



# Chapter 2: Processes & Threads

---

## **Part 2**

### Interprocess Communication (IPC) & Synchronization





# Why do we need IPC?

---

- Each process operates sequentially
- All is fine until processes want to share data
  - Exchange data between multiple processes
  - Allow processes to navigate *critical regions*
  - Maintain proper sequencing of actions in multiple processes
- These issues apply to threads as well
  - Threads can share data easily (same address space)
  - Other two issues apply to threads



# Example: bounded buffer problem

## Shared variables

```
const int n;  
typedef ... Item;  
Item buffer[n];  
int in = 0, out = 0,  
    counter = 0;
```

## Producer

```
Item pitm;  
while (1) {  
    ...  
    produce an item into pitm;  
    ...  
    while (counter == n)  
        ;  
    buffer[in] = pitm;  
    in = (in+1) % n;  
    counter += 1;  
}
```

## Atomic statements:

```
Counter += 1;
```

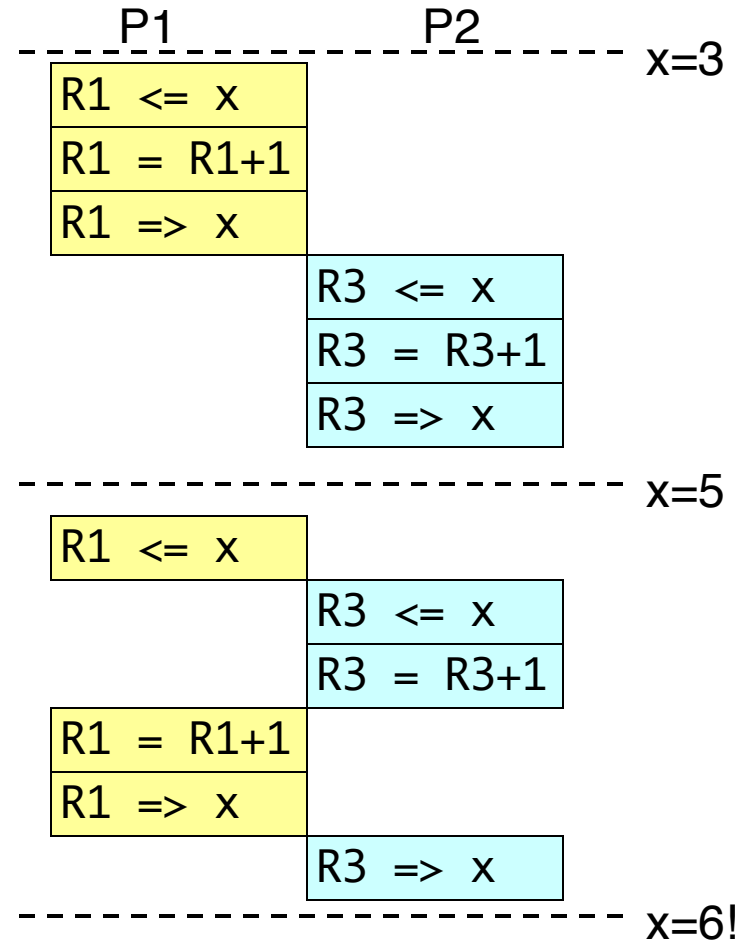
```
Counter -= 1;
```

## Consumer

```
Item citm;  
while (1) {  
    while (counter == 0)  
        ;  
    citm = buffer[out];  
    out = (out+1) % n;  
    counter -= 1;  
    ...  
    consume the item in citm  
    ...  
}
```

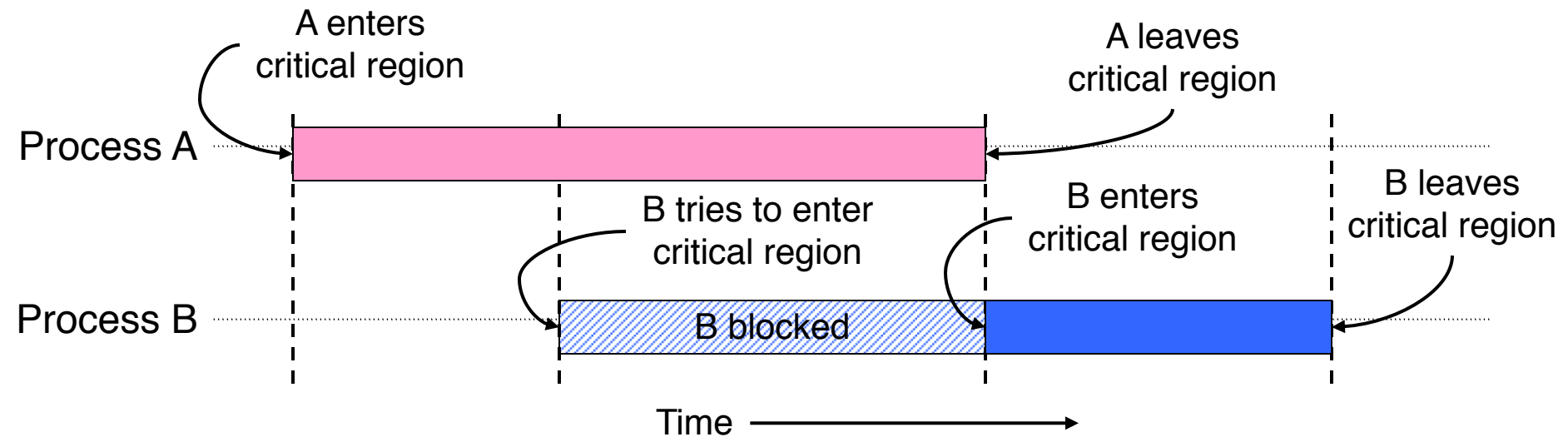
# Problem: race conditions

- Cooperating processes share storage (memory)
- Both may read and write the shared memory
- Problem: can't guarantee that read followed by write is atomic
  - Ordering matters!
- This can result in erroneous results!
- We need to eliminate race conditions...



# Critical regions

- Use critical regions to provide *mutual exclusion* and help fix race conditions
- Four conditions to provide mutual exclusion
  - No two processes simultaneously in critical region
  - No assumptions made about speeds or numbers of CPUs
  - No process running outside its critical region may block another process
  - No process must wait forever to enter its critical region



# Busy waiting: strict alternation

Process 0

```
while (TRUE) {
  while (turn != 0)
    ; /* loop */
  critical_region ();
  turn = 1;
  noncritical_region ();
}
```

Process 1

```
while (TRUE) {
  while (turn != 1)
    ; /* loop */
  critical_region ();
  turn = 0;
  noncritical_region ();
}
```

- Use a shared variable (`turn`) to keep track of whose turn it is
- Waiting process continually reads the variable to see if it can proceed
  - This is called a *spin lock* because the waiting process “spins” in a tight loop reading the variable
- Avoids race conditions, but doesn’t satisfy criterion 3 for critical regions



# Busy waiting: working solution

```
#define FALSE  0
#define TRUE   1
#define N      2      // # of processes
int turn;          // Whose turn is it?
int interested[N]; // Set to 1 if process j is interested

void enter_region(int process)
{
    int other = 1-process; // # of the other process
    interested[process] = TRUE; // show interest
    turn = process;        // Set it to my turn
    while (turn==process && interested[other]==TRUE)
        ; // Wait while the other process runs
}

void leave_region (int process)
{
    interested[process] = FALSE; // I'm no longer interested
}
```



# Bakery algorithm for many processes

- Notation used
  - $\lll$  is lexicographical order on (ticket#, process ID)
  - $(a,b) \lll (c,d)$  if  $(a < c)$  or  $((a == c) \text{ and } (b < d))$
  - $\text{Max}(a_0, a_1, \dots, a_{n-1})$  is a number  $k$  such that  $k \geq a_i$  for all  $i$
- Shared data
  - choosing initialized to 0
  - number initialized to 0

```
int n; // # of processes
int choosing[n];
int number[n];
```





# Bakery algorithm: code

```
while (1) { // i is the number of the current process
    choosing[i] = 1;
    number[i] = max(number[0], number[1], ..., number[n-1]) + 1;
    choosing[i] = 0;
    for (j = 0; j < n; j++) {
        while (choosing[j]) // wait while j is choosing a
            ; // number
        // Wait while j wants to enter and has a better number
        // than we do. In case of a tie, allow j to go if
        // its process ID is lower than ours
        while ((number[j] != 0) &&
            ((number[j] < number[i]) ||
            (number[j] == number[i]) && (j < i))))
            ;
    }
    // critical section
    number[i] = 0;
    // rest of code
}
```





# Hardware for synchronization

---

- Prior methods work, but...
  - May be somewhat complex
  - Require busy waiting: process spins in a loop waiting for something to happen, wasting CPU time
- Solution: use hardware
- Several hardware methods
  - Test & set: test a variable and set it in one instruction
  - Atomic swap: switch register & memory in one instruction
  - Turn off interrupts: process won't be switched out unless it asks to be suspended



# Mutual exclusion using hardware

- Single shared variable lock
- Still requires busy waiting, but code is much simpler
- Two versions
  - Test and set
  - Swap
- Works for any number of processes
- Possible problem with requirements
  - Non-concurrent code can lead to unbounded waiting

```
int lock = 0;
```

```
Code for process Pi
while (1) {
    while (TestAndSet(lock))
        ;
    // critical section
    lock = 0;
    // remainder of code
}
```

```
Code for process Pi
while (1) {
    while (Swap(lock,1) == 1)
        ;
    // critical section
    lock = 0;
    // remainder of code
}
```



# Solutions using busy waiting

- Problem: previous hardware solutions waste CPU time
  - Both hardware and software solutions require spinlocks (busy waiting)
  - Allow processes to sleep while they wait to execute their critical sections
- Advantage of busy waiting: multiprocessors
- Another problem of busy waiting: **multiprocessors**
- Another problem: *priority inversion* (higher priority process waits for lower priority process)
- Solution: use semaphores
  - Synchronization mechanism that doesn't require busy waiting





# Semaphores

---

- Solution: use semaphores
  - Synchronization mechanism that doesn't require busy waiting
- Implementation
  - Semaphore  $S$  accessed by two atomic operations
    - $\text{Down}(S)$ : while ( $S \leq 0$ ) {};  $S -= 1$ ;
    - $\text{Up}(S)$ :  $S += 1$ ;
  - $\text{Down}()$  or  $\text{Wait}()$  is another name for  $P()$
  - $\text{Up}()$  or  $\text{Signal}()$  is another name for  $V()$
  - Modify implementation to eliminate busy wait from  $\text{Down}()$

# Critical sections using semaphores

- Define a class called Semaphore
  - Class allows more complex implementations for semaphores
  - Details hidden from processes
- Code for individual process is simple

```
Shared variables  
Semaphore mutex;
```

```
Code for process  $P_i$   
while (1) {  
    down(mutex);  
    // critical section  
    up(mutex);  
    // remainder of code  
}
```



# Implementing semaphores with blocking

- Assume two operations:
  - Sleep(): suspends current process
  - Wakeup(P): allows process P to resume execution
- Semaphore is a class
  - Track value of semaphore
  - Keep a list of processes waiting for the semaphore
- Operations still atomic

```
class Semaphore {  
    int value;  
    ProcessList pl;  
    void down ();  
    void up ();  
};
```

```
Semaphore code  
Semaphore::down ()  
{  
    value -= 1;  
    if (value < 0) {  
        // add this process to pl  
        Sleep ();  
    }  
}  
  
Semaphore::up () {  
    Process P;  
    value += 1;  
    if (value <= 0) {  
        // remove a process P  
        // from pl  
        Wakeup (P);  
    }  
}
```

# Semaphores for barrier synchronization

- We want to execute B in  $P_1$  only after A executes in  $P_0$
- Use a semaphore initialized to 0
- Use `up()` to notify  $P_1$  at the appropriate time

## Shared variables

```
// flag initialized to 0  
Semaphore flag;
```

## Process $P_0$

```
·  
·  
·  
// Execute code for A  
flag.up ();
```

## Process $P_1$

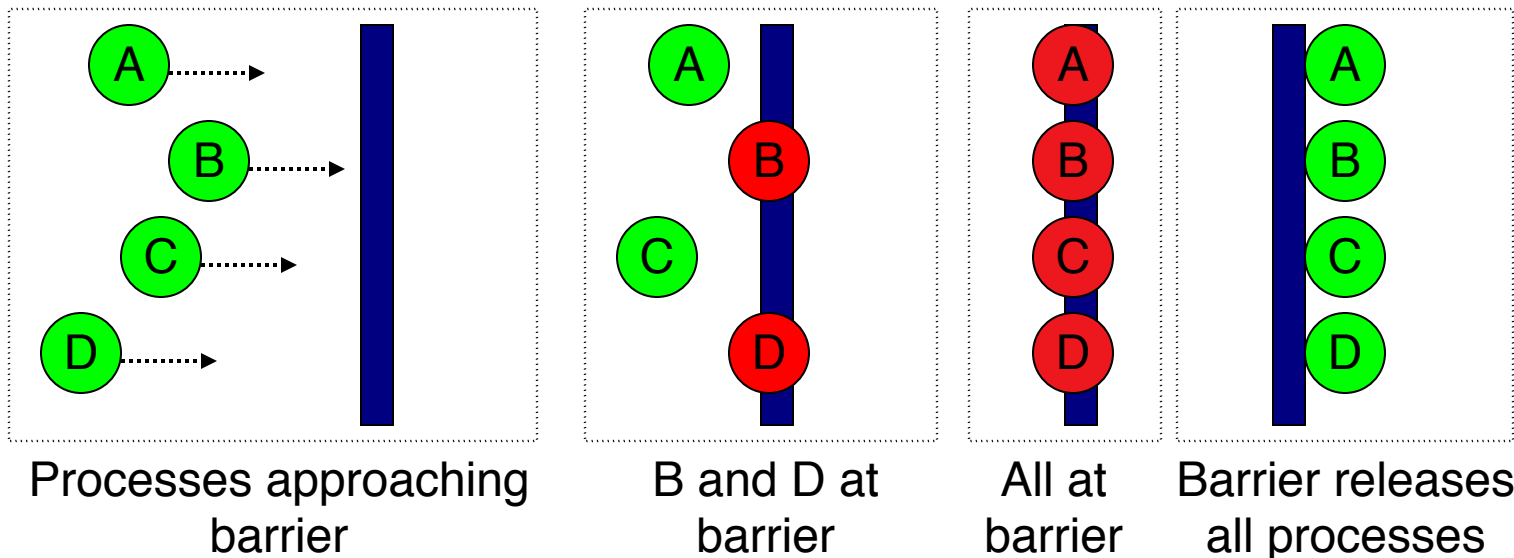
```
·  
·  
·  
flag.down ();  
// Execute code for B
```





# Barriers

- Used for synchronizing multiple processes
- Processes wait at a “barrier” until all in the group arrive
- After all have arrived, all processes can proceed
- May be implemented using locks and condition variables





# Types of semaphores

---

- Two different types of semaphores
  - Counting semaphores
  - Binary semaphores
- Counting semaphore
  - Value can range over an unrestricted range
- Binary semaphore
  - Only two values possible
    - 1 means the semaphore is available
    - 0 means a process has acquired the semaphore
  - May be simpler to implement
- Possible to implement one type using the other



# Deadlock and Starvation

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let  $S$  and  $Q$  be two semaphores initialized to 1

$P_0$	$P_1$
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
<code>...</code>	<code>...</code>
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**



# Classical synchronization problems

---

- Bounded Buffer
  - Multiple producers and consumers
  - Synchronize access to shared buffer
- Readers & Writers
  - Many processes that may read and/or write
  - Only one writer allowed at any time
  - Many readers allowed, but not while a process is writing
- Dining Philosophers
  - Resource allocation problem
  - N processes and limited resources to perform sequence of tasks
- Goal: use semaphores to implement solutions to these problems



# Bounded buffer problem

- Goal: implement producer-consumer without busy waiting

```
const int n;  
Semaphore empty(n),full(0),mutex(1);  
Item buffer[n];
```

## Producer

```
int in = 0;  
Item pitem;  
while (1) {  
    // produce an item  
    // into pitem  
    empty.down();  
    mutex.down();  
    buffer[in] = pitem;  
    in = (in+1) % n;  
    mutex.up();  
    full.up();  
}
```

## Consumer

```
int out = 0;  
Item citem;  
while (1) {  
    full.down();  
    mutex.down();  
    citem = buffer[out];  
    out = (out+1) % n;  
    mutex.up();  
    empty.up();  
    // consume item from  
    // citem  
}
```



# Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers – only read the data set; they do *not* perform any updates
  - Writers – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities
- Shared Data
  - Data set
  - Semaphore **rw\_mutex** initialized to 1
  - Semaphore **mutex** initialized to 1
  - Integer **read\_count** initialized to 0

# Readers-writers problem

## Shared variables

```
int nreaders;  
Semaphore mutex(1), writing(1);
```

## Reader process

```
...  
mutex.down();  
nreaders += 1;  
if (nreaders == 1) // wait if  
    writing.down(); // 1st reader  
mutex.up();  
// Read some stuff  
mutex.down();  
nreaders -= 1;  
if (nreaders == 0) // signal if  
    writing.up(); // last reader  
mutex.up();  
...
```

## Writer process

```
...  
writing.down();  
// Write some stuff  
writing.up();  
...
```

# Readers-Writers Problem Variations

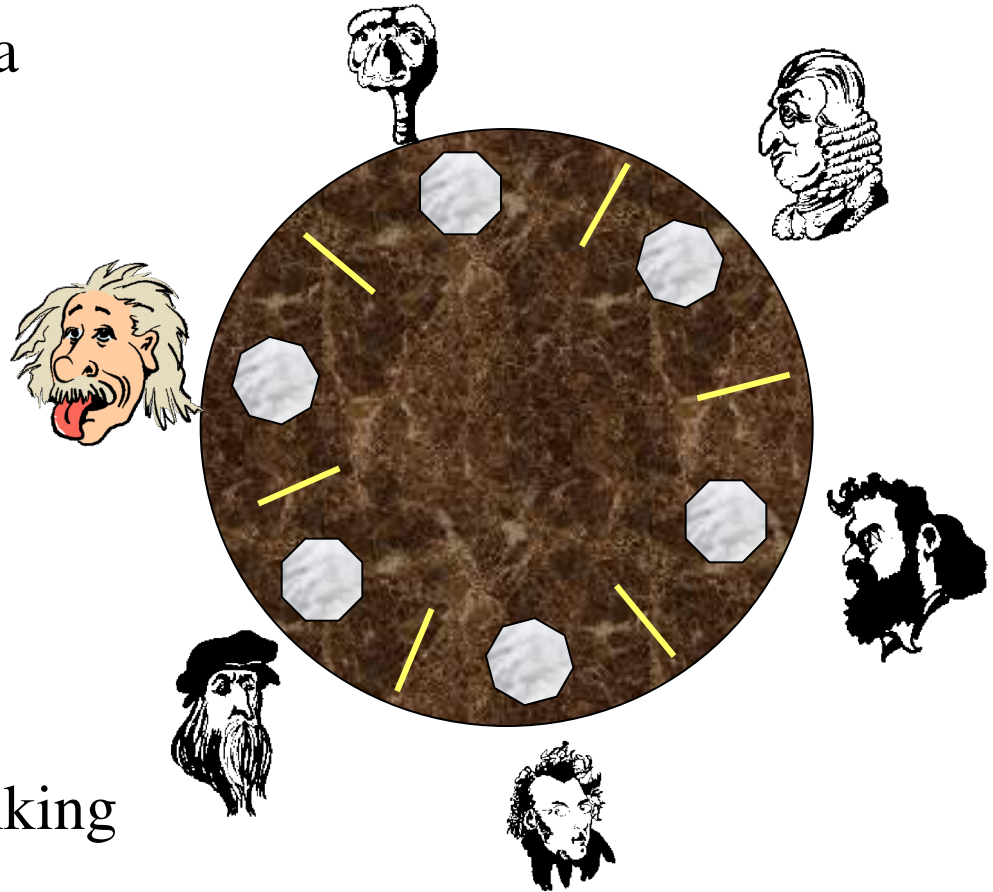
- **First** variation – no reader kept waiting unless writer has permission to use shared object
- **Second** variation – once writer is ready, it performs the write ASAP
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks





# Dining Philosophers

- $N$  philosophers around a table
  - All are hungry
  - All like to think
- $N$  chopsticks available
  - 1 between each pair of philosophers
- Philosophers need two chopsticks to eat
- Philosophers alternate between eating and thinking
- Goal: coordinate use of chopsticks



# Dining Philosophers: solution 1

- Use a semaphore for each chopstick
- A hungry philosopher
  - Gets the chopstick to his right
  - Gets the chopstick to his left
  - Eats
  - Puts down the chopsticks
- Potential problems?
  - Deadlock
  - Fairness

## Shared variables

```
const int n;  
// initialize to 1  
Semaphore chopstick[n];
```

## Code for philosopher *i*

```
while(1) {  
    chopstick[i].down();  
    chopstick[(i+1)%n].down();  
    // eat  
    chopstick[i].up();  
    chopstick[(i+1)%n].up();  
    // think  
}
```

# Dining Philosophers: solution 2

- Use a semaphore for each chopstick
- A hungry philosopher
  - Gets lower, then higher numbered chopstick
  - Eats
  - Puts down the chopsticks
- Potential problems?
  - Deadlock
  - Fairness

## Shared variables

```
const int n;  
// initialize to 1  
Semaphore chopstick[n];
```

```
Code for philosopher i  
int i1,i2;  
while(1) {  
    if (i != (n-1)) {  
        i1 = i;  
        i2 = i+1;  
    } else {  
        i1 = 0;  
        i2 = n-1;  
    }  
    chopstick[i1].down();  
    chopstick[i2].down();  
    // eat  
    chopstick[i1].up();  
    chopstick[i2].up();  
    // think  
}
```

# Dining philosophers with locks

## Shared variables

```
const int n;  
// initialize to THINK  
int state[n];  
Lock mutex;  
// use mutex for self  
Condition self[n];
```

```
void test(int k)  
{  
    if ((state[(k+n-1)%n])!=EAT) &&  
        (state[k]==HUNGRY) &&  
        (state[(k+1)%n]!=EAT)) {  
        state[k] = EAT;  
        self[k].Signal();  
    }  
}
```

## Code for philosopher $j$

```
while (1) {  
    // pickup chopstick  
    mutex.Acquire();  
    state[j] = HUNGRY;  
    test(j);  
    if (state[j] != EAT)  
        self[j].Wait();  
    mutex.Release();  
    // eat  
    mutex.Acquire();  
    state[j] = THINK;  
    test((j+1)%n); // next  
    test((j+n-1)%n); // prev  
    mutex.Release();  
    // think  
}
```

# The Sleepy Barber Problem



# Code for the Sleepy Barber Problem

```
#define CHAIRS 5
Semaphore customers=0;
Semaphore barbers=0;
Semaphore mutex=0;
int waiting=0;
```

```
void barber(void)
{
  while(TRUE) {
    // Sleep if no customers
    customers.down();
    // Decrement # of waiting people
    mutex.down();
    waiting -= 1;
    // Wake up a customer to cut hair
    barbers.up();
    mutex.up();
    // Do the haircut
    cut_hair();
  }
}
```

```
void customer(void)
{
  mutex.down();
  // If there is space in the chairs
  if (waiting < CHAIRS) {
    // Another customer is waiting
    waiting++;
    // Wake up the barber. This is
    // saved up, so the barber doesn't
    // sleep if a customer is waiting
    customers.up();
    mutex.up();
    // Sleep until the barber is ready
    barbers.down();
    get_haircut();
  } else {
    // Chairs full, leave the critical
    // region
    mutex.up ();
  }
}
```





# Monitors

---

- A *monitor* is another kind of high-level synchronization primitive
  - One monitor has multiple entry points
  - Only one process may be in the monitor at any time
  - Enforces mutual exclusion - less chance for programming errors
- Monitors provided by high-level language
  - Variables belonging to monitor are protected from simultaneous access
  - Procedures in monitor are guaranteed to have mutual exclusion
- Monitor implementation
  - Language / compiler handles implementation
  - Can be implemented using semaphores



# Monitor usage

```
monitor mon {
  int foo;
  int bar;
  double arr[100];
  void proc1(...) {
  }
  void proc2(...) {
  }
  void mon() { // initialization code
  }
};
```

- This looks like C++ code, but it's not supported by C++
- Provides the following features:
  - Variables foo, bar, and arr are accessible only by proc1 & proc2
  - Only one process can be executing in either proc1 or proc2 at any time





# Condition variables in monitors

- Problem: how can a process wait inside a monitor?
  - Can't simply sleep: there's no way for anyone else to enter
  - Solution: use a condition variable
- Condition variables support two operations
  - Wait(): suspend this process until signaled
  - Signal(): wake up exactly one process waiting on this condition variable
    - If no process is waiting, signal has no effect
    - Signals on condition variables aren't "saved up"
- Condition variables are only usable within monitors
  - Process must be in monitor to signal on a condition variable
  - Question: which process gets the monitor after Signal()?



# Monitor semantics

- Problem: P signals on condition variable X, waking Q
  - Both can't be active in the monitor at the same time
  - Which one continues first?
- Mesa semantics
  - Signaling process (P) continues first
  - Q resumes when P leaves the monitor
  - Seems more logical: why suspend P when it signals?
- Hoare semantics
  - Awakened process (Q) continues first
  - P resumes when Q leaves the monitor
  - May be better: condition that Q wanted may no longer hold when P leaves the monitor





# Locks & condition variables

---

- Monitors require native language support
- Provide monitor support using special data types and procedures
  - Locks (Acquire(), Release())
  - Condition variables (Wait(), Signal())
- Lock usage
  - Acquiring a lock == entering a monitor
  - Releasing a lock == leaving a monitor
- Condition variable usage
  - Each condition variable is associated with exactly one lock
  - Lock must be held to use condition variable
  - Waiting on a condition variable releases the lock implicitly
  - Returning from Wait() on a condition variable reacquires the lock



# Implementing locks with semaphores

```
class Lock {  
    Semaphore mutex(1);  
    Semaphore next(0);  
    int nextCount = 0;  
};
```

```
Lock::Acquire()  
{  
    mutex.down();  
}
```

```
Lock::Release()  
{  
    if (nextCount > 0)  
        next.up();  
    else  
        mutex.up();  
}
```

- Use mutex to ensure exclusion within the lock bounds
- Use next to give lock to processes with a higher priority (why?)
- nextCount indicates whether there are any higher priority waiters

# Implementing condition variables

```
class Condition {
    Lock *lock;
    Semaphore condSem(0);
    int semCount = 0;
};
```

```
Condition::Wait ()
{
    semCount += 1;
    if (lock->nextCount > 0)
        lock->next.up();
    else
        lock->mutex.up();
    condSem.down ();
    semCount -= 1;
}
```

```
Condition::Signal ()
{
    if (semCount > 0) {
        lock->nextCount += 1;
        condSem.up ();
        lock->next.down ();
        lock->nextCount -= 1;
    }
}
```

- Are these Hoare or Mesa semantics?
- Can there be multiple condition variables for a single Lock?



# Message passing

---

- Synchronize by exchanging messages
- Two primitives:
  - Send: send a message
  - Receive: receive a message
  - Both may specify a “channel” to use
- Issue: how does the sender know the receiver got the message?
- Issue: authentication