

Chung

Fabien

Portfolio

Week 1

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

In supervisor mode, we have privileged instructions. Only in this mode, we can execute all of this. In this way, the operating system has the control of the entire system.

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

A trap is a software-generated interrupt. An interrupt can be used to signal the completion of an I/O. For example, when you try to divide by 0 that generates an error: a trap.

Hardware interrupt is an interruption created by the hardware. An interrupt handler is summoned to deal with the cause of the interrupt, after treating the error, control returned to the interrupted context and instruction

Week 2

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

System calls give an interface to communicate with the operating system kernel. It's a simple interface to the application programs to interact with the hardware. The system call provides that interface.

Week 3.

Tutorial question:

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

Short term: selects job from ready queue and send it for execution

Medium term: selects processes from the ready or blocked queue and removes them from memory, and recover it later to continue running.

Long term: determines which jobs are sent into the system for processing.

Week 4.

Tutorial questions:

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

Threads are very cheap to create and destroy, and they use very little resources while they exist. They do use CPU time for instance, but they don't have totally separate memory spaces. Unfortunately, threads can damage shared data. So all of them has to take care of themselves. That could not happen between processes unless they used a system feature to allow them to share data. Any program that is doing more than one task once could benefit from multitasking. For example, if we need to read input, to use it and output we can use three different threads, one for each task. On the other side, If we have "Single-minded" processes, multiple threads are not very useful. (for example if you have a meter).

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.

FCFS

We are looking only the number of the process

P1	P2	P3	P4	P5
0 ms	10 ms	11 ms	13 ms	14 ms
19 ms				
10 ms	1 ms	2 ms	1 ms	5 ms

SJF

Whoever has the smallest burst goes first

P2	P4	P3	P5	P1
0 ms	1 ms	2 ms	4 ms	9 ms
19 ms				
1 ms	1 ms	2 ms	5 ms	10 ms

Nonpreemptive priority

The lowest priority goes first

P2	P5	P1	P3	P4
0 ms	1 ms	6 ms	16 ms	18 ms
19 ms				
Priority 1	Priority 2	Priority 3	Priority 3	Priority 4

RR

The process enters in order but remain just 1 millisecond in execution time

P1	P2	P3	P4	P5	P1	P3	P5	P1	P5	P1	P5	P1	P5	P1	P1	P1	P1	P1	P1
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

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

	P1	P2	P3	P4	P5
FCFS	10	11	13	14	19
SJF	19	1	4	2	9
Nonpreemptive	16	1	18	19	6
RR	19	2	7	4	14

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

FCFS

P1=0, P2=10 ms, P3=11 ms, P4=13 ms, P5= 14 ms

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

$$\text{FCFS} = \frac{10+11+13+14+19}{5} = 13.4$$

$$\text{SJF} = \frac{19+1+4+2+9}{5} = 7$$

$$\text{Nonpreemptive} = \frac{16+1+18+19+6}{5} = 12$$

$$\text{RR} = \frac{19+2+7+4+14}{5} = 9.2$$

The minimal result is obtained by SJF.

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) Use your forktest.c program as the basis, but modify it by deleting the printf's that occur under the if and the else that output the PIDs and replace them with code for the parent to output the series of odd (in French 'impair') numbers in an infinite loop (e.g. 1, 3, 5, 7, 9,...) and for the child process to output the series of even (in French 'pair') numbers in an infinite loop (e.g. 2, 4, 6, 8, 10,...). You could do this by using a for loop under the relevant branch of the if..else statement.

I choose to use parents and child instead of odd and even because it's easier to see two different words than odd/even number.

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

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

```

printf("PID befor fork(): %d\n", (int)getpid());
if(val=fork())
{
    printf("PID after fork(): %d - so I am the parent ?\n",
(int)getpid());

    for(;;) // infinite loop
    {
        printf("Parent\n"); //CPU to quick, even with de for
loop, so Parent and Child are more explicite
    }

    for(n=0; n<500000000;n++); //waiting loop
}

else
{
    printf("PID after fork(): %d - so I am the child
?\n", (int)getpid());

    for(;;) // infinite loop
    {
        printf("Child\n");
    }
}
}

```

21)Up till now the parent process has continued to execute while the child process has executed. We can use the wait() system call to make the parent process suspend itself and wait until it is woken up by the termination of the child process.

So modify forktest4.c to make the parent process wait for the child. You will need to use the wait() system call.

You will need to add to the #include list

```
#include <sys/wait.h>
```

The wait() system call takes a pointer to an int as a parameter. It is the location where the exit status of the child will be placed when it terminates. So you will need to add an int declaration,

```
int status;
```

to your code. Then you can call wait() as

```
wait(&status);
```

This needs to go in the code that is to be executed by the parent process.

The & in the parameter gives the address of the variable status to wait() rather than the value.

```

#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 befor fork(): %d\n", (int)getpid());
    if(val=fork())
    {
        printf("PID after fork(): %d - so I am the parent ?\n",
(int)getpid());
        wait(&status);
    }

    else
    {
        printf("PID after fork(): %d - so I am the child
?\n", (int)getpid());
        execl("/bin/ls", "ls", "-l", 0);
        status;
    }
}

```

Week 6.

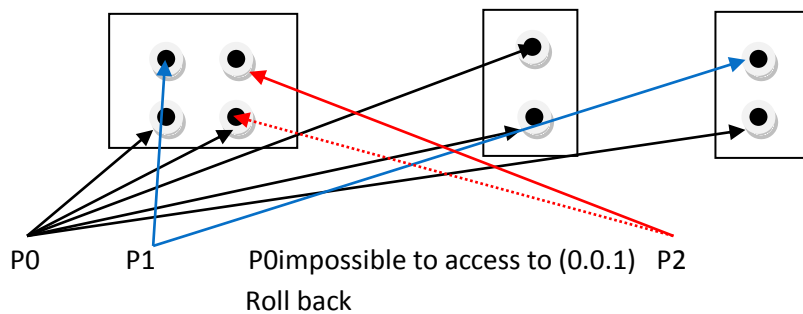
Tutorial question:

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 occur?

Deadlock cannot occur because preemption exists with P0 witch it can roll back.



b) Can indefinite blocking occur?

Yes, if it is necessary. A process may not access it needs if these resources are used/ preempted by other process. We will illustrate that.

Process	Allocation	Need	Available	Result
P0	2.2.1	0.0.1	1.0.0	Blocked
P2	0.0.0	2.0.0	1.0.0	Satisfied
P0	1.2.1	1.0.1	1.0.0	Blocked
P1	1.0.1	1.0.0	0.0.0	Satisfied
P0	0.2.1	2.0.1	0.0.0	Blocked

Virtual machine exercises – simple windows system programming.

14) Now we are in a position to modify the program above. We are going to change it so that now instead of copying a file from one file to another, the code will copy a file but output it to the console window. So it is a little like the DOS Type command.

To do this you need to know about one further function.

GetStdHandle – this returns a HANDLE and takes as an argument nStdHandle a DWORD that is the numerical value of the standard input/output stream. The permissible values are:

STD_INPUT_HANDLE – for keyboard input

STD_OUTPUT_HANDLE – for console output

STD_ERROR_HANDLE – error output (normally the console)

These identify which of the standard I/O values you need to use. You can now modify the code to send the file contents to the console screen.

a) Use WriteFile with the appropriate HANDLE you now get from GetStdHandle.

b) NOTE – YOU NEED TO DELETE (OR COMMENT OUT) THE CODE THAT CLOSES THE hOut handle. This is because it will close the console for output and so you won't get ANY further output to the console

c) You will have to change the number of arguments in the if statement that guards the usage prompt as the program will now be invoked with only one parameter, as

FileCopy <file to be copied to screen>.

```
/* Basic cp file copy program. Win32 Implementation. */
/* cp file1 file2: Copy file1 to file2. */

#include <windows.h>
#include <stdio.h>
#define BUF_SIZE 4096

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;
    }

    /*hOut = CreateFile (argv[2],GENERIC_WRITE, 0,
    NULL,CREATE_ALWAYS,FILE_ATTRIBUTE_NORMAL,NULL);
    if(hOut==INVALID_HANDLE_VALUE)
    {
        printf("Cannot open input file. Error: %x\n", GetLastError());
        return 3;
    }*/
    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?

For simplicity, we'll name each process:

212k P1

417k P2

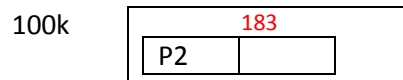
112k P3

426k P4

100k	
500k	
200k	
300k	
600k	

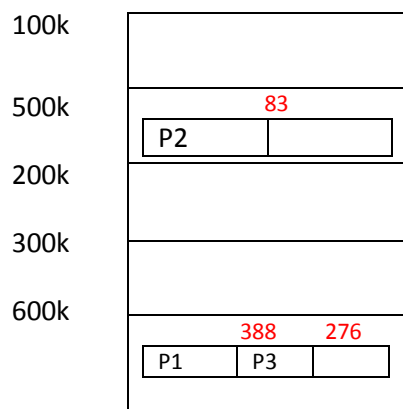
First fit: the process is put in the first hole which is enough large.

	288	176
P1	P3	



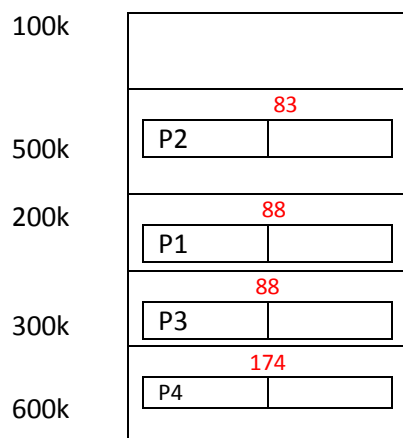
Not enough place for P4, it will wait

Best fit: the process is put in the largest hole



Not enough place for P4, it will wait

Worst fit: the process is put in the smallest hole where it can be put



Every process fit in the memory

Virtual machine exercise:

Linux interprocess communication.

14) Use pipes to communicate between the parent and child process.

Remember the parent process repeatedly writes the alphabet to the pipe using an infinite loop and the child process repeatedly reads from the pipe and actually outputting the characters to the screen.

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

#define SIZE 26

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

    if (pipe(fda)==-1)
        printf("error creating pipe\n");
    if (fork()) //parent process
    {
        close (fda[0]);
        for(i=0;i<26;i++)
        {
            buf[i]=i+97;
        }

        if(write(fda[1],buf,SIZE)!=SIZE)
            printf("error writing to pipe\n");
    }
    else //child process
    {
        for(;;)
        {
            close (fda[1]);
            if (read(fda[0],buf,SIZE)==-1)
                printf("error reading pipe\n");

            for(i=0;i<26;i++)
            {
                printf("%c",buf[i]);
            }
        }
    }
    for(i=0;i<50000;i++); //infinite loop it is a wait time
}
```

A revoir pour comprendre

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

It's thrashing, too more resources are used. That means RAM is too light or the degree of multiprogramming is too high. That creates many page faults, so access disk. And the access to the disk is slower than the access in the main memory. It isn't a good state.

b. CPU utilization 87 percent; disk utilization 3 percent

The CPU frequency is high enough, but we can increase a little the degree of multiprogramming.

c. CPU utilization 13 percent; disk utilization 3 percent

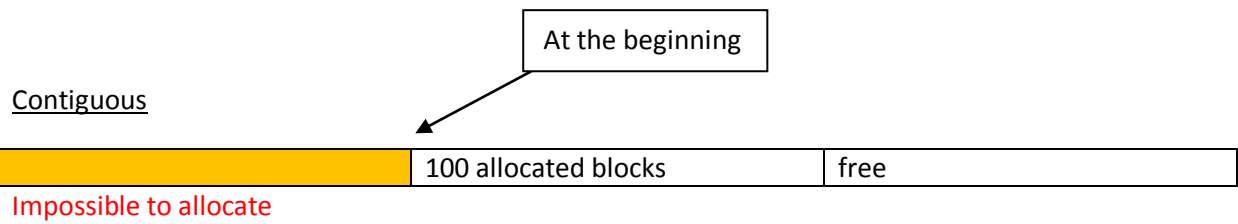
In this case, we can significantly increase the CPU utilization, because in the majority of time it waits so increase the degree of multiprogramming is possible. But we don't do many access disk so the global performance is good.

Week 9.

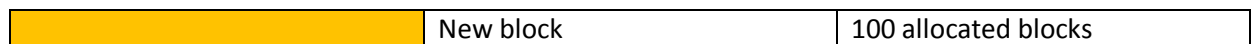
Tutorial questions.

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.

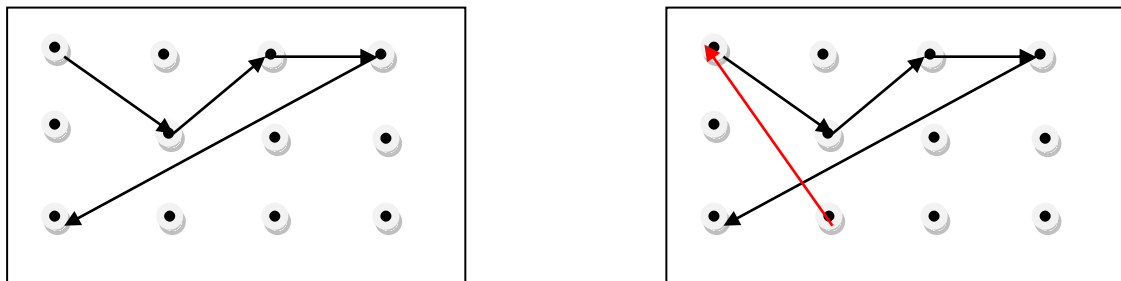
a. The block is added at the beginning.



We shuffle the 100 blocks in the right. 1 I/O for read, 1 I/O for write. So we need 200 I/O and one more for writing at the beginning.

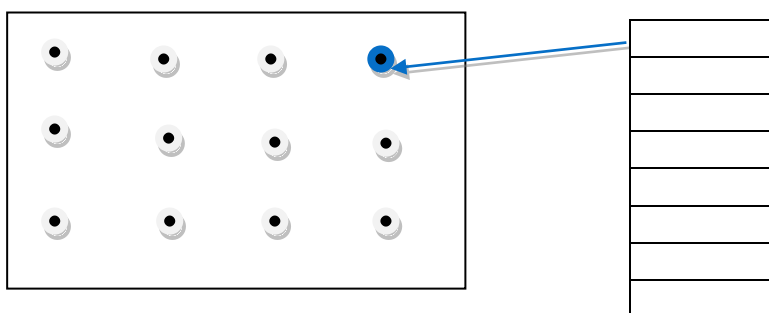


Linked



When we create a new block, the link with the next block is automatically created so it is only 1 I/O.

Indexed

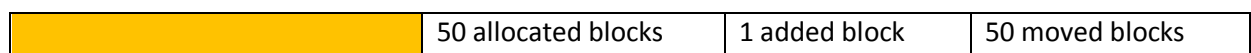


We create a new block 1 I/O. We need to move the entire indexed link. We consider that the index is in memory so no I/O are operated.

b. The block is added in the middle.

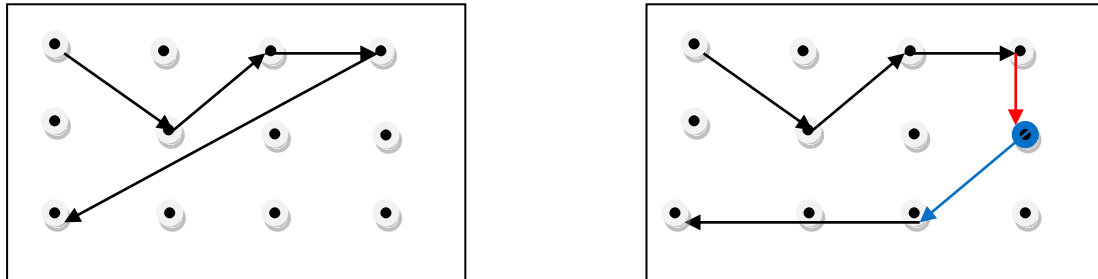
Similarly

Contiguous



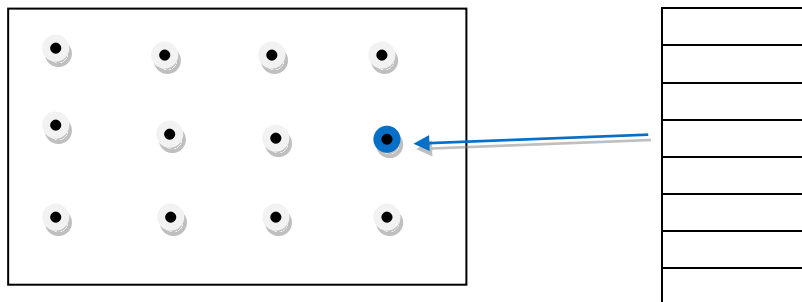
101 I/O

Linked



We need another I/O to link the previous block to the one. So we need 2 I/O.

Indexed



Similarly

We need only 1 I/O

c. The block is added at the end.

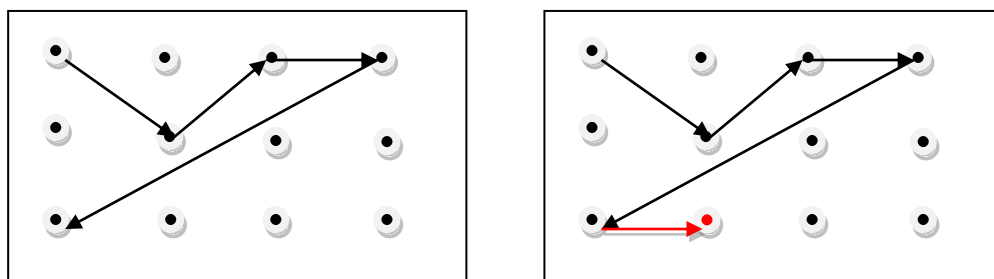
Similarly

Contiguous

	100 allocated blocks	1 added blocks	free
--	----------------------	----------------	------

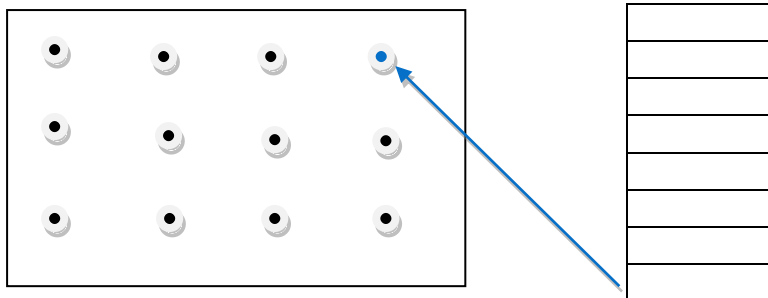
1 I/O

Linked



We need 1 I/O to add a new block and one to link the previous block to the new.

Indexed



Similarly

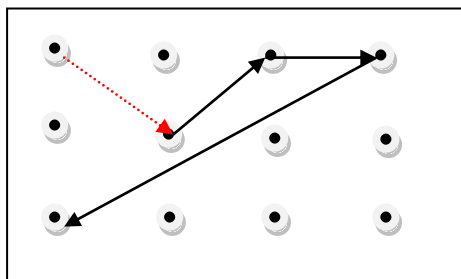
We need only 1 I/O

d. The block is removed from the beginning.

Contiguous

When we removed just one block so it is 1 I/O

Linked



We need just one I/O to remove the block

Indexed

We remove the first block but we don't need I/O

e. The block is removed from the middle.

Contiguous

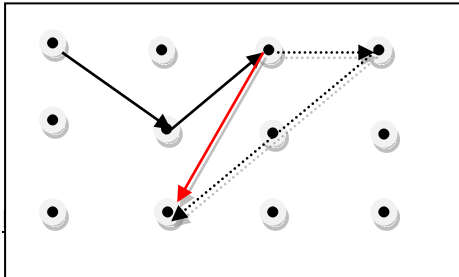
When we removed one block so it is 1 I/O

One block is removed

	50 allocated blocks	49 added blocks	free
--	---------------------	-----------------	------

But we need to move the 49 last blocks. At the end, we need 99 ($49 \times 2 + 1$) I/O.

Linked



We need 1 access to remove the block and another to link the previous and the next block

Indexed

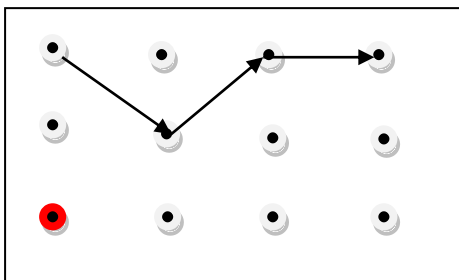
We remove the first block but we don't need I/O

f. The block is removed from the end.

Contiguous

When we removed just one block so it is 1 I/o

Linked



We need 1 I/O to remove the link to the previous link and one to remove the block

Indexed

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

If a file is deleted a file whereas another try to access to this file there will have an error.

9. How can we solve this problem?

Put a counter, for each user linked on the file the counter increase by one. When one user tries to delete the file the counter is decrease by one. Only when the counter equal to 0 the file is deleted.

	Contiguous	Linked	Indexed
A	201	1	1
B	101	2	1
C	1	2	1
D	1	1	0
E	9	2	0
F	1	2	0