Chapter 6: Synchronization

Communications and Synchronization

- Objectives
  - Recalling communications between processes
  - Introduce synchronization and the critical section problem, whose solutions can be used to ensure the consistency of shared data
  - Present both software and hardware solutions of the critical-section problem
- The Critical Section Problem
  - Peterson’s Solution
  - Synchronization Hardware
  - Semaphores
- Classical Problems of Synchronization

Inter-process Communications (Recall)

- Processes within a system may be
  - Independent: process cannot affect or be affected by the execution of another process
  - Cooperating: process can affect or be affected by the execution of another process
- Reasons for cooperating processes:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience
- Cooperating processes need Inter-process Communication (IPC) mechanisms (Chapter 3)

Aside: Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?
- How are parallel/concurrent processes being handled?

General Concurrent Process Structure

```
while (TRUE) {
    entry section
            critical section    // prone to
                          // race conditions
    exit section
            remainder section
};
```

A race condition is an undesirable situation that occurs, when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

Race Conditions (1)

- Two simple execution examples

- `count++` could be executed on the CPU as

```
register1 = count
register1 = register1 + 1
count = register1
```

- `count--` could be executed on the CPU as

```
register2 = count
register2 = register2 - 1
count = register2
```

Race Conditions (2)

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

    S0: producer executes `register1 = count` {register1 = 5}
    S1: producer executes `register1 = register1 + 1` {register1 = 6}
    S2: consumer executes `register2 = count` {register2 = 5}
    S3: consumer executes `register2 = register2 - 1` {register2 = 4}
    S4: producer executes `count = register1` {count = 6 }
    S5: consumer executes `count = register2` {count = 4}

- Major problem!

Solution to Critical Section (CS) 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, 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.

Under the assumption that each process executes at a non-zero speed, but w/a any assumptions concerning relative speeds of *N* processes.

Peterson’s Solution

- The two processes software solution
  - Assume that the LOAD and STORE instructions are atomic, they cannot be interrupted (not true in many of today’s architectures!).
- These two processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`
- Variable `turn` indicates whose turn it is to enter the critical section.
- 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!

Algorithm for Process P_i

```
while (TRUE) {
  flag[i] = TRUE;
  turn = j;
  while (flag[j] && turn == j);
                                 // do nothing
  critical section

  flag[i] = FALSE;

  remainder section
};
```

&&: Logical AND operator

Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uni-processors could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multi-processor systems
    - Operating systems using this are not broadly scalable
- Modern machines provide special atomic hardware instructions
    - Atomic = non-interruptable
  - Either: test memory word and set value
  - Or: swap contents of two memory words

Solution to Critical Section Problem Using Locks

```
while (TRUE) {

  acquire lock
    critical section
  release lock

  remainder section
};
```

TestAndSet Instruction

- TestAndSet function calls, if executed simultaneously on multiple CPUs, will be sequentialized in some arbitrary order, but are atomically executed

```
boolean TestAndSet (boolean *locked)  //
  modifiable parameter
{
  boolean answer = *locked;
  *locked = TRUE;
  return answer;
}
```

`*var` in expressions refers to the value stored in the address of `var`

- Return value indicates the current lock status
  - If false a new lock on it was acquired

Solution using TestAndSet

- If a computer architecture supports the `TestAndSet()` instruction, define a shared Boolean variable `locked`
  - Initialized to `FALSE`
  - Solution is called busy waiting

```
while (TRUE) {
  while (TestAndSet(&locked));
  // do nothing, busy wait
  //  critical section {…}
  locked = FALSE;

  //  remainder section
};
```

Swap Instruction

- Swap (in contrast to `TestAndSet()`) instruction operates on two parameters and is executed atomically
  - Arbitrarily sequentialized, data structure as follows:

```
void Swap (boolean *lock, boolean *key)
                  // modifiable parameters
{
 boolean temp = *lock;
 *lock = *key;
 *key = temp}
```

- The modified return argument key indicates the current lock status, mutual exclusion reached, not bounded wait

Solution using Swap Instruction

- Shared Boolean variable
  - `Lock` initialized to `FALSE`
  - Each process has a local Boolean variable `key`
- Solution for process

```
while (TRUE) {
  key = TRUE;
  while (key == TRUE)
    Swap(&lock, &key);  // busy wait
    //  critical section {…}
   lock = FALSE;
   //   remainder section
};
```

Bounded-waiting Mutual Exclusion with `TestAndSet()`

- Bounded-waiting more difficult than mutual exclusion
  - Needs extra data structures to identify waiting processes, Boolean array `waiting[n]`

```
while (TRUE) {
  waiting[i] = TRUE;        // process i waiting for access to CS
  key = TRUE;
  while (waiting[i] && key) // wait for lock from other process
    key = TestAndSet(&lock); // busy wait
  waiting[i] = FALSE;
  // critical section
  j = (i+1) % n;
  while ((j != i) && !waiting[j])  // check for waiting processes
    j = (j+1) % n;
  if (j == i)
    lock = FALSE;           // no one waiting, release lock
  else
    waiting[j] = FALSE;     // transfer lock
  // remainder section
};
```

Semaphores

- Synchronization tool that does not require busy waiting
  - Semaphore S as an integer variable
- Two standard operations modify S only
  - `wait()` and `signal()`
    - Originally called `P()` and `V()` to lock or release semaphore
  - Less complicated
  - Can only be accessed via two indivisible (atomic) operations

```
wait(S) {
  while S <= 0 ;  // no-op
  S--;
}
signal(S) {
  S++;
}
```

Semaphore as a General Synchronization Tool

- Binary semaphore (also known as mutex locks)
  - integer value can only be 0 or 1
  - Simpler to implement (simple mutual exclusion)
- Counting semaphore
  - integer value can range over an unrestricted domain
  - Initial value indicates level of parallelism of passing this sem.
    - Counting semaphore S can be implemented using a binary semaphore
  - Control access to finite limited resources

```
semaphore mutex;    //  initialized to 1
  while(TRUE) {
    wait (mutex);
         // critical section
    signal (mutex);
         // remainder section
```

Semaphore Implementation

- Must guarantee that no two processes can execute `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the CS problem where the wait and signal code are placed in the CS!
  - Software impl. now has busy waiting in the critical section
    - But implementation code is short
    - Little busy waiting, if critical section is rarely occupied
- Note that applications may spend lots of time in critical sections and, therefore, this is not a good solution
  - Modified implementation, where processes are blocked, when waiting for a semaphore

Semaphore Implementation without Busy Waiting

- With each semaphore a waiting queue is associated. Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list

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

- Two operations:
  - block: to place the process invoking the operation on the appropriate waiting queue
  - wakeup: to remove one of the processes in the waiting queue and place it in the ready queue
    - Block/wakeup provided by operating system as basic system calls

Semaphore Implementation

- Implementation of wait

“-->” Structure dereference: member *b* of object pointed to by *a*)

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

- Implementation of signal:

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

Deadlocks

- Deadlock (cf. Chapter 7)
  - 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

```
    P0                          P1
  wait (S);                   wait (Q);
  wait (Q);                   wait (S);
.                                      .
.                                      .
.                                      .
  signal (S);                 signal (Q);
  signal (Q);                 signal (S);
```

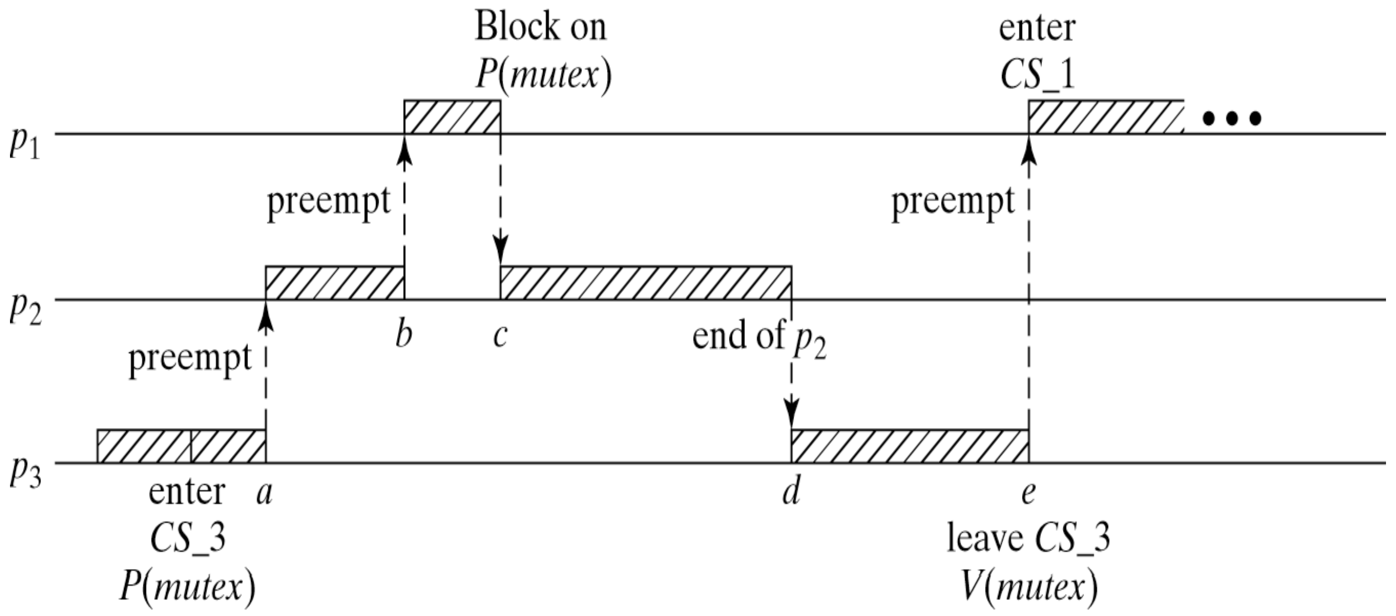
Starvation and Priority Inversion

- Starvation
  - Indefinite blocking

    A process may never be removed from the semaphore queue in which it is suspended

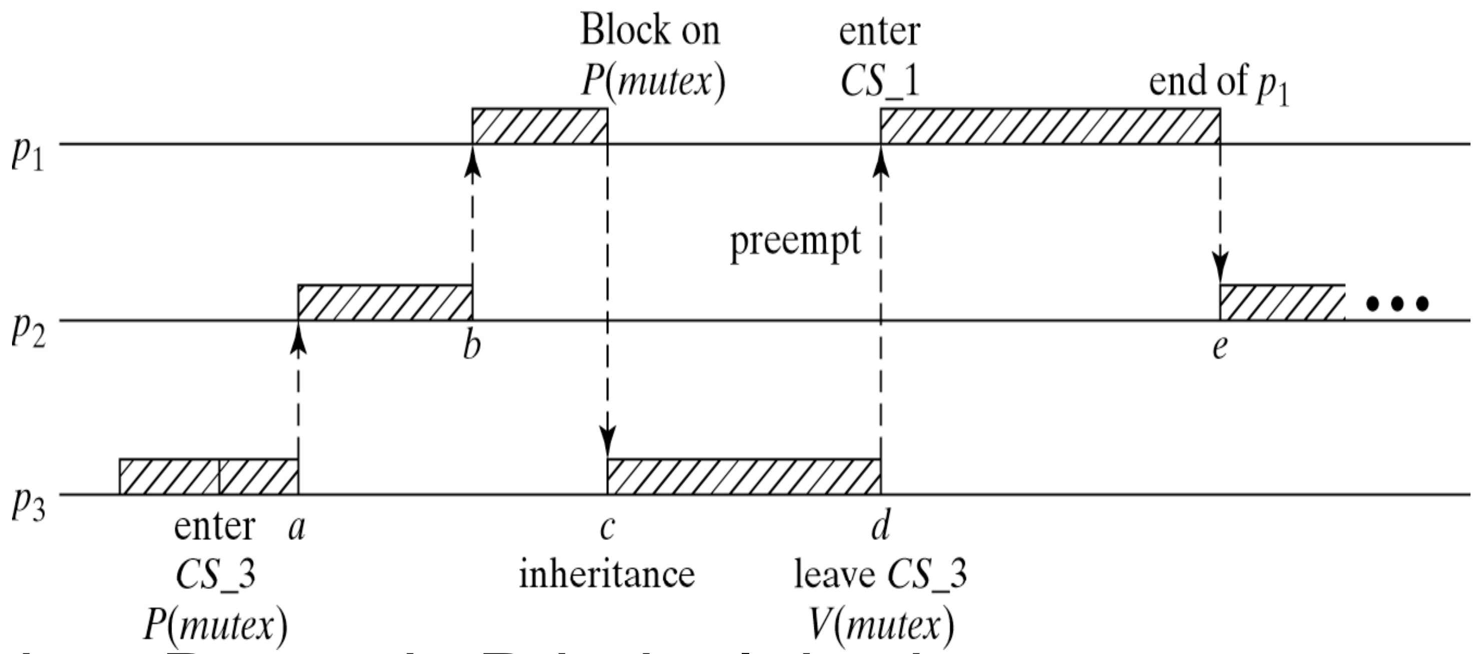
- Priority Inversion
  - Scheduling problem when lower-priority process holds a lock needed by higher-priority process
    - See priority inheritance protocol

Priority Inversion



- Assume priority order $p_1 > p_2 > p_3$
  - (unrelated) p_2 may delay p_1 indefinitely
- Naïve “solution”: always run CS at priority of highest process that shares the CS
  - Problem: p_1 cannot interrupt lower-priority process inside CS — a different form of priority inversion

Priority Inheritance



- Solution: Dynamic Priority Inheritance
  - Inherit priority from higher blocked process
    - ***p*₃** is in its CS
    - ***p*₁** attempts to enter its CS
    - ***p*₃** inherits ***p*₁**'s (higher) priority for the duration of CS

Classical Problems of Synchronization

- ❑ Bounded Buffer Problem (see Chapter 3)
- ❑ Readers and Writers Problem
- ❑ Dining Philosophers Problem

Readers and Writers Problem

- Data set is shared among concurrent processes
  - Readers only read data set; they do **not** perform any updates
  - Writers can both read and write
- Problem:
  - Allow multiple readers to read at the same time, but only one single writer can access the shared data at that same time
- Shared Data
  - Data set
  - Semaphore `mutex` initialized to 1
    - Used to exclude others from modifying the buffer state concurrently
  - Semaphore `wrt` initialized to 1
    - Used to serialize exclusive write operations
  - Integer `readcount` initialized to 0
    - Used to indicate concurrent readers

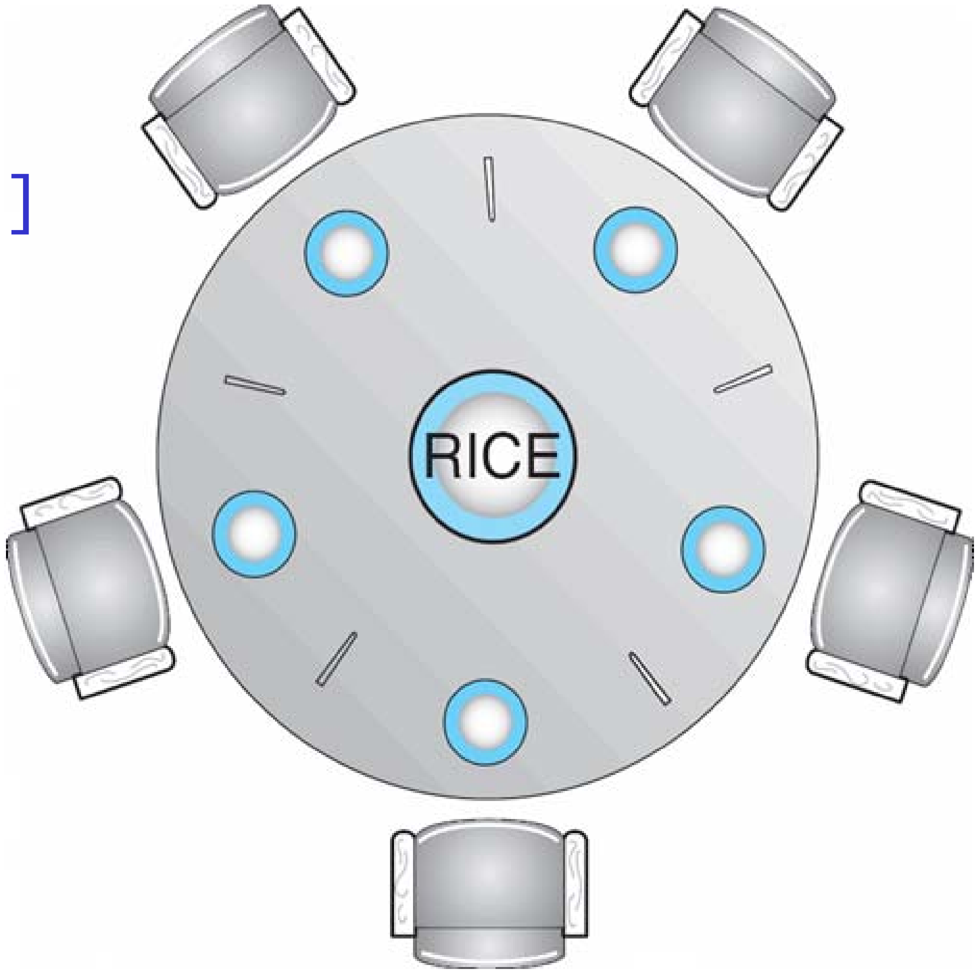
Readers and Writers Processes

```
while (TRUE) {
   wait(mutex);              // operate safely on read count
   readcount++;
   if (readcount == 1)
     wait(wrt);              // prevent any writers at this time
   signal(mutex)
   // reading is performed
   wait(mutex);
   readcount--;
   if (readcount == 0)
     signal(wrt);            // make sure writers can update data
   signal(mutex);
 };
```

```
while (TRUE) {
   wait(wrt);
   //    writing is performed
   signal(wrt);
 };
```

Dining Philosophers Problem (1)

- ❑ Shared data
  - Bowl of rice (data set)
  - Semaphore `chopstick[5]` initialized to 1
    - Each one needs two sticks to actually eat
    - Can also use alternating forks and knives between odd number of hungry philosophers



Dining Philosophers Problem (2)

- The structure of Philosopher *I*

```
while (TRUE)  {
  wait(chopstick[i] );   // get left chopstick
  wait(chopStick[ (i + 1) % 5] );
                         // get right chopstick
  //  eat

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

  //  think
};
```

Problems with Semaphores

- ❑ Incorrect use of semaphore operations (timing errors)
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- Only one process may be active within the monitor at any time
  - CS mutex applied to each monitor function

```
monitor monitor-name
{
  // shared variable declarations
  procedure F1(…) { … }
  …
  procedure Fn(…) { … }
  Initialization_code ( … ) { … }
  … }
```

Monitor Mutex using Semaphores

- Semaphore 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(next)
else
   signal(mutex);
```

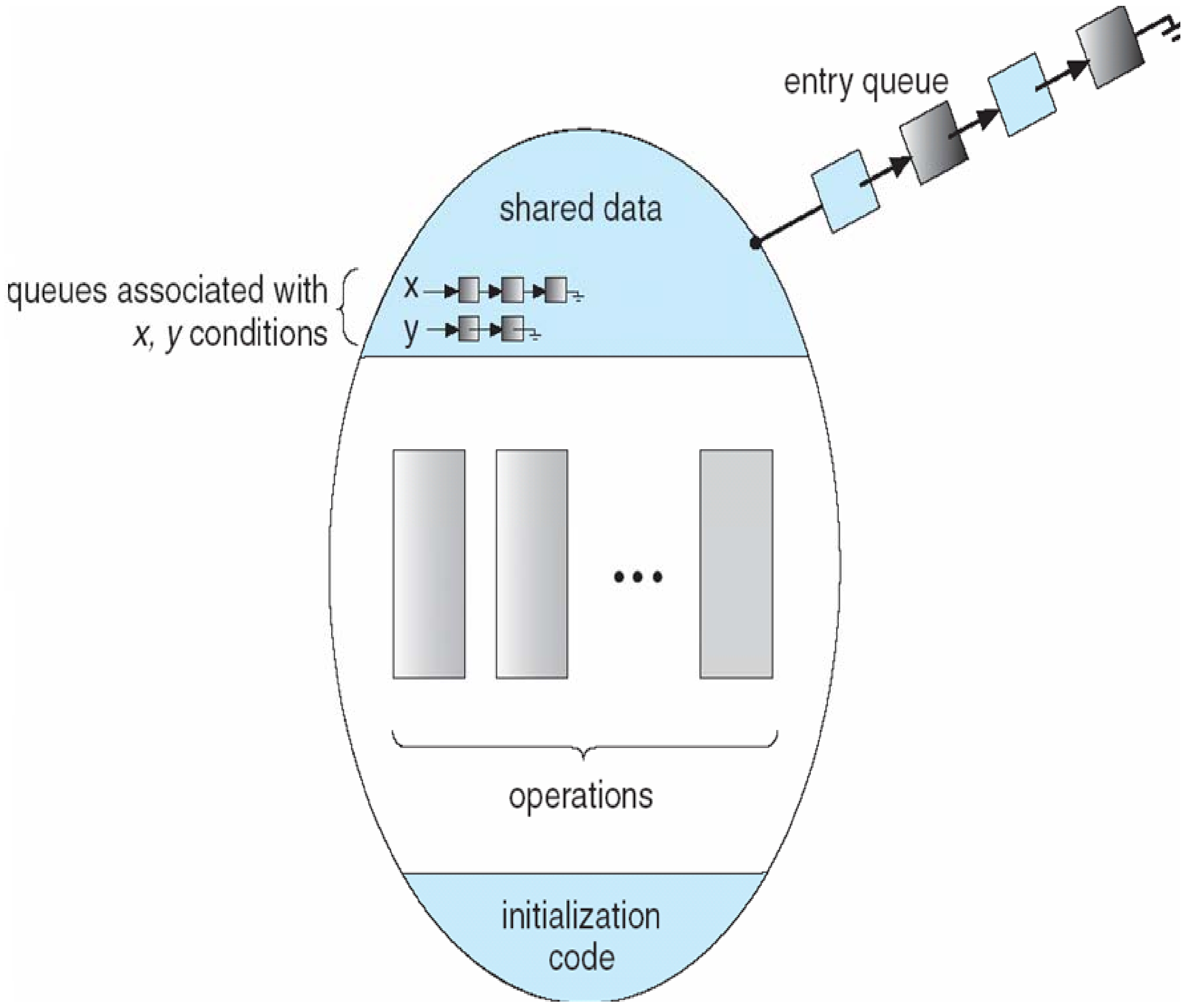
- Mutual exclusion is ensured within a monitor
  - Mutex lock is passed rather than released
  - Acquired from one to another waiting process

Condition Variables

- `Condition x, y`
- Two operations on a condition variable
  - `x.wait()`
    - A process that invokes the operation is suspended.
  - `x.signal()`
    - Resumes one of the processes (if any) that invoked x.wait ()

- Note that actual re-scheduling after suspended and resumed processes is done only at the end of the operation on condition variables

Monitor with Condition Variables



Monitor Implementation (1)

- For each condition variable *x*

```
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 (2)

- The operation `x.signal()` can be implemented as

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

Monitor Approach to Dining Philosophers

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

```
DiningPhilosophters.pickup(i);

// EAT

DiningPhilosophers.putdown(i);
```

Monitor Solution to Dining Philosophers (1)

```
monitor DiningPhilosophters
{
  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);
  }
```

Monitor Solution to Dining Philosophers (2)

```
   void test(int i) {
     if ((state[i] == HUNGRY) &&
         (state[(i+4)%5] != EATING) &&
         (state[(i+1)%5] != EATING)){
       state[i] = EATING;
       self[i].signal();
     }
   }
   initialization_code() {
     for (int i = 0; i < 5; i++)
       state[i] = THINKING;
   }
 }
```

Monitor to Allocate a 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;
}}
```

Linux Synchronization (Example)

- Linux Versions
  - Prior to kernel Version 2.6, Linux disables interrupts to implement short critical sections
  - Version 2.6 and later, Linux follows a fully preemptive approach

- Linux provides
  - Semaphores
  - Spin locks

Pthreads Synchronization (Example)

- Pthreads API is fully OS-independent
- The API provides
  - mutex locks
  - Condition variables
- Non-portable extensions include
  - Read-write locks
  - Spin locks

Mutual Exclusion

- Access to critical sections in Pthreads can be controlled using mutex locks
- A thread entering a critical section must request to acquire a lock on it first
  - Cannot go ahead, when the lock acquisition fails
- Pthreads API provides following functions for handling mutex-locks

```
int pthread_mutex_lock(pthread_mutex_t *mutex_lock);

int pthread_mutex_unlock(pthread_mutex_t *mutex_lock);

int pthread_mutex_init(pthread_mutex_t *mutex_lock,
    const pthread_mutexattr_t *lock_attr);
```

Pthreads' Types of Mutexes

- Pthreads support three types of mutexes
  - A normal mutex deadlocks, if a thread that already has a lock tries a second lock on it
  - A recursive mutex allows a single thread to lock a mutex as many times as it wants, it simply increments a count on the number of locks, a lock is relinquished by a thread when the count becomes zero
  - An error check mutex reports an error, when a thread with a lock tries to lock it again (as opposed to deadlocking in the first case, or granting the lock, as in the second case)
- The type of the mutex can be set in attributes object before it is passed at time of initialization.

Overheads of Locking

- Locks represent serialization points, since critical sections must be executed by threads one after the other
  - Encapsulating large segments of the program within locks can lead to significant performance degradation
- It is often possible to reduce idling or wait-time overhead associated with locks using an alternate function, `pthread_mutex_trylock`
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex_lock);`
  - `pthread_mutex_trylock` is typically much faster than `pthread_mutex_lock` on typical systems, since it does not have to deal with queues associated with locks for multiple threads waiting on the same lock