Tensorflow : internal libraries
implement concurrency

Python - no support for concurrent threads

```
t[i] = posix_thread.create(sum, x, y, i*25,
                           (i+1)*25);
}
for(int i=0; i<4; i++) {
    posix_thread.join(t[i]);
}
return 0;
}
```

Python
↳ Global Interpreter Lock (GIL)

sum = x+y; → vector operation that can be done using data-level parallelism

When are threads most useful?
→ Browsers w multiple tabs
→ Server with many requests

Linux keeps track of type of machine during install and does scheduling of processes and threads accordingly.

Linux assigns a new process ID for every thread. The responsibility of ensuring that an I/O doesn't block a full user process is on the scheduler, which checks if other threads of the same user are available

26.09.22

Assembly programming

Process Management

① Which system calls to use ?

② concepts of process/threads

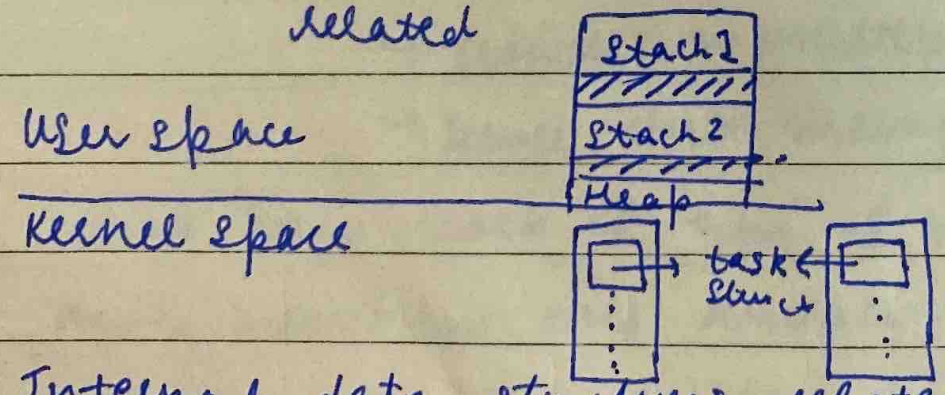
③ Internal organisation within the kernel → linux kernel in this course

④ Process Scheduler

midterms

⑤ Inter process communication & Synchronization

Address Space → Memory area where we store the process, data ~~structures~~ related.



Internal data structures related to each process kept by the Kernel. Kept in the process address space within the kernel space.

task → anything that can be scheduled
thread / process

In linux, process control block is called task_struct.

You can enter into (execute code from) kernel space in two ways:

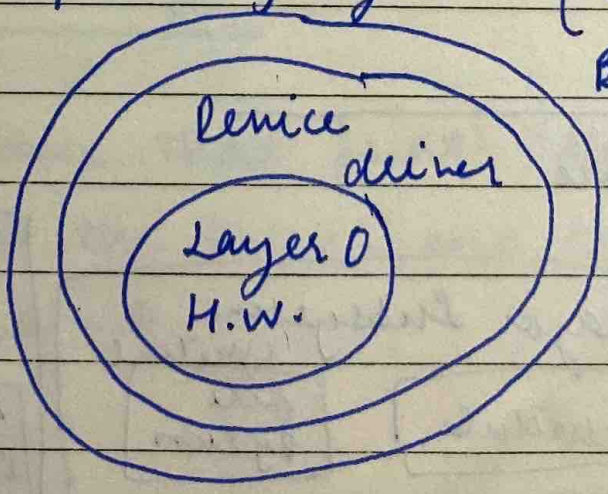
1) Process context → Enter into it using system calls.

2) Interrupt context → Enters into using hardware signals

How does the kernel handle requests?

→ Linux is a monolithic kernel
↳ any f^n in the kernel can call any other f^n in the kernel

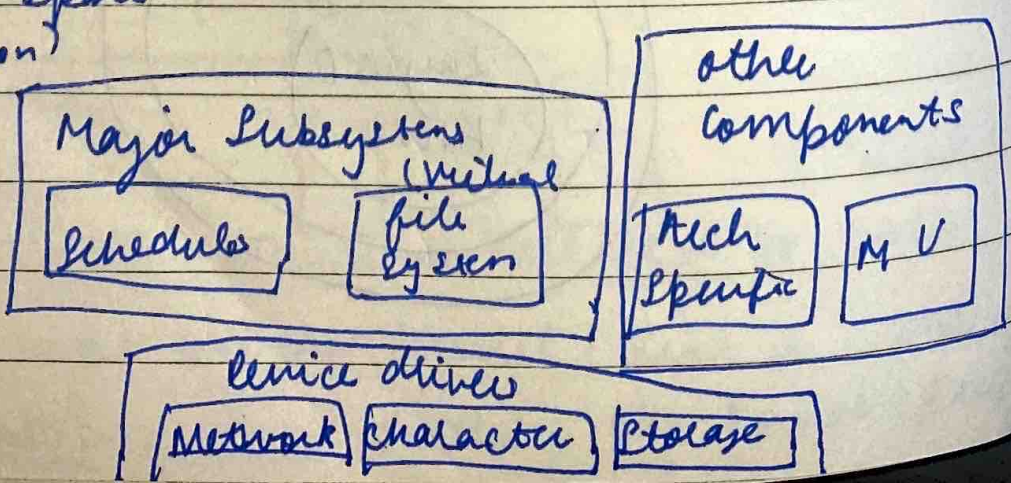
→ Layered operating system (UNIX System V, Berkeley UNIX)



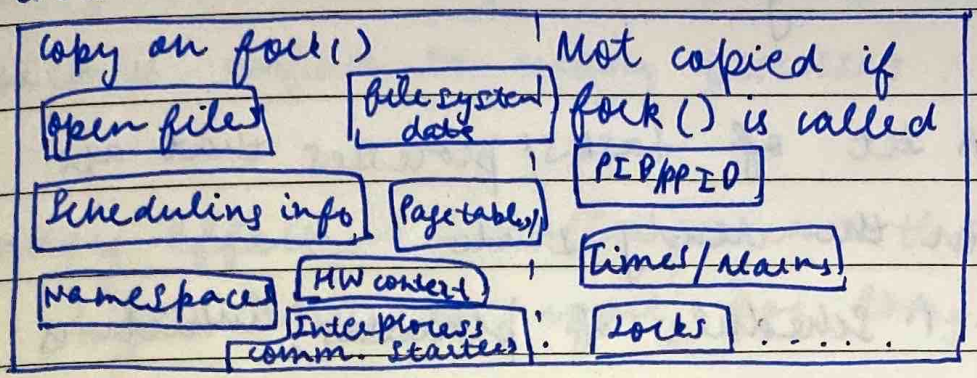
→ Microkernels → Move file system & device drivers to the user space. The OS is secure from crashes due to device driver bugs. Very reliable; used in embedded & safety-critical systems.

GNU Hurd → Attempt to create a microkernel for desktop systems

Linux Kernel Modules → Parts of the kernel that are not needed to run the basic codes of processes. Modules can be loaded when the OS is running. They can also be removed if it is not in used. Once loaded, the modules can access any linux function.



task_struct



Circular linked lists to keep track of task_structs.

There is a "current" variable which keeps track of the current task_struct. This current variable is stored in a register. Whenever the kernel wants to make a process switch, the current variable has to be modified.

terminating a process for a kernel is the same as destroying the task_struct.

A kernel module that prints "Hello world". Load it into the kernel and see whether message comes up

Parent process ID
↓
Process ID

28.09.22 Wed

Process Scheduling

Task

Input: A set of tasks/processes that are in the ready state

Output: A schedule of processes, which is the same as selecting a process to send it to the running state.

First come first serve (FCFS) → Non-preemptive

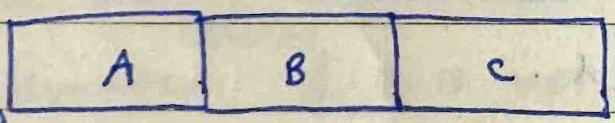
A → 15 time units, arrived at 0

B → 20 time units, " 1

C → 10 time units, " 2

Criteria: arrival time

Burst time/execution time



Gantt Chart

Mean turnaround time = Avg time taken to complete the task

↙ Completion time - ↓ arrival time

Since it's beginning

Kitne time ruka system mein

$T_{turnaround}(A) = 15$

$T_{turnaround}(B) = 35 - 1 = 34$

$T_{turnaround}(C) = 45 - 2 = 43$

$T_{turnaround}(Avg) = \frac{15 + 34 + 43}{3}$

WT = TAT - BT

↓ Pre-emptive / Non Pre emptive
time quantum,
responsiveness,
priority

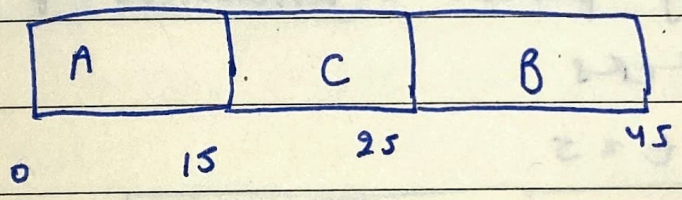
Average start time of FCFS can be large, which is unfair to many processes.

convoy effect → smaller processes that come in might have to wait for long time for a long process to ~~end~~ finish.

Shortest job first

Criteria : burst time
Non-preemptive

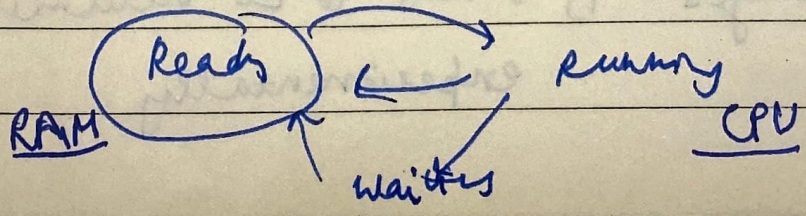
Schedule at each step the job that will take the shortest time to finish.



Starvation

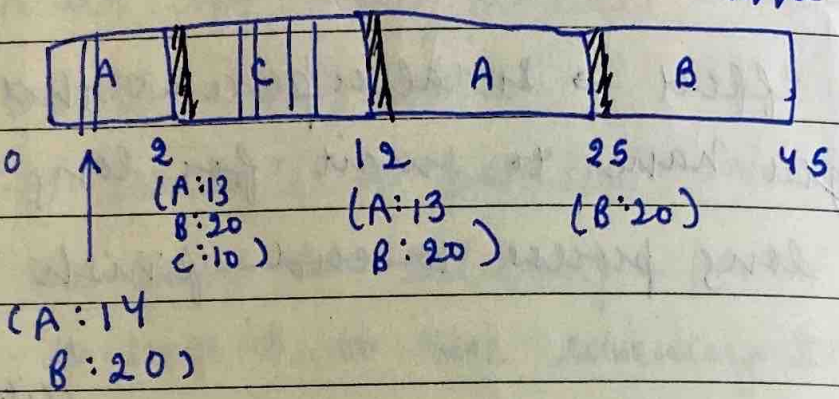
A process may not get access to the CPU for infinite time.

So far, the assumption is that the process cannot be moved back ~~from~~ to the ready state from the running state. i.e. processes are non preemptive in nature



Preemptive version of SJF → [Shortest remaining time first]

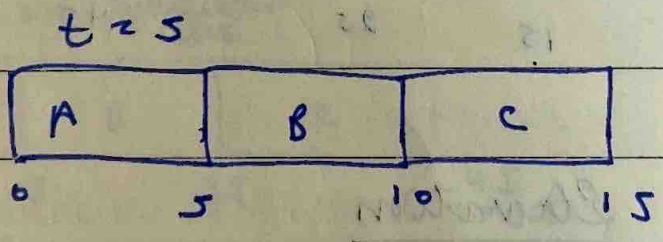
"Burst time"



Preemptive

Round-Robin Scheduling: Divide some units of time called time quantum. Allocate each ready process amount of time based on this.

"Time quantum"



Advantages - 1) No process is kept waiting for too long.

2) No need to have accurate estimates of process execution time.

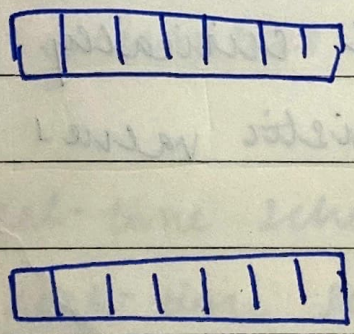
Disadvantages - 1) t has to be determined experimentally

small burst time

2) If t is large, then interactive jobs are not satisfied. If t is small, ~~then 3~~ then there are too many process switches.

3) If t is known, then a process might be written in such a way that it uses I/O only just before its t finishes.

Multi level feedback queue



① Give higher priority to a process that is at a higher queue ⇒ If P1 is at a higher queue than P2, then schedule P1.

② Higher queue has a smaller time quanta than a lower queue

③

Let every process start from Q1. If it uses the entire time quantum, then move it to lower queue for its next execution. If it does not use the entire time quantum, then keep it in the same queue.

④ Starvation-Free: After some amt of time, send all the lower-queue processes to the highest queue.

Scheduler performance is critically dependent on the heuristic values (time quanta and s)

The schedulers discussed today are fine for single processors. For multiprocessor scheduler, there needs to be proper handling of additional race conditions.

3.10.22, Monday

1) What an OS does ? ⇒ Principles

2) How does it perform its actual actions ?

⇒ Latest version of the linux kernel

Process Scheduler

↳ 2 types

1) Normal

2) Real-time scheduler

⇒ Ask for some specific guarantees

Latency guarantees

Soft real-time schedulers → (Playing videos)

Hard real-time schedulers → (Industries, cars, etc)

⇓

Linux kernel cannot

Satisfy these constraints

How can the linux kernel handle both soft-real time constraints & normal jobs together ?

→ Every process must belong to a scheduling class.

→ Scheduling class defines the scheduling policy that will be used.

→ Each class assumes that processes / threads are preemptive

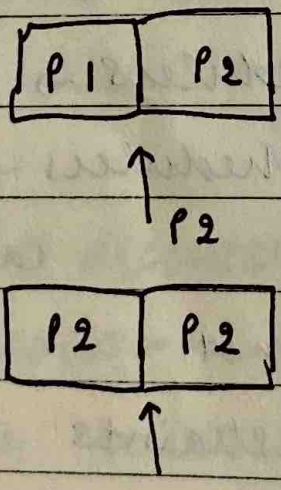
1) SCHED_FIFO → uses FIFO scheduling with no time slice; if higher priority job in this class comes, then this job can be moved back to the steady state

Priority:

50 (P1)

60 (P2)

(priority values from 0 to 99)



When P2 ends, P1 starts

2) SCHED_RR (soft real-time) → very similar to SCHED_FIFO, but w a timeslice of 10 ms after which a process is preempted.

3) SCHED_NORMAL (Normal user processes) →
Priority values of 0; but "nice" values
can be used to change priority.
Nice values → Higher values imply lower
priority
-20 to +19
default is 0

A user can increase the nice value to
indicate that the process can be slower
than usual.
Only root users can set nice values to (-) ve
↳ not ordinary users

2.6.23 of the kernel → Completely Fair
Scheduler (CFS)
Provide equal amt. of access to the
processor to each task

160 ms
50 ms 50 ms
10 ms . . . . 10 ms

Too many process switches are possible if
a fully fair scheduler is used. To avoid
it, a floor of 1 ms is kept below

* imp. part

which time slice is not divided.

"Amount of access" is the time spent at the processor (stored in a variable called vruntime).

At each point of time, choose the process w the lowest vruntime, If kept in a linked list / array, it can take O(n) time.

Red black trees to store vruntimes,
O (log n) × 2
which reduces the time complexity to O (log n).

There is a table of weights based on the nice values, where higher nice values have higher weights.

[vruntime += actual_processor_time * weight [nice_value];

System call called " nice" which can be used to change the nice value of a process.