Embedded Operating Systems

Agenda

- Ext2/3
- Journaling
- Mounting
- Process

Ext2/3

- Similar to UFS
- Ext3 = Ext2 + Journaling

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
  - Try "debugfs"
    - terminal> sudo debugfs /dev/sdb1
    - debugfs> show_inode_info filename
    - debugfs> quit
- 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 Bit maps + Inode List + Data blocks
- To format a usb drive with Ext3
  - terminal> sudo fdisk -l /dev/sdb
  - terminal> sudo mkfs -t ext3 /dev/sdb1

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.

Ext4

- https://blogs.oracle.com/linux/post/mkfsext4-what-it-actually-creates

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
# -t fs type (vfat, ntfs, ext3, ...)
# device file -- /dev/sdb1, ... (sdb1 = sata disk "b" partition 1)
# mount point -- /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 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.

Next 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.

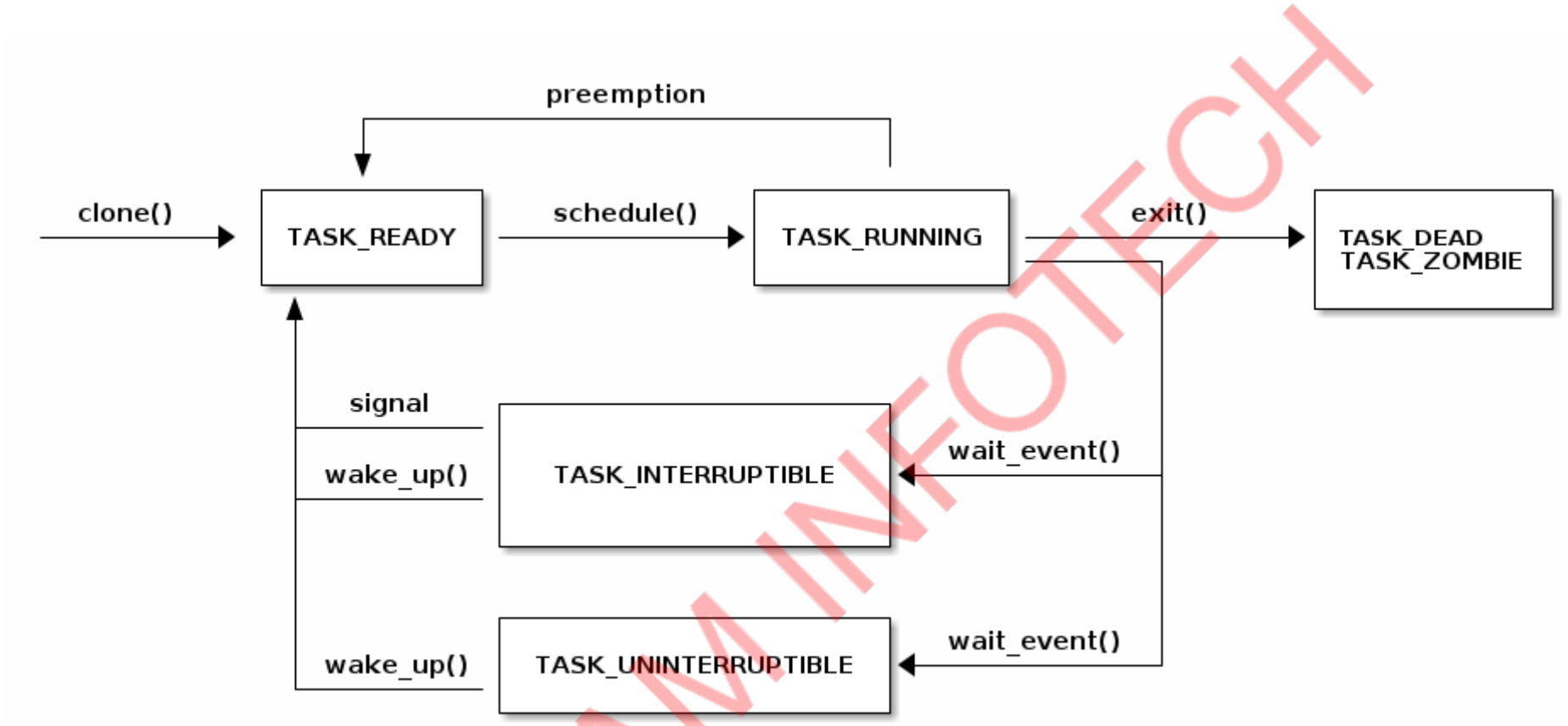
Process states

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

Linux process

- https://linux-kernel-labs.github.io/refs/heads/master/lectures/processes.html

Linux Process Life Cycle



-
- TASK_READY (R)
- TASK_RUNNING (R)
- TASK_INTERRUPTIBLE (S)
- TASK_UNINTERRUPTIBLE (D)
- TASK_STOPPED (T)
- TASK_ZOMBIE (Z)
- TASK_DEAD (X)

Process Creation

- System Calls
  - Windows: CreateProcess()

- Linux: clone(), fork(), vfork()
- BSD UNIX: fork(), vfork()
- UNIX: fork()

fork() syscall

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

```
#include <unistd.h>
pid_t fork(void);
```

  - fork() creates a new process by duplicating the calling process.
  - The new process is referred to as the child process. The calling process is referred to as the parent process.
  - The child process and the parent process run in separate memory spaces.
  - The child process is an exact duplicate of the parent process except for the following points:
    - The child has its own unique process ID
  - On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.
- Example:

```
#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;
}
```

- 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.

Assignments

1. Mount a pendrive. Write and read files from them. See mount information using "mount" command.
2. Format a pendrive with Ext3 filesystem (mkfs command). Mount it and write a few files in it. Observe block mapping in inode information using (debugfs command).