Lecture 1 - Shell part 1:

- **Process**: an instance of an application running
- Process is also known as a **text section**
- Process **information**:
  - **Process stack**: Includes temporary data(function parameters, return addresses and local variables.)
  - **Data section**: Include global variables
  - **Heap**: memory that is dynamically allocated during process run-time
- Process **changes states** on execution:
  - **New**: process is created
  - **Running**: instruction are being executed
  - **Waiting**: the process is waiting for some event
  - **Ready**: waiting to be assigned to a processor
  - **Terminated**: process has finished execution
- Processes are represented as **process control block** by the operating system. Which includes information related to the program execution:
  - Identifier (PID)
  - State
  - Counter
  - Registers
  - Scheduling
  - Memory-management
  - I/O status
- An operating system provides an environment for the program execution
- OS services:
  - **CLI** and **GUI**
  - Program execution
  - I/O operations
  - Resource allocation
  - Accounting
  - Protection and Security
- **Command interpreter** is a program that accepts commands from the user and interprets the commands, **shell** is an example.
- A **system call** allows a command to request a service from the operating system.
- Some **shells types**:
  - bourne shell
  - C shell
  - bash shell
  - tcsh shell
- The **fork()** system call creates a child process that is a duplicate of the parent.
  - Parent and child are **different** in the state, but not in the code variables or position

- **fork() failure** occurs when the limit of the processes that can be created is reached.
- **getpid()** returns the PID of the process, **getppid()** returns the PID of the parent of the process.
- Every process has a **process file descriptor table**
- OS has a system file descriptor table, saved in the OS memory space, and cotains:
    - Every open file file in the system: will have links directing to it. When links are reduced to zero, then the entry is removed.
- pid_t **wait**(int *status) return the PID of the exited process.
- pid_t **waitpid**(pid_t pid, int *status, int options)
- The term **exec** refers to a family of functions where each of the functions replace a process's program with a new loaded program
- A call to a function from **exec** loads a binary file into memory
    - execvp, execlp, execv, execve, execl, execle
    - execv**p**, execl**p**: search from the program in the PATH environment variable
    - exec**v**p, exec**v**, exec**v**e: accepts an array of arguments
    - exec**l**p, exec**l**, exec**l**e: accepts a list of arguments
    - execv**e**, execl**e**: accept an array of enviromental variables
- int **execl**(const char *path, const char *arg, ...);
- int **execlp**(const char *file, const char *arg, ...);
- int **execle**(const char *path, const char *arg , ..., char *const envp[]);
- int **execv**(const char *path, char *const argv[]);
- int **execvp**(const char *file, char *const argv[]);
- int **execve**(const char *filename, char *const argv [], char *const envp[]);

Lecture 2 - Shell part 2:

- **pipe()** provides the shared memory, returns 0 upon success and -1 upon failure
- **fd[0]** will always be the reading end of the pipe, and **fd[1]** will be the writing end
- Full pipes will block read and write calls until they are able to process data
- If read is called on an empy pipe, then the process will block until data is recieved, the process will also block when the reading end from a pipe is open but the write isn't.
- OS will have a limits for the pipe's buffer size
- int **dup**(int filedes1); duplicates filedes1 and returns the lowest available fd
- int **dup2**(int filedes1, int filedes2); duplicates filedes2 and closes it

Lecture 3 - System Calls:

- **System calls** provide an interface to OS services
    - The OS is able to do so
    - The service is permitted for this program at this time
- System function is a function in a library that invokes a system call
- System call code is part of the kernel
- **POSIX** - Portable Operating System Interface
- **TRAP** instruction switches CPU to supervise mode, also known as kernel mode

- CPUs have a **mode bit** in Process Status Word (PSW) register that defines execution capability of a program
    - **Supervisor(kernel) mode** - bit set
        - Executes **every instruction**
        - **OS** runs in this mode
    - **User mode -** bit cleared
        - Executes a **subset of instructions**
        - **User apps** run in this mode
- Every system call has a **call number** that is saved in the **system call handler** which is maintained in the OS
- When a command traps in the kernel the kernel usually calls **sys_command**(), the system call number fo that is **_NR_command**
- Methods used to **pass parameters** to the OS:
    - **Registers**: pass parameters in registers
    - **Block**: parameters are stored into memory blocks and the memory block address is passed into a register
    - **Stack**: parameters are push onto the stack by a program, popped by the OS
    - Block and stack approaches **do not limit** the number of passed parameters
- In Linux:
    - System calls with parameters **< 6** are passed to **registers**
    - **else**, store parameters using the **block** method
- **Making a system call:**
    - Push parameters onto the stack and pass parameters to registers
    - TRAP code of the command is placed into the register, execute TRAP instruction, and read the TRAP code register and map it to the system call handler using the system call table
    - Execute system call handler and return the system function
- **Polling**: keep polling the disk controller to see if your data is ready
- **Interrupt**: wait to signaled when the job you requested is complete
    - **Interrupt line** is a line between the interrupt handler and the devices
- **Interrupt handler:**
    - Saves processor state
    - Runs code to handle the interrupt process
    - Restores CPU state

Lecture 4 - Multiprogramming:

- **Multiprogramming** allows for the execution of multiple processes, however, only one process can be active at a time
- Operating systems allow for **interleaved execution**
- **Context switching**: suspend current process to run another
    - OS must store all the process information to be able to restore it
- In Linux
    - PCB is the tast_struct in C
    - task_struct include: pid, state, time_slice, siblings list, children list, files list, etc
- **Job queue** contains all the processes in the system
- **Ready queue** contains the processes that are loaded into memory and ready to go
- **Device queue** contains the processes that are waiting for I/O

Lecture 5 - Scheduling:

- Scheduling is **required** because of:
    - processes creation and termination
    - I/O blocks and interruptions
- **CPU–I/O Burst Cycle**: cycle of CPU execution + I/ O wait.
- Scheduling **occurs** when a process:
    - changes from running to waiting state or running
    - terminates
- When to **schedul**
    - **Non-preemptive** - blocks or process termination
    - **preemptive** - when a process finishes execution or timesout
- Scheduling **metrics**:
    - CPU utilization
    - Throughput
    - Turnaround time
    - Waiting time
    - Response time
    - predictability
- Scheduling algorithms may choose to alter the process' priorities either dynamically or statically
- **FCFS** scheduling: first come first serve
    - added to the tail of the ready queue
- **LIFO** scheduling: last in first out
    - added to the head of the ready queue
    - May lead to **starvation**, earlier jobs may never execute
- **SJF** scheduling: shortest job first
    - Approximates next CPU-brust duration, uses exponential averaging
- **Priority scheduling**:
    - CPU is allocated to the process with the highest priority

- SJF might also fall into the starvation problem. This can be solved by **aging**, as time progresses the priority of the process increases
- **Round Robin:** will give each process a time quantum, when the process timesout, it will be readd it to the end of the queue
    - time is equivalently distributed
- **Multilevel queue scheduling:**
    - **divides** up the ready queue into smaller queues
    - allows processes to be **classified** into different groups
    - each queue may implement their **own scheduling algorithm**
    - those queues will have fixed priority scheduling between them
    - system -> interactive -> interactive editing -> batch -> student
    - processes change queues
- **Lottery scheduling:**
    - scheduler randomly picks a number to decide who will run next

Lecture 6 - Physical Clocks

- Computers have **timer**, not clocks.
- A timer is a precisely machined quartz crystal oscillating at a frequency that depends on how the crystal was cut and the amount of tension
- Interrupts occur when the counter reaches 0, called a **clock tick**
- at each clock tick, the interrupt event adds 1 to the time in memory
- a timer can be programmed to generate n interrupts per second
- clocks can skew due to the use of the crystals

Lecture 6 - CPU Scheduling - Multicore

- **Asymmetric** multiprocessing:
    - one processor makes decisions for: scheduling, I/O, processing, system activities
    - other processes execute user code
- **Symmetric** Multiprocessing (SMP):
    - each processor is self scheduling
    - may share or have their own ready queues
- **Challenges**:
    - processor affinity: try to avoid migration, use the same processor for the same process
    - load balancing: keep the workload balanced on all processors

Lecture 7 - Threads

- A **thread** is a basic unit of CPU utilization
- processes have multiple threads that share memory but run independently
- Why not fork?
  - forking is time **consuming** and resource **intensive**
  - utilizes **shared** memory
- Threads **share**: process address space and OS space
- Threads have their **own** stacks and registers
- **Benefits** of threads:
  - responsiveness
  - resource sharing
  - context switching is faster
- fork() commands do not always fork all the threads, it depends on the system
- User-threads **map** to kernel threads
  - one to one: higher overhead, more concurrency
  - many to one: limits concurrency
- **Scheduling**: in Linux threads are treated the same way as processes, in other systems threads compete against other threads in the same process.
- Allows for more efficient use of multicomputing core
- **Single core**: process and threads interleave
- **Multi core**: allows for parallelization