

# CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Fall 2021 L12

Synchronization (Chap 6, 7)



**Slides based on**

- Text by Silberschatz, Galvin, Gagne
- Various sources

# FAQ

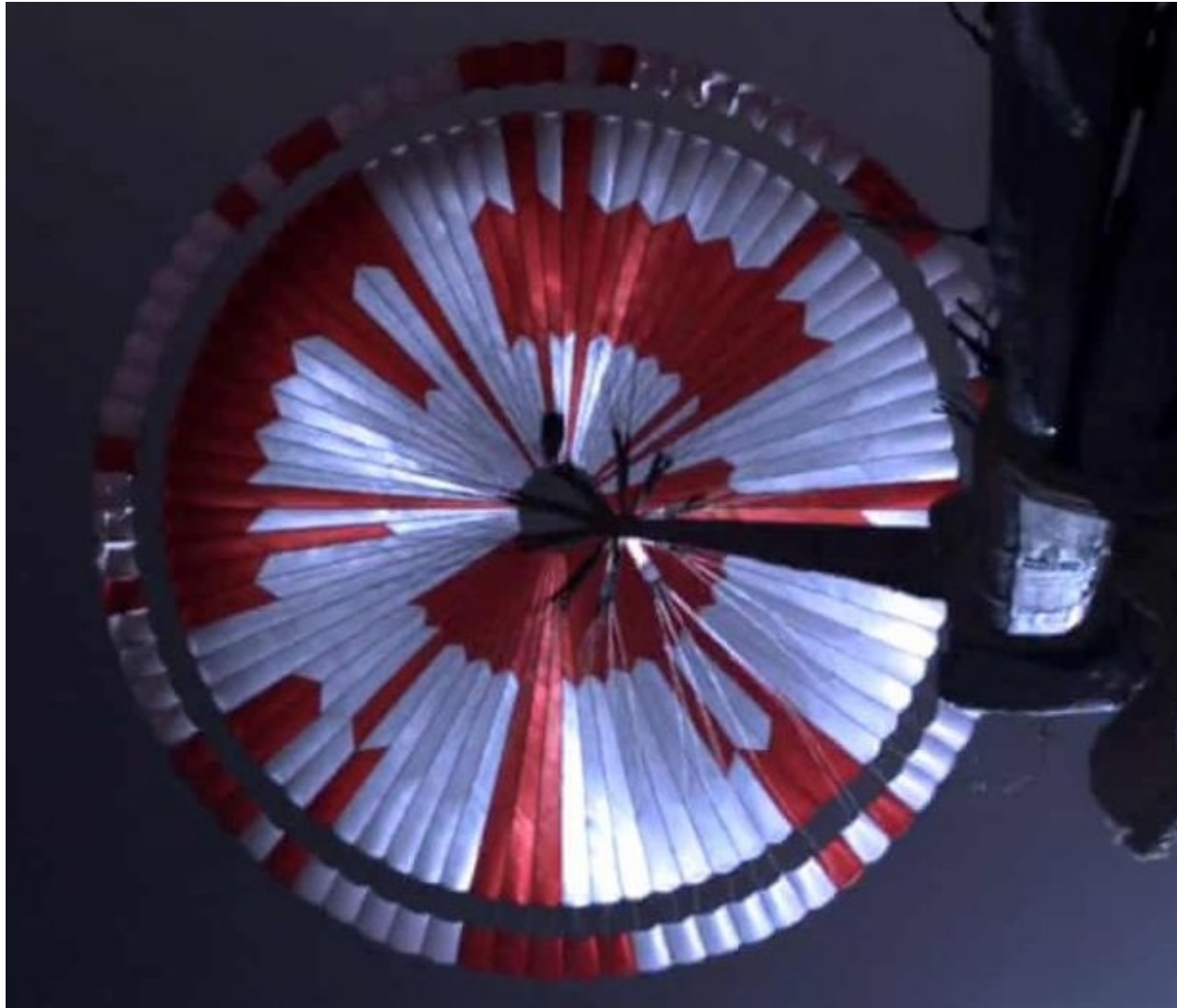
Notes: we are using the terms in a generic way. There are specific implementations for [POSIX](#) and [Java](#).

- **Atomic instructions:** Hardware (Assembly/Machine), not *high level* like C
- **Mutex (0 or 1):** for mutual exclusion (lock). Owned by the locking process which acquires/releases by
  - **wait( )** get the resource or join the waiting list
  - **signal( )** release the resource, and wake up a process
- **Semaphores (any integer value):** general, may be used for counting resources/waiting processes. Shared. Applicable to different types of synchronization problems.
  - 0: no waiting threads
  - Positive: no waiting threads, a wait operation would not put the invoking thread in queue.
  - Negative: number of processes/threads waiting
- **Semaphore implementation**
  - Hardware/software implementations to ensure wait() and signal( ) atomic.
- **Semaphore usage:** see POSIX/Java documentation.

# Project

- See [Document](#): Schedule/Proj Proposal or Canvas/Assignments
- **Choices:** Research (topics provided) or development (IoT). Some research/original thinking required for either.
- **Deadlines:** subject to revision.
  - D1. Team composition and idea proposal, Fri 10/01/2021
  - D2. Progress report, Thurs 11/04/2021
  - D3. Slides and final reports, Thurs 12/02/2021
  - D4. Presentations/demos 12/06-12/08 as arranged
  - D5: Peer Reviews due 12/11/2021 Sat

# What does the Mars parachute say?



# Classical Problems of Synchronization

- **Classical problems**
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
- **Bounded buffer Review**
  - n buffers, each can hold one item
  - A binary semaphore: mutex
    - Provides mutual exclusion for accesses to buffer pool
    - Initialized to 1
  - Two counting semaphores
    - empty: Number of empty slots available, Initialized to n
    - full: Number of filled slots available n, Initialized to 0

# 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. No readers permitted when writer is accessing the data.
- Data set, integer **read\_count** (number of readers)
  - Semaphore **rw\_mutex** (writing), Semaphore **mutex** (for read\_count)

# Readers-Writers Problem

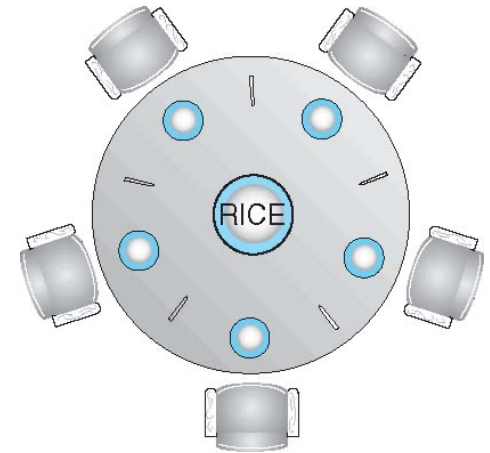
- The structure of a reader process

```
do {  
    wait(mutex);  
    read_count++;  
    if (read_count == 1)           if the first reader process  
        wait(rw_mutex);           wait for writer to finish  
    signal(mutex);  
    ...  
    /* reading is performed */  
    ...  
    wait(mutex);  
    read_count--;  
    if (read_count == 0)           if the only reader finishes  
        signal(rw_mutex);         writer can get in  
    signal(mutex);  
} while (true);
```

First reader needs to wait for the writer to finish.  
If other readers are already reading, a new reader  
Process just goes in.

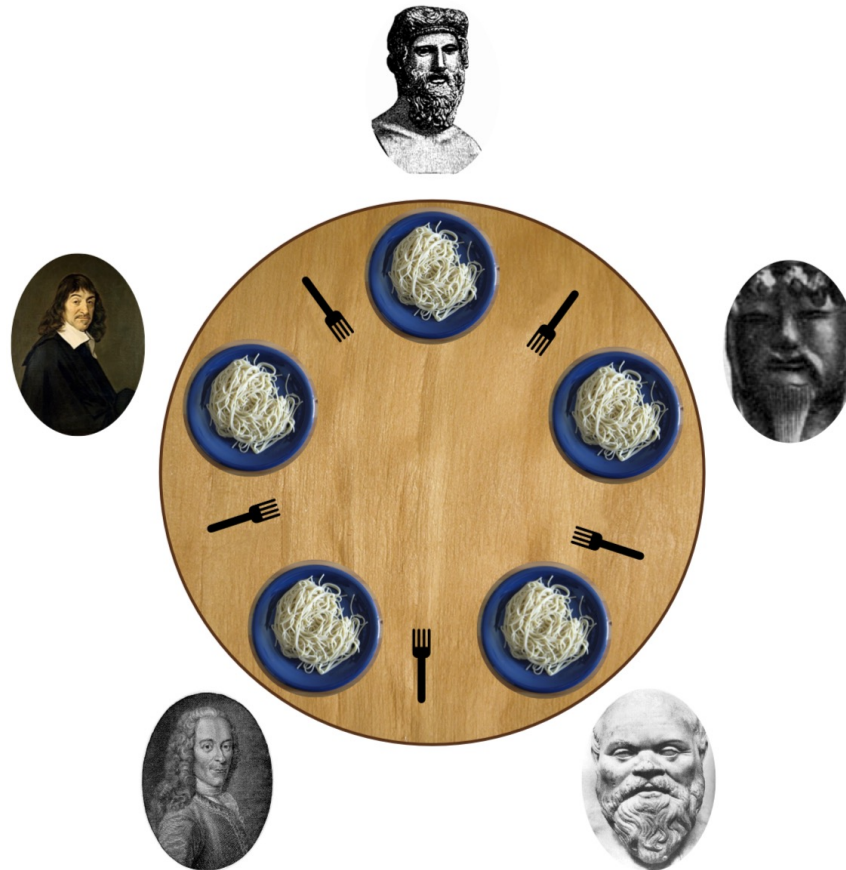
# Dining-Philosophers Problem

- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat,
  - then release both when done
- Each chopstick is a semaphore
  - Grab by executing wait ( )
  - Release by executing signal ( )
- Shared data
  - Bowl of rice (data set)
  - Semaphore **chopstick** [5] initialized to 1





# Dining-Philosophers Problem



Plato, Confucius, Socrates, Voltaire and Descartes

# Dining-Philosophers Problem Algorithm: Simple solution?

- The structure of Philosopher *i*:

```
do {  
    wait (chopstick[i] );  
    wait (chopStick[ (i + 1) % 5] );  
  
    // eat  
  
    signal (chopstick[i] );  
    signal (chopstick[ (i + 1) % 5] );  
  
    // think  
  
} while (TRUE);
```

- What is the problem with this algorithm?
  - If all of them pick up the the left chopstick first -  
Deadlock

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table (with the same 5 forks).
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section).
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

# Problems with Semaphores

- Incorrect use of semaphore operations:
  - Omitting of wait (mutex)
    - Violation of mutual exclusion
  - or signal (mutex)
    - Deadlock!
- Solution:
  - Monitors: a higher level implementation of synchronization

# Monitors

# Monitors

**Monitor:** A high-level abstraction that provides a convenient and effective mechanism for process synchronization

- Abstract data type, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
  - Automatically provide mutual exclusion
  - Implement waiting for conditions
- Queues:
  - for entry
  - for each condition
- Originally proposed for Concurrent Pascal 1975
- Directly supported by Java (see self exercise) but not C

# Monitors

- Only one process may be active in the monitor.
- A generic monitor construct is used here. Implementation varies by language.

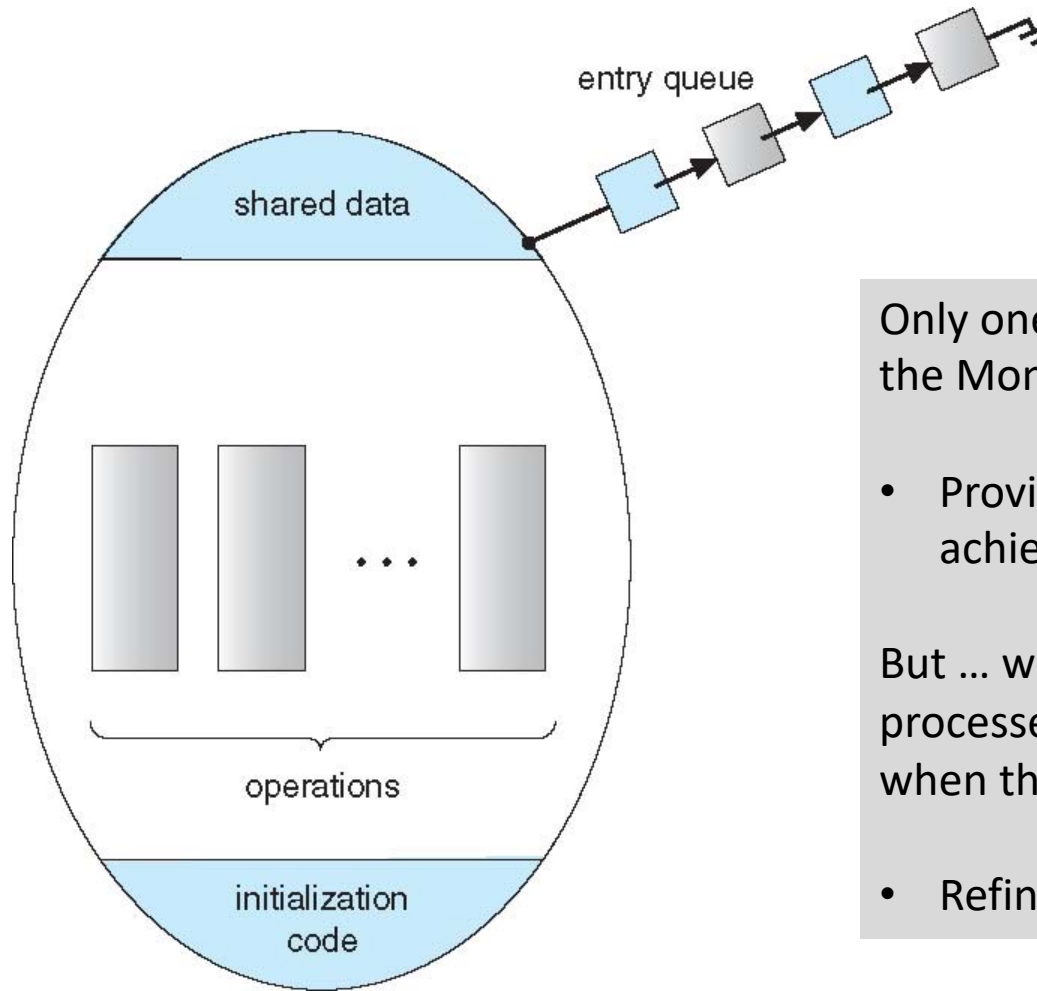
```
monitor monitor-name
{
    // shared variable declarations

    procedure P1 (...) { ... }

    procedure Pn (...) {.....}

    Initialization code (...) { ... }
}
}
```

# Preliminary Schematic view of a Monitor



Only one process/thread in the Monitor

- Provides an easy way to achieve mutual exclusion

But ... we also need a way for processes to **block** when they cannot proceed.

- Refinement next ...

*Shows 4 processes waiting in the queue.*



# Condition Variables

Some actions need some conditions to go ahead.

The **condition** construct

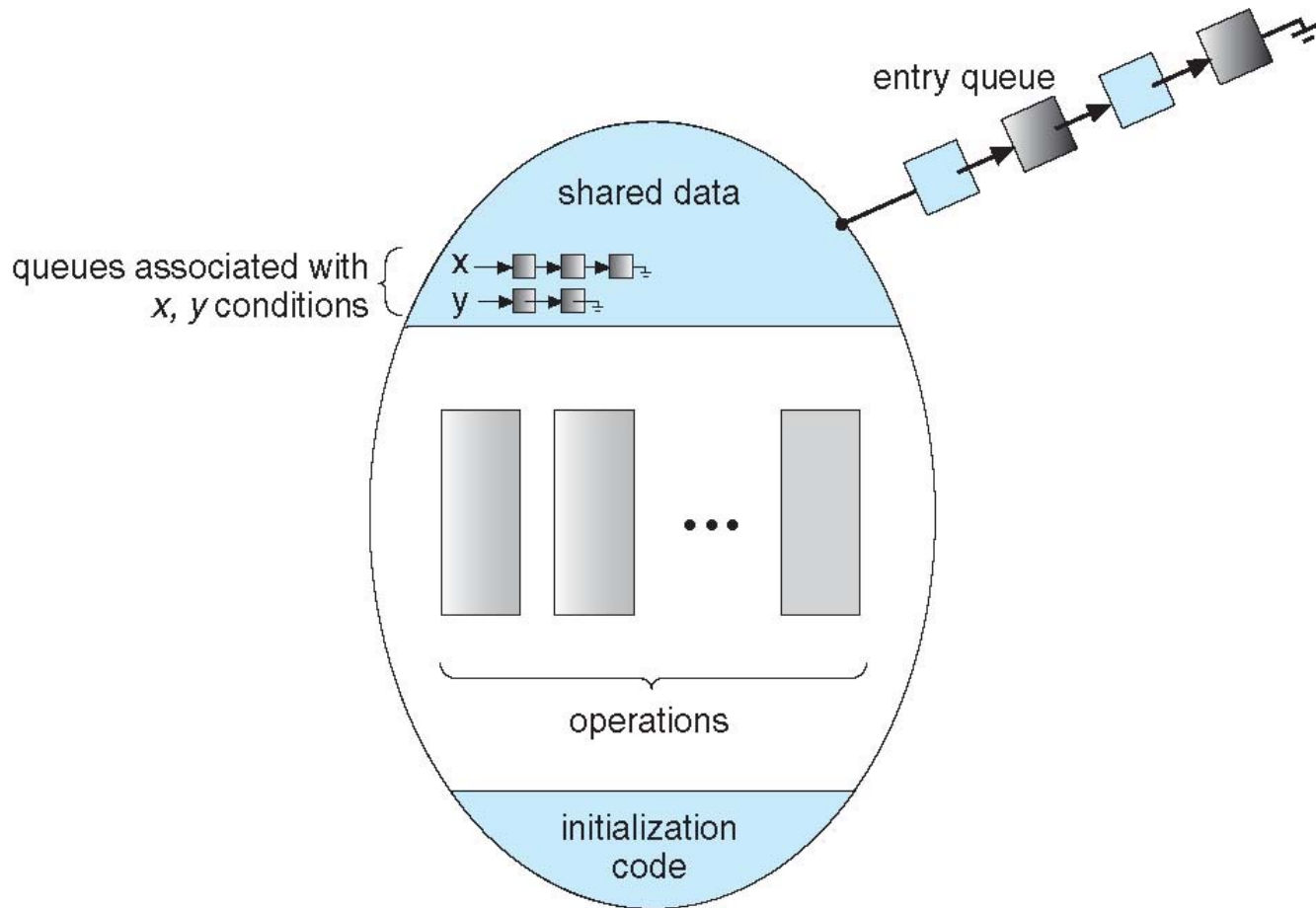
- **condition** **x**, **y**;
- Two operations are allowed on a condition variable:
  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
    - If no **x.wait()** on the condition variable,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

Compare with semaphore.  
Here no integer value is associated.

## Difference between the signal() in semaphores and monitors

- Condition variables in Monitors: Not persistent
  - If a signal is performed and no waiting threads?
    - Signal is simply ignored
  - During subsequent wait operations
    - Thread (or process) blocks
- Compare with semaphores
  - Signal increments semaphore value even if there are no waiting threads
    - Future wait operations would immediately succeed!

# 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 ('75)** compromise
    - P executing signal immediately leaves the monitor, Q is resumed
    - Implemented in other languages including C#, Java

# Monitor Solution to Dining Philosophers: Deadlock-free

```
enum {THINKING, HUNGRY, EATING} state[5];
```

- `state[i] = EATING` only if
  - `state[(i+4)%5] != EATING && state[(i+1)%5] != EATING`
- `condition self[5]`
  - Delay self when **HUNGRY but unable** to get chopsticks

## Sequence of actions

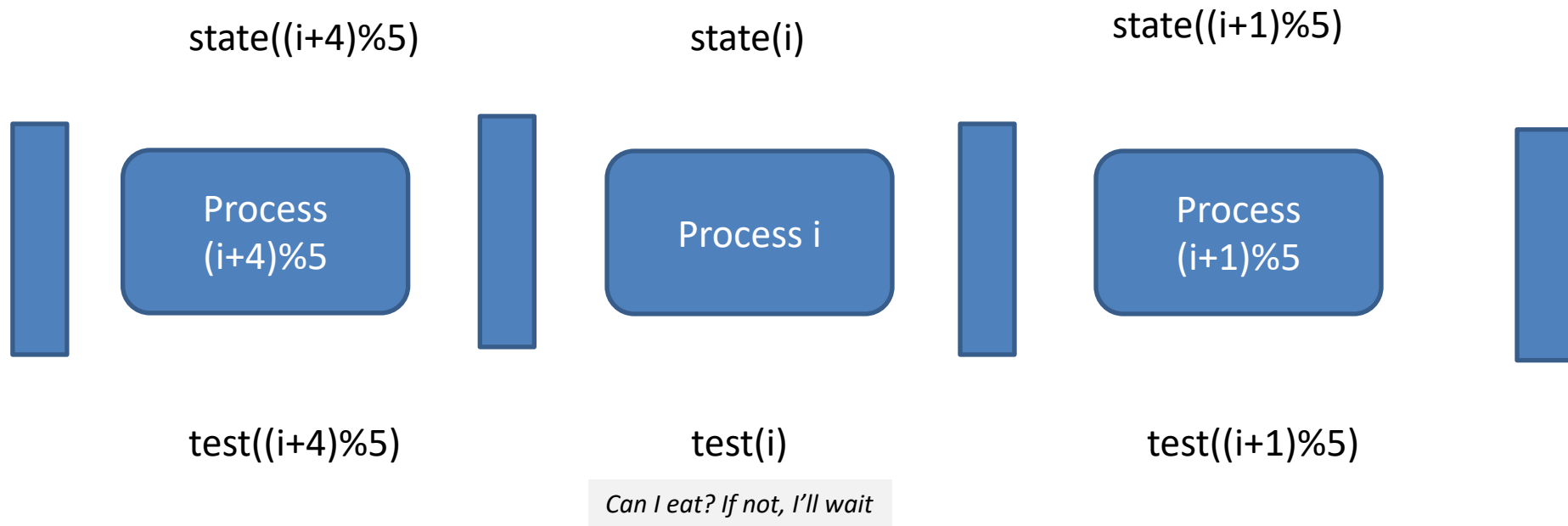
- Before eating, must invoke `pickup()`
  - May result in suspension of philosopher process
  - After completion of operation, philosopher may eat

```
think  
DiningPhilosophers.pickup(i);  
eat  
DiningPhilosophers.putdown(i);  
think
```



# Monitor Solution to Dining Philosophers: Deadlock-free

```
enum { THINKING, HUNGRY, EATING } state[5];
```



# The pickup() and putdown() operations

## monitor DiningPhilosophers

```
{  
    enum { THINKING, HUNGRY, EATING} state [5] ;  
    condition self [5];  
  
    void pickup (int i) {  
        state[i] = HUNGRY;  
        test(i);    //below  
        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);  
    }  
}
```

Suspend self if  
unable to acquire  
chopstick

Eat only if HUNGRY  
and Person on Left  
AND Right  
are not eating

Check to see if person  
on left or right can use  
the chopstick

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

Signal a process that  
was suspended while  
trying to eat

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

```
}
```

# Possibility of starvation

- Philosopher  $i$  can starve if eating periods of philosophers on left and right overlap
- Possible solution
  - Introduce new state: STARVING
  - Chopsticks can be picked up if no neighbor is starving
    - Effectively wait for neighbor's neighbor to stop eating
    - REDUCES concurrency!



# Monitor Implementation of Mutual Exclusion

For each monitor

- Semaphore mutex initialized to 1
- Process must execute
  - wait(mutex) : Before entering the monitor
  - signal(mutex): Before leaving the monitor

# Resuming Processes within a Monitor

- If several processes queued on condition  $x$ , and  $x.\text{signal}()$  is executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form  $x.\text{wait}(c)$ 
  - Where  $c$  is **priority number**
  - Process with lowest number (highest priority) is scheduled next

# Single Resource allocation

- Allocate a single resource among competing processes using priority numbers that specify the maximum time a process plans to use the resource

```
R.acquire(t) ;  
    ...  
access the resource;  
    ...  
R.release;
```

- Where R is an instance of type **ResourceAllocator**
- **A monitor based solution next.**

# A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
```

```
{
```

```
    boolean busy;
```

```
    condition x;
```

```
    void acquire(int time) {
```

```
        if (busy)
```

```
            x.wait(time);
```

```
        busy = TRUE;
```

```
    }
```

```
    void release() {
```

```
        busy = FALSE;
```

```
        x.signal();
```

```
    }
```

```
    initialization code() {
```

```
        busy = FALSE;
```

```
    }
```

```
}
```

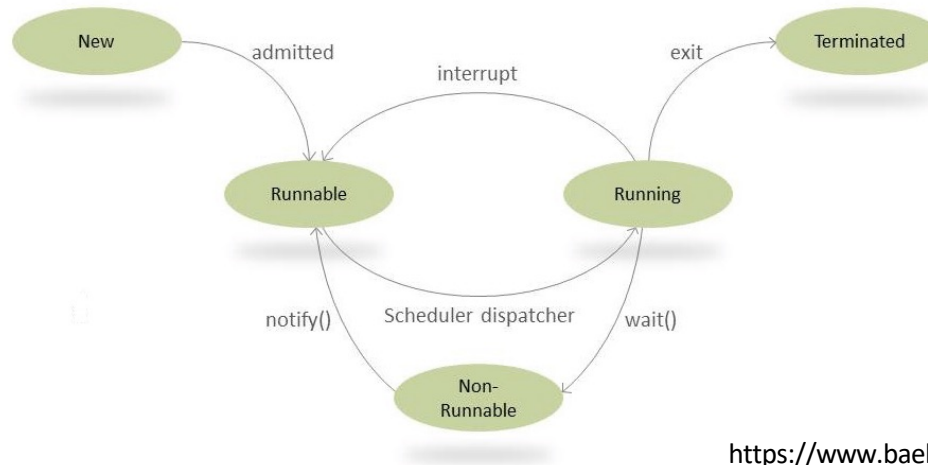
Sleep, Time used  
to prioritize  
waiting  
processes

Wakes up  
one of the  
processes

# Java Synchronization

- For simple synchronization, Java provides the `synchronized` keyword
  - synchronizing methods  
`public synchronized void increment( ) { c++; }`
  - synchronizing blocks  
`synchronized(this) {  
 lastName = name;  
 nameCount++;  
}`
- `wait()` and `notify()` allows a thread to wait for an event. A call to `notifyAll()` allows all threads that are on `wait()` with the same lock to be notified.
- `notify()` notifies one thread from a pool of identical threads, `notifyAll()` when threads have different purposes
- For more sophisticated locking mechanisms, starting from Java 5, the package `java.concurrent.locks` provides additional capabilities.

# Java Synchronization



Each object automatically has a monitor (mutex) associated with it

- When a method is synchronized, the runtime must obtain the lock on the object's monitor before execution of that method begins (and must release the lock before control returns to the calling code)

`wait()` and `notify()` allows a thread to wait for an event.

- **`wait()`**: Causes the current thread to wait until another thread invokes the `notify()` method or the `notifyAll()` method for this object.
- **`notify()`**: Wakes up a single thread that is waiting on this object's monitor. If any threads are waiting on this object, one of them is chosen to be awakened.
- A call to **`notifyall()`** allows all threads that are on `wait()` with the same lock to be released, they will run in sequence according to priority.

# Java Synchronization: Dining Philosophers

```
public synchronized void pickup(int i)
    throws InterruptedException {
    setState(i, State.HUNGRY);
    test(i);
    while (state[i] != State.EATING) {
        this.wait();
        // Recheck condition in loop,
        // since we might have been notified
        // when we were still hungry
    }
}
```

```
public synchronized void putdown(int i) {
    setState(i, State.THINKING);
    test(right(i));
    test(left(i));
}
```

```
private synchronized void test(int i) {
    if (state[left(i)] != State.EATING &&
        state[right(i)] != State.EATING &&
        state[i] == State.HUNGRY)
    {
        setState(i, State.EATING);
        // Wake up all waiting threads
        this.notifyAll();
    }
}
```

# Synchronization Examples

- Solaris
- Windows
- Linux
- Pthreads



# Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

# Windows Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** user-land which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

# Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- Linux provides:
  - Semaphores
  - atomic operations on integers
  - spinlocks
  - reader-writer versions of both
- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

# Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variable
- Non-portable extensions include:
  - read-write locks
  - spinlocks

# Alternative Approaches

- Transactional Memory
- OpenMP
- Functional Programming Languages

# Transactional Memory

- A **memory transaction** is a sequence of read-write operations to memory that are performed atomically without the use of locks.

```
void update() {  
    atomic{  
        /* modify shared data*/  
    }  
}
```

May be implemented by hardware or software.

# OpenMP

- OpenMP is a set of compiler directives and API that support parallel programming.

```
void update(int value)
{
    #pragma omp critical
    {
        count += value
    }
}
```

The code contained within the `#pragma omp critical` directive is treated as a critical section and performed atomically.

# CS370 Operating Systems

Colorado State University

Yashwant K Malaiya

Spring 2020



## Deadlock

Slides based on

- Text by Silberschatz, Galvin, Gagne
- Various sources



# Chapter 8: Deadlocks

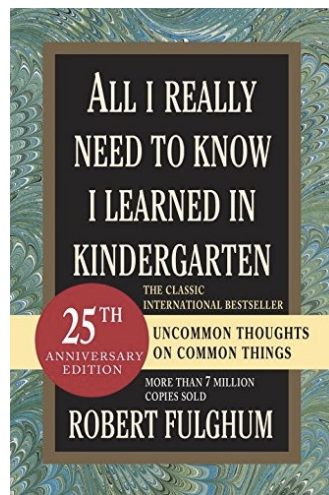
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
  - Deadlock Prevention
  - Deadlock Avoidance resource-allocation
  - Deadlock Detection
  - Recovery from Deadlock

# Deadlock

- Can you give a real life example of a deadlock?

# A Kansas Law

- Early 20<sup>th</sup> century Kansas Law
  - “When *two trains approach each other at a crossing, both shall come to a full stop and neither shall start up again until the other has gone*”
- [Story of the two silly goats](#): Aesop 6<sup>th</sup> cent BCE?



# A contemporary example

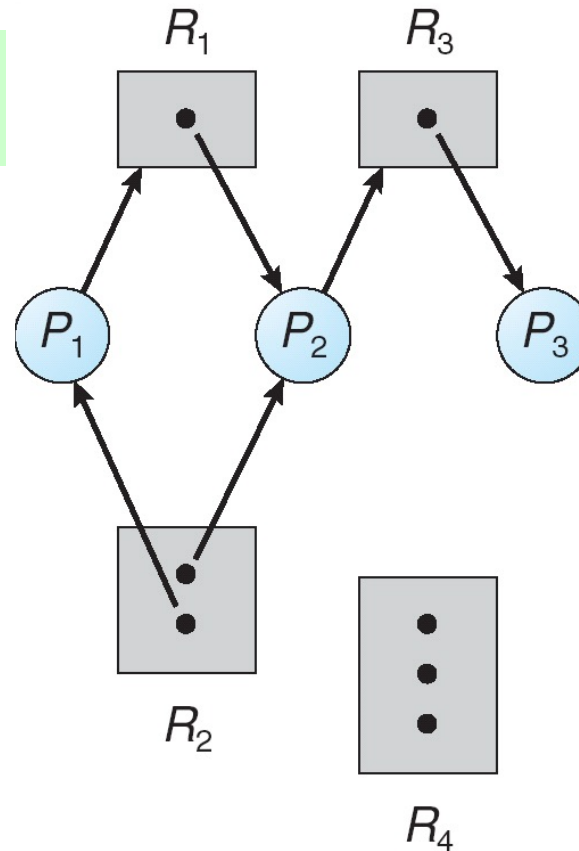


# System Model

- System consists of resources
- Resource types  $R_1, R_2, \dots, R_m$   
*CPU cycles, memory space, I/O devices*
- Each resource type  $R_i$  has  $W_i$  instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

# Example of a Resource Allocation Graph

P1 holds an instance of R2, and is requesting R1 ..



Does a deadlock exist here?

$P_3$  will eventually be done with  $R_3$ , letting  $P_2$  use it.

Thus  $P_2$  will be eventually done, releasing  $R_1$ . ...