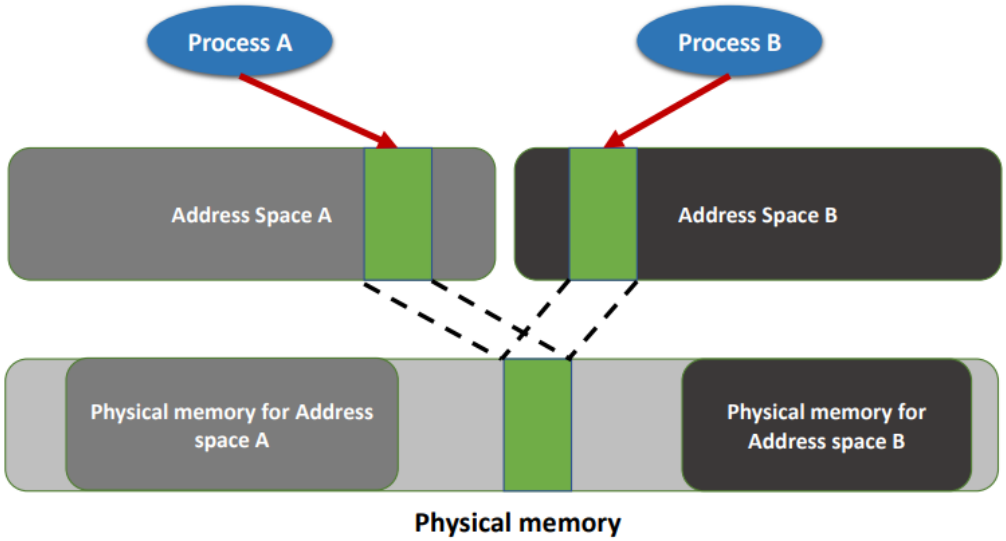
Lecture5 Synchronization

1. Process Communication

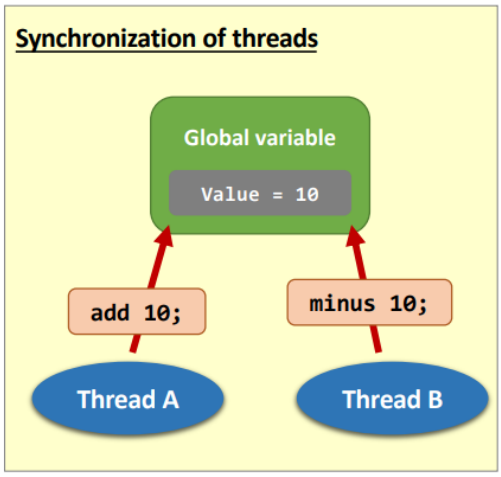
- Process may also need to communicate with each other
  - Information sharing
    - sharing between Android apps
  - Computation speed up
    - Message Passing Interface (MPI)
  - Modularity and isolation
    - Chrome's multi-process architecture

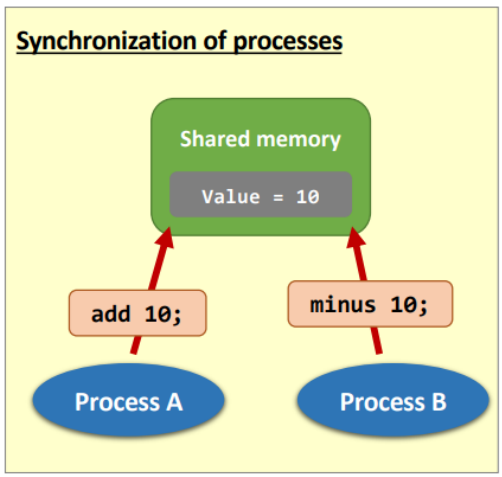
Shared Memory between Process



Synchronization of Threads/Processes

Process and thread synchronization can be considered in similar way





High-level language for Program A

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

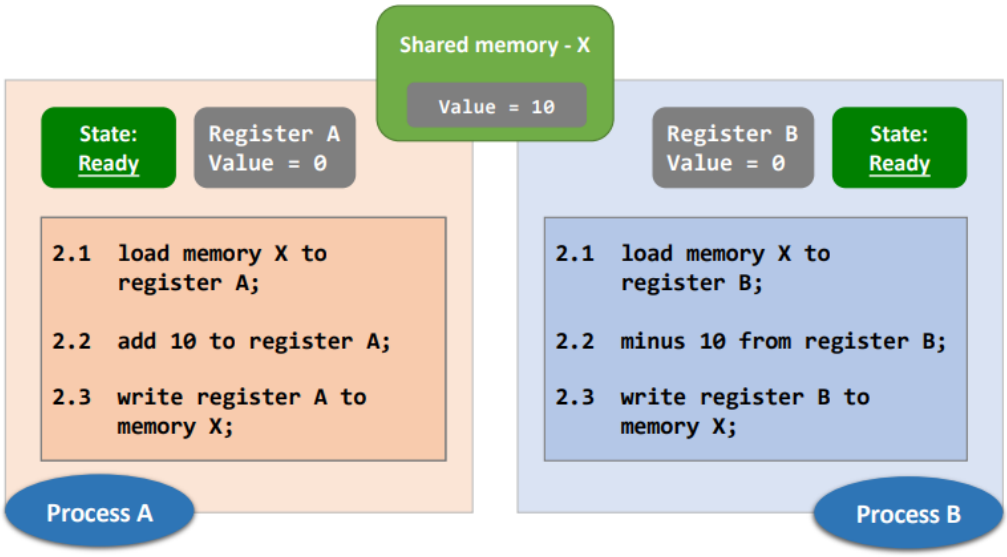
Partial low-level language for Program A

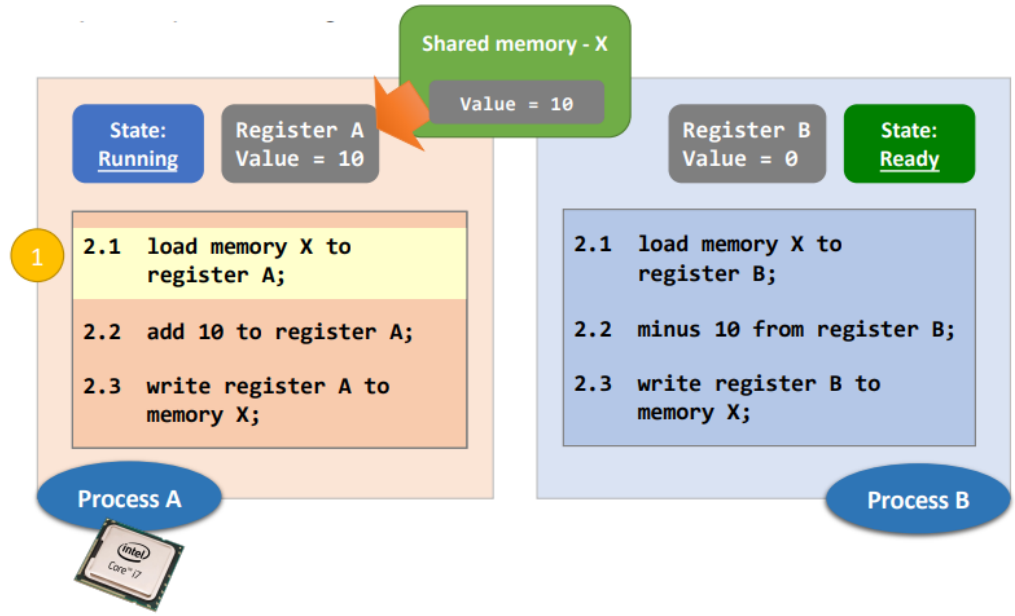
```
attach to the shared memory X;
...
load memory X to register A;
add 10 to register A;
write register A to memory X;
...
exit
```

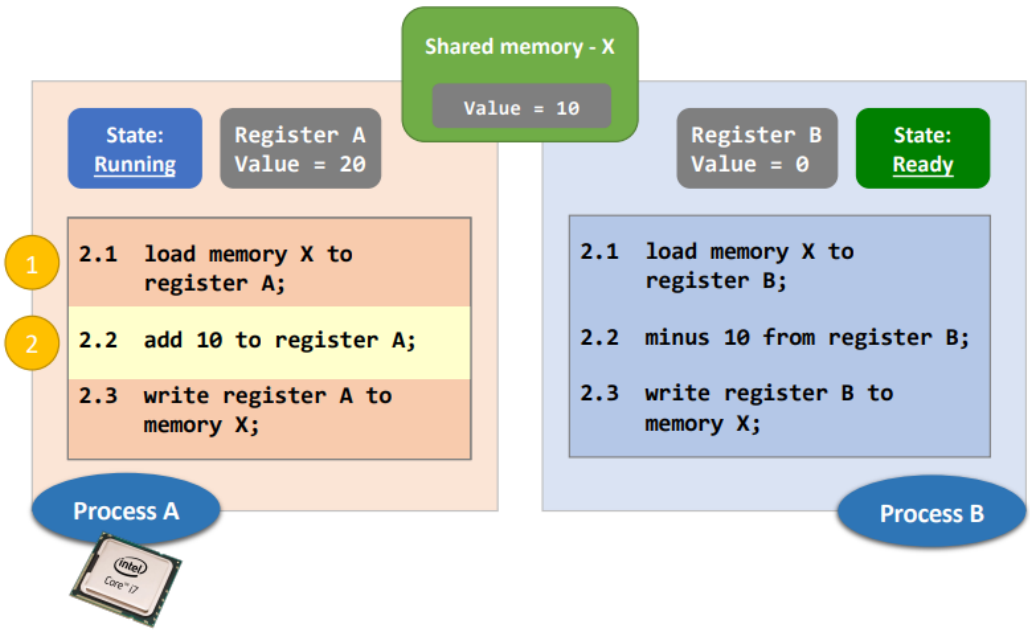
2. Race Condition

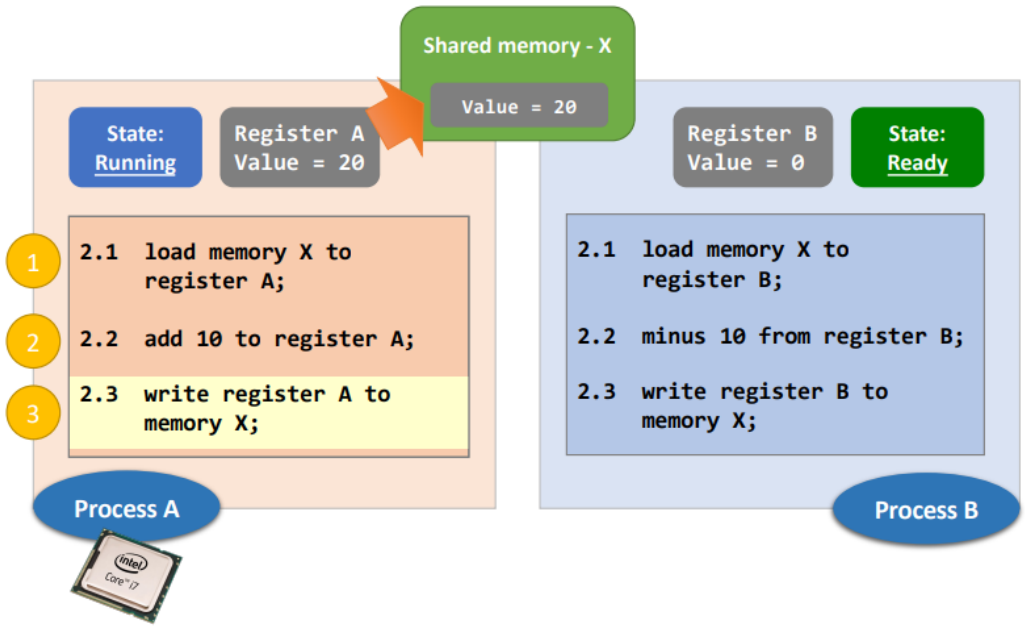
- May happen whenever "**shared object**" + "**multiple processes/threads**" + "**concurrently**"
- A race condition means: The outcome of an execution depends on **a particular order** in which the shared resource is accessed

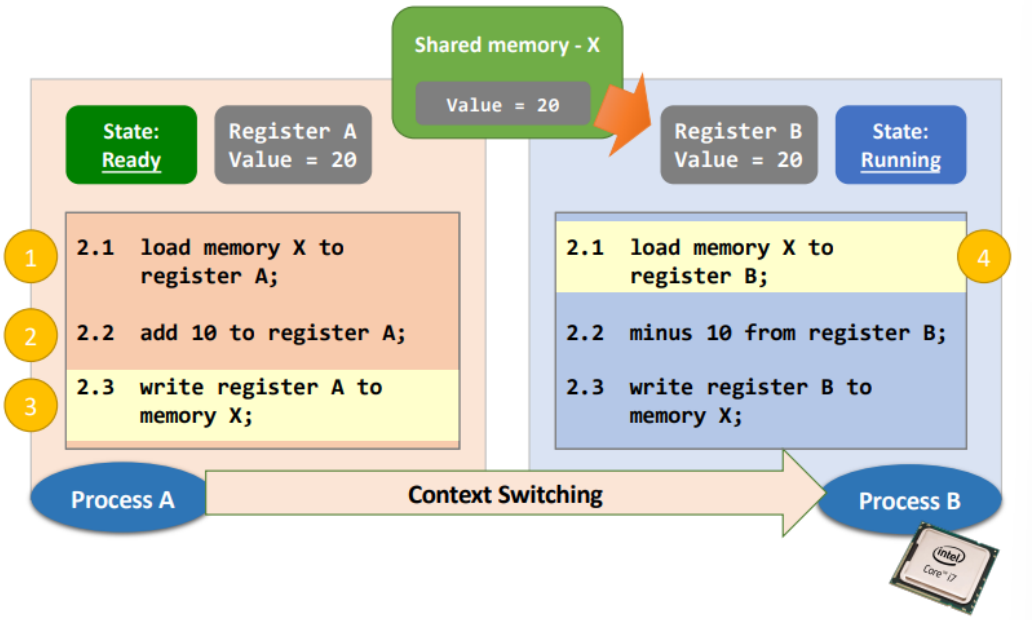
Normal Execution

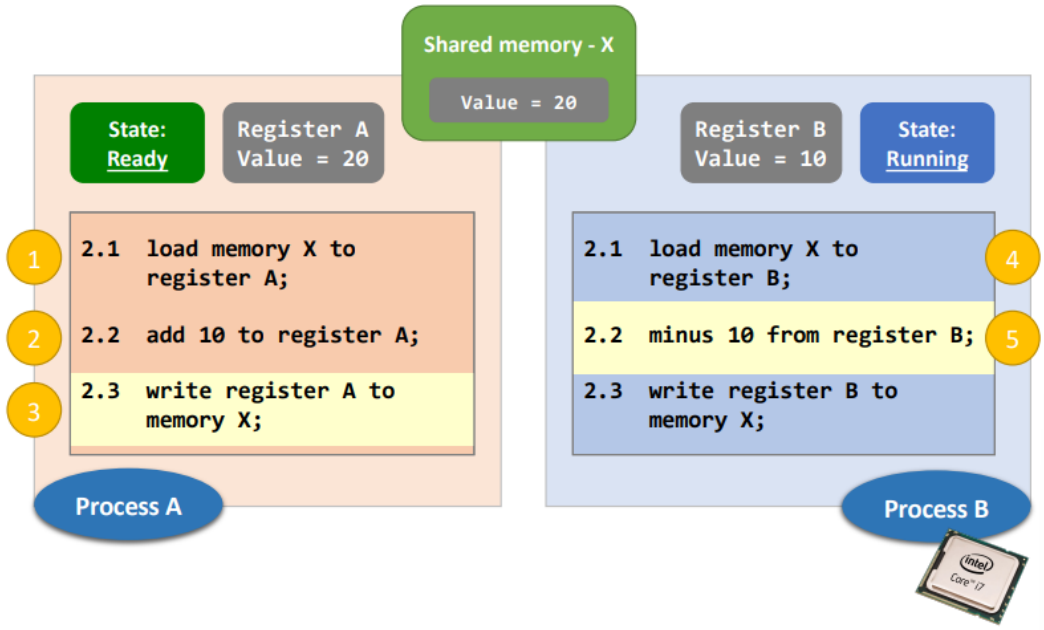


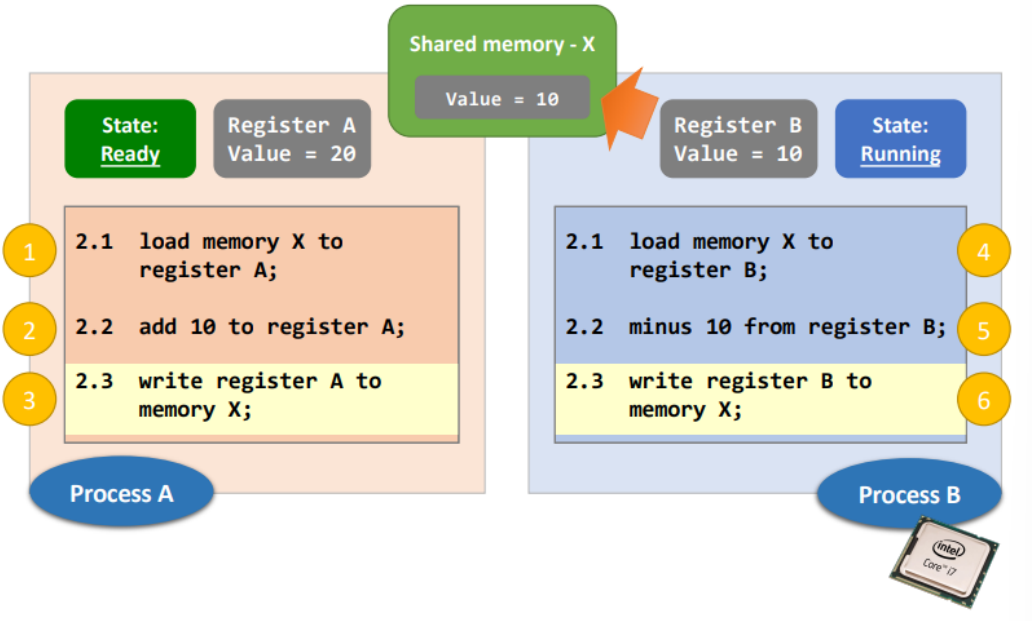

Shared memory - X
Value = 10
State: Running
Register A Value = 10
Register B Value = 0
State: Ready
1
2.1 load memory X to register A;
2.2 add 10 to register A;
2.3 write register A to memory X;
2.1 load memory X to register B;
2.2 minus 10 from register B;
2.3 write register B to memory X;
Process A
Process B


Shared memory - X
Value = 10
State: Running
Register A Value = 20
Register B Value = 0
State: Ready
1
2
2.1 load memory X to register A;
2.2 add 10 to register A;
2.3 write register A to memory X;
2.1 load memory X to register B;
2.2 minus 10 from register B;
2.3 write register B to memory X;
Process A
Process B

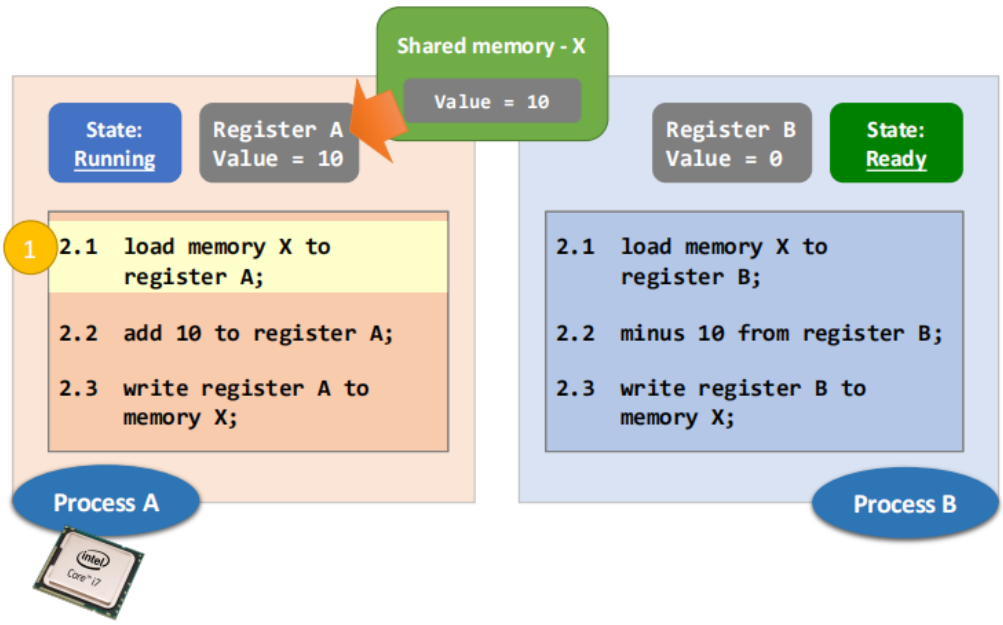

Shared memory - X
Value = 20
State: Running
Register A Value = 20
Register B Value = 0
State: Ready
1
2
3
2.1 load memory X to register A;
2.2 add 10 to register A;
2.3 write register A to memory X;
2.1 load memory X to register B;
2.2 minus 10 from register B;
2.3 write register B to memory X;
Process A
Process B

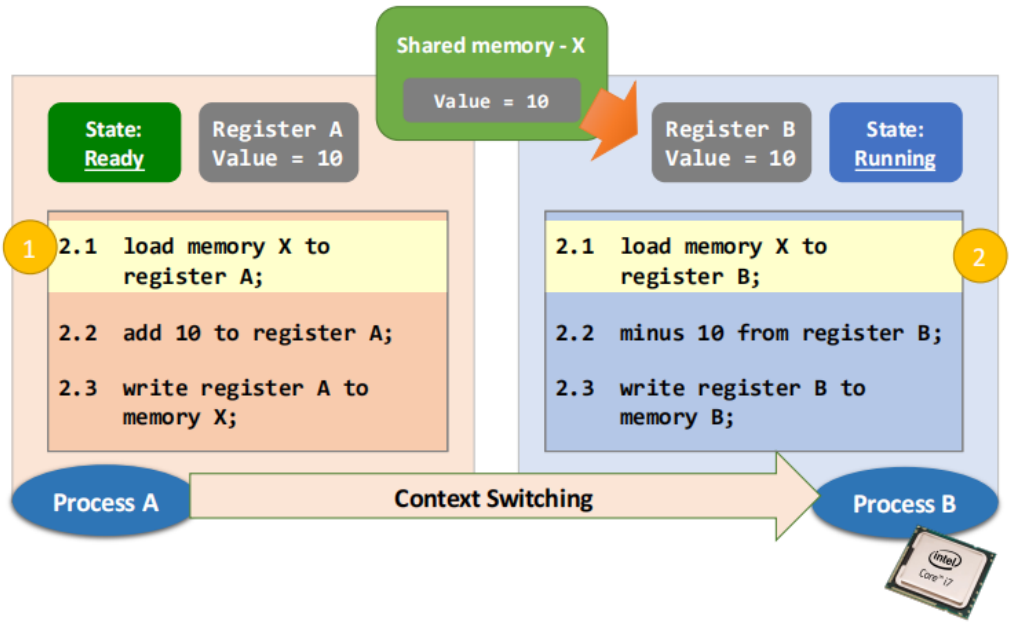

Shared memory - X
Value = 20
State: Ready
Register A Value = 20
Register B Value = 20
State: Running
1
2.1 load memory X to register A;
2
2.2 add 10 to register A;
3
2.3 write register A to memory X;
2.1 load memory X to register B;
4
2.2 minus 10 from register B;
2.3 write register B to memory X;
Process A
Context Switching
Process B

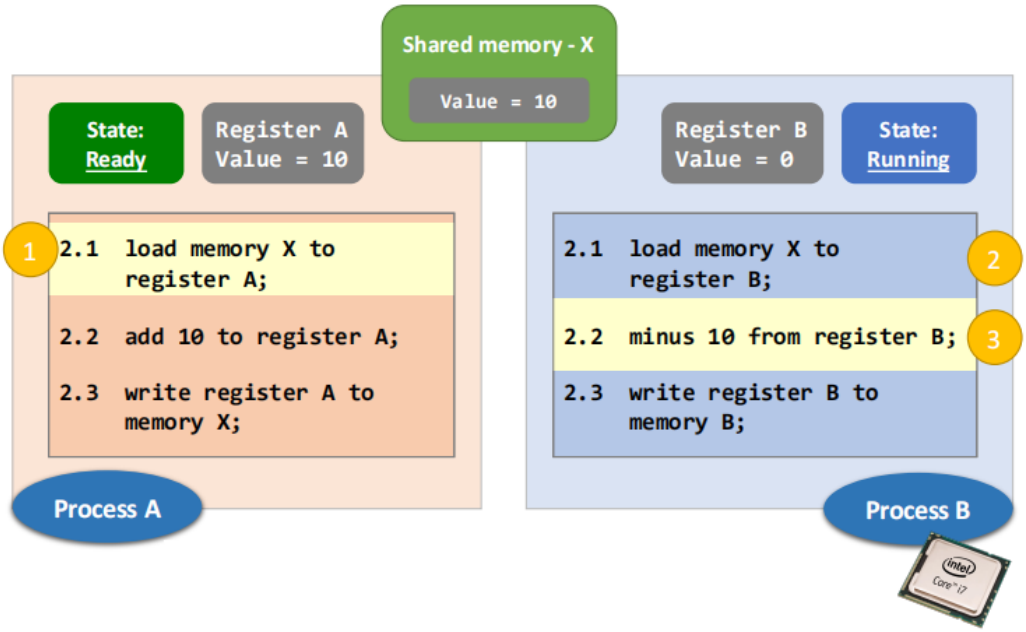

Shared memory - X
Value = 20
State: Ready
Register A Value = 20
Register B Value = 10
State: Running
1
2.1 load memory X to register A;
2
2.2 add 10 to register A;
3
2.3 write register A to memory X;
2.1 load memory X to register B;
4
2.2 minus 10 from register B;
5
2.3 write register B to memory X;
Process A
Process B

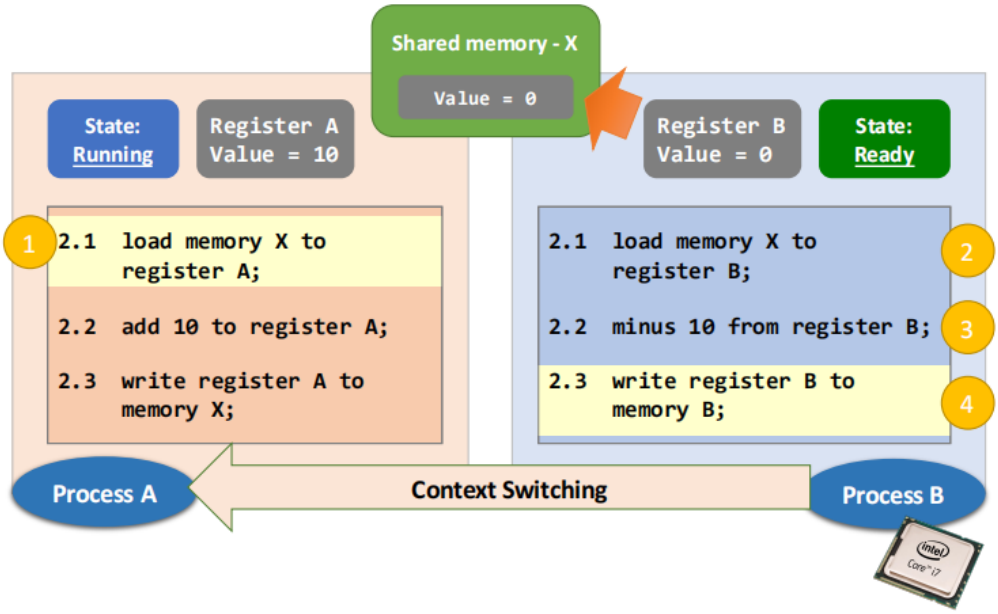

Shared memory - X
Value = 10
State: Ready
Register A Value = 20
Register B Value = 10
State: Running
1
2.1 load memory X to register A;
2
2.2 add 10 to register A;
3
2.3 write register A to memory X;
2.1 load memory X to register B;
4
2.2 minus 10 from register B;
5
2.3 write register B to memory X;
6
Process A
Process B

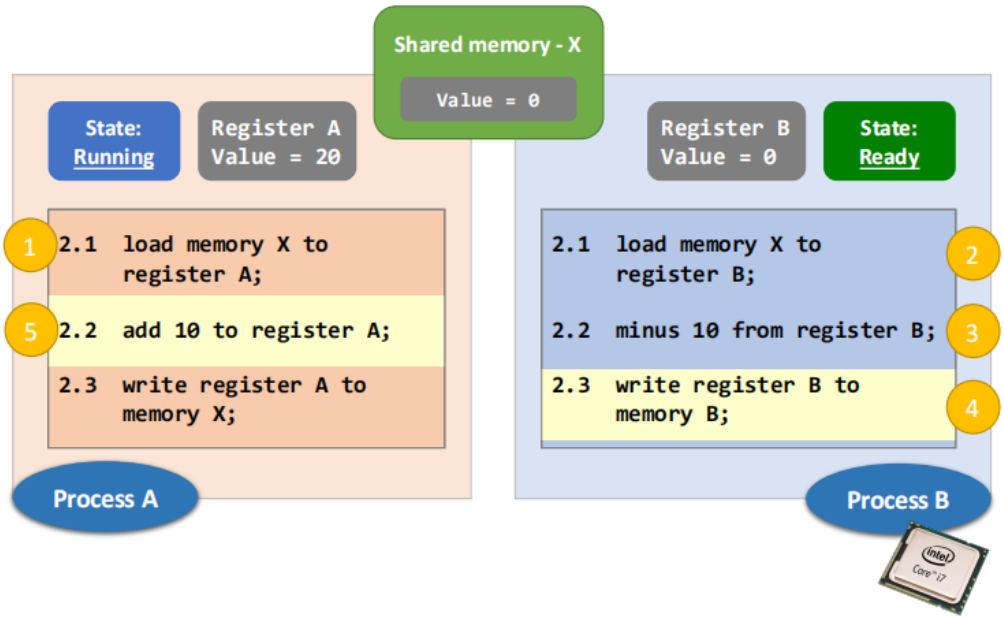
Abnormal Execution

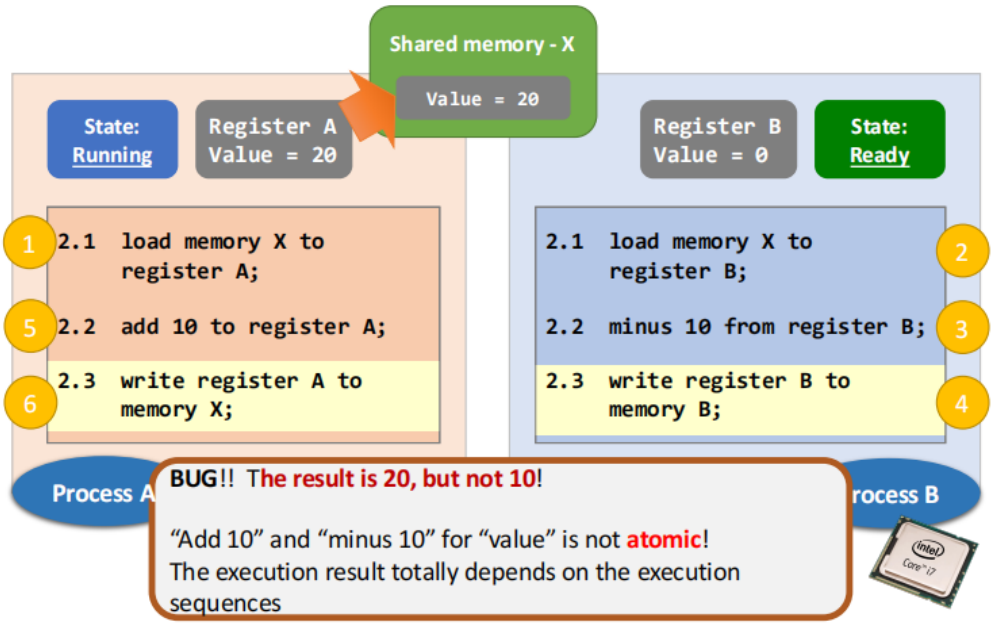







Shared memory - X
Value = 0
State: Running
Register A Value = 10
Register B Value = 0
State: Ready
1
2.1 load memory X to register A;
2.2 add 10 to register A;
2.3 write register A to memory X;
2.1 load memory X to register B;
2.2 minus 10 from register B;
2.3 write register B to memory B;
2
3
4
Process A
Context Switching
Process B

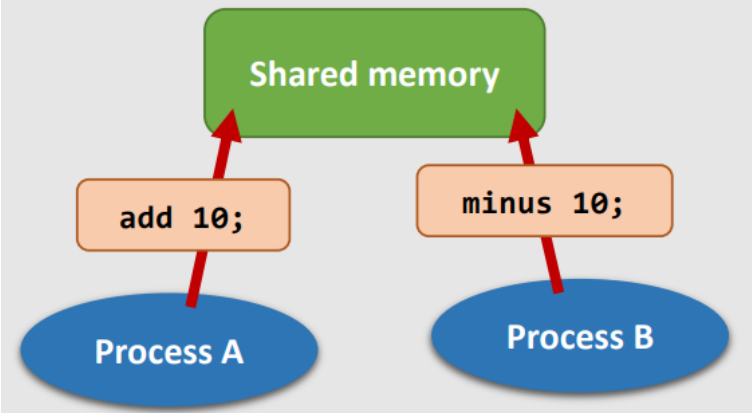

Shared memory - X
Value = 0
State: Running
Register A Value = 20
Register B Value = 0
State: Ready
1
2.1 load memory X to register A;
5
2.2 add 10 to register A;
2.3 write register A to memory X;
2.1 load memory X to register B;
2.2 minus 10 from register B;
2.3 write register B to memory B;
2
3
4
Process A
Process B


Shared memory - X
Value = 20
State: Running
Register A Value = 20
Register B Value = 0
State: Ready
1
2.1 load memory X to register A;
5
2.2 add 10 to register A;
6
2.3 write register A to memory X;
2.1 load memory X to register B;
2.2 minus 10 from register B;
2.3 write register B to memory B;
2
3
4
Process A
rocess B
BUG!! The result is 20, but not 10!
"Add 10" and "minus 10" for "value" is not atomic!
The execution result totally depends on the execution sequences

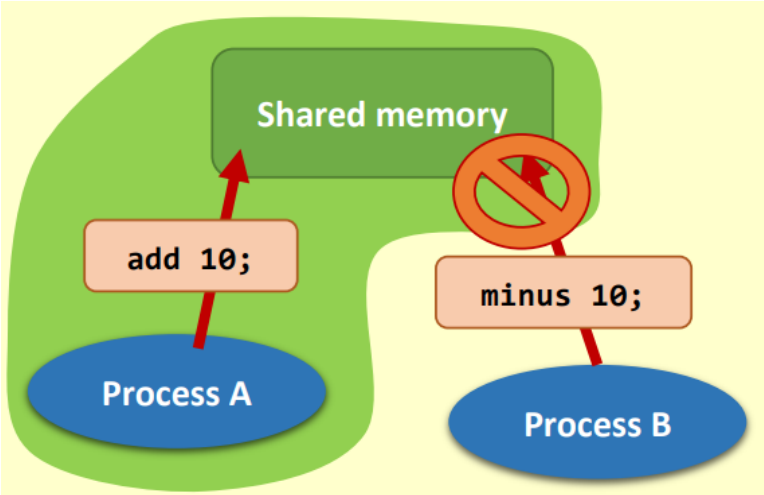
3. Critical Section

Mutual Exclusion

How to resolve race condition?



Solution: **mutual exclusion**

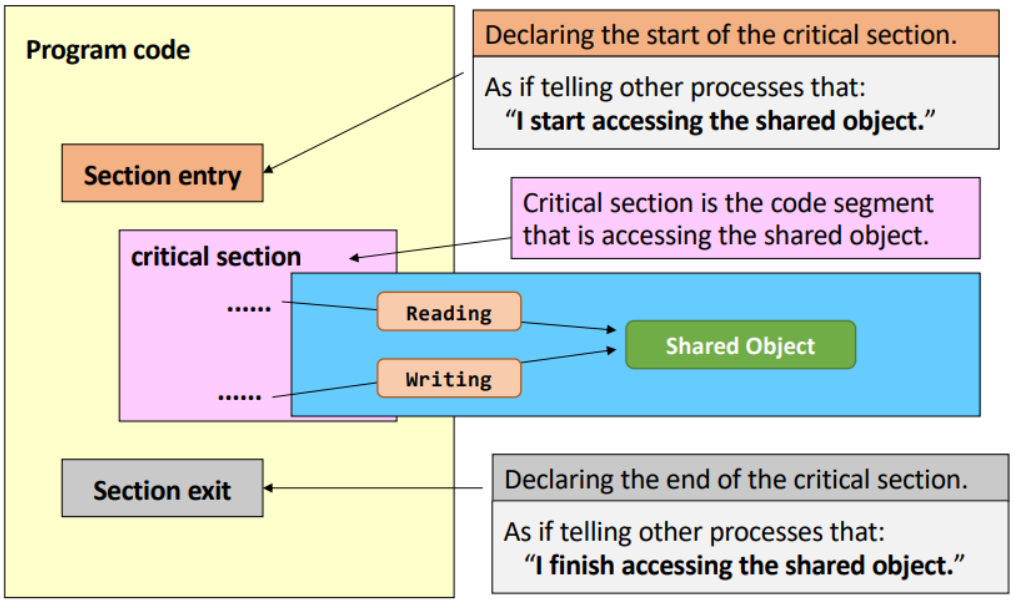


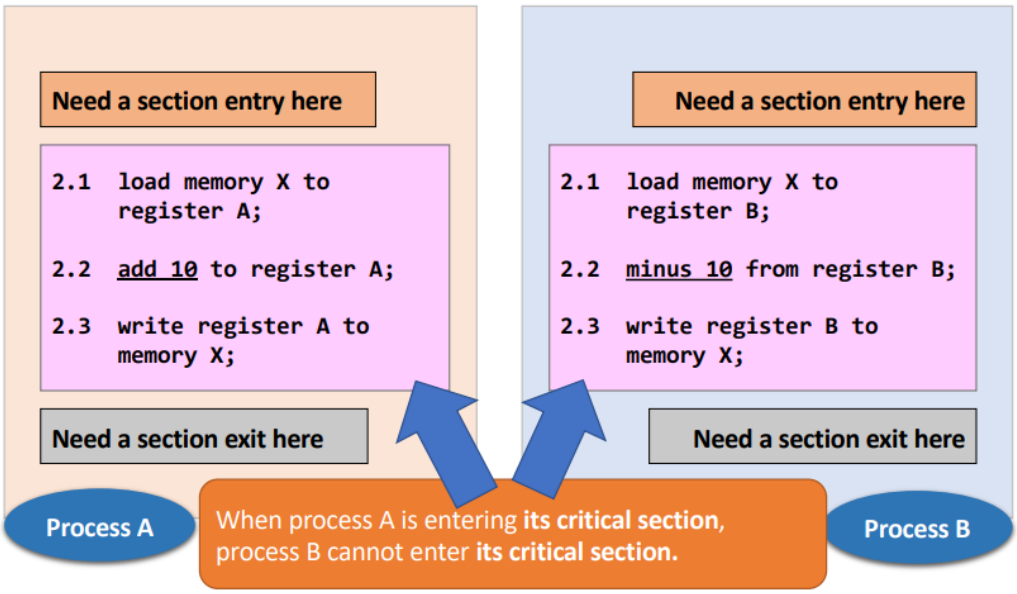
- When one process access the shared memory, no one else could access it
- A set of processes would not have the problem of race condition if **mutual exclusion is guaranteed**
- Shared object is still sharable, but
- Do not access the "shared object" **at the same time**

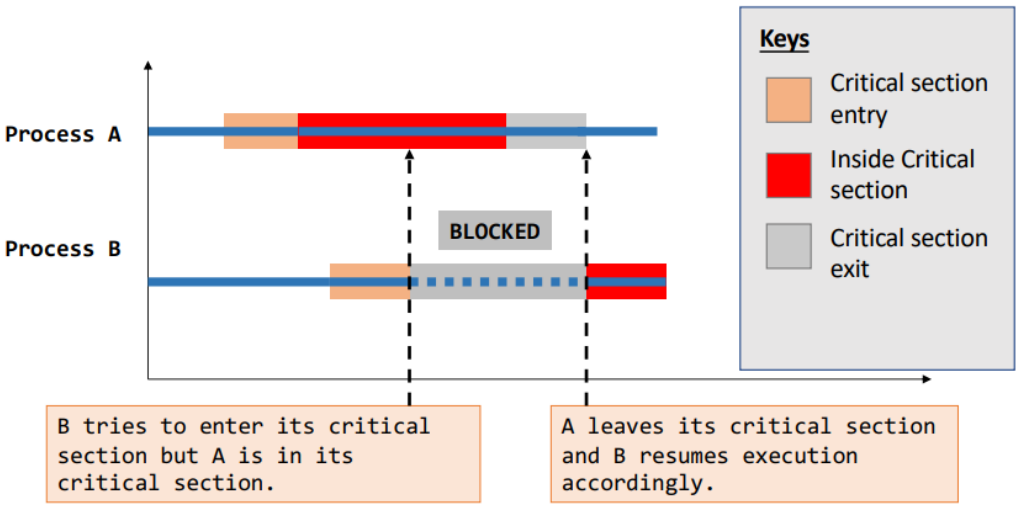
Definition

A critical section is the **code segment** that access **shared objects**

- Critical section should be **as tight as possible**
- Note that one critical section can be designed for accessing **more than one shared objects**


Program code
Declaring the start of the critical section.
As if telling other processes that:
"I start accessing the shared object."
Section entry
Critical section is the code segment that is accessing the shared object.
critical section
.....
Reading
Shared Object
.....
Writing
Section exit
Declaring the end of the critical section.
As if telling other processes that:
"I finish accessing the shared object."


Need a section entry here
2.1 load memory X to register A;
2.2 add 10 to register A;
2.3 write register A to memory X;
Need a section exit here
Need a section entry here
2.1 load memory X to register B;
2.2 minus 10 from register B;
2.3 write register B to memory X;
Need a section exit here
Process A
When process A is entering its critical section, process B cannot enter its critical section.
Process B


Process A
Process B
BLOCKED
Keys
Critical section entry
Inside Critical section
Critical section exit
B tries to enter its critical section but A is in its critical section.
A leaves its critical section and B resumes execution accordingly.

Implementation Requirements

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

4. Solution1: Disabling Interrupts

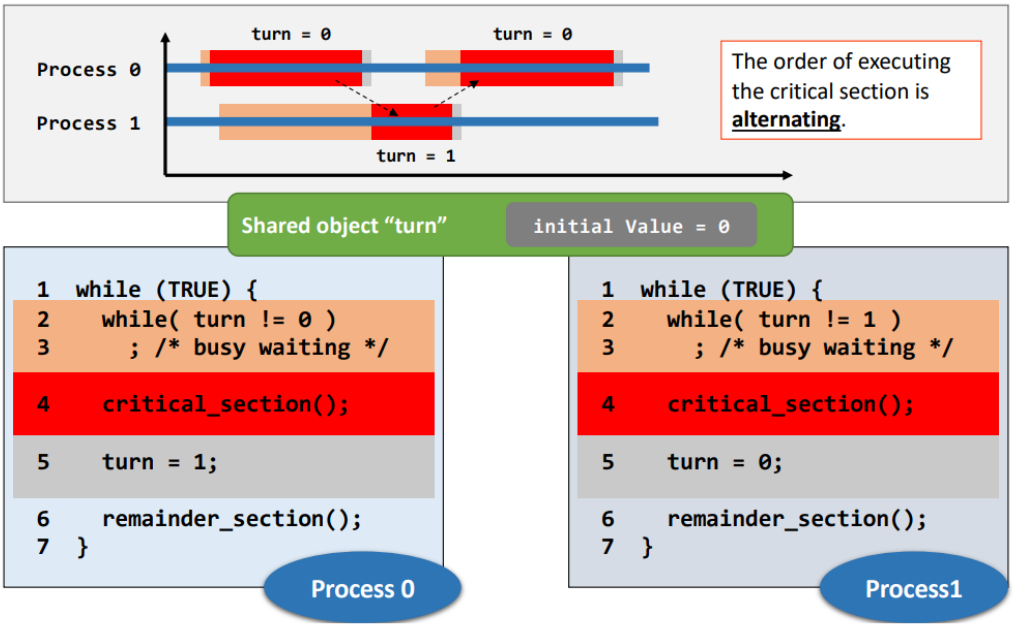
- **Disabling interrupts** when the process is inside the critical section
- When a process is in its critical section, no other processes could be able to run
- **Uni-core**: Correct but **not permissible**
    - User level: what if one enters a critical section and loops infinitely?
    - OS **cannot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

5. Solution2: 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

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

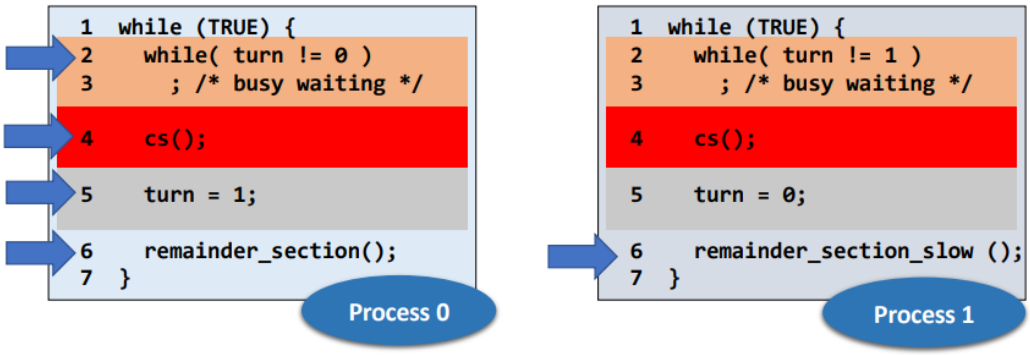


- Correct but waste CPU resources
- Impose a "strict alternating" order
  - Sometimes you give me my turn but I'm not ready to enter critical section yet

Progress Violation

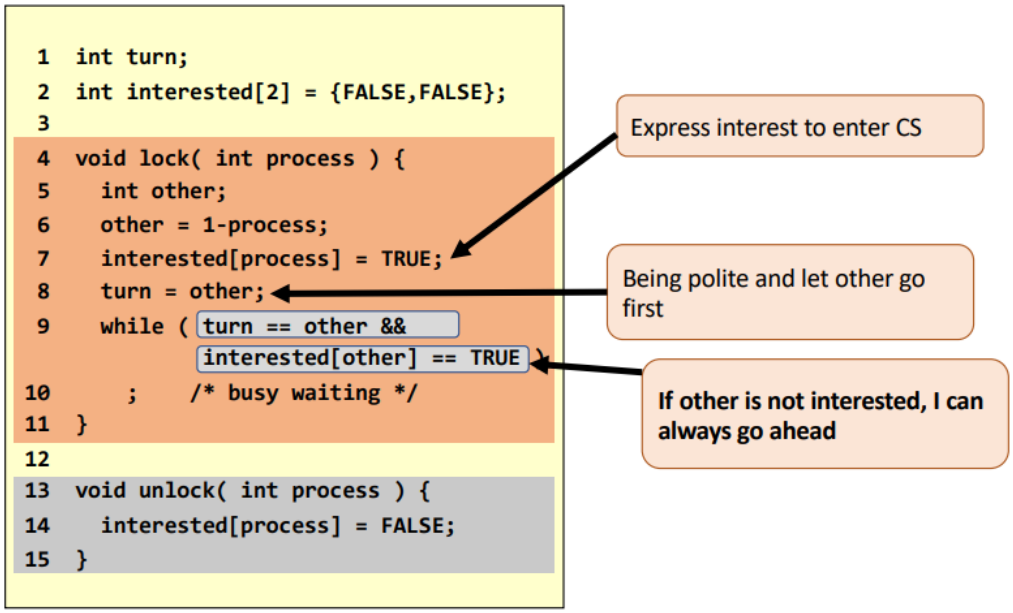
Consider the following sequence

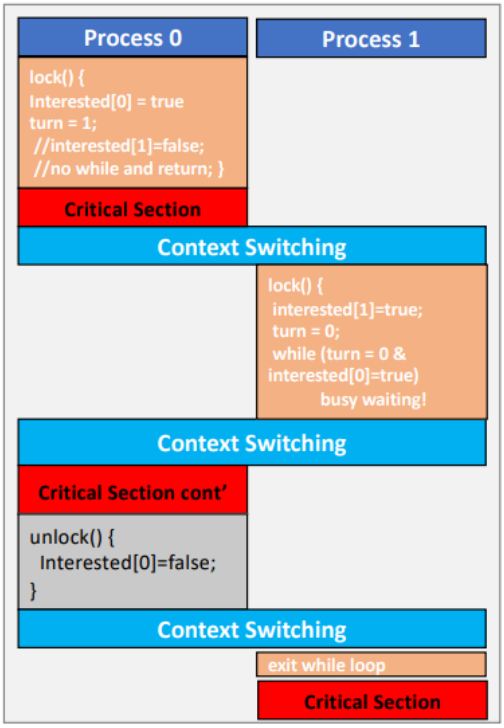
- Process0 leaves `cs()`, set turn=1
- Process1 enters `cs()`, leaves `cs()`, set turn=0, work on `remainder_section_slow()`
- Process0 loops back and enters `cs()` again, leaves `cs()`, set turn=1
- Process0 finishes its `remainder_section()`, go back to top of the loop
  - It can't enter its `cs()`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

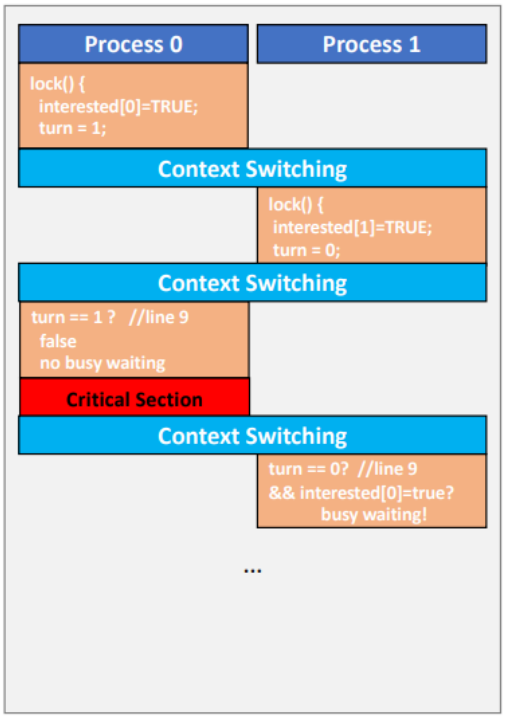


Peterson's Solution: Improved Spin-based Locks

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

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

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







- Mutual exclusion
  - interested [0] == interested [1] == true
  - turn == 0 or turn == 1, not both
- Progress
  - If only P0 to enter critical section
    - interested[1] == false, thus P0 enters critical section
  - If both P0 and P1 to enter critical section
    - interested[0] == interested[1] == true and (turn == 0 or turn == 1)
    - One of P0 and P1 will be selected

- Bounded-waiting
    - If both P0 and P1 to enter critical section, and P0 selected first
    - When P0 exit, interested [0] = false
        - If P1 runs fast: interested[0] == false, P1 enters critical section
        - If P0 runs fast: interested[0] = true, but turn = 0, P1 enters critical section

Multi-Process Mutual Exclusion

```
// Initial Setting
bool waiting[n] = false;
int lock = 0;

do{
    waiting[i] = true;
    key = true;
    while(waiting[i] && key)
        key = compare_and_swap(&lock, 0, 1);
    waiting[i] = false;

    /* critical section */
        ///...
    /* critical section */

    j = (i + 1) % n;
    while((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
        lock = 0;
    else
        waiting[j] = false;
} while (true)

void test_and_set(&lock){
    bool rv = *lock;
    *lock = true;
    return rv;
}

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

```
        *value = new_value;
    return temp;
}
```

Mutual Exclusion

- Initialize
  - The elements in the `waiting[n]` array (n processes) are initialized to `false`
  - The `lock` is initialized to `0`
- The first process `P[i]` executes the `compare_and_swap()` will find `key = 0`
  - Since then, `lock` is modified to be `1` and the rest of the process will find `lock != 0` an therefore they will return `key = 1`, which will block them to the `while(waiting[i] && key)` cycle
- `P[i]` enters the critical section. Before then, it sets the `waiting[i] = false`. After finishes the critical section, it will perform the following two cases:
  - If there are other processes waiting for entering the critical section, `P[i]` will pick next process `p[j]` and set `waiting[j] = false` to let `P[j]` enter the critical section.
    - Then `p[j]` will execute the same process as `p[i]`
  - If there is no other process, `p[i]` will set `lock = false`
    - If later any other process comes, one of them will become the first to come and find the `lock = 0` and hold the `key = 1`.

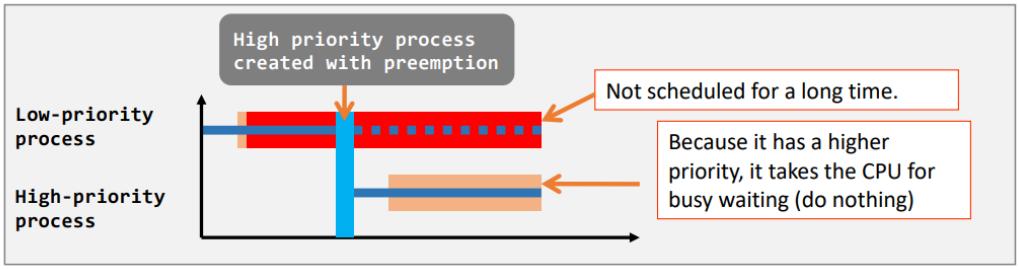
Progress

- A process exiting the critical section either sets `lock` to `0` or sets `waiting[j]` to `false`.
- Both allow a process that is waiting to enter its critical section to proceed

Bounded-waiting

- When a process leaves its critical section, it scans the array waiting in the cyclic ordering $(i+1, i+1, \ldots, n-1, 0, \ldots i-1)$.
- It designates the first process in this ordering that is in the entry section (`waiting[j] == true`) as the next one to enter the critical section
- Any process waiting to enter its critical section will thus do so within `n - 1` turns

Priority Inversion



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

Sleep-based locks - 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)

```
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 }
```

Initialize **s->value** = 1

"**sem_wait(s)**"

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

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

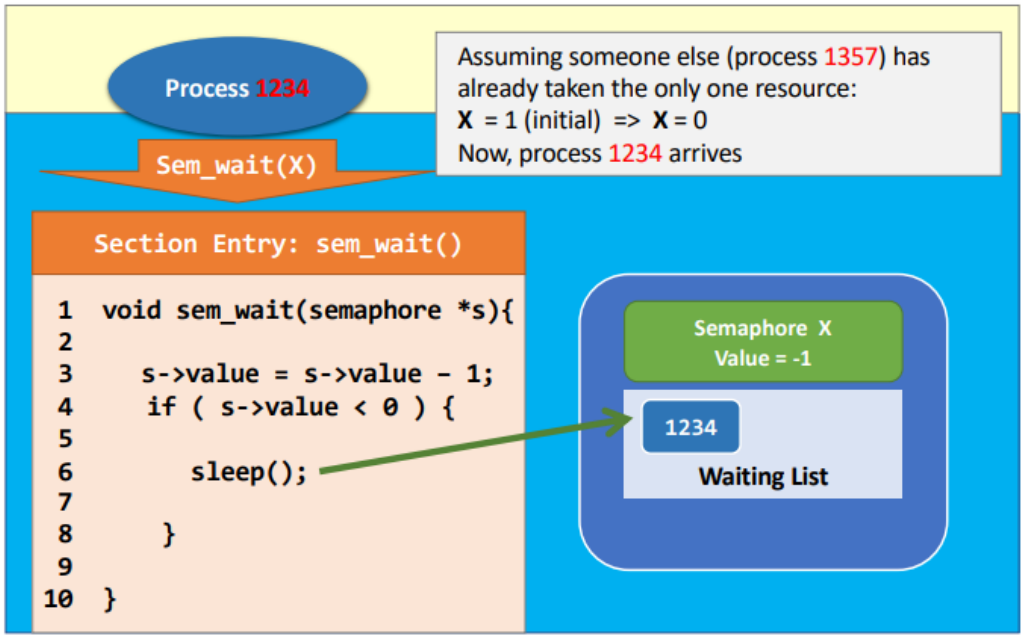
"**sem_post(s)**"

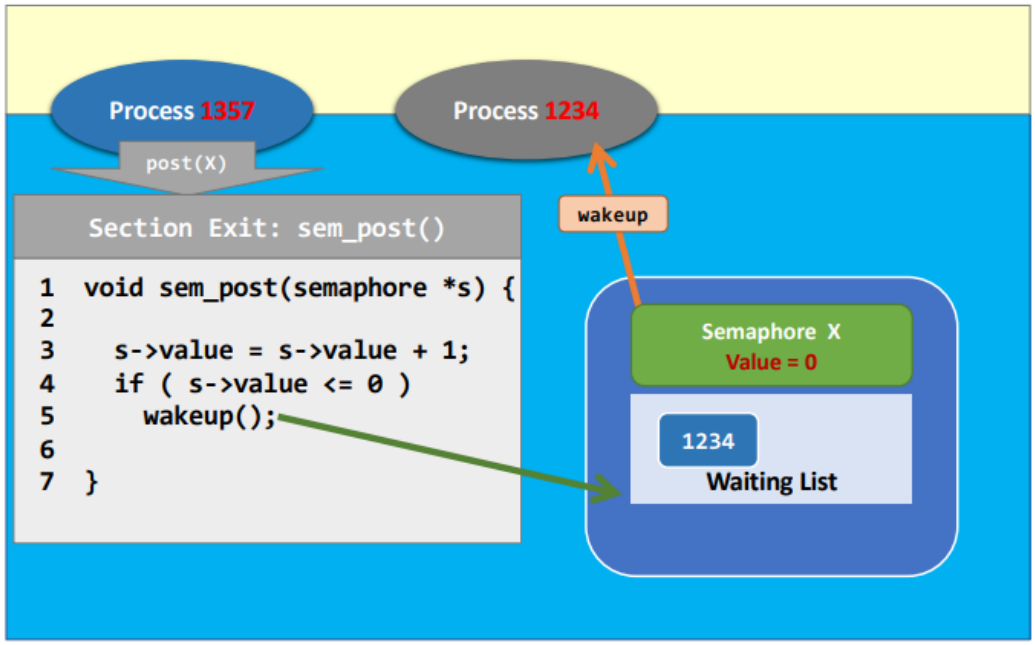
- I notify the others (if anyone waiting) that **s->value <= 0**

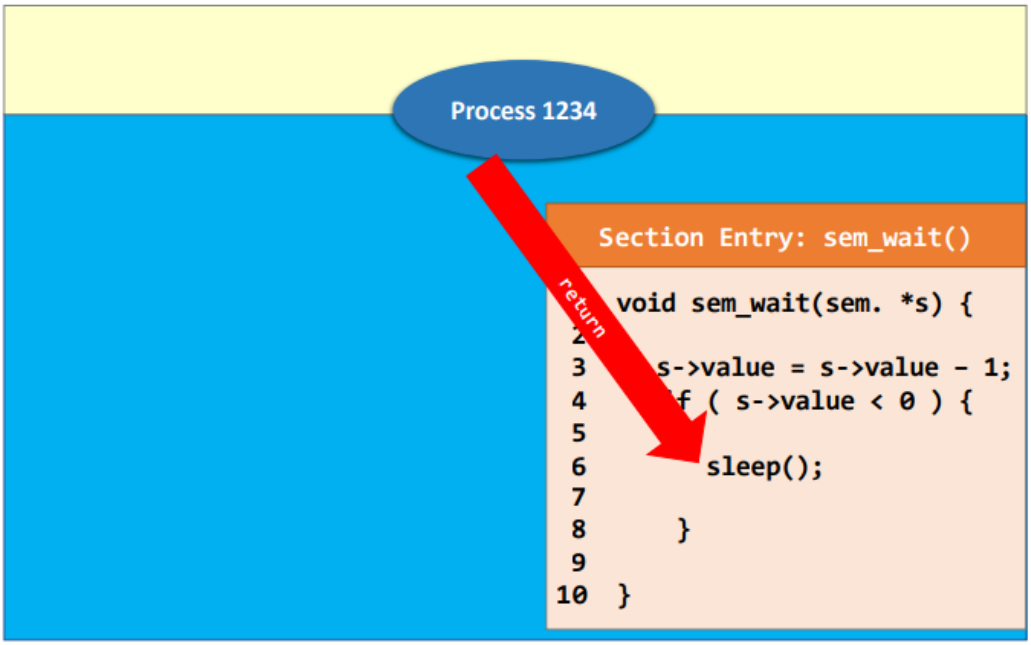
Section Exit: sem_post()

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

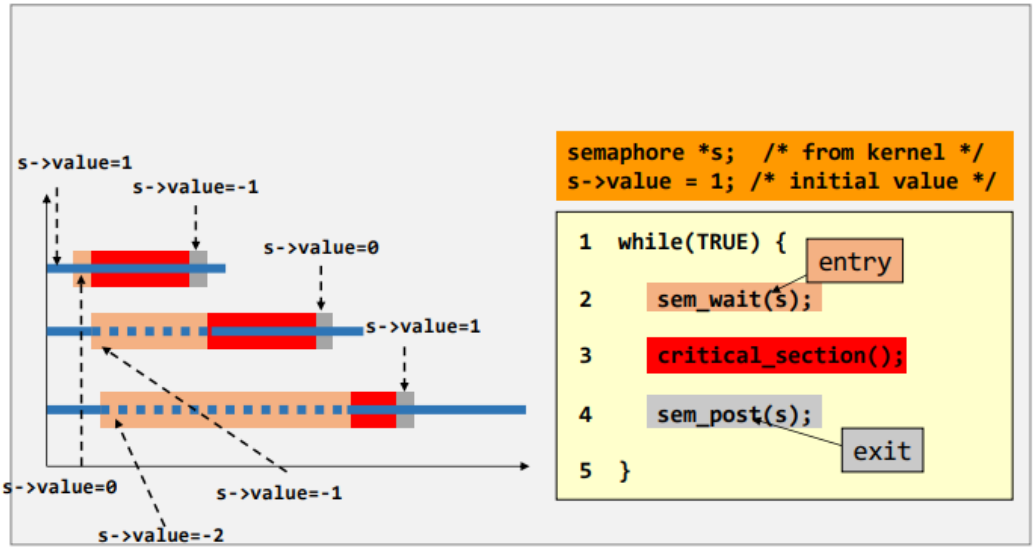
Example







Using Semaphore in User Process



Implementation

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

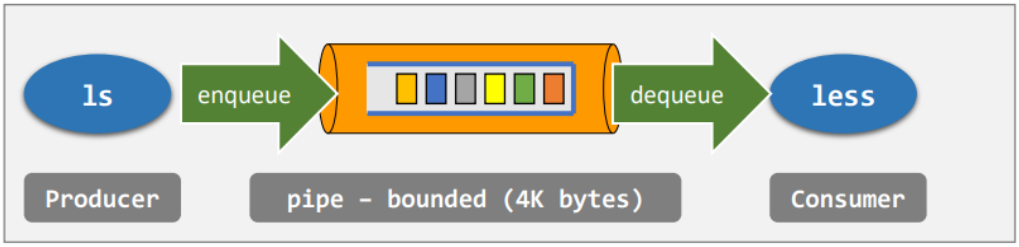
```
///one single instruction with the following
///semantics
void cmp_xchg(int *addr, int expected_value, int new_value)
{
    int temp = *addr;
    if(*addr == expected_value)
    *addr = new_value;
    return temp;
}
```

Example: Atomic increment

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

6. Using Semaphore beyond Mutual Exclusion

Producer-Consumer Problem / Bounder-Buffer Problem



Element	Description
A bounded buffer	A shared object Size is bounded, say N slots Is a qqueue

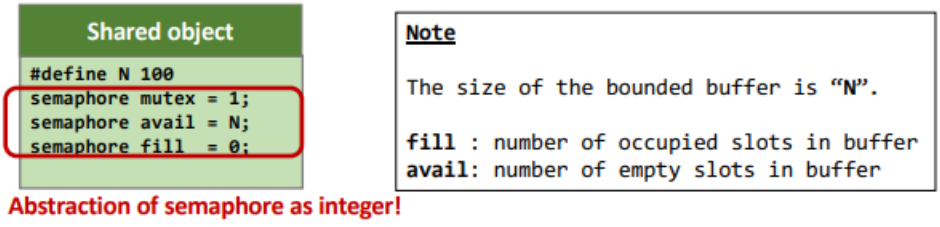
Element	Description
A producer process	Produce a unit of data **Writes** a piece of data to the **tail** of the buffer at one time
A consumer process	**Remove** a unit of data from the **head** of the bounded buffer at one time

- When the **producer** wants to put a new item in the buffer, **but the buffer is already full**. Then, the producer should wait. The consumer should **notify** the producer after she has dequeued an item.
- When the **consumer** wants to consumes an item from the buffer, **but 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



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

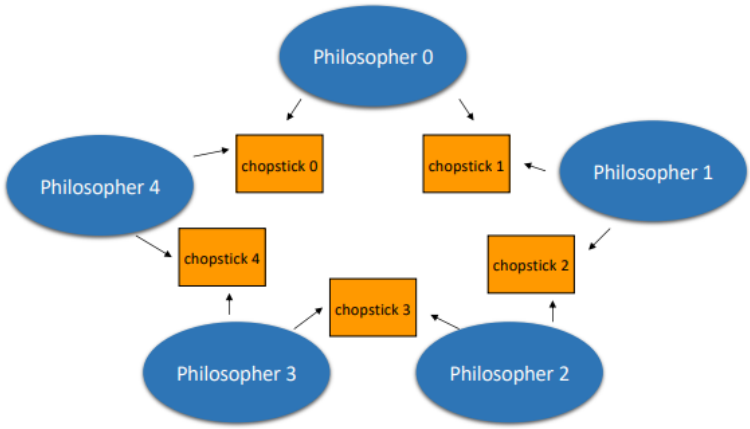
```
1  void producer(void) {
2      int item;
3
4      while(TRUE) {
5          item = produce_item();
6          wait(&avail);
7          wait(&mutex);
8          insert_item(item);
9          post(&mutex);
10         post(&fill);
11     }
12 }
```

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 }
```

- necessary to use both `avail` and `fill`: may cause NULL problem
- can not swap `avail` and `mutex` in the proucer: may cause **deadlock**

Dining Philosopher Problem

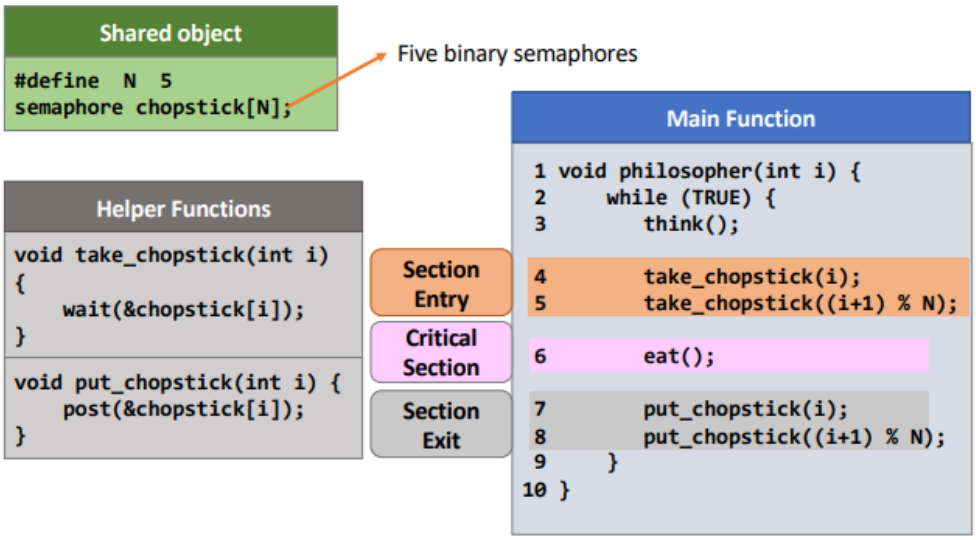


- 5 philosophers, 5 plates of spaghetti, and 5 chopsticks
- The jobs of each philosopher are **to think** and **to eat**
- They need **exactly two chopsticks** in order to eat the spaghetti
- How to construct a synchronization protocol such that they
  - will not **starve to death**
  - will not result in any **deadlock scenarios**

Requirement 1 Mutual Exclusion

Requirement	Description	Solution

Requirement	Description	Solution
Mutual Exclusion	While you are eating, people cannot steal your chopstick Two persons cannot hold the same chopstick	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



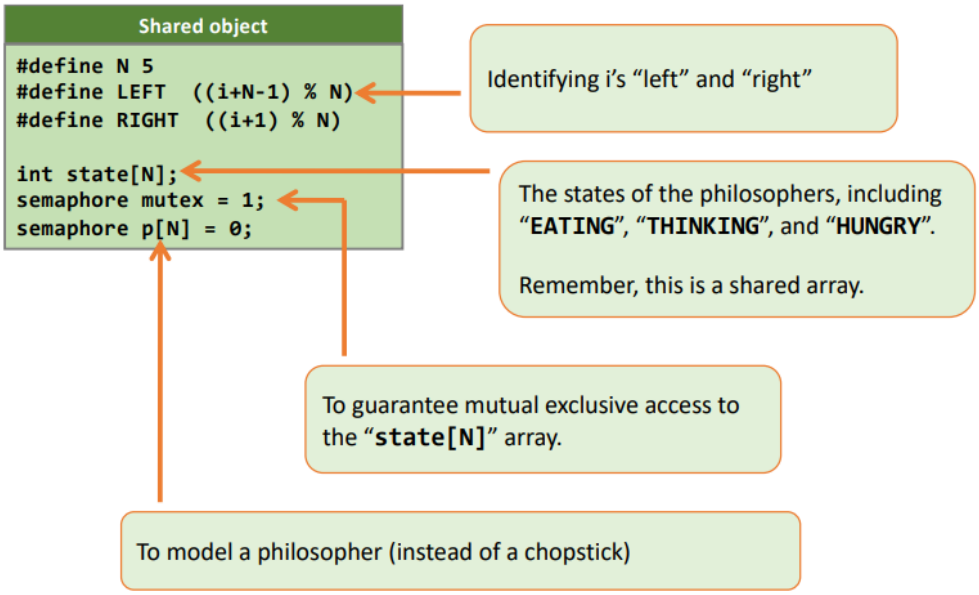
Requirement2 Synchronization

Requirement	Description	Solution
Synchronization	Should avoid **deadlock**	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

- 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, ...

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  }
```

```
void wait(semaphore *s) {

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

    sleep();

  }

}
```

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  }
```

Section exit

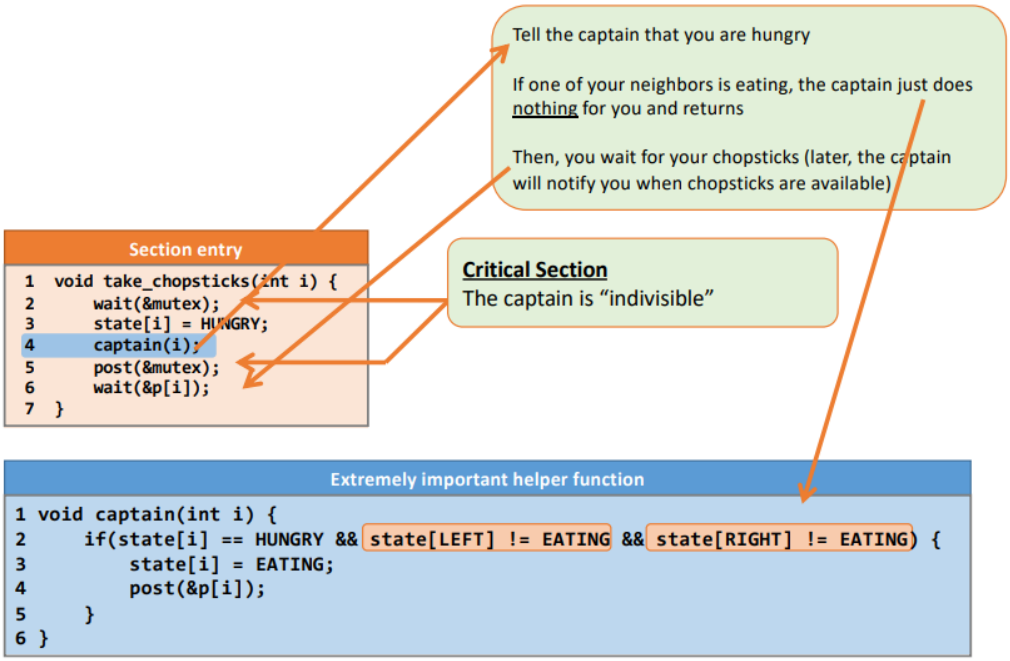
```
1  void put_chopsticks(int i) {
2      wait(&mutex);
3      state[i] = THINKING;
4      captain(LEFT);
5      captain(RIGHT);
6      post(&mutex);
7  }
```

```
void post(semaphore *s) {

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

}
```

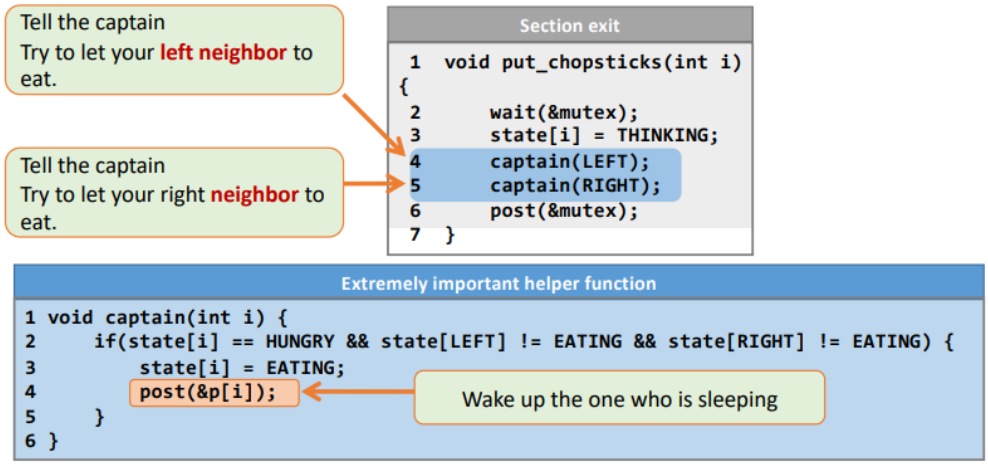
Extremely important helper function

```
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 }
```

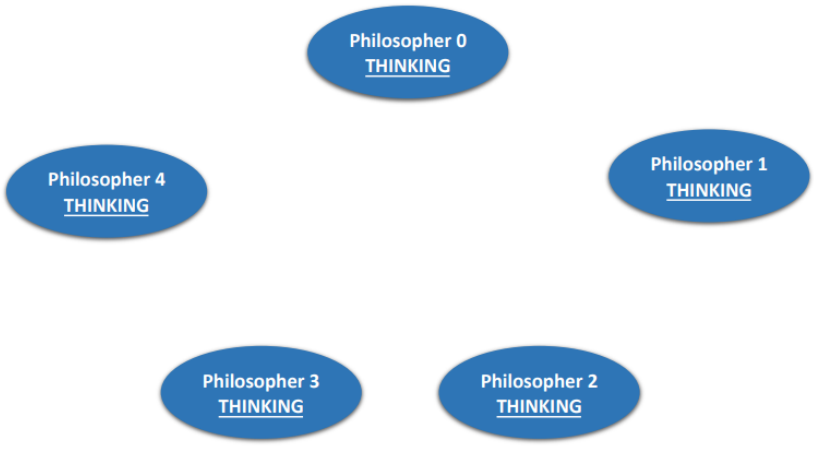
Hungry

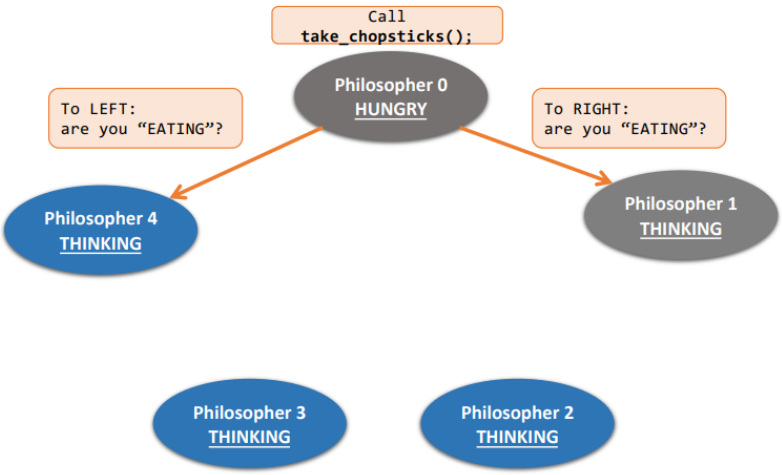
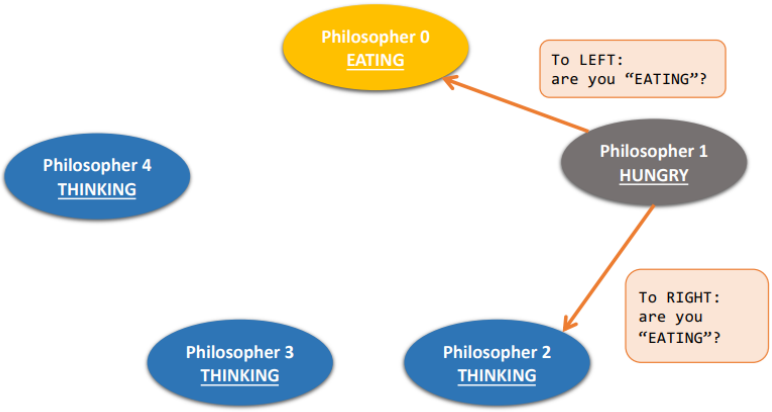
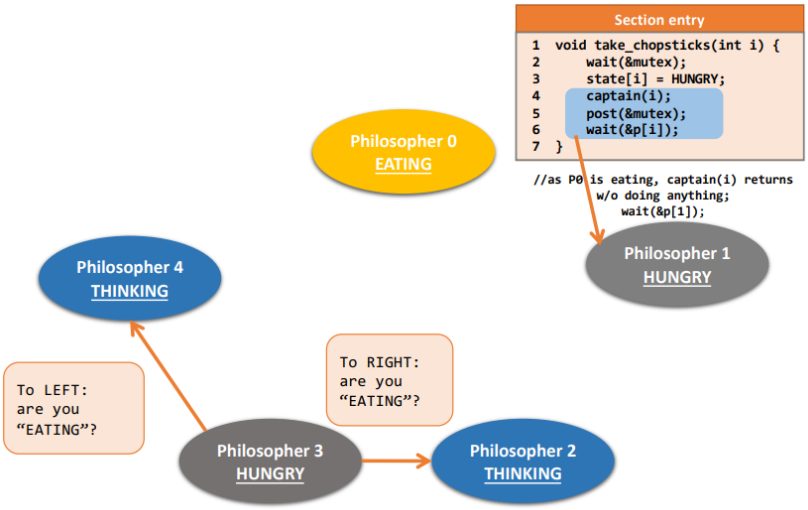
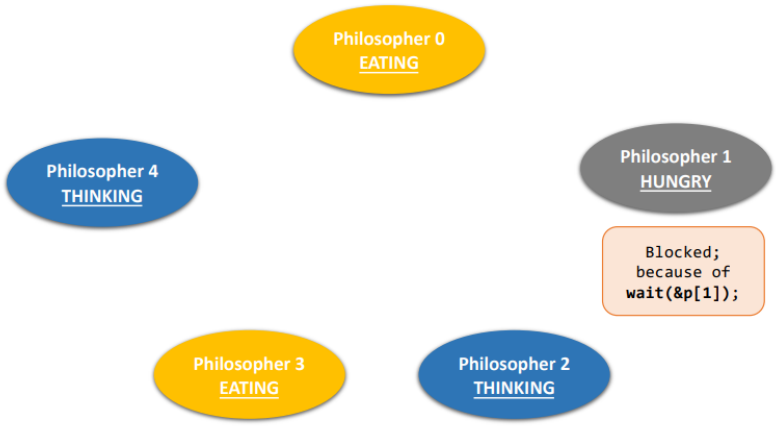


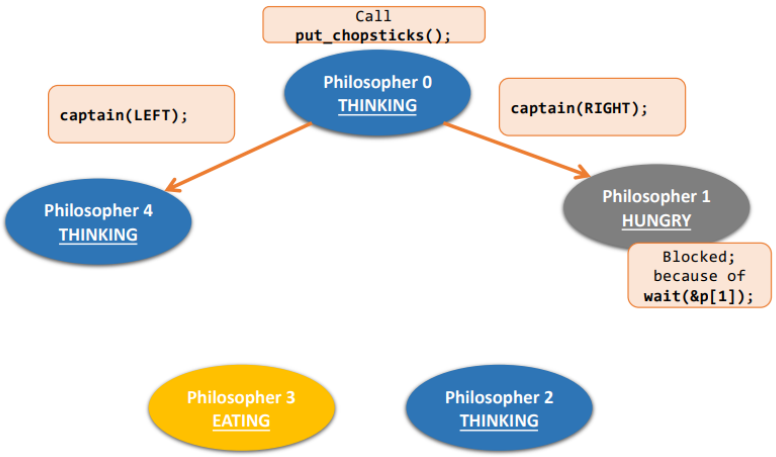
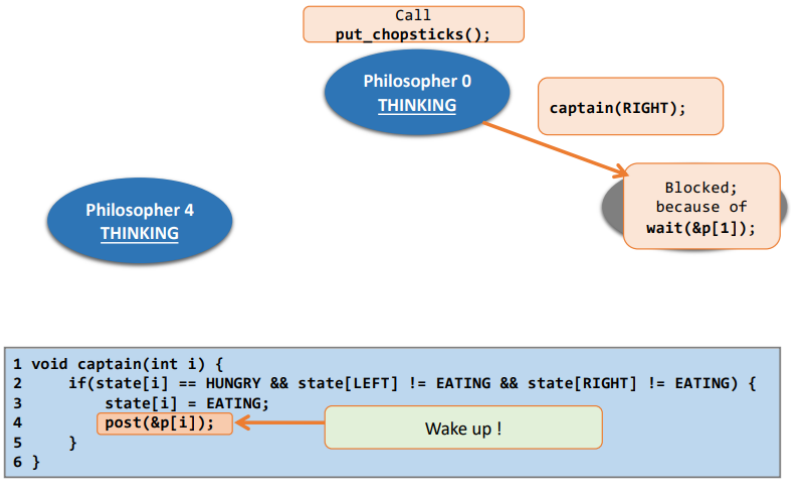
Finish eating



An illustration: How can Philosopher 1 start eating?

Sequence	Illustration
1	

Sequence	Illustration
2	
3	
4	
5	

Sequence	Illustration
6	
7	
8