Symmetric and Concurrent Subprograms

Concurrency

# Allowing more than one task or process to execute at the same time.

Physical Concurrency:
# Executing multiple program units, tasks or processes on multiple processors.

Logical Concurrency:
# Executing multiple program units, task, or processes on a single processor in an interleaved fashion.

Symmetric Unit Control:

# Execution of multiple cooperative program units called co-routines.

# Also known as quasi-concurrency.

# Co-routines have a single thread of control.
Why Study Concurrency:

1) Provides a method for conceptualizing many problem domains that are inherently concurrent.

   # Examples:
   
   # Simulation
   # A Bank
   # A manufacturing process
   # Communication among devices

2) Take advantage of multi processor computer hardware.
Fundamental Concepts of Concurrency

Task:
# A program unit that can be in concurrent execution with other program units.

# Each task in a program can provide one thread of control.

Tasks vs. Subprograms:
# Tasks can be implicitly started, subprograms must be explicitly called.

# The program does not have to wait for the task to finish its execution before it can continue its own execution. (unlike procedure calls)

Inter-task Communication:
# Tasks often have to communicate in order to solve a common problem.

1) Shared memory (Unix's shared memory)
2) Message Passing (Unix's message queue)
3) Parameter passing (Unix's Pipes, sockets)

# Tasks that do not communicated are called disjoint tasks.

Synchronization:
# The mechanism that controls the order in which tasks execute and access shared resources.

# Two synchronization methods:
1) Cooperative Synchronization:
2) Competitive Synchronization:
1) **Cooperative Synchronization:**
   - Used between two tasks that need to wait for each other.
   - Task 'A' must wait for task 'B' to finish before it can continue its execution. (Producer/Consumer model)

2) **Competitive Synchronization:**
   - Used between two tasks that are competing with each other in order to gain access to a common resource (i.e. printer, disk, memory, etc..)
   - Simultaneous access to shared data often results in integrity violation or data corruption.

   # Bank Withdraw Anomaly: (lost update)

<table>
<thead>
<tr>
<th>Task 'A' read account balance</th>
<th>Balance = $1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 'B' read account balance</td>
<td></td>
</tr>
<tr>
<td>Task 'A' withdraws $500 (in private memory) New_Balance1 = Balance - 500</td>
<td></td>
</tr>
<tr>
<td>Task 'B' withdraws $300 (in private memory) New_Balance2 = Balance - 300</td>
<td></td>
</tr>
<tr>
<td>Task 'A' writes the New_Balance back to the database.</td>
<td>Balance = $500</td>
</tr>
<tr>
<td>Task 'B' writes the New_Balance back to the database.</td>
<td>Balance = $700</td>
</tr>
</tbody>
</table>
3 Methods for Process Synchronization:
1) Semaphores
2) Monitors
3) Message Passing

Semaphore

# An abstract data type with two elements. (A counter, a process queue and two operations P() or down() and V() or Up().

# Example:

Down(resource) // resource is a semaphore variable
{
    if (resource.counter > 0) // resource available
        resource.counter--;
    else
        Enqueue (The_Caller, resource.queue);
}

Up(resource) // resource is a semaphore variable
{
    if (resource.queue == Empty) // no process is waiting
        resource.counter++;
    else
        Dequeue (resource.queue); // allow a blocked process to proceed.
}

Semaphore printer1, printer2, memory, file;
begins
....
Down(printer1);
    < Critical Section Code>    // Access to shared resource or data
send the file to the printer
\texttt{Up(printer1);} 
....
end;
Problems with semaphores:

# Deadlock

When two or more processes each have a lock on a data or resource, and each is waiting for the other to release its resources. (Circular wait)

\[\text{Dead Lock}\]

\[\text{Process}\]

\[\text{Resource}\]

# P1 has R1 and wants R2,
# P2 has R2 and wants R1.

# Leaving off the P() will cause possible synchronization and data corruption.

# Leaving off the V() will cause deadlocks.

# Semaphores are prone to programmer error. (i.e. the order of P() and V(), or simply forgetting them!)
Monitors

# The shared data itself is viewed as an ADT and is maintained by the language execution environment.

Shared Data or resource as well as the subprogram that need to be executed in mutual exclusion are maintained within the monitor.

Monitor ensures serial access to resources. Only one sub program (task, process) can run at a time. Calls to the monitor get queued if the monitor is busy.

The idea is to free the programmer from having to worry about synchronization. The monitor will take care of things for you.
Example: (special language construct, not every language has them)

```
Monitor my_monitor
    int i;
    condition c;

    procedure Consumer(X);
        .....;
        .....;
        end;

    Procedure Producer(X);
        .....;
        end;
end Monitor;
```

# No Deadlock (No circular wait).

# Needs to be a language construct. Unlike semaphores, the programmer can not create his own.

# In order to implement a Monitor one would still have to use a semaphore internally.

# Not too many languages support monitors. (ADA, and Java support it)
Message Passing

# Ideal for distributed setting (distributed processing over a LAN) and multi-processor with separate memory.

# Both Monitors and Semaphores were designed for either single processors or multi-processors with shared memory access.

# Asynchronous passing and receiving of messages.

# A process will check its message queue to see if there are any messages for it.

# Non preemptive. The process is not interrupted.

# Every shared resource is given a message queue. Processes which need an exclusive access to the resource simply send a request (a message) and wait for it to be done.

# Rendezvous: (ADA)

The act of transmitting and receiving a message between two tasks.

# Message Passing performs slightly better in a shared memory centralized system but performs much better in a distributed system.

# Message passing naturally supports the concept of separate processes, executing in parallel.
Multi Threading in JAVA

What is a thread?

- Similar to a process however with less overhead.

- Also Known as a Light Weight Process.

- A thread is a single sequential flow of control within a program.

Thread vs. Process

- Each process (executing program), in order to execute, requires many resources from the OS. (I.e. program counter, registers, local variables, code, data, stack segment, ..., a place in the process table, etc..)

- Threads run within a process therefore, there is less overhead for the operating system. However, they do have their own context within a running program. (I.e. its own execution stack, program counter, etc..) The thread shares it code and data segment with the other threads.

** Java’s Thread facilities are Platform dependent. Therefore threaded applications may behave differently on different platforms such as Windows 9x or Sun Solaris.
Two Methods for Creating a Multi-Threaded Application:

1) Declare a class to be a subclass of the THREAD class. Then override the RUN method of the THREAD class to perform the desired task.

```java
class mythread extends thread {
    ......
    ......
    public void run() {
        ....
        ....
    }
}
```

Used mostly when no other class needs to be inherited.

```java
t = new mythread(); // create a new thread
t.start(); // start the execution of the thread.
```

2) Declare a class that implements the RUNNABLE interface. Then implement the RUN method.

```java
class mythread implements runnable {
    ......
    ......
    public void run() {
        ....
        ....
    }
}
```

Used mostly when the class already extends another class. (i.e. inherits from another class).

Note: Java does not allow multiple inheritance.

```java
t = new mythread(); // create a new thread
t.start(); // start the execution of the thread.
```
Process Life Cycle

Thread Life Cycle:
Thread Methods

Thread(); // Constructor will give a default thread name to the thread. “Thread-x”

Thread(string thread_name); // Constructor

run(); // The overloaded function that get called by the Java multithreading environment. The main code for running a thread goes here...

Start(); // initiates the execution for the thread. Start() will setup the threading environment and then call the run() function

FIGURE

// Start will return immediately to the caller to continue to execute the main thread of execution.

Sleep(int); // Allow a thread to go into SLEEP for a number of milliseconds.

// sleeping threads do not use any CPU therefore a lower priority thread may execute if all the higher priority threads are sleep().

Interrupt(); // to interrupt a thread, wake it up from a sleep state and place it into ready state.

IsInterrupted(); // to determine if a thread is has been interrupted.
IsActive(); // Returns TRUE if start() has been called for a
given thread and the thread is not dead. (I.e. its
run() method has not finished)

setName(string);
getName(); // Set and get the name of the current thread.
toString() // Return a string representing the Thread-Name,
Priority and Thread-group.

CurrentThread(); // Retrun the reference to the current
executing thread object.

Join(); // wait for this thread to die, before the calling
thread can proceed. (Wait forever!! Can be
dangerous. Deadlock or Live lock can occur.)

Join(long milliseconds); // Wait at most x milliseconds for this
thread to die.

| Wait(); notify(); notifyall(); | These methods are part of the
| object class, however are used in
| thread applications. |

Object.wait(); // a running thread can invoke the wait() on an
object and wait for that object in its wait state.

Object.notify(); // one of the threads waiting on this object’s monitor
will be awakened. (Placed in READY state). The
choice is arbitrary, and occurs at the discretion of
the implementation.

Object.notifyall(); // all the thread waiting for this object are
placed in their READY state.
Thread Priorities:
Thread Scheduling:
Sample Java Multi-Threading Code

// Fig. 15.3: ThreadTester.java
// Show multiple threads printing at different intervals.

public class ThreadTester {
    // create and start threads
    public static void main( String args[] )
    {
        PrintThread thread1, thread2, thread3, thread4;

        // create four PrintThread objects
        thread1 = new PrintThread( "thread1" );
        thread2 = new PrintThread( "thread2" );
        thread3 = new PrintThread( "thread3" );
        thread4 = new PrintThread( "thread4" );

        System.err.println( "nStarting threads" );

        // start executing PrintThreads
        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();

        System.err.println( "Threads started\n" );
    }
}

} // end class ThreadTester
// Each object of this class picks a random sleep interval.
// When a PrintThread executes, it prints its name, sleeps,
// prints its name again and terminates.
class PrintThread extends Thread {
    private int sleepTime;

    // PrintThread constructor assigns name to thread
    // by calling superclass Thread constructor
    public PrintThread( String name ) {
        super( name );

        // sleep between 0 and 5 seconds
        sleepTime = (int) ( Math.random() * 5000 );

        // display name and sleepTime
        System.err.println("Name: "+getName()+";  sleep: "+sleepTime);
    }

    // control thread's execution
    public void run() {
        // put thread to sleep for a random interval
        try {
            System.err.println(getName()+" going to sleep");

            // put thread to sleep
            Thread.sleep( sleepTime );
        }

        // if thread interrupted during sleep, catch exception
        // and display error message
        catch ( InterruptedException interruptedException ) {
            System.err.println( interruptedException.toString() );
        }

        // print thread name
        System.err.println( getName()+" done sleeping" );
    }
} // end class PrintThread
programs or to the documentation contained in these books. The authors
and publisher shall not be liable in any event for incidental or
consequential damages in connection with, or arising out of, the
furnishing, performance, or use of these programs.

*************************************************************************/