OPERATING SYSTEM NOTES

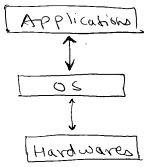
09 September 2021 16:08

-Subhojit Ghimire

* Syllabus:

1. Basics: Types of OS, Process Diagram, System Calls,
2. Process (CPU) Scheduling: FIFO, SJF, Round Robin, Priority, Preemptive
3. Process Synchronisation: Critical Section, Semaphore
4. Deadlock and threads: Banker Algorithm
5. Memory management: Paging, Segmentation, Fragmentation, Virtual Memory, Page Replacement Algorithm
6. Disk Scheduling: SCAN, CSCAN, FCFS
7. UNIX Commands: ls, mkdir, chmod, cd, open system calls
8. File management and security: Sequential, Random, linked

* Operating System and its Functions:



Primary Goal:
- → Convenience
- → Throughput: No. of tasks executed per unit of time.

Functionalities:
- → Resource Management (Parallel Processing)
- → Process Management (CPU Scheduling)
- → Storage Management (File System)
- → Memory Management
- → Security and Privacy

* Types of OS:

1) Batch : Punch Cards | Paper tape | Magnetic tape → Operator → CPU → I/O
2) Multiprogrammed
3) Multitasking
4) Real time OS
5) Distributed
6) Clustured
7) Embedded.

- Multiprogrammed OS :
  - → Non-preemptive
  - → Multiple processes in RAM. Each process is executed completely before moving to next one.
  - → Idle
- Multitasking / Time Sharing OS
  - → Preemptive
  - → Multiple processes in RAM. Executing time is predefined. Each process gets processed for a specific time and after a fixed time, the processor moves on to the next process even

predefined. Each process gets processed for a specific time and after a fixed time, the processor moves on to the next process even if the previous process has not been executed completely. The processor will come back to that process later on, after it has given equal time to every processes in RAM.
→ Responsiveness

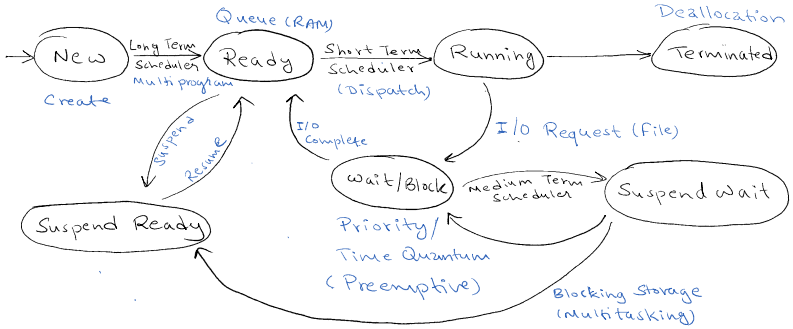
- Real-time OS
  - Hard : No delays can be tolerated.
  - Soft : Little delays can tolerated

- Distributed Environment:
  → Processing environment is distributed all over the geographical region connected by a network

- Clustered:
  → Distributed within a small area connected by LAN

- Embedded:
  → Work on fixed functionality. Cannot be changed.
  E.g.: microwave, washing machine.

✳ Process States:



- Preemptive: If CPU dismisses the currently running process for any reason like its time quantum has reached or maybe a priority process is incoming.

- Non-preemptive: If CPU completely executes the currently running process before moving on to the next one.

Example Q. Which command is used to assign only Read only Permission to all three categories of file 'note'?
A> chmod a-rw
✓ C> chmod ugo=r note
B> chmod got+r note
D> chmod u+r, g+r, o-x note

NOTE: chmod → change mode.
permissions → read (r), write (w), execute (x)

u → user
g → group
o → others
} categories

Q. chmod ugo+rw note command can be represented in octal notation as
A) chmod 555 note
✓ B) chmod 666 note
C) chmod 333 note
D) chmod 444 note

NOTE: octal notation: r → 4 w → 2 x → 1
rw → 4+2 = 6
rx → 4+1 = 5
wx → 2+1 = 3
rwx → 4+2+1 = 7

r w x | r w x | r w x
u | g | o

Q. Suppose you have a file "f1", whose contents are:

1 2 3 4 5 6 7 8 9 0 a b c d e f g h i j

here, 'lseek' is used two times sequentially

```
lseek (n, 10, SEEK_CUR);
lseek (n, 5 , SEEK_SET);   n is file descriptor
```

After applying lseek two times, what will be current position of the RTW head?

A) 0 C) 10

~~B)~~ 5 D) 15

* System Call :

- File Related : open(), read(), write(), close(), create file etc.
- Device Related : Read, Write, Reposition, fcntl, ioctl etc.
- Information : getPID, attributes, get system time and date
- Process Control : load, execute, abort, fork, wait, signal, allocate etc.
- Communication : pipe(), create / delete connections, shmget()

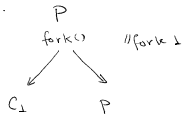
• fork() System call :

- To create child process (an exact clone of parent process)
- Returns values
  - 0 child process created
  - +ve parent process itself
  - -ve child process not created

Example:

```
{ fork();    // fork 1
  printf("Hello world!"); // Parent Process P
}
```

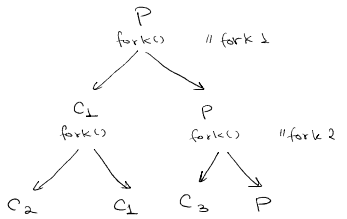
→



2x Hello world! gets printed

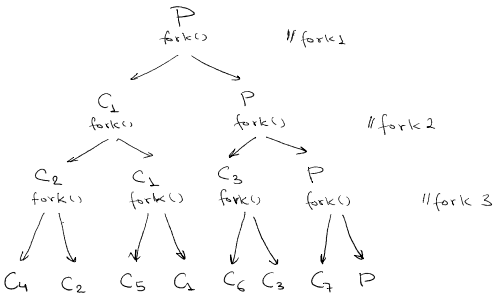
```
{ fork();    // fork 1
  fork();    // fork 2
  printf("Hello World!)"; // Parent Process P
}
```

→



4x HelloWorld! gets printed.

```
{ fork();    // fork 1
  fork();    // fork 2
  fork();    // fork 3
  printf("Hello World!"),   // Parent Process P
```



8x Hello world! gets printed.

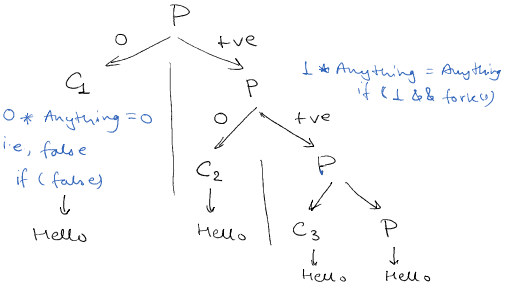
7 → Child process (C_1 - C_7)

1 → Parent Process (P)

* Total Execution = 2^n
* Total Child processes = $2^n - 1$

Q.

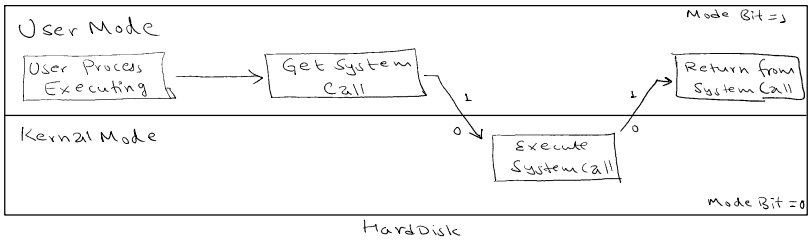
```
#include <stdio.h>
#include <unistd.h>
int main () {
      if (fork() && fork())
            fork();
      printf("Hello");
      return 0;
}
```



→ HelloHelloHelloHello

i.e., 4×Hello is printed.

* User Mode vs Kernal Mode:



* Process Vs Threads:

Process	Threads
1. System calls involved in process	1. No System call involved.
2. OS treats different processes differently.	2. All user level threads treated as single task for OS.
3. Different processes have different copies of data, files, code.	3. Threads share same copy of code and data.
4. Content switching is slower.	4. Content switching is faster.
5. Blocking a process does not affect other processes	5. Blocking a thread will block an entire process.
6. Independent.	6. Inter-dependent

* User level Thread VS Kernel Level Thread

User level Thread	Kernel Level Thread
1. Managed by User level library.	1. Managed by OS System Calls.
2. Typically fast	2. Comparitively slower than User level
3. Content Switching is faster	3. Content Switching is slower.
4. If one user level thread performs blocking operation then entire processes get blocked.	4. If one kernal level thread is blocked, no effect on others.

* Scheduling Algorithms:

Pre-emptive
- SRTF (Shortest Remaining Time First)
- LRTF (Longest Remaining Time First)
- Round Robin
- Priority

Non-preemptive
- FCFS (first come first serve)
- SJF (Shortest Job First)
- LJF (Longest Job First)
- HRRN (Highest Response Ratio Next)

- SJF (Shortest Job First)
- LJF (Longest Job First)
- HRRN (Highest Response Ratio Next)
- Multilevel Queue
- Priority (less often used)

* CPU Scheduling:

1. Arrival Time: Time at which process enters the Ready Queue or State. (Point of time)
2. Burst Time: Time required by a process to get execute on CPU. (Duration)
3. Completion Time: Time at which process completes its execution. (Point of time)
4. Turn-around Time = Completion Time − Arrival Time (Duration)
5. Waiting Time = Turn-Around Time − Burst Time (Duration)
6. Response Time = {Time at which a process gets CPU first time} − {Arrival Time} (Point of Time)

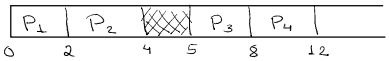
* FCFS: (First Come First Serve)

- Criteria: Arrival Time
- Mode: Non-preemptive

Example

Process No.	Arrival Time	Burst Time	Completion Time	Turn-around Time	Waiting Time	Response Time
P_1	0	2	2	2	0	0
P_0	1	2	4	3	1	1
P_2	5	3	8	3	0	0
P_4	6	4	12	6	2	2

→ Gantt Chart



(In case of preemptive, Waiting Time = Response Time)

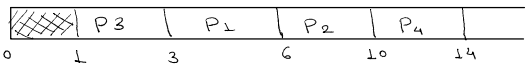
* SJF: (Shortest Job First)

- Criteria: Burst Time
- Mode: Non-preemptive

Example

Process No.	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time	Response Time
P_1	1	3	6	5	2	2
P_2	2	4	10	8	4	4
P_3	1	2	3	2	0	0
P_4	4	4	14	10	6	6

→ Gantt Chart



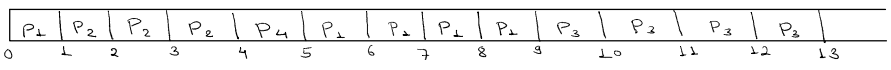
* SRTF: (Shortest Remaining Time First)

- Criteria: Burst Time
- Mode: Preemptive

Example

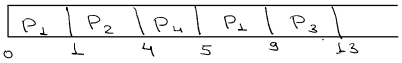
Process No.	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time	Response Time
P_1	0	5	9	9	4	0
P_2	1	3	4	3	0	0
P_3	2	4	13	11	7	7
P_4	4	1	5	1	0	0

→ Gantt Chart



SRTF → Preemptive SJF

↳ Simplified As:



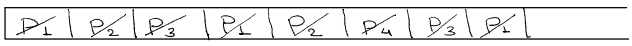
✳ Round Robin.

↳ Criteria : Time Quantum (i.e., Specified Time)
↳ Mode : Preemptive

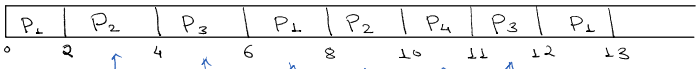
Example :

Process No	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time	Response Time
P_1	0	5	13	13	8	0
P_2	1	4	10	9	5	1
P_3	2	3	12	10	7	2
P_4	6	1	11	5	4	4

Given, Time Quantum = 2.

→ Ready Queue 

Gantt Chart

Running Queue 

Context Switching : 6 (Do not count first and last processes)

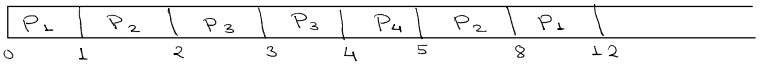
✳ Priority Scheduling :

↳ Criteria : Priority
↳ Mode : Preemptive (Can be non-preemptive as well)

Example

Priority	Process No.	Arrival Time	Burst Time	Completion Time	Turnaround Time	Waiting Time
10	P_1	0	5	12	12	7
20	P_2	1	4	8	7	3
30	P_3	2	2	4	2	0
40	P_4	4	1	5	1	0

Higher the no. Higher the priority

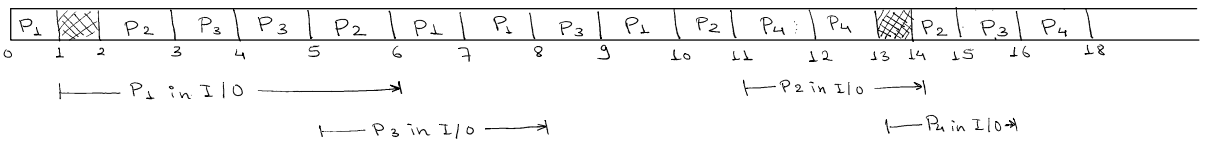
→ Gantt Chart : 

✳ Process Scheduling :

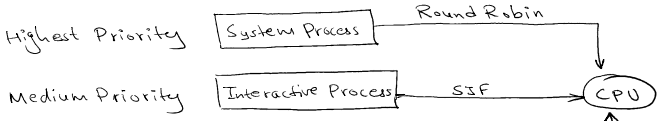
↳ Mode : Preemptive (can be non-preemptive as well)
↳ Criteria : Priority Bases.

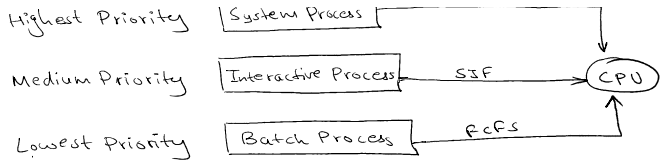
Example :

Process Number	Arrival Time	Priority	CPU	I/O	CPU
P_1	0	2	10	5	3
P_2	2	3	3	3	1
P_3	3	1	2	3	1
P_4	3	4	2	4	1

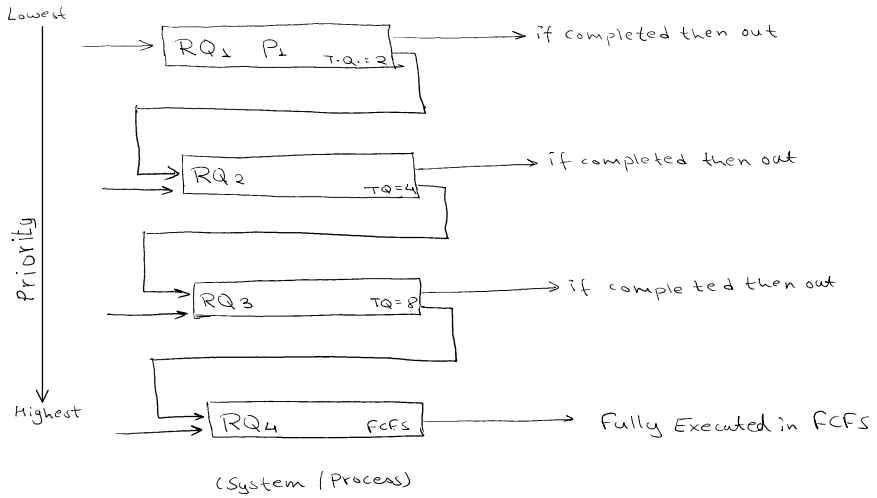
→ Gantt Chart : 

✳ Multilevel Queue Scheduling :



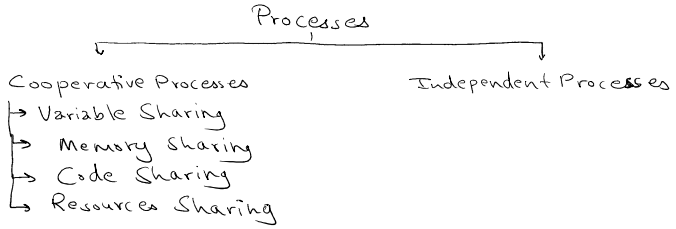


* Multilevel Feedback Queue



(System / Process)

* Process Synchronisation:



* Producer Consumer Problem.

* Printer - Spooler Problem.

* Critical Section Problem:
  ↳ Part of program where shared resources are accessed by various processes.

Critical Section is the section or place where shared varizables or resources are stored or placed.

- Synchronisation Mechanism:
  4 conditions / rules:
  - Mutual Exclusion → (when one process is already using critical section, another process should wait) } Primary
  - Progress → (If one process wants to access critical section, another process should not block the access)
  - Bounded Waiting → (There should be no starvation, i.e., one process is accessing critical section in finite time) } Secondary
  - No assumption related to Hardware speed
    ↳ (should be portable and universal)

* Critical Section Solution using 'Lock':

```
do {
        acquire lock
        critical section access
        release lock
```

```
1. while (lock == 1);   } Entry Code
2. lock = 1
3. Critical Section
4. lock = 0             } Exit Code
```

```
do {
    require lock
    critical section access
    release lock
}
```

1. while (lock == 1); } Entry code
2. Lock = 1
3. Critical Section
4. Lock = 0 } Exit code

- Execute in User Mode
- Multiprocess Solution
- No mutual exclusion guaranteed.

- Does not ensure mutual exclusion ×
- Process Synchronisation can be broken ×

* Critical Section Solution Using 'Test-and-Set' Instruction :

```
boolean test_and_set (boolean *target) {
        boolean r = *target;
        *target = TRUE;
        return r;
}
```

1. while (test_and_set (&lock)) } Entry code
2. Critical Section
3. lock = FALSE; } Exit code

```
while (test_and_set (&lock) {
        critical section access
        lock = false;
}
```

- → Always ensures mutual exclusion ✓
- → Ensures progress ✓
- → Does not ensure bounded waiting. ×
- → Independent of hardware. ✓

* Turn Variable (strict Alteration)

- 2 process solution
- Runs in user mode.

Process P0	Process P1
while (turn != 0);	while (turn != 1)
Critical section	Critical section
turn = 1;	turn = 0;

- → Always ensures mutual exclusion ✓
- → Does not ensure progress. ×
- → Ensures bounded waiting. ✓
- → Independent of hardware. ✓