Operating Systems
CS 415

Lecture 9:
Synchronization

UNIVERSITY OF OREGON

Suyash Gupta
Assistant Professor
Distopia Labs and ONRG
Dept. of Computer Science
(E) suyash@uoregon.edu
(W) gupta-suyash.github.io

Announcements

- **Suyash Gupta**
  - Office Hours: Thursday, 10-11am, Deschutes 334

- **Nihal Balivada (TA)**
  - Office Hours: Wednesday/ Friday, 11-12pm, Deschutes 335

- **Ranjitha Rani (TA)**
  - Office Hours: Monday /Tuesday, 1-2pm, Deschutes 229

Assignment 2 is out!

- **Deadline → May 12, 2026 at 11:59pm PST**
- Please start working and talk to us if you are stuck.

- **Final → June 10, 2026 at 12:30pm PST, STB 145**
  - Closed book, no cheat sheets, no discussions.

Last Class

- Scheduling (Chapter 5)

- Next, we move to Chapter 6!

Concurrency vs. Parallelism

- Recall the difference between concurrency and parallelism.

- The goal is same to give the programmer an illusion that multiple processes/threads are executing simultaneously.

Producer-Consumer Problem

```
while (true) {
     /* produce an item in next_produced */

     while (count == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

Producer

```
while (true) {
     while (count == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;

     /* consume the item in next_consumed */
}
```

Consumer

Producer-Consumer Problem

```
while (true) {
     /* produce an item in next_produced */

     while (count == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

Producer

Any Problems!

```
while (true) {
     while (count == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;

     /* consume the item in next_consumed */
}
```

Consumer

Producer-Consumer Problem

```
while (true) {
     /* produce an item in next_produced */

     while (count == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

Producer

Any Problems!

```
while (true) {
     while (count == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;

     /* consume the item in next_consumed */
}
```

Consumer

Producer-Consumer Problem

Statements *count*++ and *count*-- may be implemented as follows in machine language.

$register_2 = \texttt{count}$
$register_2 = register_2 - 1$
$\texttt{count} = register_2$

$register_1 = \texttt{count}$
$register_1 = register_1 + 1$
$\texttt{count} = register_1$

Producer-Consumer Problem

Statements *count*++ and *count*-- may be implemented as follows in machine language.

$register_2 = \texttt{count}$
$register_2 = register_2 - 1$
$\texttt{count} = register_2$

$register_1 = \texttt{count}$
$register_1 = register_1 + 1$
$\texttt{count} = register_1$

T_0:	*producer*	execute	$register_1 = \texttt{count}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{count}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{count} = register_1$	$\{count = 6\}$
T_5:	*consumer*	execute	$\texttt{count} = register_2$	$\{count = 4\}$

Interleaving and Race

- Each process executes multiple instructions concurrently or in parallel.
  - Interrupts might happen at any time!
  - Thus, a context switch can happen at any time.

- A race condition in concurrent execution is when the **outcome of the execution** depends on the **particular interleaving of concurrent instructions**.

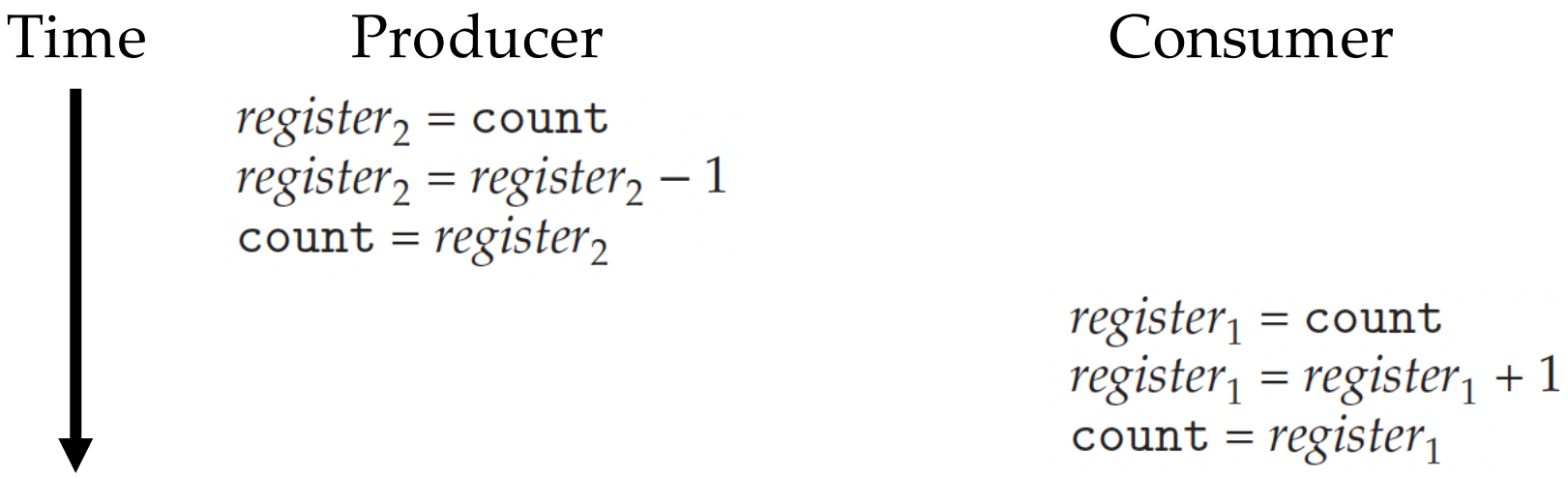
- Debugging can be challenging!
  - Race conditions are timing-dependent.
  - Errors can be non-repeatable.

How do we avoid race conditions?

How do we avoid race conditions?

- Treat a set of instructions as atomic:
  - Executed as if it were a single instruction.

- What does this mean exactly?
  - Only two possible outcomes → do nothing OR do all
  - No possible to be interrupted when partially done



Challenges with Atomicity

Challenges with Atomicity

- Hardware Restriction!
  - Atomic is typically achieved at the CPU instruction level, which applies only for operating on a Variable.

Each statement can be atomic if we use special HW instructions.

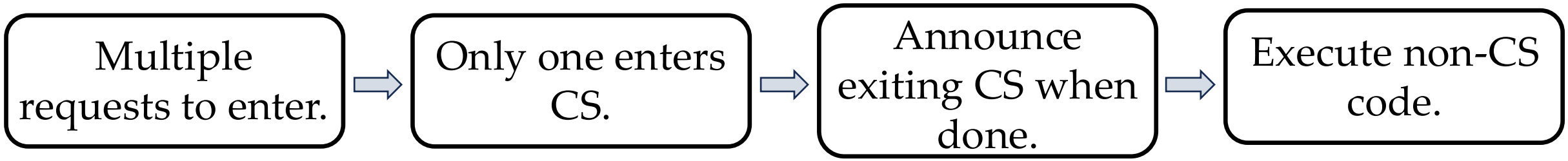
```
void Dequeue(int N, int * ret) {
    for (int i = 0; i < N; ++i) {
        ret[i] = SharedQueue[tail];
        tail = tail - 1;
    }
}
/* Assume we want to dequeue
N consecutive elements*/
```

But the whole function is NOT atomic (no HW instruction).

Critical Section Problem

Given a system of n processes {P0, P1, … Pn-1}. A Critical Section (CS) is a segment of code in each process that:

- Accesses **shared** variables/data structures/files.
- At least one process is **updating the shared** entity in the critical section.
- Has **mutual exclusion**: When one process is in its critical section, no other may be in its critical section.



Critical Section Problem

General structure of process P_i

```
do {
        code before entry (program does other things)
        code to enter the critical section
                CRITICAL SECTION
        code to exit from critical section
        code after exit (program does other things)
} while (true);
```

Only 1 process can be in critical section at a time.

Guarantees by a Solution to CS

- Mutual Exclusion
- Progress
- Bounded Waiting

Mutual Exclusion

- If process $\mathbf{P_i}$ is executing in its critical section, then no other processes can be executing in their critical sections Mutual Exclusion.

- Only one process may enter the critical section at a time!

Progress

- If no process is executing in its critical section and there exists a process that wishes to enter their critical section, then the selection of processes that will enter the critical section next cannot be postponed indefinitely.

- If no one is inside and someone wants in, the decision of who gets in should be made quickly.

Bounded Waiting

- A bound must exist on the number of times other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the N processes

- When a process asks to enter, it won't be stuck waiting forever.
  - There's a maximum number of turns others can go before it gets its chance.

How to implement CS?

Synchronization Primitives

Concurrent Applications

Shared Objects

Bounded buffer Barrier

Synchronization Variables

Semaphores Locks Condition Variables

Atomic Instructions

Interrupt disable Test-and-Set Compare-and-Swap

Hardware

Multiple processors Hardware interrupts

Disabling Interrupts

Disabling Interrupts

General structure of process P_i

do {

code before entry (program does other things)

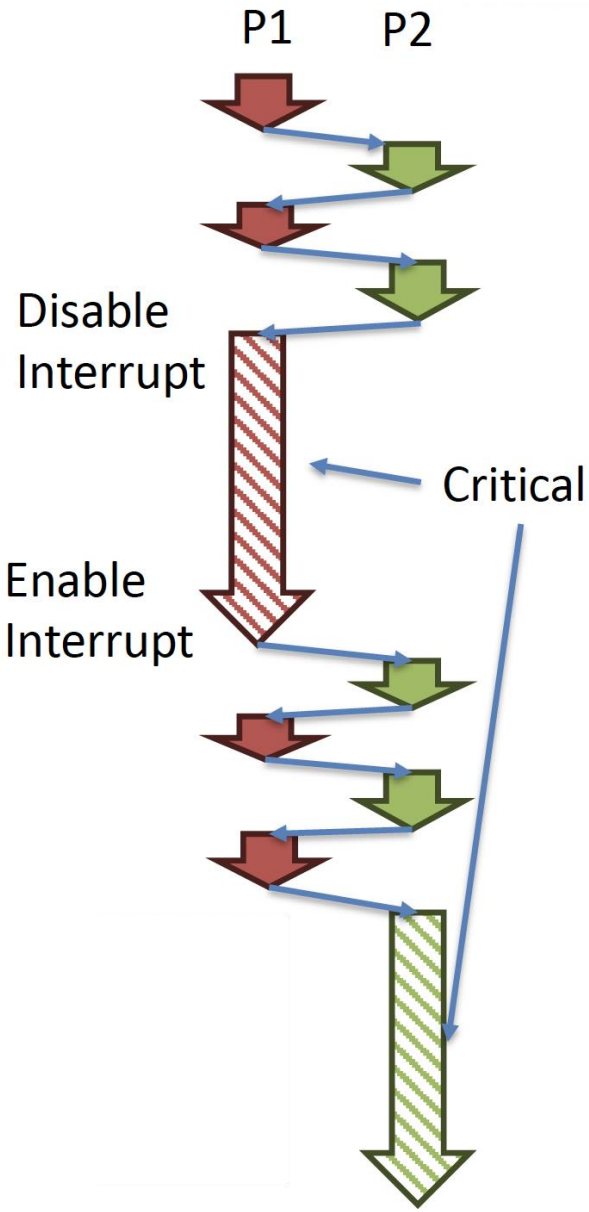
Disable Interrupts

CRITICAL SECTION

Enable Interrupts

code after exit (program does other things)

} while (true);



Disadvantages of Disabling Interrupts

- Disabling interrupts only affects the processor executing the command. In a multi-core system, threads on other cores can still access shared data!
- If a process disables interrupts and then crashes, the entire system freezes → OS scheduler cannot regain control to swap out the stuck process.
- Background tasks, like handling disk I/O, or system timers, are put on hold.
- Giving user-level programs the power to disable interrupts is dangerous!
  - A malicious program could shut down the system's ability to multitask.

Busy Waiting

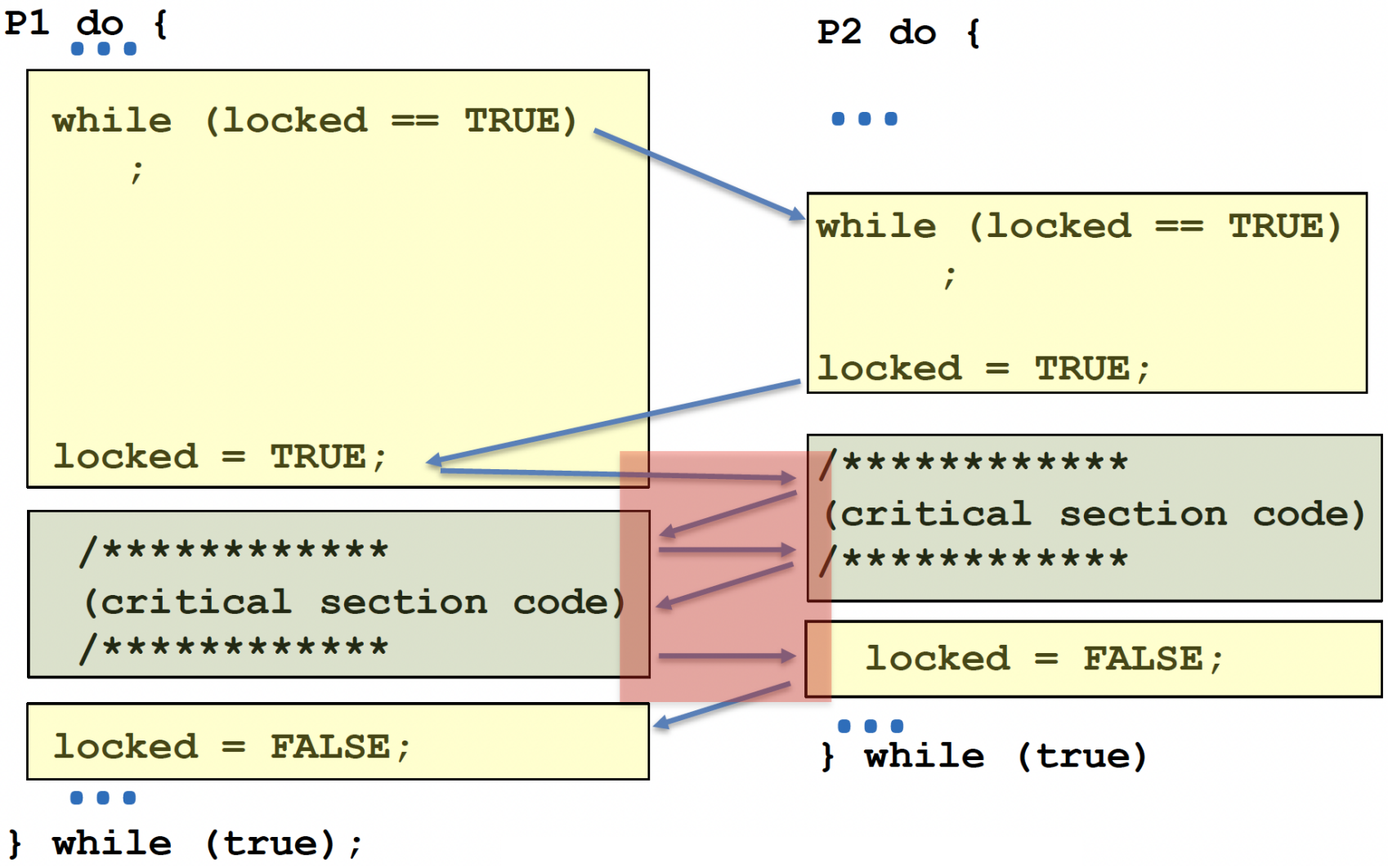
Busy Waiting

- What if we make processes wait on a shared variable?

```
locked = FALSE; // initial value (shared)

P1 do {                                  P2 do {
   . . .                                    . . .
   while (locked == TRUE)                   while (locked == TRUE)
      ;                                        ;
   locked = TRUE;                           locked = TRUE;

   /************                            /************
   (critical section code)                  (critical section code)
   /************                            /************

   locked = FALSE;                          locked = FALSE;
   . . .                                    . . .
} while (true);                          } while (true)
```

Busy Waiting

```
locked = FALSE; // initial value (shared)
```



This solution does not offer Mutual Exclusion!

Busy Waiting

```
while (locked == TRUE);
```

This is not one atomic instruction!

```
locked = TRUE;

locked = FALSE;
```

These are one atomic instructions!

Generally, if the high-level statement compiles to a single machine instruction, it is atomic.

Busy Waiting with Turns

Take turns using the critical section (CS).

- Shared variable turn is used to identify which process should enter the CS next.

```
turn = 0; // initial value
P0 do { // proces P0
   ...
  while (turn != 0);

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

  turn = 1;
   ...
} while (true)
```

```
P1 do{ // proces P1
   ...
  while (turn != 1);

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

  turn = 0;
   ...
} while (true)
```

Busy Waiting with Turns

What if a process never wants to enter?

What if a process wants to enter, but it's not its turn, and the other is not ready?

This solution does not offer Progress!

```
turn = 0; // initial value
P0 do { // proces P0
   ...
  while (turn != 0);

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

  turn = 1;
   ...
} while (true)
```

```
P1 do{ // proces P1
   ...
  while (turn != 1);

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

  turn = 0;
   ...
} while (true)
```

Busy Waiting with Multiple Variables

Each process has a flag to say that they want to enter the critical section.s

```
bool flag[2]; // initialize to FALSE
P0 do {
    • • •
    flag[0] = TRUE;
    while (flag[1] == TRUE);

      /* critical section */

    flag[0] = FALSE;
    • • •
} while (true)
```

```
P1 do {
    • • •
    flag[1] = TRUE;
    while (flag[0] == TRUE);

      /* critical section */

    flag[1] = FALSE;
    • • •
} while (true)
```

Busy Waiting with Multiple Variables

Deadlock!

This solution does not offer
Progress or bounded waiting!

```
bool flag[2]; // initialize to FALSE
P0 do {
    • • •
    flag[0] = TRUE;
    while (flag[1] == TRUE);

      /* critical section */

    flag[0] = FALSE;
    • • •
} while (true)
```

```
P1 do {
    • • •
    flag[1] = TRUE;
    while (flag[0] == TRUE);

      /* critical section */

    flag[1] = FALSE;
    • • •
} while (true)
```

Possible Solution?

Peterson's Solution

Processes share two variables:

- turn
- flag[2]

turn indicates **whose** turn it is to enter CS.

The **flag** array indicates if a process is **ready** to enter CS.

```
flag[2]; // initialize to FALSE
```

Process P_i

```
while (TRUE) {
  ...
  flag[i] = TRUE;
  turn = j;
  while (flag[j]&&turn == j);

    critical section

  flag[i] = FALSE;
  ...
}
```

Process P_j

```
while (TRUE) {
  ...
  flag[j] = TRUE;
  turn = i;
  while (flag[i]&&turn == i);

    critical section

  flag[j] = FALSE;
  ...
}
```

Peterson's Solution

- The three CS requirements are met:
- **Mutual exclusion** is preserved .
  - Pi enters CS only if: either flag[j] = false or turn = i.

- **Progress** is satisfied.
  - A process wanting to enter will be able to do so at some point.
  - If Pi is interested in entering, and Pj is not. Pi can enter regardless of turn.
  - If both processes are interested in entering, then turn determines who gets in.

- **Bounded-waiting** is met.
  - eventually it will be Pi's turn if Pi wants to enter

Limitation of Peterson's Solution

Limitation of Peterson's Solution

- ONLY works for two processes.
- Busy waiting is not efficient.
- Requires LOAD and STORE to be atomic.

Can Hardware help?

- Can Hardware provide instructions to execute the critical section atomically?
  - Perfect solution, but is it even practical?

- Can Hardware provide a shared lock?
  - Unlocked: no one is in CS
  - Locked: someone is in, needs to wait

test_and_set Instruction

- Atomic hardware instruction.
- **Test** (read) memory word and **set** value.
- Returns the original value at the "**target**" address.
- Set the new value of ***target** to TRUE (boolean types have **value 0 or 1**).

```
boolean test_and_set (boolean *target)
{
        boolean rv = *target;
        *target = TRUE;
        return rv;
}
```

test_and_set Instruction

```
while (TRUE){
  ...
     while (test_and_set(&lock) == TRUE);

      /* critical section */

     lock = FALSE;
  ...
}
```

compare_and_swap Instruction

- Atomic hardware instruction.
- **Swap** the contents of two memory words.
- Returns the original value at the "**target**" address.
- The swap takes place only if we **guess the current value correctly**.

```
int compare_and_swap(int *target, int expected, int newvalue)
{
        int temp = *target;
        if (*target == expected)
                *target = newvalue;
        return temp;
}
```

compare_and_swap Instruction

- Initially lock = 0.
- We guess it is unlocked (0), we try to swap unlocked (0) with locked (1).
  - If we are right, it should return 0 as old value.

```
while (TRUE){
  ...
  while(compare_and_swap(&lock, 0, 1) != 0);
     /* critical section */
  lock = 0;
  ...
}
```

test_and_set & compare_and_swap

- Mutual Exclusion is Satisfied!
  - The instruction is atomic (non-interruptible). If two processes call it at once, the hardware ensures only one “wins” by returning false.

- Progress is Satisfied!
  - When a process exits its critical section, it resets the lock to false. This immediately allows any waiting process to test the lock again and enter.

- Bounded Waiting is Not Satisfied!
  - Because the instruction doesn’t use a tracking system, entry is essentially a “free-for-all”.
  - A fast process could exit the CS and re-acquire the lock before a slower, waiting process has a chance to execute its next test_and_set() call.

How to achieve Bounded Waiting

CAS + Bounded Waiting

N processes: P_0 , P_1 , …, P_{N-1}

```
while (TRUE){
  ...
    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;
  ...
}
```

Spinning if I am waiting and do not have the lock

Stop waiting if I get the lock

Find the next waiting process (Round-robin order)

If no one is waiting, release the lock

If someone is waiting, stop its waiting and hand over the lock (no need to release)

Synchronization Primitives

Concurrent Applications

Shared Objects

Bounded buffer Barrier

Synchronization Variables

Semaphores Locks Condition Variables

Atomic Instructions

Interrupt disable Test-and-Set Compare-and-Swap

Hardware

Multiple processors Hardware interrupts

Mutex Locks

- Hardware-based solutions to CS problem are complicated and often inaccessible to application programmers.

- OS designers build higher-level software tools to solve the critical-section problem.

- Mutex lock (mutual exclusion)
  - Protect a critical section by first acquiring a lock then releasing the lock.
  - Implemented via hardware atomic instructions

- Mutex objects are intended to serve as a low-level primitive from which other thread synchronization functions can be built.
  - Pthreads supports mutex objects.

Mutex Locks

```
while (TRUE){
   ...
   acquire lock

   critical section

   release lock
   ...
}
```

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = false;
}
release() {
   available = true;
}
```

They be implemented with test_and_set or compare_and_swap.

Spinning (or Busy Waiting)

Spinning (or Busy Waiting)

- Busy waiting → While a process is in its critical section, any other process that tries to enter its critical section must loop continuously.

- This looping is wasteful where a single CPU core is shared among many processes.
  - wastes CPU cycles that some other process might be able to use.

- Useful only when multi-core
  - no context switch is required when a process must wait on a lock, and a context switch may take considerable time.

Semaphores

Semaphores

- Synchronization tool that provides more sophisticated ways (than mutex locks) for processes to synchronize.

- A semaphore **S** is an integer variable

- Can only be accessed via two atomic operations: wait() and signal()

```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

Semaphores

- Consider processes P1 and P2.
- Requirement → function F1() should happen before F2().
- Create a semaphore S initialized to 0.

P1:

```
F1();
signal(S);
```

P2:

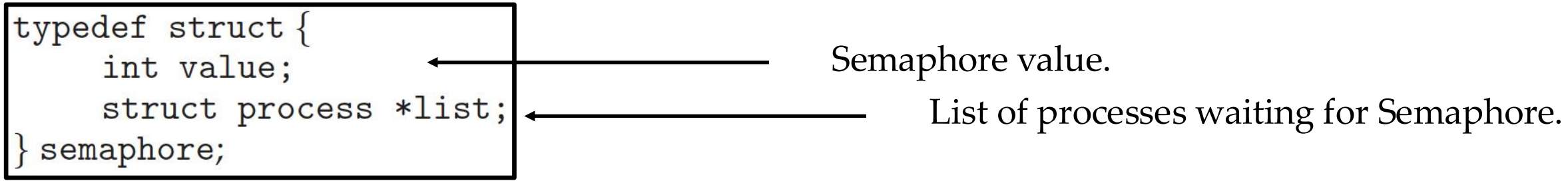
```
wait(S);
F2();
```

Semaphore Types

- **Binary semaphore**
  - An integer value can range only between 0 and 1.
  - Functionally the same as a mutex lock.

- **Counting semaphore**
  - An integer value can be set based on the problem.
  - Up to N units of resource to share and we allow up to N processors to enter.

Semaphore Implementation



Semaphore Implementation

```
typedef struct {
    int value;
    struct process *list;
} semaphore;
```

Semaphore value.

List of processes waiting for Semaphore.

```
wait(semaphore *S) {
        S->value--;
        if (S->value < 0) {
            add this process to S->list;
            sleep();
        }
}
```

Need Semaphore, decrement value.

Value less than 0, add yourself to wait list.

Sleep the process → system call!

```
signal(semaphore *S) {
        S->value++;
        if (S->value <= 0) {
            remove a process P from S->list;
            wakeup(P);
        }
}
```

Done with Semaphore, increment.

Value >= 0, remove a process from list.

Wakeup the process → system call!

Spinning vs. Suspending

- Rather than busy waiting, the process can suspend itself.
- The suspend operation places a process into a waiting queue associated with the semaphore.
  - The state of the process is switched to the waiting state, and
  - Then control is transferred to the CPU scheduler, which selects another process to execute.

- On wakeup() operation, changes the process from waiting state to the ready state.
  - The process is then placed in the ready queue.

Semaphore Details

- A negative semaphore value → number of processes waiting on that semaphore.
- Semaphore operations should be executed atomically
  - No two processes can execute wait() and signal() operations on the same semaphore at the same time.
- Implement semaphores with ***compare_and_swap()*** or spinlocks.
- Busy waiting is solved through process list!

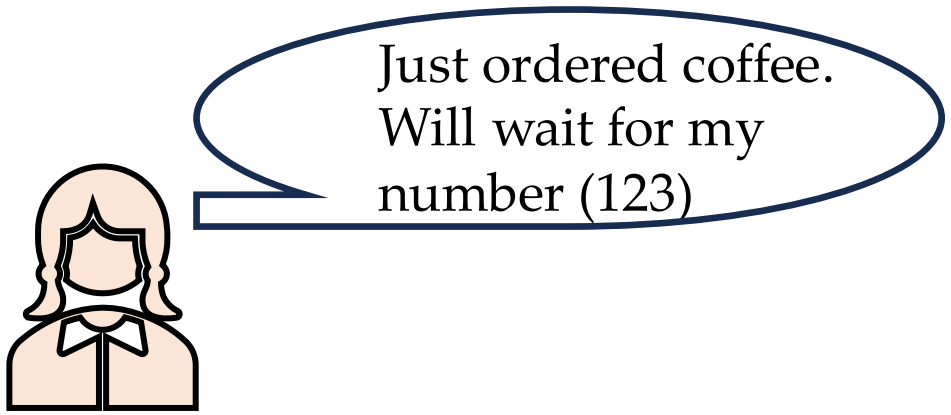
POSIX Condition Variable

POSIX Condition Variable

- What if you could get access to something when a specific condition is met?
- A condition variable allows a process/thread to wait for a certain condition to become true.
  - Always used together with a mutex.

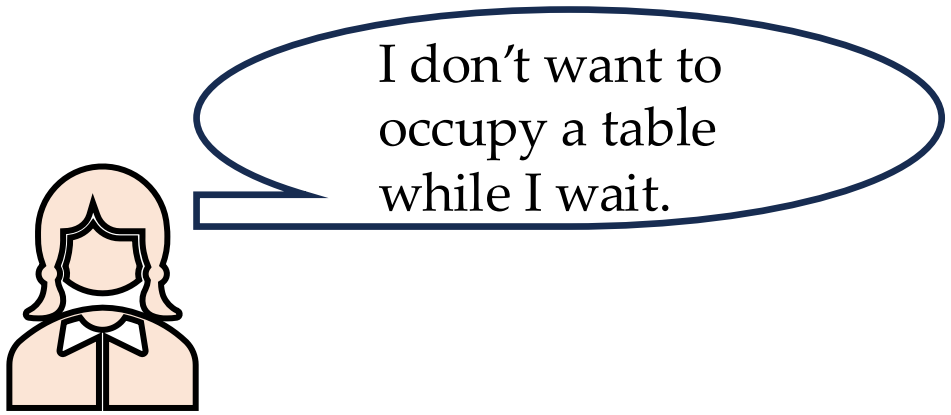
POSIX Condition Variable

- What if you could get access to something when a specific condition is met?
- A condition variable allows a process/thread to wait for a certain condition to become true.
  - Always used together with a mutex.



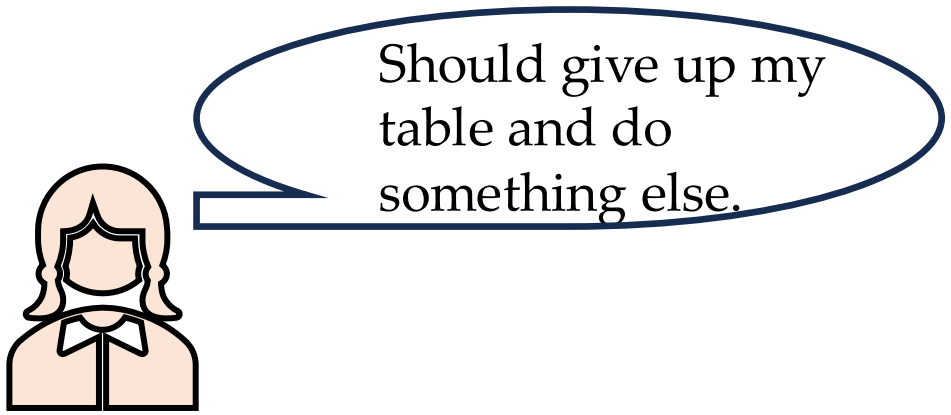
POSIX Condition Variable

- What if you could get access to something when a specific condition is met?
- A condition variable allows a process/thread to wait for a certain condition to become true.
  - Always used together with a mutex.



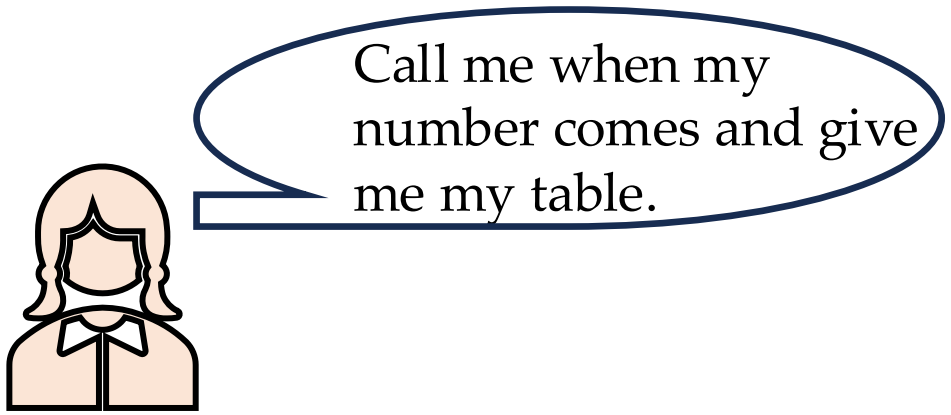
POSIX Condition Variable

- What if you could get access to something when a specific condition is met?
- A condition variable allows a process/thread to wait for a certain condition to become true.
  - Always used together with a mutex.



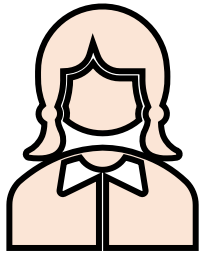
POSIX Condition Variable

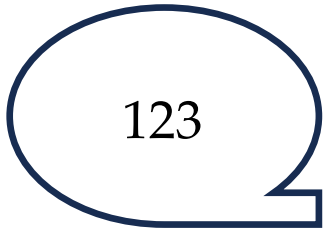
- What if you could get access to something when a specific condition is met?
- A condition variable allows a process/thread to wait for a certain condition to become true.
  - Always used together with a mutex.



POSIX Condition Variable

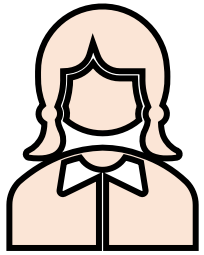
- What if you could get access to something when a specific condition is met?
- A condition variable allows a process/thread to wait for a certain condition to become true.
  - Always used together with a mutex.

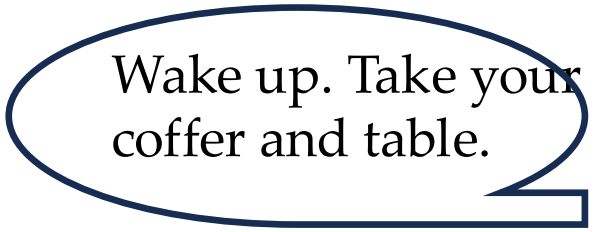




POSIX Condition Variable

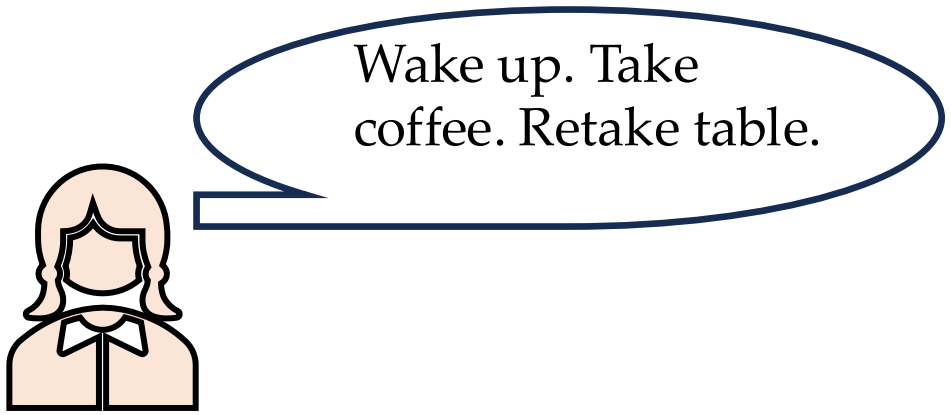
- What if you could get access to something when a specific condition is met?
- A condition variable allows a process/thread to wait for a certain condition to become true.
  - Always used together with a mutex.





POSIX Condition Variable

- What if you could get access to something when a specific condition is met?
- A condition variable allows a process/thread to wait for a certain condition to become true.
  - Always used together with a mutex.



POSIX Condition Variable

```
pthread mutex_t mutex;
pthread cond_t var;
pthread_mutex_init(&mutex, NULL);
pthread_cond_init(&var, NULL);
```

Process P1

```
pthread_mutex_lock(&mutex);
while (a != b)
        pthread_cond_wait(&var, &mutex);
pthread_mutex_unlock(&mutex);
```

Process P2

```
pthread_mutex_lock(&mutex);
a = b;
pthread_cond_signal(&var);
pthread_mutex_unlock(&mutex);
```

Synchronization Primitives

Concurrent Applications

Shared Objects

Bounded buffer Barrier

Synchronization Variables

Semaphores Locks Condition Variables

Atomic Instructions

Interrupt disable Test-and-Set Compare-and-Swap

Hardware

Multiple processors Hardware interrupts

Bounded Buffer Problem

- A classical problem, used to test newly-proposed synchronization schemes.

- Fixed-size buffer holds n items.
- **Consumers** remove items in order (wait when empty).
- **Producers** insert items in order (wait when full).
- Must **protect any modifications** to the buffer.

Bounded Buffer Problem

Semaphore **mutex**: initialized to 1

Semaphore **full**: initialized to 0

Semaphore **empty**: initialized n

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

Semaphore **mutex**: initialized to 1

Semaphore **full**: initialized to 0

Semaphore **empty**: initialized n

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

Is there an empty slot?

empty == 0, wait.

Is there a filled slot?

full == 0, wait.

Fill a slot.

Full++

Consume a slot.

Empty--

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

mutex: 1

full: 0

empty: 6

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

mutex: 1

full: 0

empty: 6

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

mutex: 1

full: 0

empty: 6

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

mutex: 1

full: 0

empty: 6

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

mutex: 1

full: 0

empty: 6

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

mutex: 1

full: 0

empty: 6

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

mutex: 1

full: 0

empty: 6

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

mutex: 1

full: 0

empty: 6

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

mutex: 1

full: 0

empty: 6

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

mutex: 1

full: 0

empty: 6

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

mutex: 1

full: 0

empty: 6

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

mutex: 1

full: 0

empty: 6

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

mutex: 1

full: 0

empty: 6

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

mutex: 1

full: 0

empty: 6

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```

Bounded Buffer Problem

mutex: 1

full: 0

empty: 6

- Producer:

```
while (TRUE){
    ...
    /* produce item*/
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add to buffer */
    ...
    signal(mutex);
    signal(full);
}
```

- Consumer:

```
while (TRUE){
    wait(full);
    wait(mutex);
    ...
    /* take from buffer*/
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume item */
    ...
}
```