CSCI 5103 Operating Systems
Assignment 2

Ritiz Tambi
tambi004

Problem 1.

CPU utilization is the fraction of time spent on CPU. In other words, it can be computed as *CPU_{util} = 1 - (Fraction of time spent on idle task)*

In the given problem, for 1GB of memory, degree of multiprogramming is 4 and each job spends 70% of time waiting for I/O.
Hence, $CPU_{util} = 1 - (0.7)^4 = 0.7599$, **$CPU_{util =}$ 76% (approx)**

For an additional 1GB of memory, degree of multiprogramming is 9.
Hence, $CPU_{util} = 1 - (0.7)^9 = 0.9596$, **$CPU_{util =}$ 96% (approx)**

Problem 2.

For a multi-threaded process, each user-level thread has a task structure. This process descriptor of type *task-struct* is resident in memory at all times. It contains vital information needed for the kernel's management of all processes, including scheduling parameters, lists of open-file descriptors, and so on. The process descriptor along with memory for the kernel-mode stack for the process are created upon process creation. On a fork() call, a new child process is created. This child process inherits all file descriptors and attributes from parent like Open File Descriptors, Scheduling Information, Real/Effective group and user ID, etc. The majority of the process-descriptor contents are filled out based on the parent's descriptor values. Linux then looks for an available PID, that is, not one currently in use by any process, and updates the PID hash-table entry to point to the new task structure.

However for a multi-threaded process a couple of issues come up on a fork() call.

- If a process with multiple kernel threads does a fork, whether the child process should have all other parent threads created or not. For a I/O blocking call in the parent process, what sort of behaviour will the corresponding thread in child depict.
- If the child process does not have all the other threads created and if one of the non-created threads hold a mutex that the only child thread tries to acquire after forking. This mutex will not be released.
- File I/O is another problem area. If one thread is blocked reading from a file and another thread closes the file or does an lseek to change the current file pointer, this will create a problem.

Problem 3.

$\text{CPU}_{\text{efficiency}} = \frac{CPU\ Time\ (None\ idle)}{Total\ CPU\ Time}$

In the given problem, we have CPU Time (Non-Idle) = T, Context Switch Time = S.

a)

For Q = infinity, RR scheduling behaves like FCFS. Hence a context switch only occurs when the process gets blocked or finishes in entirety.

$\text{CPU}_{\text{efficiency}} = \frac{T}{T+S}$

b)

For Q > T, RR scheduling essentially behaves like FCFS. Hence a context switch only occurs when the process gets blocked or finishes in entirety.

$\text{CPU}_{\text{efficiency}} = \frac{T}{T+S}$

c)

For S < Q < T, number of times context switch occurs = T/Q. Since, each context switch requires S time, total overhead for context switch = S*(T/Q)

$\text{CPU}_{\text{efficiency}} = \frac{T}{T+(ST/Q)} = \frac{Q}{Q+S}$

d)

For Q = S, in the above formula assuming Q<T and substituting S by Q. $\text{CPU}_{\text{efficiency}} = \frac{Q}{Q+S} = \frac{Q}{Q+Q}$

$\text{CPU}_{\text{efficiency}}$ = 50%

e)

For Q = 0, CPU does no work and essentially $T \simeq 0$. $\text{CPU}_{\text{efficiency}} = 0$.

Problem 4.

a)

Task A : Period 4 units. Service time 1 unit. Utilization = ¼ = 0.25

Task B: Period 5 units. Service time 3 units. Utilization = ⅗ = 0.6

Total CPU Utilization (U) = 0.25 + 0.6 = 0.85

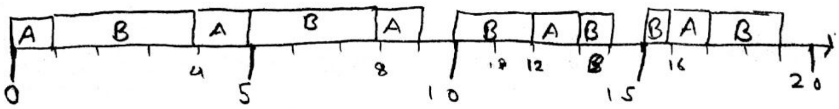
According to the RMS condition policy $U < N * (2^{1/N} - 1)$

For N = 2, $U < 2*(\sqrt{2} - 1)$ => $U < 0.828$

Hence for our case, since U > 0.828 the RMS condition for guaranteeing a feasible solution is not satisfied.

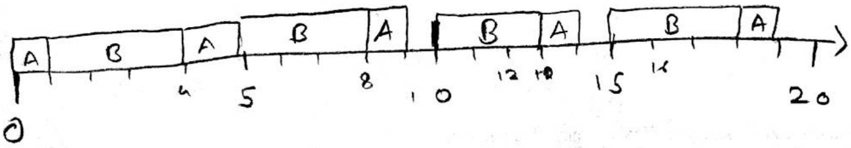
Since period Task A period < period Task B => Priority Task A > Priority Task B.

b)



Schedule is feasible

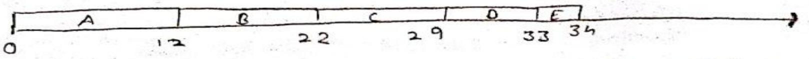
c) Assuming deadline = period



Schedule is feasible

Problem 5.

a)

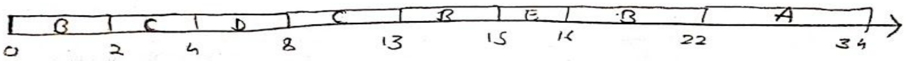


1) Average Waiting Time = $\frac{0 + 12 + 20 + 25 + 18}{5}$ = 15 units

2) Average Turnaround Time = $\frac{12 + 22 + 27 + 29 + 19}{5}$ = 21.8 units

(Turnaround Time = Wait Time + Service Time)

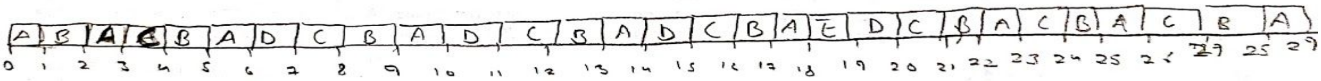
b)

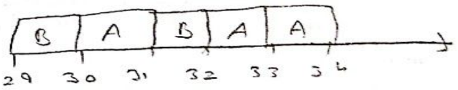


1) Average Waiting Time = $\frac{22 + 12 + 4 + 0 + 0}{5}$ = 7.6 units

2) Average Turnaround Time = $\frac{34 + 22 + 11 + 4 + 1}{5}$ = 14.4 units

c)

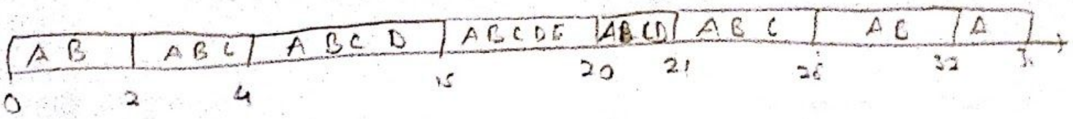




1) Average Waiting Time = $\frac{22 + 22 + 18 + 12 + 3}{5}$ = 15.4 units

2) Average Turnaround Time = $\frac{34 + 32 + 25 + 16 + 4}{5}$ = 22.2 units

d) For $T \to 0$, effectively all jobs share the system



Consider Process D, when it arrives at $t = 4$ it shares processor with ABC till $t = 15$ and then for 1 sec, Hence it effectively gets = ~~[illegible]~~ $(11+1)/4$ CPU time

Then from $T = 15$ to $T = 20$, it shares processor with ABCE for $= 5/5 =$ 1 unit of CPU time.

1) Avg Waiting Time = $\underline{\underline{0}}$

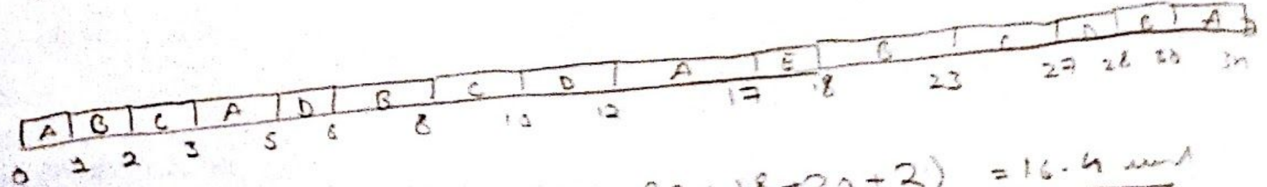
2) Avg Turnaround Time = $\frac{34+32+26+17+5}{5}$ = $\underline{\underline{22.8}}$ units

e) Queue 1 A B C D ~~E~~ E

Queue 2 A B C D

Queue 3 A B C D

SRTF A B



Avg Waiting Time = $\frac{(22+20+18+20+2)}{5}$ = $\underline{\underline{16.4}}$ units

Avg T. T = $\frac{(34+30+25+24+3)}{5}$ = $\underline{\underline{23.2}}$ units

Problem 6.

a)

Mean Utilization = Fraction of time Server is busy
= (Avg. CPU Service Time) / (Avg. Time between Job Arrivals)

Given, a job takes on avg S time. At the same moment λ jobs arrive every sec.

Utilization = (S)/(1/λ) = **λS**

b)

Avg. Turnaround Time T for a job = Service Time for the Job + Waiting Time for the Job.
An avg job taken S time on the system. Since an incoming process finds N processes in FCFS queue in front of it, the process has to wait for N*S time.

Hence, E(T) = T = S + N*S = **S*(N+1)**

c)

Acc to Little Law, N = λT
But from part a, Utilization p = λS
Substituting in equation obtained from part b,
T = S*(λT+1) => T = S/(1- λS) => ***T = S/(1-p)***

d)

From above eqn. *T = S/(1-p)*
For p = 1, T => infinity.
For p = 0.5, T = 2*S

Problem 7.

a)

The value of *variable* in the parent process remains -1.

b)

Yes, the read operation will be successfully executed by the parent (**IF** the file exists). Child process inherits a copy of the file descriptor and closes its copy. The parent maintains its file descriptor and is able to read from the file.