ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Real-Time Operating Systems M

5. Process Synchronization

Notice

The course material includes slides downloaded from:

http://codex.cs.yale.edu/avi/os-book/

(slides by Silberschatz, Galvin, and Gagne, associated with Operating System Concepts, 9th Edition, Wiley, 2013)

and

http://retis.sssup.it/~giorgio/rts-MECS.html

(slides by Buttazzo, associated with Hard Real-Time Computing Systems, 3rd Edition, Springer, 2011)

which has been edited to suit the needs of this course.

The slides are authorized for personal use only.

Any other use, redistribution, and any for profit sale of the slides (in any form) requires the consent of the copyright owners.

For solutions to exercises in this section, also refer to *The Little Book of Semaphores*, http://www.greenteapress.com/semaphores/

Process Synchronization

1. Background
2. The Critical-Section Problem
3. Peterson's Solution
4. Synchronization Hardware
5. Semaphores
6. Classic Problems of Synchronization
7. Monitors
8. Synchronization Examples: Solaris, Windows XP, Linux, Pthreads API.
9. ~~Atomic Transactions~~

Objectives

- 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: consumer-producer revisited

Background

- 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}

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, ... 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 is

```
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

- Two approaches depending on if kernel is preemptive or non-preemptive
  - **Preemptive** – allows preemption of process when running in kernel mode
  - **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
    - Essentially free of race conditions in kernel mode

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the **`load`** and **`store`** 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!

Algorithm for Process P_i

```
do {
    flag[i] = TRUE;
    turn = j;
    while (flag[j] && turn == j);
            critical section
    flag[i] = FALSE;
            remainder section
} while (TRUE);
```

- Provable that

1. Mutual exclusion is preserved
2. Progress requirement is satisfied
3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for critical section code
- All solutions below based on idea of **locking**
  - Protecting critical regions via locks
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special atomic hardware instructions
    - **Atomic** = non-interruptible
  - Either test memory word and set value
  - Or swap contents of two memory words

Solution to Critical-section Problem Using Locks

```
do {
    acquire lock
        critical section
    release lock
        remainder section
} while (TRUE);
```

test_and_set() Instruction

- Definition:

```
boolean test_and_set (boolean *target)
 {
      boolean rv = *target;
      *target = TRUE;
      return rv:
 }
```

Solution using `test_and_set()`

- Shared Boolean variable **`lock`**, initialized to **`FALSE`**
- Solution:

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = FALSE;

      /* remainder section */
} while (TRUE);
```

compare_and_swap() Instruction

- Definition:

```
int compare_and_swap(int *value, int expected, int new_value) {
   int temp = *value;
   if (*value == expected)
      *value = new_value;
   return temp;
}
```

Solution using compare_and_swap()

- Shared Boolean variable lock initialized to **FALSE**
- Solution:

```
do {
   while (compare_and_swap(&lock, FALSE, TRUE))
      ; /* do nothing */

      /* critical section */

   lock = FALSE;

      /* remainder section */
} while (TRUE);
```

Solution (?) using `test_and_set()`

- <u>Shared Boolean variable</u> `lock`, initialized to **`FALSE`**

```
boolean test_and_set (boolean *target) {
     boolean rv = *target;
     *target = TRUE;
     return rv:
}
```

```
do {
   while (test_and_set(&lock))
        ; /* do nothing */
        /* critical section */
   lock = FALSE;
        /* remainder section */
} while (TRUE);
```

Bounded-waiting Mutual Exclusion

- Shared Boolean variables `lock` and `flag[n]` initialized to **FALSE**

```
do {
   flag[i] = TRUE; /* waiting to be granted access to critical section */
   while (flag[i] && test_and_set(&lock))
         ; /* do nothing */
   flag[i] = FALSE;
            /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !flag[j])
      j = (j + 1) % n;
   if (j == i)
      lock = FALSE;
   else
      flag[j] = FALSE;
            /* remainder section */
} while (TRUE);
```

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 critical regions with it by first **`acquire()`** a lock then **`release()`** it
  - Boolean variable indicating if lock is available or not
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - Usually implemented via hardware atomic instructions

acquire() and release()

```
acquire() {
   while (!available)
      ; /* busy wait */
   available = FALSE;
}
release() {
   available = TRUE;
}

do {
  acquire lock
     critical section
  release lock
     remainder section
} while (TRUE);
```

- Solution that requires **busy waiting**
  - Called a **spinlock**
  - Not necessarily a useless solution
    - No context switch required when a process must wait on a lock
    - **Useful when locks expected for short periods of time**
    - Especially in **multiprocessor** systems

Semaphore

- Synchronization tool that does not require busy waiting
- Semaphore ***S*** – **integer variable**
- Two standard operations modify ***S***: **`wait()`** and **`signal()`**
  - Less complicated
  - Can only be accessed via two indivisible (atomic) operations

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

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

Semaphore Usage

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

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

Semaphore Implementation

- Must guarantee that no two processes can execute **`wait()`** and **`signal()`** on the same semaphore at the same time
- Thus, 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

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

Semaphore Implementation with no Busy waiting

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

```
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);
    }

}
```

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

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

P_1

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

- **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
  - Solved via **priority-inheritance protocol**

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

- Buffer with ***n*** slots, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
  - Used to grant mutually exclusive access to buffer
- Semaphore **`full`** initialized to the value 0
  - Number of full slots
- Semaphore **`empty`** initialized to the value n
  - Number of empty slots

Bounded-Buffer Problem

- The structure of the producer process

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

Bounded-Buffer Problem

- The structure of the consumer process

```
do {
   wait(full);
   wait(mutex);
      ...
   /* remove an item from buffer to next_consumed */
      ...
   signal(mutex);
   signal(empty);
      ...
   /* 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
- Several variations of how readers and writers are treated – all involve priorities
- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem

- The structure of a writer process

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

Readers-Writers Problem

- The structure of a reader process

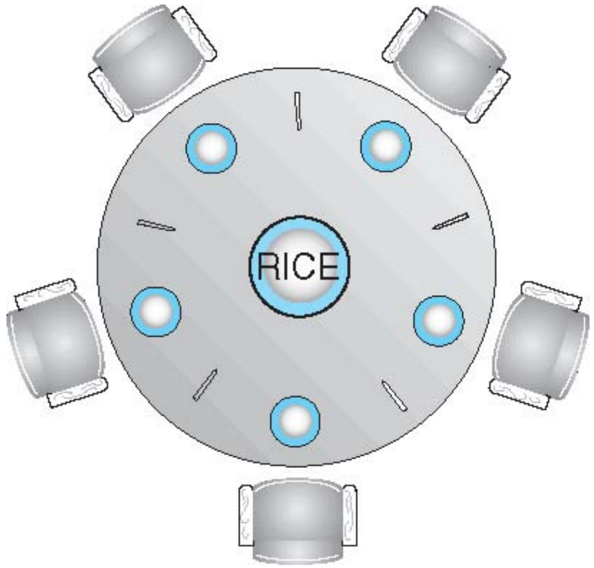
```
do {
        wait(mutex);
        read_count++;
        if (read_count == 1)
                wait(rw_mutex);
        signal(mutex);
                ...
        /* reading is performed */
                ...
        wait(mutex);
        read_count--;
        if (read_count == 0)
                signal(rw_mutex);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

Readers-Writers Problem Variations

- *First* variation – no reader kept waiting unless writer has permission to use shared object
- *Second* variation – once writer is ready, it performs write asap
- Both may have starvation leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem

- Philosophers spend their lives 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
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1



Dining-Philosophers Problem Algorithm

- 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?
- What are possible solutions?

Tanenbaum's Solution With Semaphores

```
enum {THINKING, EATING, HUNGRY} state[5];
semaphore self[5]; // initially: 0
semaphore mutex; // initially: 1

int left(i) { return (i+4)%5; }
int right(i) { return (i+1)%5; }

/* i-th philosopher */
while(TRUE) {
    /* THINK */
    pick_up(i);
    /* EAT */
    put_down(i);
}

pick_up(i) {
    wait(mutex);
    state[i] = HUNGRY;
    test(i);
    signal(mutex);
    wait(self[i]);
}

put_down(i) {
    wait(mutex);
    state[i] = THINKING;
    test(right(i));
    test(left(i));
    signal(mutex);
}

test(i) {
    if(state[i] == HUNGRY
       && state[left(i)] != EATING
       && state[right(i)] != EATING) {
        state[i] = EATING;
        signal(self[i]);
    }
}
```

Problems with Semaphores

- Incorrect use of semaphore operations:
  - signal (mutex) …. wait (mutex)
  - wait (mutex) … wait (mutex)
  - Omitting of wait (mutex) or signal (mutex) (or both)
- Deadlock and starvation

Monitors

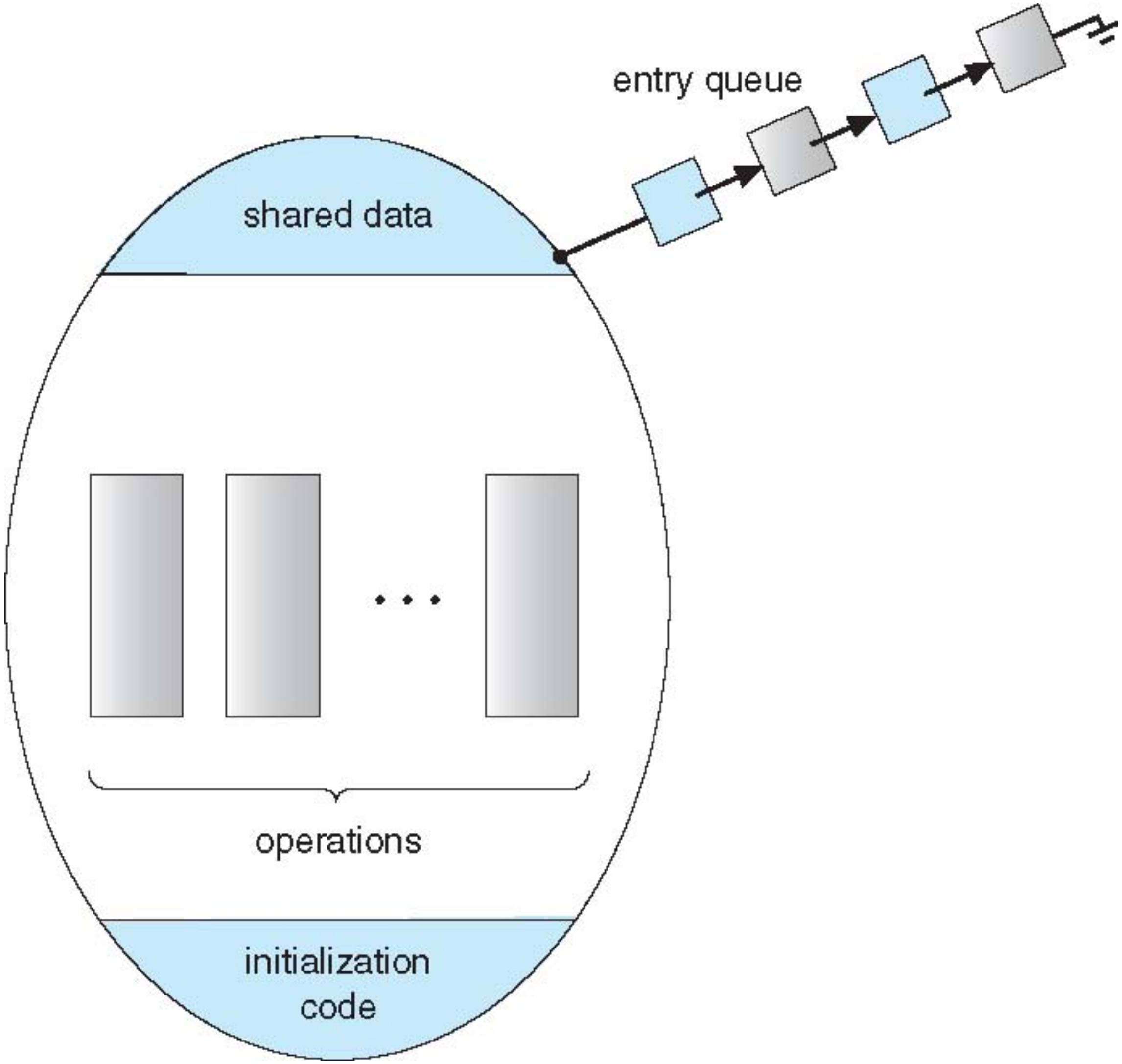
- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type*, internal variables only accessible by code within the procedure
- Only one process may be active within the monitor at a time
- But not powerful enough to model some synchronization schemes

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

          procedure Pn (…) {……}

          Initialization code (…) { … }
  }
}
```

Schematic view of a Monitor

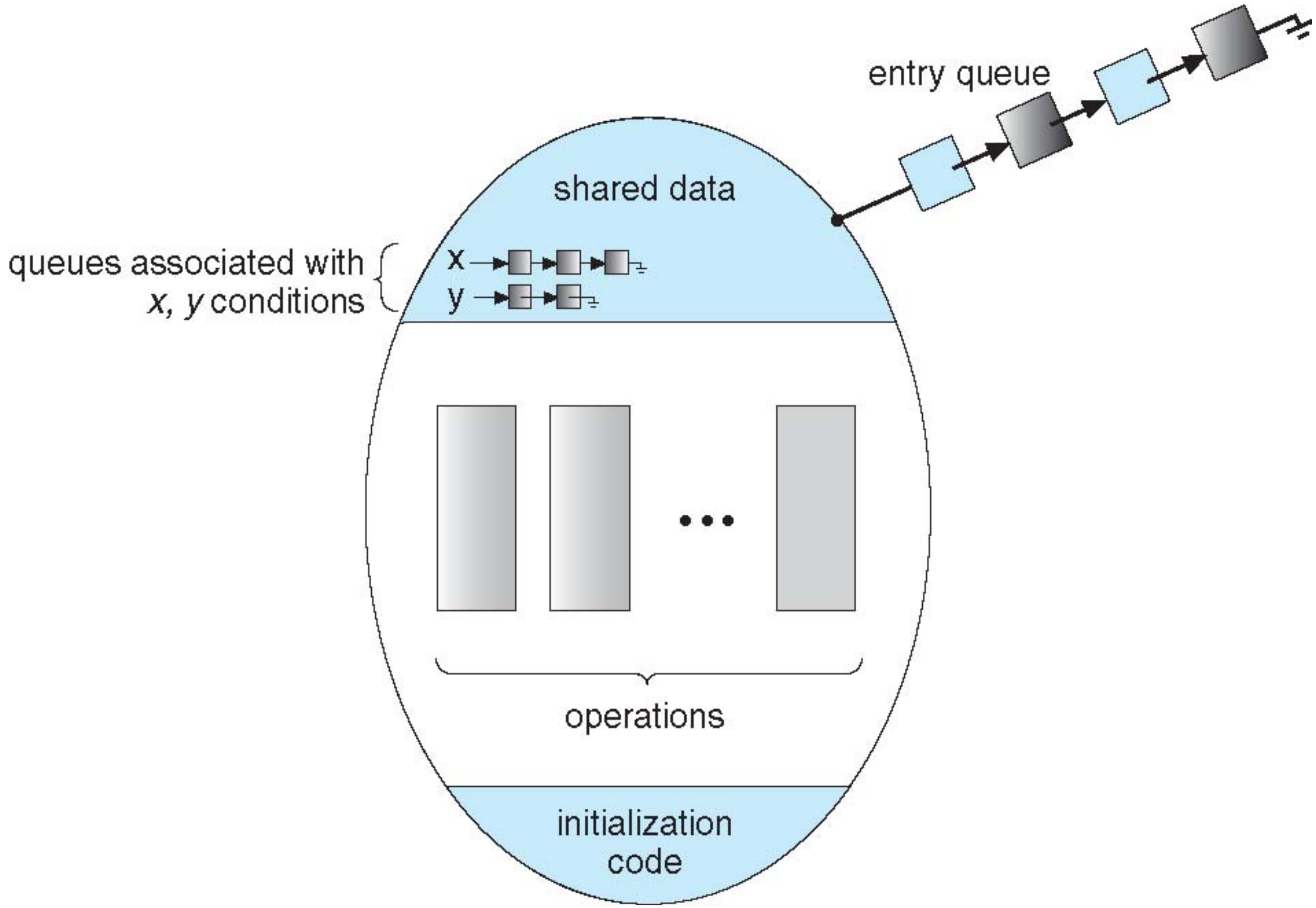


Condition Variables

- condition x, y;

- Two operations 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 (), with Q in x.wait () state, what should happen next?
  - If Q is resumed, then P must wait
- Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
  - Both have pros and cons
    - Reasonable to keep P running …
    - … but condition for keeping Q waiting may be false now
  - Language implementer can decide

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

```
    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;
        }
}
```

Solution to Dining Philosophers

- 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

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

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;
   }
}
```

Synchronization Examples

- Solaris
- Windows XP
- Linux
- Pthreads

Solaris Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing
- Uses **adaptive mutexes** for efficiency when protecting data from short code segments
  - Starts as a standard semaphore spin-lock
  - If lock held, and by a thread running on another CPU, spins
  - If lock held by non-run-state thread, block and sleep waiting for signal of lock being released
- Uses **condition variables**
- Uses **readers-writers** locks when longer sections of code need access to data
- Uses **turnstiles** to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock
  - Turnstiles are per-lock-holding-thread, not per-object
- Priority-inheritance per-turnstile gives the running thread the highest of the priorities of the threads in its turnstile

Windows XP Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses **spinlocks** on multiprocessor systems
  - Spinlocking-thread will never be preempted
- Also provides **dispatcher objects** for threads in user-mode, which may act mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Synchronization

- Linux:
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive

- Linux provides:
  - semaphores
  - spinlocks
  - reader-writer versions of both

- On single-cpu system, spinlocks replaced by enabling and disabling kernel preemption

Pthreads Synchronization

- Pthreads API is OS-independent
- It provides:
  - mutex locks
  - condition variables
- Non-portable extensions include:
  - read-write locks
  - spinlocks

Quizzes

- A wrong wait/signal sequence may cause starvation
- A wrong wait/signal sequence may cause a deadlock
- A deadlock-free solution is guaranteed to not cause starvation
- A **`x.signal()`** operation on a condition variable **`x`** may have no effect at all
- Spinlocks are effective especially on single-processor computer systems
- A way to solve the priority inversion problem is priority inheritance
- Critical sections should contain a many instructions as possible
- Peterson's solution to the critical section problem meets the bounded waiting requirement
- Semaphores cannot be implemented on computer architectures that provide the **`compare_and_swap()`** instruction only (**`test_and_set()`** is needed)

Exercise: Sleeping Barber

- Barbershop:
  - barber room with ***one* barber chair**
  - **waiting** room with ***n* chairs**
- One **barber** process, *m* **customer** processes
- System's behaviour:
  - If no customers to be served, barber falls asleep
  - If customer enters barbershop and all chairs occupied, customer leaves
  - If barber busy but chairs available, customer sits in one of free chairs
  - If barber asleep, customer wakes up barber

Exercise: Sleeping Barber

- Use the following semaphores and global variables.

```
semaphore mutex = 1;
semaphore customer, barber = 0; // is a customer/the barber available?
int customers = 0; // how may customers are in the barber shop

/* use functions cut_hair(), get_hair_cut(), and leave_shop() to represent actions */

/* a possible solution is in the next page */
```

Exercise: Sleeping Barber

```
/* customer */
while(TRUE) {
    wait(mutex);
    if(customers == n+1) {
        signal(mutex);
        leave_shop();
    }
    customers ++;
    signal(mutex);

    signal(customer);
    wait(barber);

    get_hair_cut();
    wait(mutex);
    customers--;
    signal(mutex);
}
```

```
/* barber */
while(TRUE) {
    wait(customer);
    signal(barber);
    cut_hair();
}
```

rendezvous

Exercise: Cigarette Smokers

- Three (chain-) **smoker** processes, one **arbiter** process
  - Each smoker continuously rolls a cigarette and smokes it
    - To roll a cigarette, needs **tobacco**, **paper**, **matches**
    - Has infinite supply of **only one type** of material
  - The arbiter doesn't smoke
  - The arbiter code is fixed (given)
- System's behaviour:
  - Whenever table empty, the arbiter takes two ingredients from two agents and places the ingredients on the table
  - The smoker with the third ingredient rolls & smokes a cigarette
  - Smokers do not hoard items from the table
  - Each smoker can smoke at most one cigarette at a time
  - The process continues forever

Exercise: Cigarette Smokers

- Use the following semaphores and global variables. The arbiter is composed of three agents.
- Three “pushers” are also defined (one per ingredient). A pusher recognizes the presence of a given ingredient on the table. Then:
  - if another ingredient has been recognized by the relevant pusher, he activates the smoker with the missing ingredient.
  - Otherwise, he simply sets a shared Boolen variable **isX** (isPaper, isTobacco or isMatches) to let the other pushers know that its ingredient is on the table.

```
/* semaphores and shared global variables */

semaphore agentSem = 1 // to activate an arbiter agent (see below)
semaphore tobacco, paper, match = 0 // to signal ingredient on table (or wait for it)
semaphore tobaccoSem, paperSem, matchesSem = 0 // to synchronize pushers
semaphore mutex = 1 // to access critical sections
Boolean isPaper, isTobacco, isMatches = FALSE // is there that ingredient on the table?
```

Exercise: Cigarette Smokers

```
/* arbiter is composed of three concurrent agents */
/* matches agent */          /* tobacco agent */          /* paper agent */
wait(agentSem);              wait(agentSem);              wait(agentSem);
signal(tobacco);             signal(paper);               signal(tobacco);
signal(paper);               signal(matches);             signal(matches);

/* use the functions make_cigarette() and smoke() to represent actions */

/* a possible solution is in the next page */
```

Exercise: Cigarette Smokers

- Two pushers are needed to wake up a smoker.
  - The first pusher that executes the critical section only sets the isX variable
  - The second pusher signals the relevant smoker
- The smoker waits until he is woken up by a pusher: then he makes a cigarette, wakes up a (random) arbiter, and smokes.
  - Notice that smoking comes after waking up an arbiter agent (why?)

```
/* tobacco pusher - the other pushers are similar */
while(TRUE) {
    wait(tobacco);
    wait(mutex);
    if(isPaper)          // is second pusher
        { isPaper = FALSE; signal(matchesSem); }
    else if(isMatches)  // is second pusher
        { isMatches = FALSE; signal(paperSem); }
    else                 // is first pusher
        isTobacco = TRUE;
    signal(mutex);
}
```

```
/* smoker agent with matches - the
other smoker agents are similar */

while(TRUE) {
    wait(matchesSem);
    make_cigarette();
    signal(agentSem);
    smoke();
}
```