SYNCHRONIZATION

Operating Systems 2015 Spring by Euiseong Seo

Chapter 5: Process Synchronization

- Background
- The Critical-Section Problem
- Peterson's Solution
- Synchronization Hardware
- Mutex Locks
- Semaphores
- Classic Problems of Synchronization
- Monitors
- Synchronization Examples
- Alternative Approaches

Objectives

- To present the concept of process synchronization
- To introduce the critical-section problem, whose solutions can be used to ensure the consistency of shared data
- To present both software and hardware solutions of the critical-section problem
- To examine several classical process-synchronization problems
- To explore several tools that are used to solve process synchronization problems

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.

Synchronization Problem

- Two concurrent threads (or processes) access a shared resource without any synchronization
- Creates a race condition
  - The situation where several processes access and manipulate shared data concurrently
  - The result is non-deterministic and depends on timing
- We need mechanisms for controlling access to shared resources in the face of concurrency
  - So that we can reason about the operation of programs
- Synchronization is necessary for any shared data structure
  - buffers, queues, lists, etc.

Sharing Resources

- Between threads
  - Local variables are not shared
    - Refer to data on the stack
    - Each thread has its own stack
    - Never pass/share/store a pointer to a local variable on another thread's stack
  - Global variables are shared
    - Stored in static data segment, accessible by any thread
  - Dynamic objects are shared
    - Stored in the heap, shared through the pointers
- Between processes
  - Shared-memory objects, files, etc. are shared

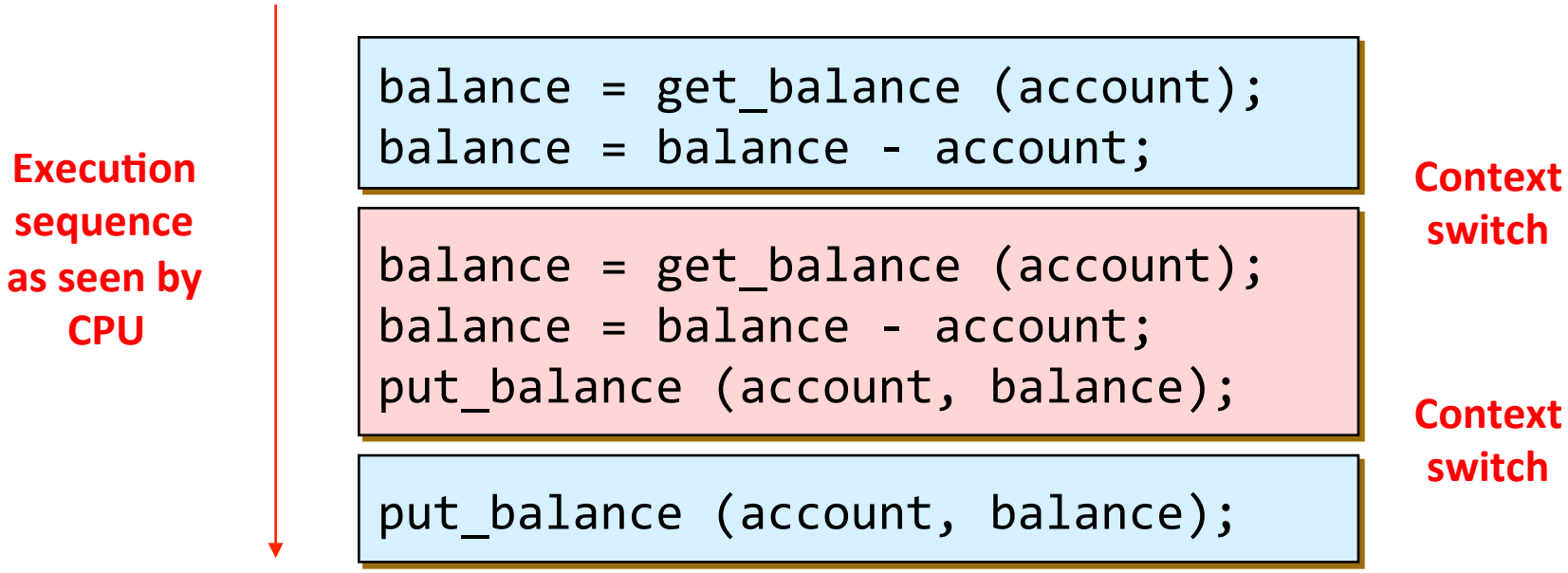
The Classic Example

- Withdraw money from a bank account
  - Suppose you and your girl(boy) friend share a bank account with a balance of 1,000,000won
  - What happens if both go to separate ATM machines, and simultaneously withdraw 100,000won from the account?

```
int withdraw (account, amount)
{
   balance = get_balance (account);
   balance = balance - amount;
   put_balance (account, balance);
   return balance;
}
```

The Classic Example

- Interleaved schedules
  - Represent the situation by creating a separate thread for each person to do the withdrawals
  - The execution of the two threads can be interleaved, assuming preemptive scheduling:



Critical Sections

- Critical sections
  - Critical sections are parts of the program that access shared memory or shared files or other shared resources
  - We want to use mutual exclusion to synchronize access to shared resources in critical sections
    - Only one thread at a time can execute in the critical section
    - All other threads are forced to wait on entry
    - When a thread leaves a critical section, another can enter
  - Otherwise, critical sections can lead to race conditions
    - The final result depends on the sequence of execution of the processes

Critical Sections

- Requirements
  - Mutual exclusion
    - At most one thread is in the critical section
  - Progress
    - If no process is executing in its critical section, then the selection of the next processes to enter the section cannot be postponed indefinitely
  - Bounded waiting (no starvation)
    - If thread T is waiting on the critical section, then T will eventually enter the critical section
  - Performance
    - The overhead of entering and exiting the critical section is small with respect to the work being done within it

Critical Sections

- Mechanisms for building critical sections
  - Locks
    - Very primitive, minimal semantics, used to build others.
  - Semaphores
    - Basic, easy to get the hang of, hard to program with.
  - Monitors
    - High-level, requires language support, implicit operations.
    - Easy to program with: Java “synchronized”
  - Messages
    - Simple model of communication and synchronization based on (atomic) transfer of data across a channel.
    - Direct application to distributed systems.

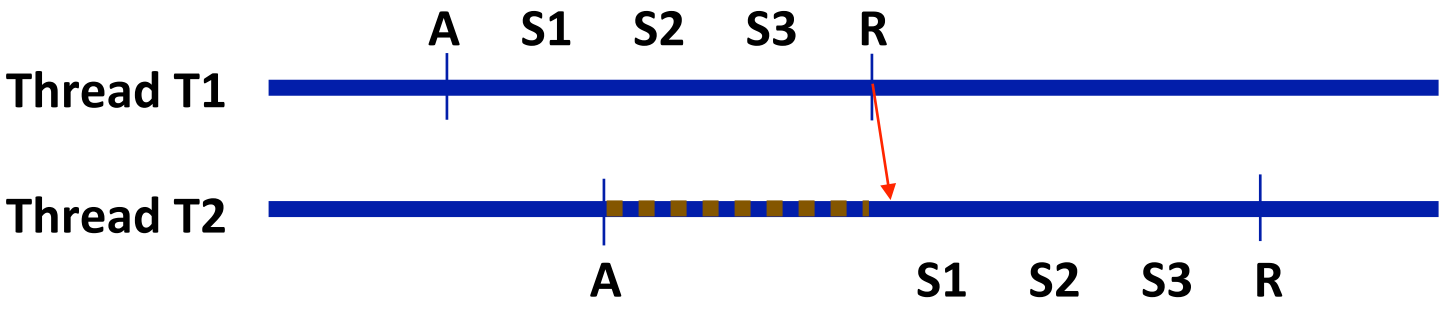
Locks

- A lock is an object (in memory) that provides the following two operations:
  - acquire(): wait until lock is free, then grab it
  - release(): unlock, and wake up any thread waiting in acquire()
- Using locks
  - Lock is initially free
  - Call acquire() before entering a critical section, and release() after leaving it
  - Between acquire() and release(), the thread holds the lock
  - acquire() does not return until the caller holds the lock
  - At most one thread can hold a lock at a time
- Locks can spin (a spinlock) or block (a mutex)

Using Locks

```
A     int withdraw (account, amount)
      {
A        acquire (lock);
S1       balance = get_balance (account);
S2       balance = balance - amount;
S3       put_balance (account, balance);
R        release (lock);
         return balance;
      }
```

Critical section



Implementing Locks

- An initial attempt

```
struct lock { int held = 0; }

void acquire (struct lock *l)  {
   while (l->held);
   l->held = 1;
}
void release (struct lock *l)   {
   l->held = 0;
}
```

The caller “busy-waits”, or spins for locks to be released, hence spinlocks.

  - Does this work?

Implementing Locks

- Problem
  - Implementation of locks has a critical section, too!
    - The acquire/release must be atomic
    - A recursion, huh?
  - Atomic operation
    - Executes as though it could not be interrupted
    - Code that executes “all or nothing”

Implementing Locks

- Solutions
  - Software-only algorithms
    - Dekker's algorithm (1962)
    - Peterson's algorithm (1981)
    - Lamport's Bakery algorithm for more than two processes (1974)
  - Hardware atomic instructions
    - Test-and-set, compare-and-swap, etc.
  - Disable/reenable interrupts
    - To prevent context switches

Software-only Algorithms

- Wrong algorithm
  - Mutual exclusion?
  - Progress?

```
int interested[2];

void acquire (int process)   {
   int other = 1 – process;
   interested[process] = TRUE;
   while (interested[other]);
}

void release (int process)   {
   interested[process] = FALSE;
}
```

Peterson's Algorithm

- Solves the critical section problem for two processes

```
int turn;
int interested[2];

void acquire (int process)   {
   int other = 1 – process;
   interested[process] = TRUE;
   turn = other;
   while (interested[other] && turn == other);
}

void release (int process)   {
   interested[process] = FALSE;
}
```

Bakery Algorithm

- Multiple-process solution
  - Before entering its critical section, process receives a sequence number
  - Holder of the smallest number enters the critical section
  - If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first
  - The numbering scheme always generates numbers in increasing order of enumeration; i.e. 1,2,3,3,3,4,4,5…

Bakery Algorithm (2)

```
int number[N];
int choosing[N];

#define EARLIER(a,b)   \\
   ((number[a] < number[b]) || \\
   (number[a] == number[b] && \\
    (a) < (b)))

int Findmax ()   {
   int i;
   int max = number[0];
   for (i = 1; i < N; i++)
      if (number[i] > max)
         max = number[i];
   return max;
}
```

```
void acquire (int me)   {
   int other;
   choosing[me] = TRUE;
   number[me] = Findmax() + 1;
   choosing[me] = FALSE;
   for (other=0; other<N; other++)
   {
      while (choosing[other]);
      while (number[other] &&
                EARLIER(other, me));
   }
}

void release (int me)   {
   number[me] = 0;
}
```

Disabling Interrupts

- Implementing locks by disabling interrupts

```
void acquire (struct lock *l)  {
    cli();          // disable interrupts;
}
void release (struct lock *l)  {
    sti();          // enable interrupts;
}
```

  - Disabling interrupts blocks notification of external events that could trigger a context switch (e.g., timer)
  - There is no state associate with the lock.
  - Can two threads disable interrupts simultaneously?

Disabling Interrupts

- What's wrong?
  - Only available to kernel
    - Why not have the OS support these as system calls?
  - Insufficient on a multiprocessor
    - Back to atomic instructions
  - What if the critical section is long?
    - Can miss or delay important events.

      (e.g., timer, I/O)
  - Like spinlocks, only use to implement higher-level synchronization primitives

Atomic Instructions

- Test-and-Set

```
int TestAndSet (int *v)   {
    int rv = *v;
    *v = 1;
    return rv;
}
```

- Using Test-and-Set instruction

```
void struct lock { int value = 0; }

void acquire (struct lock *l)   {
    while (TestAndSet (&l->value));
}
void release (struct lock *l)   {
    l->value = 0;
}
```

Atomic Instructions

- Swap

```
void Swap (int *v1, int *v2)   {
    int temp = *v1;
    *v1 = *v2;
    *v2 = temp;
}
```

- Using Swap instruction

```
void struct lock { int value = 0; }
void acquire (struct lock *l)   {
    int key = 1;
    while (key == 1) Swap(&l->value, &key);
}
void release (struct lock *l)   {
    l->value = 0;
}
```

Atomic Instructions

- Locks using Test-and-Set with bounded-waiting

```
struct lock { int value = 0; }
int waiting[N];

void acquire (struct lock *l,
              int me)
{
   int key;

   waiting[me] = 1;
   key = 1;
   while (waiting[me] && key)
     key = TestAndSet (&l->value);
   waiting[me] = 0;
}
```

```
void release (struct lock *l,
              int me)
{
   int next = (me + 1) % N;

   while ((next != me) &&
          !waiting[next])
      next = (next + 1) % N;
   if (next == me)
      l->value = 0;
   else
      waiting[next] = 0;
}
```

Problems with Spinlocks

- Spinlocks
  - Horribly wasteful!
    - If a thread is spinning on a lock, the thread holding the lock cannot make progress
    - The longer the critical section, the longer the spin
    - CPU cycle is wasted
    - Greater the chances for lock holder to be interrupted through involuntary context switch
  - Only want to use spinlock as primitives to build higher-level synchronization constructs

Higher-level Synchronization

- Motivation
  - Spinlocks and disabling interrupts are useful only for very short and simple critical sections
    - Wasteful otherwise
    - These primitives are “primitive” – don’t do anything besides mutual exclusion
  - Need higher-level synchronization primitives that
    - Block waiters
    - Leave interrupts enabled within the critical section
  - Two common high-level primitives:
    - Semaphores: binary (mutex) and counting
    - Monitors: mutexes and condition variables
  - We’ll use our “atomic” locks as primitives to implement them

Semaphores

- Semaphores
  - A synchronization primitive higher level than locks
  - Invented by Dijkstra in 1968, as part of the THE OS
  - Does not require busy waiting
  - Manipulated atomically through two operations
    - Wait (S): decrement, block until semaphore is open

      = P(), after Dutch word for test, also called down()
    - Signal (S): increment, allow another to enter

      = V(), after Dutch word for increment, also called up()

Semaphores

- Blocking in semaphores
  - Each semaphore has an associated queue of processes/threads
  - When wait() is called by a thread,
    - If semaphore is “open”, thread continues
    - If semaphore is “closed”, thread blocks, waits on queue
  - Signal() opens the semaphore
    - If thread(s) are waiting on a queue, one thread is unblocked
    - If no threads are on the queue, the signal is remembered for next time a wait() is called
  - In other words, semaphore has history
    - The history is a counter
    - If counter falls below 0, then the semaphore is closed
    - wait() decreases the counter while signal() increases it

Implementing Semaphores

```
typedef struct {
    int value;
    struct process *L;
} semaphore;
void wait (semaphore S)  {
    S.value--;
    if (S.value < 0)   {
        add this process to S.L;
        block ();
    }
}
void signal (semaphore S)  {
    S.value++;
    if (S.value <= 0)  {
        remove a process P from S.L;
        wakeup (P);
    }
}
```

wait() / signal()
are critical sections!
Hence, they must be
executed atomically
w.r.t.
each other.

HOW??

Types of Semaphores

- Binary semaphore (a.k.a mutex)
  - Guarantees mutually exclusive access to resource
  - Only one thread/process allowed entry at a time
  - Counter is initialized to 1
- Counting semaphore
  - Represents a resource with many units available
  - Allows threads/processes to enter as long as more units are available
  - Counter is initialized to N (=units available)

Bounded Buffer Problem

- Producer/consumer problem
  - There is a set of resource buffers shared by producer and consumer
  - Producer inserts resources into the buffer
    - Output, disk blocks, memory pages, etc.
  - Consumer removes resources from the buffer
    - Whatever is generated by the producer
  - Producer and consumer execute in different rates
    - No serialization of one behind the other
    - Tasks are independent
    - The buffer allows each to run without explicit handoff

Bounded Buffer Problem

- No synchronization

Producer

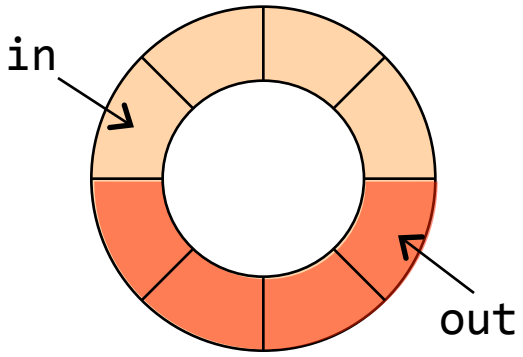
```
void produce(data)
{

  while (count==N);
  buffer[in] = data;
  in = (in+1) % N;
  count++;

}
```

```
int count;

struct item buffer[N];
int in, out;
```



Consumer

```
void consume(data)
{

  while (counter==0);
  data = buffer[out];
  out = (out+1) % N;
  count--;

}
```

Bounded Buffer Problem

- Implementation with semaphores

Producer

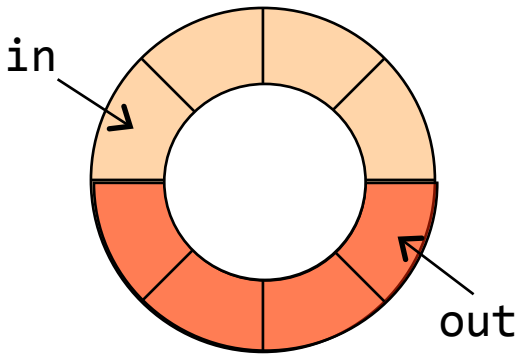
```
void produce(data)
{

  wait (empty);
  wait (mutex);
  buffer[in] = data;
  in = (in+1) % N;
  signal (mutex);
  signal (full);

}
```

```
Semaphore
     mutex = 1;
     empty = N;
     full = 0;
```

```
struct item buffer[N];
     int in, out;
```



Consumer

```
void consume(data)
{

  wait (full);
  wait (mutex);
  data = buffer[out];
  out = (out+1) % N;
  signal (mutex);
  signal (empty);

}
```

Readers-Writers Problem

- Readers-Writers problem
  - An object is shared among several threads
  - Some threads only read the object, others only write it
  - We can allow multiple readers at a time
  - We can only allow one writer at a time
- Implementation with semaphores
  - readcount – # of threads reading object
  - mutex – control access to readcount
  - rw – exclusive writing or reading

Readers-Writers Problem

```
// number of readers
int readcount = 0;
// mutex for readcount
Semaphore mutex = 1;
// mutex for reading/writing
Semaphore rw = 1;

void Writer ()
{
    wait (rw);
    …
    Write
    …
    signal (rw);
}
```

```
void Reader ()
{
    wait (mutex);
    readcount++;
    if (readcount == 1)
        wait (rw);
    signal (mutex);
    …
    Read
    …
    wait (mutex);
    readcount--;
    if (readcount == 0)
        signal (rw);
    signal (mutex);
}
```

Readers-Writers Problem

- Note
  - If there is a writer
    - The first reader blocks on rw
    - All other readers will then block on mutex
  - Once a writer exits, all readers can fall through
    - Which reader gets to go first?
  - The last reader to exit signals waiting writer
    - Can new readers get in while writer is waiting?
  - When writers exits, if there is both a reader and writer waiting, which one goes next is up to scheduler

Dining Philosopher

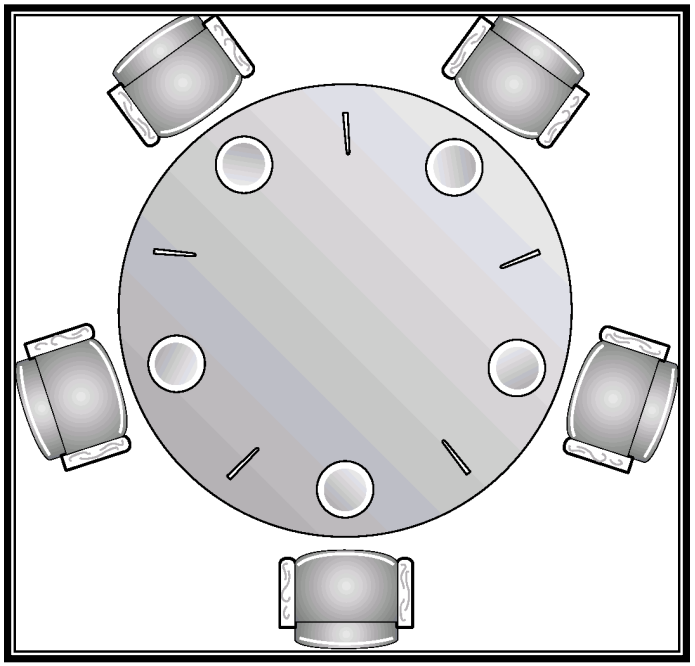
- Dining philosopher problem
  - Dijkstra, 1965
  - Life of a philosopher
    - Repeat forever:

      Thinking

      Getting hungry

      Getting two chopsticks

      Eating



Dining Philosopher

- A simple solution

```
Semaphore chopstick[N];  // initialized to 1
void philosopher (int i)
{
    while (1)  {
        think ();
        wait (chopstick[i]);
        wait (chopstick[(i+1) % N];
        eat ();
        signal (chopstick[i]);
        signal (chopstick[(i+1) % N];
    }
}
```

⇒ Problem: causes deadlock

Dining Philosopher

- Deadlock-free version: starvation?

```
#define N          5
#define L(i)       ((i+N-1)%N)
#define R(i)       ((i+1)%N)
void philosopher (int i)  {
  while (1)  {
    think ();
    pickup (i);
    eat();
    putdown (i);
  }
}
void test (int i)  {
  if (state[i]==HUNGRY &&
      state[L(i)]!=EATING &&
      state[R(i)]!=EATING)  {
     state[i] = EATING;
     signal (s[i]);
 }
```

```
Semaphore mutex = 1;
Semaphore s[N];
int state[N];

void pickup (int i)  {
  wait (mutex);
  state[i] = HUNGRY;
  test (i);
  signal (mutex);
  wait (s[i]);
}
void putdown (int i)  {
  wait (mutex);
  state[i] = THINKING;
  test (L(i));
  test (R(i));
  signal (mutex);
}
```

Problems with Semaphores

- Drawbacks
  - They are essentially shared global variables
    - Can be accessed from anywhere (bad software engineering)
  - There is no connection between the semaphore and the data being controlled by it
  - Used for both critical sections (mutual exclusion) and for coordination (scheduling)
  - No control over their use, no guarantee of proper usage
- Thus, hard to use and prone to bugs
  - Another approach: use programming language support

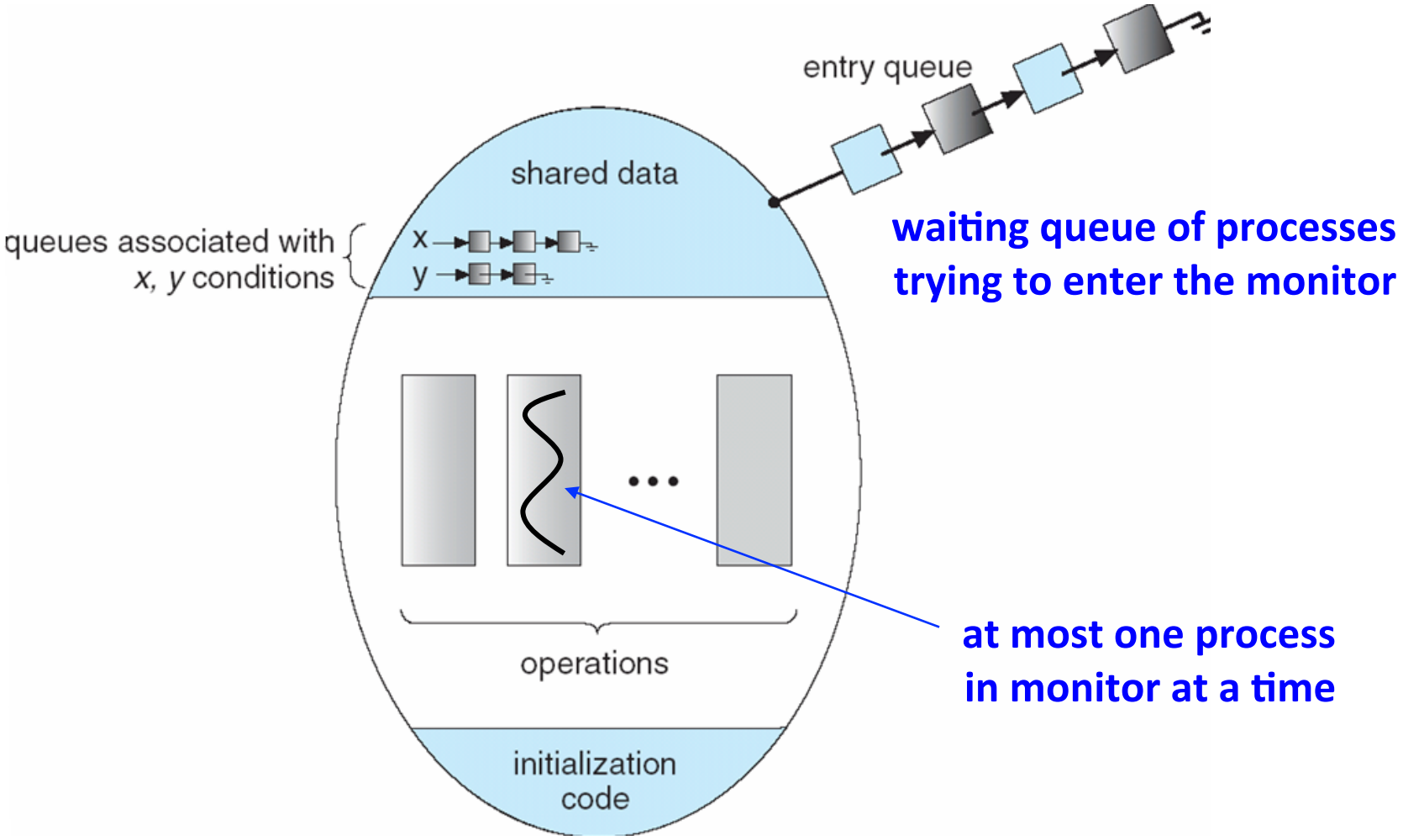
Monitors

- A programming language construct that supports controlled access to shared data
  - Synchronization code added by compiler, enforced at runtime
  - Allows the safe sharing of an abstract data type among concurrent processes
- A monitor is a software module that encapsulates
  - shared data structures
  - procedures that operate on the shared data
  - synchronization between concurrent processes that invoke those procedures
- Monitor protects the data from unstructured access
  - guarantees only access data through procedures, hence in legitimate ways

Monitors

- Mutual exclusion
  - Only one process can be executing inside at any time
    - Thus, synchronization implicitly associated with monitor
  - If a second process tries to enter a monitor procedure, it blocks until the first has left the monitor
    - More restrictive than semaphores
    - But easier to use most of the time
- Condition variables
  - Once inside, a process may discover it can't continue, and may wish to sleep, or allow some other waiting process to continue
  - Condition variables are provided within monitor
    - Processes can wait or signal others to continue
    - Can only be accessed from inside monitor

Monitors


entry queue
shared data
queues associated with
x, y conditions
x
y
operations
initialization
code
waiting queue of processes
trying to enter the monitor
at most one process
in monitor at a time

Condition Variables

- Purpose
  - provides a mechanism to wait for events
    (a “rendezvous point”)
- Three operations
  - wait (c)
    - release monitor lock, so somebody else can get in
    - wait for somebody else to signal condition
    - thus, condition variables have wait queues
  - signal (c)
    - wake up at most one waiting process
    - if no waiting processes, signal is lost
    - this is different from semaphores: no history!
  - broadcast (c)
    - wake up all waiting processes

Bounded Buffer with Monitors

```
Monitor bounded_buffer {
    buffer resources[N];
    condition not_full, not_empty;

    procedure add_entry (resource x)  {
        while (array “resources” is full)
            wait (not_full);
        add “x” to array “resources”;
        signal (not_empty);
    }

    procedure remove_entry (resource *x)  {
        while (array “resources” is empty)
            wait (not_empty);
        *x = get resources from array “resources”
        signal (not_full);
    }
}
```

Monitors Semantics

- Hoare monitors
  - signal(c) immediately switches from the caller to a waiting thread, blocking the caller
    - The condition that the waiter was anticipating is guaranteed to hold when waiter executes
    - Signaler must restore monitor invariants before signaling
- Mesa monitors
  - signal(c) places a waiter on the ready queue, but signaler continues inside monitor
    - Condition is not necessarily true when waiter runs again
    - Being woken up is only a hint that something has changed
    - Must recheck conditional case

Monitors Semantics

- Comparison
  - Usage:

Hoare monitors

```
if (notReady)
    wait (c);
```

Mesa monitors

```
while (notReady)
    wait (c);
```

  - Mesa monitors easier to use
    - more efficient
    - fewer switches
    - directly supports broadcast()
  - Hoare monitors leave less to chance
    - when wake up, condition guaranteed to be what you expect

Hoare Monitors

```
Semaphore mutex = 1;
Semaphore next = 0;
int next_count = 0;
struct condition {
    Semaphore sem;
    int count;
} x = {0, 0};

procedure F ()  {
    wait (mutex);
    …
    Body of F
    …
    if (next_count)
        signal (next);
    else
        signal (mutex);
}
```

```
procedure cond_wait (x)  {
    x.count++;
    if (next_count)
        signal (next);
    else
        signal (mutex);
    wait (x.sem);
    x.count--;
}

procedure cond_signal (x)  {
    if (x.count)  {
        next_count++;
        signal (x.sem);
        wait (next);
        next_count--;
    }
}
```

Monitors and Semaphores

- Comparison
  - Condition variables do not have any history, but semaphores do
    - On a condition variable signal(), if no one is waiting , the signal is a no-op

      (If a thread then does a condition variable wait(), it waits)
    - On a semaphore signal(), if no one is waiting, the value of the semaphore is increased

      (If a thread then does a semaphore wait(), the value is decreased and the thread continues)

Condition Variables and Mutex

- Yet another construct
  - Condition variables can be also used without monitors in conjunction with mutexes
  - Think of a monitor as a language feature
    - Under the covers, compiler knows about monitors
    - Compiler inserts a mutex to control entry and exit of processes to the monitor's procedures
    - But can be done anywhere in procedure, at finer granularity
  - With condition variables, the module methods may wait and signal on independent conditions

Synchronization in Pthreads

```
pthread_mutex_t mutex;
pthread_cond_t not_full, not_empty;
buffer resources[N];
void add_entry (resource x)  {
    pthread_mutex_lock (&mutex);
    while (array “resources” is full)
        pthread_cond_wait (&not_full, &mutex);
    add “x” to array “resources”;
    pthread_cond_signal (&not_empty);
    pthread_mutex_unlock (&mutex);
}
void remove_entry (resource *x)  {
    pthread_mutex_lock (&mutex);
    while (array “resources” is empty)
        pthread_cond_wait (&not_empty, &mutex);
    *x = get resource from array “resources”
    pthread_cond_signal (&not_full);
    pthread_mutex_unlock (&mutex);
}
```

Synchronization Mechanisms

- Disabling interrupts
- Spinlocks
  - Busy waiting
- Semaphores
  - Binary semaphore = mutex (≅ lock)
  - Counting semaphore
- Monitors
  - Language construct with condition variables
- Mutex + Condition variables
  - Pthreads