CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Fall 2025 L11
Synchronization

Slides based on

- **Text by Silberschatz, Galvin, Gagne**
- **Various sources**

Semaphores by Dijkstra

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two **indivisible (atomic)** operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`** **based on Dutch words**
- Definition of the **`wait()` operation**

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

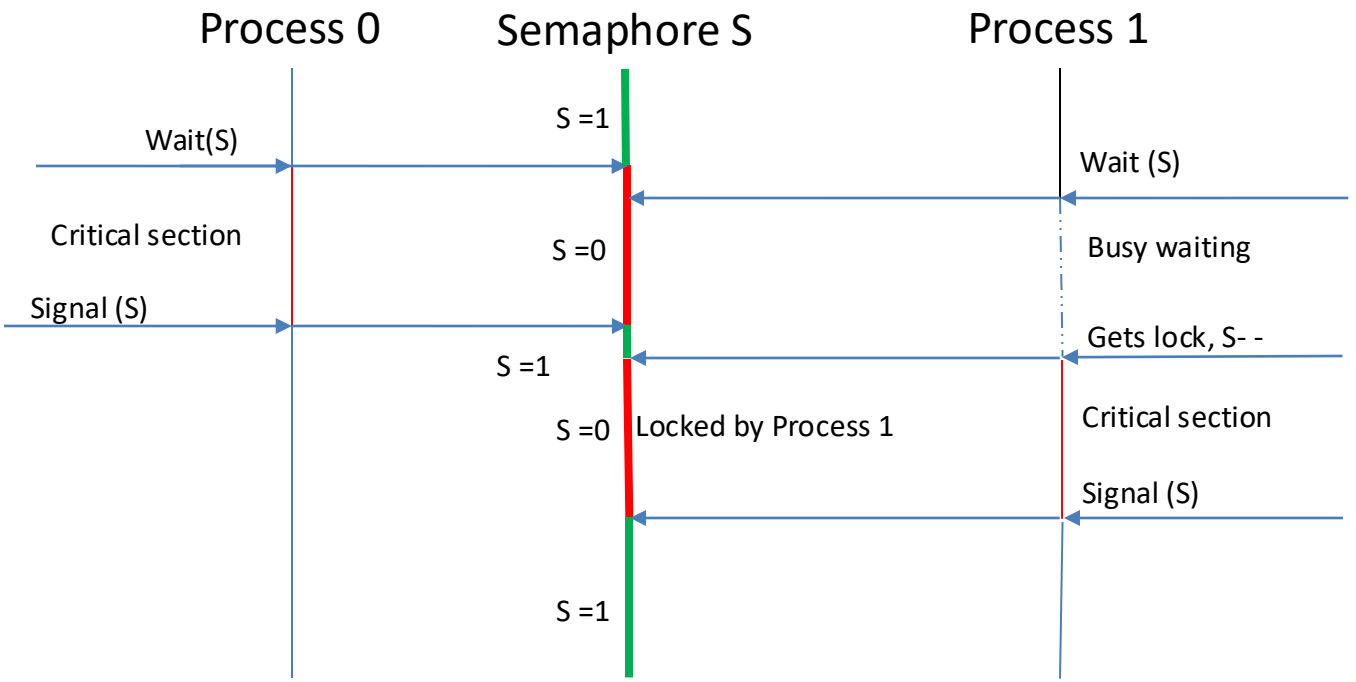
- Definition of the **`signal()` operation**

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

Waits until another process makes S=1

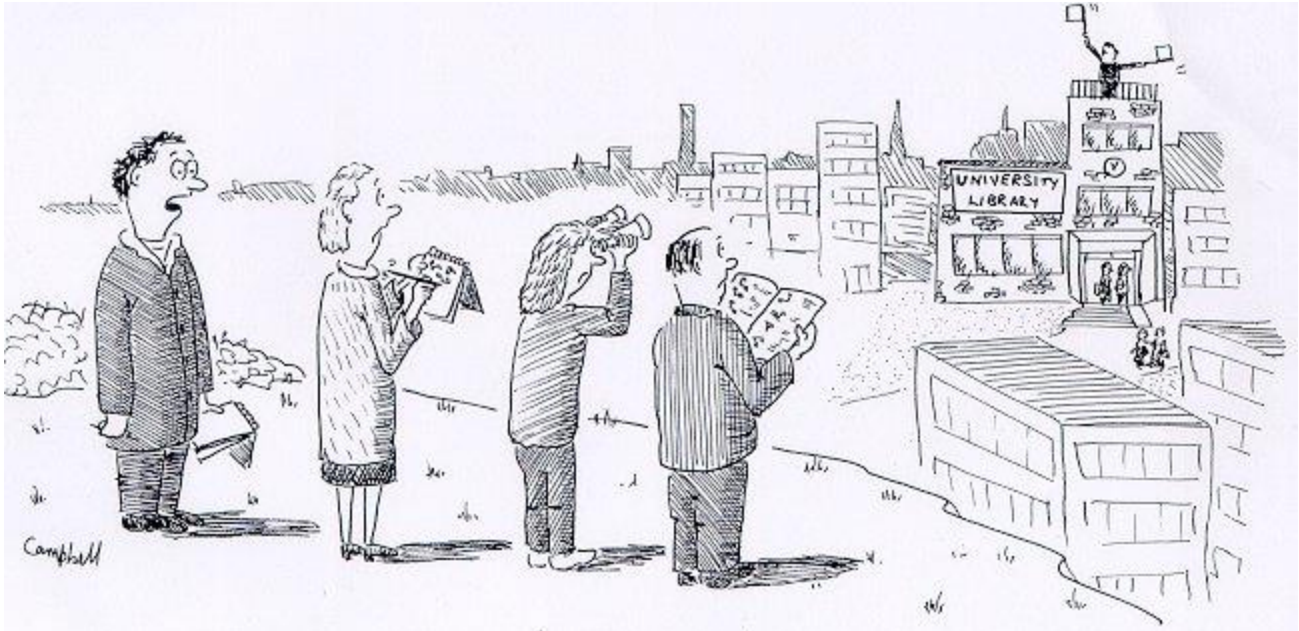
Binary semaphore: When s is 0 or 1, it is a mutex lock

Wait(S) and Signal (S)



Colorado State University

Semaphores



I was hoping the distance learning service might use more up-to-date technology

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Practically same as a **mutex lock**
- Can solve various synchronization problems
- Ex: Consider P_1 and P_2 that requires event S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0 i.e not available

```
P1:                          P2:
   S1;                           wait(synch);
   signal(synch);                S2;
```

- Can implement a counting semaphore ***S*** as a binary semaphore

The counting semaphore

- **Controls access to a finite set of resources**
- Initialized to the number of resources
- Usage:
  - Wait (S): to use a resource
  - Signal (S): to release a resource
- When all resources are being used: S == 0
  - Block until S > 0 to use the resource

Applicable to different types of synchronization problems.

0: no waiting threads (or processes)

Positive: no waiting threads, a wait operation would not put the invoking thread in queue.

Negative: number of threads waiting

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section
  - Could now have **busy waiting** in critical section implementation
    - But implementation code is short
    - Little busy waiting if critical section rarely occupied
- Note that some applications may spend lots of time in critical sections and therefore this is not a good solution
- Alternative: block and wakeup (next slide)

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

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

Implementation with no Busy waiting (Cont.)

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

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

If value < 0
abs(value) is the number of waiting processes

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

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
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

  - P0 executes wait(s), P1 executes wait(Q)
    - P0 must wait till P1 executes signal(Q)
    - P1 must wait till P0 executes signal(S) Deadlock!

Priority Inversion

- **Priority Inversion** – Scheduling problem when lower-priority process P_L holds a lock needed by higher-priority process P_H.
  - The low priority task may be preempted by a medium priority task P_M which does not use the lock, causing P_H to wait because of P_M.

Mars pathfinder
Mission problem 1997

- Solved via **priority-inheritance protocol**
  - Process accessing resource needed by higher priority process Inherits higher priority till it finishes resource use
  - Once done, process reverts to lower priority

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem
- Monitors: higher level handling f synchronization

Bounded-Buffer Problem

- *n* buffers, each can hold one item
- Binary semaphore (mutex)
  - Provides mutual exclusion for accesses to buffer pool
  - Initialized to 1
- Counting semaphores
  - empty: Number of empty slots available
    - Initialized to n
  - full: Number of filled slots available n
    - Initialized to 0

3 semaphores needed, 1 binary, 2 counting

Bounded-Buffer : Note

- Producer and consumer must be ready before they attempt to enter critical section
- Producer readiness?
  - When a slot is available to add produced item
    - wait(empty)
      - empty is initialized to n
- Consumer readiness?
  - When a producer has added new item to the
    - wait(full)
      - full initialized to 0

empty: Number of empty slots available
wait(empty) wait until at least 1 empty

full: Number of filled slots available
wait(full) wait until at least 1 full

Bounded Buffer Problem (Cont.)

The structure of the producer process

empty: initialized to n
full: initialized to 0

```
do {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);          wait till slot available
    wait(mutex);      Allow producer OR consumer to (re)enter critical section
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex); Allow producer OR consumer to (re)enter critical section
    signal(full);         signal consumer that a slot is available
} while (true);
```

Bounded Buffer Problem (Cont.)

The structure of the consumer process

empty: initialized to n
full: initialized to 0

```
Do {
    wait(full);  wait till slot available for consumption
    wait(mutex);  Only producer OR consumer can be in critical section
        ...
    /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);  Allow producer OR consumer to (re)enter critical section
    signal(empty);  signal producer that a slot is available to add
        ...
    /* consume the item in next consumed */
        ...
 } while (true);
```

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.
- Several variations of how readers and writers are considered – all involve some form of priorities

Readers-Writers Problem

- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1 (mutual exclusion for writer)
  - Semaphore **`mutex`** initialized to 1 (mutual exclusion for read_count)
  - Integer **`read_count`** initialized to 0 (how many readers?)

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
do {
        wait(rw_mutex);
                ...
      /* writing is performed */
                ...
        signal(rw_mutex);
} while (true);
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
do {
     wait(mutex);
       read_count++;
       if (read_count == 1)
            wait(rw_mutex);
     signal(mutex);
          ...
       /* reading is performed */
          ...
     wait(mutex);
       read count--;
       if (read_count == 0)
            signal(rw_mutex);
     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.

mutex for mutual exclusion to read_count

Cannot read if writer is writing

When:
writer in critical section
and if n readers waiting
1 is queued on rw_mutex
(n-1) queued on mutex

When the last reader leaves, a writer can go in.

Readers-Writers Problem Variations

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

Project

Options

- A. Research
- B. Development

Deliverable *D1 Team composition and idea proposal* specified separately in the document Fall 2025 Term Project .
Similarly, D2, D3, D4 and D5 are specified.

You have to do some research for both of them.

Research: Search Databases

Specific sources: database indexes

- Google Scholar
  - Forward links: Paper X Cited by
  - Backward Links: Paper X cites
- Researcher sites
  - Personal/Group Website
  - DBLP
  - Google Scholar: researcher
- CSU Library etc.

General (*accessible through CSU Library*)

- ACM Digital Library
- IEEEXplore Digital Library
- ScienceDirect etc

Research: Source types

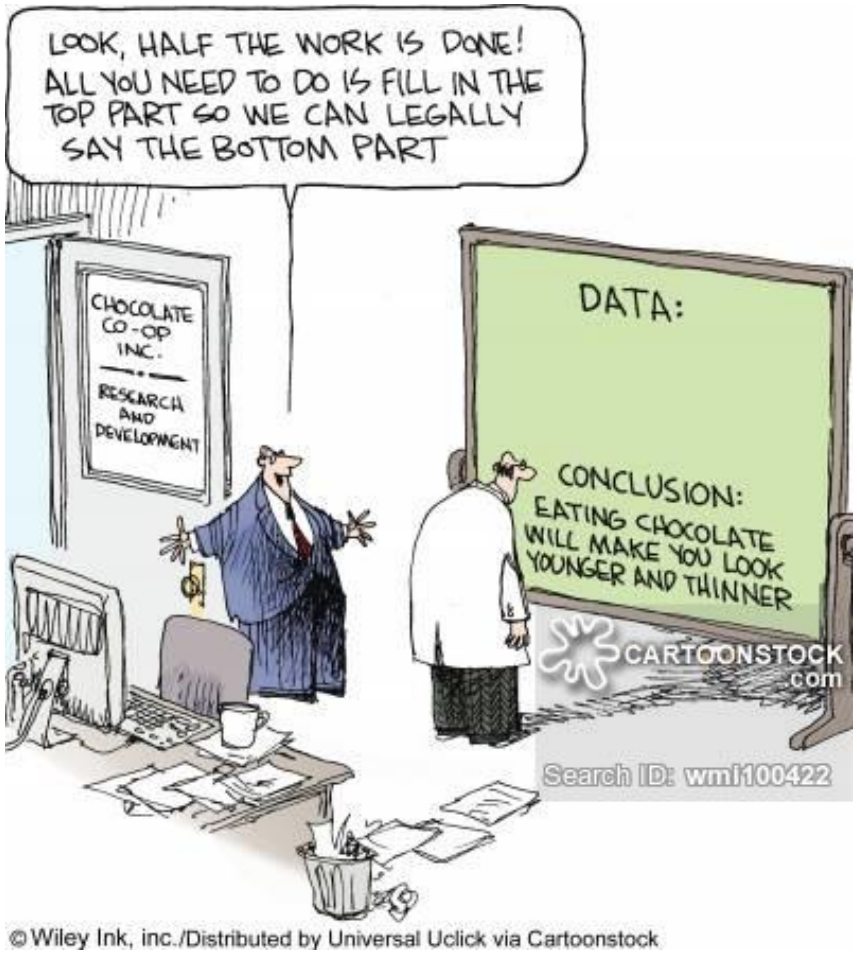
- Journals: published several times a year
  - Rigorously reviewed, long publication delay
  - Journal, Transactions, …
- Conferences: held once a year, proceedings published
  - Conference, Symposium, …
- Research groups
  - Industry, academic, consultants: web site
- News, Industry publications
  - Magazines, blogs, white papers, product website
- Books: often well-known stuff

Research: How to Read a Paper: THE THREE-PASS APPROACH

- The first pass: Read
  - the title, abstract, and introduction
  - section and sub-section headings, but ignore everything else
  - the conclusions
- The second pass: Read
  - figures, diagrams and other illustrations
  - mark relevant unread references for further reading
  - Do you need to read it in detail?
- The third pass: Read critically
  - identify and challenge assumption and views
  - Loop up references needed

Keshav, S., How to Read a Paper, ACM SIGCOMM, http://ccr.sigcomm.org/online/files/p83-keshavA.pdf

Research: Avoid Prior Bias



Evaluating research

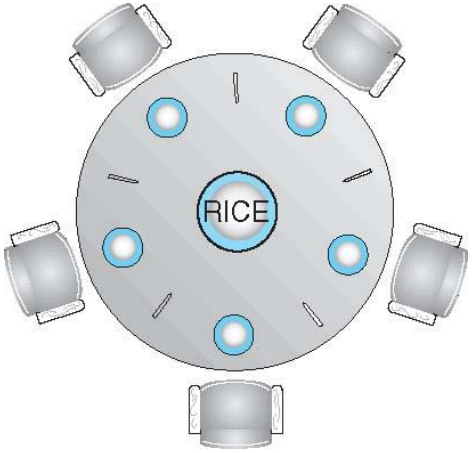
- These are the attributes generally evaluated
- Novelty/interest/Applicability
- Technical: Extent of research/contribution
  - Key sources? Recent developments?
  - citations
- Presentation
  - Visuals: Non-text: diagrams, charts, algorithms
  - Systematic/quantitative: tables, numbers
  - Readability, coherence
- Overall

Course Notes

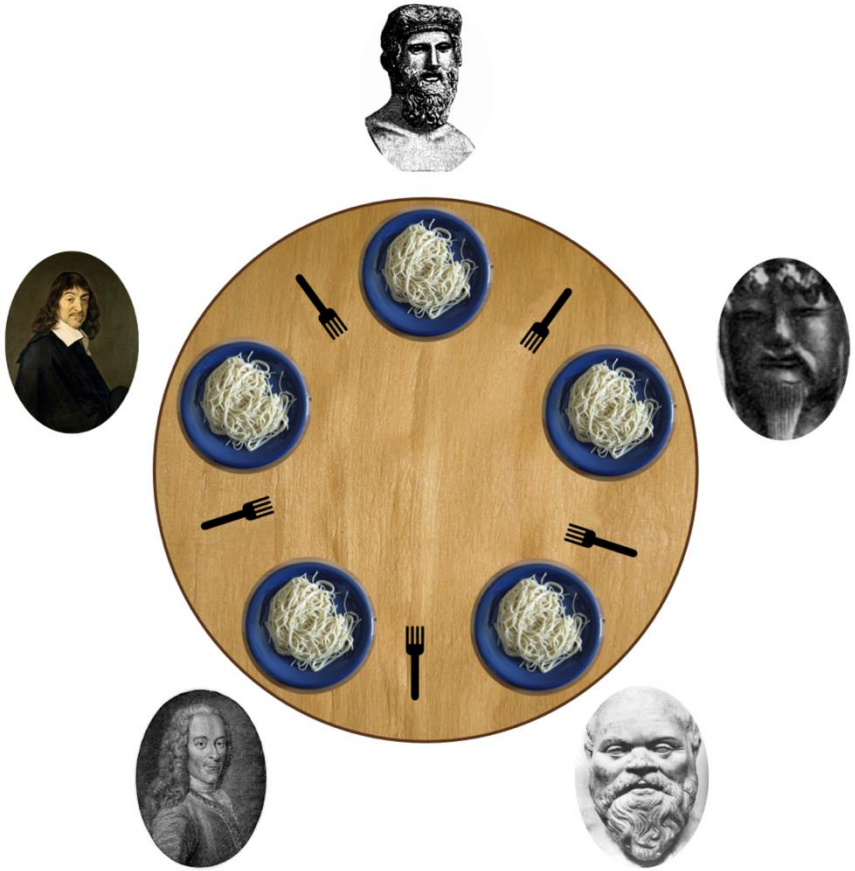
- The Midterm and the Final will use the Respondus Lockdown Browser with camera recording.
  - You must use a laptop with the Respondus Lockdown Browser installed and tested. A trial quiz is available.
- The Sec 001 students will bring the fully charged and tested laptop to the designated room.
  - The 1 hour 15 min Midterm will be on Tues Oct 14 during the regular class time.
- The Sec 801 students will take their Midterm on Oct 15 any time from 12:10 AM (early morning) to 11:50 PM.
- Anyone with a significant conflict should contact me directly.
- Quizzes: available Mondays 12:10 AM (early morning) to 11:50 PM.

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

Dining-Philosophers Problem Algorithm (Cont.)

- 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 actively under execution in the monitor.
- A generic monitor construct is used here. Implementation varies by language.

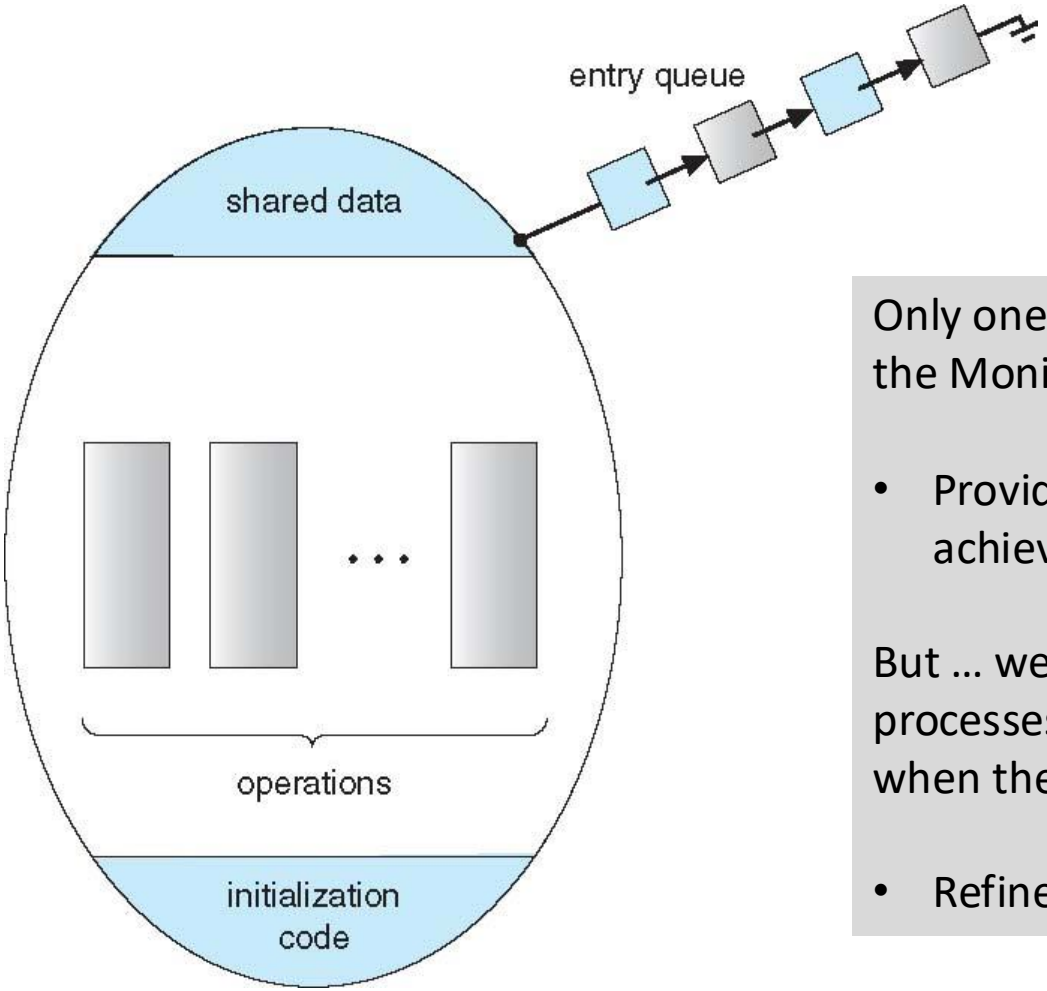
```
monitor monitor-name
{
  // shared variable declarations

  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 …

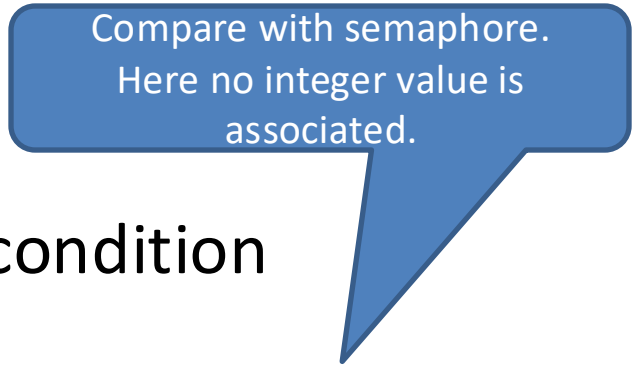
Shows 4 processes waiting in the queue.

Condition Variables

Some actions need some conditions to go ahead.

The **condition** construct

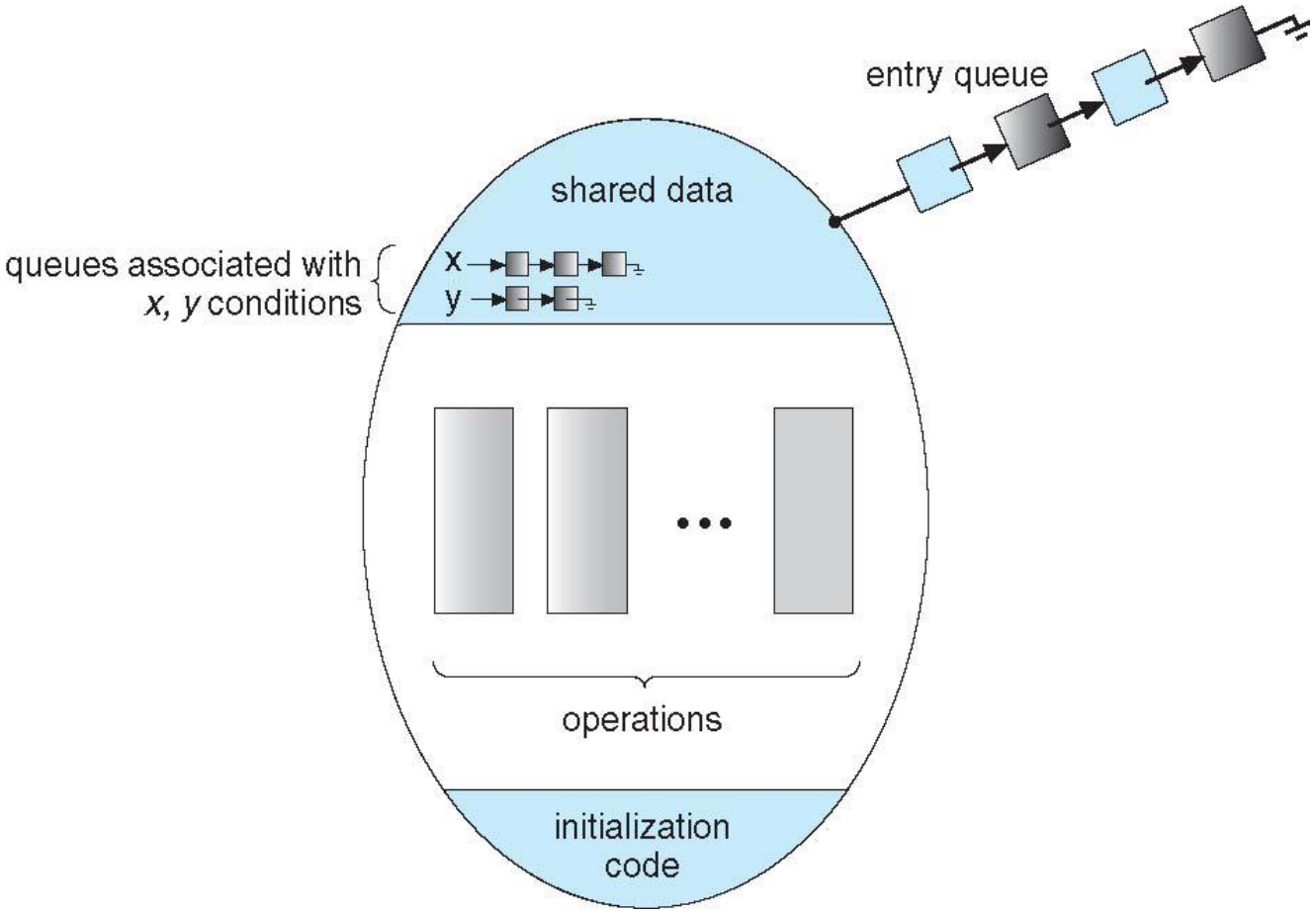
- **condition x, y;**
- Two operations are allowed on a condition variable:
  - **x.wait()** – a process that invokes the operation is suspended until **x.signal()**
  - **x.signal()** – resumes one of processes (if any) that invoked **x.wait()**
    - If no **x.wait()** on the condition variable, then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



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

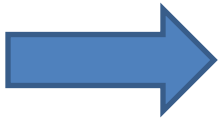
```
enum {THINKING,HUNGRY,EATING} state[5];
```

- `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

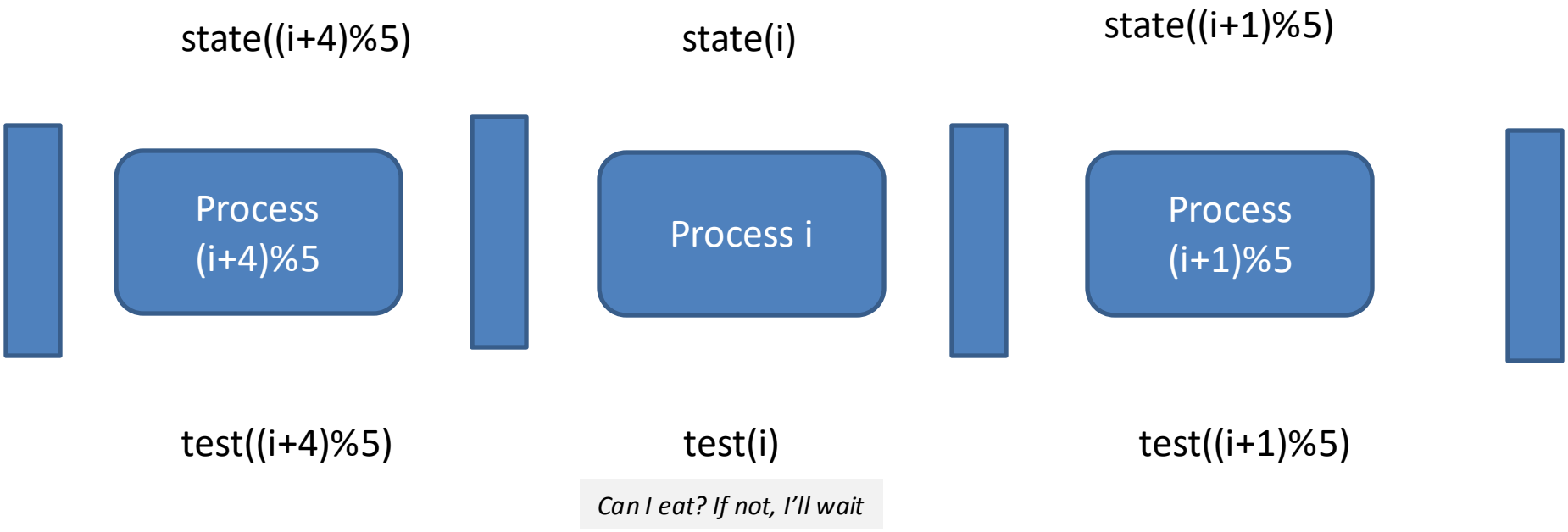
- Before eating, must invoke pickup()
  - May result in suspension of philosopher process
  - After completion of operation, philosopher may eat

```
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

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 () ;
            }
    }

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

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

Signal a process that was suspended while trying to eat

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.signal() is executed, which should be resumed?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.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

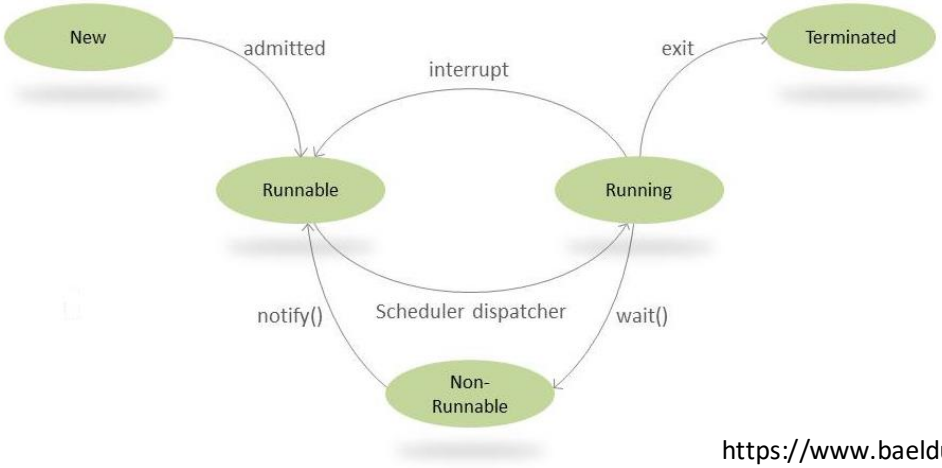
```
public synchronized void increment( ) { c++; }
```

  - 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



https://www.baeldung.com/java-wait-notify

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
        }
    }
```

```
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();
        }
    }
```

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