03. Processes
Ch. 1.6, 3
Objectives

• To understand the concept of a process vs a program, and the need for context switching
• To distinguish the states in a process’ lifecycle
• To know some of the state required for process management
Outline

• What is a process?
• Process lifecycle
• Inter-Process Communication (IPC)
Outline

• What is a process?
  • Process Control Block (PCB)
  • Threads of execution
  • Context switching

• Process lifecycle

• Inter-Process Communication (IPC)
What is a process?

• The computer is there to execute programs, not the OS!
• Process ≠ Program
  • A **program** is static, on-disk
  • A **process** is dynamic, a program in execution
  • On a batch system, one might refer to jobs instead of processes – nowadays generally used interchangeably
• Process is the **unit of protection** and **resource allocation**
  • So you may have multiple running processes created from a single program
What is a process?

• Each process executed on a virtual processor has
  • **Text** containing the program code
  • **Data** containing global variables
  • **Heap** containing memory allocating during runtime
  • ...plus one or more **threads of execution**

• Each thread has
  • **Program counter** indicating current instruction
  • **Stack** for temporary variables, parameters, return addresses, etc.
Process Control Block (PCB)

- Data structure representing a process, containing
  - **Process ID or number** – uniquely identifies the process
  - **Current process state** – running, waiting, etc
  - **CPU scheduling information** – priorities, scheduling queue pointers
  - **Memory-management information** – memory allocated to the process
  - **Accounting information** – CPU used, clock time elapsed since start, time limits
  - **I/O status information** – I/O devices allocated to process, list of open files

- Highlighted **process context** is the machine environment while the process is running
  - **Program counter**, location of next instruction to execute
  - **CPU registers**, contents of all process-centric registers
Threads of execution

• A **thread** represents an individual execution context
  • One process may have many threads
  • OS visible threads are **kernel threads**, whether executing in kernel or user space

• Each thread has an associated **Thread Control Block (TCB)**
  • Contains thread metadata: saved context (registers, including stack pointer), scheduler info, program counter, etc.

• A scheduler determines which thread to run
  • Changing the running thread involves a **context switch**
  • If between threads in different processes, the process state also switches
Context switching

• Switching between processes means
  • Saving the context of the currently executing process (if any), and
  • Restoring the context of the process being resumed

• Wasted time! No useful work is carried out while switching

• How much time depends on hardware support
  • From nothing, to
  • Save/load multiple registers to/from memory, to
  • Complete hardware “task switch”
Outline

• What is a process?
• Process lifecycle
  • Process states
  • Process creation
  • Process termination
• Inter-Process Communication (IPC)
Process states

- **New**: process is being created
- **Ready**: process is ready to run, and is waiting for the CPU
- **Running**: process’ instructions are being executed on the CPU
- **Waiting (Blocked)**: process has stopped executing, and is waiting for an event to occur
- **Terminated (Exit)**: process has finished executing
Process creation

• Most systems are hierarchical
  • Parent processes create child processes
  • Forms a tree
• E.g., a possible Linux process tree
Process creation

• How are resources shared?
  1. Parent and children share all resources
  2. Children share subset of parent’s resources
  3. Parent and child share no resources

• How is the child’s memory initialised?
  1. Child starts with a duplicate of the parent and then modifies it
  2. Child explicitly has a program loaded into it

• How is execution of parent and children handled?
  1. Parent and children execute concurrently
  2. Parent waits until children terminate
Process creation

• E.g., on Unix
  • `fork` clones a child process from parent,
  • then `execve` replaces child’s memory space with a new program,
  • meanwhile parent `waits` until child `exits`

• Alternative approach in NT/2K/XP
  • `CreateProcess` explicitly includes name of program to be executed

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        execvp("/bin/ls", "ls", NULL);
    }
    else { /* parent process */
        /* parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete\n");
    }

    return 0;
}
```
Process termination

1. Process performs an illegal operation, e.g.,
   • Makes an attempt to access memory without authorisation
   • Attempts to execute a privileged instruction

2. Parent terminates child (abort, kill), e.g. because
   • Child has exceeded allocated resources
   • Task assigned to child is no longer required
   • Cascading termination – parent is exiting and OS requires children must also exit

3. Process executes last statement and asks the OS to delete it (exit)
   • Parent waits and obtains status data from child
   • If parent didn’t wait, process is a zombie
   • If parent terminated without waiting, process is an orphan
Outline

• What is a process?
• Process lifecycle

• Inter-Process Communication (IPC)
  • Message passing vs Shared memory
  • Signals
  • Pipes
  • Shared memory segments
Inter-Process Communication (IPC)

• All communications require some protocol, with data transfer
  • ...in a commonly-understood format (syntax)
  • ...having mutually-agreed meaning (semantics)
  • ...taking place according to agree rules (synchronisation)
  • (Ignore problems of discovery, identification, errors, etc. for now)

• Communication between hosts is IB Computer Networking
  • Separate hosts means handling reliability and asynchrony

• Communication between threads is IB Concurrent & Distributed Systems
  • Shared data structures can suffer corruption, deadlock, etc.

• IPC basic requirement: access to shared memory on same host
Message passing vs Shared memory

• Two fundamental models for IPC

• **Shared memory**
  • Communicating processes establish some part of memory both can access
  • Requires removing usual restriction that processes have memory protection

• **Message passing**
  • Processes send messages to each other mediated by the kernel
  • Requires support for processes to
    • name each other or a shared mailbox (direct vs indirect communication)
    • send and receive synchronously or asynchronously (blocking vs non-blocking)
    • buffer messages to match rates if non-blocking (zero, finite, unbounded buffers)
Message passing vs Shared memory

- **Message passing**: Each process (A and B) has its own memory and communicates through a message queue. The message queue stores messages such as $m_0, m_1, m_2, m_3, \ldots, m_n$.

- **Shared memory**: Both processes share a common memory space. Access to this memory is managed by the kernel.
Signals

- Simple message passing: asynchronous notifications on another process
  - `kill` system call sends a signal to a specified process/es
  - `sigaction` examines or changes a signal handler disposition (terminate, ignore, etc)
  - `pause` suspends process until signal is caught

- Each signal mapped to an integer, different between architectures

- Among the more commonly encountered:
  - SIGHUP: hangup detected on terminal / death of controlling process (1)
  - SIGINT: terminal interrupt (2)
  - SIGILL: illegal instruction (4)
  - SIGKILL: terminate the process [cannot be caught or ignored] (9)
  - SIGTERM: politely terminate process (15)
  - SIGSEGV: segmentation fault (11) — process made an invalid memory reference
  - SIGUSR1/2: two user defined signals [system defined numbers]
Pipes

• Simple form of shared memory IPC
  • pipe returns a pair of file descriptors, (fd[0], fd[1])
  • fork creates child process

• Parent and child can now communicate
  • read/write on the pair of (read, write) fds

• Named pipes (FIFOs) extend beyond parent/child relation
  • Appear as files in the filesystem
Shared memory segments

• Obtain a segment of memory shared between two (or more) processes
  • `shmget` to get a segment
  • `shmat` to attach to it

• Simply read and write via pointers into the shared memory segment
  • Need to impose controls to avoid collisions when simultaneously reading and writing

• When finished,
  • `shmdt` to detach, and
  • `shmctl` to destroy once you know no-one still using it
Summary

• What is a process?
  • Process Control Block (PCB)
  • Threads of execution
  • Context switching

• Process lifecycle
  • Process states
  • Process creation
  • Process termination

• Inter-Process Communication (IPC)
  • Message passing vs Shared memory
  • Signals
  • Pipes
  • Shared memory segments