OS TAKE HOME ASSIGNMENT-II

Itish Agarwal (18CS30021)

Q1. We user pipe to send data from child to parent:

Code:

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>
int main() {
    char filename[200];
    printf("Enter name of file: ");
    scanf("%s", filename);
    int p1[2];                    // pipe from child to parent
    if (pipe(p1) == -1) {
        printf("Error in creating pipe\n");
        exit(EXIT_FAILURE);
    }
    pid_t pid = fork();
    if (pid == 0) {               // means child process
        close(p1[0]);
        int wc = 0
        FILE * ptr = fopen(filename, "r");
        if (!ptr) {
            wc = -1;
            write(p1[1], &wc, sizeof(wc));
            exit(EXIT_FAILURE);
        } else {
            char ch;
            int last_wc = 1;
            while(fscanf(filename, "%c", &ch) >0) {
                if (last_wc) {
                    if(!(ch==' ' || ch=='\t' || ch=='\n'))
                    {
```

```
            wc++;
            Last_wc = 0;
        }
    }else if (ch==' ' || ch=='\t' || ch=='\n'){
        last_wc = 1;
    }
  }
  write(p1[1], &wc, sizeof(wc));
  //we are done with child so close pipe
  close (p1[1]);
} else {      // parent process
    close (p1[1]);
    wait (NULL); //wait for child to finish
    int x;
    read ( p1[0], &x, sizeof (x));
    if (x == -1){
        printf("Error in opening file \n");
    } else {
        printf (" Number of words as
                  obtained by child = %d\n", x);
    }
    close (p1[0]);
}
return 0;
}
```

Q2. FCFS:

FCFS (First Come First serve) completes the jobs which arrive before others, ie, a job arriving before another will be completed first.

FCFS may unintentionally discriminate against short jobs, and may prove to be bad for such jobs. Since the scheduling is non-preemptive, a short job arriving late might have a very large waiting time due to longer jobs that arrived before. Due to this the shorter job may starve and never get the CPU for its execution.

Round Robin:

Round Robin treats all jobs similarly giving each job a specified time irrespective of its CPU burst time. Hence it does not discriminate against any type of process in any way.

Multi-Level feedback queue :

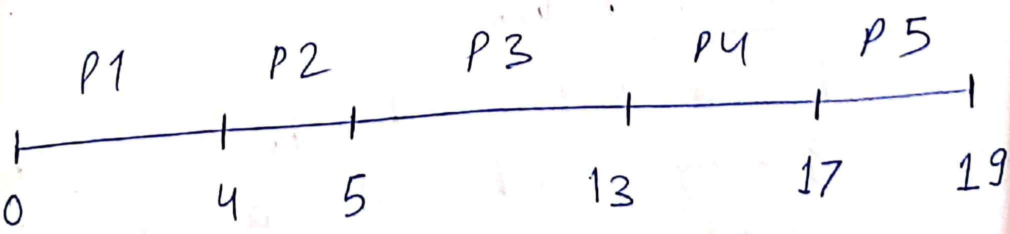
MLFQS favours a shorter process to be executed first, as longer processes are moved ~~dow~~ down to lower levels if they are not completed in the time quanta of the higher levels.

∴ Shorter processes entering the higher level are executed before the longer processes as they are at a lower level, even though they arrived earlier.

However, some larger jobs may starve if they are not promoted periodically.

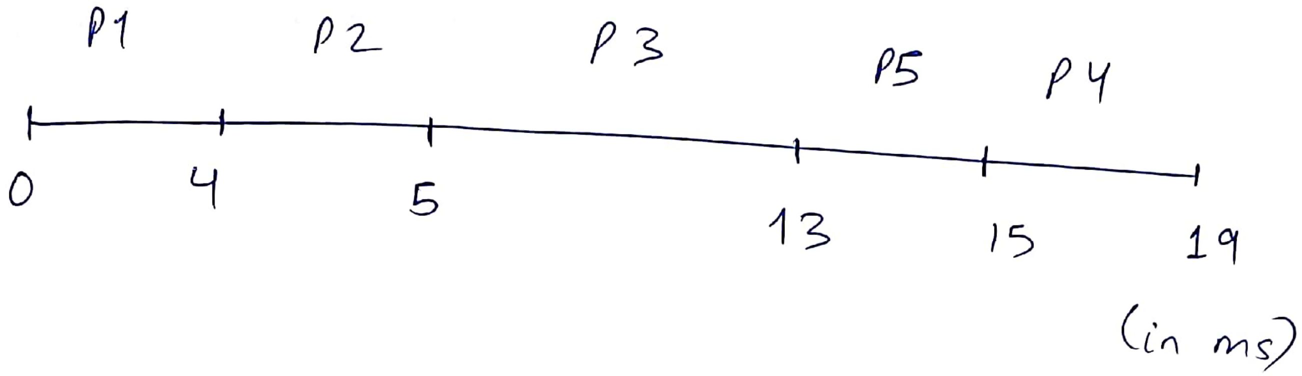
Q4.

(a) (i) Gantt chart for FCFS

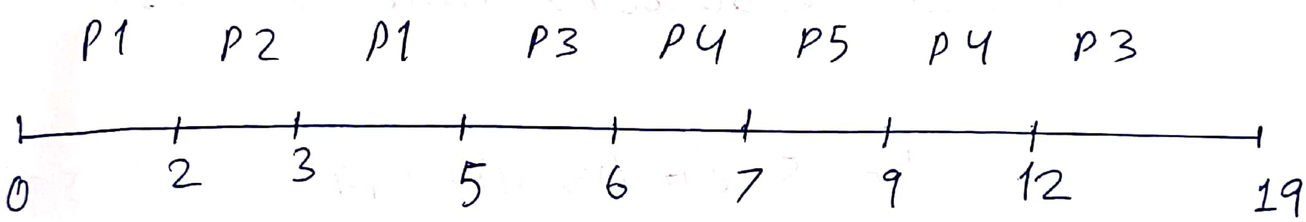


(ii) PTO

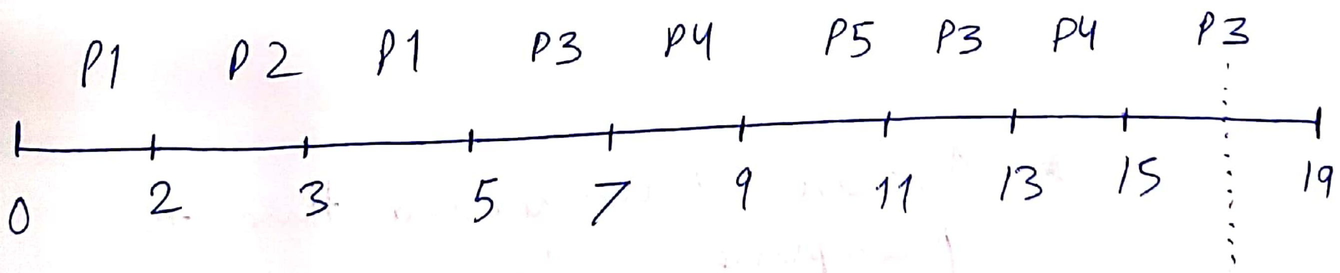
(ii) Gantt chart for SJF



(iii) Gantt chart for preemptive SJF



(iv) Gantt chart for RR (δ = 2 ms)



(b) PTO

(b) (i) ★ Average turn around time for FCFS

$$= \frac{(4-0)+(5-2)+(13-4)+(17-6)+(19-7)}{5}$$

$$= \frac{39}{5} = 7.8 \text{ ms}$$

★ Average turnaround time for SJF

$$= \frac{(4-0)+(5-2)+(13-4)+(19-6)+(15-7)}{5}$$

$$= \frac{37}{5} = 7.4 \text{ ms}$$

★ Average turnaround time for preemptive SJF

$$= \frac{(5-0)+(3-2)+(19-4)+(12-6)+(9-7)}{5}$$

$$= \frac{29}{5} = 5.8 \text{ ms}$$

* Average turnaround time for RR ($\delta = 2ms$)

$$= \frac{(5-0) + (3-2) + (19-4) + (15-6) + (11-7)}{5}$$

$$= \frac{34}{5} = 6.8\ ms$$

(ii) * Average waiting time for FCFS

$$= \frac{((4-0)-4) + ((5-2)-1) + ((13-4)-8) + ((17-6)-4) + ((19-7)-2)}{5}$$

$$= \frac{20}{5} = 4\ ms$$

* Average waiting time for SJF

$$= \frac{((4-0)-4) + ((5-2)-1) + ((13-4)-8) + ((19-6)-4) + ((15-7)-2)}{5}$$

$$= \frac{18}{5} = 3.6\ ms$$

☆ Average waiting time for preemptive SJF

$$= \frac{((5-0)-4) + ((3-2)-1) + ((19-4)-8) + ((12-6)-4) + ((9+7)-2)}{5}$$

$$= \underline{2 \text{ ms}}$$

☆ Average waiting time for RR ($\delta = 2ms$)

$$= \frac{((5-0)-4) + ((3-2)-1) + ((19-4)-8) + ((15-6)-4) + ((11-7)-2)}{5}$$

$$= \underline{3 \text{ ms}}$$

Q6. part (a) at the END

~~Q6(a)~~

~~(i) FIFO (First in first out): The pages are kept in a linked list, with the newer pages kept at end of list.~~

~~When a replacement is needed, the oldest page is removed.~~

~~This process is fast, cheap and~~

Q6.

(b)

(i) During page fault, it would help if only had to write out pages that had changed (writing out unchanged pages is a waste of time).

Here, we add a dirty bit, which is set only when the page is changed, not written. Then, only the dirty pages are written.

(ii) There can be one or more low priority background tasks writing out changed pages and resetting dirty bit, as spare CPU resource is available (Page out task).

Q7. Suppose a process needs to access a page not in memory. If all the frames are already occupied, we do:

<u>Step 1</u>: The process query's its page table

<u>Step 2</u>: If the page is not found, it tells the OS about this using trap syscall, after saving its state.

<u>Step 3</u>: The OS will find the required dates in the backing store.

<u>Step 4</u>: It will try to find a frame to write the page.

<u>Step 5</u>: If no frame is there, the OS will choose a victim page. The

OS finds some page in main memory not really in use. This is done using algorithms like LRU, LRU approximations.

Step 6: It brings this page to the frame and updates the page table corresponding to the process.

Step 7: It restarts the instruction that caused the page fault. Other registers and information is loaded. OS returns to user mode and continues executing the process.

Q5.

(a) Page size $= 8KB = 8 \times 2^{10}$ bytes

Total memory addressable $= 2^{64}$ bytes

No. of virtual pages $= \frac{2^{64}}{2^{13}} = 2^{51}$ pages

Physical memory $= 128GB = 2^7 \times 2^{30}$

$= 2^{37}$ bytes

Assume

$\boxed{\text{Frame size} = \text{Page size}}$

No. of physical frames $= \frac{2^{37}}{2^{13}}$

$= 2^{24}$ frames

* Need 24 bits for addressing a frame

= 3 bytes

(Needs to be power of 2 so 4 bytes).

* Each page table entry is 4 bytes in size

Scanned with CamScanner

$\Rightarrow$ Page table size = (Number of pages) $\times$ (PTE size)

$= (2^{51} \times 4)$ bytes

$= 2^{53}$ bytes

which is quite high, so we cannot have 1-level page tables with 64 bit address space

(b) Virtual address space = 2^{64} bytes

Page size = 8KB = 2×10^{10} bytes

Number of frames = 2^{24} (Physical memory)

Logical address

p1	p2	d

$\leftarrow$ Page number $\rightarrow$ $\leftarrow$ offset $\rightarrow$

38 , 13 , 13

Size of outer page table = $2^{38} \times \underbrace{4 \text{ bytes}}_{\text{page entry size}}$

$= 2^{40}$ bytes

Size of inner page table

$= 2^{13} \times 4$ bytes

$= 2^{15}$ bytes

$= 32$ kilobytes

Q3. (a) Time slice = 50ms, Scheduling overhead = 3ms
Response time for 1st iteration: P1: 53 ms, P2: 106 ms; P3: 159ms, P4: 212 ms, P5: 265ms, P6: 318ms, P7: 371ms, P8: 424ms, P9: 477ms and P10: 530ms

⇒ Response time for subsequent iterations:

P1: 330ms
P2: 330ms
P3: 330 ms
P4: 330 ms
P5: 330 ms
P6: 330ms
P7: 330 ms
P8: 330 ms
P9: 330 ms
P10: 330 ms

b) time slice = 20ms

Scheduling overhead = 3ms

Response time for first iteration:

P1: $23 \times 10 \times 8 + 13 = 473$ ms

P2: 486 ms

P3: 499 ms

P4: 512 ms

P5: 525 ms

P6: 538 ms

P7: 551 ms

P8: 564 ms

P9: 577 ms

P10: 590 ms

Each process had CPU for 20, 20 & 10 ms durations:

Response time for subsequent iterations:

P1: 473 ms

P2: 473 ms

P3: 473 ms

P4: 473 ms

P5: 473 ms

P6: 473 ms

P7: 473 ms

P8: 473 ms

P9: 473 ms

P10: 473 ms

Q6.

(a) We can do the following to minimize page fault:

(i) <u>Prepaging</u>: This refers to bringing some of the required pages into memory before the process starts execution to prevent high page fault frequency initially. This can be implemented in the kernel for every process start and hence software.

(ii) <u>Increase page size</u>: We can increase page size so more data is present in each page, and sometimes data can directly be obtained from pages in memory (more probable with higher page size). Page size is a kernel parameter. Hence in software.

(iii) Increase TLB: The translation lookaside buffer is a cache which allows us to store frequently accessed pages. Increasing it will allow us to store more pages in TLB. Implemented in hardware

(iv) Locality of Reference: When a page fault occurs, we bring to the memory the required page and pages in its locality. Because it is very probable that pages in its locality will be needed soon. This can be implemented in software.

(v) Use better algorithms like LRU to remove pages if necessary. Implemented in software.