ICS 143 - Principles of Operating Systems

Lectures 3 and 4 - Processes and Threads
Prof. Nalini Venkatasubramanian
nalini@ics.uci.edu

Some slides adapted from http://www-inst.eecs.berkeley.edu/~cs162/ Copyright © 2010 UCB.
Note that some slides are also adapted from course text slides © 2008 Silberschatz
Outline

- Process Concept
  - Process Scheduling
  - Operations on Processes
  - Cooperating Processes
- Threads
- Interprocess Communication
Process Concept

- An operating system executes a variety of programs
  - batch systems - jobs
  - time-shared systems - user programs or tasks
  - Job, task and program used interchangeably

- Process - a program in execution
  - process execution proceeds in a sequential fashion

- A process contains
  - program counter, stack and data section
More to a process than just a program:
- Program is just part of the process state
- I run Vim or Notepad on lectures.txt, you run it on homework.java – Same program, different processes

Less to a process than a program:
- A program can invoke more than one process
- A web browser launches multiple processes, e.g., one per tab
A process changes state as it executes.
Process States

- **New** - The process is being created.
- **Running** - Instructions are being executed.
- **Waiting** - Waiting for some event to occur.
- **Ready** - Waiting to be assigned to a processor.
- **Terminated** - Process has finished execution.
Process Control Block

- Contains information associated with each process
  - Process State - e.g. new, ready, running etc.
  - Process Number – Process ID
  - Program Counter - address of next instruction to be executed
  - CPU registers - general purpose registers, stack pointer etc.
  - CPU scheduling information - process priority, pointer
  - Memory Management information - base/limit information
  - Accounting information - time limits, process number
  - I/O Status information - list of I/O devices allocated
Representation of Process Scheduling

Process (PCB) moves from queue to queue

*When does it move? Where? A scheduling decision*
Ready Queue And Various I/O Device Queues
Process Scheduling Queues

- Job Queue - set of all processes in the system
- Ready Queue - set of all processes residing in main memory, ready and waiting to execute.
- Device Queues - set of processes waiting for an I/O device.
- Process migration between the various queues.
- Queue Structures - typically linked list, circular list etc.
Enabling Concurrency and Protection: Multiplex processes

- Only one process (PCB) active at a time
  - Current state of process held in PCB:
    - "snapshot" of the execution and protection environment
  - Process needs CPU, resources

- Give out CPU time to different processes (Scheduling):
  - Only one process “running” at a time
  - Give more time to important processes

- Give pieces of resources to different processes (Protection):
  - Controlled access to non-CPU resources
    - E.g. Memory Mapping: Give each process their own address space
Enabling Concurrency: Context Switch

- Task that switches CPU from one process to another process
  - the CPU must save the PCB state of the old process and load the saved PCB state of the new process.

- Context-switch time is overhead
  - System does no useful work while switching
  - Overhead sets minimum practical switching time; can become a bottleneck

- Time for context switch is dependent on hardware support (1-1000 microseconds).
CPU Switch From Process to Process

- Code executed in kernel above is overhead
  - Overhead sets minimum practical switching time
Schedulers

- **Long-term scheduler (or job scheduler)** -
  - selects which processes should be brought into the ready queue.
  - invoked very infrequently (seconds, minutes); may be slow.
  - controls the degree of multiprogramming

- **Short term scheduler (or CPU scheduler)** -
  - selects which process should execute next and allocates CPU.
  - invoked very frequently (milliseconds) - must be very fast
  - Sometimes the only scheduler in the system

- **Medium Term Scheduler**
  - swaps out process temporarily
  - balances load for better throughput
Medium Term (Time-sharing) Scheduler

- swap in
- partially executed swapped out processes
- swap out

- ready queue
- CPU
- end

- I/O
- I/O waiting queues
A tree of processes in Linux
Process Profiles

- **I/O bound process** -
  - spends more time in I/O, short CPU bursts, CPU underutilized.

- **CPU bound process** -
  - spends more time doing computations; few very long CPU bursts, I/O underutilized.

- **The right job mix:**
  - Long term scheduler - admits jobs to keep load balanced between I/O and CPU bound processes
  - Medium term scheduler – ensures the right mix (by sometimes swapping out jobs and resuming them later)
Process Creation

- Processes are created and deleted dynamically
- Process which creates another process is called a *parent* process; the created process is called a *child* process.
- Result is a tree of processes
  - e.g. UNIX - processes have dependencies and form a hierarchy.
- Resources required when creating process
  - CPU time, files, memory, I/O devices etc.
UNIX Process Hierarchy

- root
- page daemon
- swapper
- init
  - user 1
  - user 2
  - user 3
What does it take to create a process?

- Must construct new PCB
  - Inexpensive
- Must set up new page tables for address space
  - More expensive
- Copy data from parent process? (Unix `fork()`)
  - Semantics of Unix `fork()` are that the child process gets a complete copy of the parent memory and I/O state
  - Originally very expensive
  - Much less expensive with “copy on write”
- Copy I/O state (file handles, etc)
  - Medium expense
Process Creation

- **Resource sharing**
  - Parent and children share all resources.
  - Children share subset of parent’s resources - prevents many processes from overloading the system.
  - Parent and children share no resources.

- **Execution**
  - Parent and child execute concurrently.
  - Parent waits until child has terminated.

- **Address Space**
  - Child process is duplicate of parent process.
  - Child process has a program loaded into it.
UNIX Process Creation

- Fork system call creates new processes
- execve system call is used after a fork to replace the processes memory space with a new program.
Process Termination

- Process executes last statement and asks the operating system to delete it (exit).
  - Output data from child to parent (via wait).
  - Process’ resources are deallocated by operating system.

- Parent may terminate execution of child processes.
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting
    - OS does not allow child to continue if parent terminates
    - Cascading termination
Threads

- Processes do not share resources well
  - high context switching overhead
- Idea: Separate concurrency from protection
- Multithreading: a single program made up of a number of different concurrent activities
- A thread (or lightweight process)
  - basic unit of CPU utilization; it consists of:
    - program counter, register set and stack space
  - A thread shares the following with peer threads:
    - code section, data section and OS resources (open files, signals)
    - No protection between threads
  - Collectively called a task.
- Heavyweight process is a task with one thread.
Single and Multithreaded Processes

- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
  - Keeps buggy program from trashing the system
Benefits

■ Responsiveness
■ Resource Sharing
■ Economy
■ Utilization of MP Architectures
Threads (Cont.)

- In a multiple threaded task, while one server thread is blocked and waiting, a second thread in the same task can run.
  - Cooperation of multiple threads in the same job confers higher throughput and improved performance.
  - Applications that require sharing a common buffer (i.e. producer-consumer) benefit from thread utilization.

- Threads provide a mechanism that allows sequential processes to make blocking system calls while also achieving parallelism.
Thread State

- State shared by all threads in process/addr space
  - Contents of memory (global variables, heap)
  - I/O state (file system, network connections, etc)

- State “private” to each thread
  - Kept in TCB (Thread Control Block)
  - CPU registers (including, program counter)
  - Execution stack
    - Parameters, Temporary variables
    - return PCs are kept while called procedures are executing
Threads (cont.)

- Thread context switch still requires a register set switch, but no memory management related work!!
- Thread states -
  - *ready, blocked, running, terminated*
- Threads share CPU and only one thread can run at a time.
- No protection among threads.
Examples: Multithreaded programs

- **Embedded systems**
  - Elevators, Planes, Medical systems, Wristwatches
  - Single Program, concurrent operations

- **Most modern OS kernels**
  - Internally concurrent because have to deal with concurrent requests by multiple users
  - But no protection needed within kernel

- **Database Servers**
  - Access to shared data by many concurrent users
  - Also background utility processing must be done
More Examples: Multithreaded programs

- **Network Servers**
  - Concurrent requests from network
  - Again, single program, multiple concurrent operations
  - File server, Web server, and airline reservation systems

- **Parallel Programming (More than one physical CPU)**
  - Split program into multiple threads for parallelism
  - This is called Multiprocessing
Real operating systems have either

- One or many address spaces
- One or many threads per address space

<table>
<thead>
<tr>
<th># threads Per AS:</th>
<th># of addr spaces:</th>
<th>One</th>
<th>Many</th>
</tr>
</thead>
<tbody>
<tr>
<td>One</td>
<td>MS/DOS, early Macintosh</td>
<td>Traditional UNIX</td>
<td></td>
</tr>
<tr>
<td>Many</td>
<td>Embedded systems (Geoworks, VxWorks, JavaOS, etc) JavaOS, Pilot(PC)</td>
<td>Mach, OS/2, Linux Windows 9x???, Win NT to XP, Solaris, HP-UX, OS X</td>
<td></td>
</tr>
</tbody>
</table>
Types of Threads

- Kernel-supported threads
- User-level threads
- Hybrid approach implements both user-level and kernel-supported threads (Solaris 2).
Kernel Threads

- **Supported by the Kernel**
  - Native threads supported directly by the kernel
  - Every thread can run or block independently
  - One process may have several threads waiting on different things

- **Downside of kernel threads: a bit expensive**
  - Need to make a crossing into kernel mode to schedule

- **Examples**
  - Windows XP/2000, Solaris, Linux, Tru64 UNIX, Mac OS X, Mach, OS/2
User Threads

- Supported above the kernel, via a set of library calls at the user level.
  - Thread management done by user-level threads library
    - User program provides scheduler and thread package
  - May have several user threads per kernel thread
  - User threads may be scheduled non-preemptively relative to each other (only switch on yield())

- Advantages
  - Cheap, Fast
    - Threads do not need to call OS and cause interrupts to kernel
  - Disadv: If kernel is single threaded, system call from any thread can block the entire task.

- Example thread libraries:
  - POSIX Pthreads, Win32 threads, Java threads
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many
Many-to-One

- Many user-level threads mapped to single kernel thread

Examples:
  - Solaris Green Threads
  - GNU Portable Threads
One-to-One

- Each user-level thread maps to kernel thread

Examples
- Windows NT/XP/2000; Linux; Solaris 9 and later
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
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package
Solaris 2 is a version of UNIX with support for:
- kernel and user level threads, symmetric multiprocessing and real-time scheduling.

**Lightweight Processes (LWP)**
- intermediate between user and kernel level threads
- each LWP is connected to exactly one kernel thread
Threads in Solaris 2

- Task 1
- Task 2
- Task 3

Kernel threads

User level thread

Lightweight process

Kernel

CPU
Threads in Solaris 2

Resource requirements of thread types

- Kernel Thread: small data structure and stack; thread switching does not require changing memory access information - relatively fast.
- Lightweight Process: PCB with register data, accounting and memory information - switching between LWP is relatively slow.
- User-level thread: only needs stack and program counter; no kernel involvement means fast switching. Kernel only sees the LWPs that support user-level threads.
Two-level Model

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

- Examples
  - IRIX, HP-UX, Tru64 UNIX, Solaris 8 and earlier
Threading Issues

- Semantics of `fork()` and `exec()` system calls
- Thread cancellation
- Signal handling
- Thread pools
- Thread specific data
Multi(processing, programming, threading)

Definitions:
- Multiprocessing = Multiple CPUs
- Multiprogramming = Multiple Jobs or Processes
- Multithreading = Multiple threads per Process

What does it mean to run two threads “concurrently”?
- Scheduler is free to run threads in any order and interleaving: FIFO, Random, ...
- Dispatcher can choose to run each thread to completion or time-slice in big chunks or small chunks
Cooperating Processes

- **Concurrent Processes can be**
  - Independent processes
    - cannot affect or be affected by the execution of another process.
  - Cooperating processes
    - can affect or be affected by the execution of another process.

- **Advantages of process cooperation:**
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience (e.g. editing, printing, compiling)

- **Concurrent execution requires**
  - process communication and process synchronization
Interprocess Communication (IPC)

Separate address space isolates processes
- High Creation/Memory Overhead; (Relatively) High Context-Switch Overhead

Mechanism for processes to communicate and synchronize actions.
- Via shared memory - Accomplished by mapping addresses to common DRAM
  - Read and Write through memory
- Via Messaging system - processes communicate without resorting to shared variables.
  - `send()` and `receive()` messages
  - Can be used over the network!
- Messaging system and shared memory not mutually exclusive
  - can be used simultaneously within a single OS or a single process.
Shared Memory Communication

- Communication occurs by “simply” reading/writing to shared address page
  - Really low overhead communication
  - Introduces complex synchronization problems
Cooperating Processes via Message Passing

- IPC facility provides two operations.
  - `send(message)` - message size can be fixed or variable
  - `receive(message)`

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

- Fixed vs. Variable size message
  - Fixed message size - straightforward physical implementation, programming task is difficult due to fragmentation
  - Variable message size - simpler programming, more complex physical implementation.
Implementation Questions

- How are links established?
- Can a link be associated with more than 2 processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Fixed or variable size messages?
- Unidirectional or bidirectional links?

......
Direct Communication

Sender and Receiver processes must name each other explicitly:

- send\((P, \text{message})\) - send a message to process \(P\)
- receive\((Q, \text{message})\) - receive a message from process \(Q\)

Properties of communication link:

- Links are established automatically.
- A link is associated with exactly one pair of communicating processes.
- Exactly one link between each pair.
- Link may be unidirectional, usually bidirectional.
Indirect Communication

- Messages are directed to and received from mailboxes (also called ports)
  - Unique ID for every mailbox.
  - Processes can communicate only if they share a mailbox.
    \[\text{Send}(A, \text{message}) \quad /* \text{send message to mailbox } A */\]
    \[\text{Receive}(A, \text{message}) \quad /* \text{receive message from mailbox } A */\]

- Properties of communication link
  - Link established only if processes share a common mailbox.
  - Link can be associated with many processes.
  - Pair of processes may share several communication links
  - Links may be unidirectional or bidirectional
Indirect Communication using mailboxes
Mailboxes (cont.)

- **Operations**
  - create a new mailbox
  - send/receive messages through mailbox
  - destroy a mailbox

- **Issue: Mailbox sharing**
  - P1, P2 and P3 share mailbox A.
  - P1 sends message, P2 and P3 receive… who gets message??

- **Possible Solutions**
  - disallow links between more than 2 processes
  - allow only one process at a time to execute receive operation
  - allow system to arbitrarily select receiver and then notify sender.
Message Buffering

- Link has some capacity - determine the number of messages that can reside temporarily in it.

- Queue of messages attached to link
  - Zero-capacity Queues: 0 messages
    - sender waits for receiver (synchronization is called *rendezvous*)
  - Bounded capacity Queues: Finite length of $n$ messages
    - sender waits if link is full
  - Unbounded capacity Queues: Infinite queue length
    - sender never waits
Message Problems - Exception Conditions

- **Process Termination**
  - **Problem:** \( P(\text{sender}) \) terminates, \( Q(\text{receiver}) \) blocks forever.
  - **Solutions:**
    - System terminates \( Q \).
    - System notifies \( Q \) that \( P \) has terminated.
    - \( Q \) has an internal mechanism (timer) that determines how long to wait for a message from \( P \).

- **Problem:** \( P(\text{sender}) \) sends message, \( Q(\text{receiver}) \) terminates. In automatic buffering, \( P \) sends message until buffer is full or forever. In no-buffering scheme, \( P \) blocks forever.
  - **Solutions:**
    - System notifies \( P \)
    - System terminates \( P \)
    - \( P \) and \( Q \) use acknowledgement with timeout
Message Problems - Exception Conditions

- **Lost Messages**
  - OS guarantees retransmission
  - sender is responsible for detecting it using timeouts
  - sender gets an exception

- **Scrambled Messages**
  - Message arrives from sender P to receiver Q, but information in message is corrupted due to noise in communication channel.
  - Solution
    - need error detection mechanism, e.g. CHECKSUM
    - need error correction mechanism, e.g. retransmission
Producer-Consumer Problem

- Paradigm for cooperating processes;
  - producer process produces information that is consumed by a consumer process.
- We need buffer of items that can be filled by producer and emptied by consumer.
  - Unbounded-buffer places no practical limit on the size of the buffer. Consumer may wait, producer never waits.
  - Bounded-buffer assumes that there is a fixed buffer size. Consumer waits for new item, producer waits if buffer is full.
- Producer and Consumer must synchronize.
Producer-Consumer Problem

**PRODUCER**

1. **continue**
   - yes
   - Buffer Full
     - yes: Suspend
     - no: Produce Output
   - no

**CONSUMER**

1. **continue**
   - yes
   - Buffer Empty
     - yes: Resume
     - no: Print Output
   - no
Bounded Buffer using IPC (messaging)

- **Producer**
  
  ```
  repeat
  ...
  produce an item in nextp;
  ...
  send(consumer, nextp);
  until false;
  ```

- **Consumer**
  
  ```
  repeat
  receive(producer, nextc);
  ...
  consume item from nextc;
  ...
  until false;
  ```
Bounded-buffer - Shared Memory
Solution

- Shared data

```plaintext
var n;

type item = ....;

var buffer: array[0..n-1] of item;

in, out: 0..n-1;

in := 0; out := 0; /* shared buffer = circular array */

/* Buffer empty if in == out */

/* Buffer full if (in+1) mod n == out */

/* noop means ‘do nothing’ */
```
### Bounded Buffer - Shared Memory Solution

- **Producer process - creates filled buffers**

```plaintext
repeat
  ...
  produce an item in nextp
  ...
  while in+1 mod n = out do noop;
  buffer[in] := nextp;
  in := in+1 mod n;
until false;
```
Bounded Buffer - Shared Memory Solution

- **Consumer process - Empties filled buffers**

  repeat
  
  while \( in = out \) do \textbf{noop};
  
  \( \text{nextc} := \text{buffer}[\text{out}] \);
  
  \( \text{out} := \text{out} + 1 \mod n; \)
  
  …
  
  consume the next item in \( \text{nextc} \)
  
  …
  
  \textbf{until} \( false \)