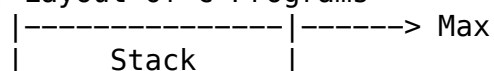


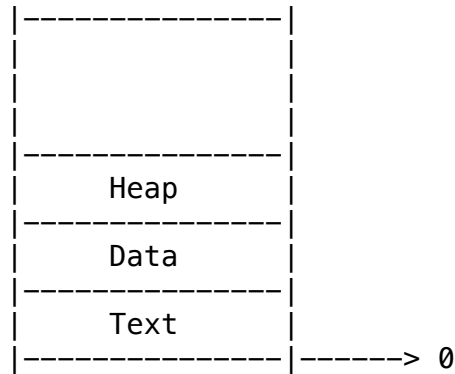
Lecture.4.Processes.txt

- Process Concept
 - In the early days of computing, processes were called jobs
 - Processes and jobs are synonymous
 - User programs run by time-shared systems are called tasks
 - Each time slot of shared time is called a task
 - Even if a computer executes only one program, the OS needs to support its own internal activities, such as concurrent activities and memory management
- What Is A Process?
 - Programs are static, and processes are dynamic
 - When you run a program, it becomes a process
 - Program in execution is the most frequently referenced one
 - A process is a program that is in execution
 - Process execution must progress in a sequential manner
 - In other words, each instruction is executed one at a time
 - This is referred to as atomicity
 - Is a process the same as a program?
 - A program becomes a process when an executable file is loaded into memory
 - A program can be several processes
 - i.e. Multiple users executing the same program
 - One process cannot be multiple programs, but one program can be multiple processes
 - Programs are passive entities stored on a disk
 - i.e. An executable file
 - Processes are active and running
 - Some processes can become the execution environment for other code
 - i.e. Virtual machine, Java program, etc.
- Address Space
 - Each process is associated with an address space
 - It is crucial for each process to get its own memory that is separate from other address spaces
 - Sharing memory between processes is bad because it can lead to an incorrect program, or the program may abort itself
 - The state of the program is stored in its respective address space
 - The state includes: Execution stack, system information, user data, etc.
 - The only memory a process can touch (read/write) is its own address space
 - But, a process can obtain a list containing all other running processes on the system
 - i.e. `ps -el` in the terminal on linux or Mac

- The benefit of using address space is protection
 - It ensures that a process can only access its own address space
 - i.e. Read and write denied to other process' memory
 - A process can itself be an execution environment for other code
 - i.e. Virtual machine, Java program, etc.
 - A process is represented by its Process Control Block (PCB)
 - PCBs contain:
 - Address space
 - Execution state
 - i.e. Program counter (PC), saved registers, etc.
 - When a process enters the system, it can go through a particular state
 - i.e. Running state, Waiting state, etc.
 - When a process is in the waiting state, the corresponding PCB is in a queue with other waiting processes
- Process Memory
- Processes have the following different kinds of memory associated to them:
 - Text
 - This is the executable code
 - The executable code is the instruction set
 - Data
 - This represents global variables
 - It is in the form of text
 - Heap
 - Dynamically allocated memory
 - This memory is allocated for variables during the execution of code
 - Stack
 - Located on top of all the other memory
 - Temporary data is pushed on top of the stack
 - The stack data structure is First In, First Out; commonly referred to as FIFO
 - When the OS switches from one process to another, all data associated with the first process is pushed on to the stack
 - When the process is resumed, its data is popped from the stack
 - Operating Systems try to give as much memory as possible to a program or process
 - A demanding program can set parameters that tell the OS that it needs more resources to perform computations
 - When main memory (RAM) gets full, the OS moves some of its memory to secondary storage, like the hard drive

- Memory Layout Of C Programs





- Data contains initialized and uninitialized data
 - Global data that is uninitialized is put under uninitialized data
 - i.e. `int x; float y; char a;`
 - The type does not matter
 - i.e. `int`, `float`, `array`, etc.
 - Global data that is initialized is put under initialized data
 - i.e. `float PI = 3.14; int SIZE = 5;`
 - The type does not matter
 - i.e. `int`, `float`, `array`, etc.
 - Text contains machine instructions
 - The stack contains local data
 - Local data is inside methods
 - The stack grows downward
 - The heap contains dynamically allocated data
 - i.e. `float[n] arr;`
 - The heap grows upward
 - The stack and heap grow toward each other
 - Must ensure that they do not overlap
 - Arguments for methods are stored at the top of the process memory hierarchy
-
- Process State
 - As a process executes, it changes state; from one state to another state
 - i.e. State A -> State B
 - All the states are at the same level, there is no hierarchy
 - Process states are:
 - New
 - The process was just created
 - Waiting
 - The process is waiting for an event to occur
 - Once the event occurs, the process can start running
 - Ready
 - The process has acquired all the resources, but the CPU
 - A process can only run during its given time slot
 - Running

- The process is running on the CPU
 - Finish
 - This process is exiting/terminating
- Processes switch from one state to another, all the time
 - This is controlled by the operating system
- Diagram Of Process States
 - An operating system sits idle until it is asked to do something
 - But the kernel runs all the time
 - Process Control Blocks (PCBs) are inside process states, and are organized in a queue (linked list)
 - This is CPU scheduling
 - When a process is in the ready state, the scheduler dispatches the process and initiates its execution
 - Before the OS stops a process, it needs to save the current state of it
 - This includes: program counter, registers, and everything else in the process control block
 - Only one process can be running on any processor core at any instant
 - However, many processes may be ready and waiting
 - The 'ready' and 'waiting' states contain several PCBs
 - The 'running' state only contains 1 PCB
 - If a process is in the running state and needs something from the I/O, the OS will push it into a waiting state
 - The operating system does this for efficiency
 - Since I/O is slow, the OS does not want to waste any CPU time
 - Note: A process cannot go back to 'running' from the 'waiting' state, it must go into the 'ready' state
- Process Control Block (PCB)
 - A process control block contains all information that uniquely defines the process
 - This information can be:
 - What is the process' priority?
 - This is scheduling information
 - How much CPU time the process needs?
 - This is accounting information
 - How many files the process opened?
 - And other miscellaneous information
 - This information helps us continue the process once it has been stopped
 - The PCB is stored in memory that is not accessible by the user
 - Only the OS can access it
 - The OS maintains a process table (a collection of all PCBs) to keep track of all the processes
- Process Representation In Linux
 - One way to represent processes is linked list

- Each node represents one structure
 - Each node can represent different information such as:
 - Process' parent
 - Process' children
 - List of open files
 - Address space of process
- Process Scheduling
 - Scheduling processes requires smart algorithms that can intelligently switch between processes based on a variety of different factors
 - i.e. Priority, CPU time, I/O, etc.
 - Process Scheduling helps maximize CPU time, by quickly switching processes onto the CPU core
 - The scheduler selects the next available process for execution on the CPU core
 - This is based on the scheduling algorithm
 - i.e. Prioritize applications that require the smallest amount of CPU time
 - i.e. First come, first serve
 - The process scheduler maintains a list (queue) of processes
 - There are two queues a process can be in:
 1. Ready queue
 - This is the set of all processes residing in main memory, ready and waiting to execute
 2. Wait queue
 - This is the set of processes waiting for an event to occur
 - i.e. I/O Event
 - Processes migrate among the various queues
- Ready & Wait Queues
 - The structure is First In, First Out (FIFO)
 - The process that enters first is the first one to be served
 - The order of the processes inside the queue are determined by the scheduling algorithm
 - As processes enter the system, they are put into a ready queue
 - The process waits there until it's selected for execution, or dispatched
 - Once the process has been selected for execution, the dispatcher steps in
 - The role of the dispatcher is to stop the previous process, and put new processes into execution
 - Processes that are waiting for a certain event to occur, such as completion of I/O, are placed in a wait queue
- Representation Of Process Scheduling
 - A process could be put into different types of wait queues
 - i.e. I/O wait queue, Child termination wait queue, Interrupt wait queue, etc.

- The process could issue an I/O request and then be placed in an I/O wait queue
- The process could create a new child process, and then be placed in a wait queue while it awaits the child's termination
- The process could be removed forcibly from the core, as a result of an interrupt or having its time slice expire, and be put back in the ready queue
- Dispatcher (1)
 - With numerous processes on the system, the OS must take care of:
 - Scheduling
 - Each process gets a fair share of CPU time
 - Protection:
 - Processes don't modify each other
 - i.e. Each process can only access its own address space
 - The job of the dispatcher is to take the process from the ready queue and load it into the CPU
 - Note: Ready queues are built by the scheduling algorithm; it also determines when the dispatcher needs to step in do its job
 - Example cycle of the dispatcher:
 1. Run process for a while
 2. Pick a process from the ready queue
 3. Save state (PC, registers, etc.)
 4. Load state of next process
 5. Run (load PC register)
- Dispatcher (2)
 - When a user process is switched out of the CPU, its state must be saved in its PCB
 - Otherwise, everything could be damaged by the next process
 - The information saved is:
 - Program counter
 - Processor status word
 - Registers
 - General purpose and floating-point
 - The dispatcher stores the state of the process in main memory (RAM), or the cache, depending on certain conditions
 - Note: It does not go to the hard drive (usually)
- CPU Switch From Process To Process
 - Switching the CPU core to another process requires performing a state save of the current process and a state restore of a different process
 - This task is known as a context switch
 - In other words: Stopping one process, saving its attributes, and then starting another process
 - The kernel saves the context of the old process in its PCB, and loads the saved context of the new process scheduled to run

- Note: PCB is stored in main memory (RAM) or cache
- Exceptions
 - The CPU can only run one process at a time
 - When a user process is running, the dispatcher is not running
 - Note: The dispatcher is part of the OS
 - The OS can regain control of the CPU through exceptions:
 - When an exception occurs, the user process gives up the CPU to the operating system
 - An exception is caused by internal events
 - i.e. A signal that says go to sleep
 - Types of exceptions
 - System call
 - Error
 - i.e. Bus error, Segmentation fault, Overflow, etc.
 - Page fault
 - Related to memory management
 - i.e. Trying to access a page that does not exist
 - Yield
 - Gives control to another process
 - In this case, exceptions are also called traps
- Interrupts
 - Some ways the OS can interrupt a user process are:
 - Completion of an input
 - i.e. Character(s) input(s) via keyboard
 - i.e. 'Return' key pressed on the keyboard
 - Completion of an output
 - i.e. Character(s) displayed at terminal
 - Completion of disk transfer
 - i.e. Copying/moving files
 - A packet is sent to the network
 - i.e. Downloading a file
 - Timer
 - i.e. Alarm clock
 - Interrupts are usually caused by external events
- Operations On Processes
 - Process creation
 - Process termination
- Process Creation (1)
 - Creating a process from scratch:
 1. Load code and data into memory
 2. Set up a stack
 3. Initialize PCB (process control block)
 4. Make process known to dispatcher

- Process Creation (2)
 - Forking is the process of making another child process from an existing one
 - Steps to forking a process:
 1. Make sure the parent process is not running and has its state saved
 2. Make a copy of the code, data, and stack
 3. Copy the parent process' PCB into the child process
 4. Make the child process known to the dispatcher
 - Even though the parent and child process look the same, they are different processes with their own PCB, address space, etc.
- Process Creation (3)
 - The parent process creates children processes, which, in turn creates other processes, forming a tree of processes
 - Generally, processes are identified and managed via process identifier (PID)
 - Resource sharing options between parent process and child process are:
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
 - Execution options between parent and child are:
 - Parent and children execute concurrently
 - Parent waits until children terminate
- A Tree Of Processes In Linux
 - The 'systemd' process serves as the root parent process for all user processes
 - It is only on Linux
 - 'systemd' will always have a PID of 1
 - On macOS it is 'launchd'
 - Note: macOS is based on Darwin
 - The 'systemd' process creates processes which provide additional services
 - The command `ps -el` lists running processes
 - This only works on Linux/macOS, and in the terminal
 - The command `pstree` shows a tree of processes
 - It shows the main process, 'systemd' and the sub-processes
 - This only works on Linux, and in the terminal
- Process Creation (4)
 - Address space
 - Child duplicate of parent
 - Child has a program loaded into it
 - UNIX examples
 - The 'fork()' system call creates a new process
 - If 'fork()' returns a PID of 0, then the child process was created

- The 'exec()' system call is used after a 'fork()' to replace the process' memory space with a new program
- The parent process calls 'wait()' for the child to terminate
- Example
 - UNIX 'fork()', 'exec()', and 'wait()'
 - The system call 'fork()' is called by one process, and a value is returned in two processes; the parent process and the child process
 - The parent process returns the PID of the child process
 - The child process returns 0
 - Sample Code
 - i.e.


```
pid = fork();
if (pid == 0) { // Child process
    exec("executable")
}
// Parent process continues
```
 - In the child process, executable overwrites the old program
 - Parent process calls 'wait()' for the child to terminate

- C Program Forking Separate Process

i.e.

```
/*
 * This code snippet forks a parent process,
 * and creates a child process
 *
 * When you fork a process, you create another
 * process that is identical, in terms of code,
 * but has a different PID
 */

#include <stdio.h>      // Includes 'printf()'
#include <stdlib.h>     // Includes 'wait()'
#include <sys/types.h>  // Includes 'fork()'
#include <unistd.h>     // Includes 'execlp()'

int main(void) {

    pid_t pid;

    pid = fork(); // Fork parent and create a child process

    if (pid < 0) { // Error occurred
        fprintf(stderr, "Fork failed");
        return 1;
    } else {
        if (pid == 0) { // Child process
            execlp("/bin/ls", "ls", NULL);
            //printf("Child process\n");
        }
    }
}
```

```

        } else { // Parent process
            /* Parent will wait for the child
             * to complete
             */
            wait(NULL); // Prevents child from turning into
                        // orphan process
            printf("Child complete\n"); // Printed after child
                                      // completes
        }
    }

    return 0;
}

```

- Process Termination (1)
 - A process terminates when it finishes the last statement, and calls 'exit()'
 - When a process is terminated, it:
 - Deallocates its memory (physical and virtual)
 - Closes open files
 - Notifies its parent process via 'wait()'
 - If the child process does not notify the parent process, then it can cause a problem called orphan process
 - An orphan process occurs when the parent process will finish and terminate, but the child process will continue to run, and occupy space in memory
- Process Termination (2)
 - A process can terminate itself, and other processes
 - i.e. A parent process can terminate a child process using the 'abort()' or 'kill()' system call
 - A child process can be terminated if:
 - The child process exceeds allocated resources
 - The task assigned to the child is no longer required
 - The parent process has terminated, the child process continues to run, but the operating system does not allow this
 - So, the OS terminates the child process
 - This is cascading termination
- Process Termination (3)
 - A zombie process occurs when a child process terminates, but its parent has not yet called 'wait()'
 - If a parent process terminates without invoking 'wait()', then its child process becomes an orphan (process)
 - Some operating systems do not allow a child process to exist if its parent process has terminated
 - Modern operating systems have cascading termination
 - This means that when a parent process is terminated, so are its children, grandchildren, etc.
 - The parent process may wait for termination of a child process

by using the 'wait()' system call

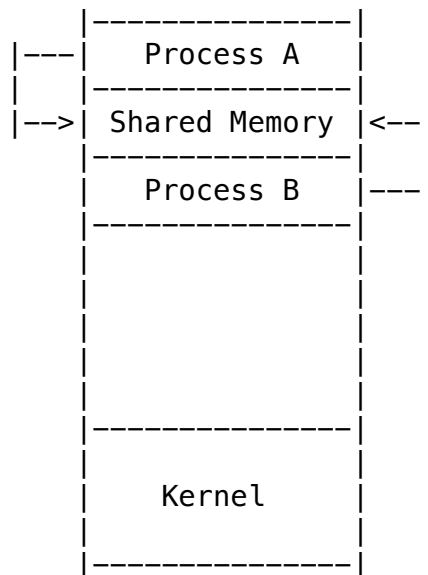
- The call returns status information and the PID of the terminated process
 - i.e. `pid = wait(&status);`

- Interprocess Communication

- Processes within a system may be independent or cooperating
- Cooperating processes can affect or be affected by other processes
 - i.e. Sharing data
- Reasons for cooperating processes:
 - Information Sharing
 - Computation Speedup
 - Modularity
 - Convenience
 - Parallelization
 - i.e. Process 'A' does one part of the computation, and Process 'B' does another part of the computation. Upon completion, they share their results
 - i.e. Dividing a huge list into two parts, and each process sorts a sub-list
- Cooperating processes need interprocess communication (IPC)

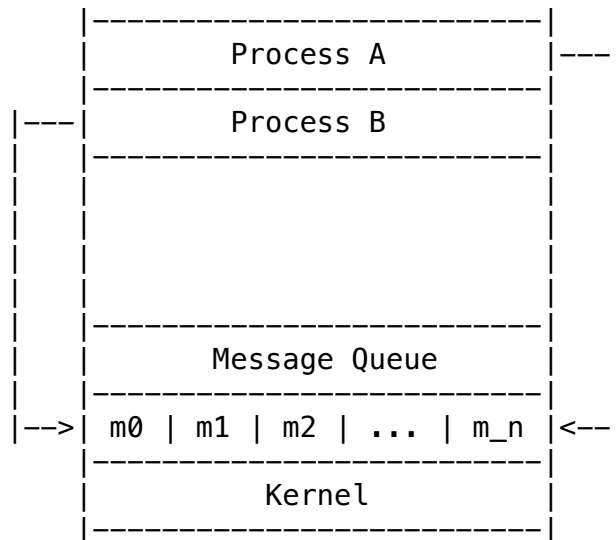
- Communication Models

- Two models of IPC:
 - Shared memory
 - i.e.



- Process A and process B use and share the same memory
- This model mimics the producer and consumer relationship
 - The producer writes data to shared memory, and the consumer uses it
 - The consumer needs to know the exact memory address

- The producer has read/write privileges, and the consumer only has read privileges
 - This is because the consumer will never change data in the shared memory
- Message passing
 - i.e.



- Messages are passed from one process to another
 - System calls are used to read and write data to the messages
- End
- Operating Systems are among the most complex pieces of software ever developed
 - The implications of orphan/zombie process:
 - Modern operating systems do not allow orphan processes due to cascading termination
 - A zombie process can have a serious impact on performance because it stays in memory for a long time and can hog CPU time
 - But, no security implications; unless the process intends to do something malicious