CS 5323 – OS II

Lecture 5 – Critical Section, Solutions, Mutexes, Semaphores, Monitors, Intro to Scheduling

Logistics

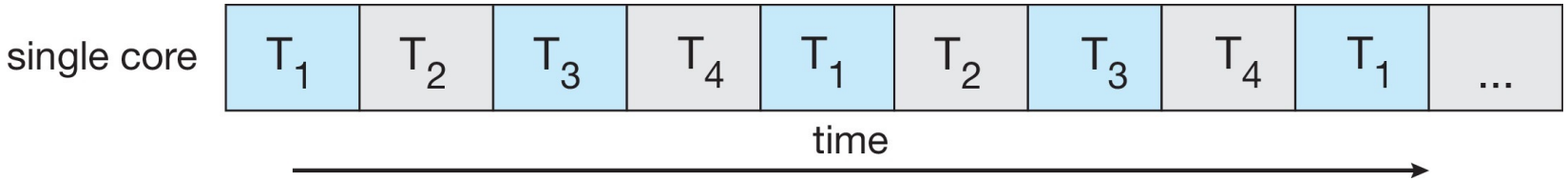
- Quiz 2 Will be posted on Canvas and will be due Monday 01/31/2022 11:59 pm.
- Assignment 1 will be posted. Due 02/04/2021 11:59 pm.

Process Synchronization

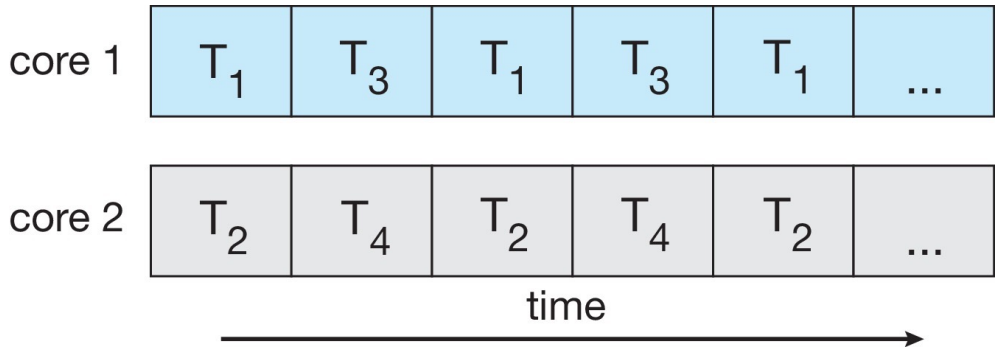
Review

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



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

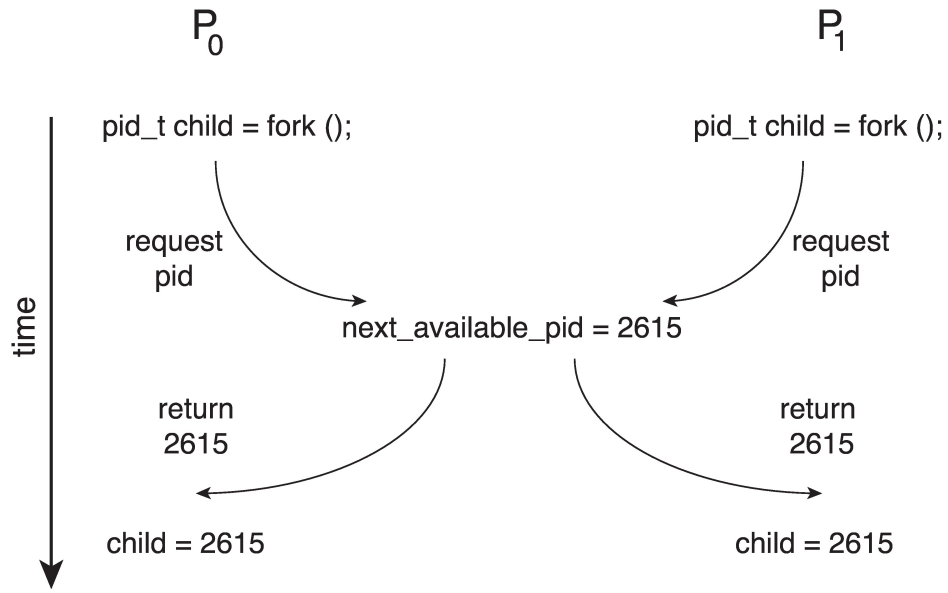
```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter` {register1 = 5}
S1: producer execute `register1 = register1 + 1` {register1 = 6}
S2: consumer execute `register2 = counter` {register2 = 5}
S3: consumer execute `register2 = register2 – 1` {register2 = 4}
S4: producer execute `counter = register1` {counter = 6 }
S5: consumer execute `counter = register2` {counter = 4}

Race Condition – Another Example

- Processes P_0 and P_1 are creating child processs 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 Problem

- Consider system of ***n*** processes {$p_0, p_1, \ldots p_{n-1}$}
- 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);
```

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 nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Peterson's Solution

- Not guaranteed to work on modern architectures! (But good algorithmic description of solving the problem)
- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section. **`flag[i] = true`** implies that process P_i is ready!

Process P_i

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

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Process P_j

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

        /* critical section */

        flag[j] = false;

        /* remainder section */

}
```

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved
     P_i enters CS only if:
     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Process P_i

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

        /* critical section */

        flag[i] = false;

        /* remainder section */

}
```

Process P_j

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

        /* critical section */

        flag[j] = false;

        /* remainder section */

}
```

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 such as compare-and-swap

Solution to Critical-section Problem Using Locks

```
while (true) {
        acquire lock

        critical section

        release lock

        remainder section
}
```

Mutex Lock Definitions

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

- ```
  release() {
      available = true;
  }
  ```

These two functions must be implemented atomically. Both test-and-set and compare-and-swap can be used to implement these functions.

Mutex lock – contd

- Mutex solution requires **busy waiting**
  - This lock therefore called a **spinlock**
- **Spin lock** can be useful
  - No context switch is required when a process is waiting for the lock
    - Context switch is CPU overhead and is avoided in this solution
  - Especially useful when locks are held for short duration.
    - One thread can wait on a processor while another thread executes in different processor.

Semaphore

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

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

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

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

Semaphore Primitives

- 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 applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can only be 0 or 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Consider P_1 and P_2 that require S_1 to happen before S_2

  Create a semaphore "**synch**" initialized to 0

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

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

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);
    }
}
```

Deadlocks and Starvation

- Implementation of semaphore with a waiting queue may result in a situation where 2 or more processes wait **indefinitely** for an event that can be caused by one of the waiting processes.
  - i.e. waiting for the **Signal()** operation
  - This is called **deadlock**
- Process will wait indefinitely within the semaphore will result in **indefinite blocking or starvation.**
  - Occurs when removing the process from the list with the semaphore in LIFO order.
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process

Deadlock Example

P_0	P_1
Wait(S)	Wait(Q)
Wait(Q)	Wait(S)
.	.
.	.
.	.
Signal(S)	Signal(Q)
Signal(Q)	Signal(S)

Initially S = 1, Q = 1

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
  - Mutual Exclusion for access to buffer
- Semaphore **`full`** initialized to the value 0
  - Counts the number of full buffer
- Semaphore **`empty`** initialized to the value n
  - Counts the number of empty buffers
- Classic producer-consumer problem
  - Producer produces full buffers for the consumer
  - Consumer produces (consumes) empty buffer for the producer

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
        ...
        /* produce an item in next_produced */
        ...
    wait(empty);
    wait(mutex);
        ...
        /* add next produced to the buffer */
        ...
    signal(mutex);
    signal(full);
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
    wait(full);
    wait(mutex);
       ...
      /* remove an item from buffer to next_consumed */
       ...
    signal(mutex);
    signal(empty);
       ...
      /* consume the item in next consumed */
       ...
}
```

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

Readers-Writers Problem Variations

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

Data structures needed

- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
    - Common to both reader and writer processes
    - Act as mutual exclusion for writers
    - Used by first or last reader that enters the Critical Section; No other reader uses **`rw_mutex`**
  - Semaphore **`mutex`** initialized to 1
    - Ensures mutual exclusion when **`read_count`** is updated.
  - Integer **`read_count`** initialized to 0
    - Keeps track of how may processes are reading the data

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
     wait(rw_mutex);

        ...
     /* writing is performed */

        ...

    signal(rw_mutex);
}
```

Readers-Writers Problem (Cont.)

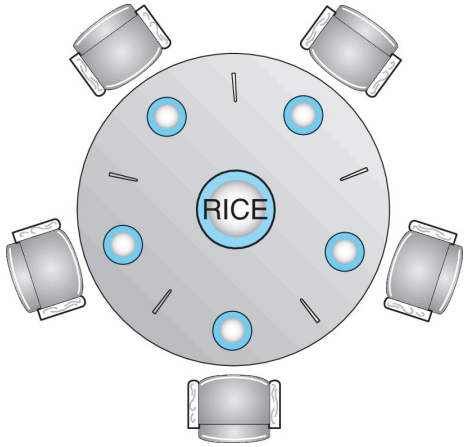
- The structure of a reader process

```
while (true){
        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);
}
```

Points to remember

- When writer is in critical section, and *n* readers are waiting
  - 1 reader is queued on **`rw_mutex`**
  - n-1 are queued on **`mutex`**
- When writer executes signal(**`rw_mutex`**), we may resume execution of either
  - Waiting readers, or
  - Single waiting writer
  - This selection is arbitrary and made by scheduler
- Can be generalized to create a reader-writer lock

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
- Chopsticks are placed between neighboring philosophers
- Cannot pick chopstick in the hand of a neighbor
- Need both chopsticks to eat, then release both when done
- Goes back to thinking when done eating
- Classic synchronization problem!

One possible solution

- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1
- Can grab chopstick by executing wait() on that semaphore
- Release chopstick through signal() operation

Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher *i*:

```
while (true){
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

    /* eat for awhile */

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

    /* think for awhile */

}
```

- **What is the problem with this algorithm?**

- Deadlock!
- What if all philosophers get hungry and grab their left chopstick!
  - All semaphores are now zero.
  - No philosopher can acquire 2 chopsticks to eat and hence block each other indefinitely
- What if all philosophers get hungry and grab their right chopstick?
  - Delayed forever!

Possible Deadlock-free solutions

1. Allow at most 4 philosophers to sit at the table
2. Allow a philosopher to pick up a chopstick if at least 2 chopsticks are available
   - Must pick up in the critical section
3. Asymmetric solution i.e. restrict order of picking up chopsticks.
   - Odd numbered philosopher picks up left chopstick then right
   - Even numbered philosopher does vise versa

- **Deadlock free solutions does not eliminate starvation!**

Monitors

- 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
  - Includes a set of programmer defined operations provided with mutual exclusion
- Only one process may be active within the monitor at a time
  - Helps with solving the critical section problem!

Monitors

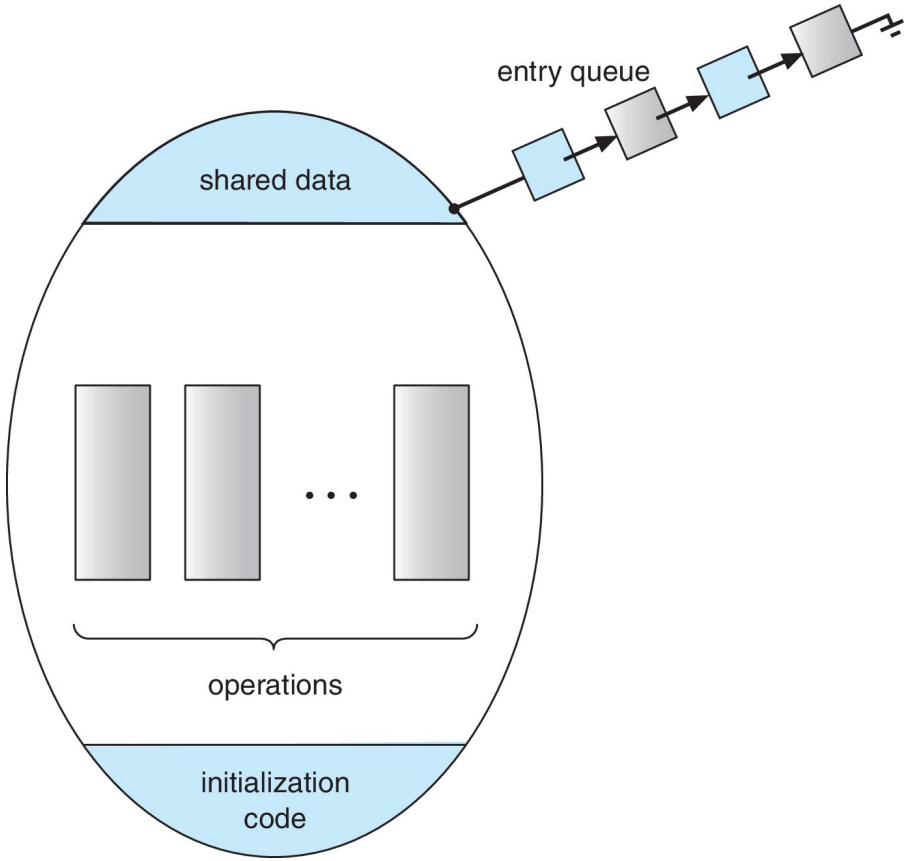
- Pseudocode syntax of a monitor:

```
monitor monitor-name
{
  // shared variable declarations
  function P1 (…) { …. }

  function P2 (…) { …. }

  function Pn (…) {……}

   initialization code (…) { … }
}
```

Monitors

- Representation of a monitor type cannot be used directly by various processes
  - A function defined within a monitor can access only those variables declared locally within the monitor and its formal parameters.
- No need to explicitly code the synchronization constraints.
  - But need to define the mechanisms

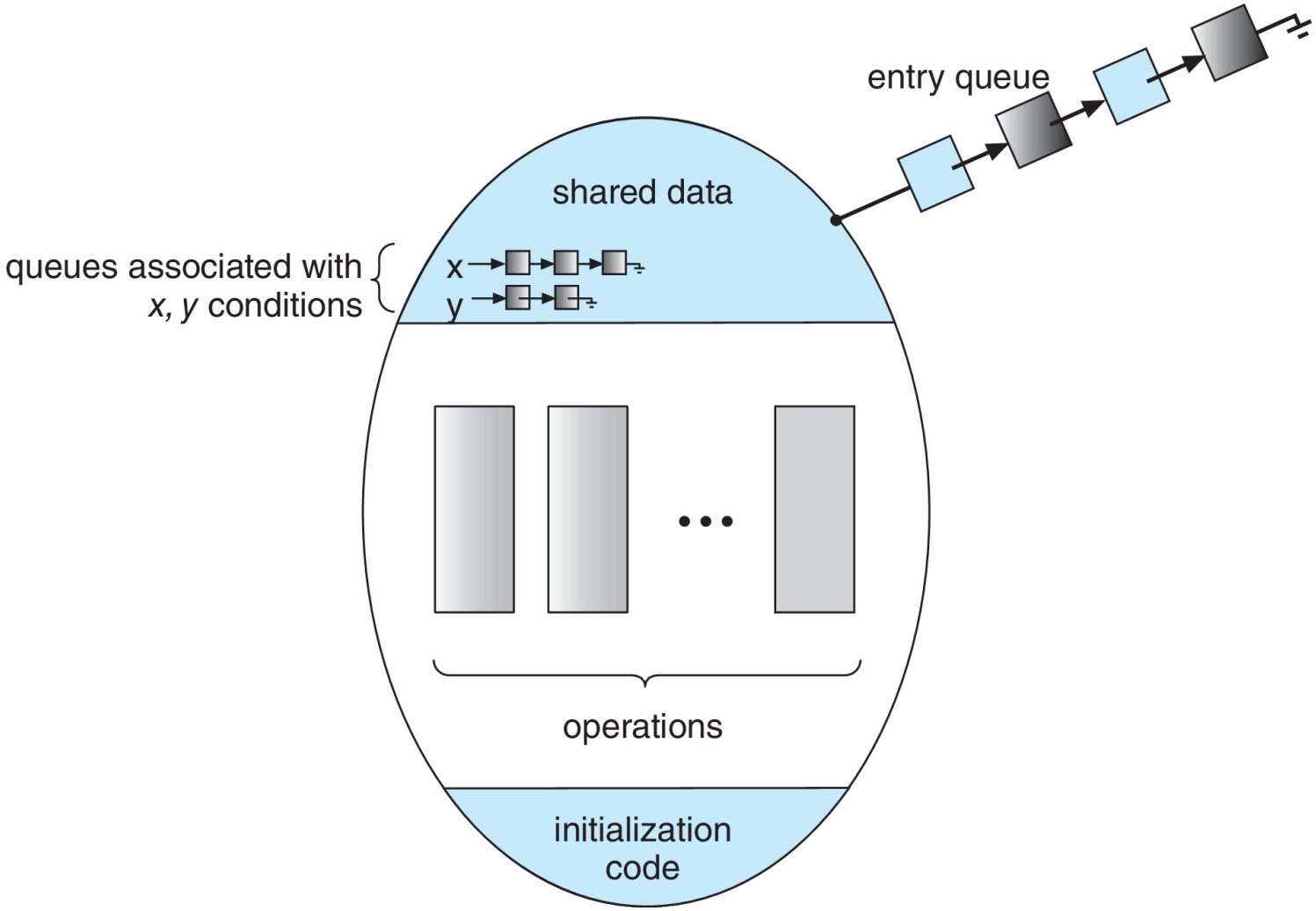
Schematic view of a Monitor



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;
         }
}
```

Solution 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 function ***F*** will be replaced by

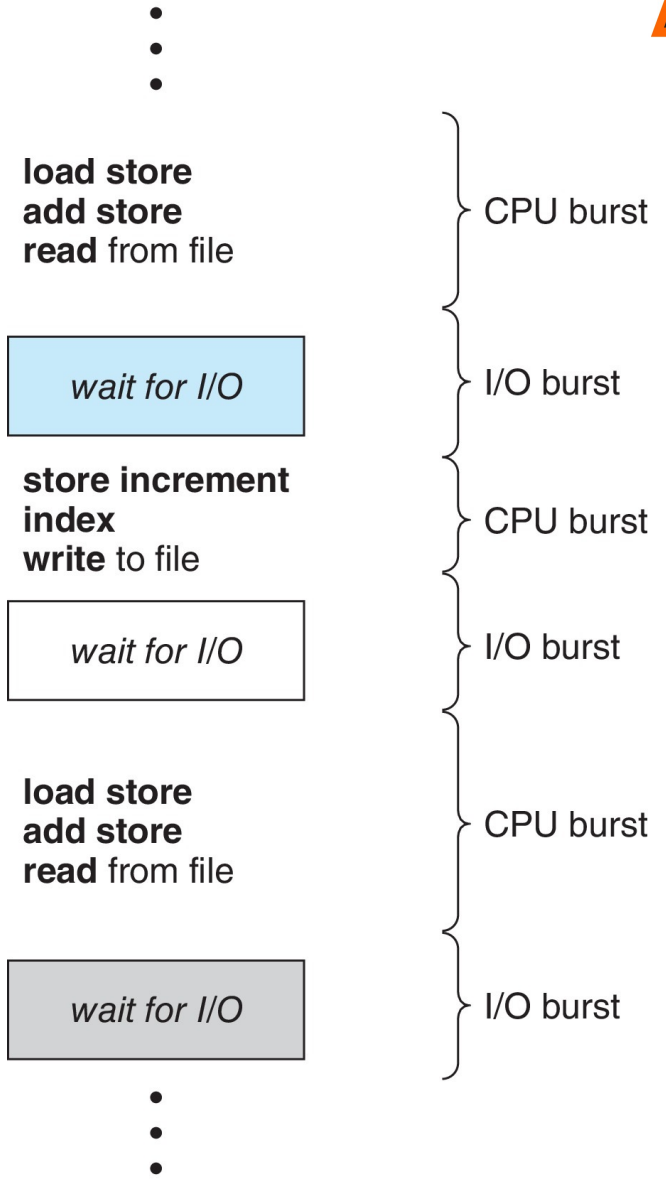
```
wait(mutex);
      …
        body of F;
      …
if (next_count > 0)
 signal(next)
else
 signal(mutex);
```

- Mutual exclusion within a monitor is ensured

Process Scheduling

Basic Concepts

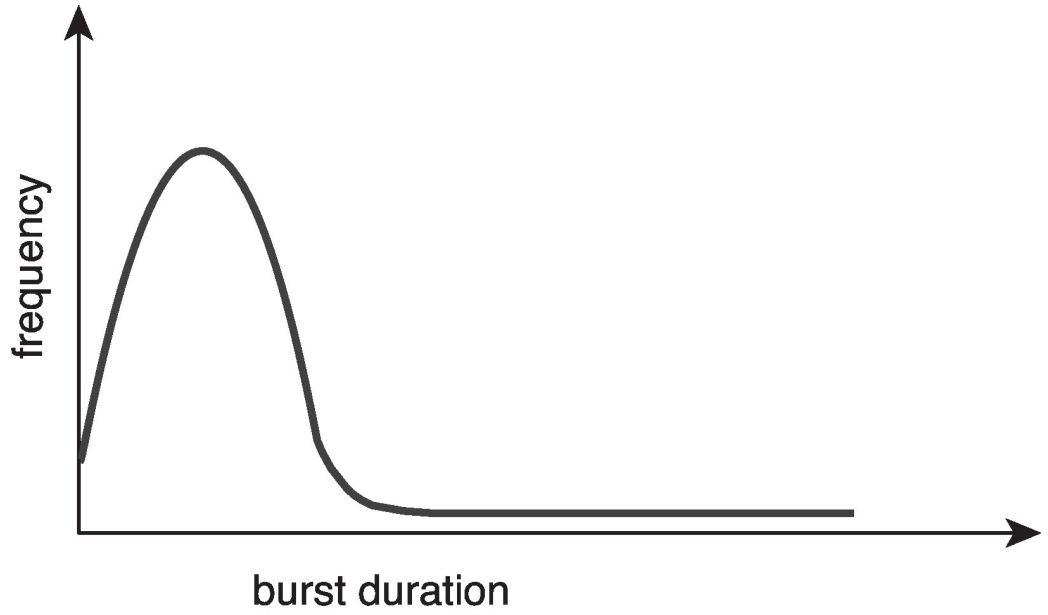
- Maximum CPU utilization obtained with multiprogramming
- CPU–I/O Burst Cycle – Process execution consists of a **cycle** of CPU execution and I/O wait
- **CPU burst** followed by **I/O burst**
- CPU burst distribution is of main concern



Histogram of CPU-burst Times

Large number of short bursts

Small number of longer bursts

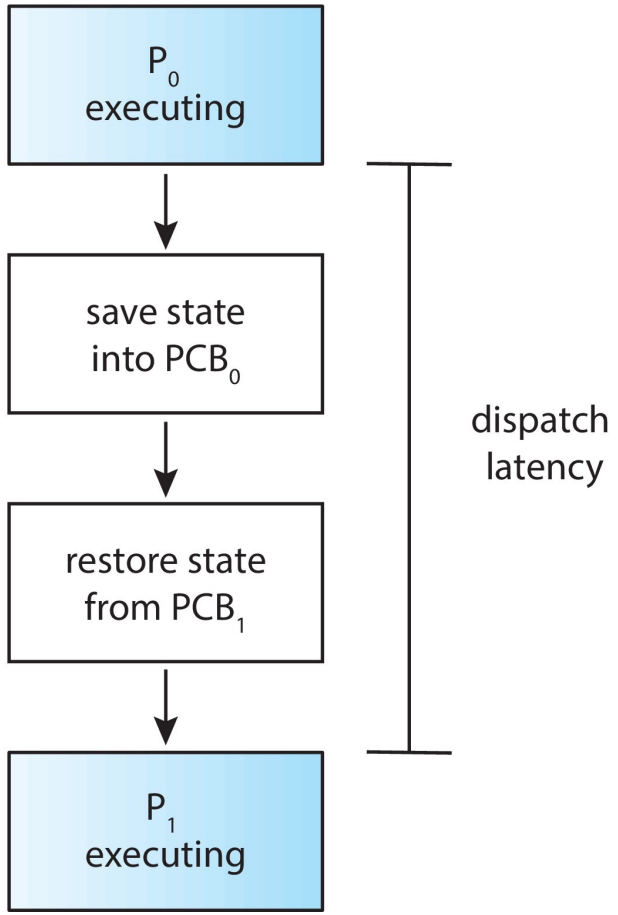


CPU Scheduler

- The **CPU scheduler** selects from among the processes in ready queue, and allocates the a CPU core to one of them
  - Queue may be ordered in various ways
  - Done by short-term scheduler
  - Queue is not necessarily FIFO!!
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates
- Scheduling under 1 and 4 is **non-preemptive**
- All other scheduling is **preemptive**
  - Consider access to shared data
  - Consider preemption while in kernel mode
  - Consider interrupts occurring during crucial OS activities

Dispatcher

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running



Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)