

Portfolio

Week 1.

6) *How does the distinction between supervisor mode and user mode function as a rudimentary form of protection (security) system?*

- User mode only allows subsets of a program. Some specific instructions can only be executed in supervisor mode. The supervisor mode allows access to all memory areas (OS and programs).

7) *What are the differences between a software interrupt (trap) and a hardware interrupt? What is the use of each?*

- Trap (also named software interrupt or programmed exceptions) is a kind of exceptions that may be caused by instructions/errors made by another software (i.e. division by zero). It can also occur when a process has a higher priority than the current: the higher can access to the CPU directly. This interrupt is used to implement system calls.

- Hardware interrupt occurs when a fault/problem/error comes from an hardware device and needs to be taken in care. (i.e. power off/new device plugged in/overheat...)

Week 2.

4) *What is the purpose of system calls? Why are they necessary?*

- The purpose of system calls is to control or ask for an action to the operating system. The user can this way make requests form applications to the system securely. It is a typical way to use software interrupt or trap.

Week 3.

Virtual machine exercise:

16) `sort -otest< test
head -n5 < test`

Tutorial question:

2) *Describe the differences among short-term, medium-term, and long-term scheduling.*

- Long term scheduler:

Also known as CPU scheduler, this scheduler determines which program (waiting in memory) is allowed by the system to process. Once admitted, a job becomes a process in the CPU. This selects which process has to be sent to the ready queue (often in a batch system).

- Short term scheduler:

It occurs when a process is in a blocked or suspended state. This selects which process has to be executed and sent to the CPU. To sum up, this is the decision of which available process will be executed by the CPU

- Medium term scheduler:

Also know as a dispatcher or memory manager, it executes most frequently. This makes the finest-grained decision of which process ready or blocked should be executed later. This scheduler makes the decision to reinstates them later. It may lead to interruption of one process by pre-emption (for a privileged process).

For example, if too many processes are requesting for a resource, and there's next a slow system time response, some of these can be paused to let other process run.



Week 4.

1) Give two advantages threads have over multiple processes? What major disadvantage do they have? Suggest one application that would benefit from the use of threads, and one that would not.

- The threads use fewer resources when they are executed and are fairly easy to program. Although they do not have their proper memory space, they all use CPU time. One inconvenient of threads is that they have the possibility of damage shared data, and we must program it with caution. The same thing can not occur between the processes, except in special cases. Any program that executes more than one task at a time can use multitasking, for example a program observing the time of execution and the CPU usage of several programs (*Task Manager* of *Windows*). The opposite of multitasking is called "Single-minded" processes, for example, a program which counts the time before the automatic shutdown of a computer (*Auto Shutdown* by *Konradp*).

6) Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

The processes are assumed to have arrived in the order P1,P2,P3,P4,P5, all at time 0.

a) Draw four Gantt charts illustrating the execution of these processes using FCFS, SJF (nonpreemptive), nonpreemptive Priority (with a smaller priority number implying a higher priority), and RR (quantum = 1) scheduling.

First Come First Serve Scheduling

P1	P2	P3	P4	P5
0	10	11	13	14
				19

Non Preemptive Scheduling

P2	P5	P1	P3	P4
0	1	6	16	18
				19

Shortest-Job-First Scheduling

P2	P4	P3	P5	P1
0	1	2	4	9
				19

Round-Robin Scheduling

P1	P2	P3	P4	P5	P1	P3	P5	P1	P5	P1	P5	P1	P5	P1
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
														19

b) What is the turnaround time of each process for each of the scheduling algorithms in part a)?

First Come First Serve

P1(Prio=16)	P2(Prio=1)	P3(Prio=18)	P4(Prio=19)	P5(Prio=6)
10	11	13	14	19

Round-Robin

P1(Prio=16)	P2(Prio=1)	P3(Prio=18)	P4(Prio=19)	P5(Prio=6)
19	2	7	4	14

Shortest Job First

P1(Prio=16)	P2(Prio=1)	P3(Prio=18)	P4(Prio=19)	P5(Prio=6)
19	1	4	2	9

c) What is the waiting time of each process for each of the scheduling algorithms in part a)?

First Come First Serve

P1(Prio=6)	P2(Prio=0)	P3(Prio=16)	P4(Prio=18)	P5(Prio=1)
0	10	11	13	14

Round-Robin

P1(Prio=6)	P2(Prio=0)	P3(Prio=16)	P4(Prio=18)	P5(Prio=1)
9	1	5	3	9

Shortest Job First

P1(Prio=6)	P2(Prio=0)	P3(Prio=16)	P4(Prio=18)	P5(Prio=1)
9	0	2	1	4

d) Which of the schedules in part a results in the minimal average waiting time (over all processes)?

- The Shortest Job First scheduling

Week 5.

Tutorial question:

7) Suppose we want to execute the statements *S1*, *S2*, and *S3* in sequence, but that *S2* has to be executed exclusively for one process at a time. Write the code needed using semaphores.

```
Q:=1;
S1;
wait(Q);
S2;
signal(Q);
S3;
```

Virtual machine exercises with Linux processes.

16)

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>

main()
{
    pid_t val;
    int ODD = 1, EVEN = 0, n;

    printf("PID before fork(): %d\n", (int)getpid());
    if(val=fork())          //process 1 : Parent
    {
        printf("PID after fork(): %d - The parent\n",
(int)getpid());
        for(;;)//Infinite loop because the CPU goes too
fast, we now see "Parent" after the fork
        {
            printf("Parent\n");
        }
    }
    else                    //process 2 : Child
    {
        printf("PID after fork(): %d - The child\n",
(int)getpid());
        for(;;)//Infinite loop because the CPU goes too
fast, we now see "Child" after the fork
        {
            printf("Child\n");
        }
    }
}
```

21)

```
#include<stdio.h>
#include<sys/types.h>
#include<unistd.h>
#include<sys/wait.h>

main()
{
    pid_t val;
    int ODD = 1, EVEN = 0, status;

    printf("PID before fork(): %d\n", (int)getpid());
    if(val=fork())          //process 1 : Parent
    {
        printf("PID after fork(): %d - The parent\n",
(int)getpid());
        wait(&status);
    }
    else                    //process 2 : Child
    {
        printf("PID after fork(): %d - The child\n",
(int)getpid());
        execl("/bin/ls", "ls", "-l", 0);
        status;
    }
}
```

Week 6.

Tutorial question:

5) Consider the following resource-allocation policy. Requests and releases for resources are allowed at any time. If a request for resources cannot be satisfied because the resources are not available, then we check any processes that are blocked, waiting for resources. If they have the desired resources, then these resources are taken away from them and are given to the requesting process. The vector of resources for which the waiting process is waiting is increased to include the resources that were taken away.

For example, consider a system with three resource types and the vector Available initialized to (4,2,2). If process P0 asks for (2,2,1), it gets them. If P1 asks for (1,0,1), it gets them. Then, if P0 asks for (0,0,1), it is blocked (resource not available). If P2 now asks for (2,0,0), it gets the available one (1,0,0) and one that was allocated to P0 (since P0 is blocked). P0's Allocation vector goes down to (1,2,1), and its Need vector goes up to (1,0,1).

a) Can deadlock occur? If so, give an example. If not, which necessary condition cannot occurs?

	Allocation	Status	Available
	A B C		A B C
P0	2 2 1	Gets	4 2 2 (-221=201)
P1	1 0 1	Gets	2 0 1 (-101=100)
P0	0 0 1	Blocked	1 0 0 (+221=321 : P0 blocked)
P2	2 0 0	Gets	3 2 1 (-200=121)

Since P0 has been blocked when resources were not available, we use pre-emption. That's why deadlock can't occur!

b) Can indefinite blocking occur?

Unfortunately, it can! In a), since the resources were lacking, P0 could not be executed. Suppose that P0 requests 3 resources: if at least one such resource is locked by other preempted processes, each has attempted execution; P0 will be blocked and may never occur!

Virtual machine exercises – simple windows system programming.

14)

```
#include <windows.h>
#include <stdio.h>
#define BUF_SIZE 2048
int main (int argc, LPTSTR argv[])
{
    HANDLE hIn, hOut;
    DWORD nIn, nOut;
    CHAR Buffer [BUF_SIZE];
    if(argc !=2)
    {
        printf("Usage: cp file1 file2\n");
        return 1;
    }
    hIn = CreateFile (argv[1],
    GENERIC_READ,FILE_SHARE_READ,NULL,OPEN_EXISTING,FILE_ATTRIBUTE_
    _NORMAL,NULL);
    if(hIn==INVALID_HANDLE_VALUE)
    {
        printf("Cannot open input file. Error: %x\n",
        GetLastError());
        return 2;
    }
    while (ReadFile(hIn,Buffer,BUF_SIZE,&nIn,NULL)&&nIn>0)
    {
        hIn=GetStdHandle(STD_OUTPUT_HANDLE);
        WriteFile(hIn, Buffer, nIn, &nOut, NULL);
        if(nIn != nOut)
        {
            printf("Fatal write error:%x\n",GetLastError());
            return 4;
        }
    }
    CloseHandle(hIn);
    CloseHandle(hOut);
    return 0;
}
```

Week 7.

Tutorial questions

4) Given memory partitions of 100K, 500K, 200K, 300K, and 600K (in order), how would each of the First-fit, Best-fit, and Worst-fit algorithms place processes of 212K, 417K, 112K, and 426K (in order)? Which algorithm makes the most efficient use of memory?

	First Fit	Best Fit	Worst Fit
100K	100	100	100
	212	417	417
500K	112	83	83
	166	112	200
200K	200	88	300
300K	300	212	300
		88	
600K	417	426	212
			112
	183	174	276
Wait Queue	426		426
Memory Left	949	533	959

“Best fit” fits the best ;) !

Virtual machine exercise:

14)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <fcntl.h>
#define SIZE 26          //Alphabet

main()
{
    int i, count, fda[2];
    char buf [SIZE];
    char ch;

    if (pipe(fda)==-1)          //Share the data
        printf("error creating pipe\n");
```

```

if (fork()) //Process 1 : Parent
{
    close (fda[0]);
    for(i=0;i<26;i++)
    {
        buf[i]=i+97;           //Letters
    }

    if(write(fda[1],buf,SIZE)!=SIZE)
        printf("error writing to pipe\n");
}
else //Process 2 : Child
{
    for(;;) //Infinite Loop
    {
        close (fda[1]);
        if (read(fda[0],buf,SIZE)==-1) //Get fda via buffer
            printf("error reading pipe\n");
        for(i=0;i<26;i++)
        {
            printf("%c",buf[i]);
        }
    }
}
}

```

Week 8.

Tutorial question.

5) Consider a demand-paged computer system where the degree of multiprogramming is currently fixed at four. The system was recently measured to determine utilization of CPU and the paging disk. The results are one of the following alternatives. For each case, what is happening? Can the degree of multiprogramming be increased to increase the CPU utilization? Is the paging helping?

a. CPU utilization 13 percent; disk utilization 97 percent

We call it “Trash”: the hard disk drive is moving data between the system memory and virtual memory excessively, it becomes overworked! There’s too much program running: you need to decrease the multiprogramming!

b. CPU utilization 87 percent; disk utilization 3 percent

The CPU is very busy but the disk isn’t in use. We can of course increase the multiprogramming, but we have to be careful and do not exceed the disk utilization : this would lead to trashing!

c. CPU utilization 13 percent; disk utilization 3 percent

We are here in a normal situation: both CPU and disk are idle. Indeed we can easily increase multiprogramming.

Week 9.

Tutorial questions.

3) Consider a file currently consisting of 100 blocks. Assume that the file control block (and the index block, in the case of indexed allocation) is already in memory. Calculate how many disk I/O operations are required for contiguous, linked, and indexed (single-level) allocation strategies, if, for one block, the following conditions hold. In the contiguous-allocation case, assume that there is no room to grow in the beginning, but there is room to grow in the end. Assume that the block information to be added is stored in memory.

a. The block is added at the beginning. (1st block)

Contiguous	Linked	Indexed
We need to write 100 blocks. After that, we firstly read then write the 100 present blocks one block on the left (100+100 I/O). After that, we write the new block in the first block (+1 I/O) we previously made free. 100+100+1= 201 I/Os	We already have a sequence of 100 blocks linked to each other. All we have to do is to create (+1 I/O) the new block wherever we want and it is automatically linked to the previous first one. 1 = only 1 I/O	After we wrote (+1 I/O) the new block, we just have to put a new link to the top of the index table and move down all the previous entries. Because it is in the memory, no I/O is performed concerning the index table. 1 = only 1 I/O

b. The block is added in the middle. (50th block)

Contiguous	Linked	Indexed
Similarly to the first block, we need to read then write blocks, but only 50 (50+50 I/O). After that, we write the new block in the first block (+1 I/O) we previously made free. 50+50+1= 101 I/Os	We already have a sequence of 100 blocks linked to each other. All we have to do is to scan the 50 first blocks (50 I/Os). Then we create (+1 I/O) the new block wherever we want and it is automatically linked to the previous 51 st block. After that, we need to modify the link of the old 50 th block to the new block (+1 I/O) 50 + 1 + 1 = 52 I/Os	Similarly to the 1 st block, after we wrote (+1 I/O) the new block, we just have to put a new link to the 51 st entry of the index table and move down all the next entries. 1 = only 1 I/O

c. The block is added at the end. (101st block)

Contiguous	Linked	Indexed
We need to read 100 blocks to get to the end (100 I/Os). After that, we write the new block in the first block (+1 I/O) to the next block (that is supposed free). 100+1= 101 I/Os	We already have a sequence of 100 blocks linked to each other. All we have to do is to scan the 100 first blocks (100 I/Os). Then we create (+1 I/O) the new block wherever we want and it is automatically linked to nothing (last block). We next have to do modify the link of the old 100 th block to the new block (+1 I/O). 100+1+1 = 102 I/Os	Similarly to the 1 st block, after we wrote (+1 I/O) the new block, we just have to put a new link to the end of the index table. 1 = only 1 I/O

d. The block is removed from the beginning.

Contiguous	Linked	Indexed
We need to remove 1 block (1 I/O). After that, the old 2 nd block becomes the 1 st block of the file. 1 = only 1 I/O	We have a sequence of 100 blocks linked to each other. All we have to do is to remove the first blocks (1 I/O), then the old 2 nd block becomes the 1 st of the file. 1 = only 1 I/O	After we removed the 1 st block from the index table (no need to delete on the disk, the block is considered free and we will be able to directly re-write), the old 2 nd block becomes the 1 st of the index table by moving up all the other entries of the index table. no I/O

e. The block is removed from the middle.

Contiguous	Linked	Indexed
We need to read the 50 first blocks and then remove the 51 st block (1 I/O). After that, we move up the 49 other blocks by reading then writing all of the 49 (+98 I/Os). Of course we don't forget to delete the old 100 th block (+1 I/O) 1+98+1 = 100 I/Os	We need to read the 50 first blocks (50 I/Os) and then remove the 51 st block (1 I/O). After that, the old 50 th block has to be re-linked to the old 52 nd block of the file (+1 I/O). 50+1+1 = 52 I/Os	We read the 50 first blocks in the index table, we remove the 51 st then the 49 next entries have to be moved up in the index. no I/O

f. The block is removed from the end.

Contiguous	Linked	Indexed
We need to read 99 blocks (99 I/Os) remove the 100 th block (1 I/O). 99+1 = 100 I/Os	We have a sequence of 100 blocks linked to each other. All we have to do is to remove the last block (1 I/O), then the old 99 th block becomes the end of the file. 1 = only 1 I/O	After we removed the last block from the index table the old 99 th block becomes the last entry of the index table. no I/O

8) What problems might arise when a file is deleted, if it is shared?

A lot of problems! If there is no secure automatism to prevent this, some user could delete an original file, making it unavailable to other users!

9) How can we solve this problem?

What we could imagine is a copy or a pointer on the original file that couldn't be deleted by anyone, letting this file available to everyone. Another way to ensure that people really want to delete that file could be a counter of the number of deleting calls that has been made on a file : when the count is zero, the file is voluntarily automatically deleted.