Operating Systems 2

I3303

Introduction

- Chapter 1:
  - Process creation
  - process termination
  - process scheduling
  - process intercommunication (IPC):
    - IPC with pipes
    - IPC with signals:
    - IPC with shared memory:
- Chapter 2:
  - memory management
    - swapping
    - virtual memory
    - segmentation
    - page replacement algorithms
    - Etc…
- Chapter 3:
  - File management system:
    - file system
      - user view
      - system view
      - Unix file system structure
- Chapter 4:
  - I/O device management
- Refresh from OS1
  - Process: program in execution
  - Program: set of instructions
  - Process state:
    - Ready state: process in ready queue (in memory), waiting for the scheduler to choose it
    - Running state: being executed (occupies the CPU)
    - Waiting/Blocking state: I/O request
  - Timings:
    - 1: New → Ready: process is put into ready queue
    - 2: Ready → Running: Scheduler keeps the CPU to the process
    - 3: Running → Ready: The scheduler picks up the process and gives the CPU to another process

- 4: Running → Blocked: Requests for I/O
- 5: Blocked → Ready: I/O are arriving
- 6: Running → End

- Process Implementation:
  - Divided into 4 parts:
    - Code: instructions to execute
      - instructions: I1 → I2 → I3
      - PC: program counter (contains the address of the next instruction to execute)
    - Data: Global & static variables
    - Stack: Contains function calls & local variables
    - Heap: Dynamic allocation of memory
  - To implement a process, the OS creates a table in memory
    - Table of processes: array of structures where each row is a structure
    - This table & process is the PCB: process control block
  - What does the PCB include:
    - pid: id of the process
    - state
    - priority
    - PC value
    - PSW: process status world
      - 2 modes of execution: User mode or Kernel mode
      - it is a bit that saves the mode of execution
    - pointer to code
    - pointer to data
    - stack
    - real user id
    - parent id
    - root directory
    - exit status
    - etc…
  - Context Switching: saving the state of the process (PCB ig)
  - CPU registers (not all):
    - IR: instruction register (Fetch)
    - PC
    - PSW
  - Command to get the programs in execution: ps
- Process Creation:
  - By duplication, except the first process of *pid=1* named *init*
  - Example:
    - P1 process with stack, heap, data, and code
    - P2 is created by duplicating P1's stack, heap, and data except the code
  - How to duplicate:
    - library <unistd.h> containing a function

    - function: int fork()
    - the child starts running the program starting from the fork it was created (exclusive)
    - Return value of the fork informs the current pid of the process
      - if return is -1: failure → no child created
      - if return is 0: I am in the child
      - else it returns the pid of the child and I am in the parent
  - Example:
    - Parent: pid, variables, file descriptor
    - child: pid, copy of variables, another file descriptor that points to the same file
    - However: the position inside the file is shared in both parent & child

11/30/2022

- Process Identification:
  - int getpid() → returns the id of the calling process
  - int getppid() → returns the id of the parent
- Process termination:
  - Normal exit:
    - finish the execution of all instructions
    - call exit()
      - void exit(int status)
        - library: <sys/wait.h>
        - the process calling exit ends its execution
        - status
          - used by the process to inform his parent about the exit status. it is between 0 & 255 (1 byte)
          - value is stored on the left of this byte. So to actually access the real data, we need to shift the status to the right by 8 bits (1 byte)
        - By convention, exit(0) is a normal exit
      - Parent calls fork() → a child is created
        - The parent & the child continue execution
        - the child decided to exit(status), but only exists if the parent knows that the child exited
        - To keep track of the child, we use a function in the parent:
          - wait(&b)
          - b>>8 →shifts the status 8 bits in order to read it
  - Abnormal:
    - It is killed by a privileged process
  - int wait(int *st):
    - function called by the parent to wait for the exit of **one** child
    - this statement is blocking (the parent stops execution until the child exits)
    - it returns the pid of the exited child
  - int waitpid(int pid, int *st, int options):
    - waits for a specific child with pid to exit

- waitpid(-1, … , …) → wait()
- options:
  - 0 → nothing
  - WNOHANG → makes the wait a non-blocking statement

- Macros:
  - int WIFEXITED (int status): returns non-zero value if the child terminated normally with exit
    - Example:
      - *wait(&str)*

        if WIFEXITED(str)

        {

        printf("my child exited normally");

        }
  - int WEXITSTATUS(int status): shifts and returns the status automatically
    - Example:
      - *wait(&str)*

        if WIFEXITED(str)

        {

        printf("my child exited normally with value %d",WEXITSTATUS(str));

        }
  - int WIFSIGNALED(int status)
    - returns a non-zero value if the child terminated abnormally because it received a signal that was not handled
  - int WTERMSIG(int status)
    - informs us of the signal that killed the child
  - int sleep(int sec):
    - the process suspends its execution sec time (blocking statement)
- If the parent dies before its child → child is called an Orphan process
  - The init adopts this child & the ppid of this child becomes 1
- If the child dies and the parent doesn't execute the wait statement → the child is a zombie process
- Command:
  - ps-a pid → this command gives the status of the child with pid.
    - if orphan → O
    - if zombie → Z
    - if normal → 0

- **Exercise**: Write a program that determines the maximum of a vector of integers by distributing the task between a parent process and his child.

5/12/2022

Executing a file

- Shell: a process executed by the system to write commands
- Inside the shell I am writing a command (example ls –l -a):
  - the shell needs to execute its own instructions? How will it execute the ls command?
- Process:
  - shell executes fork() → child has a pid = 101
  - I make the child execute the command ls
- NOTE:
  - void main(int argc, char*argv[])
    - Command line arguments: arguments of the main function. Written during the call to execution
    - Example: ls –l -a
    - argc: number of parameters (including the name of the program)
      - In the example: argc = 3
    - argv: An array of strings containing the parameter values
      - in the example: argv = ["ls", "-l", "-a" , "\0"]
  - What is a path:
    - a variable, indicating the path to reach a specific folder or file
- Execution of the "ls –l –a" command:
  1) main (int argc, char*argv[])
  2) parent executes fork()
  3) inform the child to execute a new image/process (in this case: ls –l –a)
     - The child no longer is a shell, it is a program executing the command
- How to change the main task of the child to another task?
- Family of exec functions
  - Role: replace the current context of a process with new image/process
  - These functions <u>don't return</u>
    - they never return to continue the instructions under it
  - Functions:
    - int execl (char*filename, char* arg0, char* arg1,...)
      - execl (list)
      - filename : should be the full path
    - int execlp (char*filename, char* arg0,char*arg1,...)
      - filename: only the file name (no need for a path)

      Each argv element is a parameter
    - Int execv (char*filename, char*argv[])
      - execv (vector)
      - filename: should be a full path
    - int execvp (char*filename, char*argv[])
      - filename: only the file name (no need for a path)

      argv elements in an array

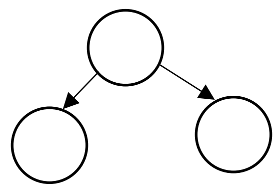
  - Example: write a program to execute the command "ls –l –a"

```
void main(int argc, char* argv[])
{
    int pid;
    pid = fork();
    if(!pid)
    {
        execl("/bin/ls","ls","-l","-a", NULL); //filename is a path
        //execlp("ls","ls","-l","-a",NULL); //filename is just a name
        printf("Never executed\n"); //this process is never executed, since the child
                                     never returns to this process after exec functions
    }
}
```

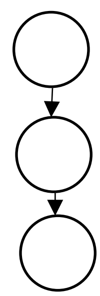
- Return value of the exec:
  - -1 → failure
  - 0 → success
  - This return value is returned to the parent

7/12/2022

Fork & Binary Tree

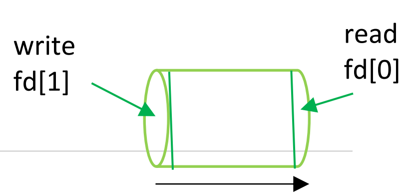


- A && B
  - are 2 boolean expressions
  - if A is false, then B is never executed/evaluated
  - fork && fork:
    - Both forks are done in the parent
- A || B:
  - if A is true → B is never executed
  - fork || fork:
    - 1st is done in the parent, 2nd is done in the child



Interprocess Communication (IPC)

- Methods:
  - Naïve: Wait & exit
    - very limited/inefficient
  - Unix pipes
  - Signals
  - Shared memory segment
  - Message passing
  - Communication by sockets (when processes are on different machines)
- Unix Pipes:
  - ls –l | wc
    - the output of ls –l, is the input of the wc process
  - Definition: A pipe is an interprocess communication tool. It is a FIFO buffer (like a queue)
    - **buffer**: zone in memory for the OS
    - It is a one way communication tool.
    - Size of the pipe is fixed in the OS (hyper-parameter)



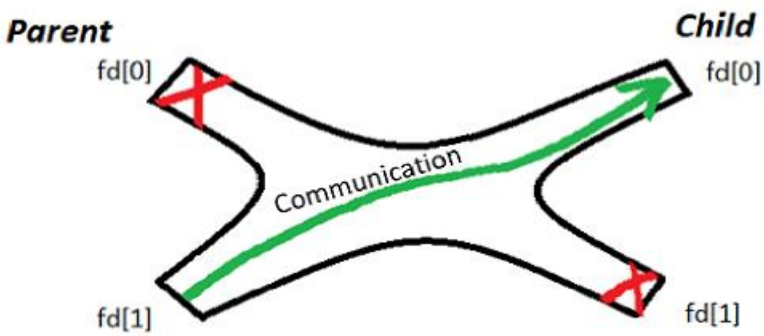
            - usually it is 4KB
    - The pipe has 2 sides for access
        - **By convention**: one side is for writing, & the other side is for reading
    - Pipe has 2 tasks: communication & synchronization
- Pipe creation:
    - we need an array of 2 integers
        - int fd[2];
    - int pipe(fd); → function that creates a pipe
        - fd[0] is the reading side
        - fd[1] is the writing side
        - return value:
            - 0 → success
            - -1 →fail (wasn't created)
- I/O with pipes:
    - In UNIX: Every device is a file
        - Everything is represented as a file that can be opened, read, and edited through a descriptor (just like a file pointer)
    - fd[0] stores the descriptor for reading, fd[1] stores the descriptor for writing
    - int write(fd[1], buf, size)
        - size: number of characters to write
        - buffer: the zone to write the data
        - remove from the buffer size characters and put them in fd[1]
    - int read(fd[0], buf, size):
        - size: number of characters to read
        - buffer: the zone to read the data
        - read from fd[0] size characters and put them in buffer

```
int x[10];
read(fd[0],&x,10*sizeof(int));
```

    - Both return the number of effective characters
- Synchronization with pipes:
    - Unix pipes are used as tools of synchronization between processes, How?
    - Reading from empty pipe is a blocking statement <u>**if there is a writer in other side**</u>
        - pipe is empty while the writer is still not closed. So the pipe waits until the writer writes something in order to read it
    - Writing in full pipe is a blocking statement <u>**if there is a reader in other side**</u>
        - pipe is full while the reader is not closed. pipe waits until the reader reads something so it gives the pipe space to write
- Pipe & fork()
    - If I want to use pipe as a way of communication, I need to create a pipe before forking (so I can communicated between the parent and the child)

- How to connect 2 processes using pipes:
  1) Make the pipe using the pipe() function
  2) fork() to create the child for reading from pipe
  3) **in the child, close the writing side (if child wants to read)**
  4) **in the parent, close the reading side (if parent wants to write)**

```
void main()
{
    int fd[2], pid;
    pipe(fd); //create the pipe --> parent has a pipe with fd[1] (write) & fd[0](read)

    int p = fopen("toto.txt");
    parent --> close(fd[0]); //if the parent wants to write
    child --> close(fd[1]); //if the child wants to read
    /*This is one way of communication, If I want 2 ways of communication between parent &
child, I have to create another pipe with opposite closing channels*/

}
```



- Descriptors:
  - By default, each created process is associated with three file descriptors:
    - 0 → standard input (Keyboard)
    - 1 → standard output (Screen)
    - 2 → standard error (Screen) (for error messages)
  - In Pipes, it is associated with 2 descriptors:
    - fd[0] for reading
    - fd[1] for writing
- **In forking;**
  - the pipe is a shared zone between parent & child
  - However, the child & parent have different descriptors, each pointing to the same location in the file
    - child fd[0] != parent fd[0]
    - but child fd[0] points to the same place in the parent fd[0]

- Example:

```
void main()
{
```

```
    char reception[100]; //for receiving the message
    char *message = "Hello my dear";
    int desc[2],nb;

    pipe(desc);
    if(!fork())
    {
        close(desc[1]);
        nb = read(desc[0],reception,100);
        printf("the child read %d characters --> %s\n",nb,reception);
        close(fd[0]);
    }
    else{
        close(desc[0]);
        write(desc[1],message,strlen(message)+1);
        close(desc[1]);
    }
}
```

12/12/2022

- Steps to execute: ls -l | wc
  - Make a pipe
  - Create a child
  - I want the parent to execute ls -l → parent writes → close fd[0]
  - I want the child to execute wc → child reads → close fd[1]

```
void main(){
    int fd[2];
    pipe(fd);

    if(fork())
    {
        //in parent
        //Missing: how can i send the output to the child?
        execlp("ls","ls","-l",NULL);
    }
    else{
        //in child
        //Missing: how can i read the output?
        execlp("wc","wc",NULL);
    }
}
```

- Redirection of I/O:
  - Each process has by default an associated table called "file descriptor table".
  - File descriptor table contains rows, each containing pointers to the file objects which do all the resource handling.

- All file descriptors have, by default, the values:

0 (keyboard)
1 (screen)
2 (screen for error)

- Each time a file opens, a descriptor is included in the table
  - Int p = open("toto.txt")
  - A file descriptor of index (for example) 100 is added to the file descriptor table

0 (keyboard)
1 (screen)
2 (screen for error)
100

- When a pipe is created, 2 descriptors fd[1] and fd[2] are added to the file descriptor table

0 (keyboard)
1 (screen)
2 (screen for error)
fd[0] (read)
fd[1] (write)

- By default:
  - Ls -l writes to descriptor 1,
  - wc reads from descriptor 0.
- However, the only communication between the parent and the child is the pipe. So **I want to make ls -l write to fd[1] and wc to read form descriptor fd[0]**
- To execute ls -l |wc:
  - I should redirect the output of the process "ls" from screen to the writing side of the pipe
  - We should redirect the input of the process "wc" from the keyboard to the reading side of the pipe

- How to redirect the I/O:
  - **The 2 duplicate functions:**
    - int dup(int newfd)

- it function duplicates the content of a file descriptor, given as a parameter, in the first **free** cell of the file descriptor table
- So to put the fd[1] instead of 1 in the ls -l, I have to free 1.
- How to free a cell:
  - **Close(1);**
- **Then I do: dup(fd[1]);**
- Now there exists 2 fd[1] in the file descriptor table → close(fd[1]);

▪ Int dup2(int newfd,int oldfd):
  - Automatically does the first close for me
  - Dup2(fd[1],1); == close(1);dup(fd[1]);

```
void main(){
    int fd[2];
    pipe(fd);

    if(fork())
    {
        //in parent
        close(fd[0]);
        dup2(fd[1],0);
        close(fd[1]);
        execlp("ls","ls","-l",NULL);
    }
    else{
        //in child
        close(fd[1]);
        dup2(fd[0],0);
        close(fd[0]);
        execlp("wc","wc",NULL);
    }
}
```

- **All the above where called UNNAMED pipes:**
  - They act like any variable (deleted after program ends)
- Named pipes:
  - It's like a file, has its own existence. Other processes can access it later on
  - A named pipe or "FIFO Special file" is similar to pipe but instead of being an anonymous, temporary connection, it has a name like any other file and permanent existence.
  - The process opens the FIFO by its name in order to communicate through it
  - The named pipe has a capacity of 40Kb
  - The named pipe **can be used by independent processes other than the one that created it**

- Creation of FIFO:
  - Int mkfifo(char *filename, mode-t mode)
    - Mode can be: 0666, 0777
    - returns:

        - -1 → failure
        - 0 → success
    - Create FIFO in terminal:
        - mkfifo *canal* → creates a named pipe named canal
- Using FIFO:
    - Open FIFO:
        - Int fd = open("*canal*",...);

14/12/2022

- Monday lab → bring a laptop
- Named pipe example (discussed in previous lecture):
    - writer.c

```
void main(){
    int n;
    char Buffer[100];

    int fd_write;

    mkfifo("pipe1",0777);

    fd.write = open("pipe1",O_WRONLY); //Open for writing only
    write(fd_write,"Bonjour",7);
    close(fd_write);
}
```

    - reader.c

```
void main(){
    //in reader, i don't create the pipe since the pipe was created in writer.c
    int n;
    char buffer[100];
    int fd_read;
    fd_read = open("pipe1",O_RDONLY); //Open for reading only
    read(fd_read,buffer,100);
    buffer[100] = '\0';
    printf("%s\n",buffer);
    close(fd_read);
}
```

- Exercise: What would happen in the following programs? (Previous exam question)
    - **Program 1:**

```
void main(){
    int i,j=1,p[2];
```

```
    pipe(p);
    write(p[1],&j,sizeof(int));
    for(i=1;i<5;i++){
        if(!fork()){
            close(p[1]);
            break;
        }
    }
    read(f[0],&j,sizeof(int));
    printf("%d\n",i);
}
```

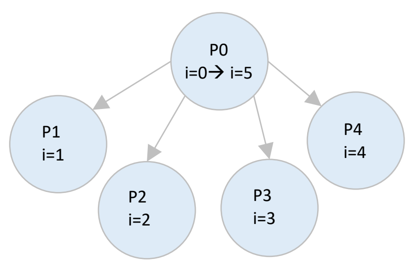
- **Answer:**

1) Pipe is created →

Parent file descriptor
0
1
2
p[0]
p[1]

2) The parent writes to the pipe the value of j=1.
3) For loop:
   a. Creation of the child
   b. If I am in the child → close its writing side p[1]

Child file descriptor
0
1
2
p[0]
~~p[1]~~



4) Here reading depends on the order of execution
   a. If one of the children finished first:
      i. Child reads 1 & removes it from the pipe & prints its i
      ii. For all the others: Reading from an empty pipe but we have one of the writer sides open (parent fd[1] is still open) →blocking statement
   b. If the parent P0 finishes firs:
      i. Parent reads 1 & removes it from the pipe & prints its i=5
      ii. Parent ends its program → all its file descriptors are closed (including its fd[1])
      iii. Children: reading from an empty pipe, but all the writer fd[1] are closed → not a blocking statement → all of them print their i values & exit.

- **Program 2:**

```
void main(){
    int i,j=1,p[2];
    pipe(p);
    for(int i=1,i<5;i++){
```

```
        if(!fork()){
            close(p[1]);
            break;
        }
    }
    wait(NULL);
    read(p[0],&j,sizeof(int));
    printf("%d\n",i);
}
```

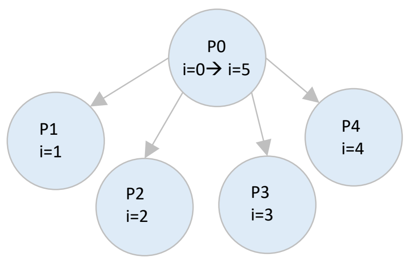
- **Answer:**

1) Pipe is created →

Parent file descriptor
0
1
2
p[0]
p[1]

2) For loop:
    a. Creation of the child
    b. If I am in the child → close its writing side p[1]

Child file descriptor
0
1
2
p[0]
~~p[1]~~



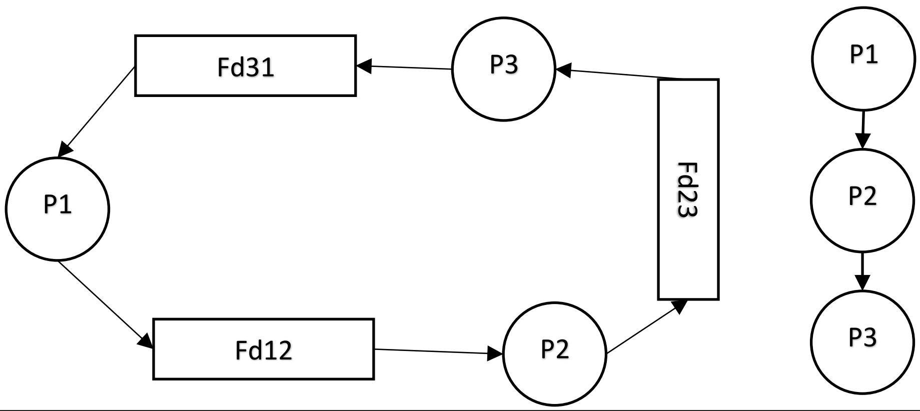
3) Parent waits for **one** of the children to finish
4) We go into one of the children:
    a. Child reads from an empty pipe (we didn't write to the pipe) and p[1] of the parent open → blocking statement → no one exits

- **Program 3:** (HOMEWORK)

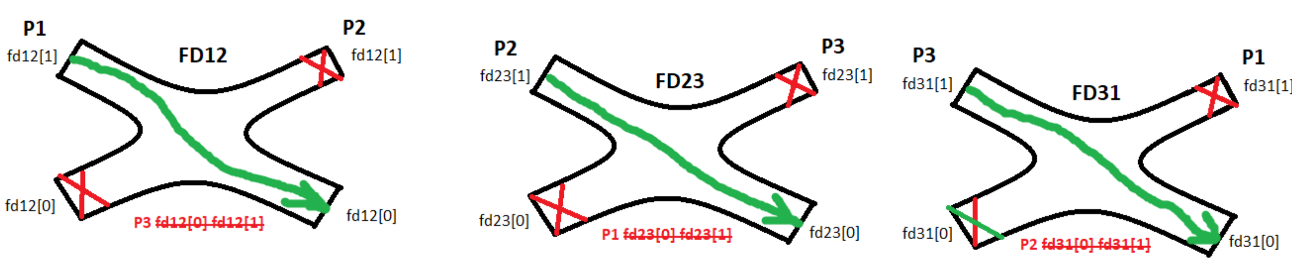
```
void main(){
    int i,j=1,p[2];
    pipe(p);
    for(int i=0;i<5;i++){
        if(!fork()){
            close(p[1]);
            break;
        }
    }
    write(p[1],&j,sizeof(int));
    printf("%d\n",i);
    read(p[0],&j,sizeof(int));
}
```

19/12/2022

- Exercise 1:




```
void main(){
    int fd12[2],fd23[2],fd31[2];
    pipe(fd12);
    pipe(fd23);
    pipe(fd31);

    if(fork()) //I am in P1
    {
        close(fd31[1]);
        close(fd12[0]);
        close(fd23[1]);
        close(fd23[0]);
    }
    else if(fork()) //P2
    {
        close(fd12[1]);
        close(fd23[0]);
        close(fd31[0]);
        close(fd31[1]);
    }
    else{
        //P3
        close(fd12[0]);
        close(fd12[1]);
        close(fd23[1]);
        close(fd31[0]);
    }
}
```

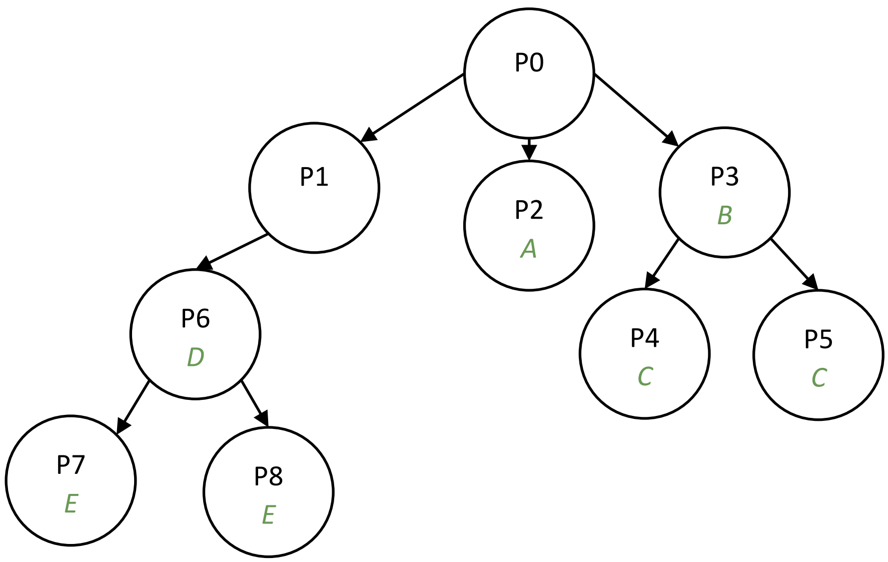


- 2014 partial: Draw the graph

```
int main(){
```

```
    if(fork())
    {
        fork() && (fork() || (fork() && fork()));
        //A         B               C
    }
    else fork() || (fork() && fork());
    //     D                E
}
```

- Answer:



- Exercise:
  - Parent creates 10 child processes
  - Parent reads variable (N) from stdin and write 'm' N times in a pipe
  - The 10 children start a race to read from the pipe
  - Each child must send to the parent the number of characters 'm' he reads
  - Finally, the parent must announce the winner (child with maximum m)
  - Steps:
    - Parent:
      - Create the pipe
      - Read N
      - Create the 10 children
      - Write N 'm' in pipe
      - Determining the max
      - Display the winner
    - Child:
      - Start needing from pipe until no data exist
      - Send the counter to parent
      - Done

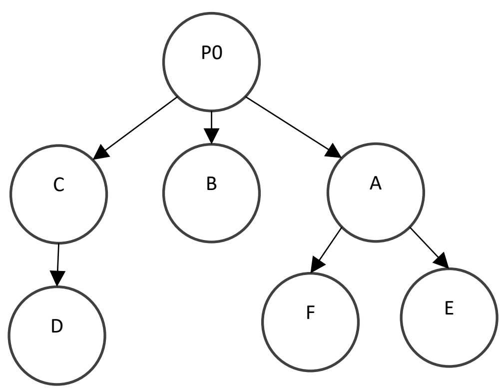
- Answer:

```
int main(){
```

```
    int Pid[10],status[10],N,parent_pid=getpid(),max,winner;
    int fd[2];
    char c = 'm';
    pipe(fd);
    close(fd[0]); //close reading side of father

    //write N 'm' in pipe
    for(int i=0;i<N;i++)
    {
        write(fd[1],&c,sizeof(char));
    }
    //close writing side of parent
    close(fd[1]);

    //create 10 children
    for(int i=0;i<10;i++)
    {
        if(!(pid[i]=fork())){break;}
    }

    if(getpid() == parent_pid) //in parent
    {
        for(int i=0;i<10;i++){waitpid(pid[i],&status[i],NULL);}

        max = WEXITSTATUS(status[0]);
        for(int i=0;i<10;i++)
        {
            if(max < WEXITSTATUS[status[i]])
            {
                max=WEXITSTATUS(status[i]);
                winner = pid[i];
            }
        }
    }
    else{ //in child
        sleep(2);
        close(fd[1]);
        while(read(fd[0],&c,sizeof(char)) != 0){count++;}
        exit(count);
    }
}
//the schedular handles the race between the pipes (round robin,FIFO....)
// I don't have to write in my code that another child has to come and end another --> it all depends on the machine & the schedular
```

21/12/2022

- Exercise:
  - Write the code of this using one statement:
    - One if-else
    - No loops

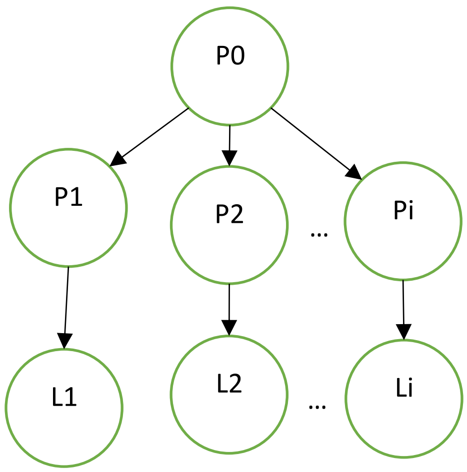


  - Answer:

```
int main(){
```

```
    if(fork()) //A
        fork() && (fork() || fork())
        //B         C         D
    else fork() && fork()
        //E         F
}
```

- Exercise:
    - Write the code of this with a condition
        - The parent doesn't create p2 before making sure that p1 created L1



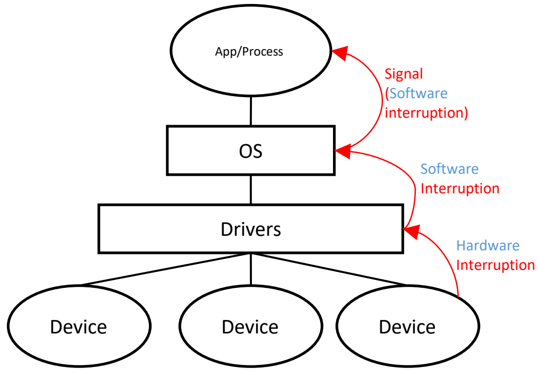
- **Answer** (kind of):

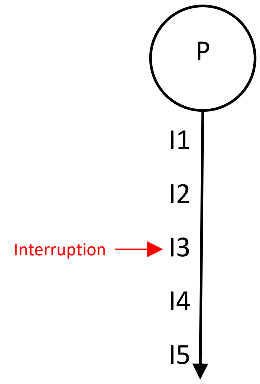
```
int main(){
    int n=10,pid,s;
    for(int i=0;i<n;i++)
    {
        if(pid=fork()) //parent
            waitpid(pid,&s,NULL);
        else { //child
            fork(); //create child of child
            break;
        }
    }
}
```

- **If I want all the children alive**: I can add an infinite while loop so that the children don't exit.

4/1/2023

- Interruption
  - 2 types: Hardware & Software





  - Interruption vector
    - zone in **memory related to the OS.**
    - It is an array of 2 fields
      - index: identifies the interruption
      - address of a handler: a function that handles this interruption

Index	@ Of handler

    - Context switching: save the state of the process (everything inside the registers, variables..) → go to handler → perform the function → resume process
    - **Example**: executing I3 → interruption occurred → go to interruption vector to get address of handler → perform function → resume the process
- Signal (Software Interruption)
  - **Definition**: a signal is a "software interruption" **delivered to a process**
  - Different sources:
    - Internal event (divide by zero, fault segmentation, Floating Point Error)
    - External event (Ctrl+C, Ctrl+Z)
    - Explicit request (I/O request)
  - A signal reports the occurrence of an exceptional event
    - Examples:
      - Program errors: such das dividing by zero
      - User request to interrupt the process: Ctrl+C, Ctrl+Z
      - The termination of a child process (exit)
      - Expiration of a timer
      - A call to "kill" or "raise" command
        - Kill → process sends a **signal to another process**
        - Raise →process sends a **signal to itself**
      - Reading from an empty pipe with no writer
  - Each signal has an id or a macro
    - Library: **<signal.h>**
    - List of signals command: **Kill -l**

      - id) name (YOU DON'T HAVE TO KNOW ALL OF THEM)

1) SIGHUP	2) SIGINT	**3) SIGQUIT**	4) SIGILL

5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) SIGKILL	**10) SIGUSR1**	11) SIGSEGV	**12) SIGUSR2**
13) SIGPIPE	**14) SIGALRM**	15) SIGTERM	16) SIGSTKFLT
17) SIGCHLD	**18) SIGCONT**	**19) SIGSTOP**	20) SIGTSTP
21) SIGTTIN	22) SIGTTOU	23) SIGURG	24) SIGXCPU
25) SIGXFSZ	26) SIGVTALRM	27) SIGPROF	28) SIGWINCH
29) SIGIO	30) SIGPWR	31) SIGSYS	34) SIGRTMIN
35) SIGRTMIN+1	36) SIGRTMIN+2	37) SIGRTMIN+3	38) SIGRTMIN+4
39) SIGRTMIN+5	40) SIGRTMIN+6	41) SIGRTMIN+7	42) SIGRTMIN+8
43) SIGRTMIN+9	44) SIGRTMIN+10	45) SIGRTMIN+11	46) SIGRTMIN+12
47) SIGRTMIN+13	48) SIGRTMIN+14	49) SIGRTMIN+15	50) SIGRTMAX-14
51) SIGRTMAX-13	52) SIGRTMAX-12	53) SIGRTMAX-11	54) SIGRTMAX-10
55) SIGRTMAX-9	56) SIGRTMAX-8	57) SIGRTMAX-7	58) SIGRTMAX-6
59) SIGRTMAX-5	60) SIGRTMAX-4	61) SIGRTMAX-3	62) SIGRTMAX-2
63) SIGRTMAX-1	64) SIGRTMAX		

- Send a signal command
  - **Kill -sigid pid**
    - Sigid → signal id
    - Pid → the process where I want to send the signal
  - **Example:** kill -9 101 (sends signal SIGKILL to process with pid=101)
- Receiving signals (3 options)
  - Ignore the signal if the signal can be ignored (like SIGCHLD, SIGUSR1…)
  - Do default action (action of the signal)
    - For most signals it is to kill the process, that's why we handle them
  - Handle the signal if it can be handled
    - Similar to the handler in the OS, but here it is related to the process & we write how to handle it
- 2 signals that I cannot ignore, and cannot handle:
  - SIGKILL (9) → kill/end the process
  - SIGSTOP (19) →pause the process not kill it (Ctrl+Z)
    - Resume the process: SIGCONT (18)

- Signal Handling:
  - Handle the signal (instead of letting the default action take place)
  - Handling function
    - Signal(sigID, handler)
      - Handler is a **function I write** (it can be empty or do any other thing)
      - **the default action of the signal doesn't take place**

```
signal(SIGFPE,myhandler);
void myhandler(int sig){ /*Anything Or Nothing*/ }
```

  - Signal function implementation

```
//how signal is written
typedef void (*sighandler_t) (int);
sighandler_t signal(int SIGNUM,sighandler_t action);
```

    - Action:
      - SIG_DFL (default) / SIG_IGN (ignore) / Handler function
    - returns the previous value of signal handler (the first return value is the default action)
- Handler:

- It takes an integer as argument. This integer is the code of the signal that triggered this function
- It is important if I want to handle different signals within the same handler function

- Example:

```
//Global
int counter = 0;
void myhandler(int sig){counter++;}

int main(){
    int i;
    signal(SIGCHLD, myhandler); //SIGCHLD --> signal sent by the OS when a child is created
    for(i=0;i<5;i++)
    {
        if(!fork()) exit(0);
        while(wait(NULL) != -1);
    }
    printf("Counter = %d\n",counter); //counter is 5
}
```

9/1/2023

- Sending signals:
  - **To another process:** int kill(int pid, int sig)
    - Pid: pid of the process to send the signal
    - Sig: signal id or name
  - **To itself:** int raise(int sig)
- Waiting signal:
  - Int pause()
    - **Blocking** function
    - It stops the current process from work
    - Any other signal wakes the process
  - int alarm(int sec):
    - sec → **waits sec times** before sending a **SIGALRM signal**
    - **doesn't block**
    - if I want to block → add pause() while handling the SIGALRM***************
    - The first alarm to execute removes all the previous non-executed alarms

```
alarm(5);
alarm(3);
//since alarm(3) executes before alarm(5), alarm(5) will be canceled by alarm(3)

alarm(0); //cancel all alarms
```

- Signals SIGUSR1 & SIGUSR2 are only for communication. (no specific purpose)

- **Exercise 1:** Write a program that takes an input from the user. If the user doesn't input a value after 20sec, the program should print "You haven't entered a value yet" & ends the program.

```c
void inputalarm(int signal)
{
    printf("\nYou haven't entered a value! (20s)");
    exit(); // stop waiting for input
}

void main()
{
    int x;
    signal(SIGALRM, inputalarm);
    printf("Enter a number: ");
    alarm(20);
    scanf("%d", &x);
    alarm(0); // cancel the alarm if the user enters a value
}
```

- **Exercise 2:** The SIGCHLD signal is sent to the parent when a child ends. Write the necessary code for a parent process that doesn't wait its child in blocking mode but the child won't become a zombie process (the parent has to wait when the child exits not before) (using signals)

```c
void main()
{
    signal(SIGCHLD, zombiehandler);
    if (!fork())
    {
        sleep(3);
        exit(0);
    }
}
void zombiehandler(int signal){ wait(NULL);}
```

- **Exercise 3:** Parent that creates one child. The parent should print odd numbers between 1 and 100, while the child should print the even numbers between 1 & 100. The **numbers should be printed in order.**

```c
void handler(int sig){}
int main(void){
    int child_pid;
    int parent_pid = getpid();
    signal(SIGUSR1,handler);

    if(child_pid = fork()){
      for(int number = 1; number <= 10; number += 2){
        printf("Parent: %d\n",number);
        kill(child_pid,SIGUSR1);
        pause();
      }
    }
    else{
      for(int number = 2; number <= 10; number += 2){
        pause(); // waiting for any type of signal
        printf("Child: %d\n",number);
        kill(parent_pid,SIGUSR1);
      }
    }
}
/*
Scenario:parent forks--> child in pause --> parent prints 1 & sends signal to child--> child awake, prints 2, sends
signal to child, then pauses --> parent wakes, prints 3, sends signal to child, pauses --> child wakes, prints 4,
send signal to parent, pause-->parent wakes.... */

Note: for a reason idk yet, the program doesn't work without \n in the printing
```

- **Exercise 4:** Write a program that displays on the screen alternating between tick & tock every 1 sec

```
int i=1;
void handler(int signal)
{
    printf("%s\n",i%2?"tick":"tock");
    i++;
}
void main(){
    signal(SIGALRM,handler);
    while(1){
        alarm(1);
        pause();
    }
}
```

- **Exercise 5:** Write a program that counts the signals it receives and displays the number of times each signal is received

```
int nsig[NSIG]; //NSIG --> number of signals present in the library <signal.h>

void handler(int signal){
    printf("Signal %d received %d times\n",signal,++nsig[signal]);
}
void main(){
    int s;
    for(s=1;s<NSIG;s++)
    {
        signal(s,handler);
        nsig[s] = 0;
    }

    while(1){
        pause();
    }
}
```

<u>11/1/2023</u>

- **Exercise 1:** parent that creates N children, all paused. The parent sends for each child: SIGCONT, waits for 1s, send SIGSTOP until all children stop;

```
int i=0;
void Conthandler(int sig)
{
    printf("Child %i continued\n",i);
}
void main(){
    int pid[CHILDCOUNT];
    signal(SIGCONT,Conthandler);
    for(i=0;i<CHILDCOUNT;i++)
    {
        if(!(pid[i]=fork()))
        {
            pause();
            pause();
        }
    }

    if(i==CHILDCOUNT)
    {
        while(1){
            for(i=0;i<CHILDCOUNT;i++)
```

```
            {
                kill(pid[i],SIGCONT);
                sleep(1);
                kill(pid[i],SIGSTOP);
            }
        }
    }
}
```

- **Exercise 2:** Ctrl+C is handled for 5 times, then it kills the program

```
int counter = 0;
void handler(int sig)
{
    counter++;
    printf("Ctrl+C handled\n");
}
void main(){
    signal(SIGINT,handler);
    while(counter<5);
    signal(SIGINT,SIG_DFL);
    pause();
}
```

- **Exercise 3**: Parent creates a child, waits for the child to print its pid, then creates the next child.

```
void handler(int sig){}
void main(){
    signal(SIGUSR1,handler);
    for(int i=0;i<CHILDCOUNT;i++)
    {
        if(fork())
        {
            pause();
        }
        else
        {
            printf("Child pid = %d",getpid());
            kill(getppid(),SIGUSR1);
            break;
        }
    }
}
```

- **Exercise 4:** Parent creates children & waits for 5 seconds & then sends for the first child a signal & pauses. Child 1 wakes, child 1 sends signal to child 2, child 2 wakes, child 2 sends signal to child 3…. And so on until the last child wakes the parent. Use alarm. **Solution on page 32**

<u>16/1/2023</u>

Unix Shared Memory

- Every process has its own address space.
- A shared memory segment is a zone that doesn't have pipe constrains. It is shared by the processes that want to communicate with each other.
- Created by one process (known as server) & attached it to its address space. This zone has an id.
- For the client (other process) to add it to its address space, it calls the zone using its id
    - this becomes the shared zone

- Problem: race conditions can occur

- Procedure for using shared memory:
  - Find a **key**. Unix uses this key for identifying shared memory segments
  - shmget(): to **allocate** a shared memory
  - shmat(): **attach** to shared memory space
  - shmdt(): **detach** shared memory space
  - shmctl(): **deallocate** shared memory
- Libraries to include:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

- Keys:
  - of type key_t
  - global entities
  - Do it yourself:

```
key_t somekey = 1234;
```

  - Using Ftok():

```
// ftok(char* path, int ID)
key_t somekey = ftok("./",'a');
```

  - Using IPC_PRIVATE:
    - System provides it
- Allocate shared memory: shmget()

```
int shm_id = shmget(
    key_t KEY,  //identity key
    int size,   //size of the shared memory
    int flag    //creation or use
);


key_t somekey = ftok("./",'a');
shmi_id = shmget(
    somekey,         //key based on the current directory
    4*sizeof(int),  //array of 4 integers
    IPC_CREAT | 0666    //creation | permits read & write
);
```

  - Process that is creating the memory: flag = IPC_CREAT | 0666
  - Process that is accessing the memory: flag= 0666
- **Now, shared memory is allocated but not part of the address space**

- Attaching shared memory: shmat()
  - For both server & client

- Returns a pointer to the shared space

```
int * shm_ptr = (int*) shmat(
    int shm_id; //return value of shmget()
    NULL,
    0
);
```

- You need to cast the return value according to memory content
  - In the above example: the memory contains an **array of 4 integers → shm_ptr is an int ***
  - Use shm_ptr to add, delete, & modify the shared data

- Detach shared memory: shmdt()

```
shmdt(shm_ptr);
```

  - The shared space is still there, but not attached to the memory
  - You can reattach to it using shmat()

- Remove shared memory: shmctl()

```
shmctl(
    shm_id,     //id returned from shmget()
    IPC_RMID,   //IPC remove ID
    NULL
);
```

- If a parent wants to communicate with his child: If parent **created & attached the memory before fork()**
  - Can use IPC_PRIVATE when creating shared memory
  - no **need for shmget or shmat() in child**
  - Because the child inherits this shared memory from the parent
  - Example:

```
void Child(int data[])
{
    printf("%d %d %d %d\n",data[0],data[1],data[2],data[3]);
}

void main(int argc,char* argv[])
{
    int shm_id, *shm_ptr, status;
    pid_t pid;

    shm_id = shmget(IPC_PRIVATE,4*sizeof(int),  IPC_CREAT | 0666);
    shm_ptr = (int*) shmat(shm_id,NULL,0);
    shm_ptr[0] = 1;
    shm_ptr[1] = 2;
    shm_ptr[2] = 3;
    shm_ptr[3] = 4;

    if((pid=fork()) == 0)
    {
        Child(shm_ptr);
        exit(0);
    }

    wait(&status);
    shmdt((void*)shm_ptr);
    shmctl(shm_id,IPC_RMID,NULL);
    exit(0);
}
```

- Communicate between 2 separate processes:
  - Need a key, shmget(), shmat(), & shmdt() for both processes

- Need only 1 shmctl()
- Key should use ftok() with the same parameters
- Server must run first to prepare the shared memory
- Example:
  - This version uses busy waiting

```
#define NOT_READY -1
#define FILLED 0
#define TAKEN 1

struct memory{
    int status;
    int data[4];
};

//Server
void main(int argc,char*argv[])
{
    key_t shm_key;
    int shm_id;
    struct memory * shm_ptr;

    //prepare shared memory;
    shm_key = ftok("./",'x');
    shm_id = shmget(shm_key,sizeof(struct memory), IPC_CREAT | 0666);
    shm_ptr = (struct memory*) shmat(shm_id,NULL,0);

    //Fill memory
    shm_ptr->status = NOT_READY;
    for(int i=0;i<4;i++) shm_ptr->data[i] = i;
    shm_ptr->status = FILLED;

    //Wait for child to be dont
    while(shm_ptr->status != TAKEN) sleep(1);

    //detach & remove
    shmdt((void*)shm_ptr);
    shmctl(shm_id,IPC_RMID,NULL);
    exit(0);
}


//Client
void main(int argc,char*argv[])
{
    key_t shm_key;
    int shm_id;
    struct memory * shm_ptr;

    //Prepare shared memory
    shm_key = ftok("./",'x');
    shm_id = shmget(shm_key,sizeof(struct memory), IPC_CREAT | 0666);
    shm_ptr = (struct memory*) shmat(shm_id,NULL,0);

    //Wait for parent to fill
    while(shm_ptr->status != FILLED);

    //print
    for(int i=0;i<4;i++) printf("%d ",shm_ptr->data[i]);
    shm_ptr->status = TAKEN;

    //detach
    shmdt((void*)shm_ptr);
    exit(0);
}
```

- Note: If you didn't remove your shared memory segments using shmctl(), they will **be in the system forever** → degrade system performance
  - Ipcs command: check if you have left shared memory segments
  - Ipcrm: remove shared memory segments

- Exercise: write a program where 2 processes communicate using shared memory. P1 fills memory with n integers. P2 calculates the average of these values. P1 prints this value.

```
void main(){

    int i,pid, seg;
    int N = 3;

    //creating & attaching the shared zone
    key_t Key1 = ftok("./",'a');
    seg = shmget(Key1, sizeof(int)*(N+1),IPC_CREAT | 0666);
    int * data = (int*) shmat(seg,NULL,0);

    //child calculates average
    if( (pid = fork()) == 0)
    {
        while(data[N-1] == 0) sleep(3); //child needs to wait for the parent to fill the array
        printf("Child: calculating average!\n");
        data[N] = 0;
        for(int i=0;i<N;i++) data[N] += data[i];
        data[N] /= N;
        exit(1);
    }
    else{
        srand(getpid()); //for randmozing the seed
        printf("Parent: filling array!\n");
        for(int i=0;i<N;i++)
        {
            sleep(rand()%3);
            data[i] = rand()%1000;
        }

        for(int i=0;i<N;i++) printf("%i ",data[i]);
        printf("\n");
        wait(NULL);

        printf("Parent: average is %d\n",data[N]);
        shmdt(0);
        shmctl(seg,IPC_RMID,0);
        printf("END\n");
    }
}
```

18/1/2023

- **Previous exercise 4:** Parent creates children & waits for 5 seconds & then sends for the first child a signal & pauses. Child 1 wakes, child 1 sends signal to child 2, child 2 wakes, child 2 sends signal to child 3…. And so on until the last child wakes the parent. Use alarm.
  - Answer:

```
//Exercise 4
int pid,fd[2],ctr,prid_main;
void handler(int signal){
    ctr++;
    if(signal==SIGALRM)
    {
        read(fd[0],&pid,sizeof(int));
        printf("Parent woke & sent signal to child with pid: %d\n",pid);
        kill(pid,SIGUSR1);
        pause();
    }
    else if(signal == SIGUSR2)
    {
        printf("Parent %d exited\n",getpid());
        exit(0);
    }
    else
    {
```

```c
        if(read(fd[0],&pid,sizeof(int)))
        {
            printf("Child %d sent signal to child %d\n",getpid(),pid);
            kill(pid,SIGUSR1);
        }
        else //last child
        {
            printf("Last child %d sent signal to parent %d\n",getpid(),getppid());
            kill(getppid(),SIGUSR2);
        }
        exit(0);
    }
}

void main(){
    int N=5;
    pipe(fd);
    signal(SIGUSR1,handler);
    signal(SIGUSR2,handler);
    signal(SIGALRM,handler);

    for(int i=1;i<N;i++)
    {
        if(!(pid=fork()))
        {
            close(fd[1]);
            pause();
        }
        else
        {
            printf("Process %d created child with pid: %d\n",getpid(),pid);
            write(fd[1],&pid,sizeof(int));
        }
    }

    close(fd[1]);
    alarm(5);
    pause();
}
```

Memory Management

- Processes are loaded in memory
- Logical address vs physical address:
  - **Logical**: **virtual address** that a process generates for accessing memory. They are usually expressed in terms of the size of the word in the computer's memory. Operating systems use logical addresses to map it to a physical address.
  - **Physical** address: a **memory** address that refers to a specific location in the physical memory.
- Questions we ask:
  - Where are processes loaded?
  - Do we load all the programs into memory or part of programs?
  - What if the size of the program > memory size?
  - How to store several processes whose total size > memory size?
  - How to manage free spaces & used spaces in memory?
  - How to protect the address space of the process in memory?
  - How to share data between processes in memory?

- Definition: the act of managing computer memory
  - Providing ways to allocate portions of memory to programs at their request, and freeing it when no longer needed.
  - It is a linear array of bytes where each byte is named by a unique memory address

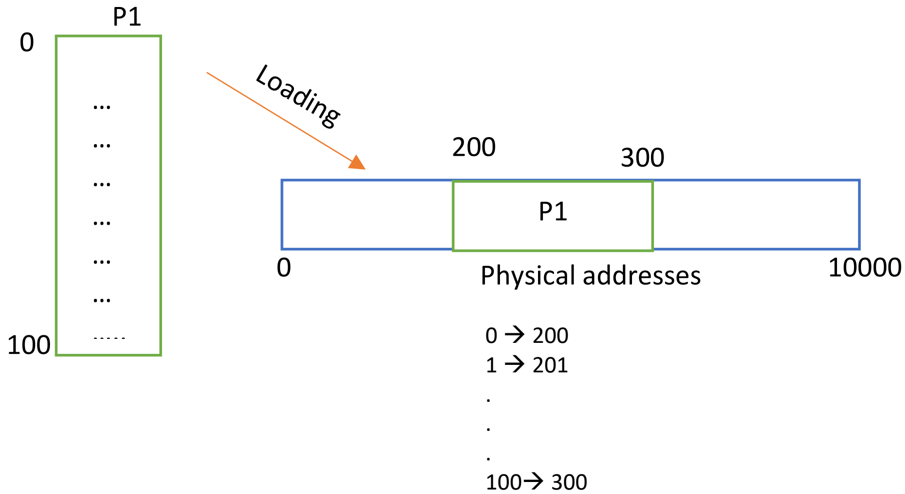
1	2	3	4	5	

- Recall: processes are defined by an address space (the addresses that are accessible by the process (code, data, heap, & stack))
  - We cannot know where a program will be loaded ahead of time
- Address Binding:
  - Definition: fixing physical address to logical address of a process' address space.
  - Types:
    - Compile time binding: if program location is fixed & known ahead of time
      - We identify the address during compilation

What we'll focus

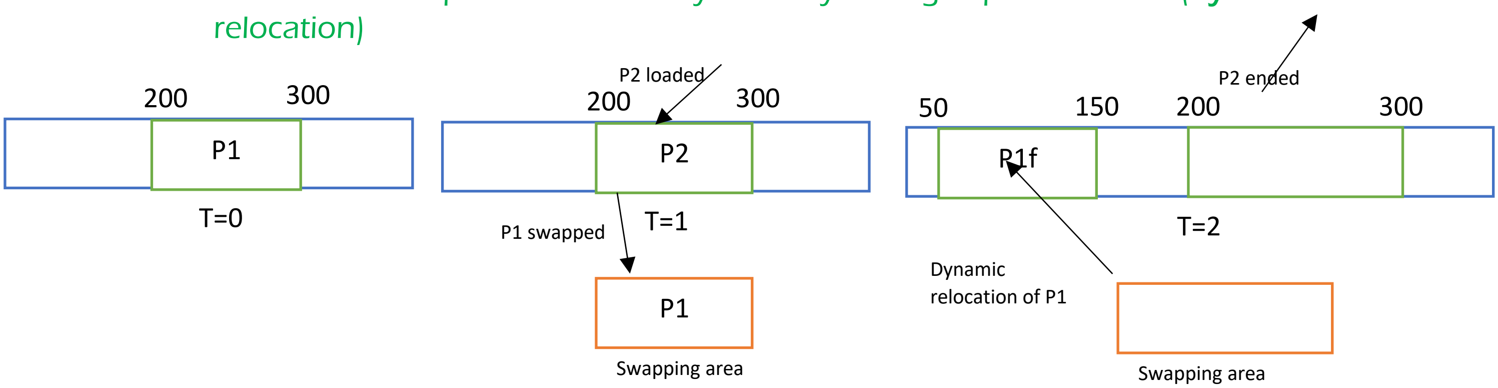
    - Load time binding: program location in memory is unknown until run-time and location is fixed
    - Execution time binding: process can be moved in memory during execution
      - Requires hardware support!
- **History** of Memory Management Schemes:
  - No memory abstraction: address binding
    - Only one program can be run
    - Physical memory is allocated from 0 to some max
    - Running several programs at the same time requires swapping
      - Swapping: there exists a swap area in the hard disk where processes are swapped to be executed.
  - Memory Abstraction:
    - Each process has its own address space
    - During compilation, we give the process logical addresses from 0 to max.
    - Physical address = base + logical address **(relocation)**
      - Base register stores the base value
      - Limit register stores the limit/max value

- The physical address (base + logical address) should be < limit **(protection)**
- In this case: physical address = 200 + logical address

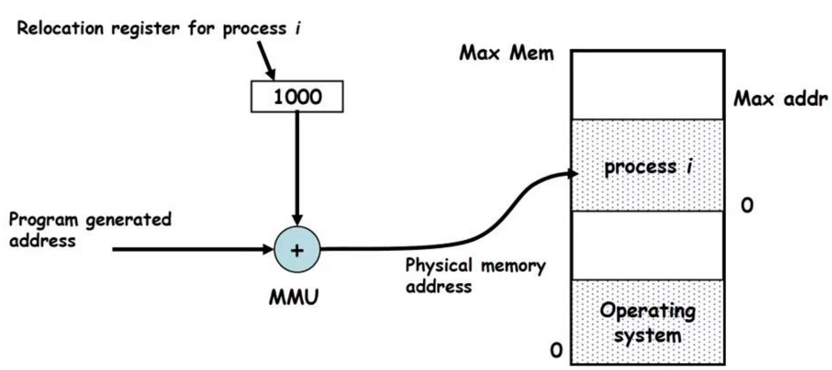


- Logical addresses are from 0 to 100
- Base register: 200
- Limit register: 300

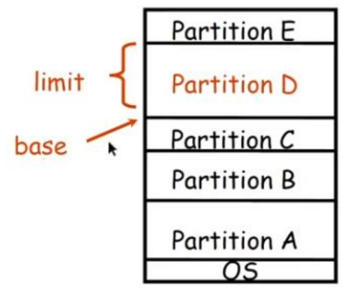
- Each process has its base register & its limit register.
- The location of a process in memory can vary during implementation (**dynamic** relocation)



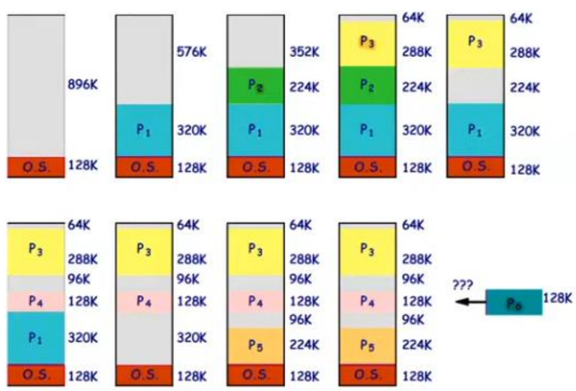
- Dynamic **Relocation**
  - Memory Management Unit (MMU) is the hardware device that converts logical to physical addresses based on the schema



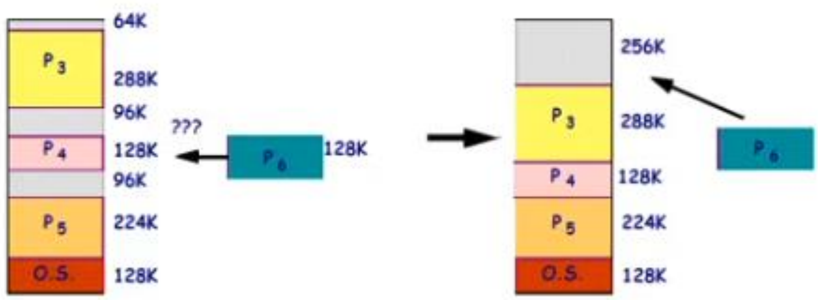
- Multiprogramming:
  - Many processing being executed in memory
  - When a program is running, the **entire program must be in memory**
  - Memory is divided into partitions
  - Each program is put into a single partition
- Swapping:
  - Occurs between processes to allow multiple programs to be executed at once
  - Processes are swapped in & out of memory
- Fragmentation:
  - Occurs when swapping is made
  - Free spaces (holes) in memory distributed in the memory. Each hole is not sufficient to hold an entire process ➔ lost space & degradation of performance.
  - 2 types of fragmentation:
    - External fragmentation: when the size of the partitions are variable according to the processes' size
    - Internal fragmentation: when I divide the continuous layout into partitions of fixed size (explained later)



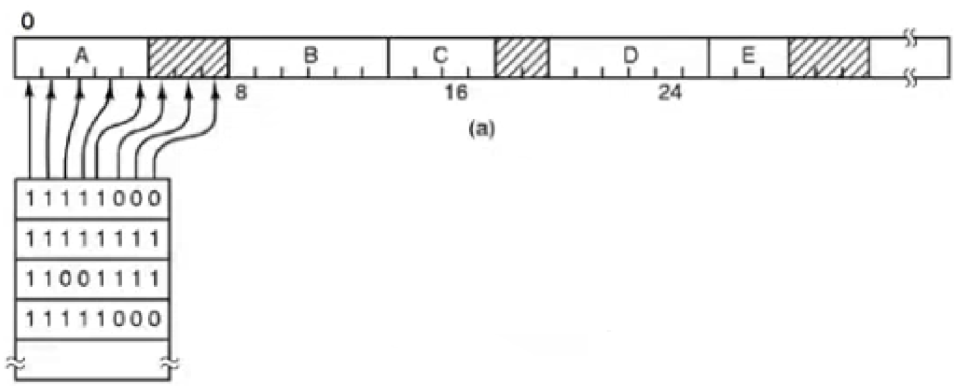
  - Example:



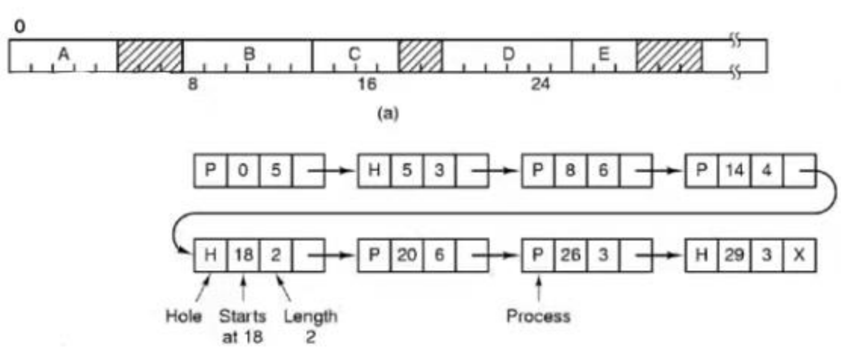
    - In here we have external fragmentation
- How to deal with external fragmentation:
  - Compaction
    - group processes beside each other, and free spaces beside each other
    - Disadvantage: takes too much time



- Partition sizes:
  - Programs grow during execution (more room for stack, heap allocation,...)
  - Problem:
    - If partition too small → programs must be moved → copying overhead & a lot of swapping
  - Temporary Solution: allocate a bit of extra space for program growth 😒
- Management Data Structures:
  - How to identify used & unused spaces?
  - A chunk of memory is either used or unused (free)
  - Bit Maps:
    - 1st data structure to keep track of used & unused memory
    - It is a long string of bits, where each bit represents a chunk of memory.
      - Bit is 1 → used space
      - Bit is 0 → unused space



  - Linked List:
    - 2nd data structure to keep track of used & unused memory
    - We have a struct consisting of:
      - Bit indicating the type of space (P → process, H → hole (free))
      - Starting address
      - Length
      - Pointer to next element
    - A linked lists of elements is saved that gives a map of the memory spaces.



- Algorithms used to load processes:
  - If I have multiple free spaces, where do I store the process in memory?
  - First Fit Algorithm: start from the **beginning of the memory** and load the process at the **first free zone** that is equal or bigger than the size of the process
  - Best Fit Algorithm: Search the **entire memory from beginning to end, and take the smallest hole that is adequate**
  - Worst Fit Algorithm: Search the entire memory from beginning to end, and take the **largest hole that is adequate**
  - Next Fit Algorithm: Like the **First Fit Algorithm, but it searches form the last placed process** (not from the beginning of the memory)
- Memory Layout scheme:
  - Memory is made up of a set of consecutive bytes (addresses)
  - **Continuous layout:**
    - Layout 1: the process is loaded entirely into the memory as a consecutive bunch.
      - Variable sized partitions according to the size of the process (like previously explained)
      - If no free space for a new process → Swapping
      - Problems:
        - overhead in writing to and reading from disk
        - External fragmentation (solved by compaction)
        - How to execute processes whose size is greater than memory size?
        - No sharing processes is possible
    - Layout 2: divide the memory into fixed partitions with different sizes
      - Each partition is associated with a queue of processes.
      - Each process is put in the minimum partition it can fit in.
      - Example:

100KB B	200KB C	50KB A	300KB D

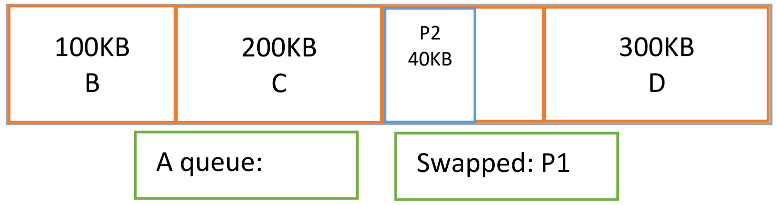
        - processes <= 50 KB → load it into A
        - processes > 50KB & processes <= 100KB → load into B
        - processes > 100KB & processes <= 200KB → load into C
        - Processes > 200KB & processes <= 300KB → load into D
      - Advantage: solves the problem of external fragmentation
      - Suppose this scenario for the above example:
        - P1: 30KB & P2: 40KB arrive
        - P1 is loaded in partition A & P2 is enqueued in the same partition

100KB B	200KB C	P1 30K B		300KB D

A queue: P2

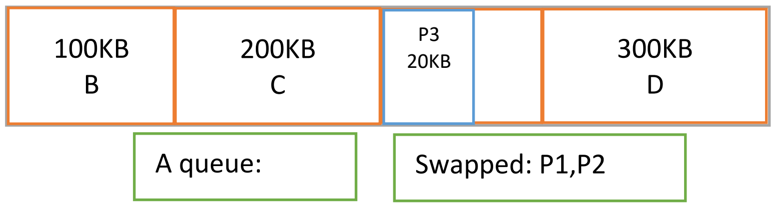
→ Free space of 20KB

- P1 swapped out (overhead) and P2 is loaded in the memory



→ Free space 10KB in A

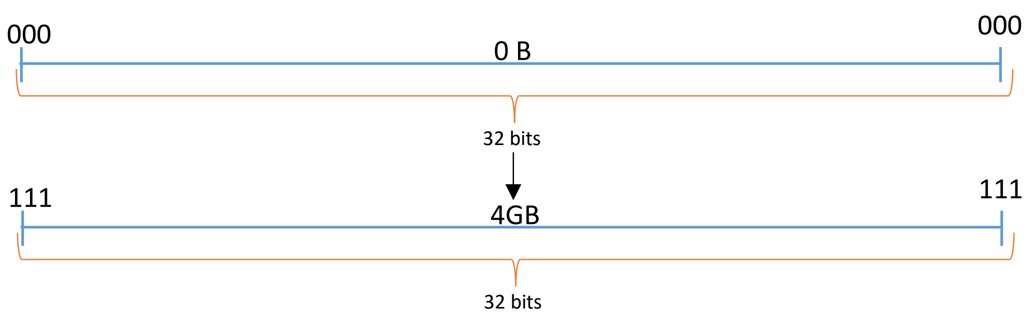
- Suppose P3: 20KB arrived
- P2 is swapped and P3 is loaded in the memory



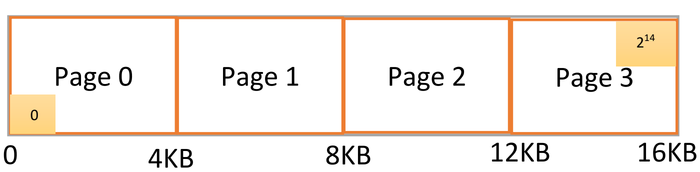
→ Free space 30KB in A

- Disadvantage: Internal fragmentation (free space inside fixed size partitions)

- **Non-continuous layout schemes: (paging)**
  - Based on virtual memory
  - Associate to each process an address space (virtual) equal to the max addressing size
  - Example: if I have an architecture: 32 bits → it can hold a max of 2^{32} bits (4GB)



1. Divide the address space into equal and fixed chunks called pages (paging)
   - The size of the page is a pre-determined parameter configured during the installation of the system (usually 4KB)
   - It should be a power of 2 (4KB = 2^{12})
   - Suppose a process with size 16KB
   - Process → set of consecutive pages from 0 to max

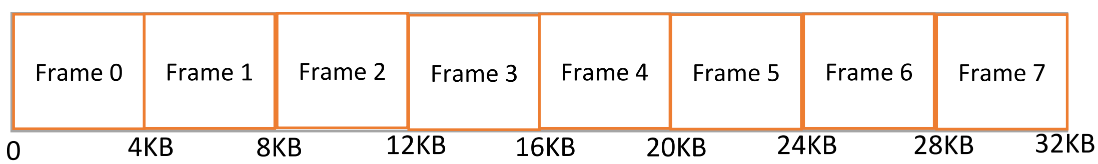


   - Number of pages = $\frac{size(process)}{size(page)} = \frac{16\ KB}{4\ KB} = \frac{2^4 \times 2^{10}}{2^2 \times 2^{10}} = \frac{2^{14}}{2^{12}} = 2^2 = 4\ pages$

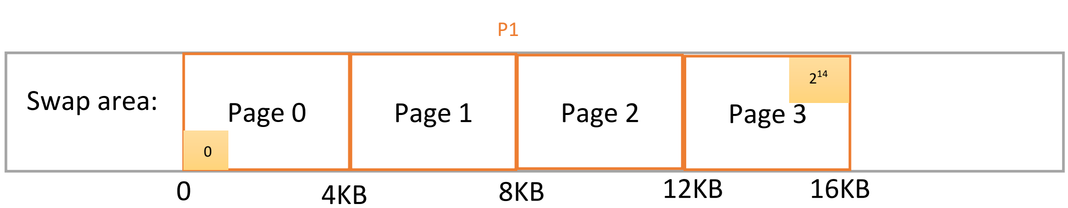
- **Logical address** now consists of 2 parts (page number, offset)
  - Offset: placement inside the page
  - How to find the values
    - Page number: $\frac{address}{size(page)}$
    - Offset: $address \ \% \ size(page)$
  - Page 0 contains addresses from 0 to 4KB → from 0 to 2^{12} B → 0 to 4095 B
    - 1st address(0, 0), 2nd address (0,1) … until last address (0,4095)
  - Page 2 contains the next 4KB → from 4096 to 8191….
    - 1st address is (1,0), 2nd is (1,1) … until last address (1,4095)

- **Example**: CPU wants to access to address 200 B
  - Which page? Which offset?
  - Page: $\frac{200}{size(page)} = \frac{200}{4096} = 0$
  - Offset: $200 \ \% \ size(page) = 200 \ \% \ 4096 = 200$
  - Logical address → (0,200)

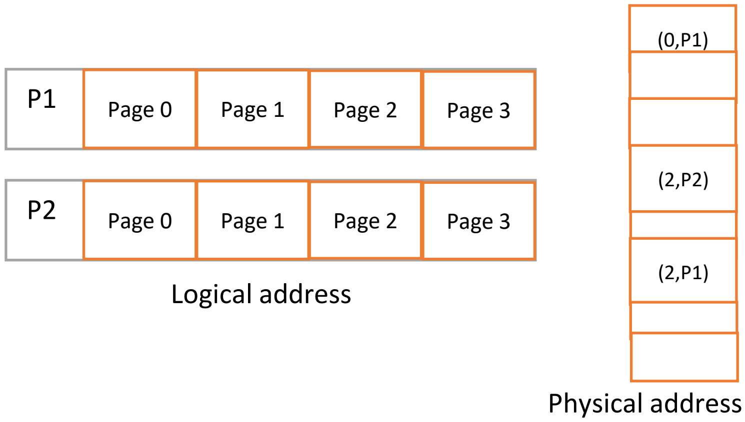
2. Divide the physical memory into fixed and equal chunks called **frames**
   - **size(frame) = size(page)**



   - **Physical address** now consists of **(frame number, offset)**
   - **Physical address = (frame position * size(offset)) + offset**
3. The process isn't loaded entirely into the memory.
   - It is loaded page by page as requested.
   - The pages of the process in memory should not be consecutive
4. How to load the process in memory?
   - The operating system **decomposes the address space of the process as set of pages.**
   - These pages are stored consecutively in the swap area.



   - **During execution:**
     i. CPU makes access to address 255 → (0,255)
     ii. Load the page that contains that @ 255 into memory (page 0)
     iii. The memory manager stores the loaded pages into one of the free frames



iv. The logical address is not the same as the physical address
- **However, offset in physical @ = offset in logical @**
- **Example**: address 8112 in P2
  a. Logical address is (8112/4KB, 8112 % 4KB) = (2,0)
  b. In physical address, page 2 was placed in frame 4 (index 3)
  c. Physical address = (4 * 4096) + 0 = 16384
- So, we have a translation from logical address (pg #, offset) to physical address (frame #, offset)
  i. Offsets are the same
  ii. Page numbers are different

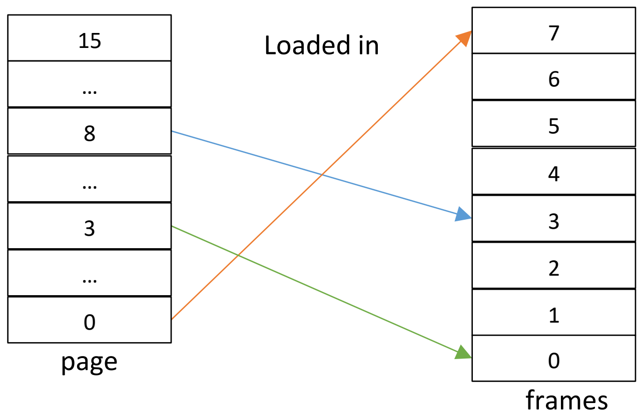
- **Example 1:** if I have an architecture of 20 bits addressing with 4KB page size
  - Questions:
    - How much addresses exist?
    - How many bits are needed to encode the page number?
    - How many pages can we have?
  - Answers:
    - The address is 20 bits. So, we can have 2^{20} different addresses
    - Each page size is 4KB = 2^{12} B. So each page contains 2^{12} addresses → I need 12 bits to represent the offset. Therefore, I need 20-12 = 8 bits to represent the page number in the address
    - Pages are encoded in 8 bits → I can have 2^8 = 256 pages
- **Example 2:** if I have an architecture of 32 bits addressing with 4KB page size
  - Questions:
    - How much addresses exist?
    - How many bits are needed to encode the page number?
    - How many pages can we have?
  - Answers:
    - I can have a total of 2^{32} different addresses
    - Size(page) = 2^{12} → $2^{32}/2^{12} = 2^{20}$ total pages → I need 20 bits to represent the pages.
    - 2^{20} pages
- **Example 3**: access address 32780 in a 32 bit address architecture and page size of 4KB

- I have to write the address in the form of (page#, offset)
- Way 1:
  - Convert the address from decimal to binary
    - 0000 0000 0000 0000 1000 0000 0000 1100
    - The offset is 12 bits (calculated in example 2) → in decimal: 12
    - The page # is the rest (20) → in decimal: 8
- Way 2:
  - Page #: Address/ size(page) = 32780/4096 = 8
  - Offset: Address % size(page) = 32780 % 4096 = 12
- → (8,12)

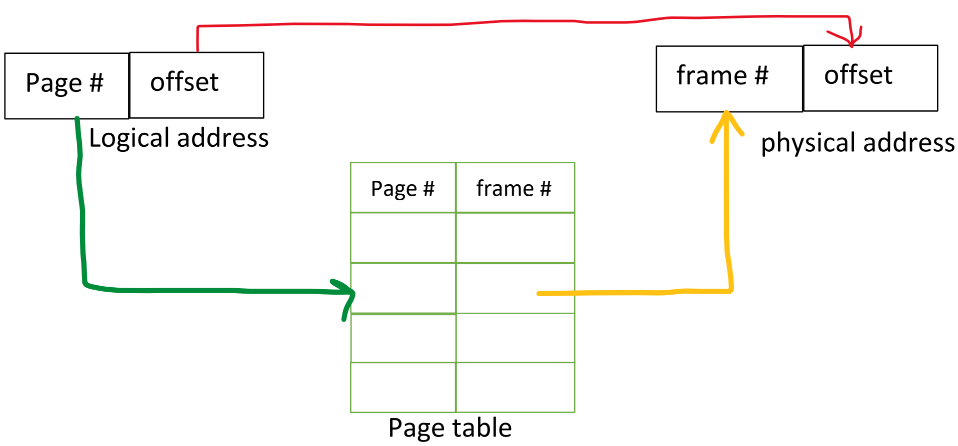
- Conversion from logical address to physical address
  - To know where and if a page is loaded in memory, the MMU uses a page table to show the mapping of the virtual address to the physical address.
  - **Each process has a page table associated with it.**
  - The memory address of the page register is stored in a special register
  - The page table has entries = the maximum number of pages
  - **Each Page Table Entry (PTE) is 32 bits**
  - **Attributes** of the page table:
    - **Presence/absence bit**
      - 1→ page is loaded in physical memory & can be used (page hit)
      - 0 → page is not loaded in physical memory (page fault)
    - **Protection bit**
      - The permissions
      - If on 1 bit:
        - 0 → read & write
        - 1 → read only
      - If on 3 bits:
        - rwx (read, write execute) → a bit for each
    - **Modified**
      - Dirty → it must be written to disk
      - Clean → it can be rejected
    - **Referenced**
      - Useful in choosing the victim for removing when I have a page fault & the memory is full
      - It is the number of times the loaded page was accessed.
      - We choose to remove the page with minimum references.
    - **Frame number**
      - The number of frame in physical memory the page is loaded in

Presence	Protection	Modified	Referenced	Frame #

- **Example:** Consider an address space 64KB = 2^{16} & size(page) = 4KB = 2^{12} & physical memory of 32KB
  - Address space = 2^{16} → I am working in 16 bit architecture
  - Size(page) = 2^{12} → I need 12 bits for the offset
  - Number of bits for the page number: 16-12 = 4
  - Number of pages: 2^4 = 16
  - Size(memory) = 32KB → I have 32KB/4KB = 8 frames



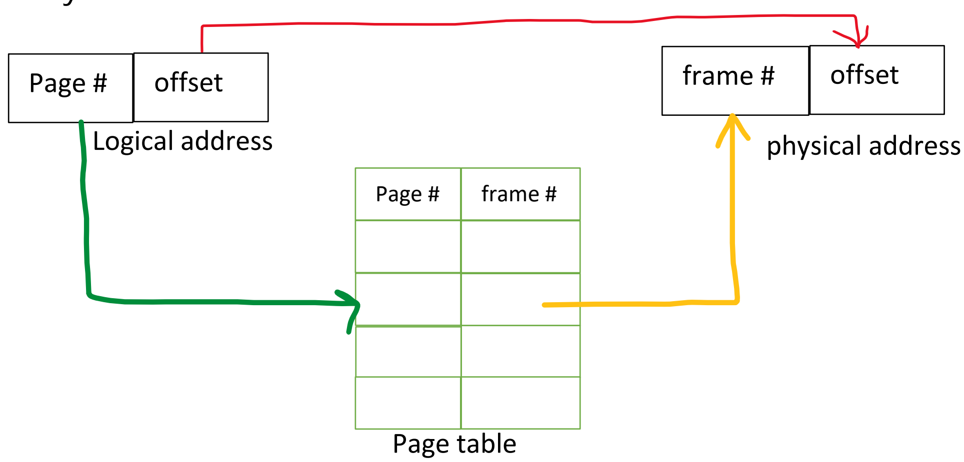
  - Suppose the CPU makes reference to the address 200 B → (0,200)
    - The MMU checks that the presence bit of page 0 is 0 → page fault
    - The MMU the loads the page 0 into memory in any free space
    - Suppose it loaded it in frame number 7
    - (0,200) → (7,200)
    - The physical address will be: (7*4096)+200



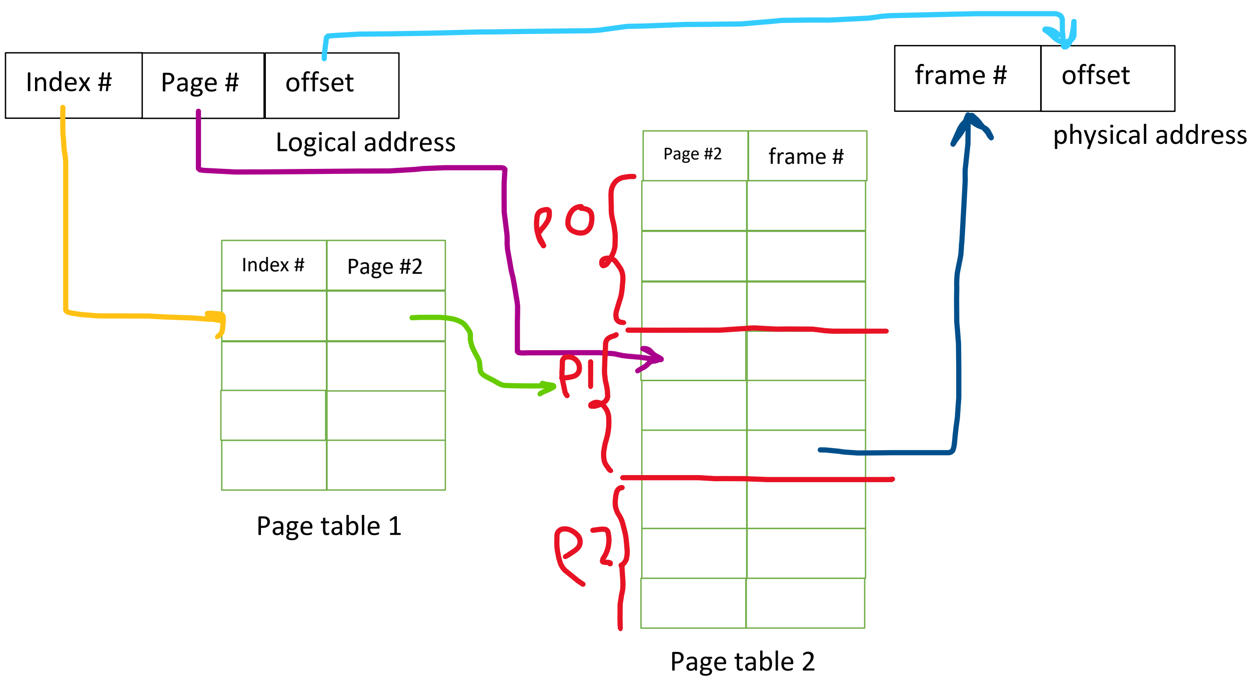
- **Question:** in 64 bit architecture & size(page)=4KB
  - How many entries are contained in the page table?
  - Calculate the size of the page table?
- Answers:
  - Entries in page table = number of pages.

            - Size(page) = 2^{12} → 12 bits for the offset → 64-12 = 52 bits for the page number → 2^{52} pages
            - Number of PTE = 2^{52}
        - Each entry is 32 bits → size(page table) = $2^{52} * 32 - 2^{53} * 25 = 2^{57}$
- Problems we solved with paging:
    - Storing pages with sizes > size(memory)
    - Store several processes with size > size(memory)
    - Protect address space of a process in memory
    - Share data between processes in memory (sharing pages)
    - Solved external fragmentation
- Small problem arises:
    - Internal fragmentation
    - Example: a process of size 9KB & size(page) = 4KB
        - The process will fill 2 pages & use 1KB from the 3rd page
        - The 3KB not used in the 3rd page is what we call internal fragmentation
- **Multi-level page table:**
    - Why do I need a multi-level page table?
        - The page table is stored in the memory → it needs frames to store the whole table. However, processes usually don't use all of the PTEs inside the page table (they only access some). So it would be a waste of space to import all the page table into memory
        - Example:
            - On 32bit architecture & 4KB page size
                - Max size of process: 2^{32} = 4GB
                - 2^{20} different pages
                - 2^{20} PTE
                - Page table size: $2^{20} * 32 = 2^{25}$ = 4MB
                - Storing the whole page table in memory:
                    - I need 1024 frames (4MB/4KB)
                - Suppose we have a process 14MB:
                    - Number of pages this process use: $14MB/4KB = 14 * 2^8 << 2^{20}$
                - So, there is a lot of empty pages stored inside the memory
        - So, we use multi-level paging, aka paging of the page table, where we only store the pages we need in the memory.
    - Now, to access the physical memory, we need to pass through different page tables.
    - The initial page table that we have is paged
        - We divide it into pages and create a new page table for the page table
        - **Example:**

- Previously:



- Now:



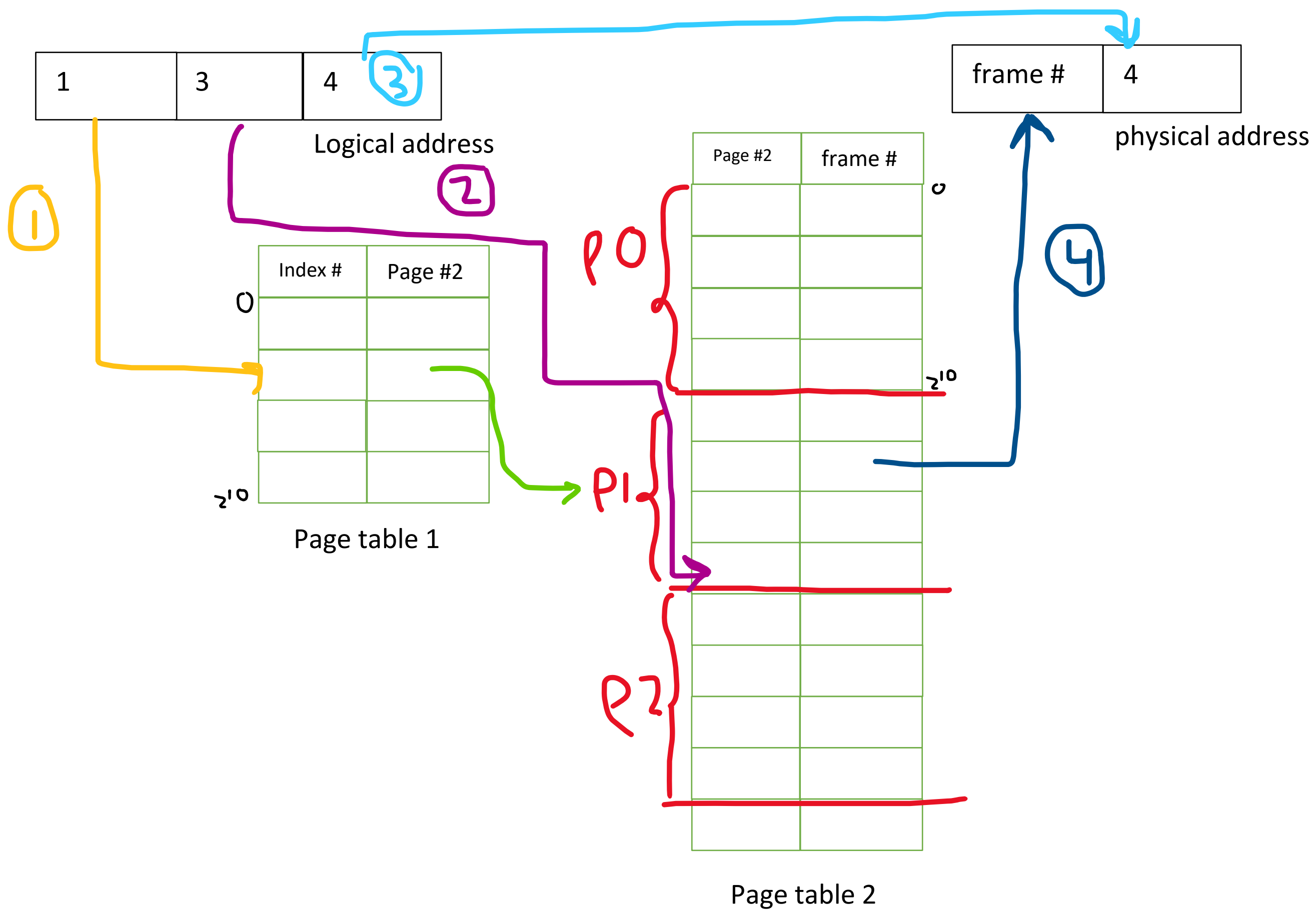
- Accessing the physical memory is divided into 2 parts
  - **First level (outer table)**
    - To indicate which chunk in the page table is the address in
  - **Second level (inner table)**
    - To indicate which frame the page is stored in memory
- Physical address = (index # * size(chunk)) + (page # * size(page)) + offset
- **Example**:
  - In a 32bit address & 4KB page size:
    - In a single-level page table: I would need 20 bits for the page table, & 12 for the offset

- In a multi-level page table: I would need 10 bits for the index number, 10 bits for the page number, and 12 bits for the offset

  - How did I know that I need 10 bits for the index:
    - PTE is 32 bits = 4B
    - Page size is: 4KB
    - I need to divide the page size by the PTE: 4KB/4B = 2^{10} entries → The page table is divided into 2^{10} pages/chunks
    - I need 10 bits to represent each chunk in the page table

- **Example:** CPU to move to register address 4,206,596 B (32bit architecture & 4KB page size)
  - We know that we have 10 bits for index number & 10 bits for page number & 12 for offset
  - Way 1:
    - Change decimal to binary → 0000 0000 0100 0000 0011 0000 0000 0100
    - Index #: 0000 0000 01 → 1
    - Page #: 00 0000 0011 → 3
    - Offset: 0000 0000 0100 → 4
  - Way 2:
    - 10 bits for index → we have 2^{10} chunks/indexes
    - Each chunk has 2^{10} pages where each page size is 2^{12}
    - Each chunk size is $2^{10} * 2^{12} = 2^{22}$ B = 4MB
    - Each chunk has 2^{22} addresses
      - First chunk: 0 → 4194303
      - Second chunk: 4194304 → 8388607 ....
    - So, the address 4206596 is in chunk: $4206596/2^{22}$ = 1st chunk
    - What page is it in the page table?
      - $4206596 - (1 * 2^{22}) = 12292$ address in 2nd page table where each page size is 2^{12}
      - $12292/2^{12}$ = 3rd page
    - Which offset? $12292 - (3*2^{12}) = 4$

- $4206596 = (1 * 2^{22}) + (3 * 2^{12}) + 4$



- Problems of Multi-level:
  - Slows down the performance/speed of CPU by a lot
- Solution: Inverted Page Table

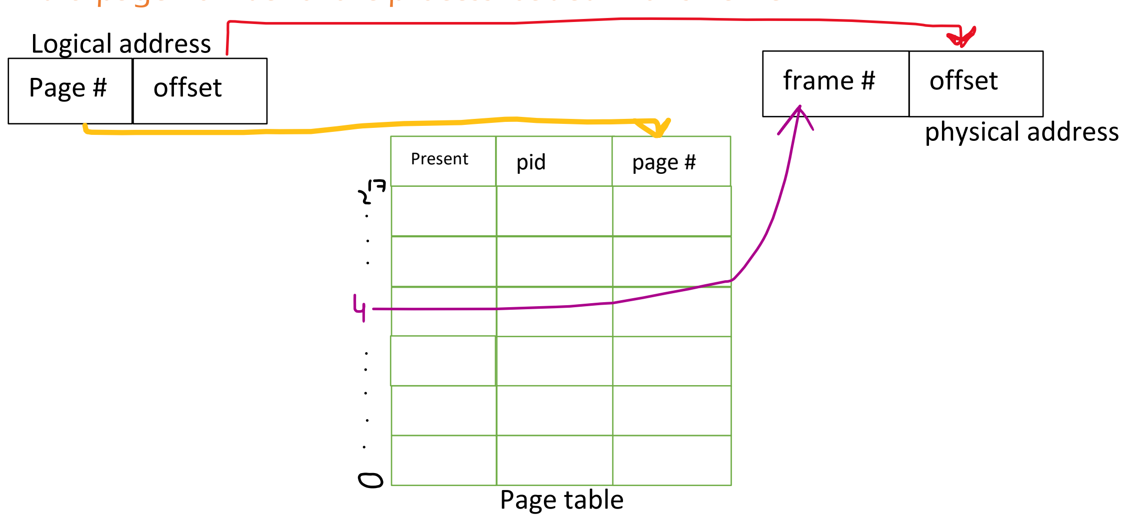
- Consider this scenario:
  - System architecture 64 bits
  - Page size 4KB
  - Memory size 512 MB
  - Questions:
    - How much space would a single-level page table take?
    - How much space would a multi-level page table take?
  - Answers:
    - Page size 4KB = 2^{12} B→ I need 12 bits for offset → 52 bits (64-12) for page number. So, I have 2^{52} PTEs where each page is 32 bits = 4B. Size of single level page table is: $2^{52} * 2^{2} = 2^{54}$ B

- Size(page) = 4KB = 2^{12} B & size(PTE) = 32 bits = 4B. I need $2^{12}/2^{2} = 2^{10}$ entries for each multi-level table. If I have 12 bits for the offset, the other 52 bits should be divided into 10bit parts.

2	10	10	10	10	10	12

  - → 6 levels on indexing → each address translation needs 6 memory references → performance problem!!!

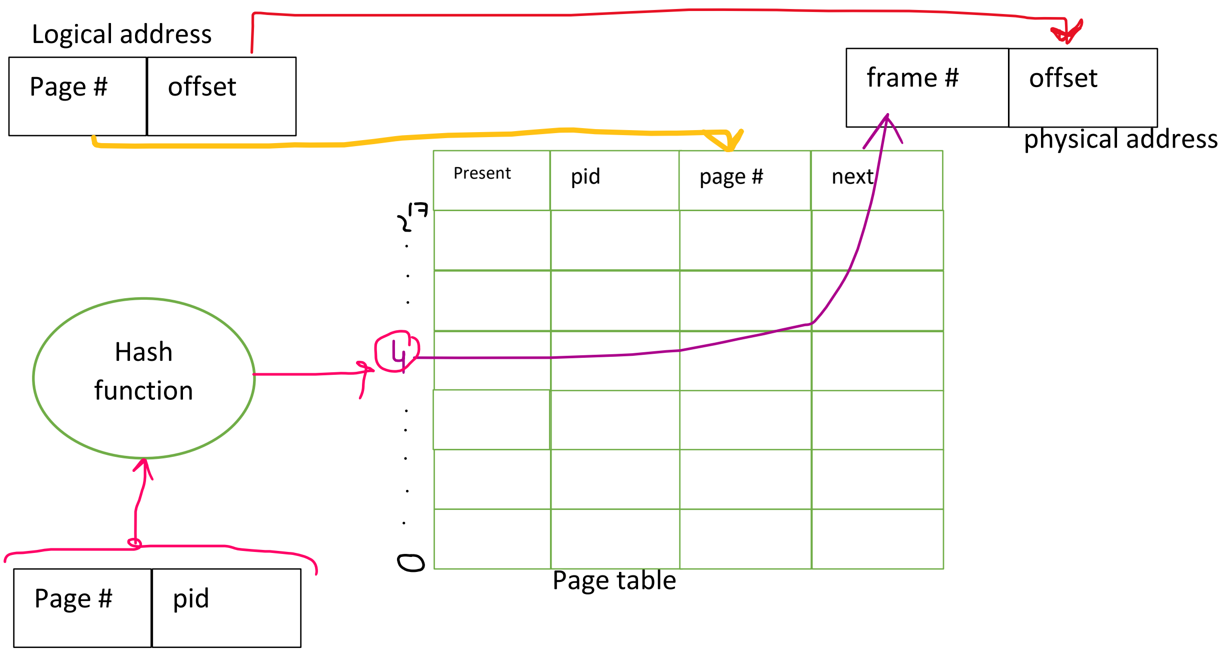
- Inverted Page table
  - Make a **single and global** page table **related to the physical memory** and not related to a process.
  - Consider the example of the above scenario
    - Size(memory) = 512 MB = 2^{29} B
    - Size(page) = size(frame) = 4KB = 2^{12} B
    - Number of frames I need is: $2^{29}/2^{12} = 2^{17}$ frames
    - Create a table with 2^{17} entries
    - Each entry is related to a physical frame in memory
    - Suppose the entry is 16B → the size of the whole page table is $2^{17} * 2^{4}$ = 2MB
  - The inverted table includes:
    - The presence/absence field, modified field, protection field (same as normal page table)
    - Pid & page number of the process loaded in this frame



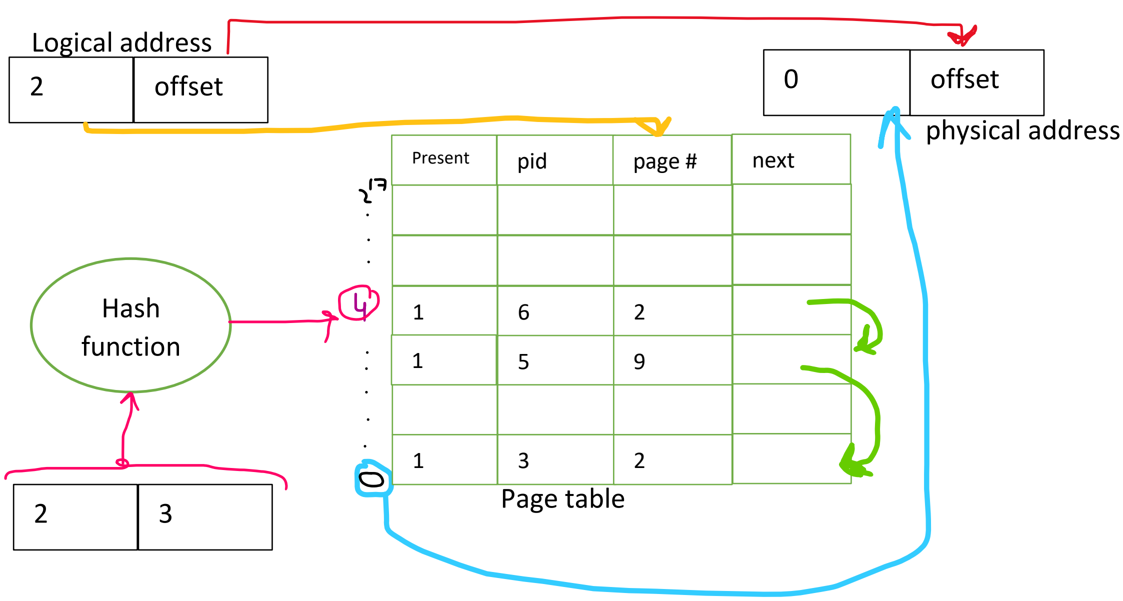
  - How is the logical address converted?
    - For each logical address, decompose it as usual (page #, offset) & we have the pid of the process
    - The MMU searches in the inverted page table in either 2 ways
      - Linear: It searches the page table entry by entry
        - If the entry is found (page hit): I get the frame number

- If the entry is not found (page fault): load page from disk and update the corresponding entry in the inverted page table

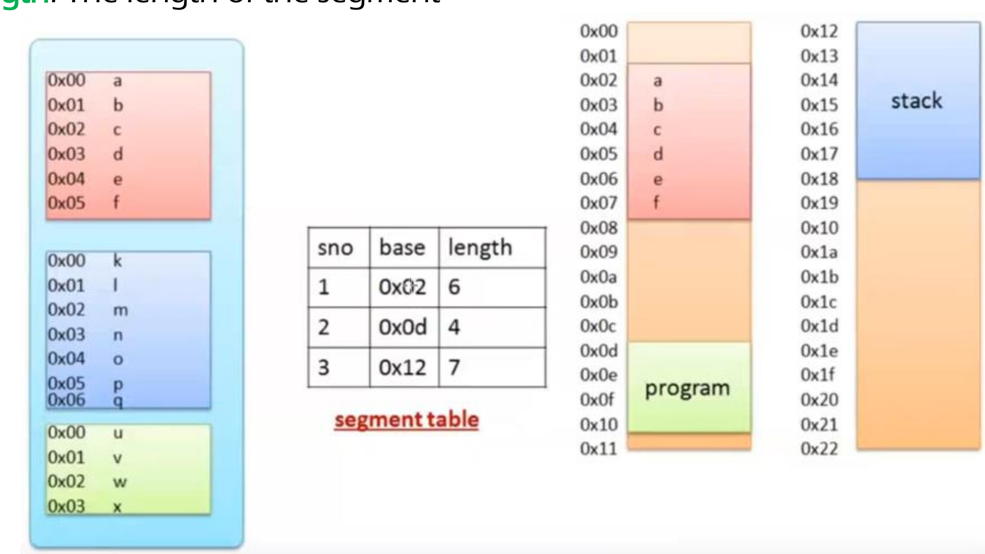
- Hashing:
  - We use a hash function that hashes a key into a value
  - This value will give the index of the frame in the inverted page table
  - For the same key, the hash function outputs the same value.



- If 2+ keys have the same value, collision occurs. We can use a linked list to keep track of the keys with the same value.
- Example with collision:

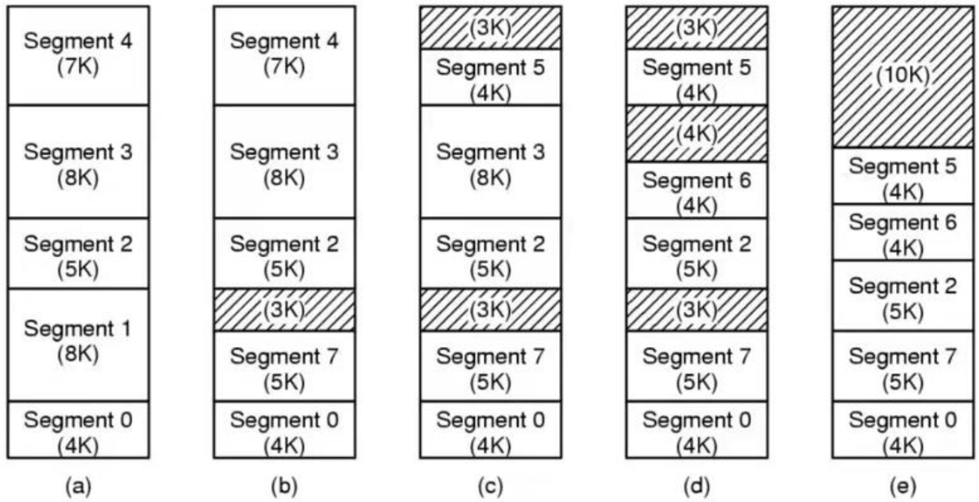


- Segmentation
    - The process is made up of data, code, heap, & stack.
    - Paging uses one continuous sequence of all virtual addresses from to the maximum needed for the process
    - Segmentation is an alternate schema that uses multiple separate address spaces for various segments of a program. (process is divided into segments)
    - A segment is a logical entity in which the program is aware of (function, array, stack…)
    - Each segment has a variable length that may change during execution
    - The **logical address is (segment #, offset)**
        - Segment # to specify which segment the address is located in
    - It is similar to dynamic partitioning (because of variable sizes). However, the difference is that segments can be not contiguous.
    - It eliminates internal fragmentation, but **suffers from external fragmentation**
    - There is no relationship between logical & physical address
    - **A segment table is needed. It includes:**
        - **Segment number**
        - **Base**: The base/starting address of the segment
        - **Length**: The length of the segment



    - **Example**: Suppose I want to find c with logical address 0x02
        - Which segment? Segment number 1 has a length of 6 → from 0x00 to 0x05 → c is in segment 1
        - Since it is in segment 1, the offset is 2
        - Logical address → (1, 2)
        - The segment table indicates that segment 1 starts at physical address 0x02. But the offset of c is 2
        - Physical address of c is 0x02+2 = 0x04
    - Segments allow sharing between processes (like sharing libraries)
        - The library can be put in a segment and shared by multiple processes
        - Avoiding the need to put the library in each process's address space
    - The user can put protection on each property/segment

- A function segment can be set to execute but not read or write.



(a) Memory initially containing 5 segments of various sizes.
(b)-(d) Memory after various replacements: external fragmentation
(e) Removal of external fragmentation by compaction eliminates the wasted memory in holes.

- Segmentation with paging
  - The process is divided into segments
  - Each segment is divided into pages
  - **Logical address → (segment #, page #, offset)**
  - **Physical address → (frame # * size(page)) + offset**
- **Exercise:** memory with segmentation & paging
  - Page size: 4KB
  - Physical memory: 64KB
  - Process P is composed of 3 segments: S1 (16KB), S2 (4KB), S3 (8KB)
  - At instance t
    - Page 1 Segment 1 loaded into physical memory 2
    - Page 2 Segment 1 loaded into physical memory 0
    - Page 0 Segment 2 loaded into physical memory 9
    - Page 0 segment 3 loaded into physical memory 12
  - Questions: For decimal address 8212, Indicate the
    1. segment number
    2. page number in the segment
    3. offset
    4. frame number
    5. physical address
  - Answers:
    1. # of frames in memory: 64KB/4KB = 16
       - Segment 1 is 16KB: from address 0 → 2^{14} (16384) → address 8212 is in **segment 1**
    2. # of pages in segment 1: 16KB/4KB = 4 each with size 4KB

- First page from 0 → 4095
- Second page from 4096 → 8191
- Third page from 8192 →12287....
- So address 8212 is in the 3rd page → **page # = 2**

3. Page # 2 starts with address 8192 → offset: 8212 – 8192 = **20**
4. Page 2 Segment 1 → **frame = 0** (from given)
5. **Physical address = (frame # * size(frame)) + offset**
    - Physical address = (0 * 4096) + 30 = 20

- Page Replacement Algorithms
    - If the memory is full & I need to load a new page into the memory, I need to evict a page in the memory to replace it with the new one.
    - Steps:
        - Know which page I need to remove
        - If it has been modified, write it back to disk
        - Load the page into memory
    - Which page do I evict? I have to leave in memory the most important pages → the most requested pages. This is because I need to avoid excessive loading from disk.
    - There are 2 strategies to find which page:
        - Local: to evict the pages that are related to this process
        - Global: to evict pages with respect to the whole memory (Our course of study)
    - There are multiple algorithms that help us choose which page to choose.
    - **Terminology**:
        - **Reference String**: the memory reference sequence generated by a program
        - Paging: moving pages to/from disk
        - Optimal: best (theoretical) strategy
        - Eviction: throwing something out
    - Algorithms:
        - Random Algorithm: choose a page randomly
        - Optimal Algorithm:
            - Theoretical (never used), It is only for comparison purposes
            - Assumes that we know the future reference string for a program → It would choose the page with the last reference in the future

	Page Refs	3 Page Frames			
		Fault?	Page Contents		
	A	yes	A		
	B	yes	B	A	
	C	yes	C	B	A
A & B will be called the most recent	D	yes	D	B	A
	A	no	D	B	A
	B	no	D	B	A
A & B will be called the most recent	E	yes	E	B	A
	A	no	E	B	A
	B	no	E	B	A
A will not be called	C	yes	C	E	B
B will not be called	D	yes	D	C	E
	E	no	D	C	E

7 faults

- FIFO:
  - The first page in is the page to go

Page Refs	3 Page Frames Fault?	Page Contents		
A	yes	A		
B	yes	B	A	
C	yes	C	B	A
D	yes	D	C	B
A	yes	A	D	C
B	yes	B	A	D
E	yes	E	B	A
A	no	E	B	A
B	no	E	B	A
C	yes	C	E	B
D	yes	D	C	E
E	no	D	C	E

9 faults

  - **Anomaly**: As the number of page frames increases, the number of faults increases

Page Refs	4 Page Frames Fault?	Page Contents			
A	yes	A			
B	yes	B	A		
C	yes	C	B	A	
D	yes	D	C	B	A
A	no	D	C	B	A
B	no	D	C	B	A
E	yes	E	D	C	B
A	yes	A	E	D	C
B	yes	B	A	E	D
C	yes	C	B	A	E
D	yes	D	C	B	A
E	yes	E	D	C	B

10 faults

- Least Recently Used:
  - Removes the page that has not been referenced for the longest time
  - Keep track of the $t_{last\ accessed}$
  - Reference string: {A, B, C, D, A , B, E, A, B, C, D ,E}

	Fault	F1	F2	F3
A $t_{last\ accessed}$: 0	Yes	A		
B $t_{last\ accessed}$: 1	Yes	B	A	
C $t_{last\ accessed}$: 2	Yes	C	B	A
D $t_{last\ accessed}$: 3	Yes	D	C	B
A $t_{last\ accessed}$: 4	Yes	A	D	C
B $t_{last\ accessed}$: 5	Yes	B	A	D
E $t_{last\ accessed}$: 6	Yes	E	B	A
A $t_{last\ accessed}$: 7	No	E	B	A
B $t_{last\ accessed}$: 8	No	E	B	A
C $t_{last\ accessed}$: 9	Yes	C	B	A
D $t_{last\ accessed}$: 10	Yes	D	C	B
E $t_{last\ accessed}$: 11	Yes	E	D	C

10 faults

- According to the number of references & dirty bit
  - **Reference bit**: increases every time the page is read or written
  - **Dirty bit**: set when page is written to

- Cases:

Reference	Dirty Bit
0	0
0	1
1	0
1	1

- Remove the page that is least referenced (R=0) & not written to (d = 0) (else the one with R=0 & D=1, else R=1 & D = 0…)

▪ Second Chance Algorithm
- Based on FIFO & Reference
- Steps:
    - Look at the oldest page
        - Reference bit is 0 → remove it
        - Reference bit is 1
            - Change reference bit to 0
            - move to next oldest
    - Repeat
    - If all the frames referenced → change all references to 0 & remove the oldest one

Page	3 Page Frames			
Refs	Fault?	Page Contents		
A	yes	A•		
B	yes	B•	A•	
C	yes	C•	B•	A•
D	yes	D•	C	B
A	yes	A•	D•	C
B	yes	B•	A•	D•
E	yes	E•	B	A
A	no	E•	B	A•
B	no	E•	B•	A•
C	yes	C•	E	B
D	yes	D•	C•	E
E	no	D•	C•	E•

- Exercise 1: Program code occupies 1024 B in memory & uses a vector or 1000 B
    - Physical memory: 1 MB
    - Memory is paginated
    - Page size: 512 B
    - Addressing in 24 B
    - Questions:

A. Indicate the

1) size of the virtual address space
2) offset
3) number of bits of the page number
4) number of bits of physical address
5) number of bits of physical frame
6) number of entries of PTE
7) size of page table

B. The loading of this page in memory causes internal fragmentation. Justify?

- Answers:

A.

1) Maximum size of the virtual address space is 2^{24}
2) Size(page) = 512 B = 2^9 B → I need 9 bits for the offset
3) 9 bits for offset → (addressing– offset = page #) 24-9 = 15 bits for page number
4) Physical memory is 1MB = 2^{20} B → 20 bits for physical address
5) Bits for physical memory - bits for offset = bits for physical frame → 20 – 9 = 11 bits
6) number of PTE = number of pages = 2^{15}
7) size of page table = (number of PTE * size of PTE) (size(PTE) is always 32 bits = 4B) → $2^{15} * 2^2 = 2^{17}$

B. The process has code 1024 B & data 1000 B. The size(frame) = size(page) = 512 B

1) Code needs 1024/512 = 2 frames
2) Data needs 1 full frame (512 B) and some of another frame (488 B)
3) → The process didn't use a full frame/page → Internal fragmentation