Operating systems

Module 7
IPC (Interprocess communication)
PART I
INTERPROCESS COMMUNICATION

- Processes within a system may be independent or cooperating.
- Cooperating process can affect or be affected by other processes, including sharing data.
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Two models of IPC:
  - Shared memory
  - Message passing
COMMUNICATION MODELS

(a) message passing
(b) shared memory

(a)

(b)
PRODUCER-CONSUMER PROBLEM

- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - *unbounded-buffer* places no practical limit on the size of the buffer
  - *bounded-buffer* assumes that there is a fixed buffer size
IPC – MESSAGE PASSING

- Mechanism for processes to communicate and to synchronize their actions

- Message system – processes communicate with each other without resorting to shared variables

- IPC facility provides two operations:
  - send(message) – message size fixed or variable
  - receive(message)

- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive

- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., direct or indirect, synchronous or asynchronous)
IPC – MESSAGE PASSING

- Implementation Questions
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?
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

- Properties of direct communication link
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bidirectional
INDIRECT COMMUNICATION

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

• Properties of communication link
  o Link established only if processes share a common mailbox
  o A link may be associated with many processes
  o Each pair of processes may share several communication links
  o Link may be unidirectional or bi-directional
MESSAGE SYNCHRONIZATION

- Message passing may be either blocking or non-blocking

- Blocking is considered synchronous
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available

- Non-blocking is considered asynchronous
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null
BUFFERING

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of $n$ messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits
Operating systems

Module 7
Thread Concept
PART II
THREADS

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Thread Cancellation
THREADS - OVERVIEW

• What is a thread?
  ◦ A lightweight process
  ◦ A unit of execution of a process
THREDS - OVERVIEW

single threaded and multithreaded

![Diagram showing single-threaded and multithreaded processes](image-url)
THREADS - OVERVIEW

- Most modern applications are multithreaded
- Multiple tasks can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Kernels are generally multithreaded
BENEFITS OF THREADS

- **Responsiveness**
  - may allow continued execution if part of process is blocked, especially important for user interfaces

- **Resource Sharing**
  - threads share resources of process

- **Economy**
  - cheaper than process creation, thread switching lower overhead than context switching

- **Scalability**
  - process can take advantage of multiprocessor architectures
Multicore or multiprocessor systems putting pressure on programmers, challenges include:

- Dividing activities
- Balance
- Data splitting
- Data dependency
- Testing and debugging

Parallelism implies a system can perform more than one task simultaneously.

Concurrency supports more than one task making progress.

- Single processor / core, scheduler providing concurrency
CONCURRENCY VS. PARALLELISM

- Concurrent execution on a single-core system:

<table>
<thead>
<tr>
<th>single core</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>T₁</th>
<th>T₂</th>
<th>T₃</th>
<th>T₄</th>
<th>T₁</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Parallelism on a multi-core system:

<table>
<thead>
<tr>
<th>core 1</th>
<th>T₁</th>
<th>T₃</th>
<th>T₁</th>
<th>T₃</th>
<th>T₁</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>core 2</th>
<th>T₂</th>
<th>T₄</th>
<th>T₂</th>
<th>T₄</th>
<th>T₂</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
USER THREADS AND KERNEL THREADS

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X
MULTITHREADING MODELS

- Many-to-One
- One-to-One
- Many-to-Many
- Two-level
MANY-TO-ONE

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
ONE-TO-ONE

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency
- Number of threads per process sometimes restricted due to overhead

![Diagram showing one-to-one mapping between user threads and kernel threads]
MANY-TO-MANY MODEL

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
**TWO-LEVEL MODEL**

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
THREAD LIBRARIES

- **Thread library** provides programmer with API for creating and managing threads

- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
IMPLICIT THREADING

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads

- Creation and management of threads done by compilers and run-time libraries rather than programmers

- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch

- Other methods include Microsoft Threading Building Blocks (TBB), java.util.concurrent package
THREAD POOLS

- Create a number of threads in a pool where they await work

- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
OPENMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

```c
#pragma omp parallel
Create as many threads as there are cores

#pragma omp parallel for for(i=0;i<N;i++)
{
    c[i] = a[i] + b[i];
}
Run for loop in parallel
```
GRAND CENTRAL DISPATCH

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue
THREAD CANCELLATION

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled