Chapter 6: Synchronization

CS 3423 Operating Systems
Fall 2019
National Tsing Hua University

Overview

- Background
- Critical Section
- Synchronization
- Semaphores
- Classical Problem of Synchronization
- Monitors

Background

- Concurrent access to shared data
  - => possible data inconsistency
- Solution
  - mechanisms to ensure orderly execution of cooperating processes

Consumer-Producer Problem

- use `count` in addition to `in`, `out`

```
// Producer
while (1) {
   nextItem = getItem();
   while(count==BUFSIZE) ;
   buffer[in] = nextItem;
   in = (in+1) % BUFSIZE;
   count++;
}
```

```
// Consumer
while (1) {
   while(count==0) ;
   item =buffer[out];
   out = (out+1) % BUFSIZE;
   count--;
}
```

- problem: `count++` and `count--` implementation in machine language

Incr/decr operation

- depending on the instruction set
  - increment (or decr.) may take multiple instructions
- CISC: one instruction
  - `INC R1` or `INC MEM` ;; VAX/PDP11 -- (this is how C got ++)
  - `INC R0` or `INC A` ;; 8051 increment
- Most other architectures (RISC): multiple instructions!
  - `LA $4, mem` ;; this is actually LUI followed by ORI on MIPS
    `LW $10, ($4)` ;; load data from address is in reg $4 to reg $10
    `ADDI $10, $10, 1` ;; perform the addition in register $10
    `SW $10, ($4)` ;; write result back to same location ($4)

Context Switching (ideal)

Producer	Consumer	Prod. $10	Cons. $10	shared count
`LA $4, mem`		-		5
`LW $10, ($4)`		5		
`ADDI $10, $10, 1`		6	-	
`SW $10, ($4)`				6
	`LA $4, mem`	6	-	6
	`LW $10, ($4)`		6	
	`SUBI $10, $10, 1`		5	
	`SW $10, ($4)`			5

context switch here

Context Switching (race: stale count overwriting consumer)

Producer	Consumer	Prod.'s $10	Cons.'s $10	shared count
`LA $4, mem`		-		5
`LW $10, ($4)`		5	-	
`ADDI $10, $10, 1`		6		
	`LA $4, mem`	6	-	
	`LW $10, ($4)`		5	
	`SUBI $10, $10, 1`		4	
	`SW $10, ($4)`			4
`SW $10, ($4)`		6	4	6

context switch here (#1)

and here (#2)

- before switching back (#2), the count value of 4 is incorrect
  - producer incremented already (count should be 6)
  - the decremented result should be 5, not 4.
- after context switch (#2), the count value of 6 is incorrect,
  - producer's value was 6 but now after consumer, it should be 5.

Race Condition

- Inconsistent computation results from concurrent execution
  - final result depends on which process finishes last
- Solutions for Uniprocessor
  - disable interrupt => disables context switching
  - nonpreemptive (cooperative) CPU scheduling
- General Solution
  - synchronization among concurrent processes
  - mechanisms for "critical section"

Critical Section

The Critical-Section Problem

- Wanted: a protocol for processes to access shared data structure
- Problem statement
  - Processes $P_0..P_{n-1}$ competing for accessing shared data
  - Critical section:
    - Code segment that accesses shared data w/ mutual exclusion
  - Mutual exclusion property:
    - when one process is executing in its critical section, no other process is allowed to execute its critical section

Critical-Section Problem

- General code structure of typical process
  - at most one process can be in critical section at a time
  - **do** {
    section entry
    critical section
    section exit
    remainder section
  } **while** (1);

3 Requirements of Critical Section

1. Mutual Exclusion (at most one process in CS at a time)
   - one process in CS => no other processes in CS
2. Progress
   - if no process in CS, and some processes wish to enter CS => they must not be blocked indefinitely.
3. Bounded Waiting (bound on wait time, not just finite)
   - A bound must exist on the number of times that other processes are allowed to enter their CS after a process has made a request to enter its CS

- Q: How to design entry and exit sections to satisfy the above requirements?

Critical Section Solution and Synchronization Tools

- Software solution
- Synchronization hardware
- Semaphore
- Monitor

Software Solution

- Nonpreemptive
  - easy - no race condition!
    (assumption: no process (or thread) yields in the middle of a critical section)
    - but no CPU protection (a process can hog CPU)
- Preemptive
  - Peterson's for two processes

Peterson's for Two Processes

- Shared variables between Processes P_0 and P_1
  - **int** turn = 0; *// 0 or 1 to indicate P_0 or P_1's turn*
  - **boolean** flag[2] = { FALSE, FALSE }; *// flag[i] true if P_i ready to enter CS*
- if (flag[j] == FALSE or turn == i), then P_i can enter its critical section
-

```
flag[0] = TRUE;
turn = 1;
while (flag[1] && turn == 1) ;
critical section
remainder
flag[0] = FALSE;
```

P_0

```
flag[1] = TRUE;
turn = 0;
while (flag[0] && turn == 0);
critical section
remainder
flag[1] = FALSE;
```

P_1

assumption: no hardware support, can context switch anywhere!

Peterson's Solution for Two Processes

- Mutual exclusion [Yes]
  - when flag[0]==flag[1]==TRUE, rely on *turn* (which can be either 0 or 1) => mutually exclusive
- Bounded wait [Yes]
  - after flag[i] executes, always sets its own flag[i]=FALSE, and *turn* is already == j, so P_j can run
- Progress [Yes]
  - while loop's condition for *i* will always have an AND condition broken by *j*, if not already

Issues with Peterson's Solution

- Software only
  - does not require hardware support
- Not guaranteed to work on modern CPU
  - Compiler or CPU may reorder instructions if no dependencies
  - a CPU may have multiple hardware threads and thus see values not in code order
- Solution: use proper synchronization tools

Hardware Support for Synchronization

Hardware Support for Synchronization

- Memory barriers
  - an instruction to explicitly ensure memory is loaded in code order
- Hardware instructions
  - atomic instructions
- Atomic variables
  - variables whose updates are atomic

Hardware support for locking primitives

- Lock must be atomic
- Uniprocessor
  - simply disable interrupt in critical section
- Multiprocessor
  - disabling interrupt either infeasible or too inefficient
- Modern machines
  - atomic test-and-set
  - swap content of two words in memory

Atomic Test-and-Set

Executes atomically

- ```
  bool TestAndSet(bool &lock) {
    bool value = lock; // save old value
    lock = TRUE; // lock anyway, whether we got the lock
    return value; // we got lock if it was unlocked,
  }       // but if someone else got lock, return TRUE
  ```
  - `lock` is a shared variable indicating locking state.
    - At most one process at a time should get the lock
- Protocol
  - if currently unlocked, lock it as part of testing! (and you own the lock)
  - To unlock, the lock owner simply sets lock = FALSE;
  ```

Example

- ```
  do {
    while (TestAndSet(lock)) ; // get lock
    critical section;
    lock = FALSE; // unlock
  } while (1);
  ```

- All processes do something like this
  - Locking by `while (TestAndSet(lock)) ;`
  - unlocking by simply `lock = FALSE;`
  - at most one process at a time will hold the lock
  ```

Atomic CompareAndSwap()

- Similar to, more general than atomic test-and-set
  - `AtomicSwap(&lock, value)`
  - value could encode the owner of lock
- Code (executed atomically)

```
• int CompareAndSwap(int *val, int expected,
                     int newVal) {
      int temp = *val;
      if (*val == expected)
          *val = newVal;
      return temp;
  }
```

Atomic Variables

- primitive types and associated functions
  - e.g., increment, assign
  - often implemented w/ atomic compare-and-swap
- not the same as critical section
  - example: single producer, multi-consumer
    - both consumers wait for count > 0
    - if count became 1, both consumers could unblock
  - Applicability limited to single update of shared data, rather than being guards on other shared data structure

Mutex Locks

- Motivation:
  - Previous solutions are complicated, inaccessible to application programmers
  - Solution provided by OS designers for critical section
- Simplest is mutex lock
  - `acquire()` a lock before entering a critical section,
    `release()` the lock when exiting the critical section => both atomic
  - Boolean variable indicating whether lock is available
  - Usually implemented via hardware atomic instructions
- Problem
  - Requires busy waiting, called a *spinlock*

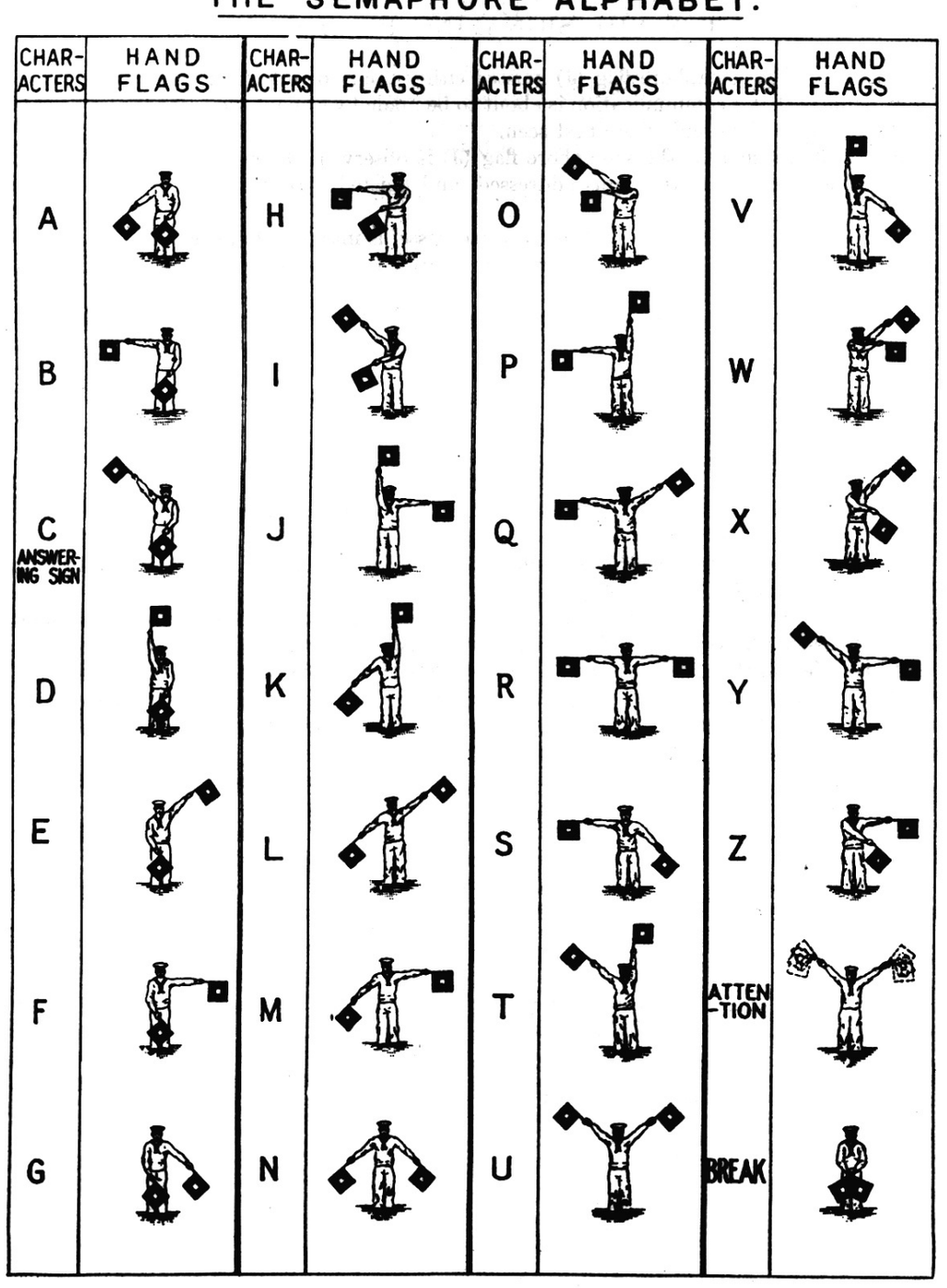
Example in Python

- import threading
- L = threading.Lock() # constructor call.
- L.acquire()
    - blocks until the lock L is acquired
    - Can also do polling
        - L.acquire(blocking=False) => try acquiring lock L, return successful lock or not, but does not block in either case.
- L.release()
    - any thread can unlock L, not just the original thread that locks it!

Semaphores

Semaphore

- Originally, a way to encode symbols using two flags
  - Real-time long-distance optical communication before the days of telecommunication
- But.. that's not what it means in CS



credit: http://www.cs.dartmouth.edu/~rockmore/semaphore.html

Semaphore - history

- Proposed by Dijkstra in 1962-1963
  - for THE multiprogramming system
  - purpose: very simple mechanism for supporting synchronization
- Two primitives (system calls) on an integer variable
  - `P()` => to probe ("proberen" in Dutch), test
  - `V()` => to increment ("verhogen" in Dutch)
- Modern OS terms
  - `P()` => `wait()`, `V()` => `signal()`

Semaphore - in computer science

- A variable `S` that contains an `int` for controlling access to shared resources
  - serves as a "count" # instances available or deficit
  - mutex is special case of semaphore where S is binary instead of int
- 2 primitives: `wait(S)` and `signal(S)`
  - must be paired up, either within the same thread or across different threads

Semaphore implementation: Busy-wait version

- busy-wait implementation
  - assume S is int, also assume preemption to switch context
  - wait(S): // originally called P()
    ```
    while (S <= 0) ; // busy wait: no rsrc avail
    S--;  // got unblocked, grab it by decr count
    ```
  - signal(S):// originally called V()
    ```
    S++;   // give back 1 instance of rsrc to pool
    ```
- issues with busy wait version
  - inefficient, for same reason as polling
  - solution: use wait queue to allow another thread/process to run

Semaphore Usage

- Counting Semaphore (initialize S = N > 1)
  - *N* instances of resource, S initialized to *N*
- Binary Semaphore (initialize S = 1)
  - Number is binary (0 or 1) => mutex
- Barrier (initialize S = 0)
  - for enforcing precedence (event A must precede event B)
- Implementation
  - Busy-wait: simple but inefficient use of CPU cycles
  - Non-busy-wait: use a waiting queue

Real-life example for semaphore

- *N* parking spots
  - Up to *N* cars can park without waiting
  - if > *N* cars, must wait for some car to leave
    - negative value = #cars waiting
      -5 means 5 cars waiting. 0 means full & no waiting
- Question: policy for unblocking
  - wait in line (FIFO)?
  - whoever grabs it first (FCFS)?

Parking lot illustration

	time	0	1	2	3	4	5	6	7	8	9	10	11	12	13
car	0		arrv	park	p	p	dep								
	1			arrv	p	p	p	p	p	p	p	p	dep		
	2			arrv	p	p	p	p	dep						
	3			arrv	p	p	p	p	p	p	dep				
	4				arrv	p	p	p	p	p	p	p	p	p	p
	5				arrv	wait	w	park	p	p	p	p	p	p	p
	6				arrv	wait	w	w	w	park	p	p	p	dep	
	7					arrv	wait	w	w	w	w	park	p	p	p
	8														arrv
value		5	4	1	-2	-3	-2	-2	-1	-1	0	0	1	2	1
spot	0	-	car0	0	0	0	-	car5	5	5	5	5	5	5	5
	1	-	-	car1	1	1	1	1	1	1	1	1	-	-	car8
	2	-	-	car2	2	2	2	2	-	car6	6	6	6	-	-
	3	-	-	car3	3	3	3	3	3	3	-	car7	7	7	7
	4	-	-	-	car4	4	4	4	4	4	4	4	4	4	4

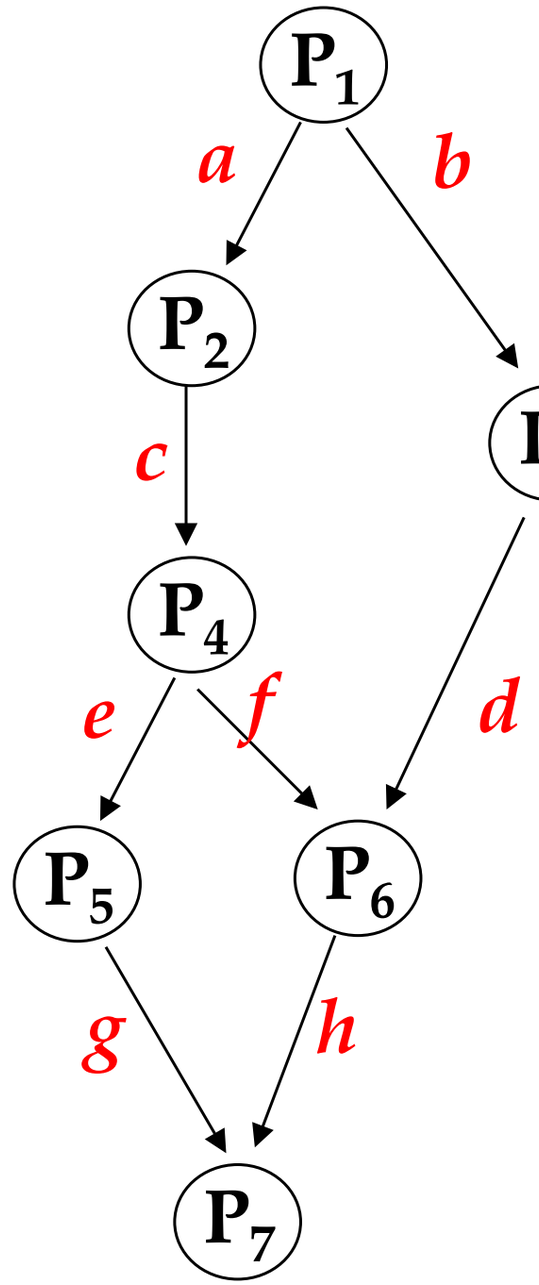
Barrier: forcing ordering between concurrent processes

- Example
  - P_1 executes S_1, and P_2 executes S_2
  - Both P_1 and P_2 require S_1 to precede S_2
- Semaphore implementation

```
semaphore sync = 0;
// in P1:                    // in P2:
   S1;                          wait(sync);
   signal(sync);                S2;
```

Example: more general sync in a DAG

(Initially, all semaphores are 0)
begin
  P_1: S_1; signal(a); signal(b);
  P_2: wait(a); S_2; signal(c);
  P_3: wait(b); S_3; signal(d);
  P_4: wait(c); S_4; signal(e); signal(f);
  P_5: wait(e); S_5; signal(g);
  P_6: wait(f); wait(d); S_6; signal(h);
  P_7: wait(g); wait(h); S_7;
end



i.e., each edge x corresponds to signal(x) to wait(x)

n-process Critical Section

- shared data:
  - `semaphore mutex;`
- Process P_i:
  - ```
    do {
        wait(mutex);  // e.g., pthread_mutex_lock()
        critical section
        signal(mutex); // e.g. pthread_mutex_unlock()
        remainder section
    } while (1);
    ```
- Progress? Yes.
- Bounded wait? Depends on implementation of `wait()`
    ```

Semaphores without busy wait

C struct looks like this:
container of int and linked list

```
typedef struct {
  int value;
  struct PCB *waitlist;
} semaphore;
```

pseudo code in python syntax

```
def wait(S):
   S.value -= 1  // deduct #avail rsrc instances
   if (S.value < 0):
      // not enough rsrc to proceed, we wait in line
      S.waitlist.append(thisProcess)
      sleep; // switch context to someone else
   // else no blocking!
```

```
def signal(S):
   S.value += 1 // give back #avail rsrc instances
   if (S.value <= 0):
      P = S.waitlist.dequeue() // could be priority
      wakeup(P) // put P back in ready queue
   // else there is nobody to unblock
```

Semaphores without busy wait

- Process linked list
  - may use any queuing policy: FIFO, LIFO, etc, or some kind of priority queue
- Assume available system calls
  - adding process itself to waitlist and sleep (like yield) => won't get waken till another signal() calls wakeup on us
  - wakeup(*P*): wake up a process *P* by moving it from waitlist back into ready queue
  - Q: how to select *P* to wake up?
    A: depends on policy

Busy Wait or Not?

- Depends on length of critical section
- Short (a few instructions) => busy wait
  - easier, rarely occupied, finishes quickly
  - lower overhead esp. for multiprocessor
- Long (minutes, hours) => non-busy-wait
  - busy waiting too inefficient!
  - better to go on wait queue, worth multiprocessor overhead

Issue with Semaphores

- too low level
  - wait() and signal() are two separate calls
    - wait first, then signal; barrier may do in either order
  - must be matched up or else in trouble
    - programmer must be sure!
- Nobody reinforces pairing or ordering...
  - what if code calls signal() then wait() or without wait() ? e.g., on mutex => breaks critical section!
  - what if code calls wait() without signal()? => hogs resources!

Monitors

Monitor

- A "synchronized object"
  - think: object-oriented programming
- Instance of abstract data type (ADT="class")
  - **encapsulates** shared data
  - provides **methods** to access resources
  - methods invocations by multiple processes or threads are serialized
- Languages that support monitor concepts
  - concurrent Pascal, C#, Java, Python, ..
- Methods synchronize by **condition variables**

Monitor declaration with condition variables

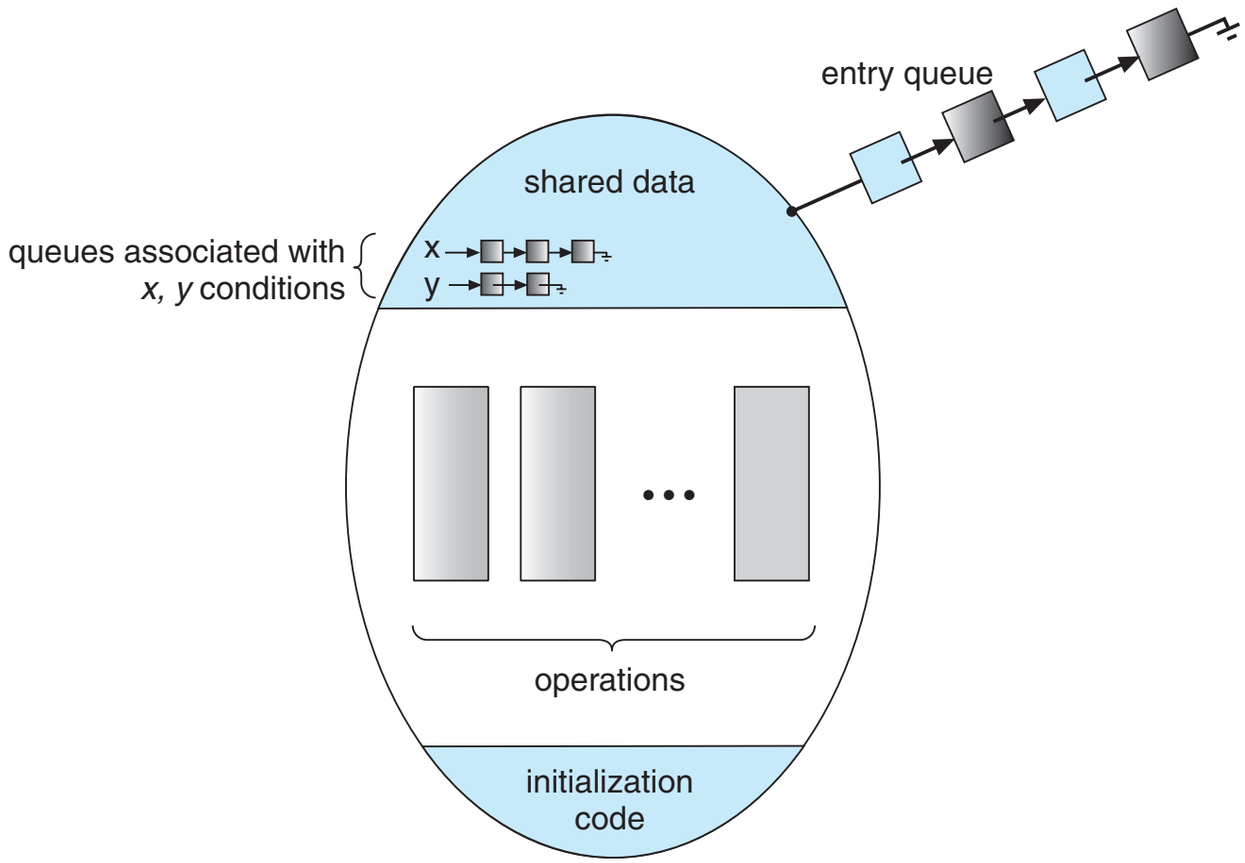
```
monitor MonitorName:
    # shared variable declaration

    def method1(...):
        # synchronized access

    def method2(...):
        # synchronized access

    def __init__(...):
        # initialize
```

- condition variables
  - x.wait(), x.signal()



Monitor Implementation (1/2) Using Semaphores

- Variables
  - semaphore mutex, /* (initially = 1) */ next; /* (initially = 0) */
    int next_count = 0;
- Each procedure F will be replaced by
  - wait(mutex);

```
body of F;
if (next_count > 0) {
    signal(next); // barrier, tell next in dependency order to wake
} else {
    signal(mutex); // no more next count, now release mutex
}
```

- Mutual exclusion within a monitor is ensured

Monitor Implementation (2/2) – Condition Variables

- For each **condition variable** x, we have:
  - semaphore x_sem; // (initially = 0), also barrier
    int x_count = 0;

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

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

Resuming Processes within a Monitor

- If several processes queued on condition x, and x.signal() executed, which should be resumed?
- FCFS frequently not adequate
- Priority scheme may be more suitable
  - conditional-wait construct in the form of `x.wait(c)`, where `c` is priority number
  - Process with lowest number (highest priority) is scheduled next

Issue with Monitors

- access permission
  - assumed, but how to be sure?
- a process might ...
  - never release a resource
  - release a resource it never requested - by mistake
  - request same resource multiple times (without releasing first)