Tasks and Intertask Communication

Introduction

Multitasking / Multithreading system
Supports multiple tasks

As we’ve noted
Important job in multitasking system
Exchanging data between tasks
Synchronizing tasks
Sharing resources

Let’s now examine these issues

Interprocess / Interthread Communication

When threads operating independently
Our systems have few if any
Conflicts
Chances for corruption
Contentions
Real systems
The interesting ones
Must deal with all such problems
Resources and inter thread communication
Must take place in robust manner

Interaction may be
Direct or indirect
Must be synchronized and co-ordinated
Want to prevent race conditions
Outcome of task or computation
Depends upon order in which tasks execute
Let’s begin by looking at shared information
Can occur in a variety of ways
Shared Variables

Simplest solution is shared memory environment

*Global Variables*

Simplest and fastest of these is

Global variables

Obvious problems

Higher priority process can pre-empt

Modify global data

*Shared Buffer*

Scheme says two processes share common set of

Memory locations

Producer

Puts data into buffer

Consumer

Removes

Several obvious problems

Arise if one process faster than other

Buffer size critical to avoid such problems

*Shared Double Buffer*

Scheme says two processes share two common sets of

Memory locations

Called ping-pong buffering scheme

Effective between processes running at different rates

One buffer being filled while other being emptied

Consumer blocks on lack of data

Producer must still avoid over running buffer

*Ring Buffer*

Scheme FIFO structure studied earlier

Permits simultaneous input and output

Using head and tail pointers

Must be careful to manage

Overflow

Underflow
Mailbox

Mutually agreed upon memory location
Two or more tasks use to pass data
Tasks rely on main scheduler to permit access
  Post operation for write
  Pend operation for read
Pend operation different from poll
  Poll task continually interrogates variable
  Pend task suspended while data not available
Variety of things passed
  Single bit
  Flag
  Single data word
  Pointer to data buffer
In most implementations
  Pend operation empties mailbox
    If several tasks pending on flag
      Enabled task resets flag
      Blocks multiple accesses to resource
        On single flag
Some implementations
  Permit queue of pending elements
    Rather than single entry
  Such scheme may be useful
    Multiple independent copies of critical resource

Messages

Message exchange is another means for communication
Now starting to move more into distributed systems

Called interprocess communication facility (IPC)
Note IPC is not mutually exclusive with shared memory
Idea to permit processes to communicate
  Without resorting to shared variables
  Particularly in different address spaces
IPC provides two operations
Send
Receive
Messages may be fixed or variable size

Basic Structure
If processes P1 and P2 wish to communicate
Must
Send and receive messages
Establish a communication link

Questions
Variety of questions one may ask
How to establish link
Can link be associated with multiple processes
How many links between pair or process
What is link capacity and are there buffers
What is message size
Are links
Unidirectional
Bi-directional

Implementation methods
Direct / Indirect communication
Symmetric / asymmetric communication
Auto or explicit buffering
Send by copy or reference
Fixed or variable sized messages

Let’s look at several of these

Communication

Direct
Each process must explicitly name sender / receiver of message
Messages logically of form
send (P1, message)  // send message to P1
receive (P2, message)  // receive message from P2
Link properties
- Link automatically established between every pair of processes
- Processes need only know each other's identity
- Link associated with only two processes
- Between each pair
  - Only single link
- Link may be
  - Uni/bi directional

Example
Consider skeletal structure
- Between producer P1 and consumer P2

```
repeat
  ...
  produce item in nextP1
  ...
  send (P2, nextP1)
until forever
```
```
repeat
  ...
  receive(P1, nextP2)
  ...
  consume item in nextP2
until forever
```

Observe scheme uses
- Symmetrical addressing
  - Sender and receiver must name each other
If want asymmetric addressing
- Sender only names recipient

Disadvantage
- Ties process name to implementation

Indirect
- Messages sent/received from shared variable
  - Generally in form of mailbox

```
send (M0, message) // send message to mailbox M0
receive (M0, message) // receive message from mailbox M0
```
Properties
- Link established
  - Only if processes have shared mailbox
- Link may be associated with multiple processes
- May be multiple links between processes
- Link may be uni/bi directional

Consider 3 processes P0, P1, P2
- All share M0
- Let P0 send and P1 and P2 receive
  - Question - who gets message

Solution
- Associate link with at most 2 processes
- Allow only one process to receive at a time
- Let system select receiver

Mailbox owner
- Process
  - If process owns mailbox
    - Can distinguish between
      - Owner
        - Who can only receive
      - User
        - Who can only send
  - Since each mailbox has unique owner
    - No ambiguity

System
- Exists independent of any process
- OS provides mechanism for process to
  - Create new mailbox
  - Send / receive messages through mailbox
  - Destroy mailbox

Creating process
- May pass access privileges
- Share mailbox
- Must manage memory associated with mailboxes
  - For which no process has access rights
Buffering

Establishes number of messages
Temporarily reside in link

Three possibilities

Zero capacity
Link cannot store message
Sender must wait for receiver to accept message
Called rendez vous

Bounded capacity
Message queue has length n
If space remaining
Sender can place message in queue
Continue
Else
Sender must wait for space

Unbounded
Potentially infinite length
Sender can post message
Continue
No wait
Thread Synchronization

Co-operating threads
One that can affect or be affected by another thread
May directly share logical address space
Code and data
Be allowed to share data only
Through files
Concurrent access to shared data
Can result in data inconsistency

Critical Sections
Consider following problem and code fragments
Exchanging messages through bounded buffer
Allow n items in buffer
Algorithm says

Producer
- If not full
  - add item
  - increment count
- else
  - wait until space

Consumer
- If item
  - get item
  - decrement count
- else
  - wait until item

Producer
repeat
...
produce an item in nextP1
while (count == n); // buffer full
buffer[in] = nextP1;
...
Consumer
repeat
  while (count == 0); // buffer empty
  nextP2 = buffer[out];
  out = (out+1) % n;
  count--;
  ...
  consume nextP2;
  ...
  until forever
Problem
Value of count
  Depends upon who accesses variable
  May be any of 3 different values
Variable count is critical variable
Within P1 or P2
  Denoted *critical section*
Critical section in general
  Section of code in which process is changing common variables
    File
    Table
    etc
While process in critical section
  Want to prevent access by all other processes
  Termed *mutual exclusion*
Abstractly may represent code as

```
repeat
  entry section
  critical section
  exit section
  remainder section
until forever
```

Semaphores
Solution to critical section problem must satisfy following requirements
  *mutual exclusion*
    If process P1 is in critical section
    No other process may enter
  *progress*
    If no process in critical section and some process wish to enter
    Only processes not in remainder section
      Can participate in decision
      Decision cannot be postponed indefinitely
bounded waiting

Must be bound on number of times other processes can enter critical section

After a process has made a request to enter
Before request granted

Methodology to protect critical section suggested by Dijkstra

Called semaphore

Semaphore

Integer or Boolean variable - S
Accessed only through two atomic operations
wait - P(S)
signal - V(S)

Operations may be defined by following code fragments

```
wait(s)
{
    while (s);
    s = TRUE;
}

signal(s)
{
    s = FALSE;
}
```
s is initialized to FALSE

These may now be used as

```
Process 1
{
    ...
    wait(s)
    critical section
    signal(s)
    ...
}

Process 2
{
    ...
    wait(s)
    critical section
    signal(s)
    ...
}
```
Consider two concurrently running processes p1 and p2 let
p1
  Contain statement s1
p2
  Contain statement s2

We require s1 be executed before s2
Thus define semaphore sync
  Initialize sync to TRUE

Observe
Because synch initialized to TRUE
  p2 will execute s2 only after
    p1 executes s1

**Spin Lock**

Main disadvantage of semaphores as described
  When wait encountered
    Encountering process blocked
      Must loop continuously while waiting
    Called *busy waiting*
    Waiting processes waste CPU cycles while waiting
      Other process could use productively
Such a semaphore called *spinlock*
  Because process spins while waiting for lock

Advantage of spinlock
  No context switch
    Can take long time
  If lock expected to be held for short time
    Spinlock useful
To overcome need for busy waiting
Modify definition of semaphore operations
When process executes wait operation
If semaphore TRUE
Must wait
Rather than wait process can *block* itself
Block operation places self in waiting queue
Associated with semaphore
Process state changed to waiting
Control transferred to scheduler
Blocked process should be restarted
Some other process executes signal operation
Process
Restarted
By *wakeup* operation
Places process in ready state
Placed in ready queue
Semaphore now defined as follows
s initialized to 1

```c
wait(s)
{ 
    s = s-1;  // on first pass s == 0
    if (s < 0)
    { 
        add process to waiting queue;
        block;
    }
}
```

```c
signal(s)
{ 
    s = s+1;
    if (s <=0)
    { 
        remove process from waiting queue;
        wakeup(p);
    }
}
```

Note semaphore now has integer value
block operation suspends invoking process
wakeup resumes execution of blocked process
Both operations provided by operating system calls
Observe
Waiting list can be implemented by linked list
Perhaps implement as FIFO queue
Mutexes and Counting Semaphores

Semaphores we’ve looked at called *binary semaphores*
Can take on either one of two values

**Mutex**
- Binary
- Used to serialize access to reentrant code
- Allows only one thread into controlled code section

**Example**
- Key to toilet

**Semaphore**
- Counting
- Can take on more than two values
  - Like previous example
- Used to protect pools of resources or track number or resources
- Restricts number of simultaneous users (threads) of shared resource

**Example**
- Number of keys to toilet

Working with a counting semaphore - let’s call these
- `wait` - `wait(s)`
- `signal` - `sig(s)`

```c
wait(s)
{
    s--;
    if (s<0)
        add this process to queue;
    block;
}
```

```c
sig(s)
{
    s++;
    if (s<=0)
        remove a process Pi from queue;
    wakeUp(Pi);
}
```

Each semaphore has
- Integer value
- List of associated processes
When process must wait on semaphore
- Added to list of processes
Signal
- Removes process from list
- Awakens it

Operations may be defined by following code fragments

**Bounded Buffer Problem**

Let’s look at one classic synchronization problem
Consider we have a pool of \( n \) buffers
Each can hold one item in this example
We define semaphores
\( \textit{mutex} \)
- Provides mutual exclusion for accesses to buffer pool
  - Initialized to value 1
\( \textit{Empty - semaphore} \)
- Count number of empty buffers
  - Initialized to \( n \)
\( \textit{Full - semaphore} \)
- Count number of full buffers
  - Initialized to 0

Code fragments illustrated as

```plaintext
Producer
repeat
    ... 
    produce an item \( \textit{anItem} \)
    ...
    wait(empty);  // check for non zero
    // dec empty cnt
    wait(mutex);
    ... 
    add \( \textit{anItem} \) to buffer \( \textit{nextProd} \);
    ...
    signal(mutex);
    signal(full);  // inc full cnt
    ...
until false

Consumer
repeat
    wait(full);  // check for non zero
    // dec full cnt
    wait(mutex);
    ... 
    remove \( \textit{anItem} \) from buffer \( \textit{nextCons} \);
    ...
    signal(mutex);
    signal(empty);  // inc empty cnt
    ...
    consume item \( \textit{anItem} \)
    ...
until false
```
Readers and Writers Problem

Data object may be shared among several concurrent processes
Some may want to read and others may want to write
Processes referred to as
Readers
Writers
If 2 readers access simultaneously
   No problem
If writer and any other process access simultaneously
   Big problem
Referred to as *readers - writers* problem
Several variations
First readers-writers
   No reader waits
   Unless writer has obtained access of shared variable
Second readers-writers
   Once writer ready
   Performs write as soon as possible
   If writer waiting
   No new reader started
Solution to first readers-writers problem
Define
Semaphores - mutex, wrtSem
   Initialize to 1
   mutex - ensure mutual exclusion when readcount updated
   wrtSem - mutual exclusion for writers
integer - numReaders
   Initialize to 0
   numReaders - count of readers currently accessing shared variable
Code fragment given as:

```
Reader Process
    wait(mutex);  // wait while mutex == 1
    numReaders++;  // inc number of readers
    if (numReaders == 1)  // if i'm the only reader
        wait(wrtSem);  // make sure no writers
        signal(mutex);  // mutex = 0
    ...  
    Perform reading;
    ...
    wait(mutex);  // wait for mutex == 1
    numReaders--;  // dec number of readers
    if (numReaders == 0)  // no readers
        signal(wrtSem);  // wrtSem = 0
        signal(mutex);  // mutex = 0
    ...
```

Note
If writer in critical section and n readers waiting
One reader queued on wrtSem
n-1 readers queued on mutex
If writer executes signal(wrtSem)
    May resume
    Waiting readers
    One waiting writer
    Decision made by scheduler

Monitors
Semaphores we’ve studied
    Fundamental synchronism mechanism
However low-level mechanism
    Easy to make errors with them
Monitors are program modules
   Offer more structure than semaphores
   Implementation can be as efficient

Monitors
   Data abstraction mechanism
   Encapsulate
       Representation of abstract object
   Provide public interface
       Only means by which
           Internal data may be manipulated
   Contains variable to
       Store object’s state
   Procedures that implement operations on object

We satisfy mutual exclusion
   By ensuring
       Procedures in same monitor
           Cannot execute simultaneously
   Conditional synchronization
       Provided through condition variables

Monitor used to group
   Representation and implementation
       The interface and body
           Of shared resource

Has *interface* and *body*
   Interface
       Specifies operations and behaviour provided by resource
   Body
       Contains
           Variables
               Represent state of resource
           Procedures
               Procedures
               Implement operations specified in interface
Schematically we have

```plaintext
monitor monName
{
  initialization statements //analogous to constructor
  procedures
  permanent variables
}
```

Procedures implement
  Visible operations
Permanent variables
  Shared by all processes
    In the monitor
      Like statics in C++ or pool variables in Smalltalk
    Denoted permanent
      Retain values on exit
        As long as monitor exists
Procedures
  May have local variables

By virtue of being an Abstract Data Type
Monitors is a distinct scope
  Only procedure names – this is the public interface
    Visible outside of monitor
Permanent variables
  Can only be changed
    Through one of the visible procedures
Statements within monitor
  Cannot affect variables outside monitor
    In different scope
Permanent variables
  Initialized before any procedure called
Accomplished by
  Executing initialization procedures
    When monitor instance created
Monitor sounds very similar to C++ class

Major difference

Monitor shared by multiple concurrently executing processes or threads

Consequently

Threads or processes using monitor

May require

*Mutual exclusion*

To monitor variables

*Synchronization*

To ensure monitor state conducive to continued execution

Mutual exclusion

Usually implicit

Synchronization

Implemented explicitly

Different processes require different forms of synchronization

Implementation achieved through

*Condition variables*

Shared variables discussed earlier

Monitor procedure

Called by external process or thread

A procedure is active

If a thread or process executing

Statement in procedure

At most one instance of monitor procedure

Active at any one time

Cannot have

Two different procedures invoked

or

Two invocations of same procedure

By definition

Execute with mutual exclusion

Ensured by

Language

Library

Operating system
Generally implemented
   Locks or semaphores
   Inhibiting certain interrupts

Condition Variables
Condition variables used as part of synchronization process
   Used to delay thread or process that
      Cannot safely continue
         Until monitor’s state satisfies some Boolean condition
   Used to awaken delayed process
      Once condition becomes true

Condition variable
Instance of variable of type cond
   cond myCondVar;
Can only be declared inside monitor
Value of condition thus it represents a queue
   Queue of delayed processes
Initially queue is empty
Value can only be accessed indirectly
   Much like private variables in C++ or Java
   Test state
      empty(myCondVar);
Thread can block on a condition variable
   wait(myCondVar);
      Execution of wait causes process to
         Move to rear of queue
         Relinquish exclusive access to monitor
Blocked process awakened
   signal(myCondVar);
      Execution of signal causes thread
         At head of queue to awaken

Execution of signal
   Seems to cause dilemma
      Upon execution two processes have potential to execute
         Awakened thread
         Signaling thread
Contradicts requirement
Only single thread active in monitor at once

Two possible paths for resolution
- Signal and continue
  Signaling thread continues
  Awakened process resumes at some delayed time
  Considered nonpreemptive
  Process executing signal
    Retains exclusive control of the monitor
- Signal and wait
  Considered to be preemptive
  Process executing signal
    Relinquishes control and passes lock
      To awakened process
  Awakened process preempts signaling process

Can describe process with following state diagram

<table>
<thead>
<tr>
<th>Operation / synchronization occurs as follows</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread calls monitor procedure</td>
</tr>
<tr>
<td>If another thread executing in monitor</td>
</tr>
<tr>
<td>Caller placed into entry queue</td>
</tr>
<tr>
<td>When monitor becomes free</td>
</tr>
<tr>
<td>Result of return or wait</td>
</tr>
<tr>
<td>One thread moves from entry queue into monitor</td>
</tr>
<tr>
<td>Else passes through entry queue</td>
</tr>
<tr>
<td>Begins executing immediately</td>
</tr>
<tr>
<td>If thread executes wait on a condition variable</td>
</tr>
</tbody>
</table>
While executing in monitor
Thread enters queue associated with that variable

When thread executes

*Signal and Continue* on a condition variable
Thread at head of associated queue
Moves to entry queue

*Signal and Wait* on a condition variable
Thread at head of associated queue
Moves to monitor
Thread executing in monitor
Moves to entry queue

**Bounded Buffer Problem with Monitor**

Let’s look at one classic synchronization problem
Looked at earlier with semaphores
Implemented with monitor
Consider we have a pool of \( n \) buffers
Each can hold one item in this example

We define a monitor \( boundedBuffer \)

We define condition variables

*notEmpty*
Signaled when buffer count \( > 0 \)
Tracks empty buffers
Initialized to 0

*notFull*
Signaled when buffer count \( < n \)
Tracks full buffers
Initialized to 0

We define procedures

*put(data)*
Puts data into a buffer
When space available
**get(data)**

Gets data from a buffer

When data available

We define the protected entity

bufferPool

We can implement our monitor as follows

```plaintext
monitor boundBuffer
    bufferPool;
    count = 0;
    cond notEmpty;  // signaled when count > 0
    cond notFull;   // signaled when count < n

put(anItem)
{
    while(count == n) wait (notFull);
    put anItem into a buffer
    signal (notEmpty);
}
get(anItem)
{
    while(count == 0) wait (notEmpty);
    get anItem from a buffer
    signal (notFull);
}
```

Code fragments illustrated as

```
Producer
repeat
...
    produce an item anItem
...
    boundBuffer.put(anItem)
...
forever
```

```
Consumer
repeat
...
    boundBuffer.get(anItem)
...
    consume item anItem
...
forever
```
Deadlocks and Starvation

Deadlocks

Implementation of semaphore or monitor with waiting queue

Can result in situation in which 2 or more processes

Wait indefinitely

Called *deadlock*

Consider 2 processes P0 and P1

Let each process have 2 semaphores

S1 and S2

May be resources each needs

R1 and R2

Need both to continue

Let R1 and R2 be set to value 1

Let

P0 set wait(S1)  // wait for R1 decrement S1 (=0)
P1 set wait(S2)   // wait for R2 decrement S2 (=0)

Now let

P0 set wait(S2)  // wait for R2 decrement S2 (=1)
P1 set wait(S1)   // wait for R1 decrement S1 (=1)

At this point

P0 must wait for signal(S2)
P1 must wait for signal(S1)

These operations cannot be executed

Processes blocked

Every process in set waiting for event

Possible only by another member in set

Will discuss in much greater detail shortly
Starvation

Problem called *starvation* can occur
- Process waiting within semaphore
- Other processes added or removed
- LIFO order

Events and Signals

Some languages provide mechanisms for handling
- Asynchronous events

Provides software interrupt
- Generally used for exceptions
  - Divide by zero
  - Arithmetic overflow
  - etc.

In addition to built in procedures
- Some permit user defined procedures to be
  - Provided and executed

ANSI-C

Provides signal and raise
- Signal
  - Software interrupt handler
  - Responds to exceptions indicated by raise
- Raise
  - Mechanism to signal an exception or event

Both implemented as function calls
- Passing pointers to functions
  - Can handle variety of events or exceptions