Computer Science **<u>3305</u>** – **<u>Operating Systems</u>** – Course Notes

__________________________ (**Lecture #1** – Course Introduction) ________________________

- No material to study from this lecture.

_________ (**Lecture #2** – OS Introduction History & Processes – p1 & Forks – p1) __________

- <u>Operating System</u> = the software layer between user applications and hardware, that manages hardware resources.

- *History of OS:*
  - 1st Gen – vacuum tubes and plug boards (direct input).
    - Ran one calculation at a time.
    - No languages or OS.
  - 2nd Gen – transistors (batch systems).
    - Programs written on paper in FORTRAN or assembly and encoded on punch cards.
  - 3d Gen – integrated circuits and multiprogramming.
    - Allowed for multiple calculations at the same time.
  - 4th and Current Gen – large scale integration and personal computers.
    - First OS offered by bill gates called DOS.
    - Doug Englehart at Stanford invented the GUI.
    - Jobs designed the first user friendly PC.
  - Next Gen – systems connected by high-speed networks.

- *History of Unix:*
  - <u>Multics</u> = first large timesharing system developed by MIT, General Electric and Bell Labs.
  - Ken Thompson found a computer that nobody was using, and wrote a stripped-down one user version of Multics in C, which became Unix.
  - <u>MINIX</u> = mini-Unix system developed by Andrew Tanenbaum.
  - A Finish student, Linus Torvalds wanted to add stuff until he eventually made his own OS called Linux.

- <u>Process</u> = an instance of a running application. As it runs it changes states.
  - New – process created.
  - Running – instructions being executed.
  - Waiting – process is waiting for an event to occur.
  - Terminated – process has finished execution.

- Process Control Block (PCB) = each process in the OS is represented by a block.
    - Context information would include:
        - Process identifier (PID).
        - Process state.
        - Program counter.
        - CPU registers.
        - CPU-Scheduling information.
        - Memory-Management information.
        - I/O status information.

- Fork() = a system call that creates a child process that is a duplicate of the parent.
    - Created by Unix.
    - Child inherits the state of its parent process, with the same instructions, variables, variable values and position.
    - The parent and child have separate copies of that state though.
    - If the call is SUCCESSFUL it returns the child PID to the parent, and returns 0 to the child.
    - If the call FAILS it returns -1 to the parent, and no child is created.

- Pid_t = data type that represents process identifiers.
    - You can call getpid() – returns the PID of calling process.
    - You can call getppid() – returns the PID of parent process.

(**Lecture #3** – Processes – p2 & Forks – p2)

- Processes get a share of the CPU to give other processes a turn. The switching between processes (like parent or child) depends on many factors, such as machine load or process scheduling.

- Nondeterministic = output interleaving, where you can't determine output by looking at code.

- Fork Example #3.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
/*
 This program forks a process.  The child and parent processes print to terminal to identify themselves
*/
int main ()
{
pid_t i, j, pid;
pid=fork();
if (pid <0) // fork unsuccessful
 {
        printf("fork unsuccessful");
        exit(1);
 }
if (pid>0) //parent
 {
        i = getpid();
        wait(NULL);
        printf("\n I am parent with PID %d\n", i);

 }
if (pid==0) //child
  {
        i = getpid();
        j = getppid();
        printf("\n I am a child with PID %d and my parent's PID is %d\n", i, j);
  }
}
```

- Fork Example #3.c – possible output:

Possibility #1	Possibility #2
PARENT 0 PARENT 1 PARENT 2 PARENT 3 PARENT 4 PARENT 5 PARENT 6 PARENT 7 PARENT 8 PARENT 9 CHILD 0 CHILD 1 CHILD 2 CHILD 3 CHILD 4 CHILD 5 CHILD 6 CHILD 7 CHILD 8 CHILD 9	PARENT 0 PARENT 1 PARENT 2 PARENT 3 PARENT 4 PARENT 5 PARENT 6 CHILD 0 CHILD 1 CHILD 2 PARENT 7 PARENT 8 PARENT 9 CHILD 3 CHILD 4 CHILD 5 CHILD 6 CHILD 7 CHILD 8 CHILD 9

- And many more possibilities as well!

- Amount of children created by forks = (2^#forks) – 1.

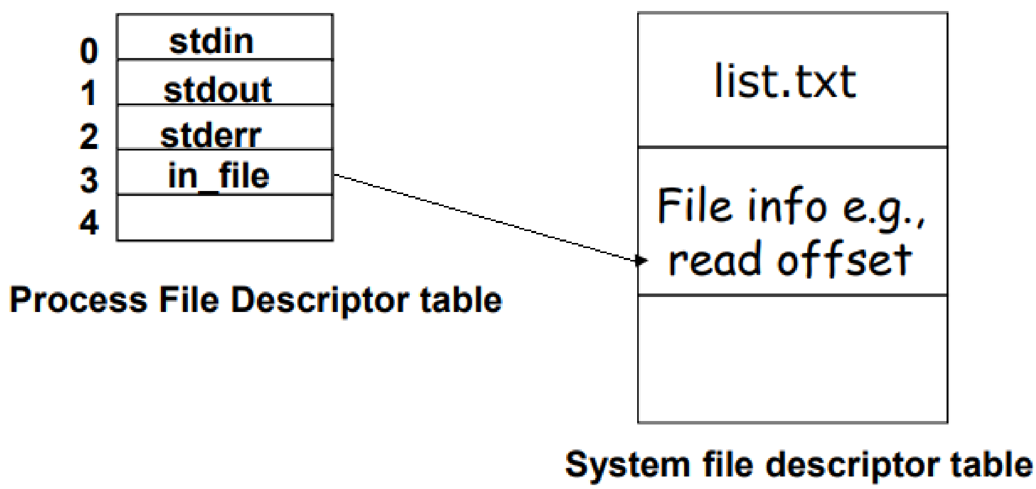
(**Lecture #4** – Processes – p3 & Forks – p3)

- <u>Process File Descriptor Table</u> = held by every process, where each entry represents something that can be read or written from (like stdin or stdout).

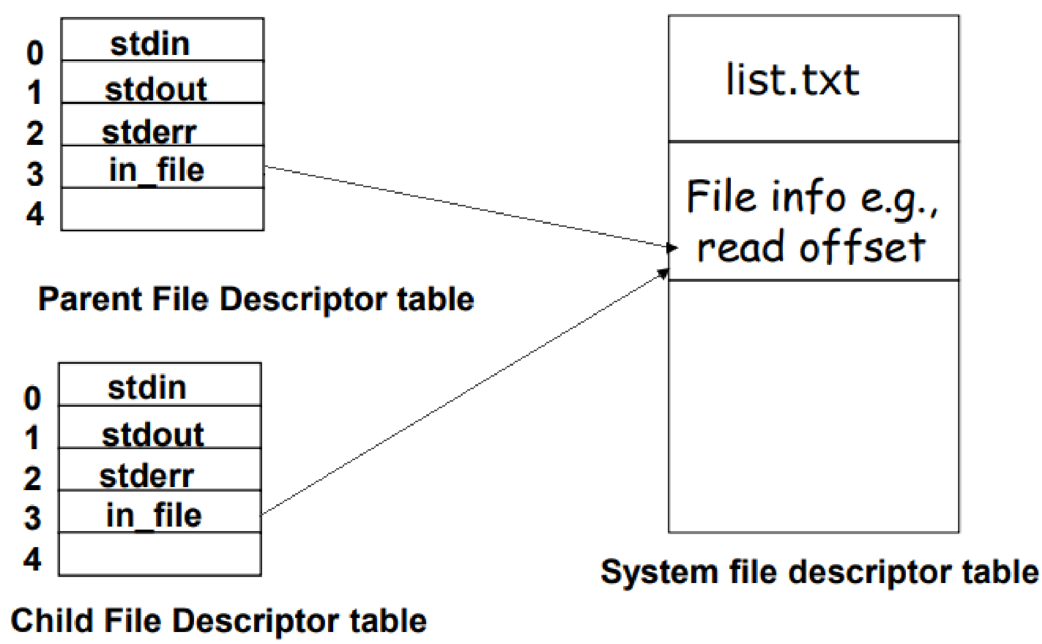
- <u>System File Descriptor Table</u> = every open file has one.

Example:

```
int in_file;
in_file = open("list.txt", O_RDONLY);
```



- Fork() = copies the process file descriptor table, meaning the parent and child point to the same entry in the system file descriptor table.



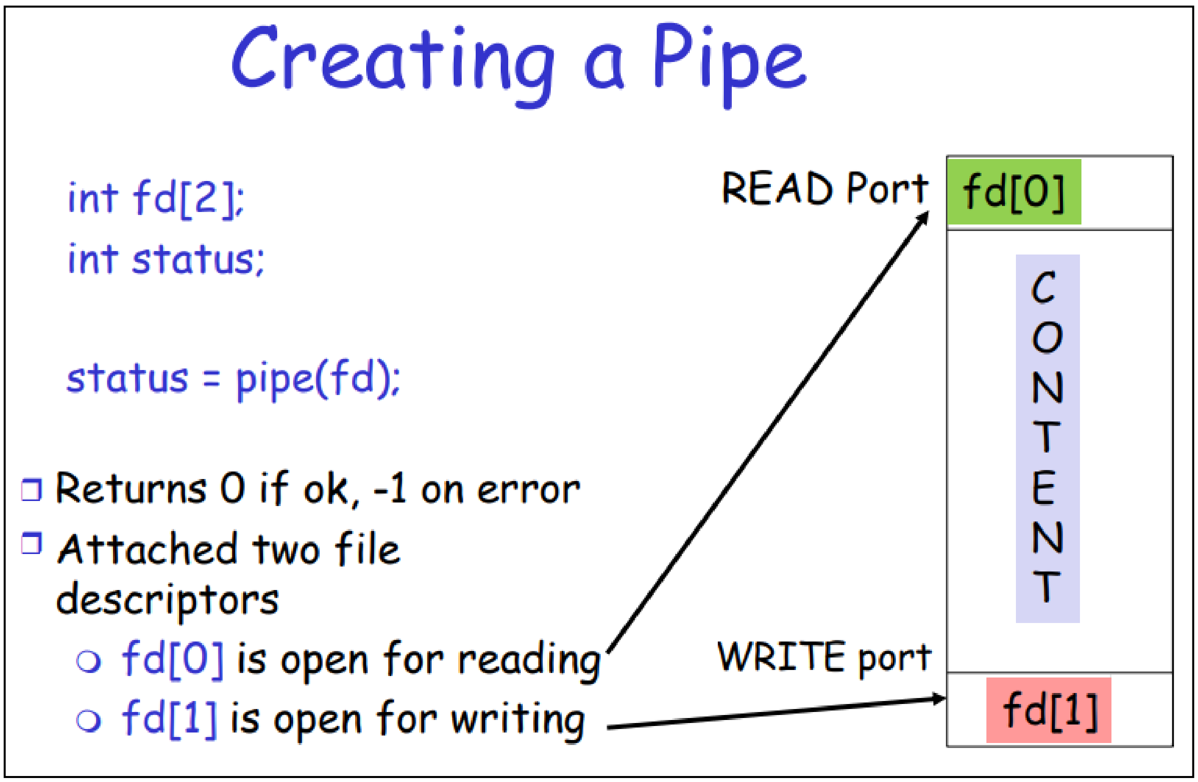
- If you open a file, then use fork() → the child process gets a copy of its parent's process file descriptor table, and the parent and child share a file pointer.
    - So if you read in "cat", the first char as parent reads is "c" and the child then reads "h".

- If you use fork() then open a file → the child and the parent process each open a file, and have their own entries in the system file descriptor table, and their own pointers.
    - So if you read in "cat", the first char as parent reads "c" and the child reads "c" as well.

- Recommended to open files after forks.

- Execl() = replaces a process with a new binary file loaded into memory. Only returns (-1) if it fails.
    - The Child will not return to the old program unless execl fails.
    - Specifically it stops running the parent (as it's destroyed), and removes the PCB. So the code before execl() is run, then execl(), and nothing else runs after it.

______________________ (**Lecture #5** – Processes – p4 & Pipes) ______________________



- Pipe won't allow you to read an empty pipe or write to a full pipe.

- Read(fd[0], &content, byte size);
    - printf("\n from child: this is what I read %c", content);

- Write(fd[1], content to write, byte size);

(**Lecture #6** – Signals & System Calls)

- Signal = notifies a process that an event has occurred, and the normal execution is interrupted.

- Signal() = captures a specific event and associates it with a programmer-defined function.
  - However you must include signal.h for this.

- SIGINT = interrupt signal from terminal (control-c).

- SIGALRM = instructs the OS to send an alarm.

  - Signal (SIGALRM, alarmHandler_Variable);
  - Alarm(3);

- All OS have an interface that is accessible by users/user programs and GUI's.

- System calls = provide an interface to OS services by passing relevant information to it.

- Application Programmer Interface (API) = where the programmer only needs to understand the system function through its parameters and results.

- System Function = used by the user / API programmer.
  - Some for process management (like pid or exit).
  - Some for file management (like fd, read, write).
  - Some for directory management (like mkdir or rmdir).

- System Call = part of the OS Kernel.

- System calls pass parameters as registers, blocks (used by Linux and Solaris) or stacks.

- Linux system calls with fewer than 6 parameters are passed in registers. Otherwise as blocks.

- The OS maintains a system call handler table which is indexed according to the system call numbers.

- TRAP = switches CPU to supervisor (kernel mode).
  - This saves the state of the user process so that the OS instructions can be executed and then the user process can execute after.

- What happens when you make a system call (in Linux)?
  - Step 1 – the input parameters are passed into register or to a block.
  - Step 2 – TRAP is executed.
    - Saving the user process as T.
    - System call number for the specific function is sent to the system call handler.

        - This call number tells the OS what system call handler (kernel code) to execute.
        - This causes a switch from the user mode to the kernel mode.
    - Step 3 – system call handler code executes (sometimes will have to wait for data from the disk to be slowly read in).
    - Step 4 – after execution, control returns to the system function.

____________________ (**Lecture #7** – Multiprogramming & Threading) ____________________

- Multiprogramming = allows for the execution of multiple processes, but only one active at a time.

- Interleaved Execution = allows for a process to be running at all times in order to maximize CPU utilization.

- Context Switching = when the OS suspends the current process so that another process can execute.
    - The OS must capture information about the process before it switches, so that it can restore the hardware to the same configuration after.

- Process Control Block (PCB) = represents each process in the OS.
    - Contains all of the states for a program, including (but not limited to):
        - Pointers.
        - Program counter (PC).
        - Current values.
        - Operating resources (open files and network connections).
        - Process identifier (PID).
        - Process priority.

- Process Execution States = an instance of a running application. As it runs it changes states.
    - New – process is being created.
    - Ready – process is waiting to be assigned to a processor.
    - Running – instructions being executed.
    - Waiting – process is waiting for an event to occur.
    - Exit – process has finished execution.

    - Note: only one process can be running on a processor at an instant, but many can be ready or waiting.

- Job Queue = contains all processes in the system, with each new process entering the system coming here.

- Ready Queue = contains all processes ready and waiting to be executed.

- Queues are implemented using a linked list.

- Fork is not efficient for multiprogramming as it copies everything.

- Threads:
  - Share code, data, open files and sockets.
  - But, they have their own CPU context with a program counter (PC), stack pointer (SP) and register state.

- <u>Pthread Library</u> = common thread library, implemented with pthread.h.
  - Each thread has a unique ID – called with pthread_self().
  - Thread ID's are of type pthread_t (typically an unsigned int).

pthread_create()

Creates a new thread

```
int pthread_create (
            pthread_t *thread,
            pthread_attr_t *attr,
            void * (*start_routine),
            void *arg);
```

- Returns 0 to indicate success, otherwise returns error code
- thread: name of the new thread
- attr: argument that specifies the attributes of the thread to be created (NULL = default attributes)
- start_routine: function to use as the start of the new thread
- arg: argument to pass to the new thread routine

pthread_create() example

Let us say that you want to create a thread that simply prints "hello world…I am a thread"

```
int main(int argc, char *argv) {
pthread_t worker_thread;
  if (pthread_create(&worker_thread, NULL,do_work, NULL)) {
    printf("Error while creating thread\n");
    exit(1);
  }
  ...
}
void *do_work() {
 Printf ("\n hello world..I am a thread");
 return NULL;
}
```

- Note: sharing global variables is dangerous as 2 threads may attempt to modify the same variable at the same time.
    - To solve this, use support for mutual exclusion primitives that can be used to protect against this problem.
    - The idea is to lock something before accessing any global variables, and to unlock when done.

________________________ (**Lecture #8** – CPU Scheduling – p1) ________________________

- Process Execution = consists of CPU execution and I/O wait/burst time.

- CPU/IO Burst Cycle = alternating between CPU bursts and I/O bursts, where the CPU must wait for I/O processes to occur in between running (which is the limited factor as the CPU is fast).

- *CPU Scheduler* = selects a process from the ready queue.
    - Non-preemptive process:
        - Runs until it relinquishes CPU control.
    - Preemptive process:
        - Runs for a maximum fixed time (set by a clock interrupt at the end time interval).

- Scheduling Evaluation Metrics = quantitative criteria for evaluating a scheduling algorithm.
    - CPU Utilization = percentage of time the CPU is being used.
    - Throughput = the number of completed process per time unit. #Cp's/T.
    - Waiting Time = time spent on the ready queue.
    - Turnaround Time = time from submission till completion of process.
    - Fairness = no process suffers starvation.

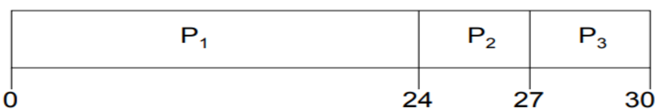
- *Scheduler Options:*
    - First Come, First Served (FCFS).
    - Last In First Out (LIFO).
    - Shortest Job First.
    - Priority Scheduling.
    - Round Robin (RR).
    - May use priorities to determine who runs next.
    - Multilevel Queuing.
    - Multilevel Queuing with Feedback.
    - Lottery Scheduling.

- First Come, First Served (FCFS) = the process that requests the CPU first is then allocated the CPU first.
    - Simple to write and understand.
    - However if the first process is the longest, the average wait time is far higher.

(FCFS) Scheduling Example

Process	CPU Burst Time
P1	24
P2	3
P3	3

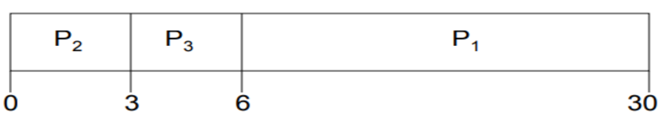
Assume processes serving request in the order: P1 , P2 , P3 The scheduling chart:



Waiting time for P_1 = 0; P_2 = 24; P_3 = 27
Average waiting time: (0 + 24 + 27)/3 = 17

Instead:

Suppose that the processes serving request in the order P_2 , P_3 , P_1
The Gantt chart for the schedule is:



Waiting time for P_1 = 6; P_2 = 0; P_3 = 3
Average waiting time: (6 + 0 + 3)/3 = 3
Much better than previous case

- <u>Last-In First-Out (LIFO):</u>
  - New processes are placed at the head of the queue.
  - However can lead to starvation.
  - <u>Starvation</u> = early process unable to ever get CPU use.

- <u>Shortest Job First (SJF):</u>
  - Estimated CPU burst time associated with each process.
  - Optimal process that gives minimum average waiting time for a given set of processes.

- Priority Scheduling:
  - A priority number (integer) is associated with each process.
  - The CPU then allocates the process with the highest priority (preemptive, or non-preemptive).
  - Problem – Starvation.
  - Solution – Aging.
  - Aging = as time progresses, increase the priority of the process.

- Round Robin (RR):
  - Each process gets a small unit of CPU time (time quantum (such as 10-100 milliseconds)), and after the time has elapsed the process is paused and added to the end of the ready queue.
  - With 'n' processes and 'q' time quantum, each process gets at most q time units at once.
  - Maximum wait time for the n^{th} process in the ready queue = (n-1)*q.
  - If 'q' is too large → RR becomes like FIFO.
  - If 'q' is too small → RR gets high context switching.

- Multilevel Queuing:
  - Used by most schedulers currently.
  - Makes the ready queue into multiple separate queues by classifying them based on scheduling.
  - Every queue has its own scheduling algorithm, such as RR with q of 5 or 8, or FIFO…etc.
  - Fixed priority scheduling occurs where all processes from a specific queue are executed before moving on to the next queue.
  - Problem → has a possibility of starvation.

- Multilevel Queuing with Feedback:
  - A process is able to move between queues.
  - Processes are separated according to their CPU bursts.
  - If a process uses too much CPU time it is moved to a lower-priority queue.
  - Includes aging for process in lower-priority queue.

- Lottery Scheduling:
  - Each process is given a random amount of lottery tickets.
  - To select the next process to run, the scheduler randomly selects a ticket and that process runs.
  - Solves starvation problem.