

# CSC139 Operating System Principles

Fall 2021, Part 2-4

Instructor: Dr. Yuan Cheng

# Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Illustration of the problem:
  - Suppose that we wanted to provide a solution to the consumer-producer problem that fills *all* the buffers.
  - We can do so by having an integer **counter** that keeps track of the number of full buffers. Initially, **counter** is set to 0. It is incremented by the producer after it produces a new buffer and is decremented by the consumer after it consumes a buffer.

# Producer

```
while (true) {  
    /* produce an item in next produced */  
  
    while (counter == BUFFER_SIZE) ;  
        /* do nothing */  
    buffer[in] = next_produced;  
    in = (in + 1) % BUFFER_SIZE;  
    counter++;  
}
```

# Consumer

```
while (true) {  
    while (counter == 0)  
        ; /* do nothing */  
    next_consumed = buffer[out];  
    out = (out + 1) % BUFFER_SIZE;  
    counter--;  
    /* consume the item in next consumed */  
}
```

# Race Condition

- **counter++** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- **counter--** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with “count = 5” initially:

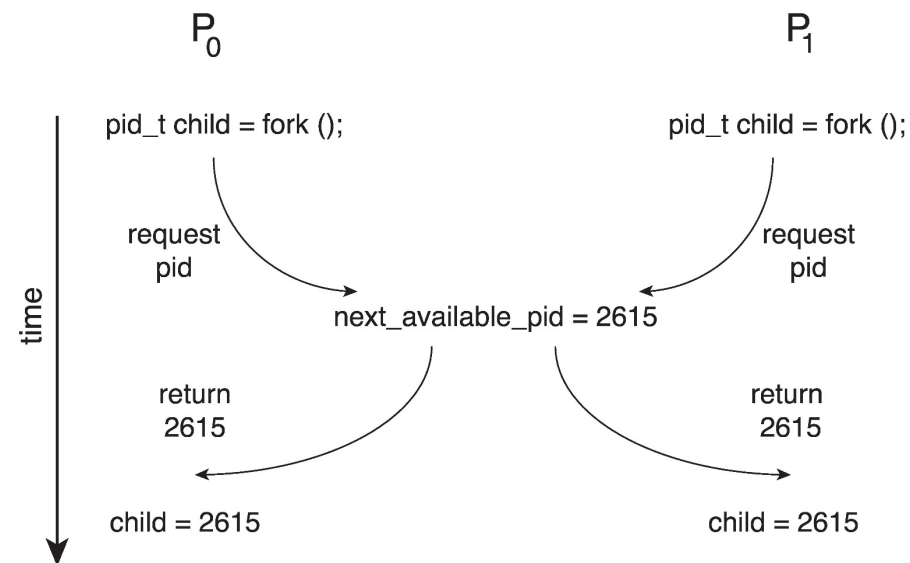
S0: producer execute	<code>register1 = counter</code>	{register1 = 5}
S1: producer execute	<code>register1 = register1 + 1</code>	{register1 = 6}
S2: consumer execute	<code>register2 = counter</code>	{register2 = 5}
S3: consumer execute	<code>register2 = register2 - 1</code>	{register2 = 4}
S4: producer execute	<code>counter = register1</code>	{counter = 6}
S5: consumer execute	<code>counter = register2</code>	{counter = 4}

# Race Condition (cont.)

- Where multiple processes are writing or reading some shared data and the final **result depends on who runs precisely when**, are called **race conditions**

# Race Condition (cont.)

- Processes  $P_0$  and  $P_1$  are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid)



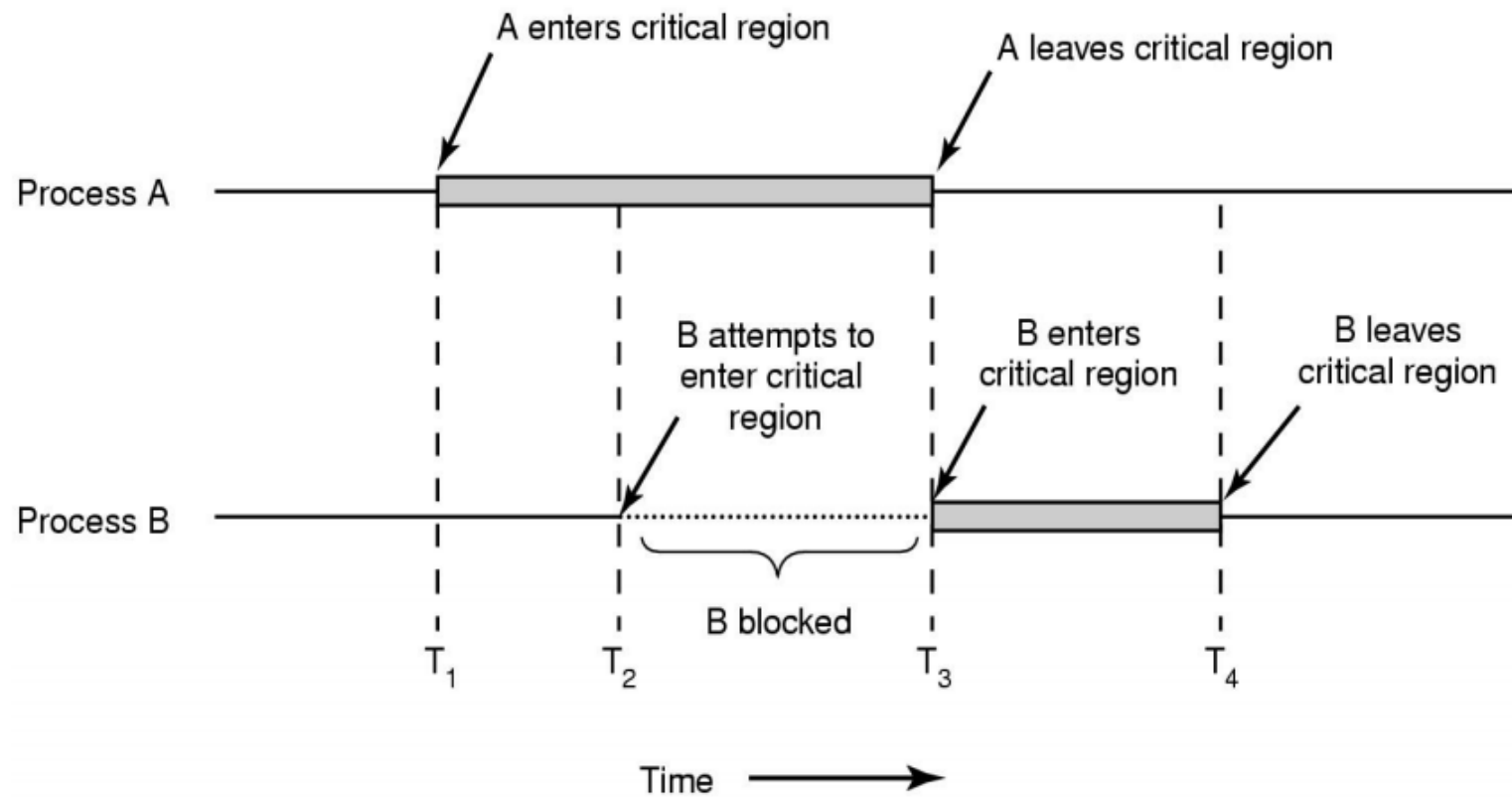
- Unless there is mutual exclusion, the same pid could be assigned to two different processes!

# Critical Section & Mutual Exclusion

- Critical Section
  - A section of code in which the process accesses and modifies ***shared variables***
- Mutual Exclusion
  - A method for ensuring that one (or a specified number) of processes are in a critical section



# Mutual Exclusion



# Critical Section Problem

- Consider system of  $n$  processes  $\{p_0, p_1, \dots, p_{n-1}\}$
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc.
  - When one process in critical section, no other may be in its critical section
- **Critical section problem** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

# Critical Section

- General structure of process  $P_i$

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

# Algorithm for Process $P_i$

```
do {  
    while (turn == i);  
        critical section  
    turn = j;  
        remainder section  
} while (true);
```

# Solution to Critical-Section Problem

1. **Mutual Exclusion** - If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist 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 and before that request is granted
  - Assume that each process executes at a non-zero speed
  - No assumption concerning **relative speed** of the  $n$  processes

# Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non-preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

# How to Implement Mutual Exclusion

- Three possibilities
  - Application: programmer builds some method into the program
  - Hardware: special h/w instructions provided to implement ME
  - OS: provides some services that can be used by programmer
- All schemes rely on some code for
  - `enter_critical_section`, and
  - `exit_critical_section`

# Application Mutual Exclusion

- Application Mutual Exclusion is
  - Implemented by the programmer
  - Hard to get correct, and very inefficient
- All rely on some form of busy waiting (process tests a condition, set a flag, and loops while the condition remains the same)



# Example

- **Producer**

```
produce
if lock = 1 loop until lock = 0
lock = 1
put in buffer
lock = 0
```

- **Consumer**

```
if lock = 1 loop until lock = 0
lock = 1
get from buffer
lock = 0
consume
```

# Locks

- A lock is a variable
- Two states
  - Available or free
  - Locked or held
- `lock()` : tries to acquire the lock
- `unlock()` : releases the lock which has been acquired by caller

# Disabling Interrupts

- One solution supported by hardware may be to use interrupt capability

```
do {  
    lock();  
    critical section  
    unlock();  
    remainder section  
} while (1);
```

```
void lock() {  
    DisableInterrupts();  
}  
void unlock() {  
    EnableInterrupts();  
}
```

# Disabling Interrupts (cont.)

- On a single CPU only one process is executed
- Concurrency is achieved by interleaving execution (usually done using interrupts)
- If you disable interrupts then you can be sure only one process will ever execute
- One process can lock a system or degrade performance greatly

# Synchronization Hardware

- Many machines provide special hardware instructions to help achieve mutual exclusion
- The Test-And-Set (TAS) instruction tests and modifies the content of a memory word **atomically**
- TAS returns old value pointed to by **old\_ptr** and updates said value to **new**

```
Int TestAndSet(int *old_ptr, int new) {  
    int old = *old_ptr;    //fetch old value at old_ptr  
    *old_ptr = new;        //store 'new' into old_ptr  
    return old;            // return the old value  
}
```

# Mutual Exclusion with TAS

- Initially, lock's flag set to 0

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock (lock_t *lock) {
    lock->flag = 0;
}
```

# Busy Waiting and Spin Locks

- This approach is based on busy waiting
  - If the critical section is being used, waiting processes loop continuously at the entry point
- A binary “lock” variable that uses busy waiting is called a **spin lock**
  - Processes that find the lock unavailable “spin” at the entry
- It actually works (mutual exclusion)
- Disadvantages?
  - Fairness?
  - Performance?

# Another HW Approach: Compare-And-Swap

- Test whether the value at the address specified by `ptr` is equal to `expected`
- If so, update the memory location pointed to by `ptr` with the new value. If not, do nothing.

```
int CompareAndSwap(int *ptr, int
expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag,
0, 1) == 1)
        ; // spin
}
```



# Hardware ME Characteristics

- Advantages
  - Can be used by a single or multiple processes (with shared memory)
  - Simple and therefore easy to verify
  - Can support multiple critical sections
- Disadvantages
  - Busy waiting is used
  - Starvation is possible
  - Deadlock is possible (especially with priorities)

# Mutual Exclusion Through OS

- Semaphores
- Message passing

# Mutex Locks

- Previous solutions are complicated and generally inaccessible to application programmers
- OS designers build software tools to solve critical section problem
- Simplest is **mutex** lock
- Protect a critical section by first `acquire()` a lock then `release()` the lock
  - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

# acquire() and release()

- ```
acquire() {  
    while (!available)  
        ; /* busy wait */  
    available = false;;  
}
```
- ```
release() {  
    available = true;  
}
```
- ```
do {  
    acquire lock  
    critical section  
    release lock  
    remainder section  
} while (true);
```

# Semaphores

- Introduced by Edsger Dijkstra
- Motivation: Avoid busy waiting by blocking a process execution until some condition is satisfied
- Major advance incorporated into many modern operating systems (Unix, OS/2)
- A semaphore is
  - a non-negative integer
  - that has two valid operations

# Semaphore Operations

- Wait(s):

```
If s > 0 then s := s - 1
           else block this process
```

- Signal(s):

```
If there is a blocked process on this semaphore
then wake it up
           else s := s + 1
```

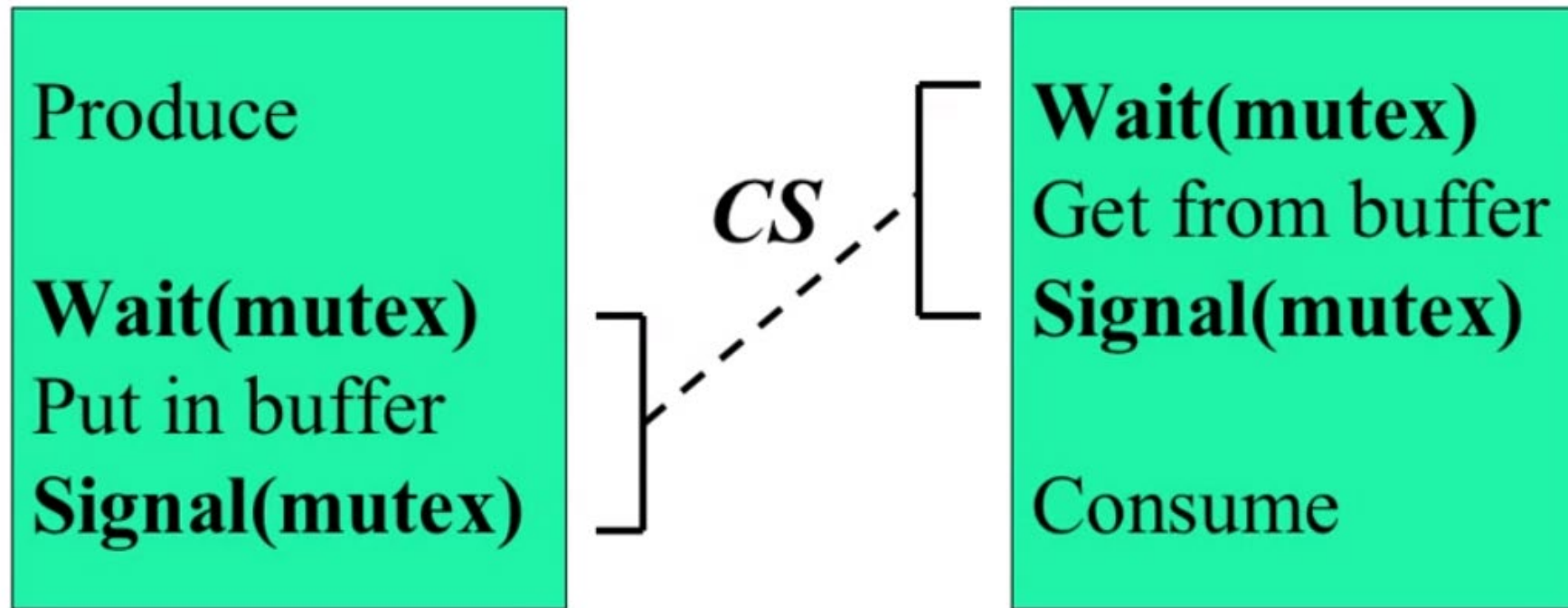
- Wait() and Signal() originally called P() and V()

- prolaag: “probeer” in Dutch means “try”, and “verlaag” means “decrease”
- verhoog: Dutch for “increase”

# More on Semaphores

- Two types of semaphores
  - Binary semaphores can only be 0 or 1
    - Same as a mutex lock
  - Counting semaphores can be any non-negative integer
- Semaphores are an OS service implemented using one of the methods shown already
  - Usually by disabling interrupts for a very short time

# Producer – Consumer Problem: Solution by Semaphores



- Initially semaphore *mutex* is 1



# Another Example

- Three processes all share a resource on which
  - one draw an A
  - one draw a B
  - one draw a C
- Implement a form of synchronization so that the output appears ABC

*Process A*

```
think();  
  
draw_A();
```

*Process B*

```
think();  
  
draw_B();
```

*Process C*

```
think();  
  
draw_C();
```

- Semaphore  $b = 0, c = 0;$

*Process A*

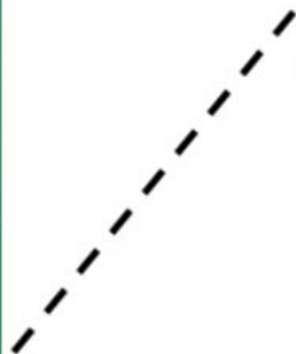
```
think();  
draw_A();  
signal(b);
```

*Process B*

```
wait(b);  
think();  
draw_B();  
signal(c);
```

*Process C*

```
wait(c);  
think();  
draw_C();
```



# Bounded-Buffer Problem

- We need 3 semaphores:
  1. We need a semaphore **mutex** to have mutual exclusion on buffer access
  2. We need a semaphore **full** to synchronize producer and consumer on the number of consumable items
  3. We need a semaphore **empty** to synchronize producer and consumer on the number of empty spaces

# Bounded-Buffer - Semaphores

- Shared data

```
semaphore full, empty, mutex;
```

Initially:

```
full = 0, empty = n, mutex = 1
```

# Bounded-Buffer – Producer Process

```
do {  
  ...  
  produce an item in nextp  
  ...  
  wait(empty) ;  
  wait(mutex) ;  
  ...  
  add nextp to buffer  
  ...  
  signal(mutex) ;  
  signal(full) ;  
} while (1);
```

# Bounded-Buffer – Consumer Process

```
do {  
    wait(full) ;  
    wait(mutex) ;  
    ...  
    remove item from buffer to nextc  
    ...  
    signal(mutex) ;  
    signal(empty) ;  
    ...  
    consume the item in nextc  
    ...  
} while (1);
```

# Notes on Bounded-Buffer Solution

- Remarks (from consumer point of view)
  - Putting signal(empty) inside the CS of the consumer (instead of outside) has no effect since the producer must always wait for both semaphores before proceeding
  - The consumer must perform wait(full) before wait(mutex), otherwise deadlock occurs if consumer enters CS while the buffer is empty
- Conclusion: using semaphores is a difficult art

# Mutual Exclusion Problem: Starvation

- Definition
  - Indefinitely delaying the scheduling of a process in favor of other processes
- Cause
  - Usually a bias in a system scheduling policies
- Solution
  - Implement some form of aging



# Another Problem: Deadlocks

- Two (or more) processes are blocked waiting for an event that will never occur
- Generally, A waits for B to do something and B is waiting for A
- Both are not doing anything so both events never occur

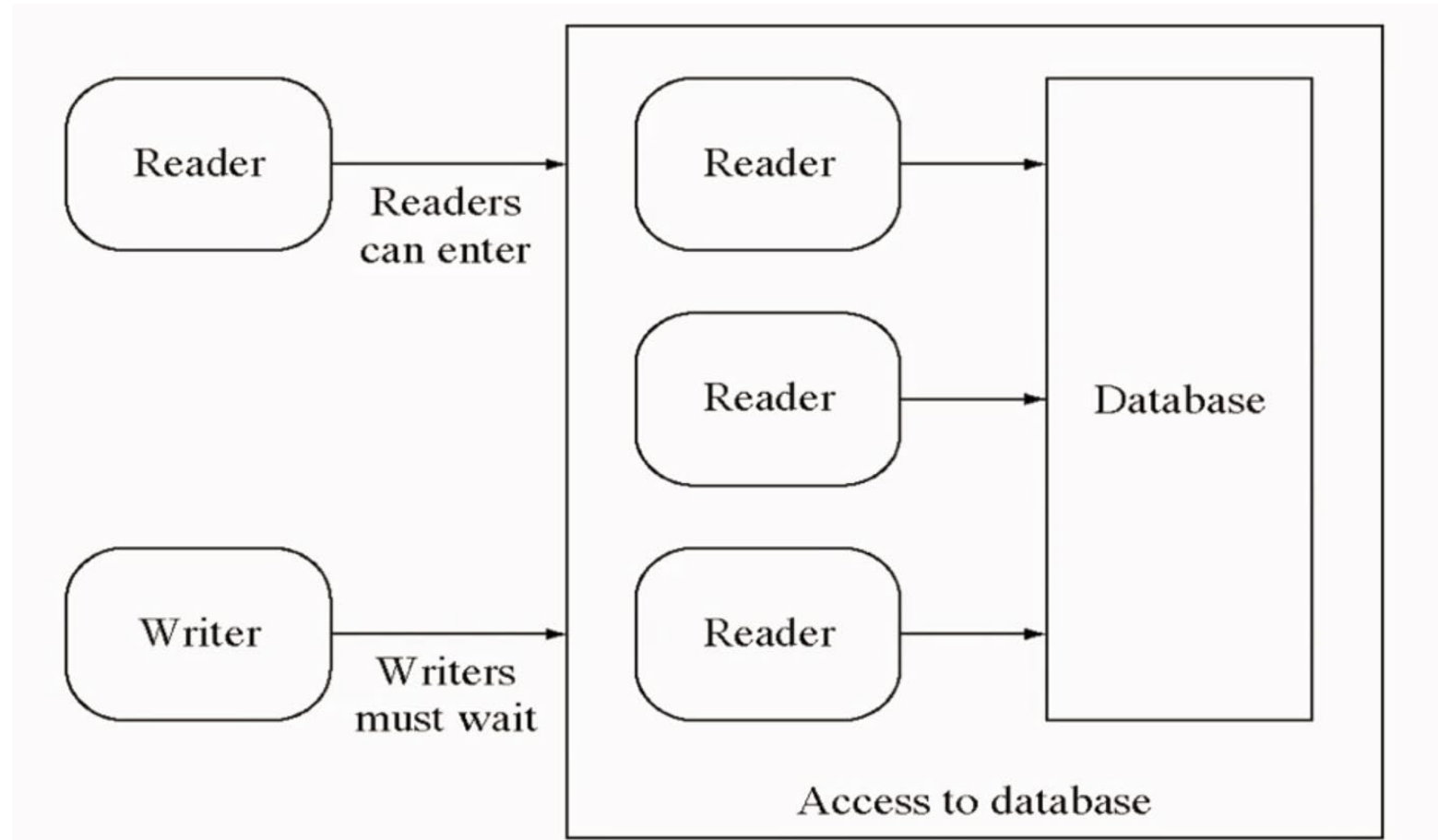
# Classical IPC Problems

- Bounded-buffer problem
- Readers and writers problem
  - Models access to a database (both read and write)
- Dining philosophers problem
  - Models processes competing for exclusive access to a limited number of resources such as I/O devices

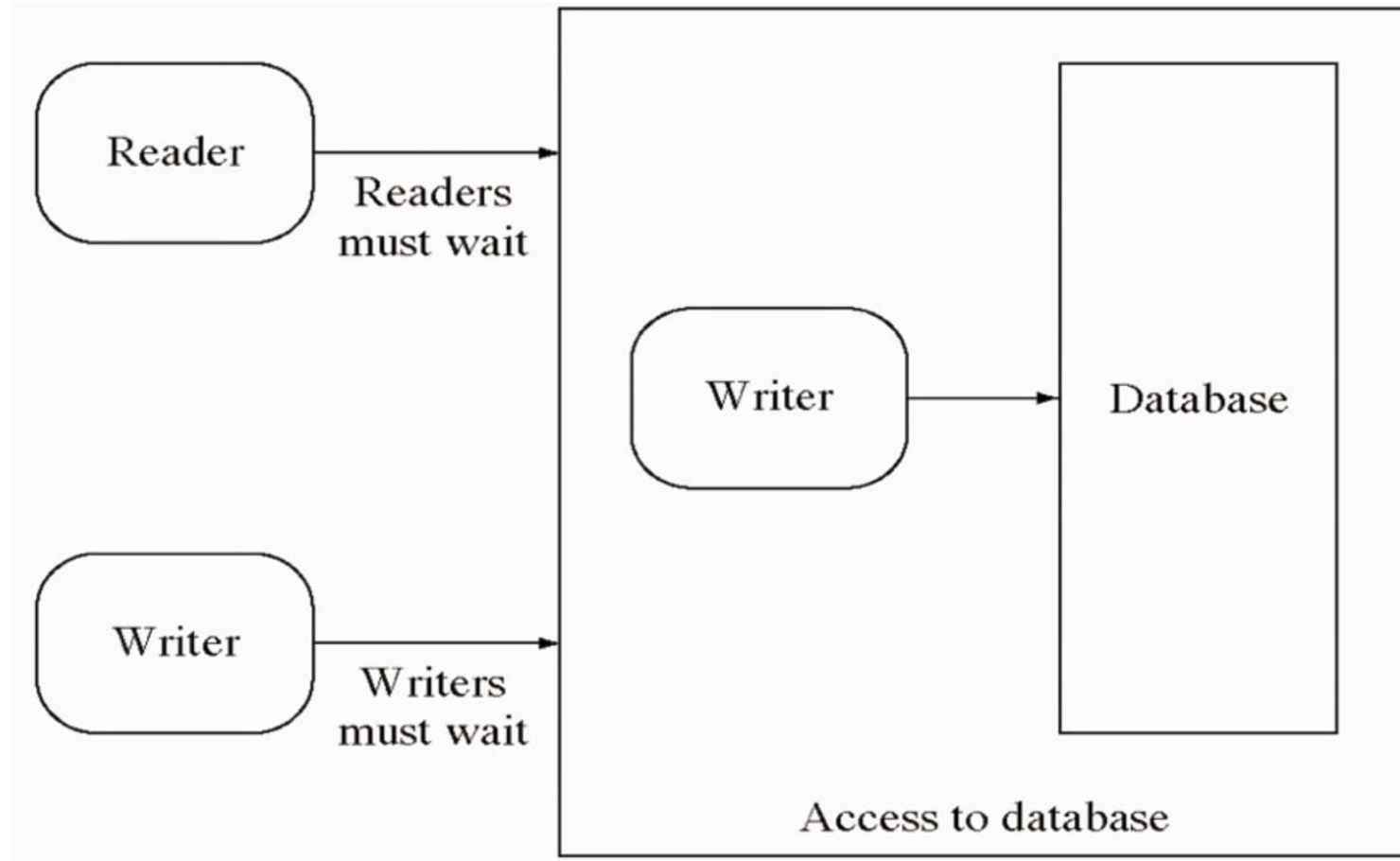
# Readers-Writers Problem

- Any number of reader activities and writer activities are running
- At any time, a reader activity may wish to read data
- At any time, a writer activity may want to modify the data
- Any number of readers may access the data simultaneously
- During the time a writer is writing, no other reader or writer may access the shared data

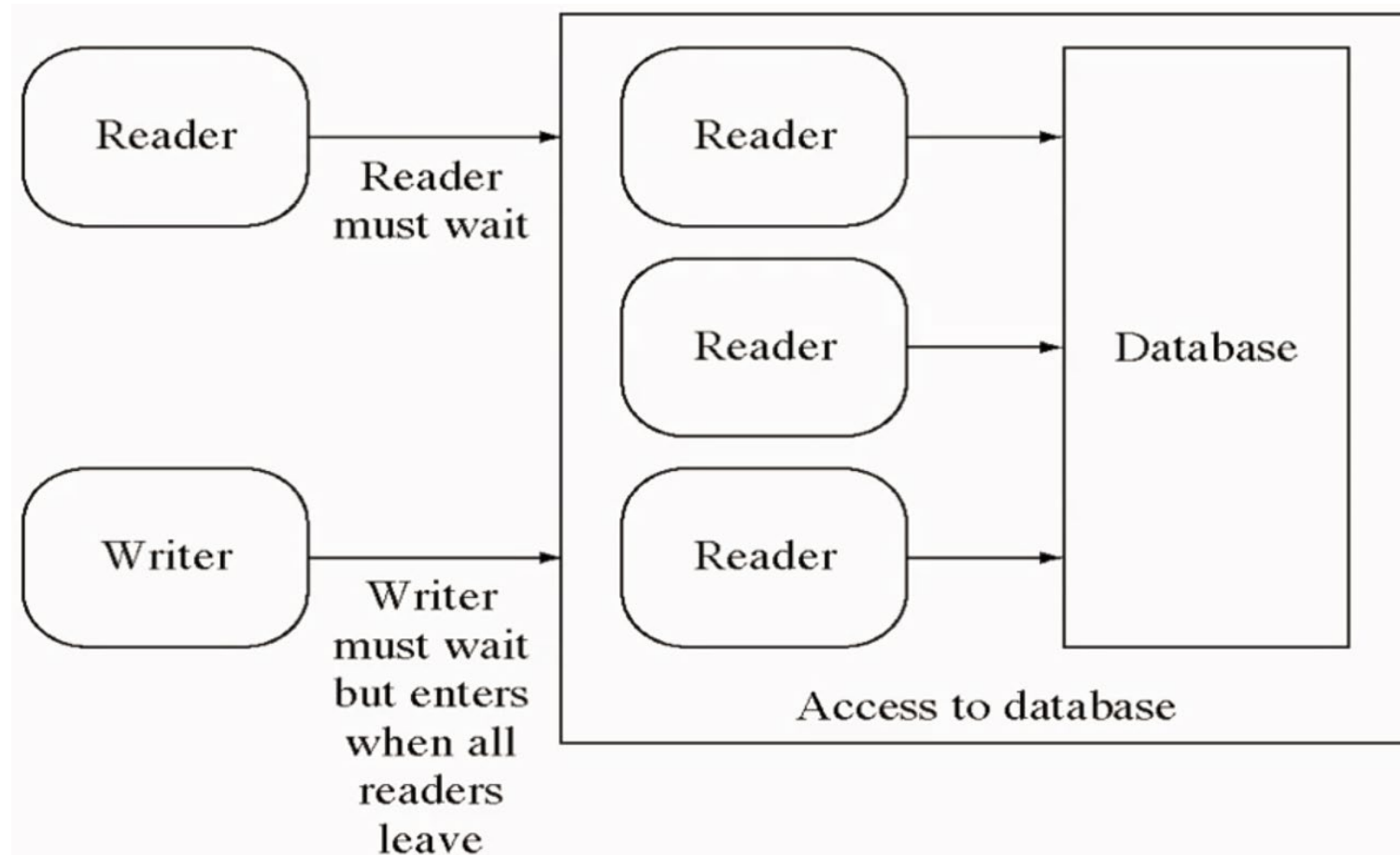
# Readers-Writers with active readers



# Readers-Writers with an active writer



# Should readers wait for waiting writer?



# Readers-Writers problem

- 1.The first readers-writers problem, requires that no reader will be kept waiting unless a writer has obtained access to the shared data
- 2.The second readers-writers problem, requires that once a writer is ready, no new readers may start reading
- 3.In a solution to the first case writers may starve; In a solution to the second case readers may starve.

# First Readers-Writers Solution

- **readcount** counter keeps track of how many processes are currently reading
- **mutex** semaphore provides mutual exclusion for updating **readcount**
- **wrt** semaphore provides mutual exclusion for the writers; it is also used by the first or last reader that enters or exits the CS



# Dining Philosophers Problem

- Five philosophers are seated around a circular table
- In front of each one is a bowl of rice
- Between each pair of people there is a chopstick (fork), so there are five chopsticks
- It takes two chopsticks to eat rice, so while  $n$  is eating neither  $n+1$  or  $n-1$  can be eating

# Dining Philosophers Problem

- Each one thinks for a while, gets the chopsticks needed, eats, and puts the chopsticks down again, in an endless cycle
- Illustrates the difficulty of allocating resources among processes without deadlock and starvation
- The challenge is to grant requests for chopsticks while avoiding deadlock and starvation
- Deadlock can occur if everyone tries to get their chopsticks at once. Each gets a left chopstick, and is stuck, because each right chopstick is someone else's left chopstick

# Dining Philosophers Solution

- Each philosopher is a process
- One semaphore per fork
  - fork: array[0..4] of semaphores
  - Initialization:  
fork[i].count := 1  
for i := 0..4

```
Process Pi:
repeat
  think;
  wait(fork[i]);
  wait(fork[i+1 mod 5]);
  eat;
  signal(fork[i+1 mod 5]);
  signal(fork[i]);
forever
```

First attempt: deadlock if each philosopher starts by picking up his left fork (chopstick)!

# Dining Philosophers Solution

- Possible solutions to avoid deadlock:
  - Allow at most four philosophers to be sitting at the table at same time
  - Odd numbered philosopher picks up left fork first, even one picks up right fork

# Weakness of the Semaphore

- The user is expected to write wait and signal in the right order
  - The user must remember to execute signal for each exit
  - Calls may be spread throughout the program
  - The logic may demand that a process must check and signal his peers
- 
- There have been a wide number of alternatives proposed
  - Monitors are one common approach

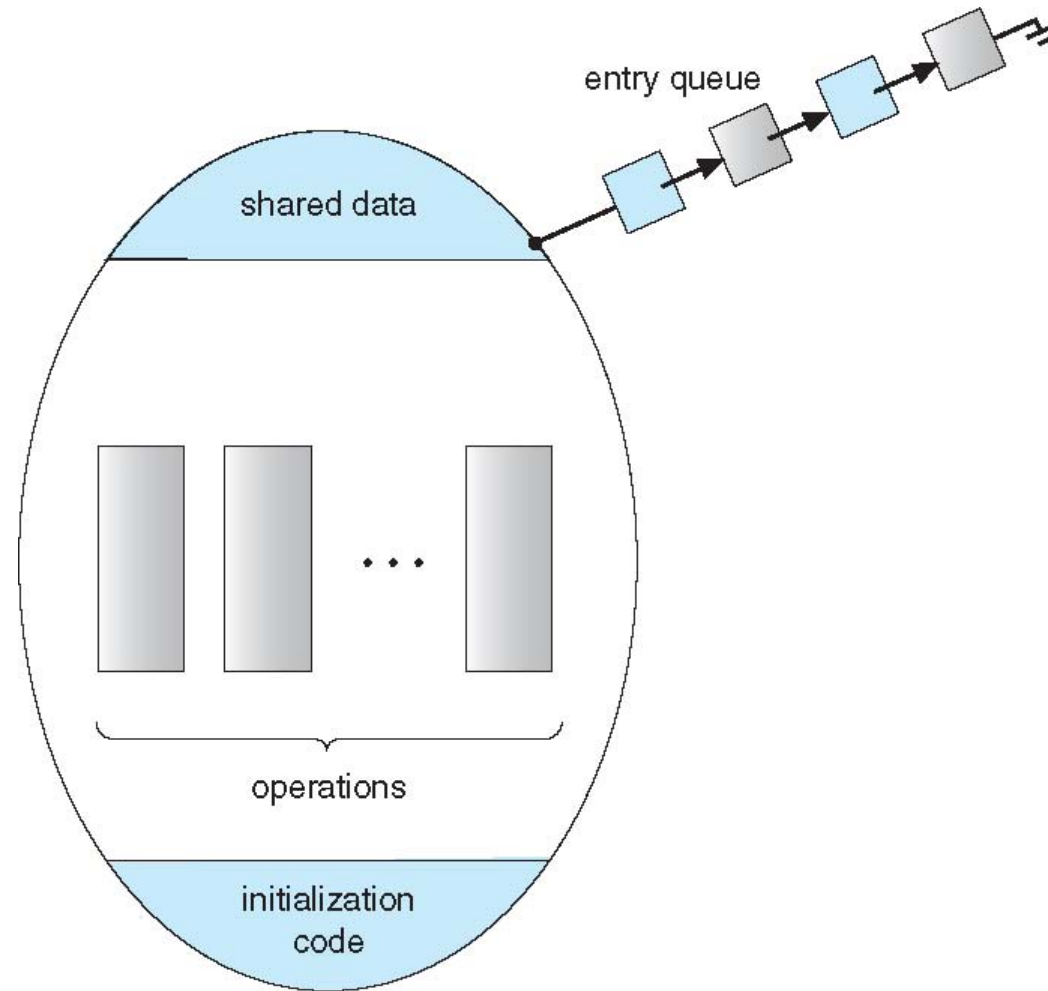
# Monitors

- A new language construct that includes synchronization
- Implemented in Java via **synchronized** keyword
- Only one process can enter a Monitor (use the entry points) at a time
- For example, compiler might generate a call to a common semaphore on entry: signal on exit
- If process P makes a monitor call, and process Q is in the monitor, P will block until Q exits

# Java Example

```
Class Account{
    private double balance;
    public Account(double deposit){
        balance = deposit;
    }
    public synchronized double getBalance(){
        return balance;
    }
    public synchronized void deposit(double amount){
        balance += amount;
    }
    public synchronized void withdraw(double amount){
        balance -= amount;
    }
}
```

# Schematic View of a Monitor

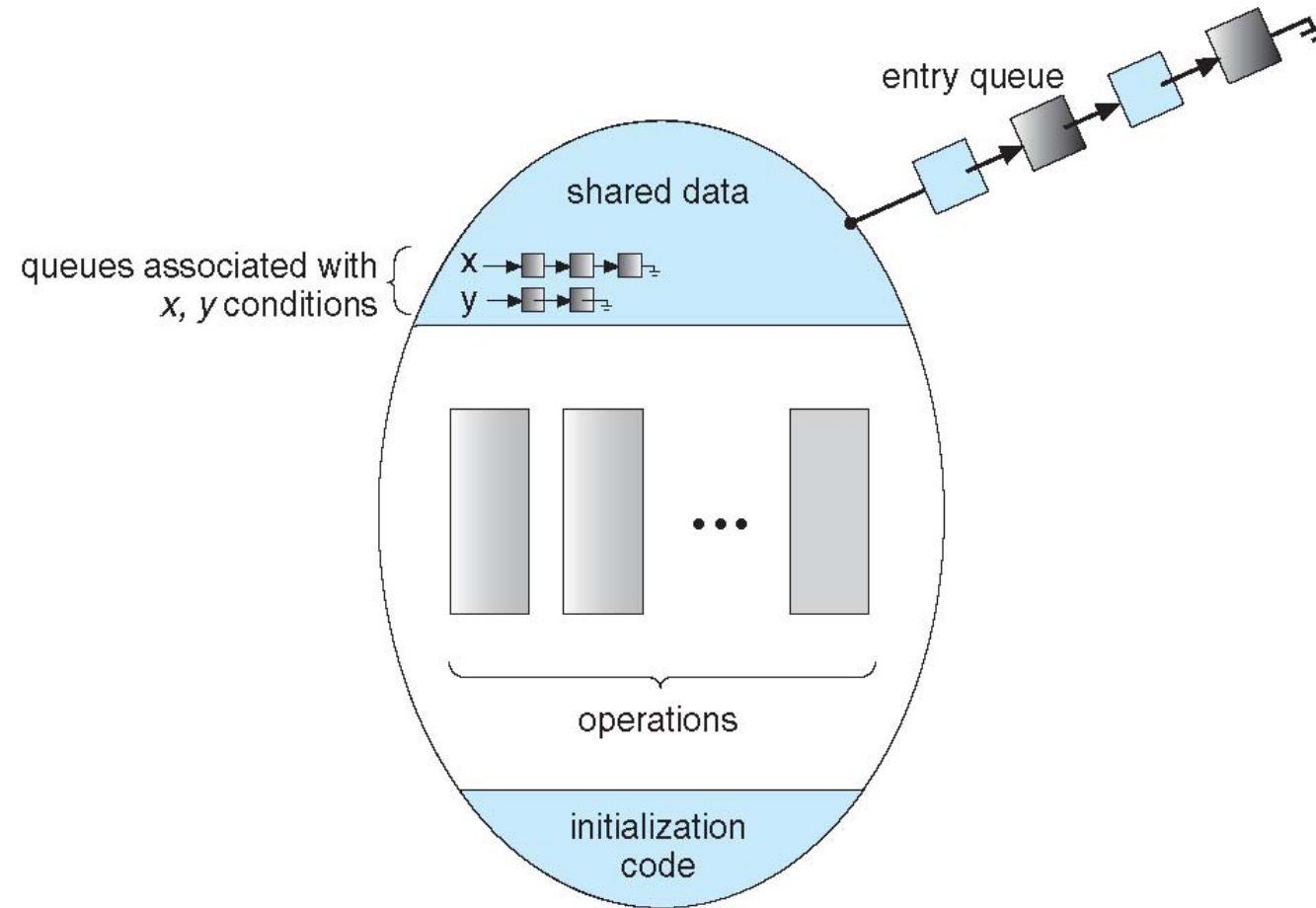




# Condition Variables

- `condition x, y;`
- Two operations are allowed on a condition variable:
  - `x.wait()` – a process that invokes the operation is suspended until `x.signal()`
  - `x.signal()` – resumes one of processes (if any) that invoked `x.wait()`
    - If no `x.wait()` on the variable, then it has no effect on the variable

# Monitor with Condition Variables



# Condition Variables Choices

- If process P invokes `x.signal()` , and process Q is suspended in `x.wait()` , what should happen next?
  - Both Q and P cannot execute in parallel. If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q either leaves the monitor or it waits for another condition
  - **Signal and continue** – Q waits until P either leaves the monitor or it waits for another condition
  - Both have pros and cons – language implementer can decide
  - Monitors implemented in Concurrent Pascal compromise
    - P executing signal immediately leaves the monitor, Q is resumed
  - Implemented in other languages including Mesa, C#, Java

# Monitor Solution to Dining Philosophers

```
monitor DiningPhilosophers
{
    enum { THINKING, HUNGRY, EATING} state [5] ;
    condition self [5];

    void pickup (int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING) self[i].wait;
    }

    void putdown (int i) {
        state[i] = THINKING;
        // test left and right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
}
```

# Solution to Dining Philosophers (Cont.)

```
void test (int i) {
    if ((state[(i + 4) % 5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i + 1) % 5] != EATING) ) {
        state[i] = EATING ;
        self[i].signal () ;
    }
}

initialization_code() {
    for (int i = 0; i < 5; i++)
        state[i] = THINKING;
}
}
```

# Solutions to Dining Philosophers (cont.)

- Each philosopher  $i$  invokes the operations `pickup()` and `putdown()` in the following sequence:

`DiningPhilosophers.pickup(i) ;`

EAT

`DiningPhilosophers.putdown(i) ;`

- No deadlock, but starvation is possible

# Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;    // (initially = 1)
semaphore next;     // (initially = 0)
int next_count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
    ...
    body of F;
    ...
if (next_count > 0)
    signal( $\bar{\text{next}}$ )
else
    signal(mutex);
```

- Mutual exclusion within a monitor is ensured

# Monitor Implementation – Condition Variables

- For each condition variable **x**, we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation `x.wait` can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```



# Monitor Implementation (cont.)

- The operation `x.signal` can be implemented as:

```
if (x_count > 0) {  
    next_count++;  
    signal(x_sem);  
    wait(next);  
    next_count--;  
}
```

# Equivalence

- Monitors and semaphores have equivalent power
- Anything you can do with Monitors, you can do with semaphores
  - Proof: we can implement a monitor with a semaphore
- Anything that you can do with semaphores, you can do with Monitors
  - Proof: we can implement a semaphore with a monitor

# Summary

- We have seen two problems
  - Critical Sections – cannot both be modifying variable
  - Synchronization – must define ordering
- Often, our problems are a combination of the two
  - Readers/writers share storage, and readers should wait if there are writers waiting
- It is difficult to use semaphores correctly
- While there has been language support for Monitors for some time, standard UNIX still only supports semaphores (man sem\_open)

# Exit Slips

- Take 1-2 minutes to reflect on this lecture
- On a sheet of paper write:
  - One thing you learned in this lecture
  - One thing you didn't understand

# Next class

- We will discuss:
  - Deadlocks
- Reading assignment:
  - SGG: Ch. 8

# Acknowledgment

- The slides are partially based on the ones from
  - The book site of *Operating System Concepts (Ninth Edition)*: <http://os-book.com/>