RADHIKA PATWARI

18CS10062

PAGE NO.
DATE: 10 / 2 / 2021

Take Home Assignment 1

1. a. 'Putting the cpu in privileged mode' is a priviledged instruction because allotment of processes to cpu and memory is done in kernel mode by the scheduler and dispatcher routine of kernel. If this is allowed in user mode then a process may enter an infinite loop and never allot the cpu to other processes. The kernel preempts a running process using the hardware timer interrupt and takes control of cpu. This prevents malicious codes and programs from accessing the core functionalities of the operating system and prevents illegal control of cpu and memory.

b. Memory protection is required to ensure that different process do not mix with each other's code or data.

Every memory address used by a process should be first checked to see whether it falls within the range of memory area that is alloted to the process. Two special purpose registers - lower bound register (LBR) and upper bound register (UBR) are used to implement memory protection by storing start and end process of the memory area alloted to that process. The kernel loads appropriate values to these registers during process execution. The memory protection hardware compares every address used by process with LBR and UBR values. If the address is smaller than address in

LBR or larger than value in UBR, a memory protection violation interrupt is raised. This prevents the process from reliably jumping to a vulnerable function in memory or accessing any unallocated memory area. This is thus a privileged instruction.

c. Loading values in CPU register should be priviledged instruction because when a user writes a program or uses an application, he does not think about the problems of memory allocation or how data is stored in registers during process execution. This enhances user experience. Thus kernel takes the job of loading and clearing cpu registers during context switching. It saves the content of current process in memory and loads the content of new process in cpu registers.

d. Disabling interrupt system should be priviledge instruction because malicious programs may take advantage of disabling interrupt and take full control of cpu or access unallocated memory area and hack the operating system. To prevent malicious activity from outside users, interrupts are handled by kernel or administrators.

e. Reading status of I/O device may not be a privilege instruction. The status registers of I/O contain bits that indicate whether the current command has executed, whether a byte is available to be read from the data-in register,

and whether there was a device error. Reading these data for any I/O output will not create any problem for the OS. Hence they may not be priviledged instructions.

---

2. When a program makes a request for memory, OS issues a syscall (malloc or new) and kernel allocates memory in stack or heap depending upon request (static or dynamic). During this, kernel may swap other process's memory in the heap to fulfil the memory request of this process. Now, making making requests for memory increases the overhead and process becomes slow. [kernel again and again checks RAM for memory, swaps processes in disk if required]. Instead we can combine all these requests into a single system call. The time taken by the kernel will be less as compared to when it was handling individual requests.

---

3. a. Throughput of a system depends on CPU speed, Instructions count and average number of clock cycles taken per instruction or I/O operation. For same set of instructions, on doubling CPU speed, the number of cache misses increases and more number of clock cycles are required to access data in main memory, then harddisk hence throughput doesn't get doubled.

b. On expanding memory to twice its size, more processes can be loaded into memory for execution. As multi-

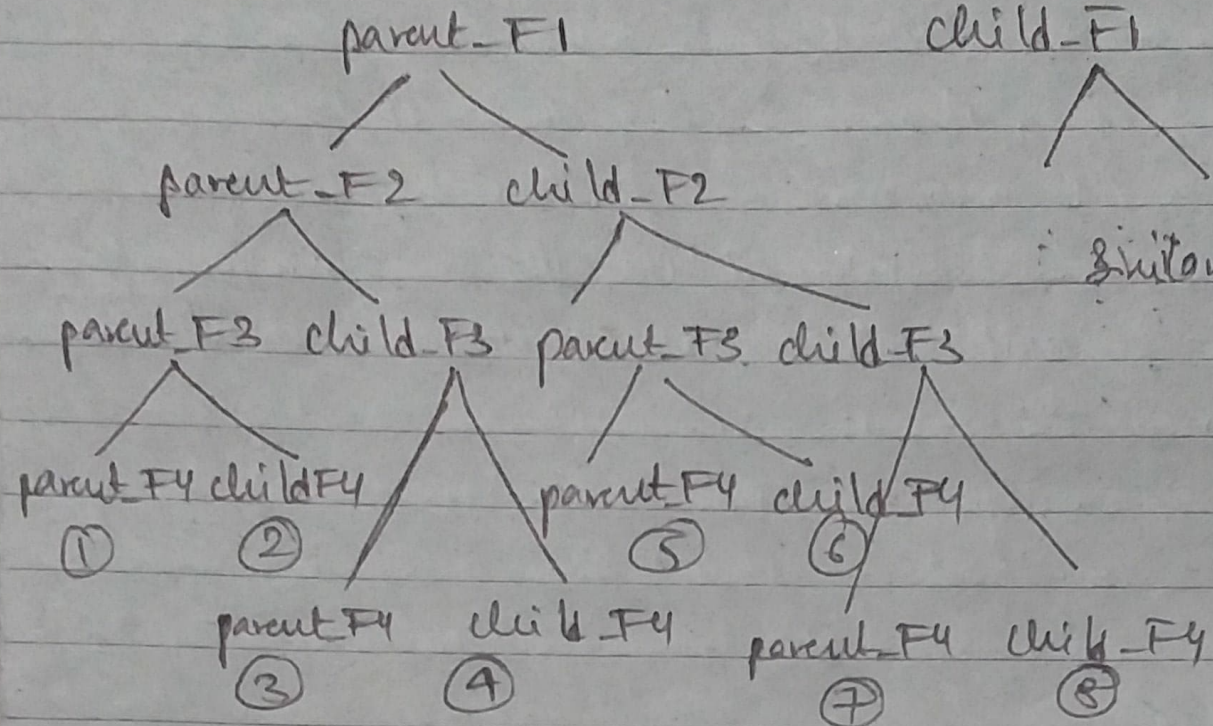
programming works well with a mixture of CPU bound & IO bound processes, it may happen that all processes present in small memory are I/O. In such case, more memory ensures equal proption of CPU & I/O bound processes. Hence throughput of the CPU will increase as CPU utilisation increases.

c. Adding new I/O devices using DMA mode does not affect the throughput of CPU. DMA allows data transfer between memory and device buffer without the involvement of CPU. On an average, it does not affect the overall CPU performance.

d. As main memory is doubled, more data can be stored in it. As cpu speed becomes twice, data can be accessed more frequently from larger memory. Hence overall throughput increases. But it does not get exactly doubled because throughput depends on how fast the data can be accessed from main memory and also on I/O operations.

4. 16 processes are created in total.

```
fork();   // F1
fork();   // F2
fork();   // F3
fork();   // F4
```



Initial 8 process.

= 16

5. System Call	Exception
• When the process wants some service from the OS, it makes a system call to use a kernel routine	• Occurs when a process is executing because of a system call, division by 0 or bad pointer reference synchronised with CPU instructions.
• process is aware that a sys call is about to occur.	• exception like division by 0 may come unexpectedly
• In the process state diagram, pcb shifts from Running to Waiting queue.	• Pcb may get terminated (move to terminated state) or to waiting queue

6. a. Average Service time = $\tau \times N$

Unit price of service time = S per user

$\therefore$ Cost of service time per user = $\frac{S}{\tau \times N}$

$\therefore$ cost of waiting time per user = $\frac{NW}{M}$

$\therefore$ Cost of (service time + waiting time) per user $C = \frac{S}{\tau N} + \frac{NW}{M}$

Minimising C, $\frac{dC}{dN} = 0$.

$\therefore 0 = -\frac{S}{\tau N^2} + \frac{W}{M} \Rightarrow \frac{W}{M} = \frac{S}{\tau N^2} \Rightarrow N^2 = \left(\frac{MS}{\tau W}\right)$

$\Rightarrow$

$N_{opti} = \sqrt{\frac{MS}{\tau W}}$ → optimised batch size.

Hence proved

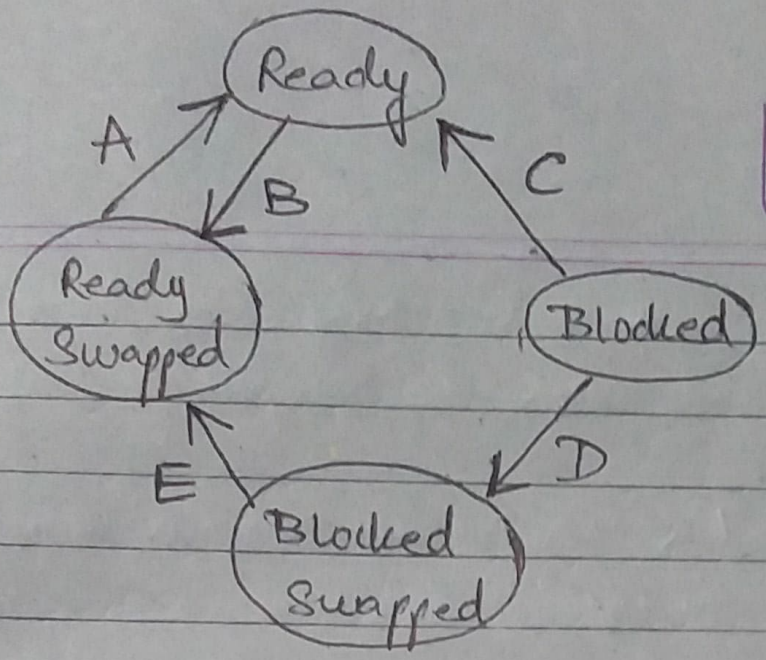
b. Given $M = 5$ min, $S = \$200/hr$

$\tau = 1$ min, $N = 50$, $W = ?$

$\therefore N^2 = \frac{MS}{\tau W} \Rightarrow 50 \times 50 = \frac{5}{1} \times \frac{200}{W}$ $/hr

$\Rightarrow W = \$0.4/hour$.

7.



PAGE NO.
DATE: / /

Transition A occurs: (swap in)

- there is enough space in main memory, so process is brought back from swap space (in disk) to main memory.
- process in 'ready swapped' is of higher priority, so it is replaced by some low priority process in 'Ready' state.
- CPU scheduler decides to execute a process and calls dispatcher. Dispatcher checks to see if this process is in main memory. If not, then it loads the process from swap space to main memory. In case there was no space in main memory, this process is swapped with some process currently in memory.

Transition B occurs: (swap out)

- Memory manager temporarily swaps out some process in backing store in disk if there is no memory for incoming processes.

- Some high priority process is in 'ready swapped'. So memory manager replaces and swaps a low priority process with this process.
- New process is loaded & insufficient main memory, an old process is swapped to disk.

Transition C occurs: (waiting for some event)

- I/O operation is over. So I/O interrupt is raised
- the process was waiting for some child process to complete and now child process has exit
- the process asked for a syscall() and now it is over, so kernel raises an interrupt.

Transition D occurs:

- While the process was waiting for some event (I/O request, syscall, child process), there is a new process loaded and no space in memory, so this process is moved to swap space by memory manager.
- There is no space for a higher priority process present in swap space. So it is brought into the main memory and 'blocked' process is sent to 'blocked swapped' queue.

Transition E occurs:

- process is waiting in swap space for i/o operation to complete. When I/O interrupt is raised, it is moved to 'ready swapped' state.

- Similarly, if a syscall operation is over and kernel raises an interrupt, process moved from 'blocked swapped' to 'ready swapped'
- Similarly, if process was waiting for some child process to exit and now it is over, it is moved from 'blocked swapped' to 'ready swapped' state.

8. All informations of a process are stored in Process Control Block (PCB). The entries stored in PCB are:

- Process state — Process may be in 'READY', 'RUNNING' 'WAITING', 'NEW', 'BLOCKED', 'BLOCKED SWAPPED' 'TERMINATED' or 'READY SWAPPED' states.

- Program Counter — Indicates next instruction to be executed for this process.

- CPU registers — The registers vary in number and type depending on the computer architecture. They include accumulators, index registers, stack

pointers and general purpose registers. This data is saved so that process can be continued correctly afterwards.

- CPU scheduling information — Stores information required during scheduling like process priority, pointers to scheduling queues and any other scheduling parameters.

- Memory management information — Stores information like value of base and limit registers, page tables, segment tables, depending on memory management used by operating system.

- Accounting information — Stores amount of CPU and real time used, time limits, account numbers, job or process numbers.

- I/O status information — Includes list of I/O devices allocated to the process, list of open files, etc.

→ Design Approaches for avoiding context switch overhead between processes :-

a) Context switching overhead can be minimised by modifying the architecture of the processor by adding

additional register files to the existing register bank of processor so that context can be saved on processor itself, thereby eliminating clock cycles for storing and restoring from memory. ~~is eliminated~~. Thus presumably rules out the extra time consumption due to memory to hardware transfers and vice versa, from overall execution time, thereby improving efficiency of OS.
This concept proves vital for Hard Real time systems where deadlines are to be met & output depends on both logic and the time at which it is produced.

•) The overhead can also be reduced by migrating kernel services such as scheduling, time tick (a periodic interrupt to keep track of time during which the scheduler makes a decision) processing and interrupt handling to hardware.
Currently scheduling coroutines run as a part of kernel which requires the processor. Just like the Hardware time and Direct Memory Access (DMA), Scheduling algorithms & interrupt handling can be moved to hardware components using suitable architecture.

PAGE NO.
DATE: / /

9. CPU-bound processes → Heavy mathematical computation like matrix multiplication, calculating factorials; video streaming

I/O bound processes → Copying/Moving/Transfering/Downloading files.

10. Short-term Scheduler	Medium-term Scheduler	Long-term Schedular
• Also called CPU scheduler. It selects from a group of processes ready to execute and allocates CPU to one of them using dispatcher.	Also called swapping scheduler. To make space for other process in main memory, it swaps 'waiting' processes into the secondary storage.	Also called Job Scheduler. It selects processes from queue & loads them into memory for execution.
• Speed is fastest compared to long and medium term schedulers.	Medium speed	Speed is less than short term scheduler
• It is insignificant in time sharing order.	This scheduler is an element of time sharing system.	It is either absent or minimal in time-sharing system

11. /*

Sharing Memory between processes
I have created a parent process that forks 2 child processes that share the common memory segment created by the parent. The child processes have identical copy of variables as the parent process and can update their own copy of variables in their own way, as they don't actually share variables. But they can share memory, so that when parent process makes change, the other processes can see the change.

*/

```
# include <stdio.h>          /* C header file */
# include <stdlib.h>         /* C header file */
# include <unistd.h>         /* for fork() */
# include <sys/wait.h>       /* for using wait() syscall */
# include <sys/ipc.h>        /* for inter process communication */
# include <sys/shm.h>        /* for enabling sharing
                                memory segment
                                Creation */

int main()
{
    int shmid;              // store shared memory
                               identifier
```

```
int status;          // Store status of child processes
int *a, *b, *c;      /* Using a shared memory for 2 elements
                        integer array, Create 3 array pointers
                        for the parent and 2 child processes
                                    respectively */

pid_t pid1, pid2;    // Storing process identifiers for 2 forks
int i;               // loop variable

/* Operating system keeps track of the set of shared memory
segments. In order to acquire shared memory, shmget()
syscall is used. The parameter IPC_PRIVATE creates a new
private memory segment. The second parameter determines the
size of the shared memory segment - (here, 2 * sizeof(int)
for a 2 integers array. The third parameter identifies
the mode (reading, writing operation on the shared memory
segment). The syscall returns the segment identifier which
is required by process that want to access this region of
shared memory.
*/

shmid = shmget (IPC_PRIVATE, 2*sizeof(int), 0777|IPC_CREAT);
/* Requesting array of 2 integers */
/* shmget() syscall returns negative value, if memory
allocation fails */
```

```
if (shmid < 0) {
    printf (" Error in Shared Memory Creation");
    exit(1);        //returning error code
}

printf (" Succesfully created shared memory ");

// Creating 4 processes using 2 fork calls
// 1 parent : 2 child processes : 1 grand-child process
// Using fork() syscall

pid1 = fork();
pid2 = fork();

if (pid1 < 0) {
    printf (" Fork 1 failed ");
    exit(1);            //returning error code 1
}
if (pid2 < 0) {
    printf (" Fork 2 failed ");
    exit(1);            // returning error code 1
}

/* Parent Process */
if ( pid1 > 0 && pid2 > 0) {

    printf (" Parent Process started \n");
```

/* To access the shared memory segment, the process need to attach it to its address space using shmat() syscall.
The 1st parameter is the segment identifier.
The 2nd parameter is pointer location to memory where shared memory will be attached. NULL means the OS will select the location on user's behalf.
The 3rd parameter allows shared memory region to be attached in read only or write only mode. By passing 0, we allow both read & write to the shared region.
*/

```
a = (int *) shmat (shmid, 0, 0);

/* shmid returns char * by default. We typecast it into
   int * */
printf (" Shared Memory attached to Parent ");

a[0]=0; a[1]=1;
for (i=0; i<10; i++) {
    sleep(1);    // syndronising the parent & child
                 // process so that when parent updates
                 // child is able to read it & vice versa
    printf(" Parent reads : %d, %d \n", a[0], a[1]);
    a[0] = a[0]+1;
    a[1] = a[1]+1;
    printf (" Parent writes : %d; %d \n", a[0], a[1]);
}
```

```
wait(&status);    // waiting for the two childs
wait(&status);    // to complete

/* the process need to detach itself from shared
   memory after it is used */

shmdt(a);    // using shmdt() syscall

/* Childs have exited, so parent process should delete the
created shared memory. Unlike attach & detach which
need to be done for each process separately, deleting the
shared memory has to be done by only one process after
making sure no one else will be using it */

shmctl(shmid, IPC_RMID, 0);    // clearing the segment
                               // then deleting it
}    // closing parent process

else if (pid1>0 && pid2==0) {

    printf("First Child Process\n");
    // attached to shared memory segment
    b=(int*) shmat(shmid, 0, 0);
    printf("Shared Memory attached to First Child");
```

```
for(i=0 ; i<10 ; i++) {
    sleep(i);    // synchronising with parent
    printf("First child reads : %d, %d \n", b[0], b[1]);
    b[0] = b[0]+1;
    b[1] = b[1]+1;
    printf("First child writes: %d, %d \n", b[0], b[1]);
}
shmdt(b);    // detaching from shared memory
exit(0);     // successfully exiting
}            // closing 1st child process

else if (pid1==0 && pid2>0) {
    printf(" Second Child Process ");
    // attached to shared memory segment
    c = (int *) shmat(shmid, 0, 0);
    printf(" Shared memory attached to 2nd child ");
    for(i=0; i<10; i++) {
        sleep(i);  // synchronising with other processes
        printf(" Second child reads : %d, %d \n",
                                c[0], c[1]);
        c[0] = c[0]+1;
        c[1] = c[1]+1;
        printf(" Second Child writes : %d, %d \n",
                                c[0], c[1]);
    }
    shmdt(c);    // detaching from shared memory
```

```
        exit(0);        // succesfully exiting
    }                   // closing 2nd child process

    else {
        printf("Grand Child process");

        /* We need to do computation in 2 child processes
           only in this question; hence grand child process
           does nothing and exits */

        printf("I did nothing");
        exit(0);        // succesfully exiting
    }                   // closing grand child process.

    return 0;           // returning from main()

}                       // closing main() function.
```

12. In multi-tasking Operating Systems, context switch occurs very frequently because of time slicing or interrupt or process termination. It adds an extra overhead to the throughput of the OS and hence should be as fast as possible. The instructions that are used to select the process to be executed next are located at a fixed address in memory to reduce overhead of searching for these instructions as they are needed too frequently.

13. Data Structures used for representing the linking of processes are:-

a) Queue (implemented as a singly linked list with a head and tail pointers)

Pros → Processes get stored in the ordered sequence of their arrival. Each PCB has an extra parameter to point to the PCB of the process that should be handled next in the sequence. Thus only one 'head' need to be maintained to get all the processes. Also variable no. of processes can be attached.

Cons → Some processes may be of higher priority like some trusted kernel coroutines. On storing processes as a FIFO structure (normally) the priority gets lost.

b) Priority Queue (head pointing to highest priority)

Pros → Processes are ordered on basis of priority and alloted to the CPU in this order. New process/Partially executed process is stored at the list depending on its priority. Any variable number of processes can be linked together.

Cons → Adding a new process/partially executed process into the list will take a best case complexity of $O(\log n)$ where n is number of processes already present in queue. But in a singly linked list, it was $O(1)$. So using priority queue may add an extra overhead to the thruput of the OS.

c) Max Heap (implemented as binary trees) (based on priority)

Pros → Highest priority process is given max weitage. A process can be accessed in $O(1)$ time (max priority → so root of the heap)

Cons → Adding a new/already executed process back may take $O(\log n)$ time where singly linked list (queue) took only $O(1)$.

Data structure implementation of trees is comparitively difficult as every node requires 2 pointers to store the left and right subtrees. This increases the memory complexity for large number of processes.