CONCURRENCY:

MUTUAL EXCLUSION & SYNCHRONIZATION

USAMA BIN AMIR

CHAPTER 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 introduce the concept of an atomic transaction and describe mechanisms to ensure atomicity.

CONCURRENCY

Concurrency encompasses design issues including:

- Communication among processes
- Sharing resources
- Synchronization of multiple processes
- Allocation of processor time

KEY TERMS RELATED TO CONCURRENCY

- Critical section
  - A section of code within a process that requires access to shared resources and which may not be executed while another process is in a corresponding section of code.

- Mutual exclusion
  - The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.

KEY TERMS RELATED TO CONCURRENCY

- Deadlock
  - A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.

- Livelock
  - A situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work.

KEY TERMS RELATED TO CONCURRENCY

- Race condition
  - A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.

- Starvation
  - A situation in which a runnable process is overlooked indefinitely by the scheduler; althoug it is able to proceed, it is never chosen.

DIFFICULTIES OF CONCURRENCY

- Sharing of global resources
  - If two processes share the same global var, and both perform reads & writes, then the order of reads and writes are important/critical
- Operating system managing the allocation of resources optimally
  - Assume P1 requests and gets an I/O channel, but was suspended before using it. OS has locked the channel, and no other process can use it.
- Difficult to locate programming errors
  - The result is not deterministic, the error/result may come from different processes.
  - E.g. Race condition

A SIMPLE EXAMPLE

```
void echo()
{
  chin = getchar(); //input from keyboard
  chout = chin;
  putchar(chout); //display on screen
}
```

Multiprogramming Single Processor

2 processes P1 and P2 call procedure echo

- Both use the same keyboard and screen
- Since both processes use the same procedure, proc echo is shared – to save memory

A SIMPLE EXAMPLE

Single processor

Process P1

```
  .
chin = getchar();
// interrupted
  .
  .
  .
chout = chin;
putchar(chout);
  .
```

Input: x

Process P2

```
      .
      .
      .
chin = getchar();
chout = chin;
putchar(chout);
      .
```

Input: y

NO CONCURRENCY CONTROL

- P1 invokes echo, and is interrupted immediately after reading x
  - char x is stored in chin
- P2 is activated and invokes echo, and runs to conclusion
  - reads y, and displays y
- P1 is then resumed.
  - Value x has been overwritten by y (from P2’s execution)
  - chin contains y $\rightarrow$ y is displayed, instead of x

WITH CONCURRENCY CONTROL

- Only one process can use codes at a time.
- P1 invokes echo, and is interrupted immediately after reading x → char x is stored in chin
- P2 is activated and invokes echo, but P1 is still inside (using) echo (though is currently suspended)
  - P2 is blocked from entering echo, i.e. suspended until echo is available
- Later, P1 is resumed and completes echo
  - char x is displayed (as intended -- correctly)
- When P1 exits echo, P2 can now resume echo
  - reads y, and displays y (correctly)

CONCURRENCY IN SINGLE- AND MULTIPROCESSORS

- Concurrency can occur even in single processor systems.
- In multiprocessors, even if both processes
  - use separate processors,
  - use different keyboards and screen,

  they use the same procedure (shared memory, i.e. shared global var)
  - Therefore, concurrency may occur in multiprocessors systems too.

A SIMPLE EXAMPLE

Multiprocessors

Process P1

```
  .
chin = getchar();
  .
chout = chin;
putchar(chout);
  .
  .
```

Input: x

Process P2

```
     .
     .
chin = getchar();
chout = chin;
     .
putchar(chout);
     .
```

Input: y

RACE CONDITION

- When multiple processes/threads read and write shared data
- E.g: Consider a shared variable a.

```
P1:                  P2:
a = 1;               a = 2;
write a;             write a;
```

- Race condition to write a.
- 'Loser' wins.

RACE CONDITION

- Example 2:
  Consider a shared variables b and c.
  Initially, b = 1, c = 2

P3:	P4:
b = b + c;	c = b + c;

- If P3 executes first => b = 3, c = 5
- If P4 executes first => b = 4, c = 3

OPERATING SYSTEM CONCERNS

- Keep track of various processes
- Allocate and deallocate resources
  - Processor time
  - Memory
  - Files
  - I/O devices
- Protect data and resources
- Output of process must be independent of the speed of execution of other concurrent processes

PROCESS INTERACTION

- Processes unaware of each other
  - Processes not intended to work together
- Processes indirectly aware of each other
  - Processes don't know each other by their id, but share access to some objects, e.g. I/O buffer
- Process directly aware of each other
  - Processes communicate by their id, and designed to work jointly

Table 5.2 Process Interaction

Degree of Awareness	Relationship	Influence that one Process has on the Other	Potential Control Problems
Processes unaware of each other	Competition	•Results of one process independent of the action of others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Mutual exclusion •Deadlock (renewable resource) •Starvation •Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	•Results of one process may depend on information obtained from others •Timing of process may be affected	•Deadlock (consumable resource) •Starvation

COMPETITION AMONG PROCESSES FOR RESOURCES

- Mutual Exclusion
  - Critical sections
    - Only one program at a time is allowed in its critical section
    - Example only one process at a time is allowed to send command to the printer
- Deadlock
- Starvation

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

REQUIREMENTS FOR MUTUAL EXCLUSION

- Only one process at a time is allowed in the critical section for a resource
- A process that halts in its noncritical section must do so without interfering with other processes
- No deadlock or starvation

REQUIREMENTS FOR MUTUAL EXCLUSION

- A process must not be delayed access to a critical section when there is no other process using it
- No assumptions are made about relative process speeds or number of processes
- A process remains inside its critical section for a finite time only

MUTUAL EXCLUSION: HARDWARE SUPPORT

Hardware approaches to enforce mutual exclusion:

- Interrupt Disabling
- Special machine instruction
  - Test and set instruction
  - Exchange instruction

MUTUAL EXCLUSION: HARDWARE SUPPORT

- Interrupt Disabling
  - A process runs until it invokes an operating system service or until it is interrupted
  - Disabling interrupts guarantees mutual exclusion - prevents a process from being interrupted

```
while (true) {
        // disable interrupts
        // critical section
        // enable interrupts
        // remainder (continue)
}
```

MUTUAL EXCLUSION: HARDWARE SUPPORT

- Interrupt Disabling
  - Processor is limited in its ability to interleave programs
  - Multiprocessing
    - disabling interrupts on one processor will not guarantee mutual exclusion

MUTUAL EXCLUSION: HARDWARE SUPPORT

- Special Machine Instructions
  - Performed in a single instruction cycle (atomically)
  - Access to the memory location is blocked for any other instructions

1. Test and Set
2. Exchange

TEST AND SET INSTRUCTION

```
boolean testset (int i){
  if (i == 0) {
      i = 1;
      return true;
  }
  else {
      return false;
  }
}
```

TEST AND SET INSTRUCTION

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true)
   {
      while (!testset (bolt))
           /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));

}
```

```
boolean testset (int i){
  if (i == 0) {
       i = 1;
       return true;
  }
  else {
       return false;
  }
}
```

TEST AND SET INSTRUCTION

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true)
   {
      while (!testset (bolt))
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));

}
```

Shared var, init to 0

```
boolean testset (int i){
  if (i == 0) {
       i = 1;
       return true;
  }
  else {
       return false;
  }
}
```

The only process that can enter its critical section is the one that finds `bolt==0`

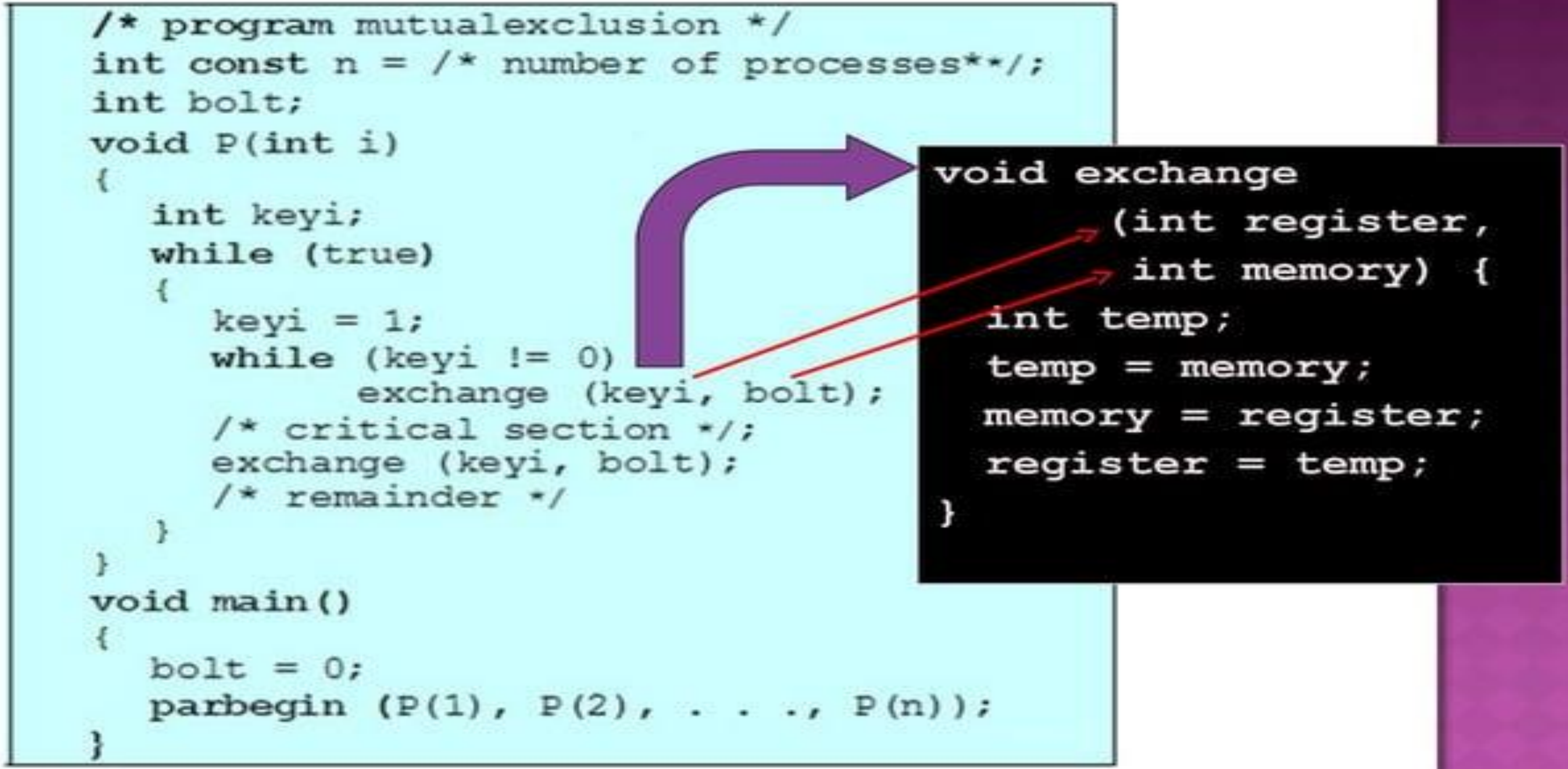
BUSY WAITING

- Busy waiting, or spin waiting
- Refers to a technique in which a process can do nothing until it gets permission to enter its critical section, but continues to execute an instruction or set of instructions that tests the appropriate variable to gain entrance.
- When a process leaves its critical section, it resets `bolt` to 0; at this point one and only one of the waiting processes is granted access to its critical section - the choice of process depends on which process happens to execute the `testset` instruction next.

EXCHANGE INSTRUCTION

```
void exchange(int register,  int
memory) {
    int temp;
    temp = memory;
    memory = register;
    register = temp;
}
```

EXCHANGE INSTRUCTION



```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi;
   while (true)
   {
      keyi = 1;
      while (keyi != 0)
             exchange (keyi, bolt);
      /* critical section */;
      exchange (keyi, bolt);
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

```
void exchange
      (int register,
       int memory) {
  int temp;
  temp = memory;
  memory = register;
  register = temp;
}
```

EXCHANGE INSTRUCTION

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi;
   while (true)
   {
      keyi = 1;
      while (keyi != 0)
            exchange (keyi, bolt);
      /* critical section */;
      exchange (keyi, bolt);
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

Local var, used by each process, init to 1

Shared var, init to 0

```
void exchange
       (int register,
        int memory) {
  int temp;
  temp = memory;
  memory = register;
  register = temp;
}
```

The only process that can enter its critical section is the one that finds **`bolt`==0**

MUTUAL EXCLUSION: MACHINE INSTRUCTIONS

- Advantages
  - Applicable to any number of processes on either a single processor or multiple processors sharing main memory
  - It is simple and therefore easy to verify
  - It can be used to support multiple critical sections

MUTUAL EXCLUSION: MACHINE INSTRUCTIONS

- Disadvantages
  - Busy-waiting consumes processor time
  - Starvation is possible when a process leaves a critical section and more than one process is waiting.
  - Deadlock
    - If a low priority process has the critical region and a higher priority process needs, the higher priority process will obtain the processor to wait for the critical region

MUTUAL EXCLUSION

1. Hardware support
   - Interrupt disabling
   - Special machine instructions
1. Software support
   - OS and programming languages mechanisms

SEMAPHORES

- Special variable called a semaphore is used for signaling
- Use – to protect particular critical region or other resources.
- If a process is waiting for a signal, it is suspended until that signal is sent

SEMAPHORES

- Semaphore is a variable that has an integer value.
- Three operations that can be performed on a semaphore:
  - May be initialized to a nonnegative number
  - Wait() operation decrements the semaphore value
  - Signal() operation increments semaphore value
- Counting semaphore – any integer value
- Binary semaphore – integer 0, 1;
  - can be simpler to implement
  - Also known as mutex locks
- Can implement a counting semaphore S as a binary semaphore

SEMAPHORES

- (If a process is waiting for a signal, it is suspended until that signal is sent)
- When a process wants to use resource it waits on the semaphores
  - If no other process currently using the resource, **wait** (or **acquire**) **call** sets semaphores to in-use and immediately returns to the process. (process has exclusive access to the resource)
  - If some other process is using the resource, semaphores block the current process by moving it to block queue.

    When process that currently holds resource release the resource, **signal** (or **release**) **operation** removes the first waiting block queue to ready queue.

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.

SEMAPHORE IMPLEMENTATION

- Semaphore requires busy waiting
  - wastes CPU cycles
  - This semaphore is also called **spinlock**, because the process “spins” while waiting for the lock.
  - (Advantage of spinlock - no context switch is required when a process must wait on a lock -useful when locks are held for short times).
- Semaphore can be modified to eliminate busy waiting
  - Not waste (as much) CPU time on processes that are waiting for some resource.
  - The ‘waiting’ process can *block* itself - to be placed in a block queue for the semaphore.

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(P) - suspends the process P that invokes it
    - place the process invoking the operation on the appropriate waiting queue.
  - wakeup(P) - resumes the execution of a blocked process P
    - remove one of processes in the waiting queue and place it in the ready queue.

SEMAPHORE PRIMITIVES

```
struct semaphore {
     int count;
     queueType queue;
}

void semWait(semaphore s)
{
     s.count--;
     if (s.count < 0)
     {
          place this process in s.queue;
          block this process
     }
}
void semSignal(semaphore s)
{
     s.count++;
     if (s.count <= 0)
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.3 A Definition of Semaphore Primitives

BINARY SEMAPHORE PRIMITIVES

```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};

void semWaitB(binary_semaphore s)
{
     if (s.value ==  1)
          s.value = 0;
     else
          {
               place this process in s.queue;
               block this process;
          }
}
void semSignalB(semaphore s)
{
     if (s.queue.is_empty())
          s.value = 1;
     else
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

Figure 5.4 A Definition of Binary Semaphore Primitives

MUTUAL EXCLUSION USING SEMAPHORES

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true)
     {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.6 Mutual Exclusion Using Semaphores

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)

MONITORS

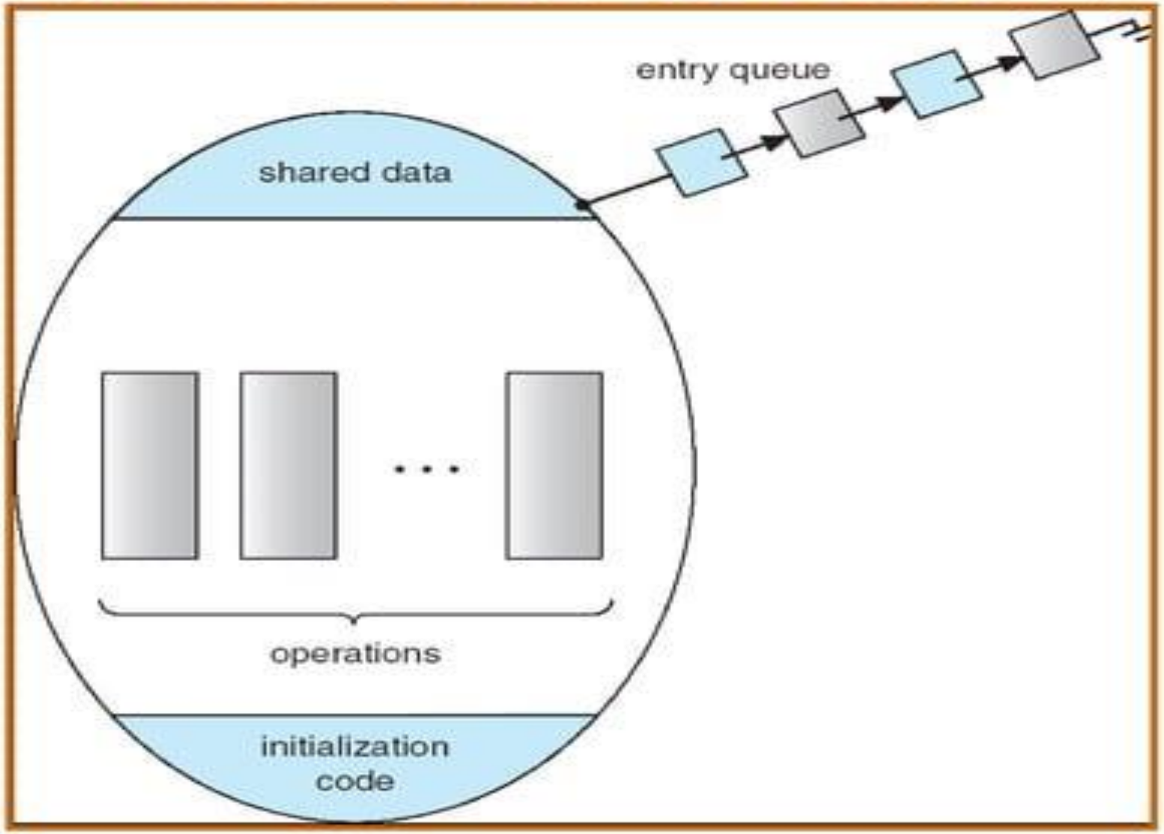
- Monitor is a software module
- Chief characteristics
  - Local data variables are accessible only by the monitor
  - Process enters monitor by invoking one of its procedures
  - Only one process may be executing in the monitor at a time - ME
    - Data variables in the monitor will also be accessible by only one process at a time, shared variable can be protected by placing them in the monitor
- Synchronization is also provided in the monitor for the concurrent processing through cwait(c) and csignal(c) operation

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 a time

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

SCHEMATIC VIEW OF A MONITOR

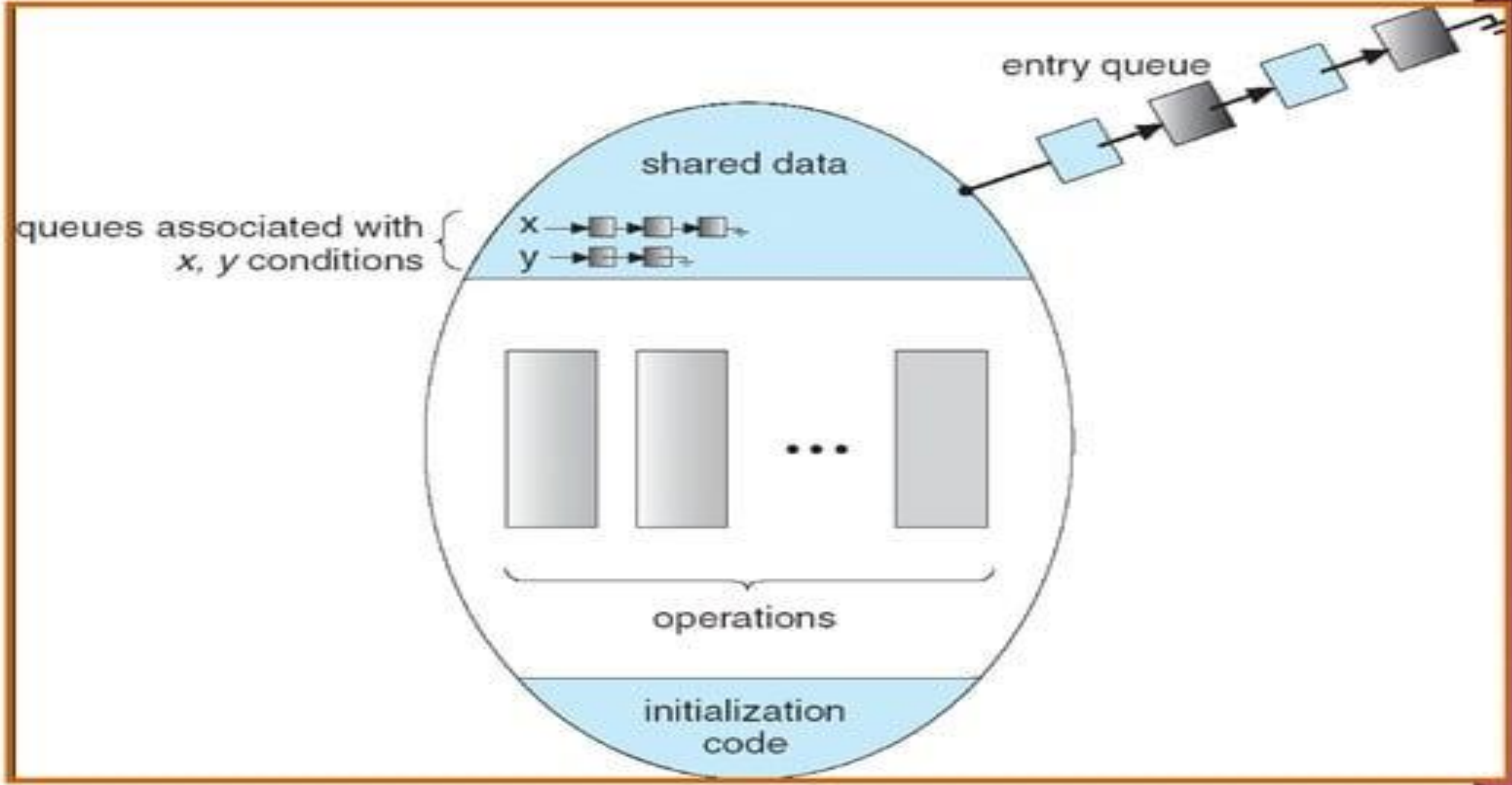


CONDITION VARIABLES

- condition x, y;

- Two operations on a condition variable:
  - x.wait () – a process that invokes the operation is suspended.
  - x.signal () – resumes one of processes (if any) that invoked x.wait ()

MONITOR WITH CONDITION VARIABLES



CLASSICAL PROBLEMS OF SYNCHRONIZATION

- 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

- Producer process

```
do {
   //   produce an item
   wait (empty);
    wait (mutex);

    //  add the item to the
   buffer

    signal (mutex);
    signal (full);
} while (true);
```

- Consumer process

```
do {
     wait (full);
     wait (mutex);

      //  remove an item from  buffer

      signal (mutex);
      signal (empty);

       //  consume the removed item

  } 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.
- Shared Data
  - Data set
  - Semaphore mutex initialized to 1.
  - Semaphore wrt initialized to 1.
  - Integer readcount initialized to 0.

READERS-WRITERS PROBLEM

- Writer process

```
do {
   wait (wrt) ;

   //    writing is
   performed

   signal (wrt) ;
} while (true)
```

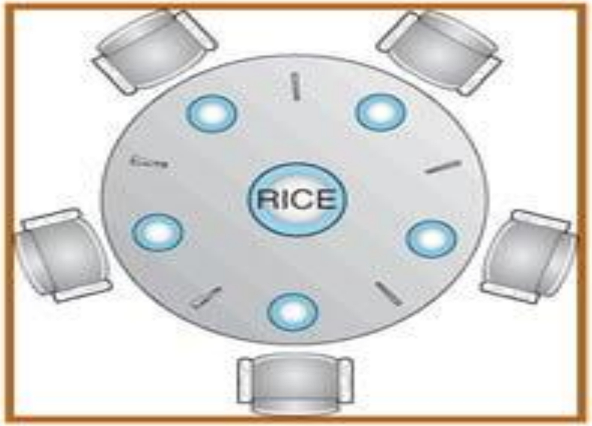
- Reader process

```
do {
     wait (mutex) ;
     readcount ++ ;
     if (readcount == 1)
        wait (wrt) ;
     signal (mutex)

        // reading is performed

     wait (mutex) ;
     readcount  - - ;
     if (readcount  == 0)
        signal (wrt);
     signal (mutex) ;
 } while (true)
```

DINING-PHILOSOPHERS PROBLEM



- 5 philosophers thinking and eating
- When eating - not thinking
  - Use 2 chopsticks
  - Will not put down chopsticks until finishes eating
  - Can't eat if neighbour is using chopstick
- When thinking – not eating
  - Puts down chopsticks

- Shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

DINING-PHILOSOPHERS PROBLEM

- 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) ;
```

DINING-PHILOSOPHERS PROBLEM

- Possible deadlock: if all philosophers become hungry at the same time, and picks up their chopsticks, but have to wait for another.
- Possible solutions:
  - Allow at most 4 philosophers to sit simultaneously.
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available - must do this in critical section.
  - Use an asymmetric solution
    - an odd philosopher picks up her first left chopstick and then her right chopstick.
    - an even philosopher picks up her first right chopstick and then her left chopstick.
- Must also guard against starvation.

SOLUTION TO DINING PHILOSOPHERS

```
monitor DP
  {
   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 + 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;
 }
}
```

BARBER SHOP PROBLEM

A classical problem similar to a real operating system for the access of different resources.

Problem definition:

- Three chairs.
- Three barbers.
- Waiting area having capacity for four seats and additional space for customers to stand and wait.
- Fire code limits the total number of customers (seating +standing).
- Customer cannot enter if pack to capacity.
- When a barber is free, customer waiting longest on the sofa can be served.
- If there are any standing customer, he can now occupy the seat on the sofa.
- Once a barber finished haircut, any barber can take the payment.
- As there is only one casher, so payment from one customer is accepted at a time.
- Barber(s) time is divided into haircut, accepting payment and waiting for customers (sleeping).

MESSAGE PASSING

- Enforce mutual exclusion
- Exchange information

```
send (destination, message)
receive (source, message)
```

SYNCHRONIZATION

- Sender and receiver may or may not be blocking (waiting for message)
- Blocking send, blocking receive
  - Both sender and receiver are blocked until message is delivered
  - Called a rendezvous

SYNCHRONIZATION

- Nonblocking send, blocking receive
  - Sender continues on
  - Receiver is blocked until the requested message arrives
- Nonblocking send, nonblocking receive
  - Neither party is required to wait

ADDRESSING

- Direct addressing
  - Send primitive includes a specific identifier of the destination process
  - Receive primitive could know ahead of time which process a message is expected
  - Receive primitive could use source parameter to return a value when the receive operation has been performed

ADDRESSING

- Indirect addressing
  - Messages are sent to a shared data structure consisting of queues
  - Queues are called mailboxes
  - One process sends a message to the mailbox and the other process picks up the message from the mailbo

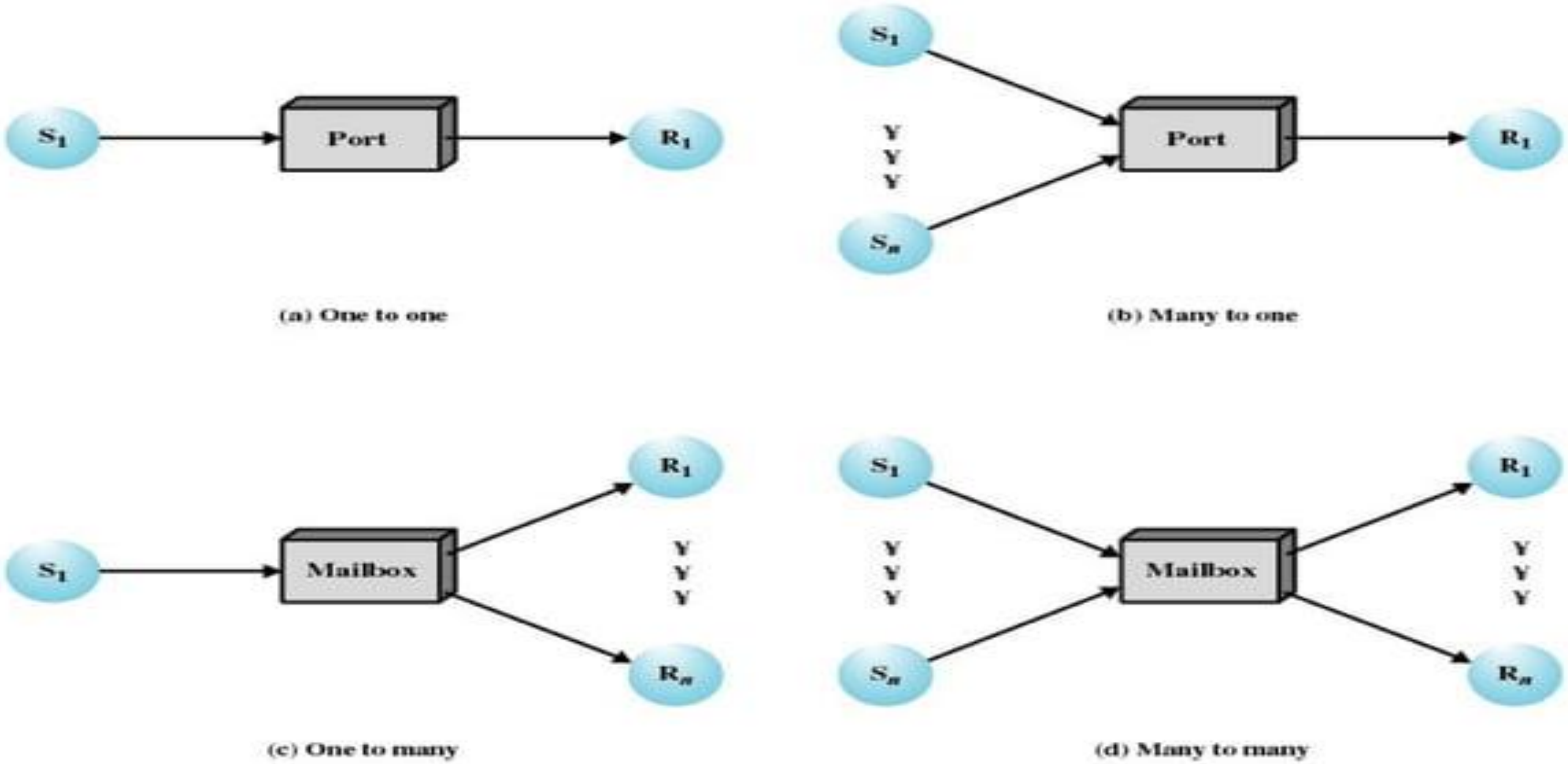


Figure 5.18 Indirect Process Communication

MESSAGE FORMAT

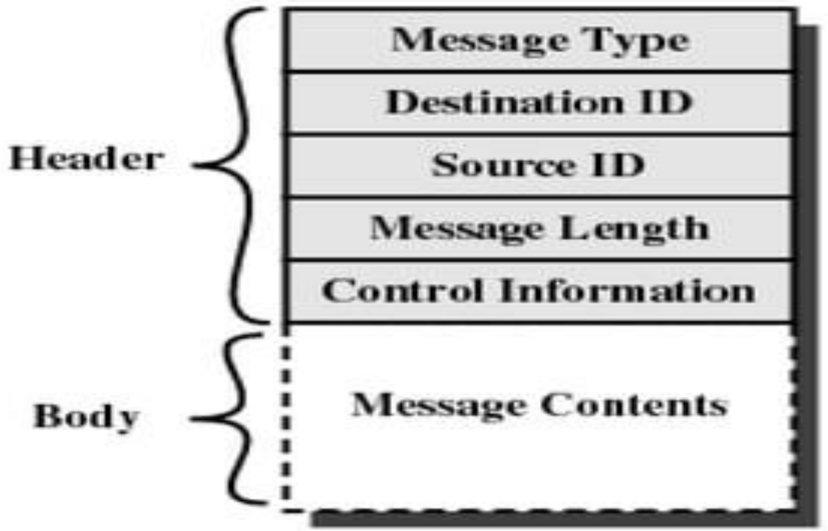


Figure 5.19 General Message Format

MUTUAL EXCLUSION: OS AND PROGRAMMING LANGUAGES MECHANISMS

- Semaphore
  - An integer value used for signaling among processes. Only three operations may be performed on a semaphore, all of which are atomic: initialize, decrement, and increment. The decrement operation may result in the blocking of a process, and the increment operation may result in the unblocking of a process. Also known as a counting semaphore or a general semaphore.
- Binary semaphore
  - A semaphore that takes on only the values 0 and 1.
- Mutex
  - Similar to a binary semaphore. A key difference between the two is that the process that locks the mutex (sets the value to 0) must be the one to unlock it (sets the value to 1).

MUTUAL EXCLUSION: OS AND PROGRAMMING LANGUAGES MECHANISMS

- Condition variable
  - A data type that is used to block a process or thread until a particular condition is true.
- Monitor
  - A programming language construct that encapsulates variables, access procedures and initialization code within a ADT (abstract data type). The monitor’s variable may only b accessed via its access procedures and only one process may be actively accessing the monitor at any one time.the acces procedures are *critical sections*. A monitor may have a queu of processes that are waiting to access it.

MUTUAL EXCLUSION: OS AND PROGRAMMING LANGUAGES MECHANISMS

- Event flags
  - A memory word used as a synchronization mechanism. Application code may associate a different event with each bit in a flag. A thread can wait for either a single event or a combination of events by checking one or multiple bits in the corresponding flag. The thread is blocked until all of the required bits are set (AND) or until at least one of the bits is set (OR).
- Mailboxes/Messages
  - A means for two processes to exchange information and that may be used for synchronization.
- Spinlocks
  - Mutual exclusion mechanism in which a process executes in an infinite loop waiting for the value of a lock variable to indicate availability.

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
- Uses condition variables and 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

WINDOWS XP SYNCHRONIZATION

- Uses interrupt masks to protect access to global resources on uniprocessor systems
- Uses spinlocks on multiprocessor systems
- Also provides dispatcher objects which may act as either mutexes and semaphores
- Dispatcher objects may also provide events
  - An event acts much like a condition variable

LINUX SYNCHRONIZATION

- Linux:
  - disables interrupts to implement short critical sections

- Linux provides:
  - semaphores
  - spin locks

PTHREADS SYNCHRONIZATION

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