Concurrency: Mutual Exclusion and Synchronization

Chapter 5

Concurrency

Concurrency arises in 3 different contexts:

- Multiple applications
  - Multiprogramming: time slicing
- Structured applications
  - Develop a single application as set of concurrent processes
- Operating system structure
  - Often implemented as set of processes or thereads

Concurrency: Related Terms

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.
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.
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.
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; although it is able to proceed, it is never chosen.

Difficulties with Concurrency

- Sharing of global resources
  - Two processes reading from and writing to the same global variable… sequence of R/W is crucial
- Operating system managing the allocation of resources optimally
  - Process A acquires resource R and blocks, Process B wants resource R
- Difficult to locate programming errors
  - Non-deterministic behavior

Currency: Design Issues

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

A Simple Example

Global Char: chin, chout

Process P1

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

Process P2

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

"chin" in P1 is lost

Another Simple Example

```
Global Char: b = 1, c = 2;

  Process P1                    Process P2

.                              .
b = b + c                      c = b + c
.                              .
```

P1 then P2	=>	b = 3, c = 5
P2 then P1	=>	b = 4, c = 3

Race Condition

Operating System Concerns

- Keeping track of multiple and distinct 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
  - Deterministic

Process Interaction

Given concurrency, how can processes interact with each other?

- Processes *unaware* of each other
  - Independent processes not intended to work together
  - Compete for resources
- Processes *indirectly aware* of each other
  - Share access to resources
  - Sharing is cooperative
- Process *directly aware* of each other
  - Designed to work jointly on some activity
  - Sharing is cooperative

Resource Sharing Among Concurrent Processes

- Mutual Exclusion
  - Critical sections: used when accessing shared resource
    - Only one program at a time is allowed in its critical section
    - Example: one process at a time allowed to send command to printer
- Deadlock
  - No computational progress can be made because a set of processes are blocked waiting on processes that will never be available
- Starvation
  - A process' resource request is never accommodated

Critical Section Problem (Revisited)

```
shared float balance;
```

```
/* Code schema for p1 */
..
balance = balance + amount;
..

   /* Schema for p1 */
   /*  X == balance */
      load R1, X
      load R2, Y
      add R1, R2
      store R1, X
```

```
/* Code schema for p2 */
..
balance = balance - amount;
..

   /* Schema for p2 */
   /*  X == balance */
      load R1, X
      load R2, Y
      sub R1, R2
      store R1, X
```

Critical Section Problem…

```
/* Schema for p1 */              /* Schema for p2 */
  ┌ load R1, X  ┐                   ┌ load R1, X  ┐
  │ load R2, Y  ┘ 1               4 └ load R2, Y  │
5 │ add R1, R2  ┐                   ┌ sub R1, R2  ├ 2
  └ store R1, X ┘ 3               6 └ store R1, X ┘
```

- Suppose:
  - Execution sequence : 1, 2, 3
    - Lost update : 2
  - Execution sequence : 1, 4, 3 ,6
    - Lost update : 3
- Together => non-determinacy
- Race condition exists

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 when accessing a critical section if 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 & Synchronization

Hardware Support

Interrupt
Test & Set
Exchange

Mutual Exclusion: Hardware Support

Interrupt Disabling

```
While (true)
{
  disable-interrupts
      critical section
  enable-interrupts
}
```

– Processor is limited in its ability to interleave programs

– Disabling interrupts guarantees mutual exclusion

– Multiprocessor Environment
  - disabling interrupts on one processor will not guarantee mutual exclusion

Critical Section Problem

```
shared float balance;
```

```
/* Code schema for p1 */
..
disable-interrupts;
  balance = balance + amount;
enable-interrupts;
..
```

```
/* Code schema for p2 */
..
disable-interrupts;
  balance = balance - amount;
enable-interrupts
..
```

```
/* Schema for p1 */
Interrupts turned off
   load R1, X
   load R2, Y
   add R1, R2
   store R1, X
Interrupts turned on
```

uninterruptible

```
/* Schema for p2 */
Interrupts turned off
   load R1, X
   load R2, Y
   sub R1, R2
   store R1, X
Interrupts turned on
```

Mutual Exclusion: Hardware Support

- Special Machine Instructions
  – Performed in a single instruction cycle
  – Performs memory access / manipulation
  – No concurrent access to that memory location

- Instructions
  – Test & Set
  – Exchange

The "Test & Set" Instruction

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

EXECUTED ATOMICALLY

The “Test & 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));

}
```

The “Exchange” Instruction

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

EXECUTED ATOMICALLY

The “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));
}
```

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
    - Different variable set for each CR

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 it, the higher priority process will obtain the processor to wait for the critical region

Mutual Exclusion & Synchronization

Language / OS Defined

The Semaphore

Semaphore

- Dijkstra, 1965
- Synchronization primitive with no busy waiting
- It is an integer variable changed or tested by one of the two indivisible operations
- Actually implemented as a protected variable type

```
var x : semaphore
```

Semaphore operations

- **`semWait(S)`** operation ("wait")
  - Requests permission to use a critical resource

```
S := S - 1;
if (S < 0) then
     put calling process on queue
```

- **`semSignal(S)`** operation ("signal")
  - Releases the critical resource

```
S := S + 1;
if (S <= 0) then
     remove one process from queue
```

- Queues are associated with each semaphore variable

Semaphore : Example

```
Critical resource   T
Semaphore           S ← initial_value
Processes           A,B
```

Process A

```
.
semWait(S);
<CS> /* access T */
semSignal(S);
.
```

Process B

```
.
semWait(S);
<CS> /* access T */
semSignal(S);
.
```

Semaphore : Example…

```
var S : semaphore ← 1
```

Queue associated with S

Value of S : 1

Process A	Process B	Process C

semWait(S);
<**CS**>
semSignal(S);
	semWait(S);
	<**CS**>
	semSignal(S);
		semWait(S);
		<**CS**>
		semSignal(S);

Types of Semaphores

- Binary Semaphores
  - Maximum value is 1
- Counting Semaphores
  - Maximum value is greater than 1
- Both use similar semWait and semSignal definitions
- Synchronizing code and initialization determines what values are needed, and therefore, what kind of semaphore will be used

The remaining discussion will focus primarily on **counting semaphores**

Using Semaphores

```
Shared semaphore mutex <= 1;
```

```
proc_1() {
  while(true) {
     <compute section>;
     semWait(mutex);
       <critical section>;
     semSignal(mutex);
     }
   }
```

```
proc_2() {
  while(true) {
     <compute section>;
     semWait(mutex);
       <critical section>;
     semSignal(mutex);
     }
   }
```

---

(1) P1 => semWait(mutex)
Decrements; <0 ?; NO (0);
P1 Enters CS;
P1 interrupted

(2) P2 => semWait(mutex)
Decrements; <0 ?; YES (-1)
P2 **blocks** on **mutex**

Non-Interruptible "Test & Sets"

(3) P1 finishes CS work
P1 => semSignal (mutex);
Increments; <=0 ?; YES (0)
P2 woken & proceeds

Using Semaphores - Example 1

```
Shared semaphore mutex <= 1;
```

```
proc_0() {
...
semWait(mutex);
balance = balance + amount;
semSignal(mutex);
...
}
```

```
proc_1() {
…
semWait(mutex);
balance = balance - amount;
semSignal(mutex);
...
}
```

Suppose P1 issues semWait(mutex) first
……
Suppose P2 issues semWait(mutex) first
……

} No Problem

Note: Could use Interrupts to implement solution,

But (1) with interrupts masked off, what happens if a prior I/O request is satisfied

(2) Interrupt approach would not work on Multiprocessor

Using Semaphores – Example 2

Shared semaphore: s1 <= 0, s2 <= 0;

Note: values started at 0... ok?

```
proc_A() {
  while(true) {
    <compute A1>;
    write(x);
    semSignal(s1);
    <compute A2>;
    SemWait(s2);
    read(y);
  }
}
```

A signals B that "write to x" has completed

A blocks until B signals

```
proc_B() {
  while(true) {
    semWait(s1);
    read(x);
    <compute B1>;
    write(y);
    semSignal(s2);
    <compute B2>;
  }
}
```

B blocks till A signals

B signals A that "write to y" has completed

- Cannot use Interrupt disable/enable here because we have *multiple distinct synchronization points*
- Interrupt disable/enable can only distinguish 1 synchronization event
- **Therefore, 2 Semaphores**

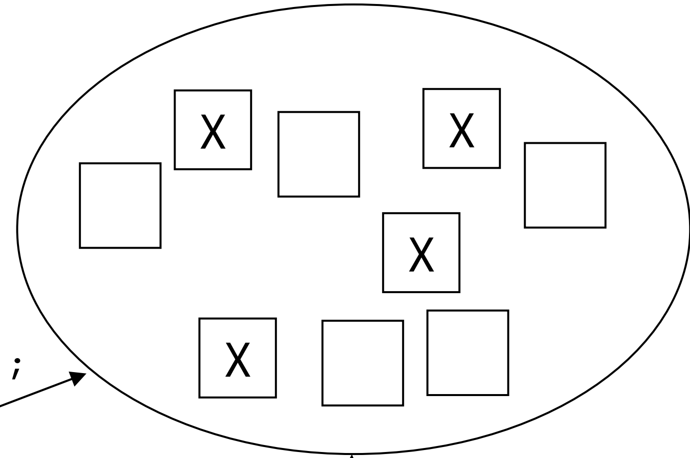
Producer / Consumer Problem (Classic)

- Critical resource
  - Set of message buffers
- 2 Processes
  - Producer : Creates a message and places it in the buffer
  - Consumer : Reads a message and deletes it from the buffer
- Objective
  - Allow the producer and consumer to run concurrently

P/C…

- Constraints
  - Producer must have a non-full buffer to put its message into
  - Consumer must have a non-empty buffer to read
  - Mutually exclusive access to Buffer pool
- Unbounded Buffer problem
  - Infinite buffers
  - Producer never has to wait
  - Not interesting nor practical
- Bounded Buffer Problem
  - Limited set of buffers

P/C - Solution




```
Shared Full: semaphore ← 0;
       Empty semaphore ← MaxBuffers;
       MEPC: semaphore ← 1;
```

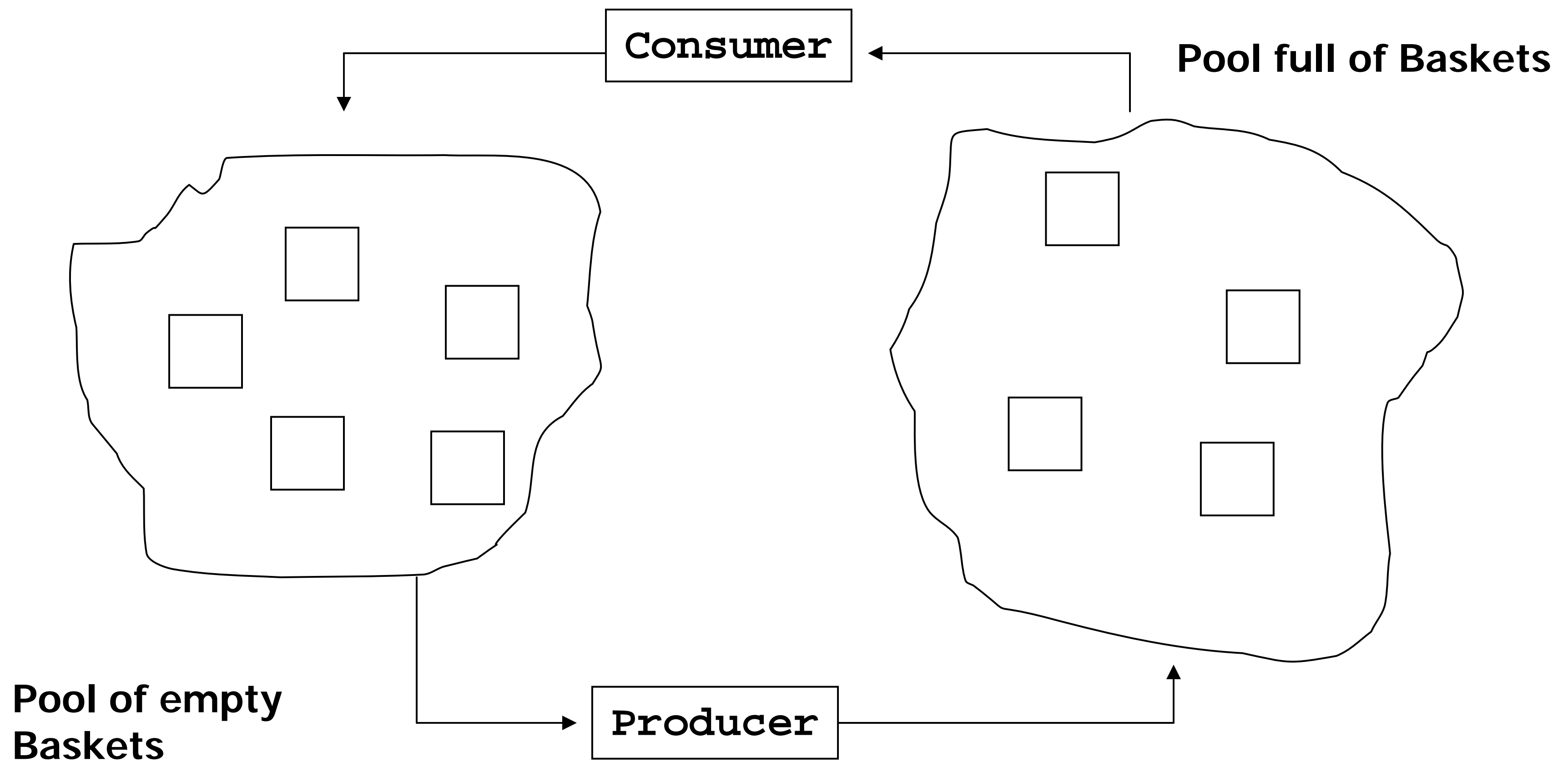
Producer

```
Begin
...
semWait(Empty);
semWait(MEPC);
<add item to buffer>
semSignal(MEPC);
semSignal(Full);
...
End;
```

Consumer

```
Begin
...
semWait(Full);
semWait(MEPC);
<remove item from buffer>
semSignal(MEPC);
semSignal(Empty);
...
End;
```

P/C – Another Look



P/C – Another Look

- 9 Baskets – Bounded
- Consumer – Empties basket
  - Can *only* remove basket from Full Pool, if one is there
    => Need "full" count
  - Empties basket and places it in Empty pool
- Producer – Fills basket
  - Can *only* remove basket from Empty pool, if one is there
    => Need "empty" count
  - Fills basket and places it in Full pool

P/C - Another Look

```
Shared semaphore: Emutex = 1, Fmutex = 1; full = 0,  empty = 9;
Shared buf_type: buffer[9];
```

```
producer() {
  buf_type *next, *here;
  while(True) {
    produce_item(next);
    semWait(empty); /*Claim empty buff*/
    semWait(Emutex); /*Manipulate pool*/
    here = obtain(empty);
    semSignal(Emutex);
    copy_buffer(next, here);
    semWait(Fmutex); /*Manipulate pool*/
    release(here, fullpool);
    semSignal(Fmutex); /*Sgnl full buff*/
    semSignal(full);
  }
}
```

```
consumer() {
  buf_type *next, *here;
  while(True) {
    semWait(full); /*Claim full buff*/
    semWait(Fmutex); /*Manipulte pool*/
    here = obtain(full);
    semSignal(Fmutex);
    copy_buffer(here, next);
    semWait(Emutex); /*Manipulte pool*/
    release(here, emptypool);
    semSignal(Enmutex);/*Sgnl empt buf*/
    semSignal(empty);
    consume_item(next);
  }
}
```

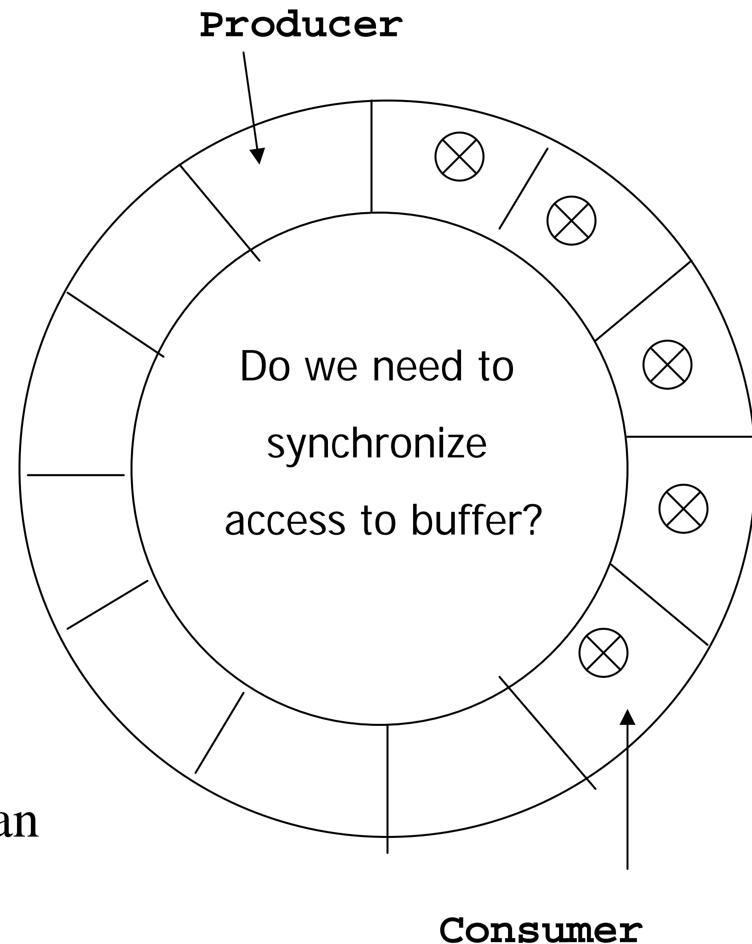
P/C - Example

- How realistic is P/C scenario?
- Consider a circular buffer
  - 12 slots
  - Producer points at next one it will fill
  - Consumer points at next one it will empty
- Don't want :

Producer = Consumer

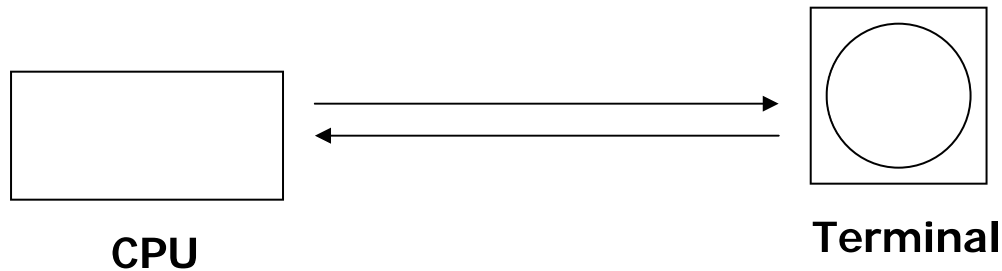
=> (1) Consumer "consumed" faster than producer "produced", or

(2) Producer "produced" faster than consumer "consumed".



P/C – Real World Scenario

- CPU can produce data faster than terminal can accept or viewer can read



Communication buffers in both

Xon/Xoff Flow Control

Semaphores: Other Primitives

Semaphore: S = 1;

- S.queue: interrogate whether the queue is empty or non-empty
- S.count: current semaphore value

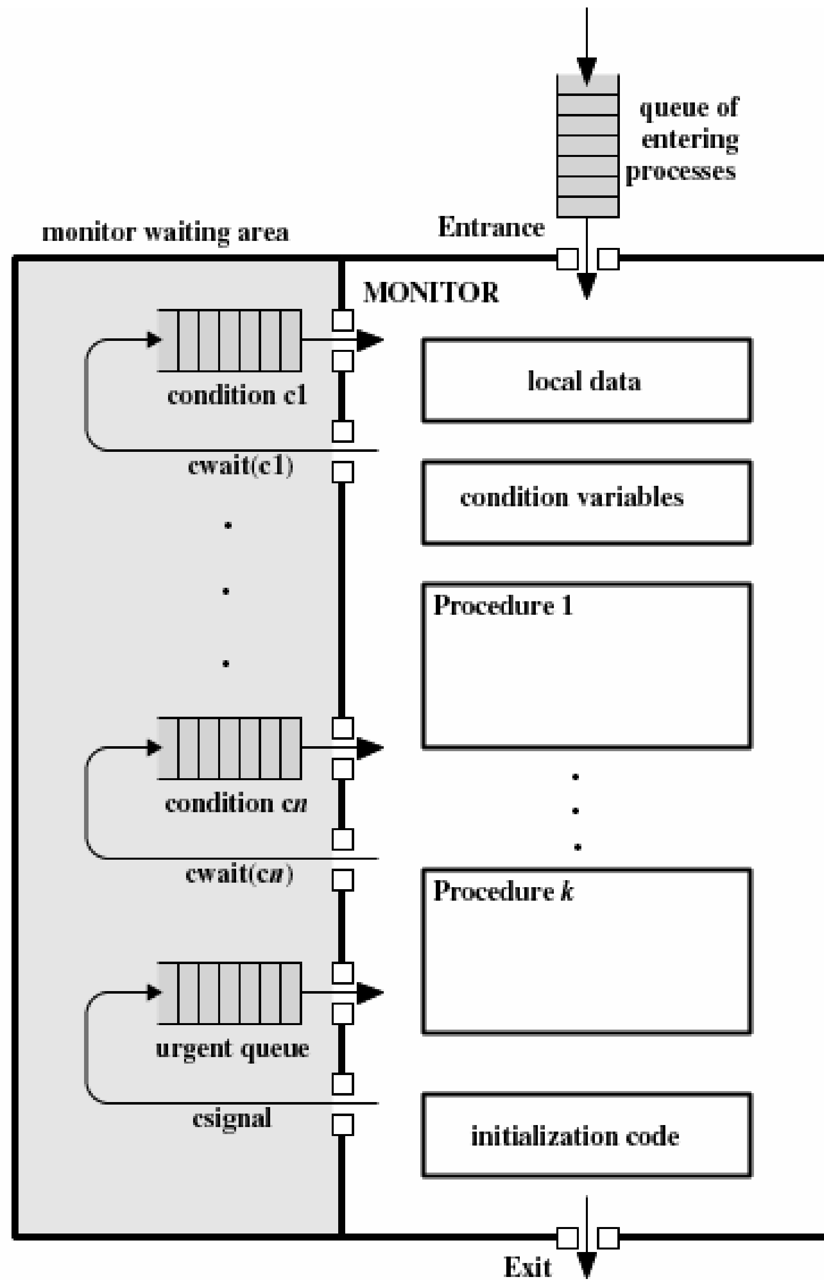
Mutual Exclusion and Synchronization

Language Defined

The Monitor

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



Monitor Structure

- Entrance only through monitor procedure
- Condition variables allows process suspension and “removal” from monitor

  Queue associated with **each** condition variable
- Local data can only be accesses through monitor procedures

Producer / Consumer: Monitor Solution

```
void producer()
char x;
{
        while (true)
        {
        produce(x);
        append(x);
        }
void consumer()
char x;
{
        while (true)
        {
        take(x);
        consume(x)
        }
}
void main()
{
   parbegin {produced, consumer};
}
```

Monitor

```
monitor boundedbuffer;
char Buffer (N)

void append (char x)
{
  :
  :
}

void take (char x)
{
  :
  :
}
```

Producer / Consumer Monitor Solution

```
monitor boundedbuffer;
char buffer [N];
int nextin, nextout;
int count;
cond notfull, notempty;
```

Condition variables

```
void append (char x)
{
     if (count == N)
        cwait(notfull);
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);
}
```

Append to buffer

```
void take (char x)
{
     if (count == 0)
        cwait(notempty);
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;
     csignal(notfull);
}
```

Take from buffer

```
{
     nextin = 0; nextout = 0; count = 0;
}
```

Monitor initialization

Monitor Accolades

- Provides equivalent functionality to that of semaphore
- Monitor construct itself enforces mutual exclusion
- Abstract Data Type – data, procedures, encapsulation
  - Initialization procedures
  - Local data only accessible to monitor procedures
  - Procedures (methods)
- All access and data manipulation defined / controlled at one place

Mutual Exclusion & Synchronization

through

Message Passing

Message Passing

- Enforce mutual exclusion
- Exchange information

<u>Typical Forms</u>

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

Send / Receive Scenarios

- Send primitive is executed
  - Sender is blocked until message is received, or
  - Sender continues
- Receive primitive is issued
  - Message previously sent, message received, execution continues, or
  - No message waiting and
    - Process blocks until message arrives, or
    - Process continues executing… abondons attempt to read a message

Send / Receive Synchronization

- Blocking send, blocking receive
  - Both sender and receiver are blocked until message is delivered
  - Called a *rendezvous*
- 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

Direct Addressing

- Send primitive includes a specific identifier of the destination process
  - Send (452, Msg)
- Receive primitive could know ahead of time from which process a message is expected
  - Receive (384, &Msg)
- Receive primitive could use source parameter to return a value when the receive operation has been performed
  - Receive (&PID, &Msg)

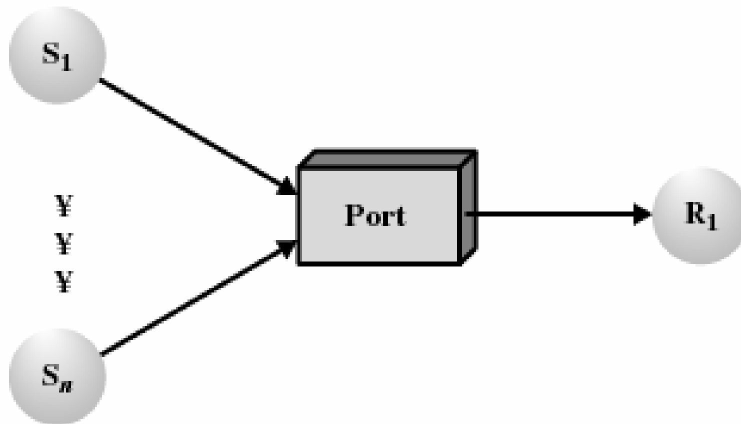
Indirect Addressing

Messages are sent to a shared data structure NOT to a specified process

- Queues are called mailboxes / tuple-space
- One process sends a message to the mailbox and the other process picks up the message from the mailbox
  - Mailboxes may / may not be tied to process instances

Indirect Addressing

One to One

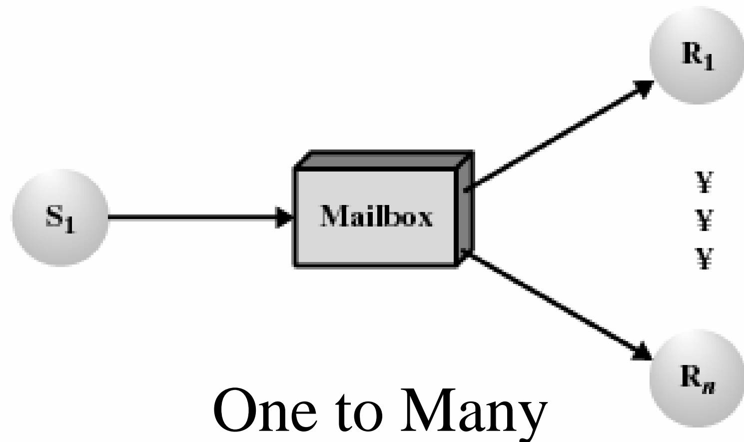


Many to One

- Private communication link
- Connections through *ports*
- Reduces potential interference from other processes

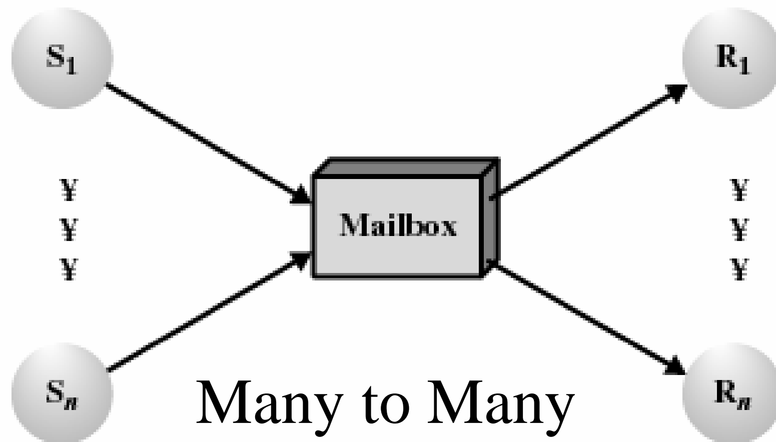
- Client / Server Applications
- Mail referred to as a *port*

Indirect Addressing



One to Many

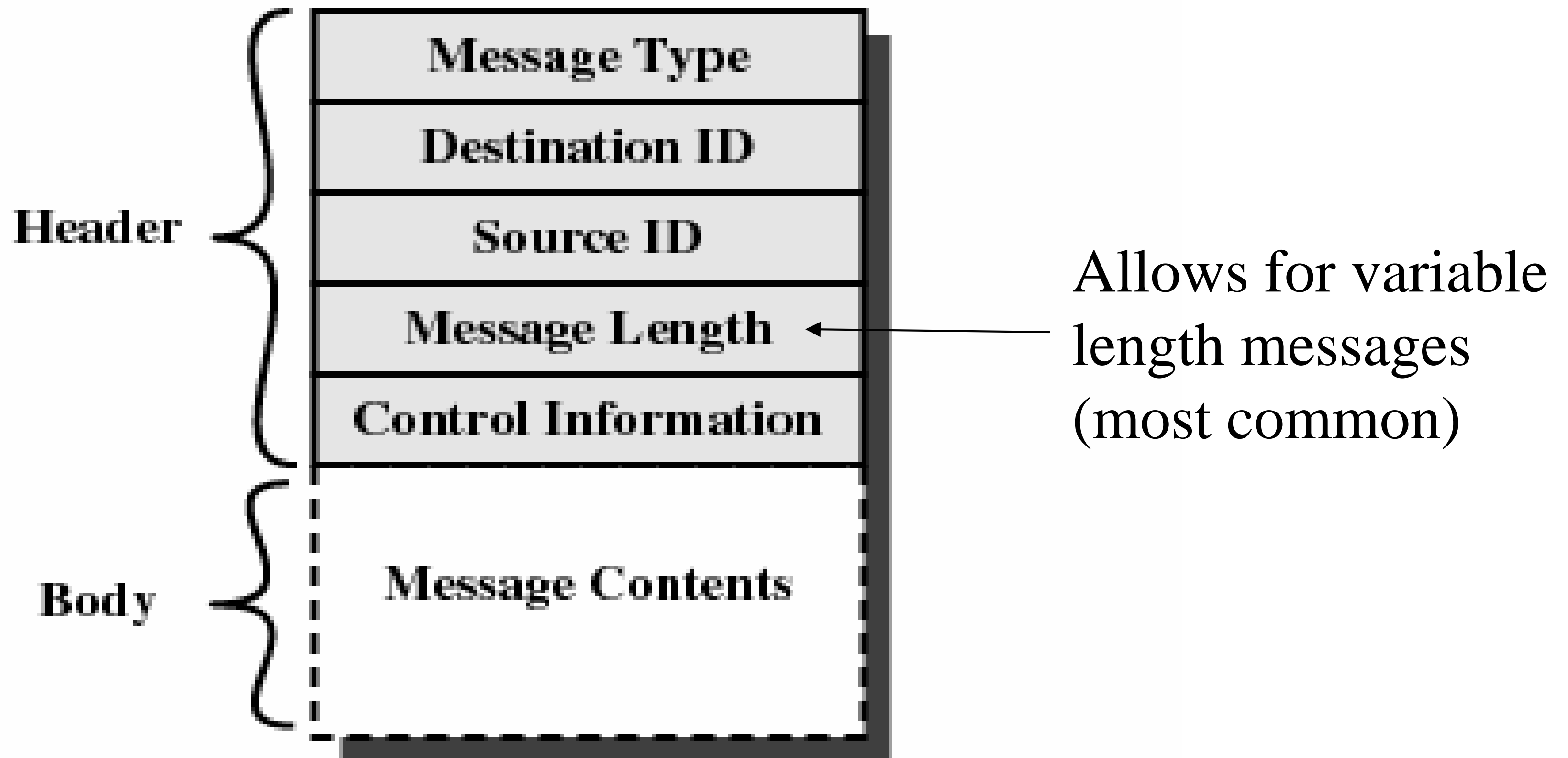
- One sender, multiple receivers
- Broadcast



Many to Many

- Multiple Servers providing *concurrent* services to multiple clients

General Message Format



Allows for variable length messages (most common)

Achieving Mutual Exclusion via Messages

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true)
    {
     receive (mutex, msg);
     /* critical section   */;
     send (mutex, msg);
     /* remainder   */;
    }
}
;
void main()
{
    create_mailbox (mutex);
    send (mutex, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

- Blocking Receive / Send
- One message - "token"

He who gets the token, enters the Critical Section

Solving the P/C Problem

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true)
    {
     receive (mayproduce, pmsg);
     pmsg = produce();
     send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true)
    {
     receive (mayconsume, cmsg);
     consume (cmsg);
     send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++)
        send (mayproduce, null);
    parbegin (producer, consumer);
}
```

wait
produce
signal

wait
consume
signal

Send out "capacity" mayproduce messages

Readers/Writers Problem

- Any number of readers may simultaneously read the file
- Only one writer at a time may write to the file
- If a writer is writing to the file, no reader may read it

Readers have Priority

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true)
    {
     semWait (x);
     readcount++;
     if (readcount == 1)
          semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if (readcount == 0)
          semSignal (wsem);
     semSignal (x);
    }
 }
void writer()
{
    while (true)
    {
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}
```

```
void main()
{
    readcount = 0;
    parbegin (reader, writer);
    }
```

- "x" guards updating of readcount
- "wsem" informs writer process if
  - one or more readers reading, or
  - another writer writing

Reader Priority: As long as any reader is "reading", another reader can enter to read

Writers Have Priority

```
/*program readersandwriters*/
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true)
    {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1)
                         semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0)
               semSignal (wsem);
     semSignal (x);
    }
}
```

```
void writer ()
{
    while (true)
    {
     semWait (y);
          writecount++;
          if (writecount == 1)
               semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0)
               semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```