I3303


جمعية طلاب النفس
كلية العلوم

Lebanese University	I3303- E- Operating system II	Final 19-20
Faculty of Sciences 1	Faour & Sbeity	Duration: 2 hours

Problem I — 10 points

Write a program to create 100 child processes. The parent process must then execute the function P(), and each of 100 children must execute the function F(). In addition, at each instant, at most 10 child processes can execute simultaneously the function F(), i.e., when 10 processes are in parallel execution of the function F(), any other process wishing to execute its function F() must be blocked until one of 10 processes is terminated.

Problem 2 — 30 points

A) Consider a system with physical memory composed of 4 frames. A program makes references to the following pages: **A-C-B-D-B-A-E-F-B-F-A-G-E-F-A**. Determine the number of page faults for the following replacement algorithms:

a) LRU
b) Optimal
c) Second chance

B) In a 32-bit machine we subdivide the virtual address into 4 pieces as follows:

8-bit 4-bit 8-bit 12-bit

We use a 3-level page table, such that the first 8 bits are for the first level and so on. Physical addresses (frame#, offset) are 44 bits and there are 4 protection bits per page (i.e., the PTE contains the 4 protection bits in addition to the frame number). Answer the following questions, showing all the steps you take to reach the answer.

1. What is the page size in such a system? Explain your answer.
2. How much entries can contain an inverted page table?
3. How much memory is consumed by the page table and wasted by internal fragmentation for a process that has 64K of memory starting at address 0?
4. How much memory is consumed by the page table and wasted by internal fragmentation for a process that has a code segment of 48K starting at address **0x1000000**, a data segment of 600K starting at address **0x80000000** and a stack segment of 64K starting at address **0xf0000000** and growing upward (towards higher addresses)?

C) Consider a logical address space of eight pages of **1024** bits each, represented in a physical memory of 32 frames. How many bits does the logical address contain? The physical address? Size of main memory? Explain

Problem 3 — 30 points

A) Consider a system with FAT where the index of a disk block is encoded on 24 bits. For a 32-GB disk determine:

a) The number of entries in the FAT table? Justify your answer.
b) The size of the FAT table?
c) The size of the disk block

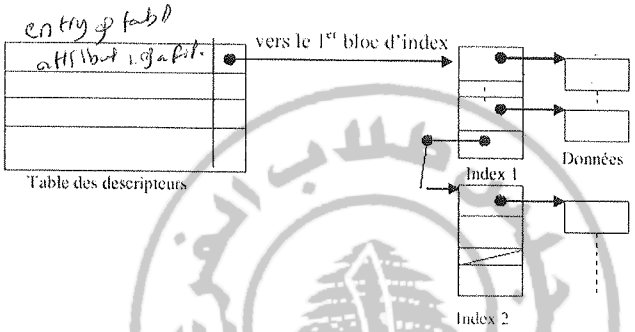
B) Consider an index-based file system with the i-node containing 64 direct indexes (numbers of direct data blocks), 1 indirect index pointing to a disk block containing an array of direct indexes, and 1 2-level index in the usual way. Assume that each index takes 4 bytes.

a) What is the maximum file size under this arrangement, if a disk block is 1024 bytes? Explain how do you compute this maximum size.

b) How many disk accesses does it take to read one disk block at location 3000321 bytes within a file, assuming no caching. Justify your answer.

C) Consider a file system with the following characteristics and illustrated in Figure:

- Size of a block is 8 KB.
- Linear numbering of disk blocks from 0 to n-1.
- Use of the indexed allocation technique with chaining for the management of physical blocks: all the descriptions of files created are grouped in the table of descriptors. Each entry in the table contains the attributes of a file and a pointer to the first index block. All pointers in an index block, except the last one, point to data blocks in the file. The last pointer to a full index block points to the next index block.
- Size of a pointer is 32 bits.



a) Calculate the maximum number of blocks a file can have by specifying the number of index and data blocks.

b) Determine the number of blocks that must be read from the disk in order to access data that is in the 9000th data block of the file. You can assume that there are K pointers per index block, the descriptor table is already loaded into memory, and the index and data blocks are not yet loaded into memory.

LEBANESE UNIVERSITY
Faculty of Sciences - Section I
Info 3303-324 - E

Final 2017-2018
Duration : 2 hours

Part I — Process Management — 20 Points

The goal of this problem is, starting from a **sorted set of integers**, filter it successively to keep only prime numbers. Consider integers from the number 2 (which is the first prime number). To build the rest of the prime numbers, let's first remove the multiples of 2 from the rest of the integers, we get a list of integers starting with 3 which is the next prime number. Let's eliminate the multiples of 3 from this list, which builds a list of integers starting with 5 and so on.

Write a C under UNIX program to make this filter where each filter is a process that reads a list of integers, displays the first prime number p on screen, and leaves the integers it has not filtered. The main process write the list of integers into a pipe.
Two solutions are requested as follows:

a) **Sequential processing**: at the beginning, only the main process is created, it creates the first child that displays the first prime number (that is 2) and filters its multiples, then the child process gives the hand to another child process created by itself and so on.

P0 P1 P2 Pn

b) Now, the main process creates a sufficient **number of processes** (to be determined) to display prime numbers between 2 and 120. These processes are in "pause" state after creation. The father process wakes up the first child process which displays the first integer first and eliminates its multiples. The first child wakes up the second child process to display the second prime number and filters its multiples and so on. The last process must display all prime numbers that do not have multiples in the list of integers between 2 and 120.

P0

P1 P2 Pn

P.S: remember that reading from pipe is erasing

Part II — File Management — 25 Points

Given a FS (File system) where the table topo contains 10 entries, the first one points to a data block and the other nine entries have one level of indirection. Knowing that each block occupies 4 kilobytes, and the number of a block occupies 4 bytes and each block inode contains 16 inodes:

A)

a. Which is the maximum size of a file supported by this FS?

b. How much, the file of maximum size, does it occupy of *effective* space on the disc?
c. We consider a file containing 8,000,000 bytes. How much blocks (data and map) it is necessary to represent this file on disk? Justify briefly your answer.
d. Define the structure file descriptor (fdesc) correspondent to this FS.

B) Refer to the function **block_release()** written in class, then describe the steps to do for releasing a block without writing the code and calculate the number of I/O disk access (worst case) required to release a block.

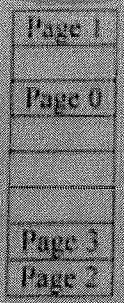
Part III — Memory Management — 25 Points

1. Given free memory partitions of 300k, 125k and 260k (in this order), describe, step by step, how each of the algorithms First-fit, Next Fit and best-fit allocate processes of 100k, 250k and 250k (in this order).

2. Let m the number of memory cases allocated to a program and W the set of referenced pages in a given time. Let S (m, W) the set of pages present in memory after the set of references W. Consider a program with a size of 5 pages. These pages designated as A, B, C, D, E are initially in secondary memory (disk).

   a. Construct the set S (m, ABCDABEABCDE) with m = 4 in the case of a FIFO replacement algorithm. How many page faults resulting from this series of references?
   b. The same question of a) for m=3.
   c. The same question of a) and b) in the case of LRU page replacement algorithm.

3. Consider a memory system with pagination that contains 8 frames (frames) of 16 bytes (each), and a process occupying 4 pages in this system. The placement of pages in memory is represented in the diagram below.



a. Indicate the contents of the page table of this process.
b. Explain how the logical address 35 bytes (decimal) is converted into a physical address (specify the decimal value of this physical address).

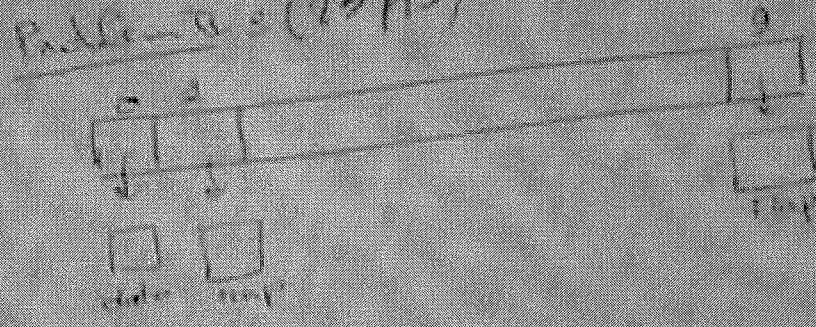
Ex1 - a (10 points)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

#define N 120

void main() {

int fd1[2],fd2[2],fd3[2],fd4[2],fd5[2],x,y,i,j;
int flag=0;
pipe(fd1);pipe(fd2);pipe(fd3);pipe(fd4);pipe(fd5);

for (i=2;i<=N;i++)
   write(fd1[1],&i,sizeof(int));

for(i=1;i<=5;i++)
{
  if(!fork()) {
    switch(i) {
      case 1:
        read(fd1[0],&x,sizeof(int));  //read 2
        printf("%d ",x);
        for(j=3;j<=N;j++) {
          read(fd1[0],&y,sizeof(int));
          if(y%x!=0) {write(fd2[1],&y,sizeof(int));}
        }//end for
      close(fd2[1]);close(fd1[0]);

      case 2:
        read(fd2[0],&x,sizeof(int)); //read 3
        printf("%d ",x);
        for(i=5;i<=N;i++) {
          if(read(fd2[0],&y,sizeof(int))>0){
               if(y%x!=0) {write(fd3[1],&y,sizeof(int));}}
           else {break;}

        }//end for
      close(fd3[1]);close(fd2[0]);

       case 3:
        read(fd3[0],&x,sizeof(int)); // read 5
        printf("%d ",x);
        for(i=7;i<=N;i++) {
          if(read(fd3[0],&y,sizeof(int))>0){
               if(y%x!=0) {write(fd4[1],&y,sizeof(int));}}
          else {break;}
        }//end for
      close(fd4[1]);close(fd3[0]);

     case 4:
        read(fd4[0],&x,sizeof(int));//read 7
        printf("%d ",x);
        for(i=11;i<=N;i++) {
          if(read(fd4[0],&y,sizeof(int))>0){
               if(y%x!=0) {write(fd5[1],&y,sizeof(int));}}
          else {break;}

        }//end for
      close(fd5[1]);close(fd4[0]);
      case 5:
         close(fd2[0]);close(fd2[1]);close(fd3[0]);close(fd3[1]);close(fd5
[1]);close(fd4[1]);close(fd4[0]);close(fd1[1]);close(fd1[0]);
        for(i=11;i<=N;i++) {
```

```
        // printf("hdgagdgsfgdisfd\n");
          if(read(fds[0],&y,sizeof(int))>0){
            printf("%d ",y);}
          else {break;}

        }//end for
      close(fds[1]);close(fds[0]);
    }// end switch
  }
  else {break;}
}//end main loop

}//end main function
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#define N 120

static int cnt=0;
void handler(int no){cnt++; printf("signal number %d arrived
\n",cnt);}
void main() {

int fd1[2],fd2[2],fd3[2],fd4[2],fd5[2],x,y,i,j,pid,p;
int flag=0; int fd[2];
pipe(fd1);pipe(fd2);pipe(fd3);pipe(fd4);pipe(fd5);pipe(fd);

for (i=2;i<=N;i++)
   write(fd1[1],&i,sizeof(int));

signal(SIGUSR1, handler);

for(i=1;i<=5;i++) {
if((pid=fork())) pause();
else {write(fd[1],&pid,sizeof(int));} }

read(fd[0],&p,sizeof(int));
kill(p,SIGUSR1);  //awake the first process
while(wait(NULL));

for(i=1;i<=5;i++)
{
  if(!pid) {
    switch(i) {
      case 1:
        read(fd1[0],&x,sizeof(int));   //read 2
        printf("%d ",x);
        for(j=3;j<=N;j++) {
          read(fd1[0],&y,sizeof(int));
          if(y%x!=0) {write(fd2[1],&y,sizeof(int));}
        }//end for
        close(fd2[1]);close(fd1[0]);
        read(fd[0],&p,sizeof(int));
        kill(p,SIGUSR1); //awake the second process
        exit(0);
      case 2:
        read(fd2[0],&x,sizeof(int)); //read 3
        printf("%d ",x);
        for(i=5;i<=N;i++) {
          if(read(fd2[0],&y,sizeof(int))>0){
              if(y%x!=0) {write(fd3[1],&y,sizeof(int));}}
          else {break;}

        }//end for
      close(fd3[1]);close(fd2[0]);
        read(fd[0],&p,sizeof(int));
        kill(p,SIGUSR1); //awake the third process
```

```
    exit(0);
  case 3:
    read(fd3[0],&x,sizeof(int)); // read 2
    printf("%d ",x);
    for(i=2;i<=N;i++) {
      if(read(fd3[0],&y,sizeof(int))>0){
        if(y%x!=0) {write(fd4[1],&y,sizeof(int));}
      else {break;}
    }//end for
    close(fd4[1]);close(fd3[0]);
    read(fd[0],&p,sizeof(int));
    kill(p,SIGUSR1); //awake the fourth process
    exit(0);


  case 4:
    read(fd4[0],&x,sizeof(int));//read 3
    printf("%d ",x);
    for(i=3;i<=N;i++) {
      if(read(fd4[0],&y,sizeof(int))>0){
        if(y%x!=0) {write(fd5[1],&y,sizeof(int));}
      else {break;}
    }//end for
    close(fd5[1]);close(fd4[0]);
    read(fd[0],&p,sizeof(int));
    kill(p,SIGUSR1); //awake the fifth process
    exit(0);

  case 5:
    close(fd2[0]);close(fd2[1]);close(fd3[0]);close
(fd3[1]);close(fd5[1]);close(fd4[1]);close(fd4[0]);close(fd
[1]);close(fd1[0]);
    for(i=1;i<=N;i++) {
      if(read(fd5[0],&y,sizeof(int))>0){
        printf("%d ",y);}
      else {break;}

    }//end for
    close(fd5[1]);close(fd4[0]);
    exit(0);
  }// end switch

  else {break;}
}//end main loop

}//end main function
```



B) <u>block_release</u> (7 pts)

if table of free cells not full

- => add the block nb to this table

~~else~~

- increment the pointer
- if not = write the block to disk
- set bfree = 0
- set the next_table pointer to [illegible]

=> Just one disk access

Problem

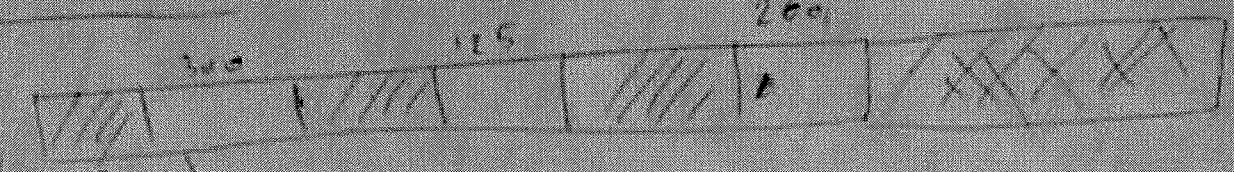
(3 pts)

d) struct fdesc{

```
    int topo[10];
    int map[1024];
    char buffer[4096];
    :
    :
}
```

Problem (1):

1)



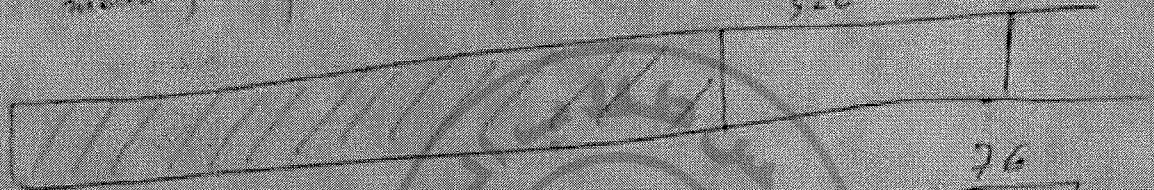
(6 pts)

First Fit

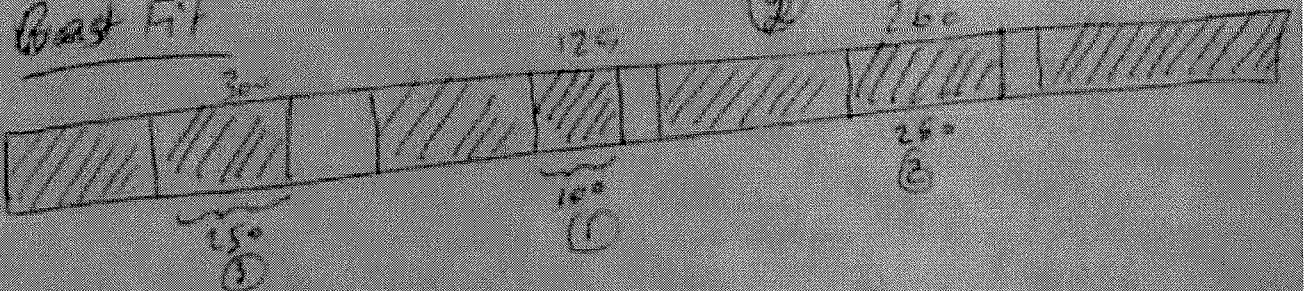
100, 250 and 200

⇒ we need compaction

necessary after compaction



Next Fit & Worst Fit same as First Fit

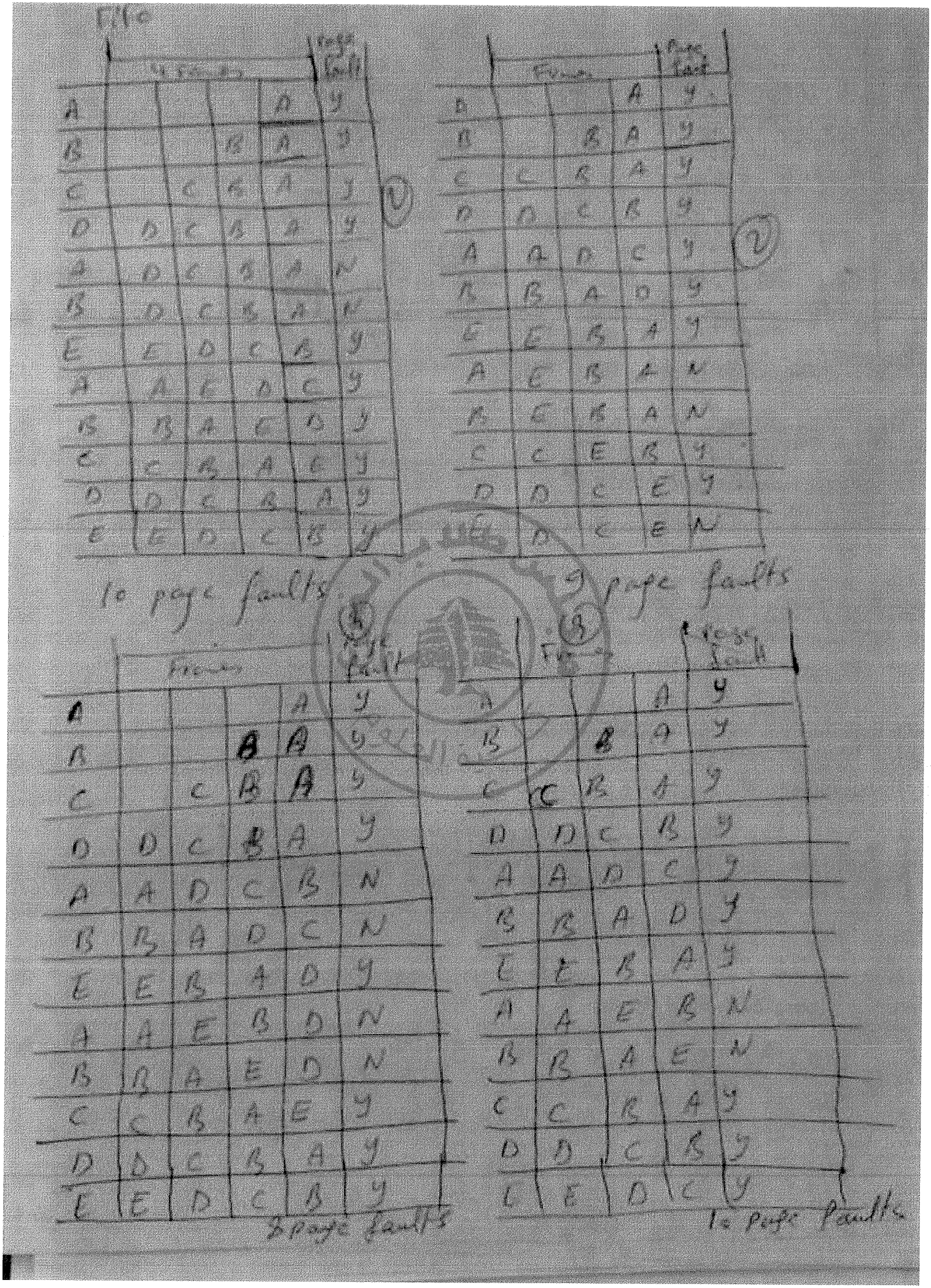


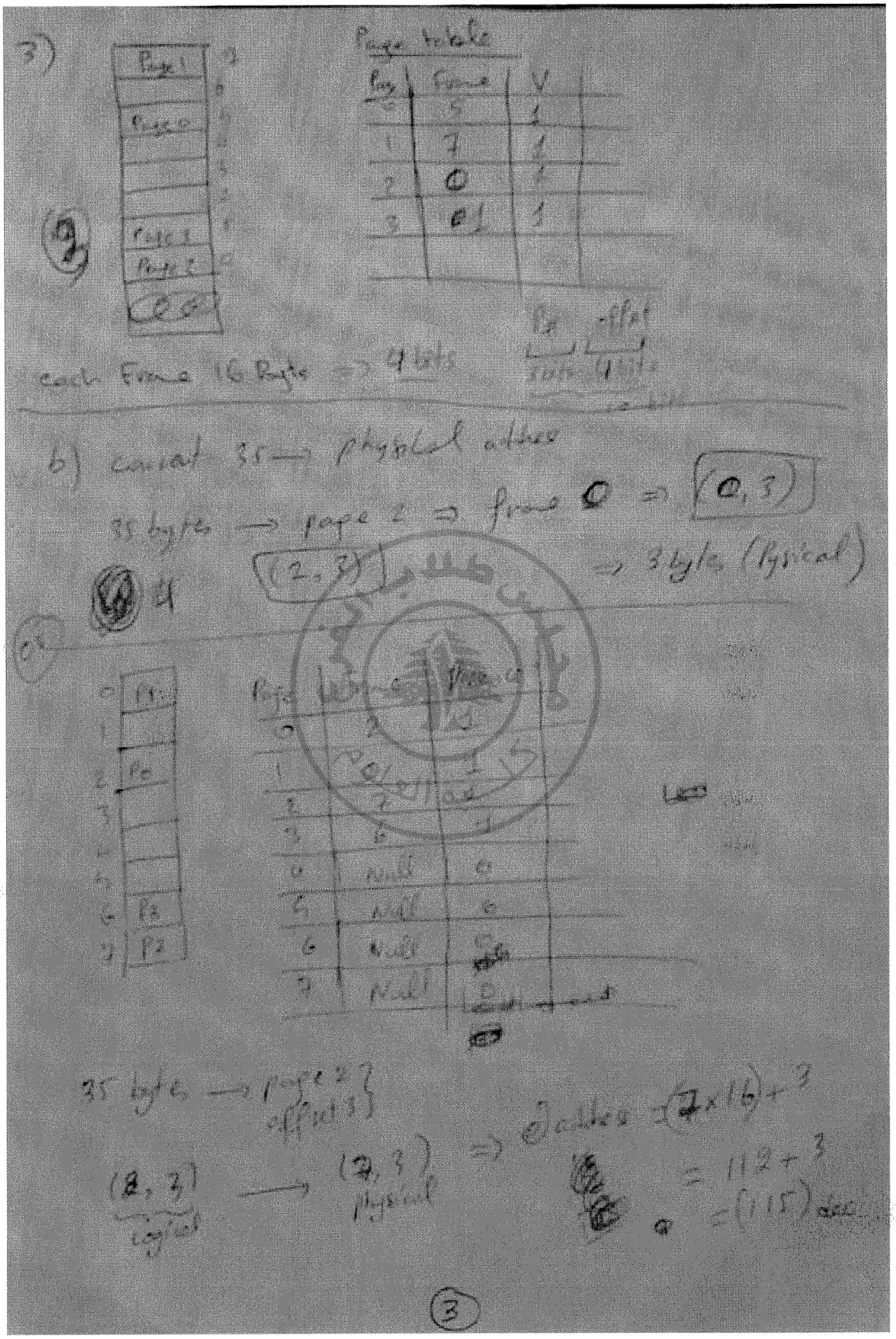
2) Process → 5 pages

memory → 4 frames

ref = A, B, C, D, A, B, E, A, B, C, D, E

(2)






جامعة طلاب النفس
كلية العلوم

Lebanese University Faculty of Sciences1	INFO 324 Operating System II	9 June 2016 Duration : 2 hours

Part I (27 pts)

A) **Answer the following questions with justification :**

a. Explain the difference between segmentation and pagination, the relative interest of each other (3)

b. What is virtual memory? give the reasons why it is advantageous to have a virtual memory mechanism on a computer (2)

c. What is a fragmentation and how to remedy it in contiguous memory allocation (2)

(8 pts)

B) **Consider the following C program:**

1. Draw the graph generated by calling **f(3).**

```
void f(int i){
    if(i<0) exit(0);
    if(i%2==0)   fork();
    f(i-1);
    }
```

(2)

2. **Draw the graph generated by the following program**

```
void main(){
    for(int i = 0 ; i <= 3; i ++)
        f(i);
}
```

(5)

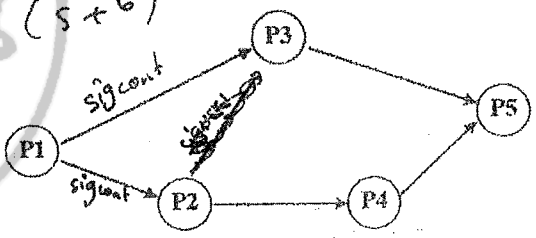
(8 pts)

3. Add (without any modification) to the function f the required statements such that no zombie processes are generated in part 2. (2)

C) A parent process creates 5 child processes P1, P2, .., P5. Then each child process displays its PID in respect to the order in the figure: For example the process P2 can't display its PID before P1. Also P5 can't before P3 and P4 and so on. (6)

1. Write the program using pipes of communication.
2. Rewrite the same program using signals.

(5+6)



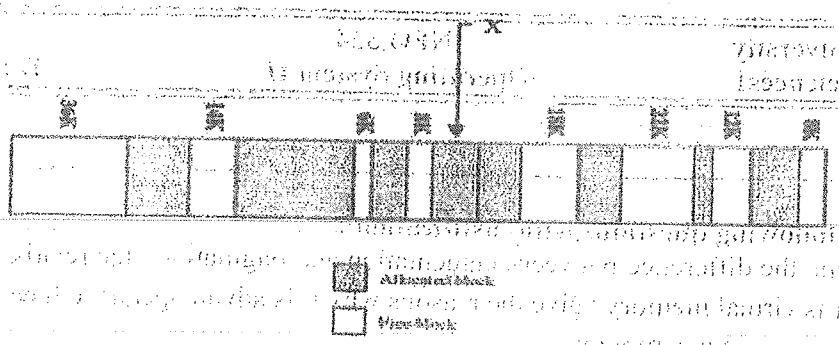
Part II (18 pts)

A) Consider a paginated memory with size of 48 KB and pages of size 12 KB. The following references are requested in memory: **A, B, C, D, E, B, E, F, D, A, B, C, G, F, C, B, A, B, C, F**
How many page faults are generated using the following replacement algorithms?

a. LRU (*Least Recently Used*)?
b. Second chance algorithm? (10)

B) We wish to allocate memory space for a process of 16KB. (4 pts)

1. Simulate the functioning of First Fit and Best Fit algorithms on the following mapping.
2. What gives the Next Fit algorithm if the former allocated block is indicated by the arrow X



C) We consider the following table of segments for a process P1 (4 pts)

Segment	Base	Limite
0	540	234
1	1254	128
2	54	328
3	2048	1024
4	976	200

1) Calculate the real addresses corresponding to the following virtual addresses (you may report addressing errors): (0:128), (1:100), (2:465), (3:888), (4:100), (4:344)
2) Is the virtual address (4,200) valid?

Note: the format of address is (segment#, offset)

Part III (25 pts)

A) Suppose that a disk drive has 10,000 cylinders, numbered 0 to 9999. The driver is currently serving a request at cylinder 1400. The queue of pending requests is, in the order received:
100, 1200, 900, 8000, 8100, 100, 8200, 1000, 4200
Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for the following scheduling algorithms? (For the algorithms in which the head is in constant motion, indicate the direction in which you assume it is moving initially.)
(a) FCFS (b) SSTF (c) Scan (d) C-look

B) We consider a file system that uses i-nodes like UNIX with few modifications as follows:

- 3 fields each of 8 bits containing information about the file
- 11 directs pointers to data blocks
- One pointer to simple indirection block where the last pointer of this block make another simple indirection

Given that each block has 1 Kb of size and occupies 2 bytes, (7 pts)
a) What is the maximum size of a file in this system?
b) Describe by figure the reading of the byte number 20992 of a file stored on disk.

C) Refer to the function written in class (i.e., **file_open** , ...): (10 pts)
a. Describe (without writing code) the steps needed to open a file.
b. How many **I/O disk** request is required to perform this task

Part I

a) **Difference between Segmentation and Pagination:** (3 pts)

→ Paging is used to get a large linear address space without having to buy more physical memory. Segmentation allows programs and data to be broken up into logically independent address spaces and to aid sharing and protection.

→ Paging does not distinguish and protect procedures and data separately.

→ Segmentation distinguishes and separately protects procedures and data.

→ Unlike segmentation, Paging does not facilitate sharing of procedures.

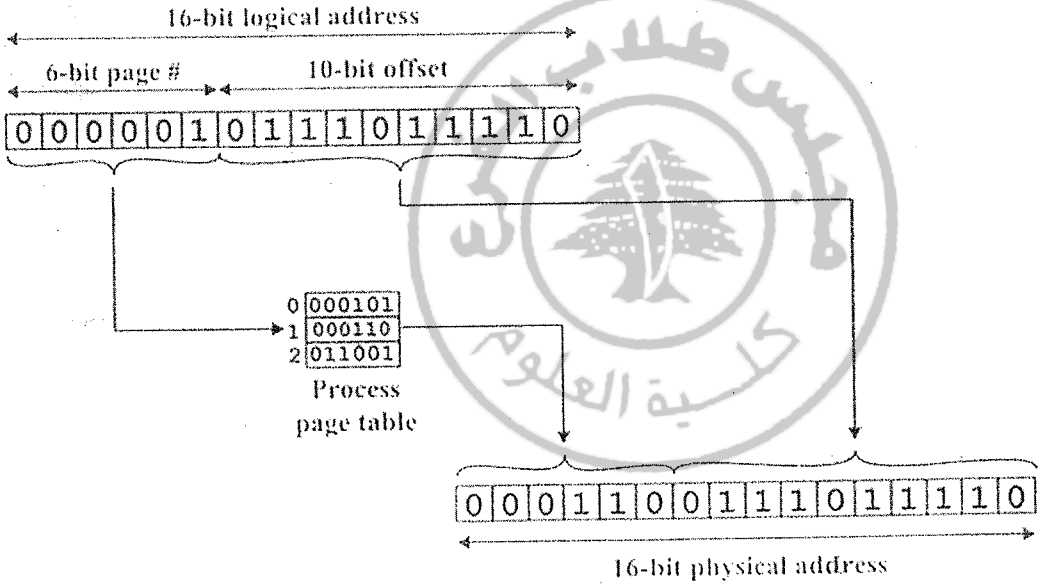
→ Paging is transparent to programmers (system handles it automatically).

→ Segmentation requires programmer to be aware of memory limits as programmer tries to allocate memory to functions and variables or tries to access read only memory violation, which results in segmentation fault.

→ Mapping from logical to physical address is different for paging and segmentation.
Here's an illustration based on 16 bit-address space:

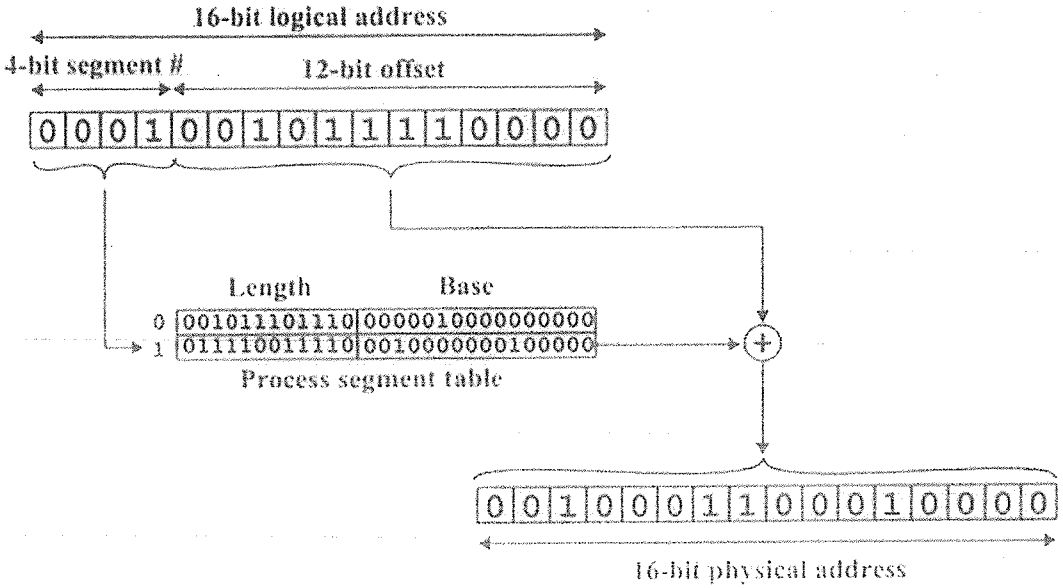
For paging:

The 6-bit page value is used to select a proper entry in process page table. the 6-bit process entry occupying the six most significant bit and the 10-bit offset occupying the 10 least significant bit forms a 16-bit physical address.



For segmentation:

The 4-bit segment of a logical address selects the proper entry in the process segment table. The base value is added to the 12 bit offset value to get the 16 bit physical address.



b) **What is virtual memory?** (2 pts)

In computing, virtual memory is a memory management technique that is implemented using both hardware and software. It maps memory addresses used by a program, called **virtual addresses**, into physical addresses in computer memory. Main storage as seen by a process or task appears as a **contiguous address space or collection of contiguous segments**. The operating system manages virtual address spaces and the assignment of real memory to virtual memory. Address translation hardware in the CPU, often referred to as a memory management unit or MMU, automatically translates virtual addresses to physical addresses. Software within the operating system may extend these capabilities to provide a virtual address space that can exceed the capacity of real memory and thus reference more memory than is physically present in the computer.

The primary benefits of virtual memory include freeing applications from having to manage a shared memory space, increased security due to memory isolation, and being able to conceptually use more memory than might be physically available, using the technique of paging.

c) **What is a fragmentation?** (3 pts)

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types

S.N.	Fragmentation	Description
1	**External fragmentation**	Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous so it cannot be used.
2	**Internal fragmentation**	Memory block assigned to process is bigger. Some portion of memory is left unused as it cannot be used by another process.

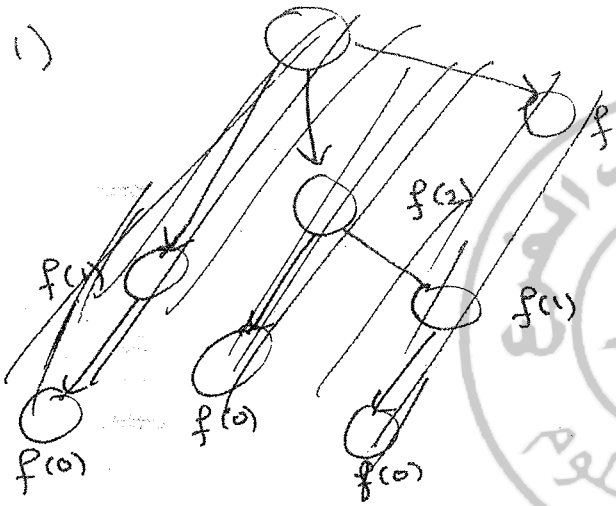
External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

Part I (27 pts: A → 8 (3+2+3), B → 8 (2+5+1), C → 11 (5+6))

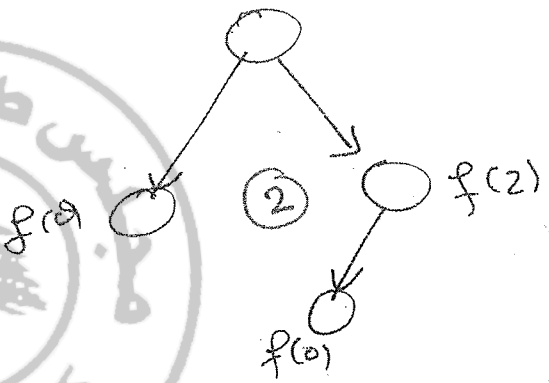
B)
```
void f(int i)
{
    if (i<0) exit(0);
    if (i%2 == 0) fork();
    f(i-1);
}
```

f(3) → f(2) $\xrightarrow{fork()}$ f(1) $\xrightarrow{fork()}$ f(0) $\xrightarrow{fork()}$ f(-1) ×

1)



~~8 processes~~ 4 processes



2)
```
void main()
{
    for (int i=0; i<=3; i++)
        f(i);
}
```

f(0) $\xrightarrow{fork()}$ f(-1) ×
f(1) → f(0)
f(2)
f(3)

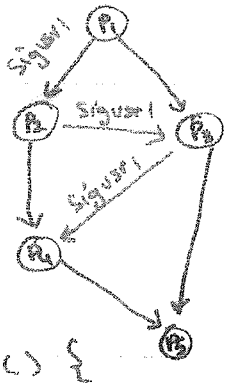
⇒ ~~8 processes~~

i=0 f(0) $\xrightarrow{fork()}$ f(-1) ×
i=1 f(1) → f(0) $\xrightarrow{fork^2()}$ f(-1) ×
i=2 f(2) $\xrightarrow{fork^3()}$ f(1) → f(0) $\xrightarrow{fork^4}$ f(-1) ×
i=3 f(3) → f(2) $\xrightarrow{fork^5}$ f(1) → f(0) $\xrightarrow{fork^6}$ f(-1) ×

we have 6 calls to fork, so 2^6 processes = 64 (4)

3)
```
void f(int i)
{
  if (i<0) exit(0)
  if (i%2==0)
    {
      fork();
      wait(NULL);
    }
  f(i-1);
}
```
(1)

c)
1)
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
void main() {
int P12[2], P13[2], P24[2],
    P35[2];
int x;
pipe(P12); pipe(P13); ----
for (int i=1; i<5; i++)
{
  if (!fork()) break;
  if (i==1){ // P1
    printf("process P1 with pid
        %d\n", getpid());
    write(P12[1], &x, sizeof(int));
    write(P13[1], &x, sizeof(int));
  }
  else if (i==2) // P2
  {
    read(P12[0], &x, sizeof(int));
    printf("Process P2 ----");
    write(P24[1], &x, sizeof(int));
  }
  else if (i==3) // P3
```
5 pts

2) signals
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>
int pid, int fd[2]; int next;
static counter=0;
void handler(int nsig)
{ ctr++
printf("I'm the process number %d,
    with pid %d\n", ctr, getpid());
if(read(fd[0], &next, sizeof(int))){
  kill(next, sigusr1); exit();
}
else
{
exit;
}
} //end handler
```


6 pts
```
void main() {
  pipe(fd);
  signal(sigusr1, handler);
for (int i=1; i<=4; i++)
{
  if(!(pid=fork()))
    {
      close(fd[1]);
      pause();
    }
  write(fd[1], &pid, sizeof(int));
} //end for
printf("I'm the process P1 with
    pid=%d\n", getpid());
close(fd[1]);
while(wait(NULL));
} //end main
read(fd[0], &next, sizeof(int));
kill(next, sigusr1);
counter++;
```

Part II

A) Memory :- size 48 KB
- page 12 KB
⇒ 4 Frames

a) LRU (5) (3+2)

Page ref	Faults					
A	yes	A				
B	yes	A	B			
C	yes	A	B	C		
D	yes	A	B	C	D	
E	yes	E	B	C	D	
B	No	E	B	C	D	
E	No	E	B	C	D	
F	yes	E	B	F	D	
D	No	E	B	F	D	
A	yes	E	A	F	D	
B	yes	B	A	F	D	
C	yes	B	A	C	D	
G	yes	B	A	C	G	
F	yes	B	F	C	G	
C	No	B	F	C	G	
B	No	B	F	C	G	
A	yes	B	F	C	A	
B	No	B	F	C	A	
C	No	B	F	C	A	
E	No	B	F	C	A	

12 page faults

b) Second chance (5) (3+2)

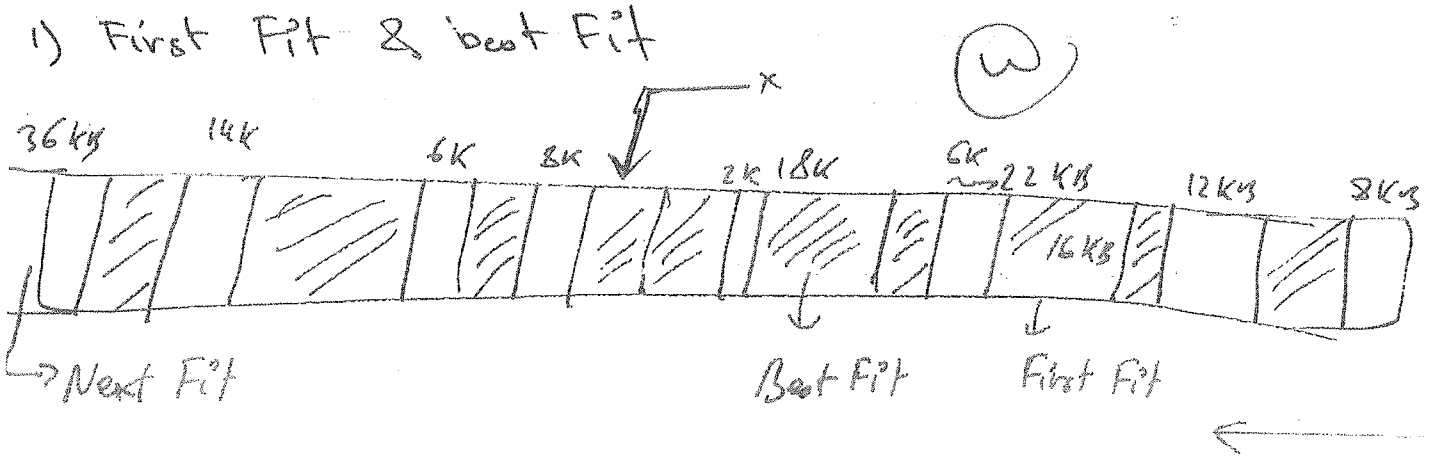
Page ref	Faults	4 Page Frames: Page contents			
A	yes	A*			
B	yes	B*	A*		
C	yes	C*	B*	A*	
D	yes	D*	C*	B*	A*
E	yes	E*	D	C	B
B	No	E*	D	C	B*
E	No	E*	D	C	B*
F	yes	F*	E*	D	B
D	No	D*	E*	D*	B
A	yes	A*	D* F*	E*	E* D*
B	yes	B*	A	F	E
C	yes	C*	B*	A	F
G	yes	G*	C*	B*	A
F	yes	F*	G*	C*	B*
C	No	F*	G*	C*	B*
B	No	F*	G*	C*	B*
A	yes	A*	F	C	B
B	No yes	B* A*	A F	F C	B*
C	No yes	A* C*	F B*	C*	B*
F	No	F*	C*	B*	A*

12 page Faults

12 → see next page

B) process = 16 KB

1) First Fit & best Fit



2) 1) (0; 128) → 540 + 128 = 668
① (1; 100) → 1254 + 100 = 1354
(2; 465) → error
① (3; 888) → 2048 + 888 = 2936
(4; 100) → 976 + 100 = 1076
① (4; 344) → error

2) (4, 200) NO ①

Part III Disk : 10 000 cylinders

A) - currently at cylinder 1400

queue : 100, 1200, 900, 8000, 8100, 100, 8200, 1000, 4200

a) FCFS → 36400
b) SSTF → 9400
c) Scan → 9600 or 18498
d) C-look → 13400 or 16000

5300
5300
8100
x
5300

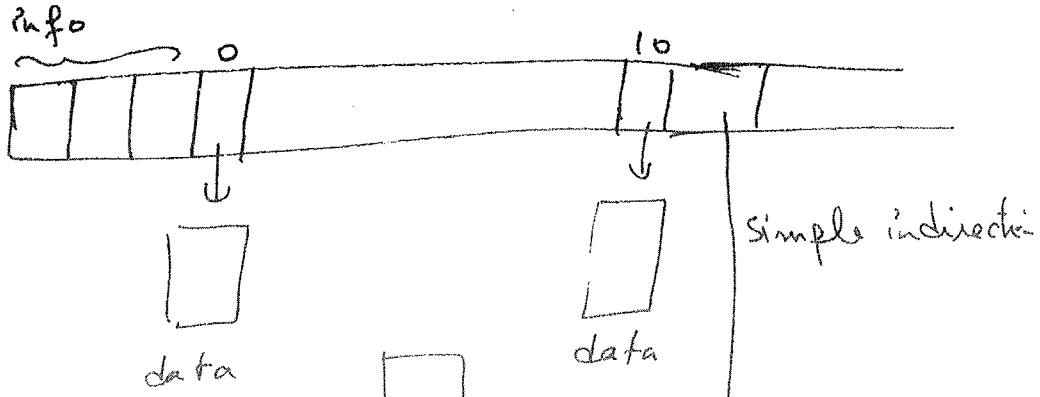
Second chance Alg

Page	fault	4 pages frames			
A	Yes	A			
B	Yes	B	A		
C	Yes	C	B	A	
D	Yes	D	C	B	A
E	Yes	E	D	C	B
B	No	E	D	C	B*
E	No	E*	D	C	B*
F	Yes	F	E*	D	B
D	No	F	E*	D*	B
A	Yes	A	F	E*	D*
B	Yes	B	A	E	D
C	Yes	C	B	A	E
G	Yes	G	C	B	A
F	Yes	F	G	C	B
C	No	F	G	C*	B
B	No	F	G	C*	B*
A	Yes	A	F	C	B
B	No	A	F	C	B*
C	No	A	F	C*	B*
F	No	A	F*	C*	B*

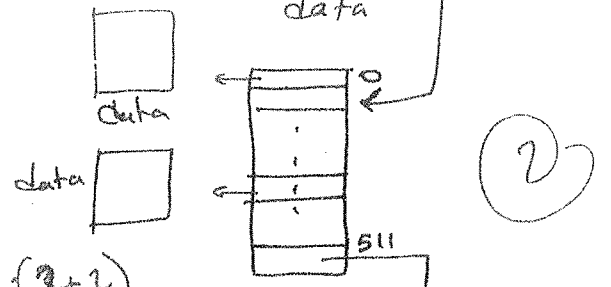
12 page faults

Part III

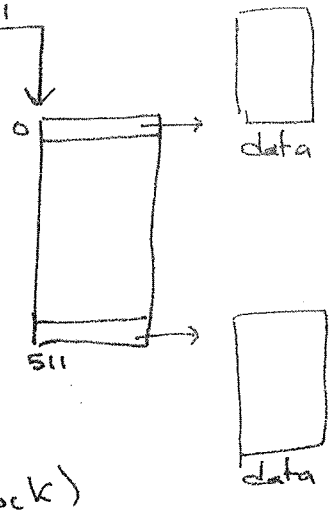
B) Inode :



(7) (4+1)



(2)



Block size = 1 KB

Block # = 2 bytes

a) Maximum size of a file ? (2+2)

b) reading of byte 20992 (2)

Answers :

a) each block can contains $\frac{2^{10}}{2} = 2^9 = 512$ number of blocks

Max size : 11 direct block + 512 (data block)

+ 512 (data block)

= 11 + 511 + 512 = 1034 KB (2)

11264

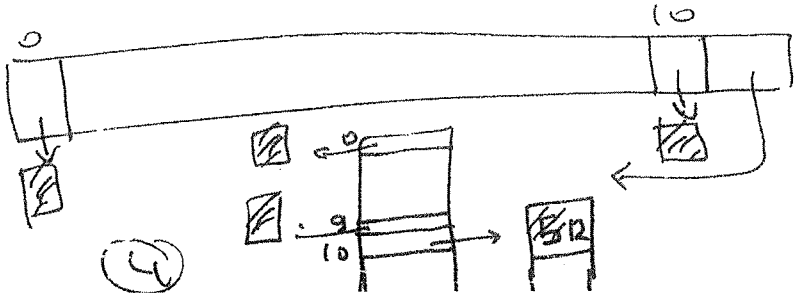
b) $20992 - (11 \times 1024) = 20992 - 11264 = 9728$

$9728 = (9 \underbrace{\times 1024}_{9216}) + 512$

So the file covers the 11 direct data block

- the first 0 ⟶ 9 (10) from the simple indirecto
- In the block 10 (simple indirection), we have

offset 512 (3)



(4)

c) file_open (10 = 6+4)

a) steps

1- for opening the file we need to find its associated inode #

2- search the global folder for the (external name)

2-1 research loop in the entry 0 of (fdesc)

2-2 load the data blocks for the global folder into memory (buffer)

2-3 each 16 bytes corresponds to (namefile, inode)

3- when the file is found, get the inode #

4- search for a free entry in the table of descriptor (fdesc)

5- load in memory the block of inodes that contains the inode #

6- initialize the fdesc entries

~~-inopen~~

b) I/O disks?

* file_set_position(0,0) → 1 disk read (I/O)
* in worst case we need to read all blocks of the global directory (entry 0) : Suppose x I/O
* disk_read (binode, 2 + (fold.inode_nb/16)); → 1 I/O

So we need : x+2 I/O

Lebanese University	INFO 324	17 June 2015
Faculty of Sciences1	Operating System II	Duration : 2 hours

Part1 (20 minutes - 15 points):

A) Answer the following questions with justification :

a) What is the advantage of contiguous file allocation policy vs linked allocation policy? (2)

b) Describe in detail the problem of fragmentation and how to manage it? (4)

B) Given the following program

```
#include <stdio.h>
#include <unistd.h>
int main(){
int i;
for (i=0;i<3;i++)
        if (fork()) i++;
while(1);
return i;}
```

(5)

How many processes does this program generate? Draw the generated graph.

C) How many processes are generated by the following code :

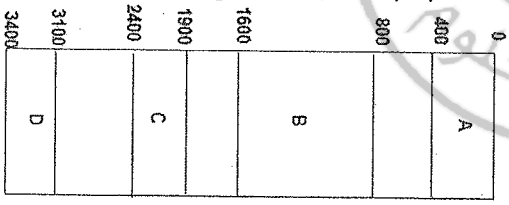
```
int main () {
  while (fork())
    execv (path, com);
return 0;
 }
```

(4)

Where **path** is the path to the executable and **com** is the executable process.

Part 2: Memory Management (60 minutes - 30 points)

A) Consider a contiguous memory system with memory allocated as shown below.



Suppose the following actions occur:

- Process E starts and requests 300 memory units. (1)
- Process A requests 400 more memory units. (1) + ½
- Process B exits. (1/2)
- Process F starts and requests 800 memory units. (1)
- Process C exits. (1/2)
- Process G starts and requests 900 memory units. (1) + ½

(12)

(a) Describe the contents of memory after each action using the first-fit algorithm. (6)

(b) Describe the contents of memory after each action using the best-fit algorithm. (5)

(c) For this example, which algorithm is best? (1)

P.S: you can compact the memory in case of need such that moving used blocks or free blocks

B) Consider a memory paginated system with page size 256 bytes. In this system each process is authorized max 4 frames in main memory. The page table of a process P1 is given in the following table:

page	0	1	2	3	4	5	6	7
frame	011	001	000	010	100	111	101	110
presence	1	0	1	0	0	0	1	0

1) What is the size of the virtual address space of the process P1
2) What is the size of the physical memory?
3) Convert the following virtual addresses to physical (signal the error if any): 546, 2072.
4) What will happen if P1 generates the virtual address 770?

C) Consider a system with memory capacity 2GB, page size 4KB and addressing on 32 bits. Given that each table entry contains: a reference to a frame + 1 bit presence/absence.

a. What is the size of the page table (justify your answer).
b. How many pages are needed to load the page table in memory?

D) Consider a process with virtual address space of 600 Bytes. the set of virtual addresses referenced is: 34; 123; 145; 510; 456; 345; 412; 10; 14; 12; 234; 336; 412.

a. Give the list of referenced pages given the size of the page is 100 Bytes.
b. Determine the number of page faults for the LRU algorithm. The memory is initially empty and contains 3 frames

Part 4: File System (40 minutes – 25 points)

A) Consider a file currently consisting of 100 blocks of data. Assume that the file control block is loaded in memory and there is no cache disk. The size of the block is 4KB. Calculate the number of disk I/O operations required for **contiguous** and **linked** allocation strategies to make the following changes to the file. In the contiguous case, you may assume there is no space to grow in the end. Also assume that the new information to be added to the file is not stored in memory.

a. **Add 2 blocks at the beginning**
b. **Add 2 blocks at the end**
c. **Remove the middle block**

B) Refer to the functions written in class (i.e., **create_inode** , ….):

a. Describe (without writing code) the steps needed to create an inode.
b. How many **I/O disk** request is required to perform this task

C) Consider a disk of size 20GB in which the system installed is 16-bit DOS (FAT). The disk is divided into a set of blocks of fixed size (128KB). This disk contains 520 files: 200 files of size 16K, 200 files of size 256KB and 120 files of size 1Mb.

a. Calculate in MB the disk space.
b. Calculate the number of blocks on disk.
c. How many blocks do occupy each of these three categories of files?
d. Calculate in Kb the size of the FAT table. Justify your answer

INFO SEC4

Part I

A) a) advantage of contiguous file allocation Vs linked allocation policy

- quick and easy calculation of blak holding data
  * just offset from start of file
- for sequential access, no seeks required
- the read performance is excellent bcz the entire file can be read from the disk in a single operation only one seek is needed (the first block).
- No problem of reliability whereas this is a big problem in linked allocation
- the amount of storage is a power of 2

(2)

b) Problem of fragmentation

- During its lifespan, a process can request and free many chunks of memory.
  * when a process is started, the free memory areas are long and contiguous.
  * Over time and with use, the long contiguous regions become fragmented into smaller and smaller contiguous areas. Eventually, it may become impossible for the program to obtain large contiguous chunks of memory.

(3)

* there exist two types of fragmentation:
  - internal: Due to the rules governing memory allocation (such as paging), more computer memory is sometimes allocated than is needed. For example, in asyste~~m~~ with page size 512Kbytes, a file with size 400 KB is allocated one page, and thus there is a loss of 112 KB => this waste is called internal frag—

(1)

▷ External : arises when free memory is separated into small blocks and is interspersed by allocated mem

- The term external refers to the fact that the unusable storage is outside the allocated regions
- For example in a contiguous allocation strategy with dynamic partitions, consider a situation where in the memory allocator allocates 3 contiguous regions of memory for 3 different processes, and then free the middle region due to a swap or exit.
If a new demanded region of mem is larger than the free space ⇒ this free space is called external fragmentation

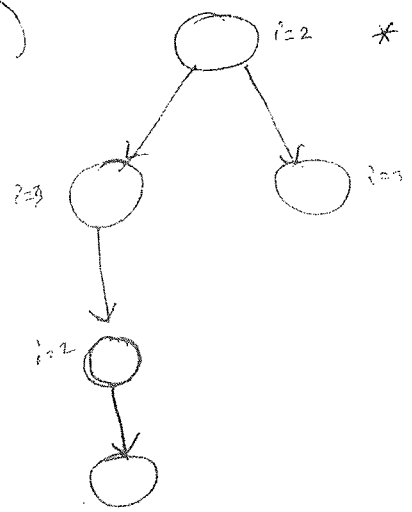
→ internal fragmentation : view by the process
→ external fragmentation : view by the system

Remediation

* there is no complete solution for internal fragmentation
- the internal fragmentation is always, at max with the size of a mem page
* the external fragmentation can be avoided by decomposing the memory into fixed size blocks such as paging system.

Part I (continued)

B)



* There is 5 processes including the main process

(5)

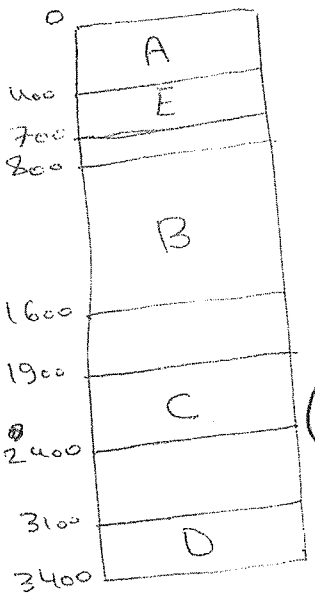
C) just one process is generated in this code by the parent (main).

the main process change the address space and executes another process and the child exits. (4)

Part II (Memory management)

A)

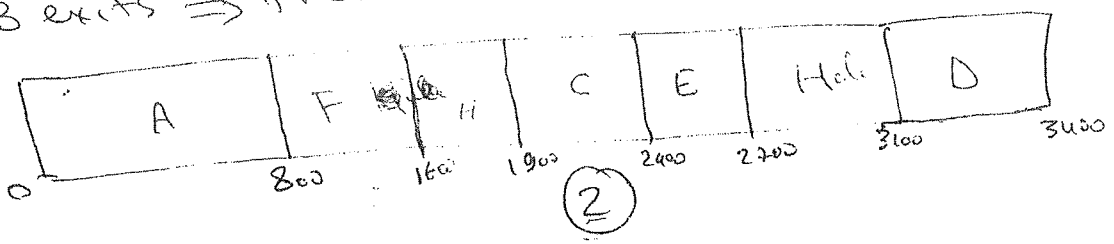
a) First Fit algorithm:



① * process E starts and requests 300 memory units
⇒ allocation in the first free zone 1100-2700,

* process A requests 1100 more memory units A
~~⇒ no free spaces to give to process~~
~~⇒ compact the memory~~
⇒ cannot fit because the entire process to allocated in a single contiguous chunck of memory in a contiguous memory system
⇒ Move B to 1100 - 1900
Move E to 2400 - 2700
* Give A additional addresses 1100 - 800
* There is a hole between 800 - 1900 (12)

* B exits ⇒



(2)

* process F starts and requests 800 memory units
  ⇒ F is allocated in 800 – 1600 (1)
* C exists ⇒ there is a hole between 1600 – 2400 (1/2)
* G starts and requests 900 memory units
  ⇒ no hole that is big enough
  ⇒ compact memory : move E to 2800 – 3100 (1) + (1/2)
  ⇒ G is allocated in 1600 – 2500

b) Best-Fit algorithm

E requests 300 ⇒ E is allocated in 1600 – 1900 (1)
A requests 400 more ⇒ this 400 is allocated in 400 – 800 (1)
B exits ⇒ there is a hole between 800 – 1600 (1/2)
F requests 800 ⇒ F is allocated in 800 – 1600 (1)
C exits ⇒ there is a hole between 1900 – 3100 (1/2)
G requests 900 ⇒ G is allocated in 1900 – 2800 (1)

c) Worst-Fit Algorithm (not required)

E requests 300 ⇒ E is allocated in 2400 – 2700
A requests 400 more ⇒ this additional 400 is allocated in 400 – 800
B exits ⇒ there is a hole between 800 – 1900
F requests 800 ⇒ F is allocated in 800 – 1600
C exits ⇒ there is a hole between 1600 – 2400
G requests 900 : no hole big enough ⇒ need to compact
⇒ move E to 2800 – 3100, give 1600 – 2500 to G.

For this example, Best-Fit is the best (1)

B) - page size = 256 Kbytes
- each process is authorized max 4 frames

Page	0	1	2	3	4	5	6	7
frame	011	001	000	010	100	111	101	110
Presence	1	0	1	0	0	0	1	0

1) Size of virtual address space of P1

$8 \times 256 = 2048$ bytes (2)

2) Size of physical memory

The frames are coded in 3 bits $\Rightarrow 2^3 = 8$ frames

$\Rightarrow$ physical memory size = nb(frames) x size(frame) (2)
$= 8 \times 256 = 2048$ bytes
$= 2$ KB

3) Convert to physical address

The conversion from virtual address to physical is realized as the following:
a) calculation of page # and offset
b) search in page table the frame #
c) physical address = frame address + offset

Then:

$546 = 2 * 256 + 34$

$\Rightarrow$ page 2 & offset 34
$\Rightarrow$ frame 0 (page table)
$\Rightarrow$ physical address is 34 (1)

2072 is out of the virtual address space of P1
$\Rightarrow$ error (1)

4) $770 = 3*256 + 2 \Rightarrow$ page 3. But this page is not loaded in memory $\Rightarrow$ page fault (1)

C) Memory size = 2 GB

page size = 4 KB

addressing 32 bits

each PTE : reference to frame + 1 bit p/A

(a) size of page table

- addressing on 32 bits $\Rightarrow 2^{32}$ virtual address space
- page size = 4 KB $= 2^{12}$

$\Rightarrow$ [20 bits: page #] [12 bits: offset]

$\Rightarrow$ page table contains 2^{20} entries

Or the number of frames is : $\frac{\text{Memory size}}{\text{page size}} = \frac{2\,GB}{4\,KB} = \frac{2^{31}}{2^{12}}$

(3) $= 2^{19}$

$\Rightarrow$ we need 19 bits for the frame #

$\Rightarrow$ each PTE size is 19 + 1 = 20 bits

$\Rightarrow$ the size of page tables is $2^{20} \times 20$

$= \cancel{2^{22} \times 5}$ 20 MB

$= \cancel{2559\ KB}$

(b) How many pages ? / 20 MB

pages $= \frac{\cancel{2559}\ KB}{4\ KB} = \cancel{640}\ 5120$ (2) pages

D) process : virtual address space 600 Bytes page size = 100 Bytes

a) 0, 1, 1, 5, 4, 3, 4, 0, 0, 0, 2, 3, 4 (2)

9 page faults

page / frame			*				*		*	*				
f0	0	0	0	0	4	4	4	4	4	4	4	3	3	
f1		1	1	1	1	3	3	3	3	3	2	2	2	(4)
f2				5	5	5	5	0	0	0	0	0	4	

Part 4 : File system

- File : 100 blocks of data
- File control block is loaded in memory
- size of block = 4KB

Number of disk I/O ?

	contiguous	linked
Add 2 blocks at begin	2R + 2W ②	2R + 2W ②
Add 2 blocks at end	102 R + 102 W ②	3R + 3W ②
Remove the middle block	50R + 50W ②	50 R + 1W ②

i) add 2 blocks at the begining

- Contiguous : 2 reads block for new information
  + 2 write blocks "
- Linked : 2 reads for new inf
  - update pointers in memory
  - 2 writes for new inf

ii) add 2 blocks at the end

- contiguous : there is no room to add at the end
  ⇒ shifting all the blocks two places at the end
  ⇒ 100R + 100W
- read 2 new inf blocks } 2R + 2W
- write 2 new inf blocks
  ⇒ 102R + 102W

- Linked :
  - Read in the last block → 1R
  - Read in the 2 new blocks → 2R
  - update pointers in memory → 0R, 0R
  - write the three blocks → 3W

  } 3R + 3W

iii) Remove the middle block

- contiguous:

to delete position 50, read all blocks after #50 and write them back one place ~~closer to the front~~

⇒ 50 R + 50 W

- ~~read 2 new inf blocks and write it back~~

~~⇒ 2R + 2W~~

⇒ 50R + 50W

- linked:

read and follow links to position 50

⇒ 50 R → update in memory block 49 (Next) → 51

then write block 49 to link to former block 51 ⇒ 1 W

⇒ 50R + 1W

B) create_inode

a) the steps to create inode are:

- search for free inode:
  1 - in cache
  2 - not found, in disk
- bring the block that contains the inode to memory
- initialize the inode
- write back the block containing the inode to disk.

b) # of disk I/O?

* if there is free inode in cache ⇒ 1R + 1W
* if no free inode in cache ⇒ search on disk

⇒ worst case read all blocks of inodes

⇒ size R + 1 W

c) disk: _ size = 20 GB

- 16 bit DOS
- block = 128 KB
- 520 Files → 200 files (16K)
  → 200 files (256 KB)
  → 120 files (1 MB)

a) disk space = 20×2^{10} MB = 20480 MB (1)

b) Number of blocks : $\frac{\text{disk size}}{\text{block size}} = \frac{20 \times 2^{10} \times 2^{10}}{128 \text{ KB}}$ KB = ~~$\frac{20 \times 2^{13}}{2^7 \times 2^{10}}$~~

= 163840 blocks

$= 20 \times 2^{13}$ (1)

$= 5 \times 2^{15}$

c) - 200×16 KB = 200×2^4 KB

= 3200 KB

other: in fact each file needs one block so 200 blocks but I consider the 2 solutions true

$\Rightarrow 25 \times 128$ KB = 25 blocks (not real solution but I'll accept)

- 200×256 KB $\Rightarrow$ 400 blocks (2)

- 120×1 MB $\Rightarrow 120 \times 8$ = 960 block

d) size of the FAT table ?

size (FAT) = Number of entries * size (entry)

or FAT on 16 bits $\Rightarrow$ max 2^{16} clusters (3)

= 65535 clusters

or each cluster must contain power of 2 sectors

$\Rightarrow$ each cluster must contains 4 blocks

$\Rightarrow$ the entry in FAT is 18 bits (clust # | block #)

$\Rightarrow$ Size (FAT) $= 2^{16} \times 18 =$ **143 KB**

Lebanese University	INFO 324- OS II	Sept 2015
Faculty of Sciences1	Second Session	Duration : 2 hours

Part1: Process Management (30 points):

A) **Answer the following questions with justification :**
   a. Explain in detail how the use of two-level or more page tables will reduce the space required for the one-level page table.
   b. Explain the difference between logical and physical addresses

B) After explaining what it means:
   a. A process is in the zombie state.
   b. A process is orphaned.
   Write a program that creates a process that fall within the zombie state before becoming orphan.

C) Write a C program where a parent creates N child processes. Then, the processes behave in the following manner:
Every T second, the father write in a pipe a character 'G' and sends to all "alive" child the "SIGUSR1" signal for unlock them. The child awakened by the signal, start the race to read the character already written in the tube by the father. The winning child process must terminate after displaying a number indicating the termination order.
P.S: Remember that the father sent the signal "SIGUSR1" only for living child (who are not already terminated.)

Part 2: Memory Management (15 points)

Consider a system with paginated main memory. The memory is composed of 4 frames (frames) each has a size of 64 bytes. At a given moment, the memory is empty, then two processes P1 (4 pages) and P2 (2 pages) are launched in the system. The processor sends requests submissions in the following order in the format [hexadecimal logical address, Process]:

[3F, P1]
[4A, P1]
[1D, P1]
[00, P2]
[CA, P1]
[87, P1]
[39, P2]
[2B, P1]
[00, P1]
[11, P2]

1. What is in bytes the size of the main memory?
2. What is the size of the virtual address space?
3. Indicate by figures the evolution of the memory and the number of page faults using the following page replacement algorithms :
   a) FIFO
   b) LRU
   c) Second chance.

Part 3: File System (25 points)

A) Consider a file system with linked list and indexed allocation strategy as illustrated in the figure. The size of block is 8 KB.

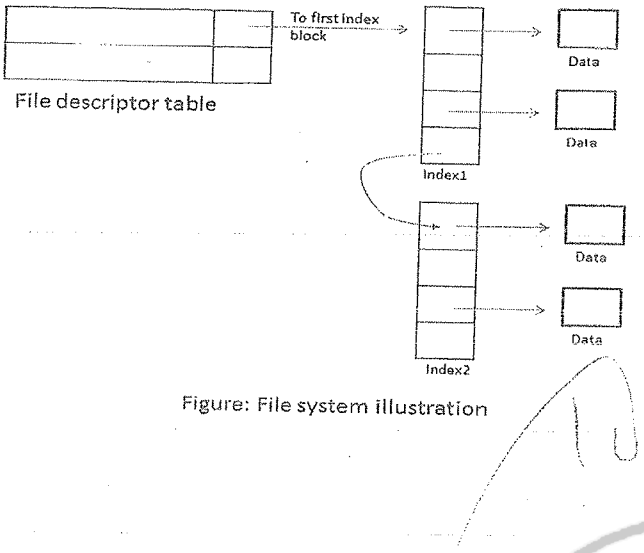


Figure: File system illustration

The disk blocks are numerated from 0 to n-1. Each opened file has an entry in the file descriptor table which is loaded in memory. This entry contains all attributes of the file and a pointer to an index block. The index block contains k pointers to other blocks. These pointers except the last points to data block. The last pointer points to another index block and so on as shown in figure. The pointer to block occupies 32 bits.

Question: What is the maximum size (in number of blocks) of a file in this system? Indicate the number of data blocks and index blocks

B) Given a FS where the topo table contains 10 entries, each corresponding to a single level indexing (each block points to a map). Given that each block occupies 2 kilobytes, the number of a block occupies 4 bytes and each block contains 16 inode:
Write the function ***delete_block (int lbd, int f)*** that delete the physical data block corresponding to the logical data block lbd of the file with descriptor f.
Note that if this block is not at the end of file, you must shift all the other data blocks that follow.

P.S: the use of the functions seen in the course is permitted if needed.

Part 1 : Process management

1) a) using 2 or more level page table

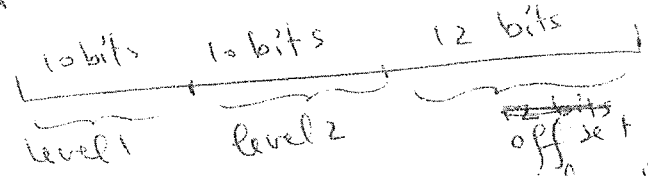
since each page in the virtual address space needs an entry in the page table, we may have a huge number of entries in the page table where actually a few of them are needed by the process. for example in a virtual address space of 32 bits with page of size 4KB $= 2^{12}$

We need 2^{20} entries. Each entry occupy 4 bytes

$\Rightarrow$ 4MB the page table are reserved for a process which it actually needs a few entries. For a 64 bits virtual address space we need $2^{52} \times 4$ bytes for the page table for each process

→ using multi-level page table can reduce the overhead as follows:

for example in two level page table the virtual address space is divided into 3 parts: (32 bits)

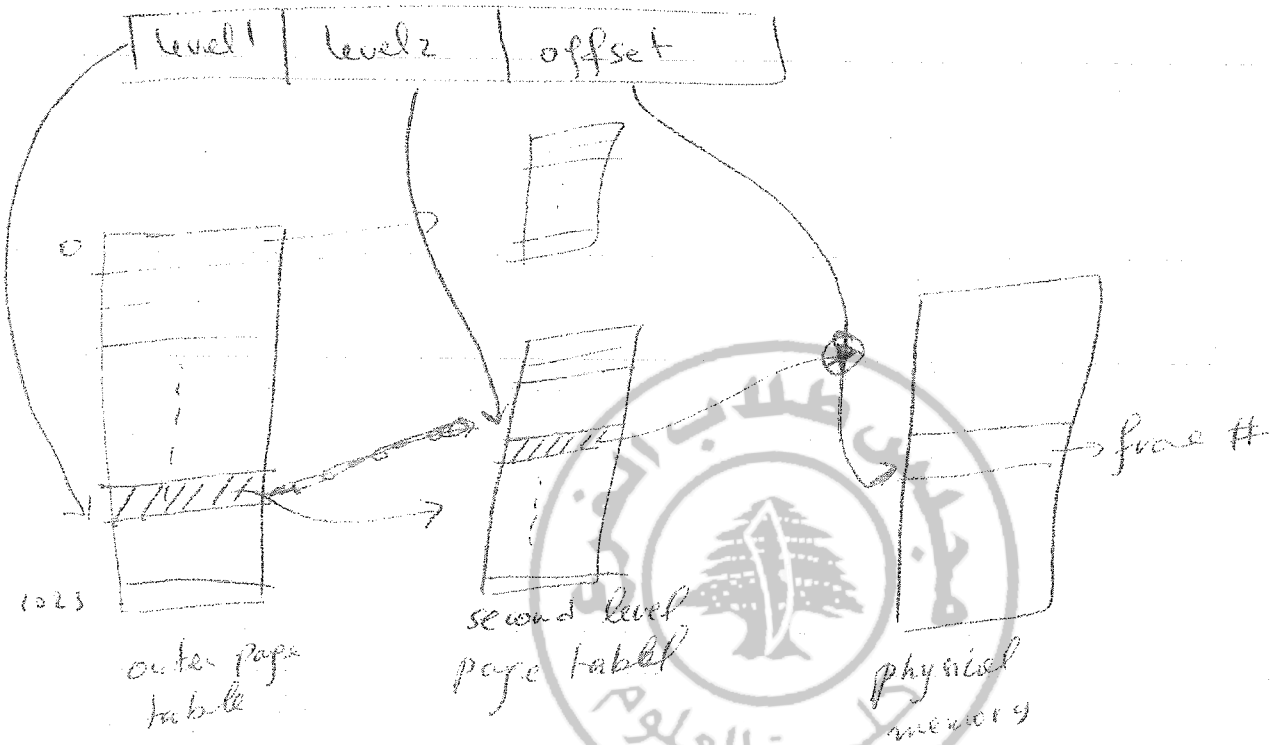


(B1)

there are two page table indexes:

- the first page index is used to index the outer page table
- each entry in the outer page table can possibly point to a second level page table
- the present bit in the outer page table is used to indicate if there is a second level page table for this entry

* the second page index is used to index the second level page table pointed to by the outer page table
* the entry of the second level page table might contain the page frame number of the desired page



For example :

for a process with 8MB of code
4MB heap
2MB stack

we need :
2 entries for the code
1 entry for the stack
1 entry for the heap
outer page table → 4x1024 bytes = 4KB

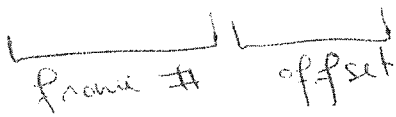
So we need just
- the outer page table
- the inner page table } for data and heap
- the inner page table for stack

=> 4KB + 4KB + 4KB = 12KB

while in single-level page table it needs 4MB

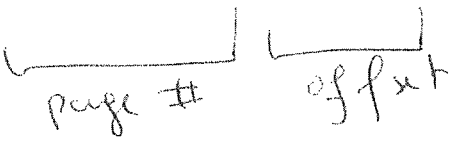
b) logical & physical address

the logical address is used by each process in ~~each~~ its virtual address space starting from 0 to max-1 bytes depending on the architecture and it is private for each process, whereas the physical address ~~space~~ is the real address on physical memory. ~~It~~ It has two parts



(2)

virtual



B) a) zombie state

a zombie process is a process that has completed execution (via the exit system call) but still has an entry in the process table waiting his parent to get its status by the (wait call) (2)

b) orphaned state

An orphan process is a process that is still executing, but whose parent has died (2)

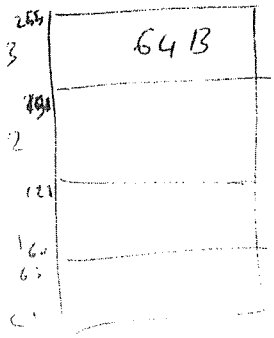
→
```
#include <unistd.h>
#include <stdio.h>

int main()
{ int pid; int i=0;
  pid = fork();
  if(!pid)
     while(i<=100) sleep(10);
  else
     exit(1);
} return 0;
```

(5)

(2)

c) (15) pts

Part 2 : Memory management (15 pts)



$P_1 \rightarrow$ 4 pages

$P_2 \rightarrow$ 2 pages

1) Size of ~~each~~:

$64 \times 4 = 256$ Byte $= 2^8$ Bytes (3)

2) Size of the virtual address space

~~Three~~ Page size $= 64$ By $= 2^6$ Bytes

$\Rightarrow$ 6 bits is needed for the offset

- The logical address is in hexadecimal (2 digits)
- each is encoded on 4 bits $\Rightarrow$ the logical address is encoded on 8 bits $\Rightarrow$ # (2) offset (6)

$\Rightarrow 2^2$ pages $\Rightarrow 4 \times 64 = 256$ Bytes $= 2^8$ (3)

3)

3F $\Rightarrow$	0011 1111	63	0	P_1
4A $\Rightarrow$	0100 1010	74	1	P_1
1D $\Rightarrow$	0001 1101	29	0	P_1
00 $\Rightarrow$	0000 0000	0	0	P_2
CA $\Rightarrow$	1100 1010	202	3	P_1
87 $\Rightarrow$	1000 0111	135	2	P_1
39 $\Rightarrow$	0011 1001	57	0	P_2
2B	0010 1011	43	0	P_1
00	0000 0000	00	0	P_1
11	0001 0001	17	0	P_2

$\Rightarrow$ ~~The W = {0, 1, 0, 0, 3, 2, 0, 0, 0, 0}~~

$W = \{(0;P_1), (1;P_1), (0;P_1), (0;P_2), (3;P_1), (2;P_1), (0;P_1), (0;P_2), (0;P_1), (0,P_1), (0;P_2)$

~~4 page faults~~

a) FIFO

0	0	0	0*	0*	0	0	0*	0*	0*
1		1	1	1	:	1	1	1	1
2					3	3	3	3	3
3						2	2	2	2

(3)

a) FIFO

Page refs	Fault?	4 page frames: Page contents			
(0, P1)	✓	(0,P1)			
(1, P1)	✓	(1,P1)	(0,P1)		
(0, P1)	X	(1,P1)	(0,P1)		
(0, P2)	✓	(0,P2)	(1,P1)	(0,P1)	
(3, P1)	✓	(3,P1)	(0,P2)	(1,P1)	(0,P1)
(2, P1)	✓	(2,P1)	(3,P1)	(0,P2)	(1,P1)
(0, P2)	X	(2,P1)	(3,P1)	(0,P2)	(1,P1)
(0, P1)	✓	(0,P1)	(2,P1)	(3,P1)	(0,P2)
(0, P1)	X	—	—	—	—
(0, P2)	X	—	—	—	—

6 page faults (3)

b) LRU

Page refs	Fault	4 Page Frames: Page contents			
(0, P1)	✓	(0,P1)			
(1, P1)	✓	(1,P1)	(0,P1)		
(0, P1)	X	(0,P1)	(1,P1)		
(0, P2)	✓	(0,P2)	(0,P1)	(1,P1)	
(3, P1)	✓	(3,P1)	(0,P2)	(0,P1)	(1,P1)
(2, P1)	✓	(2,P1)	(3,P1)	(0,P2)	(0,P1)
(0, P2)	X	(0,P2)	(2,P1)	(3,P1)	(0,P1)
(0, P1)	X	(0,P1)	(0,P2)	(2,P1)	(3,P1)
(0, P1)	X	—	—	—	—
(0, P2)	X	(0,P2)	(0,P1)	(2,P1)	(3,P1)

5 page faults (3)

c) Second chance

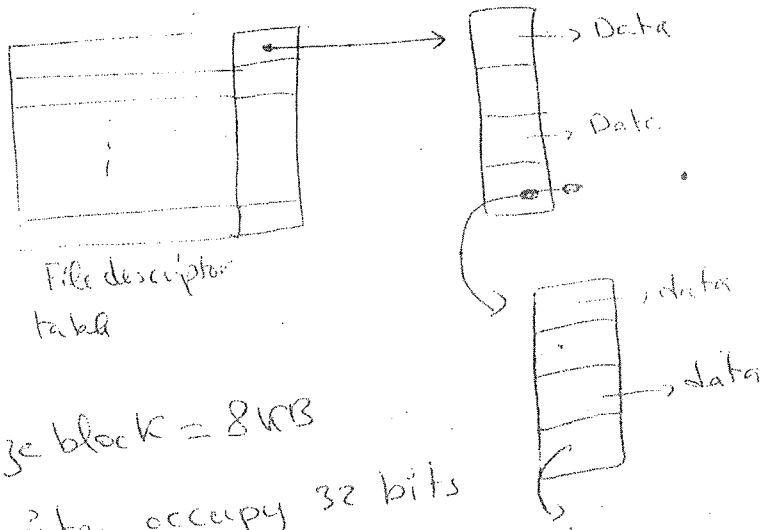
Page refs	Fault?	4 Page Frames: Page contents				out
(0, P1)	✓	(0,P1)				
(1, P1)	✓	(1,P1)	(0,P1)			
(0, P1)	X	(1,P1)	(0,P1)*			
(0, P2)	✓	(0,P2)	(1,P1)	(0,P1)*		
(3, P1)	✓	(3,P1)	(0,P2)	(1,P1)	(0,P1)*	
(2, P1)	✓	(2,P1)	(3,P1)	(0,P2)	(0,P1)	(1,P1)
(0, P2)	X	(2,P1)	(3,P1)	(0,P2)*	(0,P1)	
(0, P1)	X	(2,P1)	(3,P1)	(0,P2)*	(0,P1)*	
(0, P1)	X	—	—	—	—	
(0, P2)	X	—	—	—	—	

5 page faults

(3)

Parts : File system

A) Linked list and indexed allocation strategy



* size block = 8KB
* pointer occupy 32 bits

(10)

Question : max size (# blocks) of a file ?

0 ⟶ n-1 blocks ⇒ n blocks

* each block can contains 8KB/4B = 2K entries

Max File size : data : n * (K-1) * 8KB

index : (n-1) * 4 bytes / 8KB

* each index block points to 2K-1 data blocks = 2047
* pointer occupies 32 bits ⇒ 2^{32} blocks on the system
* if m is the max number of index blocks and all these blocks are used by the file (index + data) then we have :

$$2^{32} = \underbrace{m}_{\text{index}} + \underbrace{(2047 \times m)}_{\text{data}} \Rightarrow m = \frac{2^{32}}{2048} \Rightarrow \boxed{m = 2^{21}}$$

⇒ the number of data block = $2^{21} \times 2047$

⇒ Max file size is : 2^{32} blocks ⟶ 2^{21} index block
⟶ $2^{21} \times 2047$ data blocks

B) void delete_block (int F, int log_b_d)

{ ~~if the file is short, so~~

// the file is long, so we should know in which map
// block is the block to delete.
// if it is in the last so no problem, we load the
// map block and shift all entries back one
// if not ⇒ we should shift all entries in the map
// block containing the block to delete and all
// other following map blocks

int nb_of_map_blocks=0;
int map_of_block_to_delete; int remaining_map_blocks=0;

nb_of_map_blocks = fdesc[F].lg%512==0 ? fdesc[F].lg/512
: (fdesc[F].lg/512)+1;

map_of_block_to_delete = log_b_d/512;

if (map_of_block_to_delete == (nb_of_map_block)-1)
{ //the block to delete is on the last map block of the file
// so, just shift the entries of this map block
load(F, map_of_block_to_delete);
x = log_b_d % 512; fdesc[F].pbd = fdesc[F].map[x];
for (int i=x+1; i<512; i++)
map[i-1] = map[i];
block_release (fdesc[F].pbd);
}

15 pts

```
else   // the block to delete is inside a map block but not
{      // the last one, so we should shift all entries
       // of the following map block one back

   remaining_map_blocks = nb_of_map_blocks - (map_of_
                                        block_to_delete) - 1;
   x = log_b_d % 512;
   for ( i = map_of_block_to_delete; i < nb_of_map_block; i++)
   {
      for (j = x+1; j <((new_map_block * 512) + 512 - x); j++)
      {
         if (j % 512 == 0) Load_map (F, j);
            map[j-1] = map[j]
            map [(j-1) % 512] = map [j % 512];
         } // end for
      } // end for
   } end else

} // end funch—
```


جامعة طلاب الفنون
كلية العلوم

Lebanese University
Faculty of Sciences I

Final 2013/2014
Duration: 2 hours

INFO 324
Operating System II

Problem I

20 points

We consider a system with paginated memory. This memory is composed of 4 frames where the size each frame is 4 bytes. This memory is initially empty. Two processes P1 (3 pages) and P2 (4 pages) are executed. The processor send request to memory in the following order with the format [logical address, process]:

[00100, P1]
[00011, P1]
[01010, P1]
[01001, P2]
[00001, P1]
[00101, P1]
[00111, P2]
[00011, P1]
[01000, P1]
[01110, P2]

W = {1, 0, 2, 4, 0, 1, 3, 0, 2, 5}

1. What is the size of the memory?
2. What is the size of the virtual address space?
3. Indicate using figures the evolution of the memory and the number of page defaults using the following page replacement strategies :
   a) FIFO
   b) LRU (least recently used)
   c) Second chance.
4. We consider using the FIFO strategy, and the above requests by processes1 and 2 to memory are executed. What is the physical address corresponding to the logical address [00110, P2]?

Problem II

25 points

We consider a FS where the topo table contains 12 entries where the first and the last have a single level of indirection and the reminder entries points to a data blocks directly. The size of the block is 64 KB and its number (address) occupies 4 bytes.

1. What is the maximum size of a file in the system?
2. Suppose we have a table fdesc in memory where each entry has the following attributes:

```
#define maxf
struct{
      int lg;
      int ldb, pdb, lmb, pmb; // logical and physical map and data block
      int topo[ x ];
      int map[ y ];
      char buffer[ z ];
      ...
} fdesc[maxf];
```

Rewrite the functions already seen in the class ***add_map(int lmb, int f)***, ***load_map(int lmb, int f)*** and ***add_data(int ldb, int f)*** in function of the above FS.

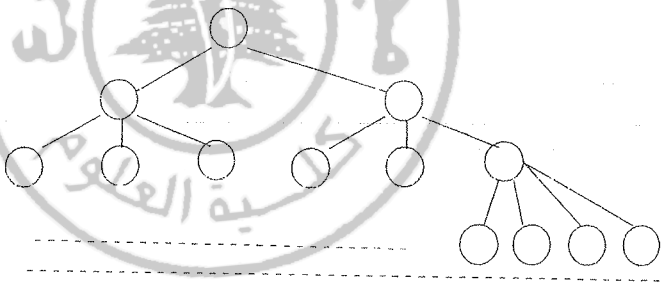
The use of other functions seen in the class is permitted.

Problem III — 25 points

1. Draw the graph generated by the following program. Justify your answer.

```
void main(){
        int N = 5, i, M;
        int t[5][2];

        for(i=1 ; i<=N; i++){
                pipe(t[i-1]);
                M = N-i;
                if(fork())
                        write(t[i-1][1], &M ,sizeof(int));
                else{
                        read(t[i-1][0], &N ,sizeof(int));
                        i=0;
                }
        }
}
```

2. We want to write a program C under Unix that creates the following tree of processes :
The parent at level 1 must create two child processes. The two Childs at level 2 must each create three child processes and so on. In general, each process at level M creates (M+1) child process.

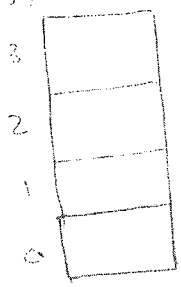


   a. Write the program such a way that the creation of processes as described doesn't stop (infinite program)

   b. Rewrite the program such that a **maximum number of processes (N)** must be created in the global tree of processes.

INTO 524 2014 – first session

Problem 1 : (20pts)



size of frame = 4 bytes

P_1 : 3 pages

P_2 : 4 pages

1) the size of physical memory is 4 × 4 = 16 bytes (2)

2) logical address is on 5 bits ⇒ max address = 2^5 bytes

⇒ 3 bits (#pages) | 2 bits (offset)

⇒ nb of pages is 2^3 = 8 pages (3)

size of virtual address space is 8 × 4 bytes = 32 bytes

3) FIFO w = (P_1, P_1, P_1, P_2, P_1, P_1, P_2, P_1, P_1, P_2)
= (1, 0, 2, 2, 0, 1, 1, 3, 2, 3)

			P_2	P_1	P_1	P_2	P_1	P_1	P_2	
1	1	1	1	1	1	1	1	1	1	
	0	0	0	0	0	0	6	0	3	
		2	2	2	2	2	2	2	2	
			2	2	2	2	2	2	2	
F	F	F	F	h	h	F	h	h	F	

nb faults = 6 (4)

For the following suppose

P_1 : 0, 1, 2

P_2 : 3, 4, 5

⇒ w = {1, 0, 2, P_2 4, 0, 1, P_2 3, 0, 2, P_2 5}

b) LRU

1	1	1	1	1	1	1	1	1	5
	0	6	0	0	0	0	6	0	0
		2	2	2	2	3	3	3	3
			4	4	4	4	4	2	2
F	F	F	F	h	h	F	h	F	F

nb faults = 7 (4)

c) Second Chance :

$w = \{1, 0, 2, 4^{P_2}, 0, 1, 3^{P_2}, 0, 2, 5^{P_2}\}$

Page	Faults	0	1	2	3
1	yes	1*			
0	yes	0*	1*		
2	yes	2*	0*	1*	
4	yes	4*	2*	0*	1*
0	No	4*	2*	0*	1*
1	No	4*	2*	0*	1*
3	yes	3*	4	2	0
0	No	3*	4	2	0*
2	No	3*	4	2*	0*
5	yes	5*	3*	2	0

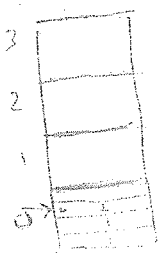
page faults = 6

d) FIFO

[00 | 10, P_2] $\Rightarrow w = \{1, 0, 2, 4^{P_2}, 0, 1, 3^{P_2}, 0, 2, 5, 1^{P_2}\}$

0	1	1	1	1	1	1	3	3	3	3	3	3
1		0	0	0	0	6	0	6	0	5	5	5
2			2	2	2	2	2	2	2	2	1	1
3				4	4	4	4	4	4	4	4	4
	F	F	F	F	F	F	F	F	F	F	F	F

Page Faults = 6



this page is loaded into Frame 2 with offset 2

⇒ the physical address is ~~bytes~~ 2

$2 \times 4 + 2 = 10$

Problem 1 (25 pts)

Topo : 12 entries

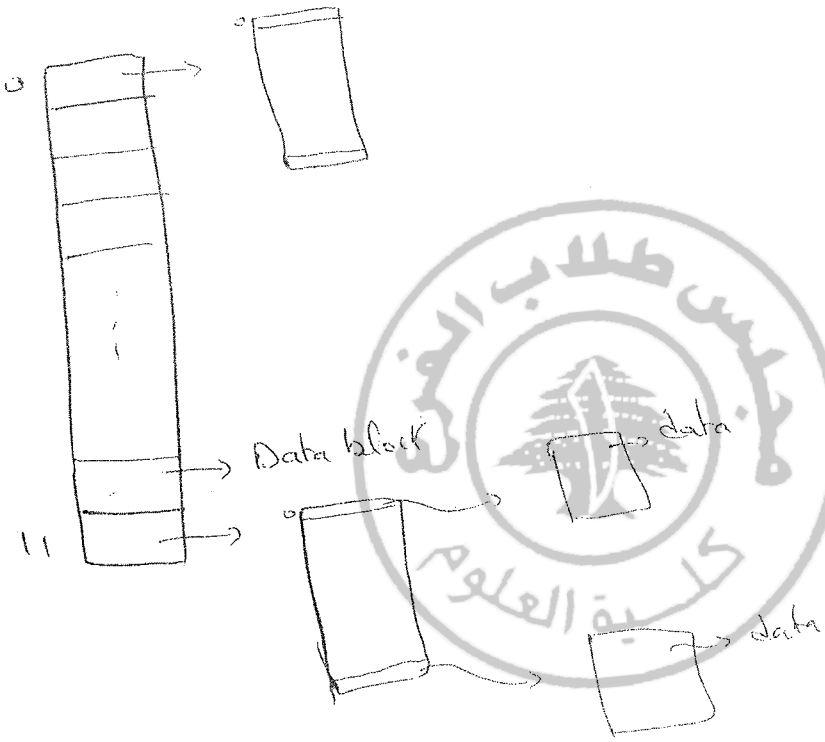
block size = 64 KB

block_nb = 4 bytes

1) The max size is :

the map block contains $\frac{64\,KB}{4\,B}$ = 16 K entries

So it can covers 16000 x 64 KB



The max size is : 2 x 16000 x 64 + 10 x 64 = (2) KB

2) x = 12 (3)

y = 16000

z = 64000

```
void add_map (int F, int log_b_m)
{
  if (log_b_m <> 0 || log_b_m <> 11)
      error ("invalid");
  if ( fdesc[F].map_modified)
      { fdesc[F].map_modified = 0;
        disk_write ( fdesc[F].map, fdesc[F].pbm);
      }
  fdesc[F].lbm = log_b_m;
  fdesc[F].topo[log_b_m] = block_allocate();
}

void load_map(int F, int log_b_m)
{
  if ( log_b_m <> 0 || log_b_m <> 11)
      error ("Invalid");
  if (log_b_m) != fdesc[F].lbm)
    { if ( fdesc[F].map_modified)
        { disk_write (fdesc[F].map, fdesc[F].phm);
          fdesc[F].map_modified = 0;
        }
      fdesc[F].lbm = log_bm;
      disk_read ( fdesc[F].map, fdesc[F].topo[log_b_m]);
    }
}
```

(4)

(6)

```
void add_data (int t, int log_b_d)
{
  int mapblocknb, entryInMap;
  if (fdesc[F].buffermodified)
  { disk_write (fdesc[F].buffer, fdesc[F].pbd);
    fdesc[F].buffermodified = 0;
  }
  fdesc[F].pbd = block_allocate ();
  if (log_b_d >=16000 && log_b_d <=16000)   // direct data
  {
    (2) fdesc[F].topo[log_b_d - 16000] = fdesc[F].pbd;
  }
  else if (log_b_d < 16000 && log_b_d >= 0)   // first map block
  (2) {
    mapblocknb = 0;
    entryInMap = log_b_d % 16000;
  }
  else { mapblocknb = 1;
    entryInMap = (log_b_d - 16010) % 16000;
  }
  if (entryInMap == 0)
    add_map (F, mapblock_nb);
  else
    load_map (F, mapblocknb);
  fdesc[F].map[entryInMap] = fdesc[F].pbd;
  fdesc[F].map_modified = true;
}
fdesc[F].lbd = log_b_d;
}
```

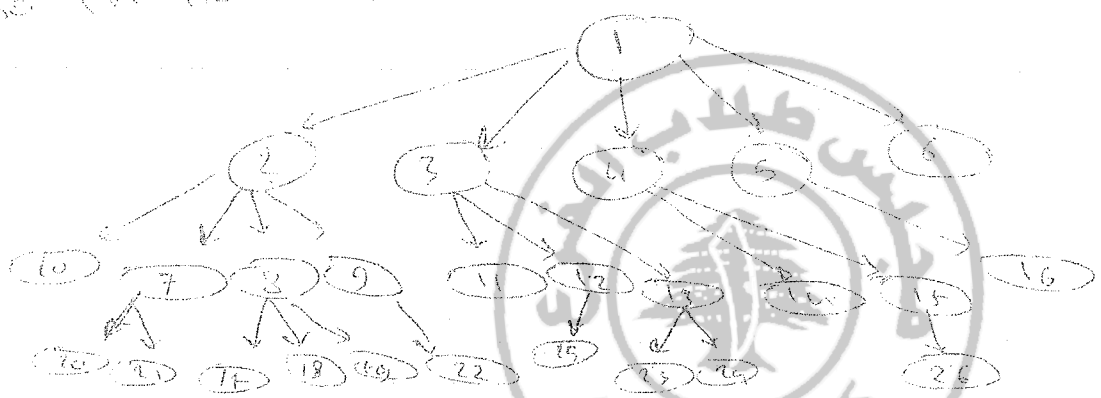
Problem II (25 pts)

the main process should fork 5 child process

* the first child have N = 4
* the sec child have N = 3
* the 3 child have N = 2
* the 4 child have N = 1
* the 5 child have N = 0

each child should fork N child given the value of N he has

So the tree is 3



$5+4+3+2+1$

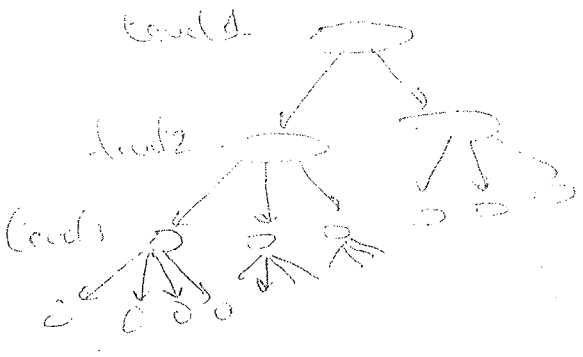
The total nb of processes is $2^5 + 1$

$1+5+4+3+2+1+3+2+1+2+1+1 = 26$ processes

```
void main()
{
  int level = 0;
  int N = 1; int i; int fd[2]; pipe(fd);
  for(i=0; i<N; i++)
  {
    N = N+1;
    if (fork())
      write(fd[1], &N, sizeof(int));
    else {
      read(fd[0], &N, sizeof(int));
      i = 0;
    }
  }
}
```

b) Maximum number of processes is created

$1 + 1 \times (1+1) + 2 \times (2+1) + 3 \times (3+1) + 4 \times (4+1) + \dots + m \times (m+1)$ (5)



$f_0 = 1$

$f_1 \longrightarrow 1 \times 2$

$f_2 \longrightarrow 2 \times 3$

$f_3 \longrightarrow 6 \times 4$

$f_4 \longrightarrow 24 \times 5$

$\ell_2 = \ell_1 \times M$

$\ell_3 = \ell_2 \times M$

```
void main()
{
  int level=1; int i;   int fd[2]; int fd1[2];
  int N = 1;
  int NbProcesses = level x N; pipe(fd); pipe(fd1);
  for( i=0; i<N; i++)
  {
  N = N+1;
   NbProcesses += NbProcesses + NbProcesses x M; while(1)?
   level++; if (NbProcesses > atoi(arg[1]) ;
   if (fork())
   {write(fd[1], &N, sizeof(int));
   write(fd1[1], &NbProcesses, sizeof(int)); x
  else {
   read(fd[0], &N, sizeof(int));
   i=0;
   read(fd1[0], &NbProcesses, sizeof(int)); x
  }
  }
}
```

$1! + 2! + 3! + 4!$


جمعية طلاب الفنون
كلية العلوم

Lebanese University
Faculty of Sciences 1

INFO 324 - E

Final 2012-2013
Duration: 2 hours

Problem I — Memory — 25points

1. Given a paging system with 3 frames and the following page-reference string :

   3, 4, 3, 2, 1, 3, 5, 1, 4, 3, 1, 3

   Calculate the page faults number using the following algorithms:
   a. FIFO
   b. LRU
   c. Second chance

2. Assume that you have a page-reference string for a process with m frames (initially all empty). The page-reference string has length p; n distinct page numbers occur in it. Answer these questions for any page replacement algorithms:
   a. What is a lower bound on the number of page faults?
   b. What is an upper bound on the number of page faults?

3. Consider a memory paginated system where the addressing is on 64 bits. The page size is 4 KB. The size of the physical memory is 512 MB.
   a. How much entry do we need in single-level page table?
   b. Given that each entry requires 4 bytes, what size will occupy the page table?
   c. If we want to use multi-level page table, what number of bits in the virtual address is needed for each level? Then conclude the number of levels and how much memory access do we need in each address translation?
   d. If we use a linear inverted page table, indicate the number of entries we need? The size of the inverted page table if each entry takes 16 bytes.

Problem II — Process — 20 points

The purpose of this exercise is to create a client-server application to find an approximation of the number Π. The method used called Monte-Carlo consists as the following:

Repeat (N times):

- Create two random real numbers x and y between 0 and 1
- if $x^2 + y^2 <= 1$ then M=M+1
- $\Pi \simeq (4M)/N$.
- print Π (the current approximation)

End

A. For this problem, a process called pi (parent) will make calls to another process called drawing (chiid) to ask a random number between 0 and 1. For this, pi will send SIGUSR1 to child. The child process will generate x and y and send them through a pipe. When the parent receives x and y, it will calculate if $x2 + y2 \leq 1$. It will display after each request of x and y its evaluation of Π which should be more precise in each iteration.

B. Modify the program on (A) so that if the child is late more than 20 sec to send x and y, the parent send a message to kill it

P.S:

- The Instruction var = ((float) random () / RAND_MAX) place in the float variable a random real number between 0 and 1.
- The headers required to manipulate the signals are signal.h and unistd.h.

The program stops when the number calculated (Π - (4M / N) <epsilon) where epsilon is a given value.

Problem III — FS — 25points

I. We consider a FS in which the table topo contains 10 entries and each entry point to a map (all entries have a single level of indexing). The size of a block is 2 KB (kilobytes), and the number of a block occupies 4 bytes.

1. What is the maximum size of a file given in this system?
2. Assume that this FS assigns for each opened file, a buffer with four entries to manipulate the data in this file (four blocks of data can simultaneously exist in memory). Part of the descriptor- structure is given by:

```
struct fdesc {

...

char buffer [2048][4]

...

}
```

In addition, in case of the 4 entries are full, and upon loading of a new block, the FS uses the FIFO replacement algorithm (First In First Out) to free place in the buffer.

a. Give a definition of minimum relevant structure fdesc taking into account the circumstances mentioned above and which permit to implement the following functions :load_ data(), add_data().
b. Write the function **add_data(int f, int log_b_d)** that add the block (log_b_d) to the file (f).

II. We consider a FS (File System) where the topo table contains 10 entries where each one has two levels of indirection. Given that each block occupies one kilobyte, the number of block occupies 4 bytes and each block contains 32 inodes:

a. What is the maximum size of a file supported by this FS?
b. How much effective disk spaces occupy the file with maximum size?
c. A file containing one million bytes is considered. How many blocks (data and maps) are they required to represent this file on disk? Briefly explain your answer.

The use of other functions seen in the course (supposedly adapted to this FS) is authorized

INFO 324 - E

2012 - 2013

Final

Problem 1 :

1) paging system with 3 frames

3, 4, 3, 2, 1, 3, 5, 1, 4, 3, 1, 3

a) FIFO

3	3	3	3	1	1	1	1	4	4	4	1
	4	4	4	4	3	3	3	3	3	4	4
			2	2	2	5	5	5	5	5	3
*	*		*	*	*	*		*	*	*	*

(3)

9 page faults

b) LRU

3	3	3	3	3	3	3	3	4	4	4	4
	4	4	4	1	1	1	1	1	1	1	1
			2	2	2	5	5	5	3	3	3
*	*	*	*	*		*		*	*		

(3)

7 page faults

c) Second chance

3*	3*	3*	3*	1*	1*	1*	1*	3*	3	1*	1*
	4*	4*	4*	4	3*	3*	3	4*	4*	4	4
			2*	2	2	5*	5*	5	3*	3	3*
*	*		*	*	*	*		*	*		

(4)

8 page faults

2) m frames
length = p
n distinct page

a) what is the lower bound of the number page faults?

n (2 1/2)

b) Upper bound (2 1/2)

p

3) Memory paginated system
- virtual address on 64 bits
- page size is 4 KB
- physical memory 512 MB

a) single-level page table

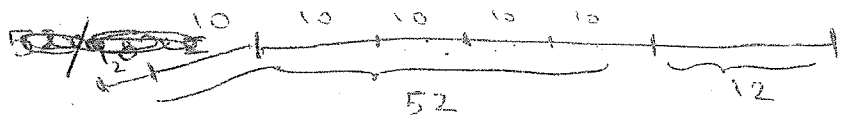
4KB = 2^{12} => 12 bit offset (2)

So, we need 2^{52} entries

b) Each entry requires 4 bytes => the size is 2^{52} x 4 bytes (2)

c) Multi-level page table

since the size of block is 2^{12} = 4KB and each entry is 4 bytes => each block can contain 4KB/4B = 1024 entries => we need 10 bits for each level (3)

=> 

=> 6 levels required

=> we need 6 memory access for each address translation

d) Inverted page table => 512 MB = 2^{29}

=> the nb of entries is $2^{29}/2^{12}$

=> the size is 2^{17} x 16 bytes = 2^{17} entries (2)

Problem II

Approximation of π

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#define Epsilon ...
#define pi 3.14

int main()
{
   int N=0, π=0;
   int fd[2];
   int pid;
   double s=0;
   double x, y;   pipe(fd);  ①
   if (pid=fork())   // Parent context
   { while(true) { ①
     N++;
  ④ kill(pid, SIGUSR1);
     sleep(1);   // wait for 1 seconds to send another request
  ⑥ read(fd[0], &x, sizeof(double));
  ⑦ read(fd[0], &y, sizeof(double));
     if (x*x + y*y <= 1)
        π++;   ①
  ⑤ s = 4*π/N;
     printf("the current approximate value of π in
             iteration %d is : %f\n", N, s);
  ① if ((pi-s) <= Epsilon)
     { printf("the final approximation is : %f\n", s);
       kill(pid, SIGSTP);  ①
       exit(0);
       return(0);
     } // end while
   } // end if        ②
```

```
else // child context
{
  signal (SIGUSR1, handler);
  while (true)
      pause();
  } // end of child
} // end of main

void handler (int sig)
{
  if (sig == SIGUSR1)
  {
      x = (float) random() / RAND_MAX;
      y = (float) random() / RAND_MAX;
      write (fd[1], x, sizeof(double));
      write (fd[1], y, sizeof(double));
      signal (SIGUSR1, handler);
  }
```

3) Modifications in parent context

```
signal (SIGALARM, handler1);
alarm (20);
read (fd[0], ---);
read (fd[0], ---);
alarm (0);

void handler (int sig)
{
  printf ("the time is out... I'll kill you!!!");
  kill (pid, SIGKILL);
}
```

Problem II

1) FS: - topo[10]
- one level of indexing
- size of block is 2KB = 2^{11}
- number of block occupies 4 bytes

1) Maximum size of a file

- the nb of entries in each map block is : $2^{11}/2^2 = 2^9 = 512$

⇒ each map hold 512 data block

⇒ the max size is : $10 \times 512 \times 2KB$ (1)
$10 \times 2^9 \times 2^{11} = 5 \times 2^{21}$

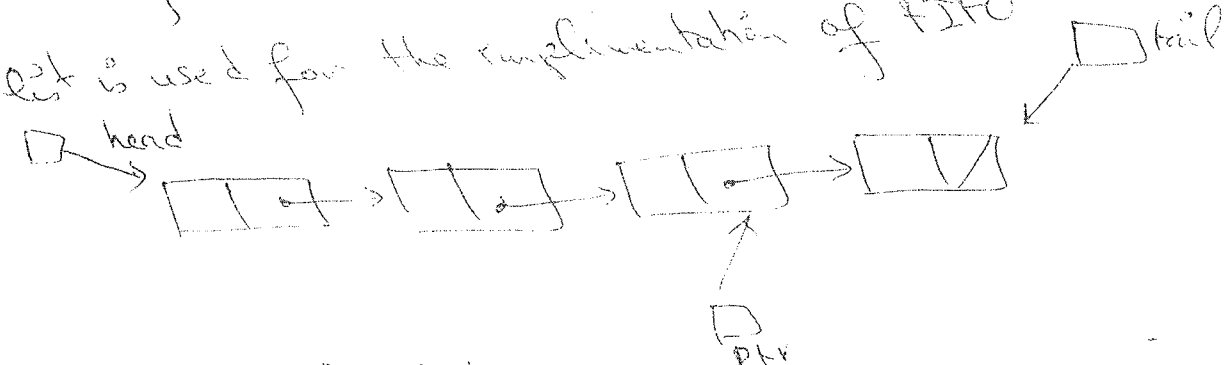
2) a)

```
struct fdesc
{
  ...
  int topo[10];
  int map[512];
  int pbd[4];
  int lbd[4];
  Boolean map_modified;
  Boolean buffer_modified[4];
  char buffer[2048][4];
  struct list *head;
  struct list *tail;
  int buffer_free[4]; // 0 free
                      // 1 full
}
```

(5)

```
struct list
{
  int data;
  //index of buffer
  list *next;
}
```

* the list is used for the implementation of FIFO



- the new node is added on the head
- release from the tail

- for the implementation of FIFO replacement algorithm we will write 3 functions

1) int isfree (int buffer_free[])
this function is used to test if there is a room in the buffer.
it returns the index of the free zone or -1 if not.

2) int release_buffer (struct list *tail)
free a zone into buffer and return its index using FIFO

3) void insert_into_list (int buffer_index, struct list *head)
/* add the recently loaded buffer into the list */

```
int isfree (int buffer_free[])
{ int tab[4];
  tab = buffer_free;
  i=0;
  while (tab[i] <> 0)
        i++;
  if (i>3)
      i=-1;
  return i;
}
int release_buffer (struct list *head, struct list *tail)
{ int buffer_index;
  list *ptr = head;
  while (ptr -> next != nil)
        ptr = ptr -> next;
  buffer_index = tail -> data;
  ptr -> next = Nil;
  tail = ptr;
  return buffer_index;
}
```

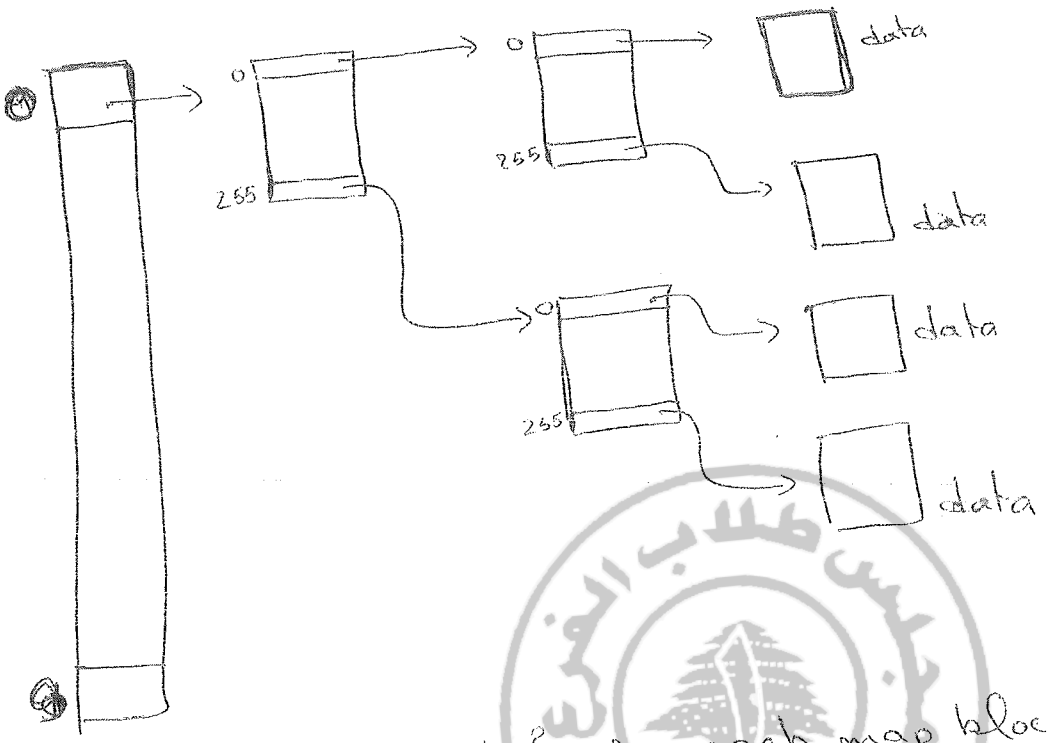
```
void insert_into_list(int buffer_index, struct list *head)
{
    struct list node;
    node.data = buffer_index;
    node.next = head;
    head = &node;
}

void add_data (int F, int log_b_d)
{
    int mapBlockNb, entryInMap;
    buffer_index = isfree(fdesc[F].buffer_free)
    if (buffer_index == -1)  // buffer is full
    {
        buffer_index = release_buffer(fdesc[F].head);
        if (fdesc[F].buffer_modified[buffer_index])
        {
            disk_write(fdesc[F].buffer[buffer_index],
                fdesc[F].pbd[buffer_index]);
            fdesc[F].buffer_modified[buffer_index]=0;
        }
    }
    fdesc[F].pbd[buffer_index] = block_allocate();
    mapBlockNb = log_b_d / 512;
    EntryInMap = log_b_d % 512;
    if (entryInMap == 0)
        add_map(F, mapBlockNb);
    else
        load_map(F, mapBlockNb);
    fdesc[F].map[entryInMap] = fdesc[F].pbd[buffer_index];
    fdes[F].map_modified = true;
} fdes[F].lbd[buffer_index] = log_b_d;
```

(2)

(1)

(4)

II) FS: - size of block is 1KB
- number of block occupies 4 bytes
- topo contains 10 cells with 2 levels of indirection
- each block contains 32 inodes



- we have 256 entries in each map block

a) Maximum size of a file supported by this FS

$10 \times 256 \times 256 \times 1KB = 2^8 \times 2^8 \times 2^{10} \times 10$ bytes

b) Effective disk space for the file with max size

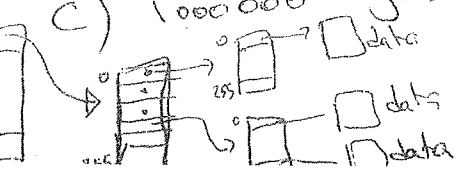
effective = nb of data blocks x size of block
+ nb of map blocks * size of block
+ size of (inode)

$= \underbrace{256 \times 256 \times 10 \times 1KB}_{\text{data}} + \underbrace{(256+1) \times 10 \times 1KB}_{\text{map}}$

1024 ~~+128~~ 64

~~1096~~/32 = ~~128 bytes (size of inode)~~ $= \frac{2^{10}}{2^5} = 2^5 = 64$ bytes

c) 1000000 bytes => 1000 data blocks
=> 0 + ~~4 map blocks + size of inode~~



each contains 256 data block
→ we need 1 map (level 1) and 4 map (level 2)
1000 data block

3	4	4	2	1	3	5	5	4	4	1	3
	3	3	4	2	1	3	3	5	5	4	1
			3	4	2	1	1	3	2	5	4
			*	*	*	*		*		*	*

9 page faults (FIFO)

3	4	3	2	1	3	5	1	4	3	1	3
	3	4	3	2	1	3	5	1	4	3	1
			4	3	2	1	3	5	1	4	4
*	*		*	*		*		*	x		

7 page faults (LRU)

3*	4*	3*	2*	1*
	3*	4*	3*	2
			4*	3
*	*		*	

3*	4*	4*	2*	1*	3*	5*	5*	4*	4*	1*	3*
	3*	3*	4*	2	1	3*	3*	5	5	4	1
			3*	4	2	1	1*	3	1*	5	4
*	*		*	*	*	*	*		*	*	*

9 page faults

string	faults			
3	yes	3*		
4	yes	4*	3*	
3	no	4*	3*	
2	yes	2*	4*	3*
1	yes	1*	2	4
3	yes	3*	1*	2
5	yes	5*	3*	1*
1	no	5*	3*	1*
4	yes	4*	5	3
3	no	4*	5	3*
1	yes	1*	4	3
3	no	3*	4	3*

8 page faults

Lebanese University
Faculty of Sciences 1

INFO 324 - E

2nd Session 2012-2013
Duration: 2 hours

Problem I — Memory — 30 points

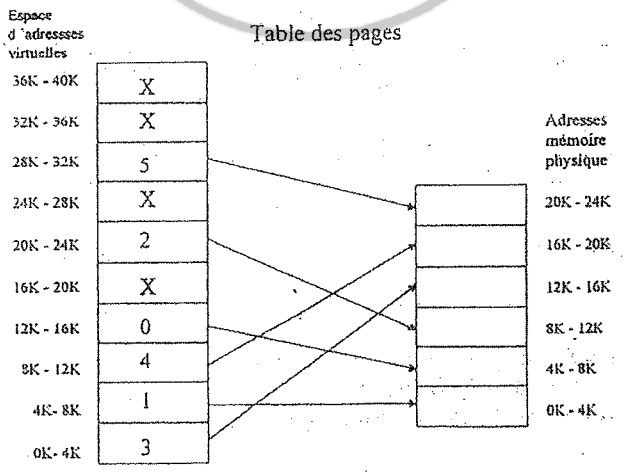
1. Consider the two-dimensional array A:
   int A[][] = new int[100][100];

   Where A[0][0] is at location 200 in a paged memory system with pages of size 200. A small process that manipulates the matrix resides in page 0 (locations 0 to 199). Thus, every instruction fetch will be from page 0. For three page frames, how many page faults are generated by the following array-initialization loops, using LRU replacement and assuming that page frame 0 contains the process and the other two are initially empty?

   a)
   ```
   for (int j = 0; j < 100; j++)
       for (int i = 0; i < 100; i++)
           A[i][j] = 0;
   ```
   b)
   ```
   for (int i = 0; i < 100; i++)
       for (int j = 0; j < 100; j++)
           A[i][j] = 0;
   ```

2. We consider a paginated memory system in two levels with pages of size 4kb each. the addressing scheme is byte by byte on 32 bits, where the addressing on the second level is on 10 bits
   a. How many hyper-pages (first level) are in the system?
   b. Give the decimal address of the following address (39, 1002, 2931).
   c. Determine the following address 561 732 092 in the form of (hyper-page, page, offset)

3. Considering the page table below, give the physical address corresponding to the virtual address: 24, 4300, 13200



Problem II — ES — 15 points

A. A process reads sequentially a file with size 37MB at a rate of 256 bytes at a time. Assume that the size of the block is 512 bytes and a block number occupy 4 bytes. Moreover, the average disk access time is 12ms.

1) The system does not provide disk caching mechanism, i.e., that each read request requires disk I/O operation. Give the total number of disk accesses required and the total I/O disk access time.
2) Now, the system maintains a cache mechanism in the disk, which stores in main memory 1000 disk blocks most recently accessed. Give the total number of disk accesses required and the total I/O disk access time.

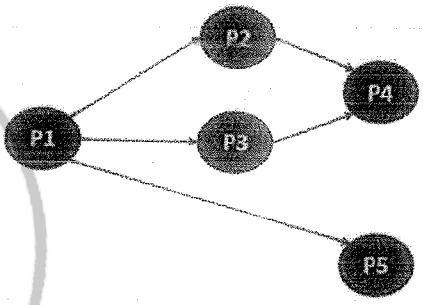
B. Write the function **int similar (int f, char * name_file1, char * name_file2)** that compare the contents of the two files existing in the same open folder with descriptor f. the function must return 1 if the contents are similar, otherwise 0.

The use of other functions seen in the course (supposedly adapted to this FS) is authorized

Problem III: Process — 25 points

Part A :

1. Write a C program where a parent process creates 5 child processes P1, P2, P3, P4 and P5. Suppose that each process Pi execute the function Gi() (you don't have to write the codes). In addition, using the pipes as tool of communication, the order of execution of processes must respect the order presented in the graph (for example P3 does not execute its function G3() before P1 has executed its function G1(), P4 does not execute its function G4() before P2 and P3 have executed their G2() and G3() respectively).



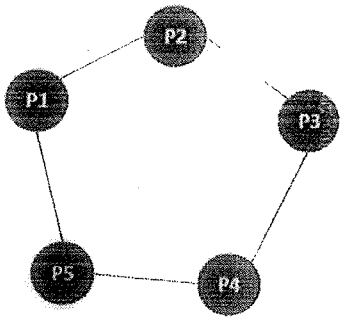
2. Taking into account the solution of the previous question, and assuming that the codes of the Gi() functions are the following:

G1() { i = i+1;} G2() { i = i*2; }	G3() { i = i+3; } G4() { i = i*2; }	G5() { i = i+1; }

i is assumed to be initialized to 0 and its value is exchanged between processes using pipes, provide all possible values of i at the end of the execution of the five processes.

Part B :

Assume now that the 5 child processes of Part A, are placed as presented in the graph, indicating that each process has two neighbors processes (e.g., neighbors of P2 are P1 and P3). We would like to apply synchronization on five processes such as: if the process Pi is executing its function Gi (), neither of these two neighboring process can execute its function. For example, if P2 is currently executing G2 (), then P1 and P3 should be prevented from executing their functions G1 () and G3 (), respectively.



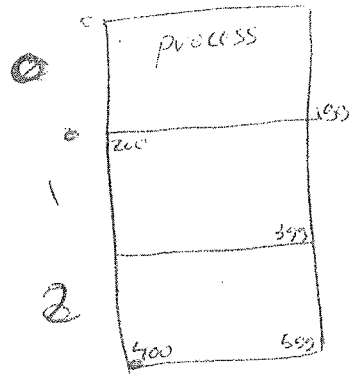
Using communication pipes, write the code needed for this synchronization.

Problem 1

1) int A[][] = new int [100][100];

- memory : page size = 200

- LRU replacement Alg



a) for (int j=0; j<100; j++)
for (int i=0; i<100; i++)
A[i][j] = 0;

Answer : 5000

b) Answer : 50

Explanation :

We assume that the contents of the 2D matrix A are in row order, which means that rows are stored in contiguous main memory locations.

Assume that each integer in the matrix occupies a single memory location

a) the row index (i) is incremented in the inner loop, while the column index is incremented in the outer loop.

As a result, each iteration of the inner loop skips half of the page size by skipping to the same column within the next row. Thus a new page is referenced every 2 iterations around the inner loop.

b) 10000 / 200 = 50 page faults

since each page covers 2 rows or 200 integers

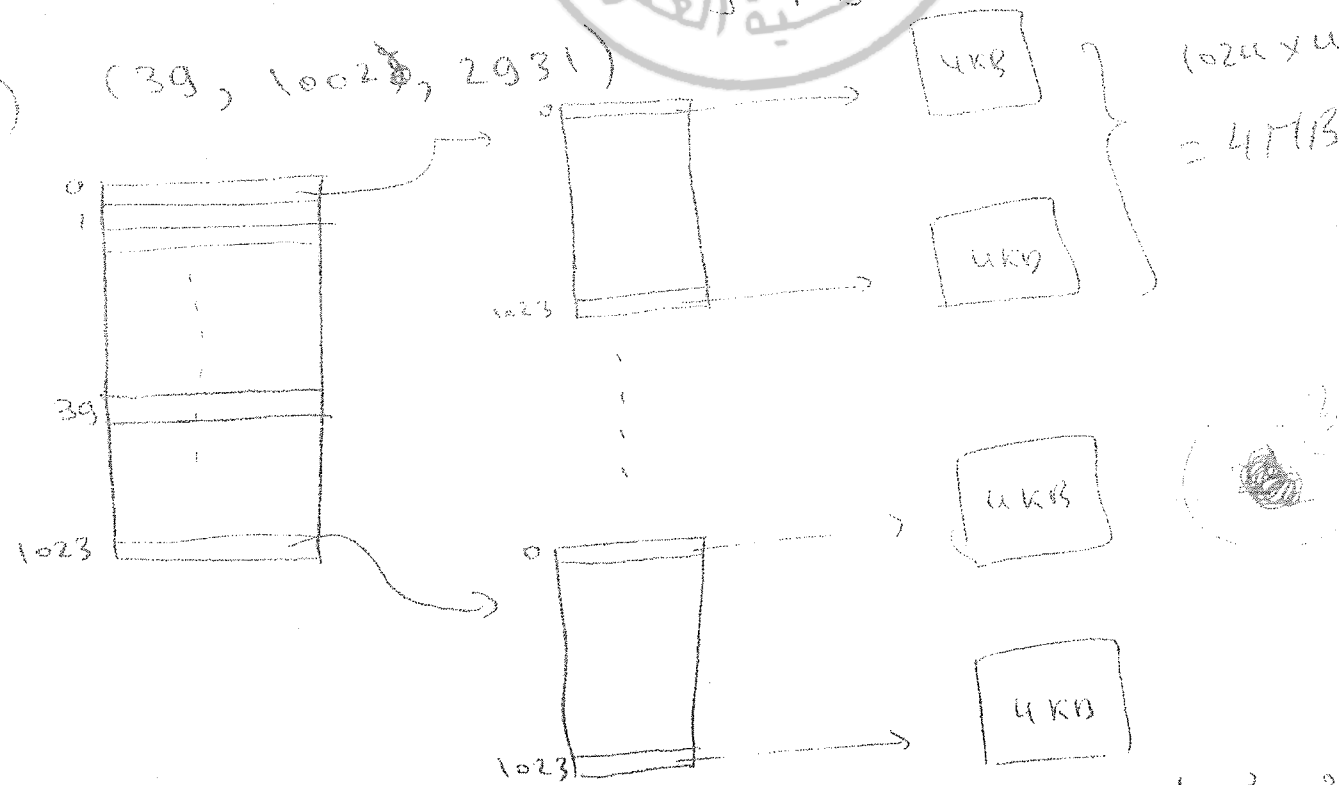
=> 1 page faults for each 200 initializations

2) - Memory paginated with 2 levels
- page size = 4KB = 2^{12}
- addressing on 32 bits
- second level 10 bits

a) How many hyper-pages in the system?

there is 2^{10} = 1024 hyperpages in the system

b) (39, 1023, 2931)



since the page size is 4KB and the nb of entries is 1024

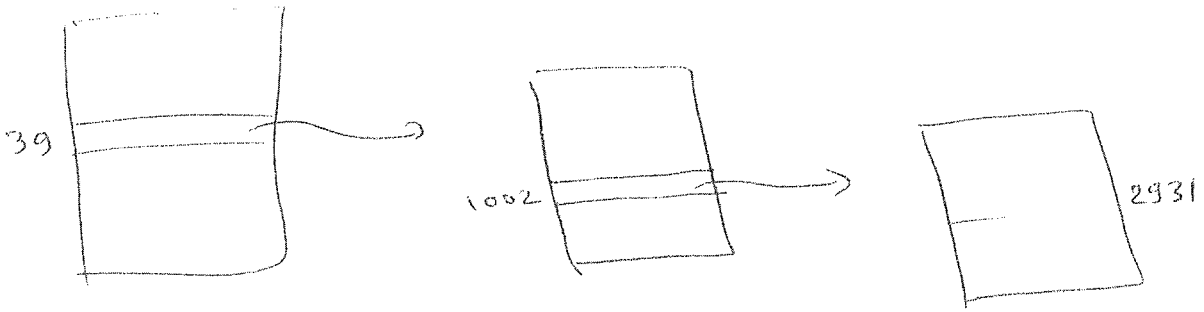
=> the size of each entry is $2^{12} / 2^{10} = 2^2 = 4$ bytes

So, the decimal address is as following

~~39 * 4bytes = 156 +~~ ...

each hyper-page covers 4 MB

$\Rightarrow$ 38 x 4 MB are covered, plus



1001 x 4KB + 2931

So: 38 x 4MB + 1001 x 4KB + 2931 =

~~1593835552~~ + 4100096 + 2931 =

16348679 bytes

(3)

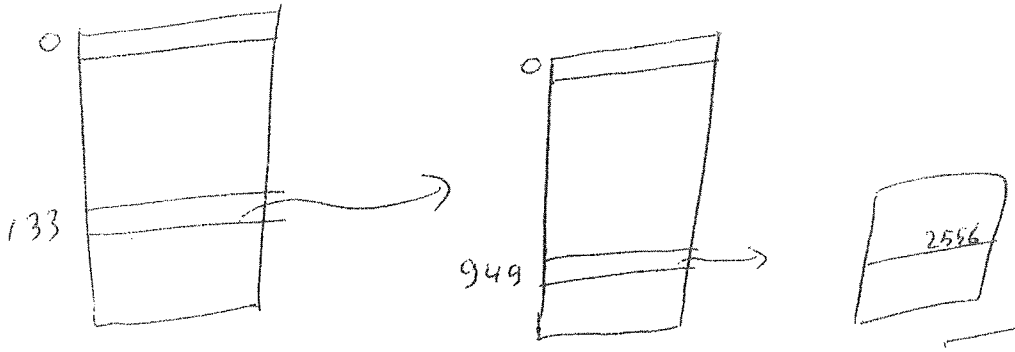
c) 561 732 092 $\rightarrow$ (, ,)

561732092 = 4 x 1024 x 1024 + 3889660

133 * 4MB

$\Rightarrow$ we covers the first 133 entries in the first level

3889660 = 949 x 4KB + 2556



$\Rightarrow$ (133, 949, 2556)

(2)

3) 24
4300
13200

36-40	X	
	X	
	5	
	X	
	2	
	X	4
	0	3
	4	2
4-8	1	1
0-4	3	0

5		20K - 24K
4		16K - 20K
3		12K - 16K
2		8K - 12K
1		4K - 8K
0		0K - 4K

24 → in page 0
⇒ in frame 3 ⇒ 12K + 24 = 12 × 1024 + 24
= 12312

4300 → in page 1
⇒ in frame 1 ⇒ 4 × 1024 + 204 = 4300
the same

13200 ⇒ in page 3
⇒ ~~unmapped~~ frame 0
⇒ 13200 ~~[illegible]~~ = 3 × 4096 + 912
⇒ 0 × 1024 + 912 = 912

Problem II

A)
- file with size 37 MB
- 256 bytes at a time
- block size 512 bytes
- block # occupy 4 bytes
- average disk access time is 12 ms.

1) without cache
- total nb of disk access

the file is composed of 75776 blocks
each block require one disk access ⇒ (5)
24 × 75776 disk access
time : 12 × 75776 × 4 = ...

2) with cache 1000 disk block in memory
each disk block serves 8 read requests
⇒ - 75776 disk access (15)
- time = 12 × 75776

B) int similar (int f, char * name_f1, char * name_f2)
to compare the contents of the two files (5)
to write after()

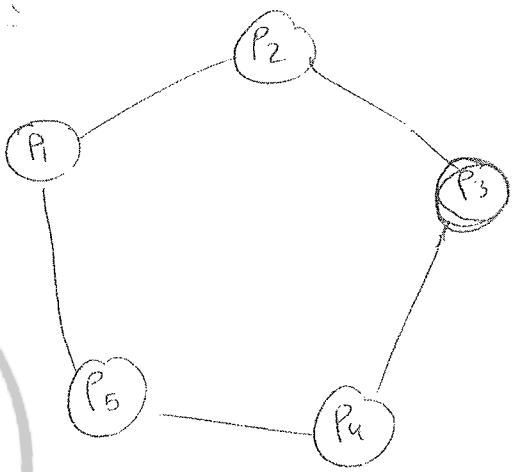
Problem III

Part A :

1)
```
int p12[2], p13[2], p15[2], p24[2], p34[2];
int x;
pipe(p12); pipe(p13); pipe(p15); pipe(p24); pipe(p34);
for(int i=0; i<5; i++)
{
  if (!fork())
  {
    if (i==0) // process p1
    {
      g1();
      write(p12[1], x, sizeof(int));
      write(p13[1], x, sizeof(int));
      write(p15[1], x, sizeof(int));
      break();
    }
    else if (i==1) // process p2
    {
      read(p12[0], &x, sizeof(int));
      g2();
      write(p24[1], x, sizeof(int));
      break();
    }
    else if (i==2) // process p3
    {
      read(p13[0], &x, sizeof(int));
      g3();
      write(p34[1], x, sizeof(int));
      break();
    }
    else if (i==3) // process p4
    {
      read(p24[0], &x, ---);
      read(p34[0], &x, ---);
      g4();
      break();
    }
    else // i=04
    {
      read(p15[0], &x, sizeof(int));
      g5();
      break(); } } }
```

2) The order of execution is :

g_1	g_1	g_1	g_1	g_1	g_1	g_1	g_1 =
g_2	g_3	g_2	g_5	g_5	g_2	g_3	g_3 =
g_3	g_2	g_5	g_2	g_3	g_3	g_2	g_2 =
g_4	g_4	g_3	g_3	g_2	g_5	g_5	g_5 =
g_5	g_5	g_4	g_4	g_4	g_4	g_4	g_4 =
↓	↓	↓	↓	↓	↓	↓	
(11)	(17)	(12)	(14)	(20)	(12)	(18)	(20)



3)
```
int p12[2], p15[2], p23[2], p34[2],
    p45[2]; p21[2], p51[2], p32[2], p43[2]
    p54[2];
int x;
pipe(p12); pipe(p15); pipe(p23); pipe(p34);
pipe(p45); pipe(...
for(int i=0; i<5; i++)
{
  if(!fork()) (1)
  {
    if(i==0)   // P1
    {
      g1();
      write(p12[1], x, sizeof(int));
      write(p15[1], x, sizeof(int));
    } break();
    else if(i==1)   // P2
    {
      read(p12[0], &x, sizeof(int));
      g2();
      write(p23[1], x, sizeof(int));
      write(p21[1], x, sizeof(int));
    } break();
    else if(i==2)   // P3
    {
```

read(p21[0], &x, sizeof(
read(p51[0], &x, sizeof(

```
read(p23[0], &x, sizeof(int));
g3();
write(p3u[1], x, sizeof(int));
write(p32[1], x, sizeof(int));
break();
}
else if (i == 3) // Pu
{
read(p3u[0], &x, sizeof(int));
g4();
write(pu5[1], x, sizeof(int));
write(pu3[1], x, sizeof(int));
break();
}
else if (i == 4) // P5
{
read(p15[0], &x, sizeof(int));
read(pu6[0], &x, sizeof(int));
g5();
write(p51[1], x, sizeof(int));
write(p5u[1], x, sizeof(int));
break();
}
}
}
}
```

Lebanese University
Faculty of Sciences I

02 July 2012
Duration : 2 hours

INFO 324
Operating System II

Problem I — FS — 30 points

A) **[10 points]** We will study a UNIX file system in which the sequential files are on a single disk. The topo table used by the FS consists of 10 entries where the first 9 entries point directly to a data block, and the tenth has only one level of indexing (pointing to a map block). Knowing that each block occupies 2 Kbytes, and the number of a block is 4 bytes:

1. **[3 points]** What is the maximum size of a file supported by this FS?
2. **[3 points]** How much the file with maximum size occupies space on the actual disk?
3. **[4 points]** Consider a file containing 100.000 bytes. How many blocks (data and maps) are required to represent this file on disk? Explain your answer with a draw that shows the configuration of topo and indicating the size of internal fragmentation if they exist.

B) **[20 points]** We are interested in the internal management by UNIX of the open files. First, we suppose that only two data structures are used to implement the management of the open files in the system:

- A local table of file descriptors for every process (that the user can indirectly reference the file using the file descriptor);
- A table of memory i-nodes memory, located in the kernel space, in which each inode contains a reference counter to memorize the number of currently opened files and using this i-node.

1. Justify in which of the two structures must be the following two attributes:
   a. The access rights of opening the file (issued by open ())
   b. The current position of reading and/or writing in the file
2. Describe the impact of the open () system call on these structures. Illustrate the state of these structures if a second process opens the same file.
3. It is possible, with this implementation, to implement the dup() system call? Why?

Now, we propose an intermediate structure of data, (the global table of the opened files in the system), which it intercalates between the table of the descriptors of files and the table of the i-nodes.

4. Where (in which structure) must now be each of the two fields (access rights and current position) of the question (1)?
5. Illustrate the state of these structures if a process opens a file with open(), duplicates it with dup(), and re-opens the same file.
6. Which field must have the entries of the table of the opened files, to allow a correct management of this resource?

Problem II — Processes — 15 points

1. **[6 points]** Write a program C under UNIX where the parent process creates N Childs process P1, P2..., PN which own each a non-common variable **prev** (previous). Each process P_i must have in its variable the identity of the process P_{i-1} ((with the exception of P_1 which must have the value prev = -1).
   In addition, no inter-process communication (pipes, wait / exit, signals) is permitted!
2. **[9 points]** Given three processes (supposed having the same parent) that execute the following programs:

P1 :	P2 :	P3 :
while(1){	while(1){	while(1){
A1();	A2();	A3();
}	}	}

Use just the pipes to synchronize the 3 processes in each of the following **(you do not have the right to use common variables):**


جمعية طلاب النفس
كلية العلوم

a) **[4 points]** The 3 functions $A_i()$ (i=1, 2, 3) should never run concurrently (i.e., at most two of the three can run simultaneously).

b) **[5 points]** The functions $A_i()$ should always run in any order sequentially, for example: (A2 A1 A3) (A2, A3, A1) (A3 A1 A2) ...

Problem III — Memory — 25 points

A) Given the following program:

```
Void main ( ) {
char tab[1999]; // char 1 byte
int i; // integer is 2 bytes
for (i= 0; i < 1999; i++)   tab[i]:= '*'; }
```

The size of the segment of codes is 20 bytes. This program is executed on machine with paginated memory with 1 MB of physical memory and page size 512 bytes. The virtual address is coded on 24 bits.

Questions

a) Determine the size of the virtual address space, the number of bits of the offset, the number of bits of a virtual address, the number of bits of the physical address, the number of bits of the page number and the number of entries in the table of pages.

b) Does the loading of the program into memory generate an internal fragmentation? If not, why? If yes prove it. The pages of the program (0, 1, 2 and 3) are respectively loaded into memory on frames (9, 17, 65, and 33).

c) Give the virtual address and the correspondent physical address [in decimal in the form (page #, offset) generated during the execution of the following instructions : tab[10]= '*'; tab[515]= '*'; tab[1026]= '*' ; tab[1999]= '*'.

B) Consider a virtual memory with pagination where the size of the page is 2 KB and the virtual address is on 32 bits. A: tab [512,512] of integer; // an array is stocked line by line into memory
Given the following two programs:

```
For (j=0;j<512;j++)
    for (i=0; i<512; i++)
        A[i,j]=0;
```

```
For (i=0;i<512;i++)
    for (j=0; j < 512; j++)
        A[i,j]=0;
```

Question: It is assumed that no page is first loaded into memory (allocation on demand); compare the number of page faults in both programs.

C) Consider a memory with demand paging which has 4 frames that are all occupied. In the table below, we give for each page in microseconds, the loading time, the date of last access and the status (referenced, modified and presence).

frame	$t_{loading}$	t_{access}	referenced	modified	presence
0	126	279	0	0	1
1	230	260	1	0	1
2	120	272	1	1	1
3	160	280	1	1	1

What is the replaced page for each of the following four replacement algorithms: LRU, FIFO, NRU and second chance? Justify your answer.

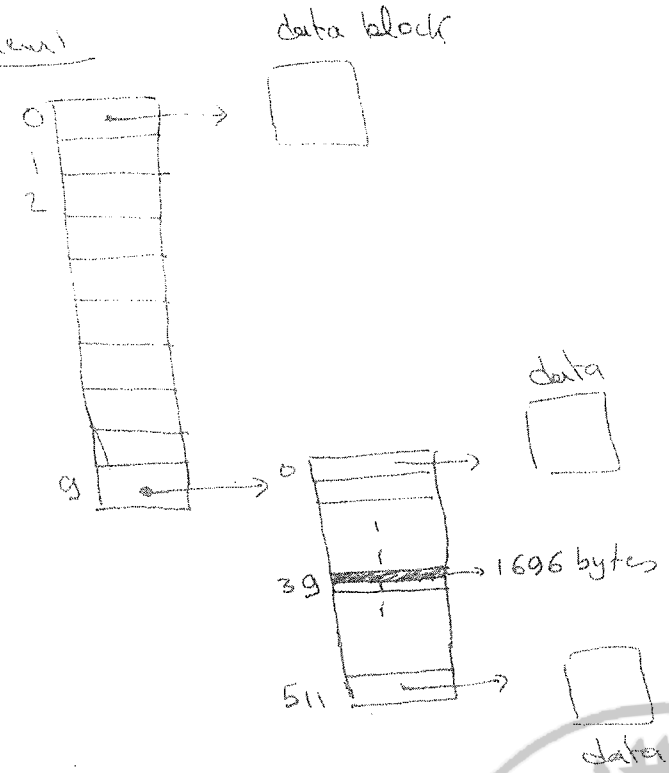

إتحاد طلاب النفس
كلية العلوم

INFO 324

2011 - 2012
Final

Problem 1

A)



each block → 2 KB = 2 x 1024 bytes

block # → 4 bytes

1) Max size = 9 x 2KB + 512 x 2KB

(B1) = 18KB + 1024 KB = 1042 KB = 1067008 bytes

- we have 9 + 512 data blocks
- 1 map block → 2^{11} bytes

⇒ max size on disk is

$2^6 + 2^{11} + 1067008$

$= 1069120$ bytes

2) [redacted] the file with max size occupy:

(A1) - each block contains 32 inodes

(A2) - size of block is 2^{11} bytes ⇒ inode size is $2^{11}/2^5 = 2^6$ bytes

3) 100 000 bytes = 48 x 2048 + 1696

- we covers the first 9 blocks

(A4) - in the map : 0 → 38 are covered

- Internal fragmentation in block 39 = 2048 - 1696

= 352 bytes

B) Part1 : 2 data structures :

- file descriptor table
- Memory i-node table

1) a) Access rights : (2)

- access rights may differ between 2 differents openning
⇒ the access rights should be in the file descriptor table

b) Position of reading & writing

idem. (2)

2) The impact of open()

open() ⟹ - inode disk is copied into inode table in memory if the file is already created
(2) - if not ⇒ it is initialized
- the reference count is set to 1
- a new entry in the file descripton table is associated to the file and the access rights is positioned

If second process open the same file?

⇒ the system detect that an memory inode already exist
(2) ⇒ reference counter ++;
⇒ another new entry is created in the file descripter table of the second process that points to the memory inode and ~~access~~ current position is positionned to the start of the file

3) Dup? why? (3)

=t is not possible, with this implementation to make the dup() call because dup() duplicate the file descriptor

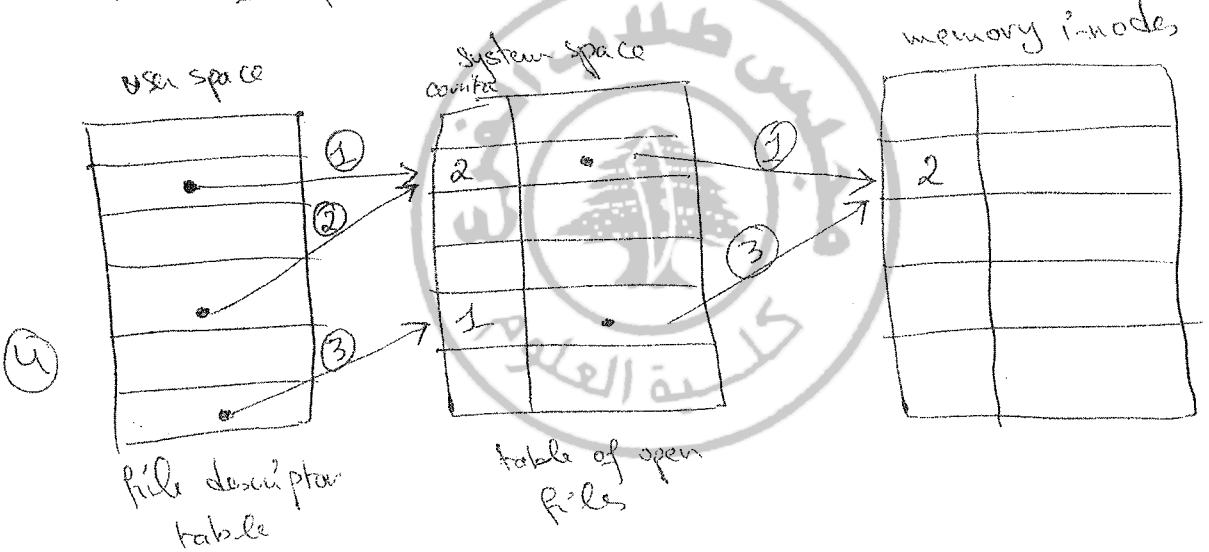
So two différents fd reference the same open file.
Given that with this implementation, the current position is stored in the file descriptor table, duplicate the fd means copying the current position in the new file descriptor.

⇒ we get 2 current positions that can continue independently the evolution.

Part II

4) This new structure can resolve the pbs of dup(). For this reason, the 2 fields must be int it (2)

5)



6) Since several file descriptors can access to the same entry in the table of open files ⇒ every entry must have a

(3) reference counter field, where the value zero indicate that this file must be closed ⇒ decrement to zero the counter field in the table of memory i-nodes

Problem II

1) (6)

```
void main()
{
int pred, i, x, y;
for (i=0; i<N; i++)
  {
    x = fork();
    if( !x && i==0)   // P0
      {
        pred = getppid();
        break;
      }
    if( !x && i>0)
      {
        pred = y;
        break;
      }
    y = x;          // save the id of Pi-1 in y
  } // end for
}
```

2)

```
P1:
while(1)
{
  A1();
}
```

```
P2
while(1)
{
  A2();
}
```

```
P3
while(1)
{
  A3();
}
```

a) at most 2 of 3 must run simultaneously

b)

```
int fd[2];
pipe(fd);
char c;
```

```
P1;
{ read( fd[0], &x, sizeof(char x));
  A1();
  write( fd[1], &x, sizeof(char x));
}
```

P2 :

```
{
write (fd[i], &x, sizeof (char x));
read (fd1[0], &x, sizeof (char x));
A2();
write (fd1[1], &x, sizeof (char x));
}
```

P3 :

```
{ while (1)
  {
    A3();
    write (fd1[i], &x, sizeof (char x));
  }
```

b) should always run in any order sequentially

⑤ Solution : each process should wait the other

ex : P_1 wait P_2 and P_2 wait P_3

Problem III

A)

```
void main()
{
  char tab [1999];
  int i;
  for (i=0; i < 1999 ; i++)
      tab[i] = '*';
}
```

- code segment is 20 bytes
- 1 MB of physical memory
- page size = 512 bytes
- virtual address is coded on 24 bits

Questions

a) - virtual address space is 2^{24} bytes

- offset is coded on 9 bits
- The # of bits of virtual page is 15
- physical address is on 20 bits
- # of bits of physical page number is 20 − 9 = 11 bits
- # of entries in the table of pages is 2^{15} = 32 KO

b) Internal fragmentation ?

Segment code = 20 bytes

array 1999 ⟹ the program needs 1990 + 20 = 2019 bytes

2019 = 3 × 512 + 483

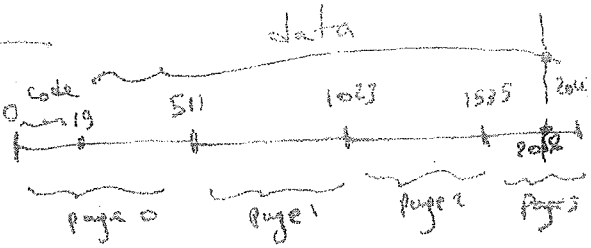
Yes, there is internal fragmentation of 512 − 483 = 29 bytes

c)

page 0 ⟶ case 9
page 1 ⟶ 17
page 2 ⟶ 66
page 3 ⟶ 33

tab[10] = '*' ⟶ (0, 10 + 20) / 10 = 0 × 512 + 10

address	virtual add	Physical address
tab[10] = '*'	(0, 30)	(9, 30)
tab[515] = '*'	(1, 23)	(17, 23)
tab[1026] = '*'	(2, 2)	(65, 2)
tab[1999] = '*'	(3, 483)	(33, 483)



20 ⟶ 2018 data
0 ⟶ 19 code

N.B : tab(10) ⟶ tab[10 + 20] = tab[30]

code

Problem III

:)

Frame	t loading	t access	R referenced	M modified	Presence
0	126	279	0	0	1
1	230	260	1	0	1
2	120	272	1	1	1
3	160	280	1	1	1

what is the replaced page?

- LRU : the page in frame 1
- FIFO : the page in frame 2
- NRU : the page in frame 0
- Second chance : the page in frame 0

(4)

B) page size = 2 KB = 2048 bytes

virtual address = 32 bits

A : tab [512, 512] of integer

A(0,0) A(0,1) ---, A(0,511), A(1,0), A(1,1) --- A(1,511), ----

Prog1 :

```
for (j=0; j<512; j++)
    for (i=0; i<512; i++)
        A[i,j] = 0;
```

Prog2 :

```
for (i=0; i<512; i++)
    for (j=0; j<512; j++)
        A[i,j] = 0;
```

nb of page faults ?

in prog2, the matrix is accessed row by row and thus each needs 512 × 4 = 2048 bytes

(3)

=> each row generates 1 page fault

=> the # of page faults is 512

in prog 1

In this case, the array is accessed column by column so in each outside loop the process, references 512 pages ⇒ the nb of page faults is 512 × 512 (Given we use LRU with 3 frames for ex) (5)

03/164272