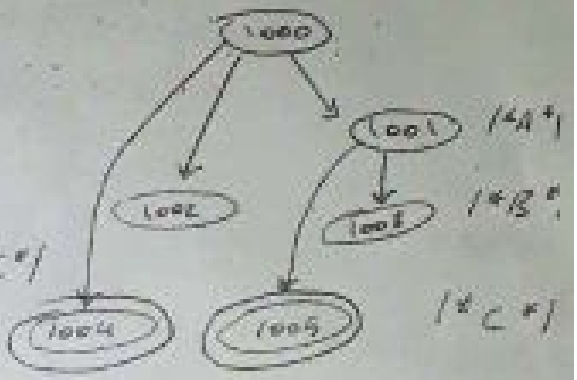
INFO 324 Second Session
2010 - 2011

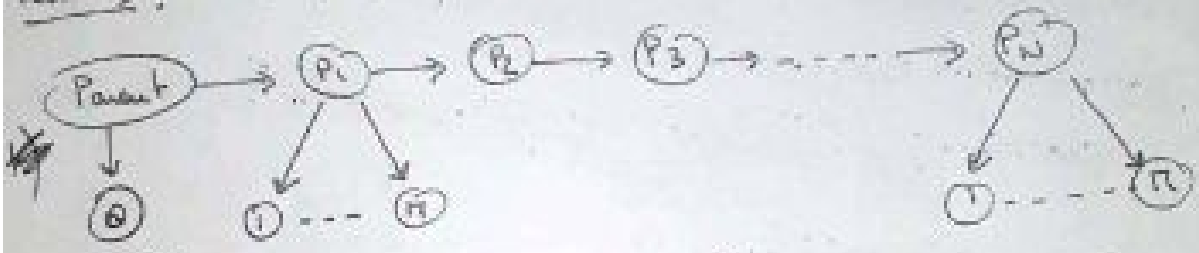
Pb1 (Part A) (15)

```
void main ()
{ fork(); /*A*/
  if (fork())
    printf ("%d\n", fork()); /*C*/
}
```



1004
1005
0
0

Part B: (15 pts)



1) 8 pts

```
#define N 10
int main()
{
  int found = 0;
  int pid[N];
  int current; int parentpid;
  if (fork == 0)
    break;          // 0 process ID
  else              // Parent context
  {
    for (i=1; i<N; i++)
      if (pid[i] = fork())
        break;
  }
  sleep(2);
  current = getpid();
  for (i=1; i<N && !found; i++)
  { if (current = pid[i])
      found = 1;
```

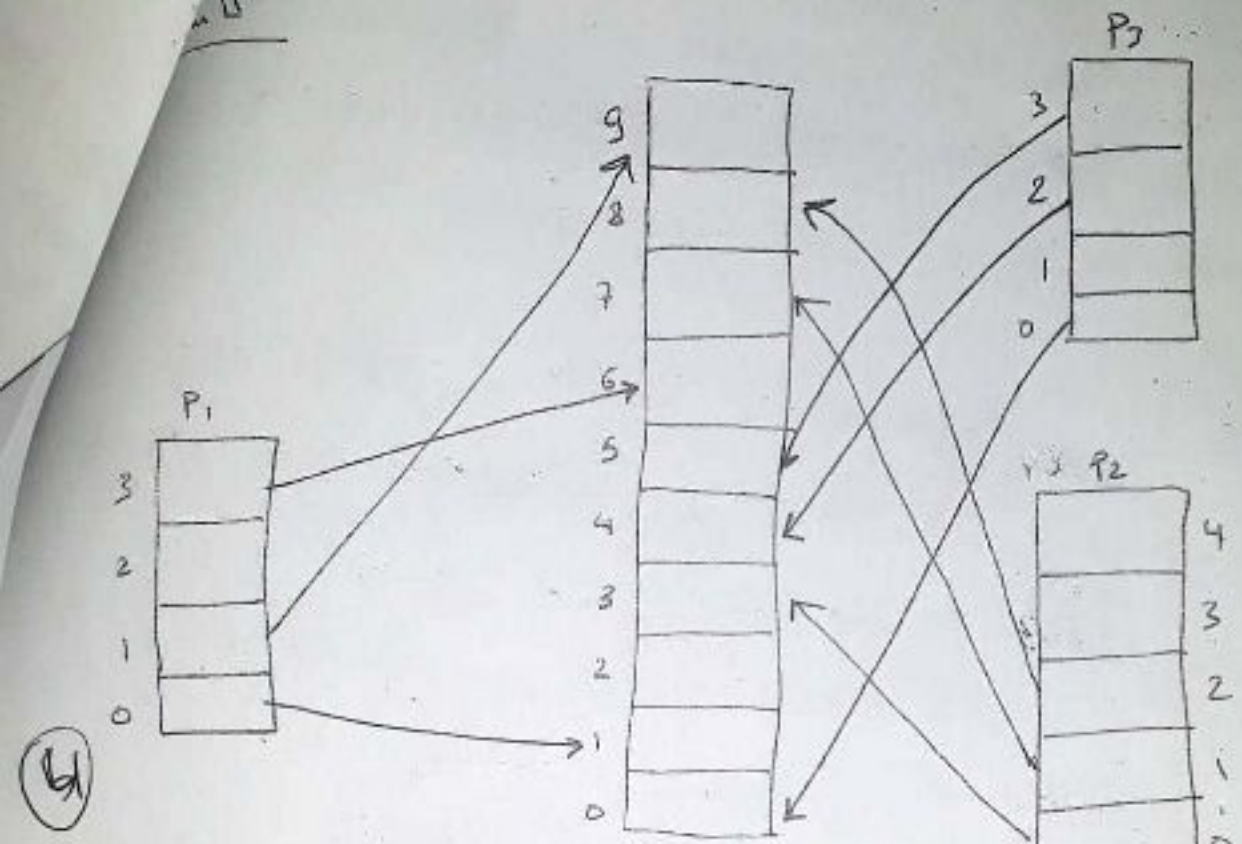
(1)

```
for (j=1; j<=n; j++)
    if (fork()==0)
        break;
}
return 0;
}
```

2) (7pts)

2)

```
parentpid = getpid()
① /* we put this statement before creating all
      other processes */
② if (parentpid = getpid())
      fork(); /* creation of 0 process */

③ while (wait(NULL)); */
   /* put in the loop
```



(b)

Page tables:

P_1

Page #	Frame #	Present	R
0	1	1	1
1	9	1	1
2	—	0	0
3	6	1	1

P_2

Page #	Frame #	Pres	Ref
0	3	1	1
1	7	1	1
2	8	1	1
3	—	0	0
4	—	0	0

Page #	Frame #	Pres	ref
0	0	1	1
1	—	0	0
2	4	1	1
3	5	1	1

<Page 12, offset 4> : access out of memory ⇒ segmentation fault (1)

2) $P_3 = <$ page 2, offset 14 $>$ $\equiv (3, 2) \Rightarrow$ hit (2)

3) ~~$(P_3$, page 2$) \equiv (3, 2) \Rightarrow$ hit~~

$(P_1$, page 2$) \equiv (1, 2) \Rightarrow$ fault page $\Rightarrow$ load into frame 2

$(P_3$, page 2$) \equiv (3, 2) \Rightarrow$ hit

$(P_2$, page 2$) \equiv (2, 2) \Rightarrow$ hit

$(P_2$, page 4$) \equiv (2, 4) \Rightarrow$ fault page $\Rightarrow$ load into frame 9

$(P_2$, page 3$) \equiv (2, 3) \Rightarrow$ fault page $\Rightarrow$ load into frame 6

b) $(P_1$, page 0$) \equiv (1, 0) \Rightarrow$ fault page $\Rightarrow$ Load into frame 3

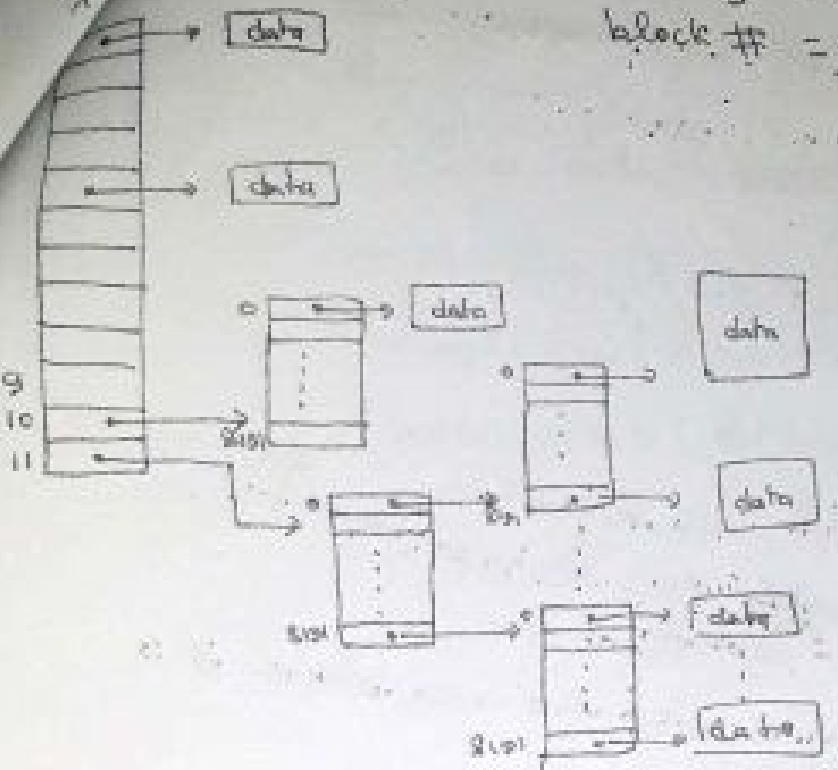
$(P_3$, page 5$) \equiv (3, 3) \Rightarrow$ hit

$(P_1$, page 1$) \equiv (1, 1) \Rightarrow$ fault page $\Rightarrow$ Load into frame 7

$(P_1$, page 2$) \equiv (1, 2) \Rightarrow$ hit

$\Rightarrow$ The nb of page faults is 9 + 5 = 14

block size = 16 KB
block # = 2 bytes



1) what is the max size of a file? (3)

16 KB = 16×1024 = $2^4 \times 2^{10} = 2^{14}$ byte

=> the block occupy $2^{14} \div 2 = 2^{13} = 8192$ entries

=> the max size is: $10 \times 16 + 8192 \times 16 + 8192 \times 8192 \times 16 =$

$10 \times 2^4 + 2^{13} \times 2^4 + 2^{13} \times 2^{13} \times 2^4 =$

$10 \times 2^4 + 2^{17} + 2^{30}$ Kbytes

a) $x = 12$

$y = 8192$ (4)

$z = 8192$

$t = 2^4 \times 2^{10} = 2^{14}$

b) data_free (int fs, in ldb)

(return the logical data block from the file f)

```
void free_data (int F, int logbno)
{
    fdesc[F].pbd = fdesc[F].topo[logbd];
    block_release (fdesc[F].pbd);
}
```

(4)

```
if (! fdesc[F].logg)
    fdesc[F].topo[logbno] = f;
```

c)

```
void mapi_free (int F, int logbm)
{
    load_mapi (F, fdesc[F].topo[logbm]);
    // the buffer mapi contains the map block #
    for (i=0; i< 8192 ; i++)
        block_release (fdesc[F].mapi[i]);
    block_release (fdesc[F].topo[logbm]);
}
```

(a)

d)

```
void map2_free (int F, int logbm)
{
    load_map2 (F, fdesc[F].topo[logbm]);
    // load into buffer map2
    for (i=0; i< 8192 ; i++)
    {
        mapi_free (fdesc[F].map2[i]);
    }
}
```

(4)

(6)

```
void file_delete (int F)
{
    for (i=0; i<10; i++)
        free_data ( F, i);
    map1_free (F, 11);
    map2_free (F, 12);
    inode_release ( fdesc[F].inode_nb);
}
```

Solution 2014 - 2015

2- Process management

a) using 2 or more level page table

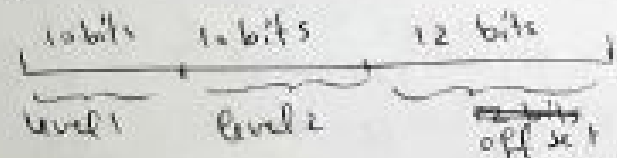
Since each page in the virtual address space needs an entry in the page table, we may have a huge number of entries in the page table where actually a few of them are needed by the process. for example in a virtual address space of 32 bits with page of size 4KB = 2^{12}

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

⇒ 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)



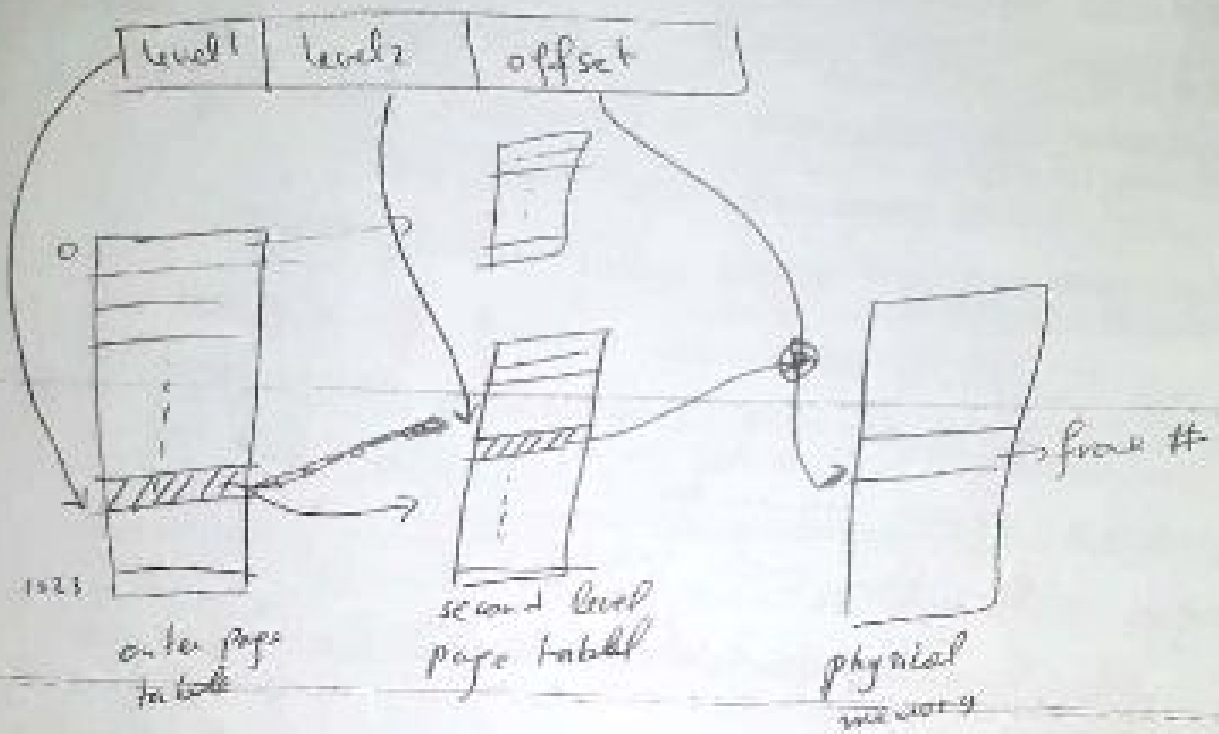
(4)

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 the entry

* the second page table is used to index the second 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

So we need just - the outer page table → 4 x 1024 byte = 4KB
- the inner page table for data and heap
- the inner page table for stack

⇒ 4KB + 4KB + 4KB = 12KB

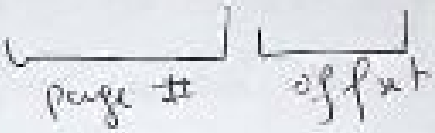
while in single-level page table it needs 4MB

a) a physical address

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 ~~set~~ It has two parts

(2)

virtual



(3) 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)

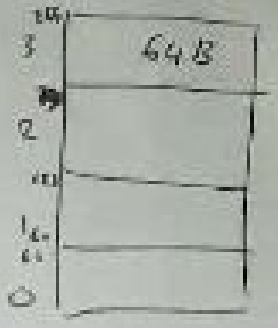
```
#include <unistd.h>
#include <stdio.h>

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

(5)

(2)

Memory management (13 pts)

4 pages
→ 2 pages



Size of memory:

$64 \times 4 = 256 \text{ Byte} = 2^8 \text{ Byte}$ (3)

2) Size of the virtual address space

Page size = 64 By = 2^6 Bytes

⇒ 6 bits are needed for the offset

- The logical address is in hexadecimal (2 digits)
- each is encoded on 4 bits ⇒ the logical address is encoded on 8 bits ⇒ [2 bits] [6 bits]

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

3)

3F ⇒	0011 1111	63	0	A
4A ⇒	0100 1010	74	1	P1
10 ⇒	0001 1101	29	0	A
00 ⇒	0000 0000	0	0	P2
CA ⇒	1100 1010	202	3	A
87 ⇒	1000 0111	135	2	P1
39 ⇒	0011 1001	57	0	P2
2B	0010 1011	43	0	P1
00	0000 0000	00	0	P1
11	0001 0001	17	0	P2

⇒ ~~The W = {0, 1, 0, 0, 3, 2, 0, 0, 0, 0}~~

W = {(0; P1), (1; P1), (0; P1), (0; P2), (3; P1), (2; P1), (0; P2), (0; P1), (0; A), (0; A)}

4) FIFO

~~4 page faults~~

0	0	0	0	0	0	0	0	0	0
1		1	1	1	1	1	1	1	1
2				3	3	3	3	3	3
3					2	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)

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)

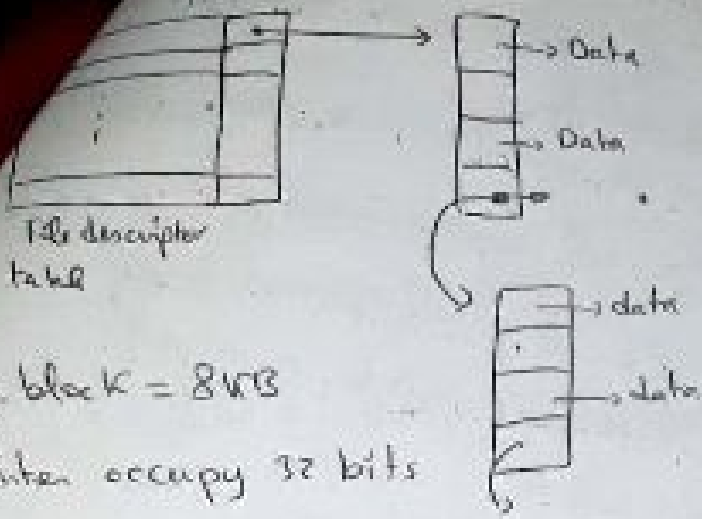
5 page faults (3)

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, P1)*	(0, P1)*	
(0, P1)	X	—	—	—	—	
(0, P1)	X	—	—	—	—	

5 page faults (3)

... 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

data) : $\frac{\text{file size}}{\text{block size}}$

index : $\frac{(n-1) \times 4 \text{ bytes}}{8 \text{ KB}}$

* each index block points to 2K-1 data blocks = 2047

* pointer occupies 32 bits $\Rightarrow 2^{32}$ 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 :

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

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

$\Rightarrow$ Max file size is : 2^{32} blocks $\rightarrow 2^{21}$ index block

$\rightarrow 2^{21} \times 2047$ data blocks

(4)

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
// 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)

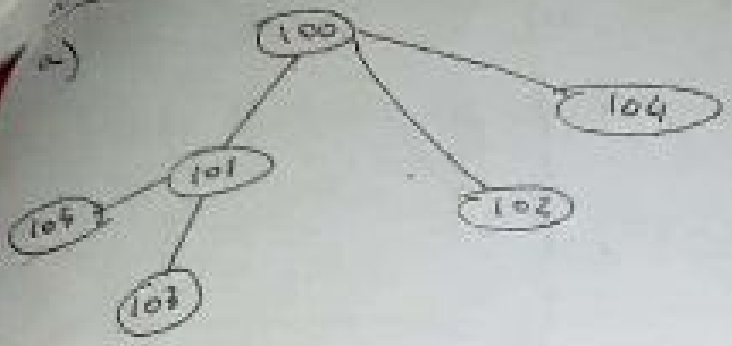
{ //the block to delete is on the last map block of the file
// so, just shift the entries of this map block
load(F, map_of_block_to_delete);
x = log_b_d % 512; fdesc[F].pbd = fdesc[F].map[x];
for (int i=x+1; i<512; i++)
map[i-1] = map[i];
block_release (fdesc[F].pbd);
}

15 pts

```
// ... in the 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—
```

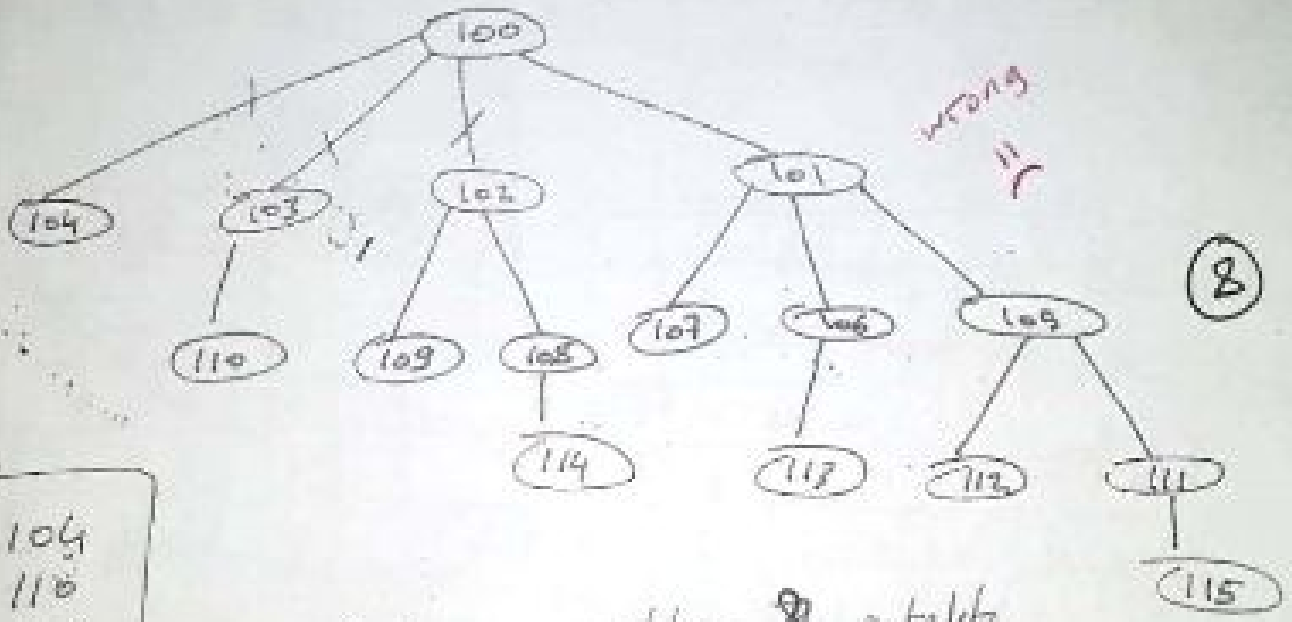
problem 1

a)



5

output: 104
105
} pid of last 2 childs created



wrong

8

104
110
109
112
113
114
115
107

8 outputs

other 8 outputs
prints 0
(child)

the output is 0 0

5

1

a)
```
void handle(int sig) { } (2)
void main() {
  srand(time(0));
(1) signal(SIGALRM, handle);
(1)(1) alarm(1);
(1) pause();
(1) int num = (rand() % 100) + 1;
(1) printf("%d\n", num);
}
```

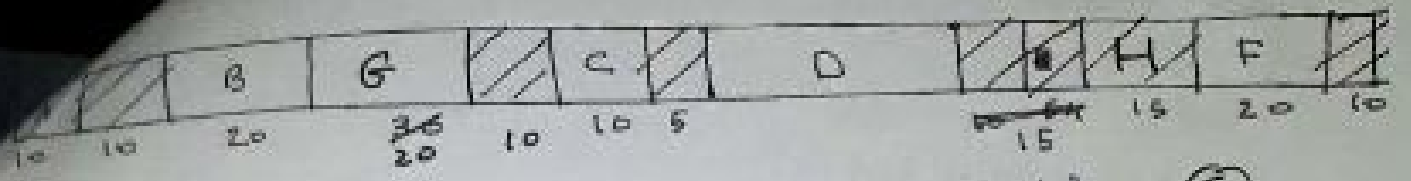
(6)

b)
```
void handler(int sig){
  printf("I'm the parent and I got a msg from my child %d
(3) signal(SIGUSR1, handler);
}
void main(){
(1) signal(SIGUSR1, handler);
(1) for(int i=0; i<10; i++){
      if(fork()) pause(); (2)
      else {
(3)     printf("I'm the child with rank %d\n", i);
        kill(SIGUSR1, getppid()); (1)
      } // end if
    } // End for
} // End main
```

(10) { 7 x 3

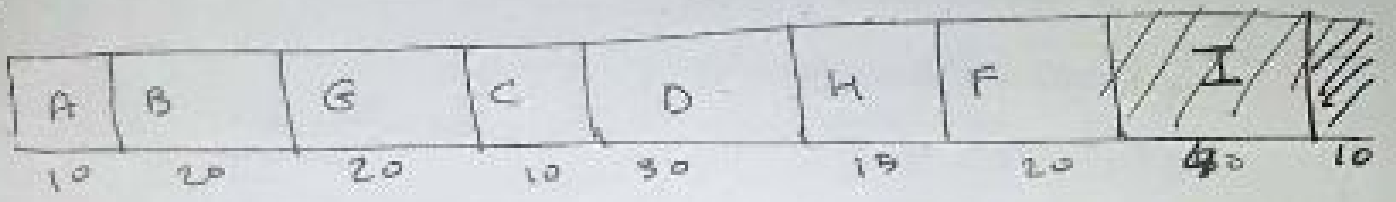
(1) and pid: %d\n (getpid(

(1) break;



no place for pg J (wok) ⇒ fragmentation (5)

⇒



2) $w = \{3, 4, 4, 1, 5, 2, 3, 1, 4\}$

FiFo (4)

ref	Fault	A	B	C	D
3	Y	3			
4	Y	3	4		
4	N	3	4		
1	Y	3	4	1	
5	Y	3	4	1	5
2	Y	2	4	1	5
3	Y	2	3	1	5
1	N	2	3	1	5
4	Y	2	3	4	5

7 page faults

LRU (4)

ref	Fault				
3	Y	3			
4	Y	4	3		
4	N	4	3		
1	Y	1	4	3	
5	Y	5	1	4	3
2	Y	2	5	1	4
3	Y	3	2	5	1
1	N	1	3	2	5
4	Y	4	1	3	2

7 page faults

optimal

optimal (5)

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

6 page faults

(2)

1) void, move. (char *file_name, int f1, int f2)

{

folder_entry ent;

inode binode[16]

int fd, inode_file;

(10)

fd = file_open(file_name)

~~fd = file_open~~

// f1 and f2 are already loaded into

folder in memory

// Just we should save the link of

inode of the file for directory f1

to f2 in entry, then write to disk

// search in data of ~~ft~~ desc[f1] for the

entry containing (filename and inode_file)

and delete this entry

// fp->inode_file = fd.inode_nb.

// add new entry in the ~~desc~~ desc[f2]

write(f2, file_name);

write(f2, inode_nb);

}

```
compiler ( int fp, char *file1, char *file2 )
// fp is the descriptor of directory loaded into memory
// fdesc[fp].topo  contains the data block of fp
int fd1 = open_file ( file1);
int fd2 = open_file ( file2);
char *c1, *c2;
if fdesc[fd1].log <> fdesc[fd2].log
    return 0;
else
{
  while ( !eof(fd1) && !eof(fd2) )
  {
    read (fd1, c1);
    read (fd2, c2);
    if (!strcmp(c1, c2))
      return 0;
  }
  return 1;
}
}
```

(10)

Lagrange University
Faculty of Sciences I

Second Session 2010 - 2011
Duration : 2 hours

INFO 324
Operating System II

Dr. Ahmad Faour

Problem I — Process — 40 points

The parts A et B are independents

Part A) (15)

Consider the following C program under Unix:

```
void main(){
     fork();
     if(fork())
          printf("%d\n",fork());
}
```

Assume that the parent has an identity Pid = 1000, and there is no active processes other those created by the program. Display all the results obtained by executing the above program

Part B) (15)

1. Write a C program under Unix that creates the following tree of processes.

The parent process creates the process P1, and each process Pi creates a process P_{i+1} (i=1...N-1). In addition the parent creates another process Q; and each process Pi (i=1..N) creates a sub-tree of M processes.

Now, we add to the program in part (1) the following two criteria:

- The process Q must be the last process created in the tree (global) of processes.
- After the creation of all processes in the tree, all these process can be in running mode, i.e. the creation of a process should have no relation with the termination of another process

Write the program (the extension of Part 1) that creates the tree of processes in accordance with the two criteria described above.

Problem II — Memory — 15 pts — 20 points

Consider a machine where the memory is managed by the mechanism of paging. The physical memory is divided into a set of 10 frames (frames) numbered from 0 to 9. In this context, three processes P1, P2 and P3 which address spaces are respectively composed of four pages, five pages and four pages (the pages are also numbered from 0). At time t, the configuration of main memory is as follows:

- For P1: pages 0, 1 and 3 are loaded in the frames 1, 9 and 6 respectively.
- For P2 only pages 0, 1 and 2 are loaded into the frames 3, 7 and 8 respectively.
- For P3 pages 0, 2 and 3 are loaded in the frames 0, 4 and 5 respectively. The frames not mentioned above are free.

1. Represent the page tables of processes according to the previous configuration. In this context, the process P2 requests access to the virtual address <page 12, offset 4>, what's going on?
2. The process P3 then asks access to the virtual address, <page2, offset 14>, what's going on?
3. We assume that the pages are loaded in that order i.e., page 0 is loaded from P1, page 1 and page 3; after page 0 is loaded for P2, page 1 and page 2 and likewise for P3. What will be the behavior of the

memory for the following requests as follows: (P3, page2), (P1, page 2), (P3, page2), (P2, page2), (P2, page4), (P2, page3) (P1, page0), (P3, page3), (P1, page 1), (P1, page 2) if we adopt the FIFO strategy? what is the number of page faults?

25 points

Exercise III

We consider a FS where the topo table contains 12 entries, each of the first 10 entries point directly to a data block, the 11th entry has only one level of indexing, and the 12th entry has two levels of indexing. The size of a block is 16 KB (kilobytes), and the number of a block occupies two bytes.

1. What is the maximum size of a file in this system?
2. Descriptors files in main memory are assumed open in the following table tdesc:

```
#define maxf
struct{
      int lg;
      int bld, bpd, blc1, bpc1, blc2, bpc2;
      int topo[x];
      int map1[y];
      int map2[z];
      char buffer[t];
      ...
} tdesc[maxf];
```

- Descriptor may be complemented if necessary by other fields covered in course.

a. Give the values of x,y,z and t.
b. Write the function data_free (int f, int blogd) that permits the release of the logical data bloc (blogd) of the file loaded in the entry f of table of descriptors.
c. write the function map1_free (int f, int blogm) that permits the release of the logical map bloc of first level of indexing (blogm)
d. write the function map2_free (int f, int blogm) that permits the release of the logical map bloc of second level of indexing (blogm)
e. Write the function file_delete (int f) that deleles the file loaded into the entry f, deleting a file is to release all the blocks reserved by the file and its inode.

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

Lebanese University Faculty of Sciences1	INFO 324- OS II Second Session	Sept 2015 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 'C' 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]
[28, 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.

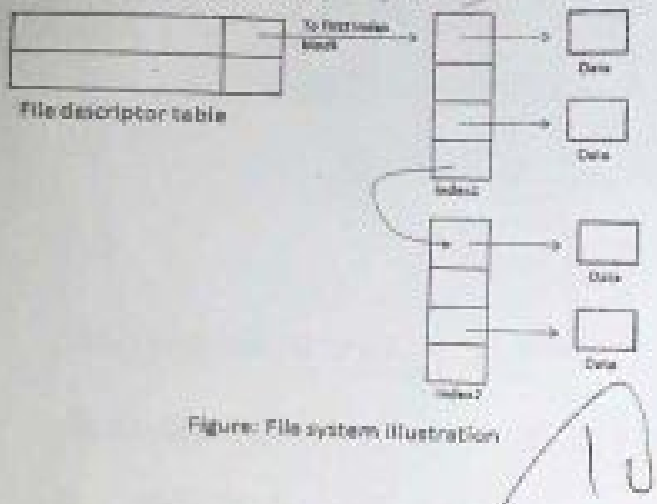


Figure: File system illustration

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.

Duration: 2 hours

Université Virtuelle
Faculté des Sciences I

I3303 / INFO324
Operating System II

Problem I — 18 points

In each of the following programs, it is assumed that the parent process has Pid = 100, and that there are no other active processes in the system than those created by the program. Give all possible display results obtained by running each of the following programs.

Program 1

```
void main(){
      fork();
      if(fork())
          printf("%d\n",fork());
}
```

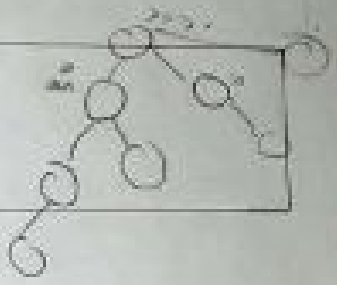
(5)

Program 2

```
void main(){
      int x;
      x = (fork() + fork()) * fork();
      printf("%d\n",x);
}
```

(2)

fork
fork
fork



Program 3

```
void main(){
      int p[2], x = 0, y;
      if(fork()){
            pipe(p);
            x = getpid();
            write(p[1],&x,sizeof(int));
      }
      else{
            read(p[0],&y,sizeof(int));
            printf("%d %d",x,y);
      }
}
```

(5)

Problem II — 16 points

(6) 1. Using the SIGALRM signal, write a program that draws a number between 1 and 100 after one second and displays it.

(10) 2. Write a C program under UNIX where the parent process creates 100 child processes as follows: The father creates the first process and waits for the child to display its PID, before creating the second child. Then he waits for the second to display his PID before creating the third child, and so on.

Problem III — 16 points

1. We consider a memory managed by contiguous allocation. The allocation of processes is made according to the first fit algorithm. That is, the first encountered area whose size is greater than or equal to the size of the process to be loaded is the one that is allocated

process. At the instant t, the state of the main memory is described in the figure below (where A, B, C, D, E, F are processes already loaded in memory, and the other parts are free zones):

Represent the evolution of the main memory according to the arrival of each of the following successive events.

- Event 1. Arrival of program G (20k).
- Event 2. Start of program B.
- Event 3. Arrival of program H (15k).
- Event 4. Departure of the program E.
- Event 5. Arrival of program I (40k).

2. The memory of a computer contains 4-page frames and, at the beginning, all the frames are empty. How many page faults produces the following page references 3, 4, 4,1, 5, 2, 3,1,4 using, respectively, FIFO, OPTIMAL, and LRU replacement algorithms? Justify your answers by showing the contents of the frames after each reference.

Problem IV **20 points**

1. Describe (in pseudocode) the steps to be followed by the function **void Move (char * file_name, int f1, int f2)** which moves a file from a source directory (f1 is the index of its descriptor entry) to a destination directory (f2 is the index of its descriptor entry).

2. Describe (in pseudocode) the steps to be followed by the function **int Similar (int f, char * file_name_1, file_name_2)** which compares the contents of two files belonging to the opened directory with entry descriptor f. If both files have the same content, the function returns 1, and 0 if not.

P.S: Feel free to use the structures and functions seen in the course.

Bon travail