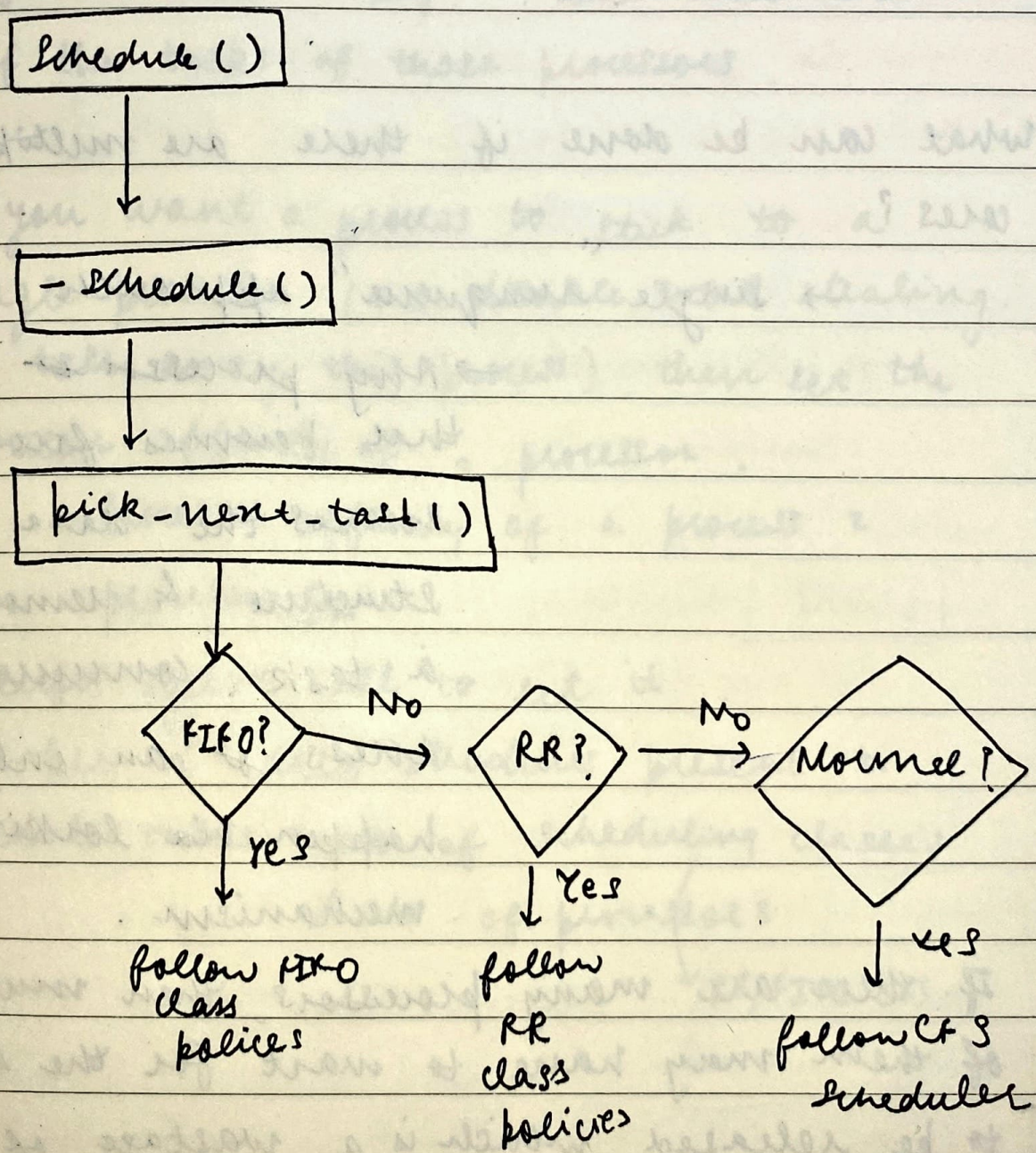
How exactly are processes scheduled across classes?

There is a priority among classes. If a process w higher "class priority" is present, that process will always be selected.



FIFO, RR classes have their own linked lists; their processes are not kept in the red-black tree.

CFS eliminates the need to separately classify processes based on interactive & non-interactive.

What can be done if there are multiple cores?

→ Single "runqueue" approach

→ Any processor that becomes free accesses the data structure & removes a task. Concurrent access can only happen via locking mechanism.

If there are many processors, then multiple of them may have to wait for the lock to be released, which is a wastage of processor time.

Cache effect

→ Individual "runqueue" approach →
Each processor has it's own set of processors, and the tasks are distributed among the processors.
An ideal processor would check the runqueues of other processors, if found non-empty, it will take some of the tasks of those processors.

If you want a process to stick to a single processor (i.e. disable work stealing to take away this process), then set the processor affinity of a processor.

Processor affinity of a process = processor #

Specific system call to set it.
Number of data structures present to schedule tasks = No. of scheduling classes × No. of processors
↳ RR, FIFO etc.

10.08.2022, Monday

1) Assembly

2) Process Management

→ Process Data Structures

→ Process System Calls

→ Threads

→ Process Scheduling

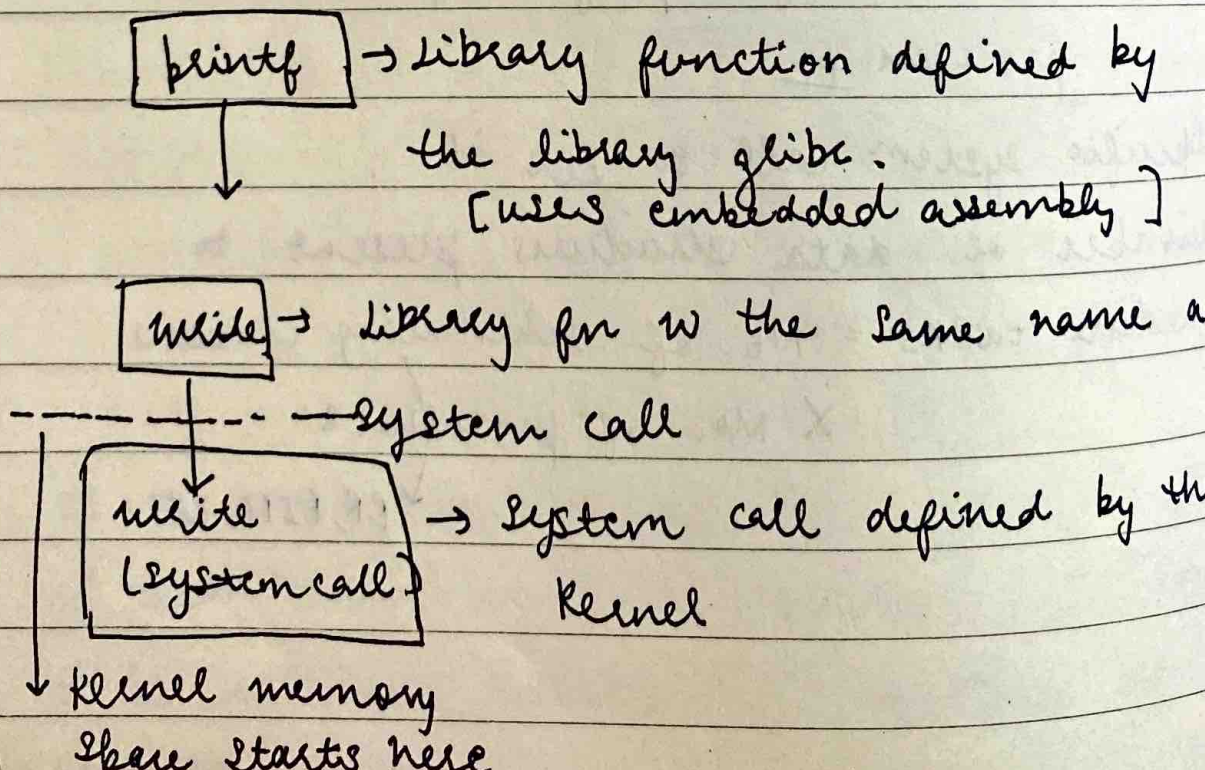
3) Structure of Kernel → layered kernel, microkernel

4) How do system calls work?

5) How does the bootup process work?

System calls

→ A mechanism of a user program communicating with the OS.



(no access to C library functions)

copy-to-user to copy the data ~~from~~ to the user's memory space ~~to~~ from kernel's memory space.

copy-from-user to copy the data ~~from~~ to the kernel's memory space ~~to~~ from users memory space

→ defined by linux kernel itself are functions in the kernel

↳ System calls use them

write ⇒ this is done in process context, so multiple processes can make similar requests at the same time

Syscall number

↓

Find correct system call from the system call table

↓

check the arguments whether they are valid

↓

copy-from-user ← processing of requests by making requests to device drivers or kernel data structures.

↓

copy_to_user

↓

copy return value to
eax register

↓

Return control to
user program

Boot Loader

1) Processors start in 16-bit mode (real mode)
   → No proper memory management
   → BIOS (Basic I/O System)
   → checks the hardware present; on error it stops the boot process.

processor, RAM, Keyboard

BIOS stored on ROM; needs to load data from disk.

Small disk drivers are present in BIOS to access the disk.

→ First sector of the disk needs to be in a specific format if it has the OS loaded into it.

Linux has a bootloader called GRUB.

→ Boot loader takes control from the BIOS.

→ Reads each of the disks to find out the OSes present.

→ If linux kernel is chosen, then it loads the kernel's image into the main memory.

→ Set up the data structures required to enable memory management unit & then set a flag in the control register CR0 to move to 32-bit mode (protected mode)

Linux kernel has different levels of service called "init levels".

All GUI + other processes allowed → init level 5

Everything except GUI → init level 3

GUI and network → init level 2

Single (root) user → init level 1

The init process is the first process created; the rest of the processes are spawned by using fork() and exec() one-by-one.

The command "init 0" shut downs the machine.

Daemon process: A type of process that runs in background and provides different user services (eg: → networkd, sshd).
Bash shell not daemon process.

Boot Process

BIOS: Basic I/O System
↳ 16-bit operations
→ Hardware checks

1) Booting process was slowed down by moving to 32-bit/64-bit mode.

2) Firmware (ROM) can be upgraded by downloading and flashing the newer images
↳ introduces a security risk by allowing malware to be installed

The idea of trusted computing came in where an OS had to "sign" to be allowed access into the firmware.
only if the signature (digital key) is present, then access is allowed. This makes it difficult for a new OS to be booted

Master boot record (MBR)

UEFI → Unified intel finance interface

Booting from multiple partitions of same disk is possible.

hardware check for more number of hardware is possible.

12 Qs.

8 3 mark Qs.

2 5 mark Qs.

2 8 marks

50

90 mins

1) a) when is FCFS scheduling not good for processes in terms of turnaround time.?

b) what alternate scheduling can resolve this?

create new process & load it

process / threads

Robert love : Linux Systems Programming

↳ talking to the kernel

31.10.22
Monday

Assembly programming → apart from arithmetic, logical ops, branches & loops, concept of memory alignment

Process Management → lifecycle, data structures, threads, scheduling

Structure of OS

Bootup process

Process Management

How can processes coordinate among each other? → Since processes are isolated, then why is process communication needed?

⇒ Inter-process communication (IPC) is needed to coordinate access to shared resources

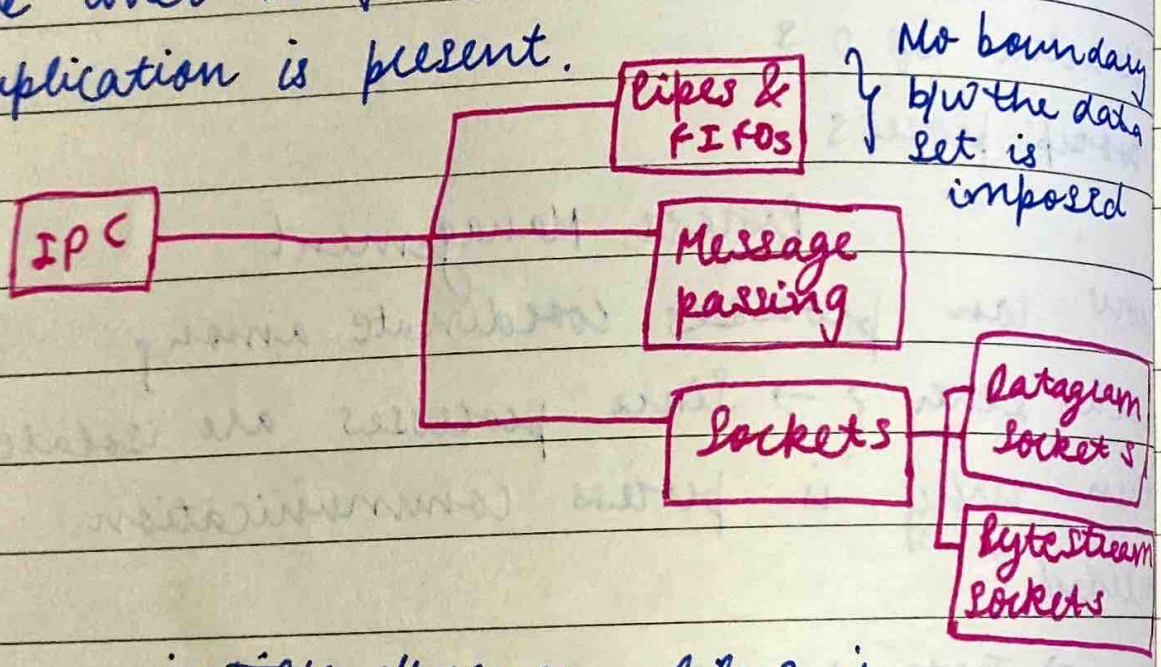
All data communication existing today uses IPC.

POSIX is the common standard used by all UNIX-based OS to provide process communication mechanisms

System V standard coming from AT&T's version of UNIX.

⇒ Also provided mechanisms of IPC.

Most UNIX systems have support for IPC from 2 different libraries, where we cover only POSIX libraries, wherever duplication is present.



communication through files is considered to degrade performance.

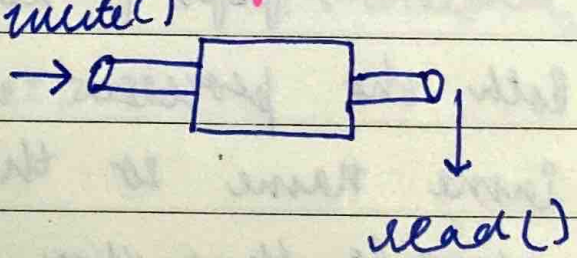
① Most of the IPC mechanisms in UNIX use the standard file-handling machines. The advantage of using it is that there is no need to build familiarity with a new set of system calls.

④ Shared memory → If you need to use many read's & write's, then the file handling mech can reduce performance due to context switches. If you use shared memory, then reading & writing is similar to reading & writing from an array. But consistency has to be ensured by the user processes through use of proper locks.

Pipe ("Anonymous" pipe) → doesn't have location, kept on kernel memory space itself

→ Virtual or pseudo-file which can be used for IPC

returns 2 functions




```
int fd[2];
fd = pipe();
write(fd[0]);
read(fd[1]);
pid = fork();
if (pid==0) { /* child process */
    char *buf = "abc";
    write(fd[0], buf);
    ⋮
}
else {
    char *buf;
    buf = malloc(100);
    ~~write~~ read = (fd[1], buf); }
```

1)
Disadvantage of pipe is that there is no way of communicating across processes that are not related.

2) Each process can only read or write but not both. Communication is unidirectional. Bidirectional communication using a single pipe would require 2 different pipes.

FIFO or Named pipe can be used to communicate across processes that are not related.

fifo ("abc");

Both the processes would need to use the same name so that the kernel can identify that they want to communicate with each other. After opening using the system call, read/write/close can be used as usual. Other functions are similar to the original pipes.

Pipes do not guarantee that they would read/write the entire data at a time.

```
ret_val = write(fd, buf, buf_len)
                          512/1024, 4096
```

```
/* ret_val indicates how many bytes
   have been written */

total_char = 0;
while (total_char < 1024) {
        ret_val = write(fd, buf[total_char],
                          1024 - total_char);
        total_char += ret_val;
    }
    close(fd);
```

Read to a pipe is blocking in nature.

```
while(tot_char < 1024) {
        ret_val = read(fd, buf[tot_char],
                          1024 - tot_char)
        tot_char += ret_val;
    }
```