Module 4 - Concurrency

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- ***Independent*** process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

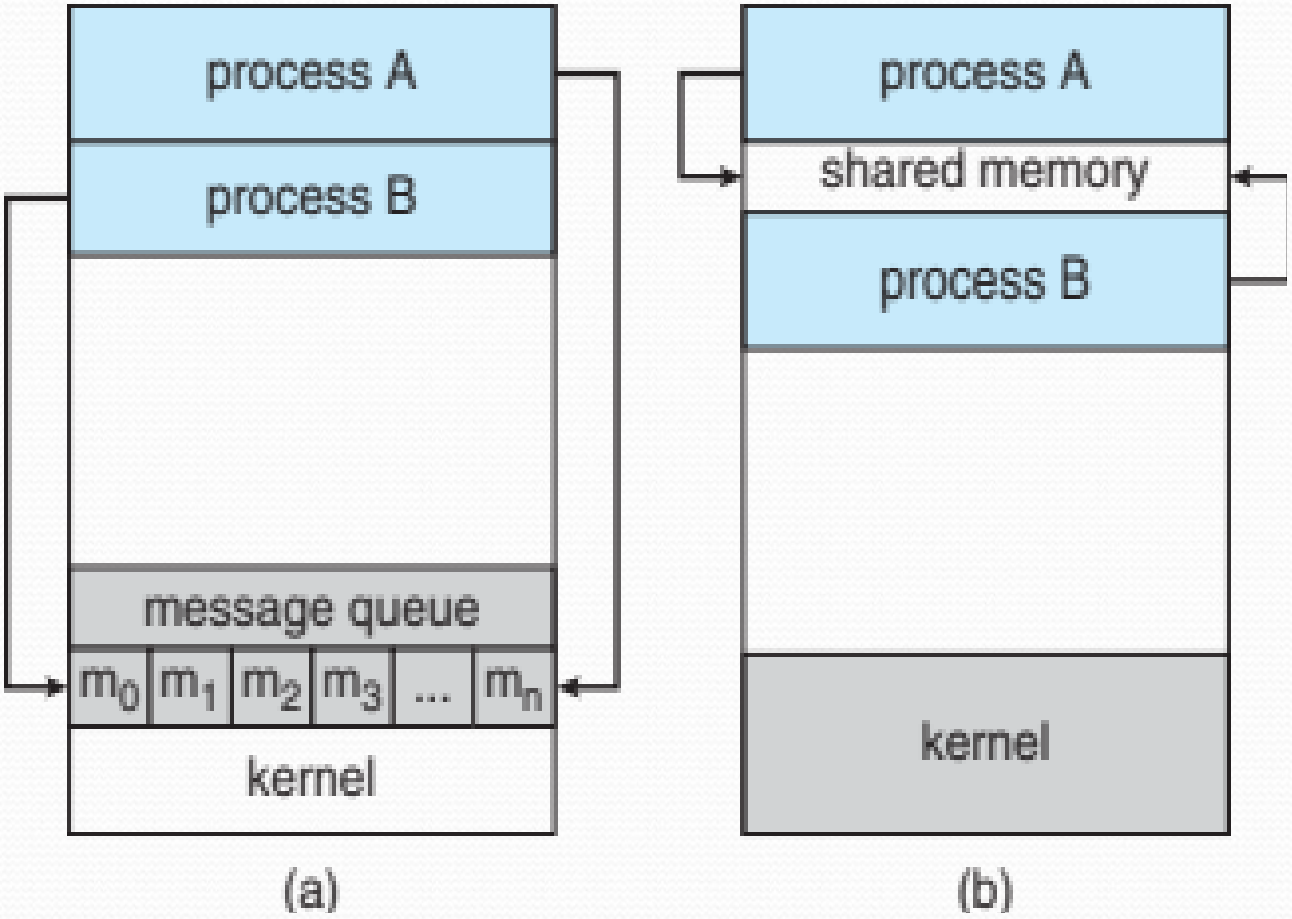
Interprocess Communication

- Cooperating processes need **interprocess communication** (**IPC**)

- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing. (b) shared memory.



Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system
- Major issues - to synchronize the actions of various process when they access shared memory

IPC in Shared memory systems

Producer-Consumer Problem

- *Producer* process produces information that is consumed by a *consumer* process
- Producer and consumer processes uses buffer for communication
  - Producer: fills the items in the buffer
  - Consumer: picks the items from the buffer
- The buffer will reside in the shared region of producer and consumer processes
- Types of buffer
  - **unbounded-buffer -** no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- buffer is implemented as a circular array with two logical pointers: in and out

Bounded-Buffer – Shared-Memory Solution

- in - points to the next free position in the buffer
- out - points to the first full position in the buffer
- The buffer is empty when in == out;
- the buffer is full when ((in + 1) % BUFFER SIZE) == out

- Solution is correct, but can only use BUFFER_SIZE-1 elements

Bounded-Buffer – Producer

```
item next_produced;
while (true) {
    /* produce an item in next produced */
    while (((in + 1) % BUFFER_SIZE) == out);
    /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}
```

Bounded Buffer – Consumer

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;

        /* consume the item in next consumed */
}
```

Interprocess Communication – Message Passing

- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable
- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive

Message Passing (Cont.)

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Direct and Indirect Communication

Direct Communication

- Processes must name each other explicitly:
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox

Indirect Communication

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:

  `send`(*A, message*) – send a message to mailbox A

  `receive`(*A, message*) – receive a message from mailbox A

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message

Buffering

- Queue of messages attached to the link is implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     Sender must wait for receiver
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Need for Process Synchronization

- 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

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
```

Race Condition

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

  S0: producer execute `register1 = counter`

  S1: producer execute `register1 = register1 + 1`
  S2: consumer execute `register2 = counter`
  S3: consumer execute `register2 = register2 - 1`
  S4: producer execute `counter = register1`
  S5: consumer execute `counter = register2`

Critical-Section problem

- The regions of a program that try to access shared resources and may cause race conditions are called critical section.

- To avoid race condition, we need to assure that only one process at a time can execute within the critical section

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 request 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);
```

Critical-Section Problem

Solution to Critical-Section Problem must satisfy the following three requirements:

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

Peterson's Solution

- Software based solution to critical-section problem
- Restricted to 2 processes

- 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 $\mathbf{P_i}$ 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);
```

Peterson's Solution

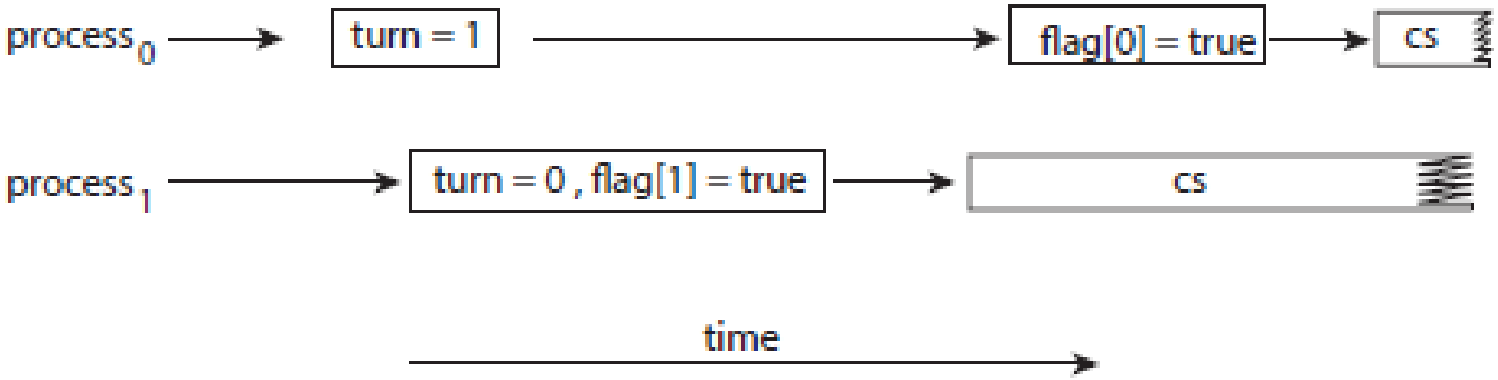


Figure 6.4 The effects of instruction reordering in Peterson's solution.

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

Bakery algorithm

- Can be used for more than 2 processes
- Preserves first come first serve
- Each process is associated with ID
- Before entering its critical section, a process receives a number.
- The holder of the smallest number enters its critical section.
- Tie breaker is done using the process id: if processes i and j hold the same number and i < j then i enters first.

Bakery algorithm

Initial declaration:

boolean choosing[n] = { false, … }; int number[n] = { 0, … } ;

Pseudo code:

```
choosing[i] = true;
number[i] =  max(number[0], number[1], …) + 1;
choosing[i] = false;
for (j = 0; j < n; ++j) {
      while (choosing[j])
                continue;
      while ((number[j] != 0) && ((number[i], i) > (number[j],
j)))
                continue;
}
…critical section…
number[i] = 0;
```

Mutex Locks

- Software tool to solve critical section problem
- Protect a critical section by first **acquire**() a lock then **release**() the lock
  - Boolean variable indicates if lock is available or not
- Calls to **acquire**() and **release**() must be atomic
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

acquire() and release()

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

Solution using Mutex locks

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

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

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

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

```
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
  - Same as a **mutex lock**
- Can solve various synchronization problems

Semaphore Implementation

- 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 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);
    }
}
```

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

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
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

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

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
do {
    wait(full);
    wait(mutex);
       ...
      /* remove an item from buffer */
       ...
    signal(mutex);
    signal(empty);
       ...
      /* consume the item */
       ...
} while (true);
```

Readers-Writers Problem

- A datadase 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

- Shared Data
  - Database
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a writer process

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

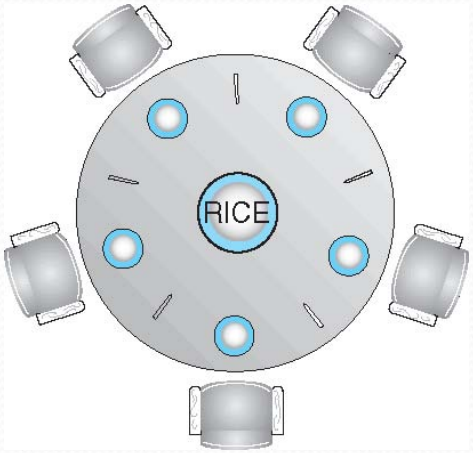
Readers-Writers Problem (Cont.)

- 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);
```

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
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice
    - 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?

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table
  - Allow a philosopher to pick up the forks only if both are available
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.

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 are possible

Monitors

- A high-level abstraction that provides a convenient and effective mechanism for process synchronization
- *Abstract data type* – encapsulates data with a set of functions to operate on that data
- includes a set of programmer-defined operations that are provided with mutual exclusion within the monitor
- Only one process may be active within the monitor at a time

Monitors

monitor *monitor name*

{

/* shared variable declarations */

function P1 (. . .) *{. . .}*

function P2 (. . .) *{. . .}*

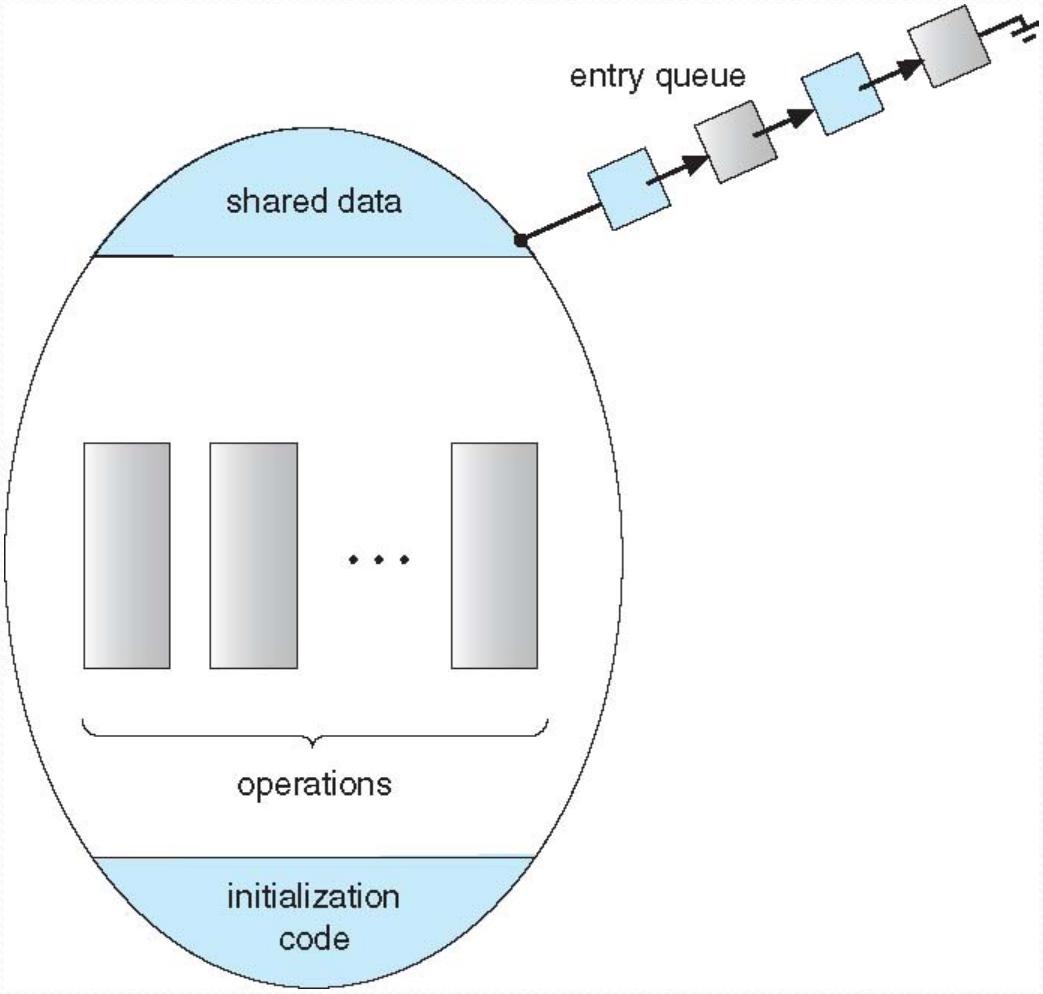
. . .

function Pn (. . .) *{. . .}*

initialization code (. . .) *{. . .}*

}

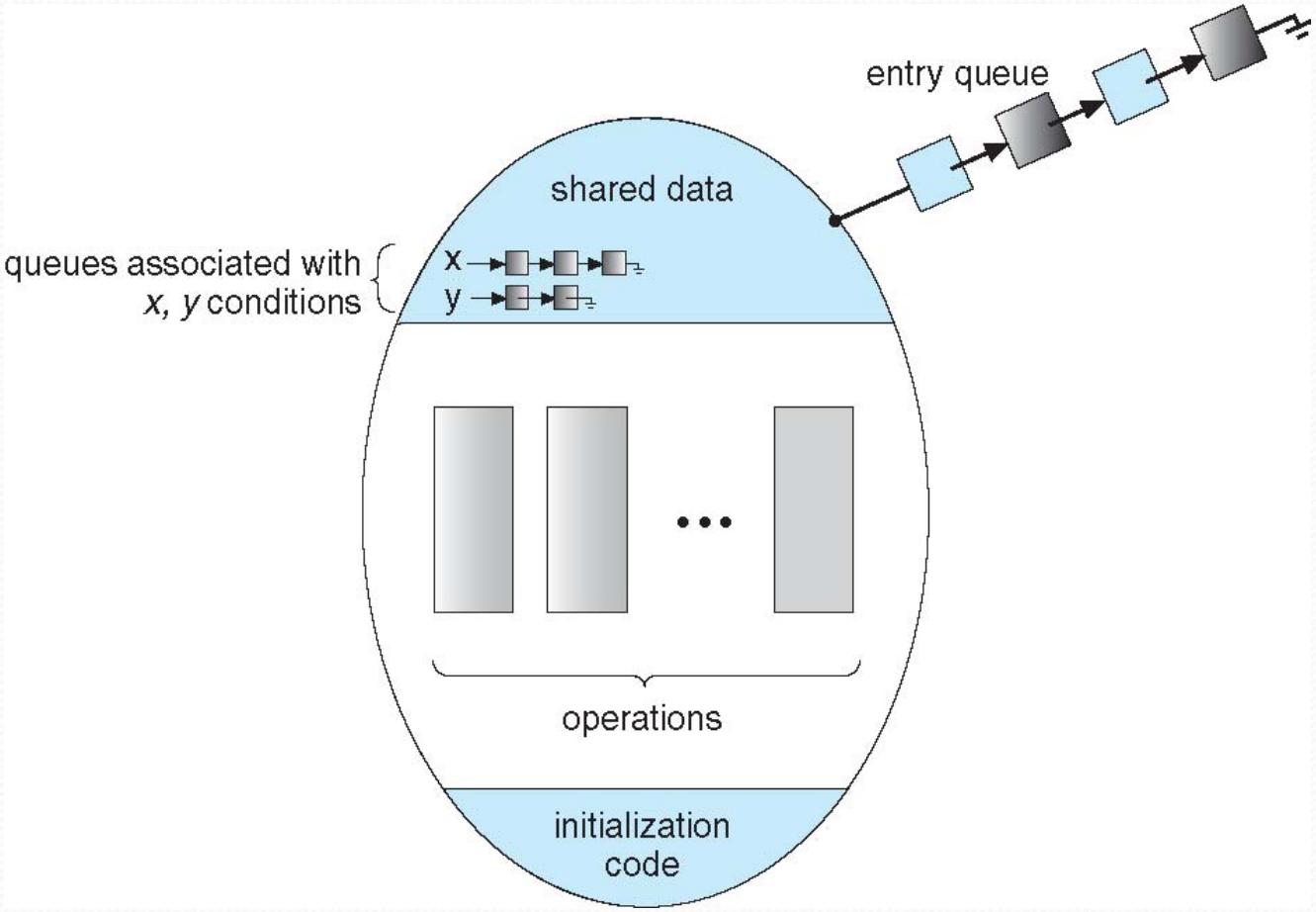
Schematic view of a Monitor



Condition Variables

- **`condition x, y;`**
- Two operations are allowed on a condition variable:
  - **`x.wait()`** – a process that invokes the operation is suspended until another process invokes **`x.signal()`**
  - **`x.signal()`** – resumes exactly one suspended process (if any) that invoked **`x.wait()`**

Monitor with Condition Variables



Monitors contd…

- 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

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

synchronization hardware

- Hardware features can make any programming task easier and improve system efficiency.
- The critical-section problem could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified.

Definition of test and set() instruction

```
boolean test and set(boolean *target) {
        boolean rv = *target;
        *target = true;
        return rv;
}
```

synchronization hardware

Mutual-exclusion implementation with test and set()

```
do {
        while (test and set(&lock)); /* do nothing */
        /* critical section */
        lock = false;
        /* remainder section */
} while (true);
```

synchronization hardware

- The compare and swap() instruction, in contrast to the test and set() instruction, operates on three operands

The definition of the compare and swap() instruction

```
int compare and swap(int *value, int expected, int new value) {
        int temp = *value;
        if (*value == expected)
                *value = new value;
        return temp;
}
```

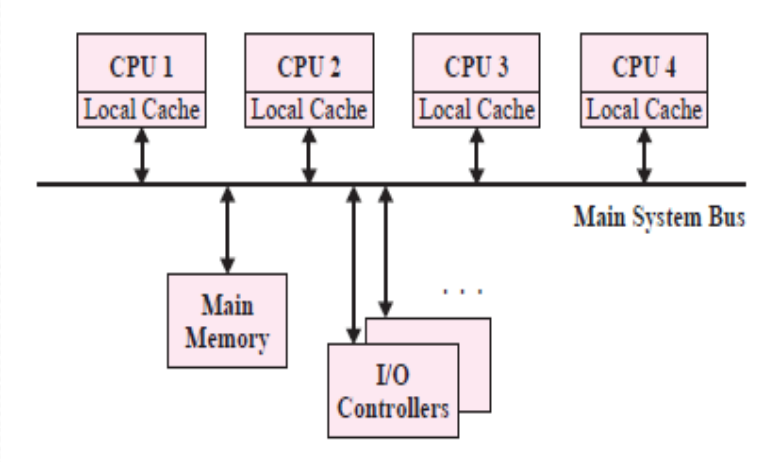
synchronization hardware

Mutual-exclusion implementation with the compare and swap() instruction

```
do {
        while (compare and swap(&lock, 0, 1) != 0)
        ; /* do nothing */
        /* critical section */
        lock = 0;
        /* remainder section */
} while (true);
```

Multiprocessing

- Multiprocessing systems are those that run multiple CPUs in a single system
- Multiprocessing is a less expensive option because a single system can share many expensive hardware components such as power supplies, primary and secondary storage, and the main system bus.



A simplified multiprocessor system architecture

Types of Multiprocessing

- Asymmetric multiprocessing
  - The OS runs on only one designated CPU.
  - The other CPUs run only applications.
  - not commonly used because of performance bottlenecks due to running the OS only on one processor.

- Symmetric multiprocessing (SMP)
  - OS can be running on any CPU
  - A running program obviously will be modifying its state
  - Multiple instances of the OS running on different CPUs must be prevented from changing the same data structure at the same time.

Locking

Why locks?

- In multiprocessing, different tasks run concurrently on different CPUs but concurrency can cause inconsistent results
- To make it consistent, serializability is required
- To ensure serializability, locks are used
- A lock is an object with two operations: acquire(L) and release(L)
- The lock has state: it is either locked or not locked

Drawbacks:

- Locks can ruin performance

Lock free coordination

why do we want to avoid locks?

- Locks limit scalability
- Degrades performance
- Complexity
- Possibility of deadlock
- priority inversion

- A lock-free data structure can be used to improve performance.

- A lock-free data structure increases the amount of time spent in parallel execution rather than serial execution, improving performance on a multi-core processor, because access to the shared data structure does not need to be serialized to stay coherent.

Lock free coordination

- A lock free algorithm protects a shared data structure through a non-blocking algorithm
- A **non-blocking algorithm** ensures that threads competing for a shared resource do not have their execution indefinitely postponed by mutual exclusion
- A non-blocking algorithm is **lock-free** if there is guaranteed system-wide progress and **wait-free** if there is also guaranteed per-thread progress

References

1. Abraham Silberschatz, Peter B. Galvin, Greg Gagne-Operating System Concepts, Wiley (2018).

2. Dhamdhere, Dhananjay M. Operating systems: a concept-based approach, 2E. Tata McGraw-Hill Education, 2006.

3. Ramez Elmasri, A.Gil Carrick, David Levine, Operating Systems, A Spiral Approach - McGrawHill Higher Education (2010).

4. https://pdos.csail.mit.edu/6.828/2012/lec/l-lockfree.txt

5. https://en.wikipedia.org/wiki/Non-blocking_algorithm

Any queries?