Operating System Concepts & Linux programming

Explain regex commands and wildcard characters.

- Regex commands
  - grep: Regular Expression Parser (basic wildcards)
    - grep "pattern" file.txt
    - grep "pattern" -R dirpath
  - egrep: Extended Regular Expression Parser (extended wildcards)
    - egrep "pattern" file.txt
  - fgrep: Fixed Regular Expression Parser (no wildcards)
    - fgrep "word" file.txt
- Basic wildcards
  - `^`: find at beginning of the line
    - grep "^is" file.txt
  - `$`: find at end of the line
    - grep "is$" file.txt
  - `[a-z]`: find any one char from given set or range.
    - `[apx]`: a or p or x
    - `[a-z]`: any char from a to z
    - `[a-zA-Z0-9]`: any char from a-z, or A-Z, or 0-9
  - `.`: any single char
    - grep "b.g" file.txt
      - will match with any words like big, bag, beg, bug, b*g.
    - grep "b[.]g" file.txt
      - will match with word "b.g".
  - `*`: 0 or more occurrences of prev char
    - grep "wo*w" file.txt
      - will match with any words like ww, wow, woow, wooow, woooow, ...

        - grep "b.g" file.txt
            - will match with any words like big, bag, beg, bug, b*g.
        - grep "b*g" file.txt
            - will match with any words like g, bg, bbg, bbbg, bbbbg, ...
- Extended wildcards
    - `?`: 0 or one occurrence of prev char
        - egrep "wo?w" file.txt
            - will match with any words like ww or wow.
    - `+`: 1 or more occurrence of prev char
        - egrep "wo+w" file.txt
            - will match with any words like wow, woow, wooow, woooow, ...
    - `{n}`: exactly n occurrences of prev char
        - egrep "wo{3}w" file.txt
            - will match with word wooow
    - `{m,n}`: minimum m occurrences and maximum n occurrences
        - egrep "wo{2,4}w" file.txt
            - will match with any word like woow, wooow, woooow
    - `(word1|word2|word3)`: any one word/pattern
        - egrep "(dac|dmc|desd|dbda)" file.txt
            - will match with any one word from dac, dmc, desd, dbda
    - egrep "dac.*dmc" file.txt
        - will show all lines that contains dac as well as dmc word
- Examples:
    - Phone: egrep "^(0|+91)?[0-9]{10}$" numbers.txt
        - 9527331338
        - 09527331338
        - +919527331338

Which Linux command is used to kill all running instances of the same program.

- Kill command is used to send signal to the process.

    - Many of the signals do terminate the process e.g. SIGINT, SIGTERM, SIGKILL, SIGHUP, ...
- Commands: pkill, killall
    - pkill -SIGNAL program-name
- pkill -9 java
    - Kill all java processes
- pkill -9 chrome
    - Kill all chrome processes

How security is implemented in Linux file systems? Tell commands related to it.

- File permissions: read, write, execute
- File permissions/mode: user/owner level, group level, other level
    - (u)rwx (g)rwx (o)rwx
    - example: rwxr-xr-- (binary: 111 101 100) (octal: 754)
        - user: r w xhmod 0754 file.txt --> rwxr-xr--
    - chmod 0754 -R dirpath
        - group: r x
        - other: r
- Linux commands:
    - c --> change permissions of all files in directory
    - sudo chown sunbeam:root file.txt
        - change owner of file.txt to sunbeam user in root group.

How to use redirection and pipe in Linux commands?

- By default input is taken from the terminal and output is given to the terminal.
    - redirection: changing input/output from/to file (instead of terminal)
    - pipe: giving output to another command
- Input redirection: command < in.txt
    - wc < in.txt
- Output redirection: command > out.txt

    - ls > out.txt
- Error redirection: command 2> error.txt
    - wc -x 2> error.txt
    - 2> represent error redirection
        - 0 -- stdin
        - 1 -- stdout
        - 2 -- stderrs
- Pipe: command1 | command2
    - who | wc
        - output of who command is sent to wc command.
        - counts number of lines, words and characters in output of "who" command.

Explain OS booting.

- Power ON
- Base ROM firmware --> RAM
- POST -- check if all peripherals are working correctly.
- Bootstrap Loader -- find the bootable device (as per boot priority in BIOS)
- Bootloader --> Bootstrap Loader will start the Bootloader program from the bootable device.
    - Kept in second 512 bytes of the bootable partition/device.
    - e.g. GrUB, bootmgr, bootcamp.
    - Reads the config file (e.g. grub.cfg, bcd) and shows multiple options.
    - End user select any one option.
- Bootstrap Program --> Bootloader will start the Bootstrap program of the OS (selected by end user).
    - Specific to the OS version.
    - Kept in first 512 bytes of the bootable partition/device.
- Kernel --> Bootstrap programs loads OS kernel in RAM.

Explain Linux booting.

- Power ON

- Base ROM firmware --> RAM
- POST -- check if all peripherals are working correctly.
- Bootstrap Loader -- find the bootable device (as per boot priority in BIOS)
- GrUB Stage 1 -- In first 512 bytes of the bootable device.
    - Loads Stage 1.5
- GrUB Stage 1.5 -- Between MBR (Master Boot Record) and 1st sector (15-20 kb)
    - Contains basic FS drivers (ext3, ...)
    - Loads Stage 2 (from the Linux /boot partition)
- GrUB Stage 2 -- In /boot partition
    - Reads grub.cfg file and present the options to end user.
    - grub.cfg --> Linux OS entry
        - root partition e.g. (hd0,5)
        - linux /boot/vmlinuz.... root=...
        - initrd /boot/initrd....
    - As per user selection, it loads vmlinuz and initrd
- vmlinuz: Linux kernel (in zipped format)
    - When kernel loads in memory (by grub stage 2), extract itself.
    - Gets temporary file system from initrd.
- initrd: Initial RAM disk
    - Contains basic device drivers (including disk drivers)
    - Initial FS for Linux kernel, until the root fs from disk is accessible
- init/systemd process (pid=1)
    - The kernel start the user space process "init" or "systemd" from root fs.
    - Older Linux: init -- single processor/no parallel startup
    - Modern Linux: systemd -- multi-processing/services starts in parallel
- Linux boot sequence
    - man 7 bootup

How many Linux run-levels are there? Which features are enabled in each runlevel?

- Linux services are started step by step (in runlevels).

    - 1 - single-user mode (rescue mode)
    - 2 - multi-user mode (user login is possible, but no networking)
    - 3 - network mode (multi-user + networking -- usually used for servers (cli))
    - 4 - reserved
    - 5 - graphical mode (multi-user + networking + gui)
    - 6 - command> init 6 --> reboot
    - 0 - command> init 0 --> shutdown
- Homework: How to change the runlevels?

What is OS? What are its important functions?

- OS is intermediate between computer hardware and user programs.
- It is resource manager and control program.
- OS Functions
    - Process Management
    - CPU Scheduling
    - Memory Management
    - File & IO Management
    - Hardware Abstraction
    - User Interfacing
    - Networking
    - Security & Protection

Explain terms: Multi-programming, Multi-tasking, Multi-threading, Multi-processing and Mutli-user?

- Multi-programming
    - Loading multiple programs in main memory
    - Degree of Multi-programming: Max number of programs that can be loaded in main memory.
    - Done by Job scheduler
    - To increase CPU utilization
- Multi-tasking / Time-sharing

    - Sharing CPU time among all the tasks present in main memory and ready for executions
    - To decrease response time (< 1 sec)
    - Types: Process-based and Thread-based
    - Process-based Multi-tasking
        - Multiple independent processes are running concurrently
    - Thread-based Multi-tasking
        - Multiple threads (within process) are running concurrently
        - Also called as Multi-threading
- Multi-threading
- Multi-processing
    - Using multiple processors for executing application processes/threads
    - Also called as parallel systems
    - Types: Symmetric and Asymmetric
    - all modern OS do multi-processing e.g. Linux 2.6+, Windows Vista+
- Mutli-user
    - Multi-users can connect and execute multiple programs concurrently.

What is difference between process and thread? How can you create them in Linux program?

- P: Program under execution (with PCB)
- T: Light-weight process, Unit of execution (with TCB)
- P: Like a container that hold resources required for execution of the program.
- T: Unit of execution/scheduling which use the resources.
- P: Independent/isolated from each other.
- T: Threads in same process do share the resources.
- P: All sections are Independent.
- T: Only stack section is Independent, Other sections are shared.
- P: IPC is slower
- T: ITC is faster (within a process)
- P: Each process by default have single thread (main thread). Additional threads can be created.
- T: Main thread created by default. Others created programmatically.

```
ret = fork();   // create a new process (duplicating calling process)
if(ret == 0) {
    // child process
    err = execl("/new/program", ...); // loads a new program in calling (child) process addr space
}
else {
    // parent process
    wait(&s); // wait for child process to complete
}
```

```
void* thread_func(void *param) {
    // code to be executed by thread
}
int main() {
    pthread_t t1;
    pthread_create(&t1, NULL, thread_func, NULL);
    // ...
}
```

Explain process life cycle.

- Process state diagram
  - New
  - Ready
  - Running
  - Waiting/Blocked
  - Terminated

What are Linux IPC mechanisms? Explain any three with diagram.

- Linux IPC
  - Shared memory: Fastest IPC mechanism
    - Only in user space
  - Signals: Set of predefined signals
    - kill command
  - Message queue: Packet based data transfer
    - Bi-directional
    - Built-in sync (waiting)
  - Pipe: Stream based data transfer
    - Uni-directinal
    - Built-in sync (waiting)
    - Types: Unnamed (|) vs Named (fifo)
  - Socket: Commn on same or different machines
    - Bi-directional
    - Built-in sync (waiting)
    - Types: UNIX vs INET
    - UNIX socket: For commn on same machine
    - INET socket: For commn on same/different machine
      - INET socket = IP address + Port number
        - e.g. MySQL server socket = localhost:3306

Explain Linux kernel design (monolithic or modular).

- Monolithic: All functionalities in single binary image.
  - e.g. MS-DOS, BSD UNIX.
- Modular: Functionalities as a dynamically loadable modules.
  - e.g. Windows.

What is difference between semaphore and mutex? How can you create them in Linux program?

- Botn are synchronization mechanisms.

- Semaphore: Counter
  - Operations
    - P - wait op - decrement op
      - If count < 0, the current process is blocked.
    - V - signal op - increment op
      - If one/more processes are blocked, one of the process is resumed.
  - Applications
    - Counting -- Count the resources/processes (used in producer/consumer problem)
    - Mutual Exclusion -- count=1/0 (used to give resource access to only one process at a time)
    - Event/Flag -- A process wait for completion of some task in another process.
- Mutex: Like Binary Semaphore (for Mutual Exclusion)
  - Operations
    - Lock: When a process lock mutex, become owner of that mutex.
      - If already locked by other process, the current processs will be blocked.
    - Unlocks: Only process that locked it can unlock the mutex
      - If one/more processes are blocked, one of the process is resumed.
- Homework: Linux API

Semaphore vs Mutex vs Spinlock

- Semaphore - Counter (dec/inc)
- Mutex - Flag (lock/unlock)
- Spinlock - Flag (lock/unlock)
  - Hardware level synchronization mechanism
  - Uses bus locking instructions of architecture
    - e.g. SWP, LDREX, STREX
  - Usage

```
lock(spinlock);
// task
unlock(spinlock);
```

- Only one process can lock the spinlock and only that process will unlock it.
- If spinlock is already locked, the current process will busy wait (like infinite loop -- running state)
- Available only in kernel space.
- Psuedo code

```
lock = 0;

//lock op:
while(lock == 0);
lock = 1;

// unlock
lock = 0;
```

What is system call? How it is executed?

- System calls are functions exposed by the kernel so that user program invokes kernel functionality.
  - UNIX -- 64 syscalls
  - Linux -- 300+ syscalls
- System call execution
  - User program
  - Library function/System call wrapper --> Syscall No in Regr (r7) + Sw interrupt
  - Sw interrupt handler
  - Find syscall impl in SCT & call it
  - Sw interrupt handler returns

open() syscall

- fd = open("/file/path", flags, mode);

- 1. file path is translated into inode number (i.e. namei())
- 2. load inode from disk into in-memory inode table
- 3. create open file table entry that keep address of inode (via dentry)
- 4. keep address of OFT entry into OFDT of the current process
- 5. return index of OFDT entry (file descriptor)
- inode cache & dentry cache are kept so that time to access them from disk each time should be saved.

Current directory

- Linux commands --> "." represents current directory.
- current directory is specific to a process -- stored in fs_struct associated with task_struct of the current process.
- The current directory is used while processing relative path (in namei()).
- The current directory is changed using chdir() syscall.

dup() syscall

- Used in redirection and pipe.
- dup(fd);
  - Copies given fd on lowest numbered available file descriptor.
- dup2(fd, newfd);
  - Given newfd is closed (if not alread closed).
  - Copies given fd on newfd.

Copy-on-write

- Modern fork() creates logical copy of the calling process. Only the page tables of the calling process are copied.
- When any process (parent/child) try to modify any of the page, that page is copied and it's address+flags are updated into the the page table.

Virtual Page vs Logical Page

- Virtual page
  - Binding with physical page gets changed (due to swap in/out).
- Logical page

    - Binding with physical page do not changed (pages are not swapped)
    - e.g. most of kernel space pages
    - Pages after mlock(), mlockall()

vfork()

- vfork() -- virtual fork
- by BSD UNIX
- In UNIX, fork() does physical copy of the calling process -- slower.
    - Later when exec() is done, all resources allocated by fork() are released and new memory is allocated as per need of new program.
- In BSD UNIX, added vfork() -- same syntax as of fork().
    - Will create virtual copy (not real copy).
    - Also borrow thread of execution from the parent and execute the child code -- until exec() is called.
    - exec() will allocate the actual for new process (as per requirement) and create new thread of execution,
    - After exec() the parent & child runs independently.