• Introduction
  o Processes are a key concept in operating systems
    ▪ Abstraction of a running program
    ▪ Contains all information necessary to run
  o On modern systems, many processes are active at the same time (multiprogramming)
    ▪ How do we time-multiplex the CPU(s)?
  o A solid understand of processes is key to several later concepts
• Process Model – method for keeping track of multiple programs running at the same time
  o All runnable software organized into a set of sequential processes
    ▪ Shortened to a process
  o Process – instance of an executing program
    ▪ All current values of program counter, registers, and variables
    ▪ Program counter – address of next instruction to be executed
    ▪ Each process has its own virtual CPU
  o Conceptually
    ▪ View the collection of processes are running in parallel independently

  o Reality
    ▪ The CPU switches between each process rapidly (multiprogramming)

• Reality
  ▪ Only one program is active at a given time
Multicore CPUs will each run a process at a given time, so that complicates things even further

Implications
- With multiprogramming, the amount of time a process is on the CPU will vary
  - Consequence: timing should never be built in to a process
  - Example: Using an idle loop to wait until some event has occurred
    - Could get kicked off the CPU and mess up the timing
- Processes must be scheduled in some way
  - We’ll study scheduling algorithms later (after threads)
- If a program is “started” multiple times, it is run in two distinct processes

Manipulating Processes
- Creation - The operating system needs some method for creating processes
- Three principle events cause process creation
  - System initialization
  - Execution of a process creation system call by a running process
  - User request to create a new process
- Technicality
  - All processes are created by having an existing process execute a system call
  - fork (Linux) or CreateProcess (Windows)
- System Initialization
  - Several processes of different types are created
  - Foreground – interact with humans and perform work for them
    - E.g., shell or GUI
  - Background – programs that do not directly interact with users and perform a specific function
    - Email, news, serving webpages, etc.
    - A.k.a. daemons
  - Any process can be run in the background in Linux
    - Example: sort a.txt > b.txt &
- Effect of process creation
- Parent and child processes have distinct address spaces
- No memory is shared between the two
- Other resources might be shared (e.g., open files)

- Implications
  - OS must create all internal infrastructure for new process
  - New memory must be allocated
  - Will possibly need some method for communication between parent and child

- System Calls for Creating Processes
  - `pid = fork()` – Create a child process identical to the parent
    - Returns process ID (zero in child and PID in parent)
    - Copies all file descriptors, registers, etc.
    - The address space is identical right after fork, but any changes will not affect the other
    - We usually want to execute different code
  - `s = execve(name, argv, environp)` – Replace a process’s core image
    - name – name of the file to be executed
    - argv – pointer to an argument array (command line)
    - environp – pointer to environment array (name=value strings for home directory, path, etc.)
  - `pid = waitpid(pid, &statloc, options)` – wait for a child to terminate
    - pid – specific process or -1 for any
    - &statloc – after return will contain child’s exit status
    - options – changes behavior
      - return immediately
      - return if stopped
    - returns PID of process that caused return or -1 if error

- Example shell

```c
#define TRUE 1

while (TRUE) {
    type_prompt();
    read_command(command, parameters);
    /* repeat forever */
    /* display prompt on the screen */
    /* read input from terminal */

    if (fork() != 0) {
        /* fork off child process */
        /* Parent code. */
        waitpid(-1, &status, 0);
        /* wait for child to exit */
    } else {
        /* Child code. */
        execve(command, parameters, 0);
        /* execute command */
    }
}
```

- Termination – the OS needs some way to get rid of processes that are no longer needed
- Four situations
- Normal exit - process terminates because the work is done
  - Voluntary
  - Compiler exit when finished
  - Voluntary exit: closing web browser
- Error exit – process discovers a fatal error
  - Voluntary
  - Due to bad parameters, etc.
  - Controlled by the process itself
- Fatal error – error caused by the process (due to program bug)
  - Involuntary
  - Illegal instruction, dividing by zero, bad memory access, etc.
  - Can sometimes be intercepted by the process and handled internally
- Killed by another process with authorization
  - involuntary
  - tells OS to terminate

  ○ Effect of termination
    - Process is removed from internal infrastructure
    - If fatal error, core dump is often produced
  ○ System calls for terminating processes
    - exit(status) – Terminate process execution and return status
      - return EXIT_SUCCESS to indicate “normal exit”
      - return EXIT_FAILURE to indicate “error exit”
      - can also return different non-negative values to indicate error
    - s = kill(pid, signal) – send a signal to a process
      - pid – process ID to send signal to
      - signal – signal to send to the process
      - Returns zero if signal is successfully sent
    - Examples
      - SIGKILL – 9 (kill that can’t be ignored)
      - SIGINT – 2 (terminal interrupt ctrl-c)
      - SIGFPE – 8 (floating point exception)
      - SIGSEGV – 11 (invalid memory segment access)

- Process States
  ○ Processes can be in three different states
    - Running – actually using the CPU at that moment
    - Ready – runnable but temporarily stopped so another process can run
      - Technicality of the system where the CPU must be shared
      - Main cause of unpredictability in timing for multiprogrammed systems
    - Blocked – unable to run until some external event happens
      - Example: sort a.txt | uniq > b.txt
Both “sort” and “uniq” are started
While “sort” is working, “uniq” will have nothing to do as it is waiting for input from “sort”
• When blocked, the process cannot be run even if the CPU has nothing to do
  o Ex: reading from file, reading from network, waiting on a child

Four transitions are possible between these states

1) Process blocks for input – happens when OS realizes that the process can’t continue
2) Scheduler picks another process
   • Caused by the process scheduler without knowledge of the process itself
   • When scheduler feels the process has had enough access to the CPU for now
3) Scheduler picks this process
   • Caused by process scheduler without knowledge of the process itself
   • When scheduler feels other processes have had enough time
4) Input becomes available
   • Happens when external event that caused the process to be blocked occurs
   • Could immediately transition to “running” if nothing on the CPU
   • Could also wait for some time as “ready” if other processes are busy

Implications of processes being blocked
• Most processes alternate between bursts of CPU-intensive computation and I/O.
• To fully utilize the CPU, a number of processes are necessary
  • Queuing theory can be used to provide an analytical model of CPU utilization, but it’s outside the scope of this class
• We can try to design our code to minimize I/O waits
  • Batch disk reads/writes
  • Batch packet sends/receives
• Have enough processes running such that not all are blocked at the same time

• Implementing Processes – simple summary of how processes are implemented
  o The OS uses an array of structures to store information about processes
    ▪ Indexed by the PID
  o The struct for each process contains all the important information about the process
  o Examples
    ▪ Process management
      • Registers
      • Program counter – next instruction to be run
      • Program status word – register of process condition (user/kernel mode, etc.)
      • Stack pointer – top of stack that contains one frame for each procedure run
      • Process state
      • Priority – for scheduler
      • Process ID
      • Parent process
      • Time when started
      • CPU time consumed
      • Time of next alarm
    ▪ Memory management
      • Processes have three segments – text, data, and stack

```
Address (hex)

Stack
FFFF

Gap

Data

Text

0000
```

• Text: program code (fixed when process is created)
• Data: program variables
  o Allocated with malloc or C++ new
• Stack: grows automatically with each procedure call
  o Contains a frame with arguments, local variables, return values, etc.
• The process table contains information about current allocation, etc.
  o How processes are switched on the CPU
    ▪ I/O interrupt occurs or scheduler preempts a process
- All registers, etc. are saved to the process table
- Interrupt service runs
- Scheduler decides which process to run next
- New process information is loaded onto the CPU and it continues