MAIN.ALGORITHMS.OF.A.LINUX.KERNEL

Signals

- Signals are one of the oldest facilities of Inter Process Communication.
- Signals are used to inform the processes about the events.
- signals will be sent via the following function (by the Kernel)

int send_sig_info(int sig, struct siginfo *info, struct task_struct *);

sig – refers the signal number

info – refers the sender

t – refers to the tasks (the kernel may send signals to many processes)

Booting the System

There are many bootloaders available for linux, the common ones being the LILO and the GRUB loader

LILO – LInux LOader

GRUB – GRand unified Bootloader

The steps while booting the kernel (only relevant steps are given)

- Entry point at ***start*** which is available at arch/x86/boot/setup.S **(This is responsible for initializing the hardware (assembler code)**
- Once the hardware is initialized, **the process is switched to protected mode by setting a bit word in the machine status word.**
- Next the assembler instruction, ***jmpi 0×100000 _KERNEL_CS*,** jumps to the start address of the 32 bit code of the actual operating system kernel and continues from ***startup_32*** and in the file **arch/x86/kernel/head.S .** More sections of the hardwares are initialized here like Memory Management unit (Page tables), the Co processor, and the environment (stack, environment,etc)
- The first C function **start_kernel()** from **init/main.c** is called
- the following list the assembly linkage of the start_kernel function

```
asmlinkage void __init start_kernel(void)

{

char * command_line;

printk(linux_banner); //print kernel, the banner

setup_arch(&command_line);//architecture dependent codes relevant to x86

trap_init();

init_IRQ(); //hardware interrupt initialization

sched_init(); //initialize the schedules

time_init();

softirq_init(); //soft interrupts

console_init();//initialize the console

init_modules();//initialize the modules (device drivers)

…..}
```

- the init is called (will be searched in **/sbin/init or /etc/init or /bin/init**). if the init is not available, then a shell (/bin/bash) will be opened for debugging

Hardware interrupts (IRQ)

Interrupts are used to allow the hardware to communicate with the operating system, there are two problems while writing interrupt routine,

- firstly, The interrupt routines should serve the hardware as quickly as possible

- secondly, large amount is to be handled by the interrupt routine

This can be solved by the following mechanisms

- disabling all the software interrupts while servicing the hardware interrupts.
- the processing of data is carried out asynchronously by the software interrupts through "**tasklets** " or **"bottom halves"**

Software Interrupts

- It is like a hardware interrupt but can be started only at certain times
- The number of interrupts is limited
- enum {HI_SOFTIRQ, NET_TX_SOFTIRQ, NET_RX_SOFTIRQ,TASKLET_SOFTIRQ}; the date types tells the software interrupts for hi priority software interrupts, Network Tranmssion and Receiving Interrupt and tasklet interrupt). upon interrupt is generated, the Interrupt routine will be executed

Timer Interrupts

- There is one hardware timer that generates interrupts every 10ms and all the software timer synchronizes with it.
- Usually the timer stored in the variable jiffies.

unsigned long volatile jiffies;

The variable jiffies is modified by the timer interrupt every 10ms and hence it is declared as volatile.

volatile struct timeval xtime:

This is the actual time which again modified by the timer interrupt

Other functions of timer interrupt like

do_timer();

updates the jiffies

timer_bh();

updates the timer and processing of the timer related functions

update_process_time();

collects data for the scheduler and decides whether it has to be scheduled.

The Scheduler

schedule () is function declared in **kernel/sched.c**

The actions of the scheduler is given below, once the schedule() function is called,

- Upcoming software interrupts are processed (so interrupts are given higher priority over the other entities in the system)
- process with highest priority determined (if two tasks has equal priority, then the OS will determine which task to be executed first.)
- real time process takes over normal ones. (Real time processes area associated with deadlines, whereas the normal ones doesn't have deadlines, this factor is determined by the **rt_priority** of the schedule structure)
- new process becomes current process. (whenever a process is getting scheduled by the scheduler, then it will become the current process)