Embedded Operating Systems

Agenda

- Disk scheduling algorithms
- Mounting
- Process life cycle
- Process creation

File modes - fopen()

- "w" --> O_WRONLY | O_CREAT | O_TRUNC
- "r" --> O_RDONLY
- "a" --> O_WRONLY | O_CREAT | O_APPEND
- "w+" --> O_RDWR | O_CREAT | O_TRUNC
- "r+" --> O_RDWR
- "a+" --> O_RDWR | O_CREAT | O_APPEND

Linux Ext2 FileSystems

- Disk allocation mechanism -- Indexed allocation -- Like UFS
  - inode keep array of 15 members to maintain information about data blocks.
    - 1-12 --> direct data blocks
    - 13 --> single indirect data blocks
    - 14 --> double indirect data blocks
    - 15 --> triple indirect data blocks
- Free space management mechanism -- Bit vector/map
  - nth bit into bit vector mapped to nth data block on the disk.
    - if bit = 0, corresponding block is free.
    - if bit = 1, corresponding block is used.
  - Similar mechanism is also used to maintain information about free inodes.
- Directory entry -- Stored in data blocks of Directory file -- Similar to UNIX
  - Each dentry contains inode number and name of the file.
- File System layout
  - File System = Boot block + Super block + Inode List + Data Blocks
  - Ext2/3 FS = Boot block + Block group 0 + Block group 1 + Block group 2 + ... + Block group n
  - Block group = Superblock + Group descriptor with Bitmaps + Inode List + Data blocks

Journaling

- For each file directory entry, inode and data block(s) are created. If any of these components is missing, file system will be inconsistent/corrupted.
- File System checks are for detecting and repairing file system inconsistencies.
  - Windows: chkdsk utility
  - Linux: fsck command

- However, fs checks are slower; because they needs to check all FS data structures like super block, inodes and directory entries.
- Journaling mechanims speed-up detecting and repairing file system inconsistencies.
- All file system transactions are written into journal file before actually peforming on disk/filesystem.
- If system crash/power cut during any of the transactions the operation remains pending in journal file.
- On next boot these transactions (from journal file) are verified and corrected. This speed-up repairing file system inconsistencies.
- During boot if journal file is empty, indicates that all pending operations are completed and file system is in consistent state.
- Journaling also speed-up file searching.

Disk Scheduling

Hard disk structure

- Time required to perform read/write operation on particular sector of the disk, is called as "disk access time".
- Disk access time includes two components = seek time and rotational latency.
- Seek time is time required to move head to desired cylinder (track).
- Rotational latency is time required to rotate the platters so that desired sector is reached to the head.

Disk Scheduling Algorithms

- When number of requests are pending for accessing disk cylinders, magnetic head is moved using certain algo. They are called as "disk scheduling algorithms".

FCFS

- Requests are handled in the order in which they arrived.

SSTF - Shortest Seek Time First

- Request of nearest (to current position of magnetic head) cylinder is handled first.

SCAN or Elevator

- Magnetic head keep moving from 0 to max cylinder and in reverse order continuously serving cylinder requests.

C-SCAN

- Magnetic head keep moving from 0 to max cyliner serving the requests and then jump back to 0 directly.

LOOK

- Implementation policy of SCAN or C-SCAN.
- If no requests pending magnetic head is stopped.

Mounting

- When CD/DVD or Pen drive is connected to the system it is auto-mounted under some directory /media/cd (in Linux), F:CD-DVD drive (in Windows).
- Internally the mounting is done by some system utility or command.
  - Windows: mountvol
  - Linux: mount
- Mounting enable us to access file system contents on a device/partition under another (current) file system.
- The mount command mounts given device partition to given mount point directory.
  - device partition e.g. /dev/sdb1, /dev/sr0, etc.
  - mount point is an empty directory under current file system. e.g. /mnt. After mounting, device contents will be visible under that directory.
  - fs type can also be specified while mounting. If not given, it will be auto-detection.
- mounting and unmounting

```
sudo fdisk -l
ls /mnt
sudo mount -t vfat /dev/sdb1 /mnt
ls /mnt
echo "Hello Linux!" | sudo tee /mnt/hello.txt
ls /mnt
cat /mnt/hello.txt
sudo umount /mnt
ls /mnt
mkdir ~/mydir
sudo mount -t vfat /dev/sdb1 ~/mydir
sudo umount ~/mydir
```

- mount command internally calls mount() system call. For each mount kernel keep information in a struct called "vfsmount".
- All disk all (configured) partitions are auto-mounted (mentioned in fstab) while booting. The partitions to be auto-mounted are configured in /etc/fstab file (in Linux).

```
terminal> cat /etc/fstab
terminal> man 5 fstab
terminal> mount | grep "sda"
```

UNIX Philosophy

- Files have spaces and processes have lives.
- UNIX architecture = File control subsystem + Process control subsystem.

Further readings

- Operating System Concepts - Galvin (book/slides) -- Process, Thread, Synchronization, Deadlock introduction
- Beginning Linux Programming - Neil -- Process & thread related syscalls
- Linux Programming Interface - Michel -- Process & thread related syscalls
- Design of UNIX OS - Bach -- IPC, Semaphore
- Professional Linux Kernel Architecture - Wolfgang -- Linux Process
- Linux Kernel Development - Love -- Linux process internals

Process

- Process is program in execution.
- Process has multiple sections i.e. text, data, rodata, heap, stack. ... into user space and its metadata is stored into kernel space in form of PCB struct.
- PCB contains pid, exit status, scheduling info (state, priority, time left, scheduling policy, ...), files info (current directory, root directory, open file descriptor table, ...), memory information (base & limit, segment table, or page table), ipc information (signals, ...), execution context, kernel stack, ...

Process Life Cycle

OS Data Structures:

- Job queue / Process table: PCBs of all processes in the system are maintained here.
- Ready queue: PCBs of all processes ready for the CPU execution and kept here.
- Waiting queue: Each IO device is associated with its waiting queue and processes waiting for that IO device will be kept in that queue.

Process states:

- Galvin: New, Ready, Running, Waiting, Terminated.

Process Creation

- System Calls
  - Windows: CreateProcess()
  - UNIX: fork()
  - BSD UNIX: fork(), vfork()
  - Linux: clone(), fork(), vfork()

fork() syscall

- To execute certain task concurrently we can create a new process (using fork() on UNIX).

```c
#include <stdio.h>
#include <unistd.h>
int main() {
    int ret;
    printf("start!\n");
    ret = fork();
    printf("return value: %d\n", ret);
    printf("end!\n");
    return 0;
}
```

```
terminal> man fork
```

- fork() creates a new process by duplicating calling process.
- The new process is called as "child process", while calling process is called as "parent process".
- "child" process is exact duplicate of the "parent" process except few points pid, parent pid, etc.
- pid = fork();
  - On success, fork() returns pid of the child to the parent process and 0 to the child process.
  - On failure, fork() returns -1 to the parent.
- Even if child is copy of the parent process, after its creation it is independent of parent and both these processes will be scheduled sepeately by the scheduler.
- Based on CPU time given for each process, both processes will execute concurrently.

Questions on fork()

- Write such a program in which if and else both blocks are executing.

```
if(fork() == 0) {
    // child process
}
else {
    // parent process
}
```

- How fork() return two values i.e. in parent and in child?
  - fork() creates new process by duplicating calling process.
  - The child process PCB & kernel stack is also copied from parent process. So child process has copy of execution context of the parent.
  - Now fork() write 0 in execution context (r0 register) of child process and child's pid into execution context (r0 register) of parent process.
  - When each process is scheduled, the execution context will be restored (by dispatcher) and r0 is return value of the function.
- getpid() vs getppid()
  - pid1 = getpid(); // returns pid of the current process
  - pid2 = getppid(); // returns pid of the parent of the current process

- Why child execute statements only after the fork() and not before that?
  - fork() creates new process by duplicating calling process.
  - The child process PCB & kernel stack is also copied from parent process. So child process has copy of execution context of the parent. Note that the execution context was stored due to software interrupt.
  - Now child PC will be the same address as of parent process. The PC always stores address of next instruction to be executed. So when fork() was called, the PC contains address of next instruction (i.e. instruction after software interrupt).
  - When parent is scheduled it start executing from that address and when child scheduled, it continues execution from the same address (i.e. after software interrupt).
- When fork() will fail?
  - When no new PCB can be allocated, then fork() will fail.
  - Linux has max process limit for the system and the user. When try to create more processes, fork() fails.
  - terminal> cat /proc/sys/kernel/pid_max