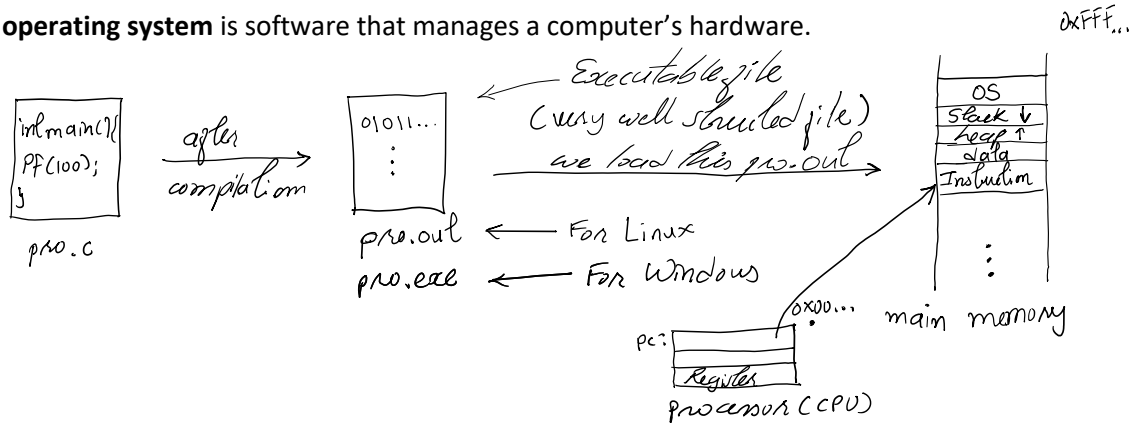
1. Introduction to Operating system

//Lecture 1

1.1) Operating system :

An **operating system** is software that manages a computer's hardware.



CPU or processor : There is clock (hardware) in CPU which generate timing signals at regular interval to fetch, decode and execute and this cycle repeats until program ends.

Multitasking and multiprogramming : when multiple programs execute at a time on a single device. It is non-preemptive. Multitasking is an extension of multiprogramming we can have preemption and additional feature of time sharing.

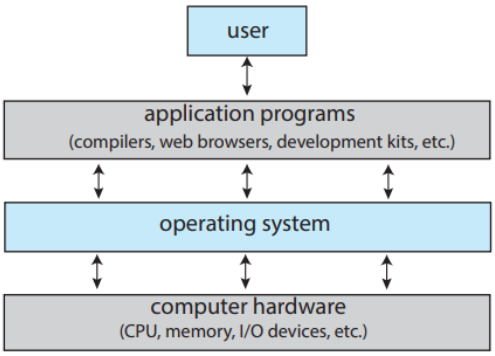
Running a small program on CPU :

Consider following program,

a = 1; b = 2; c = a + b;

Without OS : we have to do things manually like setting up Program counter to start of main function. Then CPU fetch decode and execute c = a + b (after storing values of a and b in CPU register). This a + b (summation operation) is done by CPU. But after computing we cannot printf value of variables because "printf" needs OS support because CPU does not own the monitor (I/O) device. CPU has to take permission from monitor.

We can say that OS manage memory, resources (I/O devices), file system,…



Above diagram shows the abstract view of the components of a computer system.

The first version of UNIX : It was one successful attempt to build stable OS. There were also OS 360 but it was having more than 1000 major bugs.

When the start button is pressed in your Computer following sequence of events occurs

- CPU fetches the instruction into RAM from the BIOS which is stored in ROM.
- BIOS does some basic checks to make sure that all the components of computer are working properly.
- The BIOS then starts the boot sequence and will look for Kernel.
- BIOS will fetch operating system from the hard drive and load into the RAM.
- The BIOS then transfers control to the operating system.

1.1.1) System calls :

System calls provide an interface to the services made available by an operating system. Before we discuss how an operating system makes system calls available, let's first use an example to illustrate how system calls are used: writing a simple program to read data from one file and copy them to another file. The first input that the program will need is the names of the two files: the input file and the output file. These names can be specified in many ways, depending on the operating-system design.

Usually, a program resides on disk as a binary executable file— for example, a.out or prog.exe. To run on a CPU, the program must be brought into memory and placed in the context of a process. In this section, we describe the steps in this procedure, from compiling a program to placing it in memory, where it becomes eligible to run on an available CPU core.

Source files are compiled into object files that are designed to be loaded into any physical memory location, a format known as a **relocatable object fil**e. Next, the **linker** combines these relocatable object files into a single binary executable file. During the linking phase, other object files or libraries may be included as well, such as the standard C or math library. A **loader** is used to load the binary executable file into memory, where it is eligible to run on a CPU core. An activity associated with linking and loading is **relocation**, which assigns final addresses to the program parts and adjusts code and data in the program to match those addresses so that, for example, the code can call library functions and access its variables as it executes.

The process described thus far assumes that all libraries are linked into the executable file and loaded into memory. In reality, most systems allow a program to **dynamically link libraries** as the program is loaded.

//Lecture 2a

1.1.2) Creating a new process :

A **process** is a program in execution. To start a new process, the shell executes a fork() system call. Then, the selected program is loaded into memory via an exec() system call, and the program is executed. We will see this in details after introducing fork and exec system call…

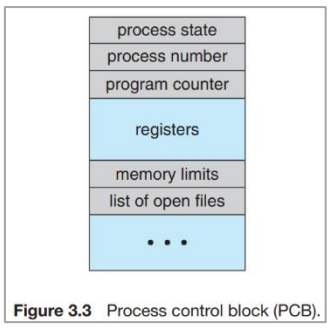
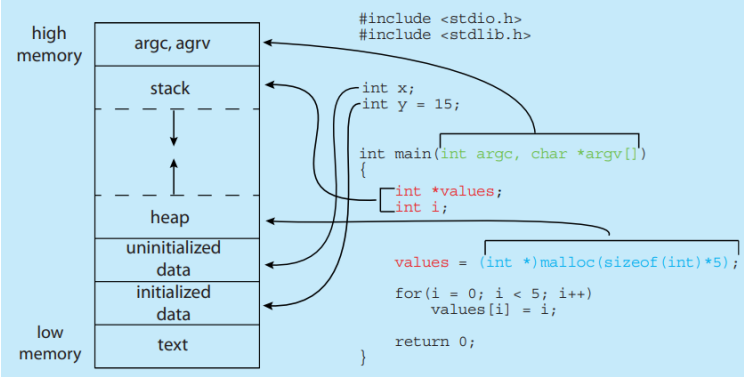
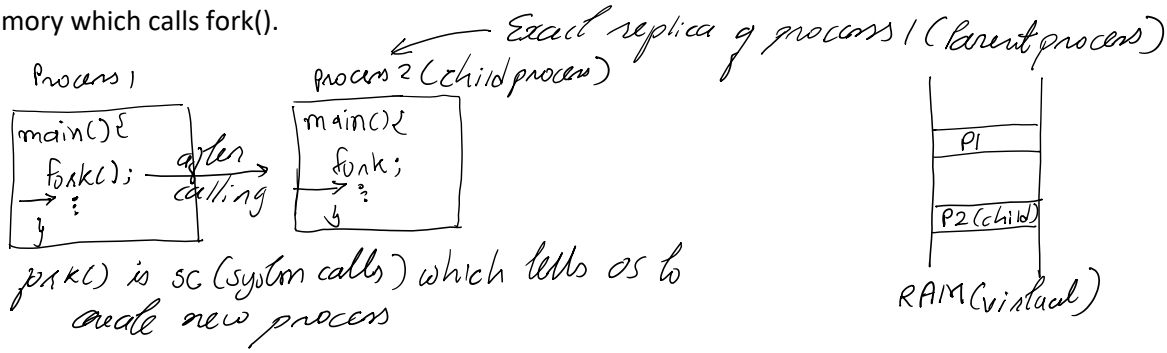


Figure 3.3 Process control block (PCB).



Layout of the process in memory (logical memory)

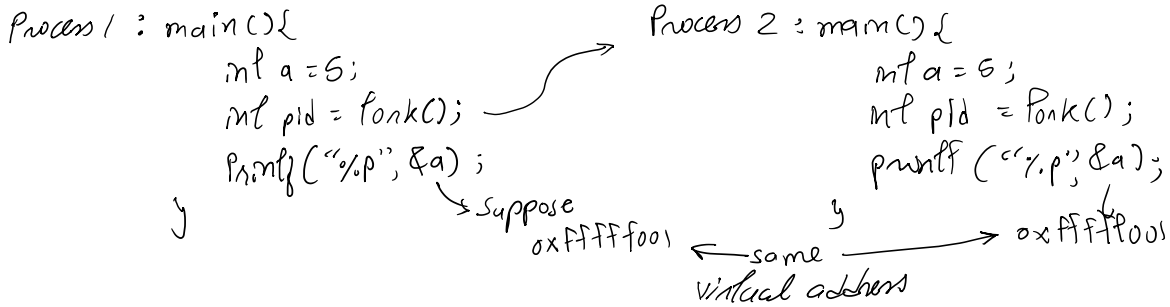
To understand fork() system calls let's take one example, suppose there is one process running in main memory which calls fork().



Q : Which process runs after doing fork() sc by process 1 ? – Any process can run. And both parents and child both will start executing after the line which had fork(). (In diagram it is shown by arrow) and fork system call will return some integer value. Fork system will return 0 value to child process and non-zero value (this value is nothing but **process Id** of child process) to parent process. When fork returns a negative integer, an error has occurred, but no child process is actually created.

Printf always prints logical address

Suppose,



Because when fork call is made, OS made copy of process meaning all the info related to all variable whatever you can think of, it made exact replica of parent process.

```
int main(){
    int a = 50; //Let's say &a = 0xffff1000
    int pid = fork();
    if(pid == 0){
        a = a+5;
        printf("I am child and here a, &a is %d, %p\n", a, &a);
    }
```

} child will execute this

```
    else{
        a = a-5;
        printf("I am parent and here a, &a is %d, %p", a, &a);
    }
    return 0;
}
```

} parent will execute this

Output : `I am child and here a, &a is 55, 0xffff1000`

`I am parent and here a, &a is 45, 0xffff1000`

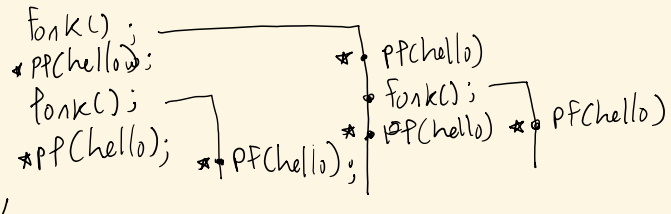
As you can see both values of a are independent but addresses are same because although their logical addresses (sometimes called **address space**) are same but in main-memory, they both occupy different space and that is why their physical addresses are different because of both are different variable. Which means <u>all the variable in parent process are different from child process but their logical addresses are same.</u>

Q : how many hello's are printed ? (assume fork does not fail)

process diagram

```
int main(){
    int i;
    for(i = 0;i<2;i++){
        fork();
        printf("hello\n");
    }
    return 0;
}
```

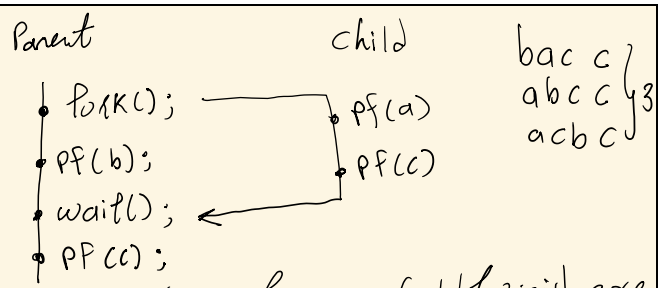
whenever loop is given open it !



6 times !

Q : Write down all possible patterns printed by following code.

```
int main(){
    if(fork()==0) printf("a");
    else {
        printf("b");
        wait();
    }
    printf("c");
    exit(0);
}
```



bac c
abc c } 3
acb c

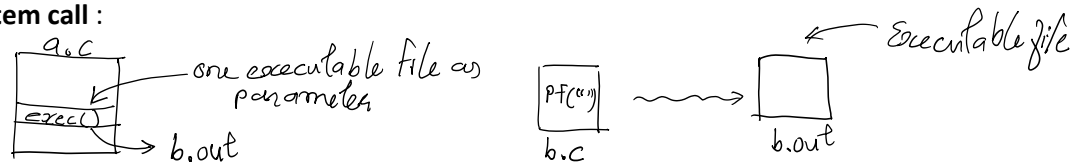
after printing b, parent waits for child to finish exe.

//Lecture 3

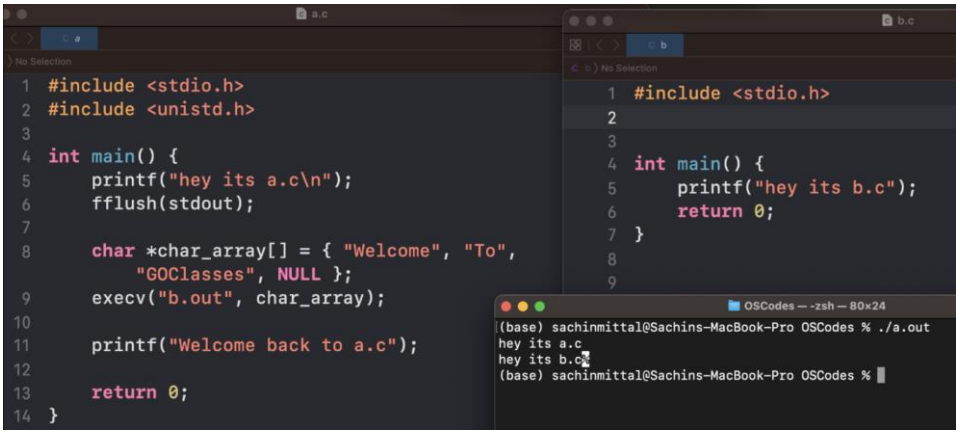
All time you cannot make this diagram so there is shortcut : if there are n fork calls in loop then total processes are created is 2^n (including parent process).

//Lecture 4

Exec system call :



When you run exec() system call a.out file gets replaced with b.out file in memory.



As you can see “Welcome back to a.c” will not be printed as a.out gets replaced by b.out

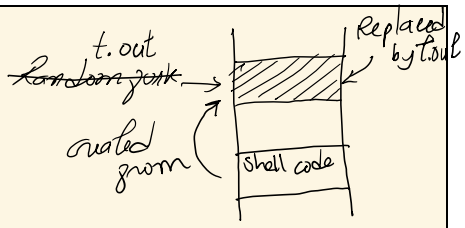
1.1.3) Process creation :

Suppose you are running your default program shell. And in shell process itself you call create process by fork(), so now you have one replica of shell. And before executing replica you do exec() system call which replaces child shell to new executable file with same pid.

```
//Shell code//
while(true){
    scanf("%s", @t);
    pid = fork();
    if(pid == 0) exec(t);
    else wait();
}
```

some command

wait till child finish



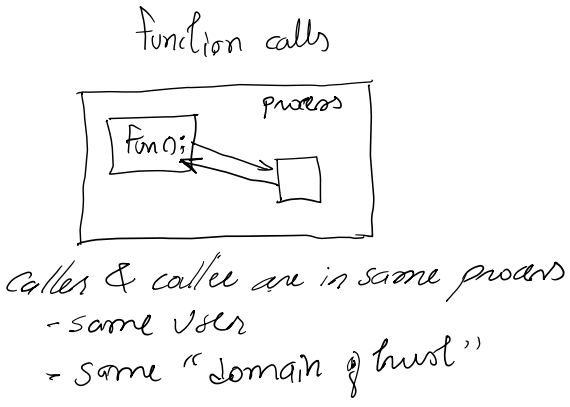
Q : How this shell is created ? – Shell is very first program created automatically at boot time.

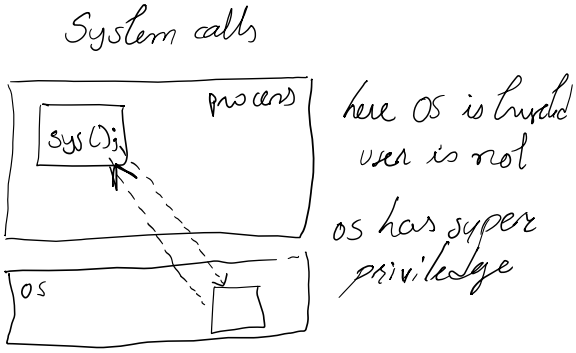
This is how process gets created !

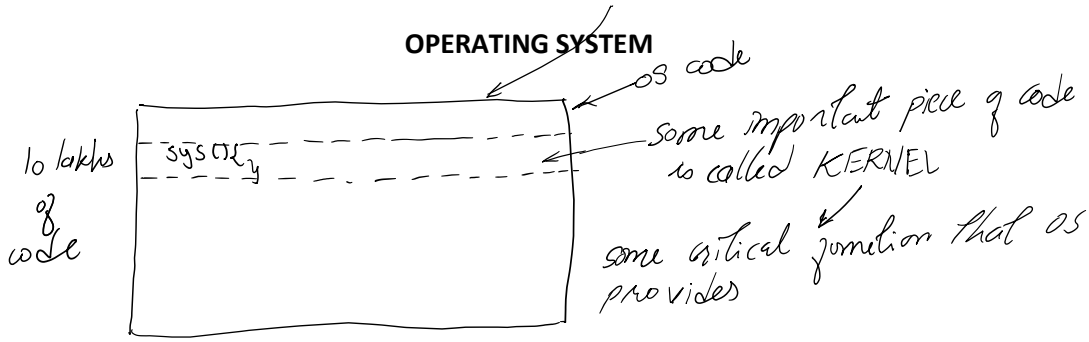
Q : What if same t process gets replaced by t process, I mean will it’s variables changes or remains same ? – off course, variables gets changed to their original value as whole process again runs as it is.

When a user program successfully calls exec(), its process is replaced with a new program. Everything about the old process is destroyed, including all of the CPU registers, program counter, stack, heap, background threads, and virtual address space but only file descriptor remains because file descriptor is a table which has entries corresponding to read, write, etc I/O operations. So, we can’t delete it.

More about system calls :







All the system calls are part of kernel. But you can't just enter into Kernel as it contains sensitive code, you can enter into kernel using entry point of kernel also called **system calls**.

You know that every user don't know internals of computer or OS so he or she can make incorrect operations while managing the resources. This can lead to system failure.

OS says Never trust a user while dealing with critical tasks

Dual modes of operations :

1) **User mode** : compute operations like addition, subtraction, etc and reading and writing into program's memory
2) **Kernel mode** : CPU and memory management, I/O handling.

A user process cannot invoke system calls like regular functions because they are not part of the kernel-space. Modern CPU provide a special instruction generally called a **software interrupt**, which can be performed by a user-space software and which causes the CPU to switch to the system mode and give the control to the kernel, just like when serving a hardware interrupt or an exception.

User processes usually do not directly invoke system calls. Instead they use something which is called an *Application Programming interface* (an **API**, for short). It is a set functions, variables and other software constructs that allow the processes to perform the most needed operations and also to cooperate with the operating system.

//Lecture 5

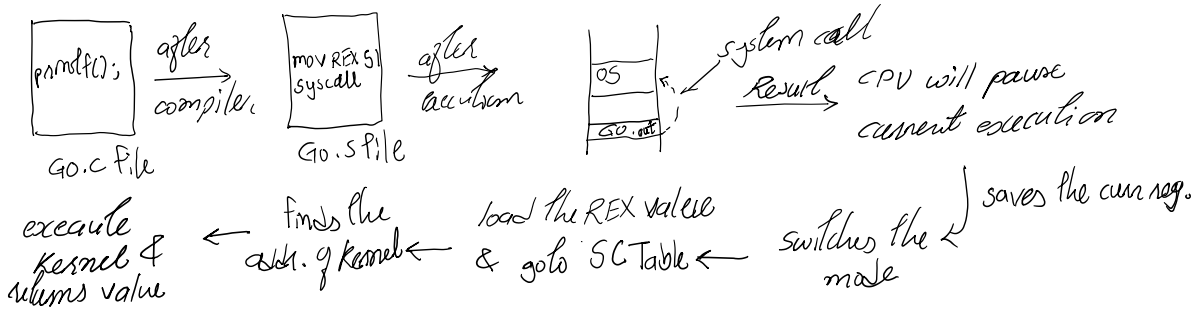
So, we have another meaning of system calls : we can say

- it is request to kernel to perform a service like open a file, execute a new program, create a new process.
- **Entry point** providing controlled mechanism to execute kernel code.

Set of system calls is operating system specific.

1.1.4) System call execution :

Each system call has unique numeric identifier. OS has a system call table that maps numbers to functionally requested. When invoking a system call, user places system call number and associated parameters in a specific register, then executes the syscall instruction.



Steps :

1) Program calls printf function in c library
2) Printf function : packages syscall arguments into registers and puts syscall number into a register
3) Program flips CPU to kernel mode (user mode to kernel mode) and execute special machine instruction (e.g, syscall) and main outcome of these sequence of procedure is now CPU can touch memory marked as accessible only in kernel mode.

Inside the syscall instruction :

- saves the old(user) SP value,
- switches the SP to the kernel stake
- it saves the old(user) PC value (= return addr)
- saves the old privilege mode
- sets the new privilege mode to 0
- sets the new PC to the kernel syscall handler

4) Kernel executes syscall handler.
   Invokes service routine corresponding to syscall number and do the real work, generate desired output.
   Places the return value from service routine in a register

Switches back to user mode. Passing control back to wrapper (source code) ■

So, as you can see system call requires lots of task (indirectly cycle) so it is slower than function calls.

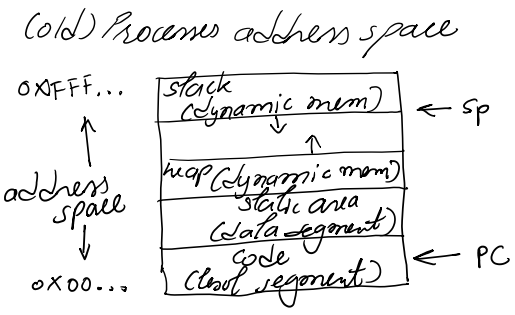
System call table : it is a protected entry points into the kernel for each system call. (why kernel because we don't want application to randomly jump into any part of the OS code.)

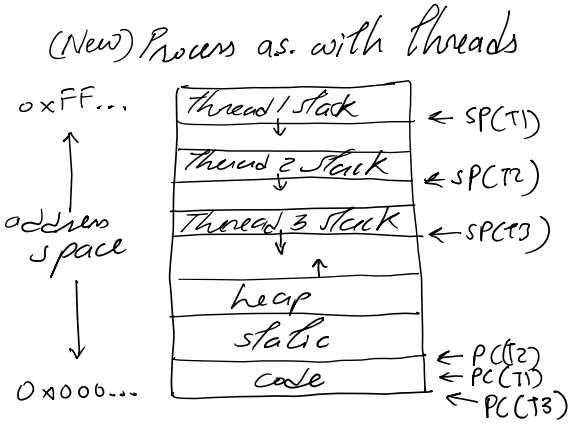
Syscall table is usually implemented as an array of function pointer, where each function implements one system call.

//Lecture 6

1.2) Threads :

In certain situations, a single application may be required to perform several similar tasks. Instead of creating different process we can have a something called **threads** because creating different processes for same task requires IPC (inter process communication) and process context requires is bit costlier than thread context switching.

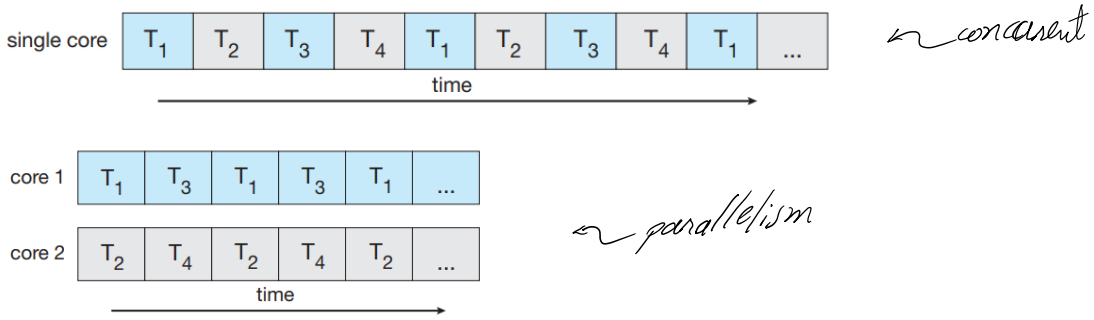




This feature is especially beneficial on multicore systems, where multiple threads can run in parallel. A multithreaded word processor, for example, assign one thread to manage user input while another thread runs the spell checker.

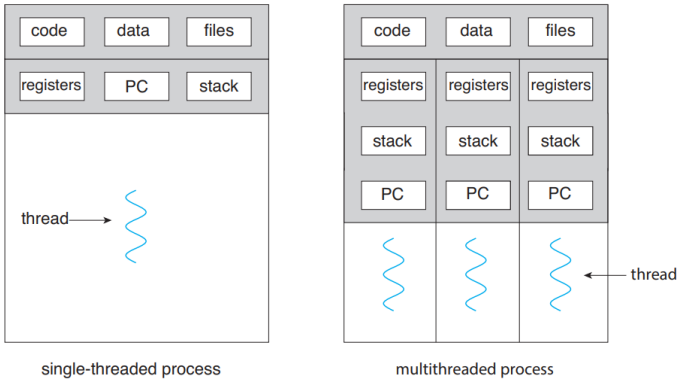
1.2.1) Concurrency Vs. parallelism :

A *concurrent* system supports more than one task by allowing all the tasks to make progress. In contrast, a *parallel* system can perform more than one task simultaneously. Thus, it is possible to have **concurrency** without **parallelism**.



Q : If create multiple threads out of same process then what are the things that threads should NOT share ? –

Registers because what if two threads are performing diff tasks like a=b & a=b+1
Stack because threads may calls functions so if they share stake then they might get interleaved
Similary for PC we cannot share that because operation or instruction gets interleaved.



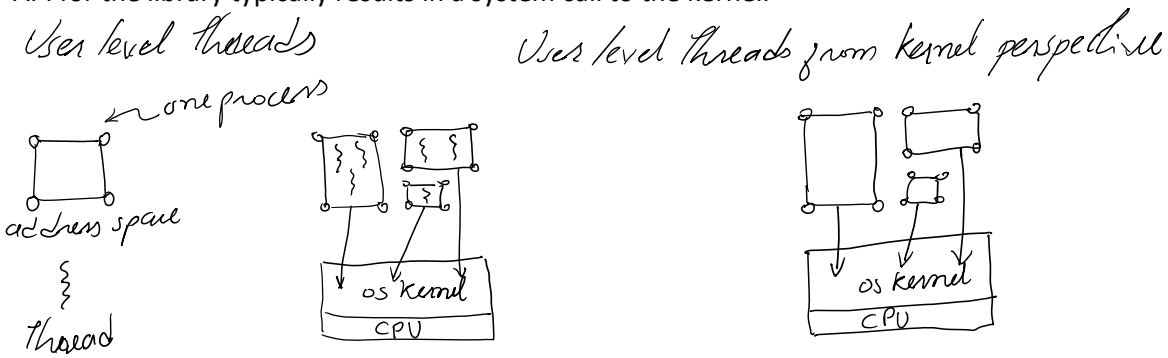
They share heap also if each thread has its own separate heap, it would require more memory overhead, and memory allocation and deallocation would be less efficient.

Q : How to initiate threads ? -

A **thread library** provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library. The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space.

This means that invoking a function in the library results in a local function call in user space and not a system call.

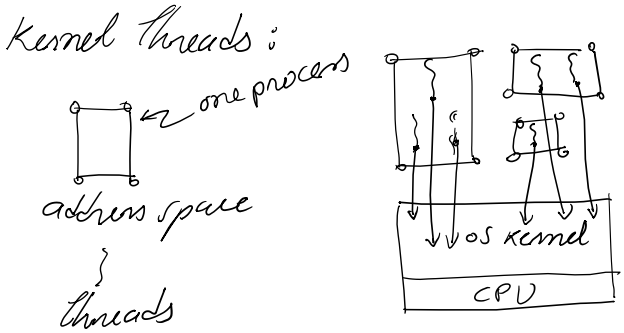
The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.



So here from kernel perspective it does not see threads we say kernel is not aware of threads. So, it executes whole process.

User-level thread advantage : They are fast since no OS involvement (no context switch and all)

Disadvantage : OS doesn't know about threads so if one threads block because of I/O then all threads get block.

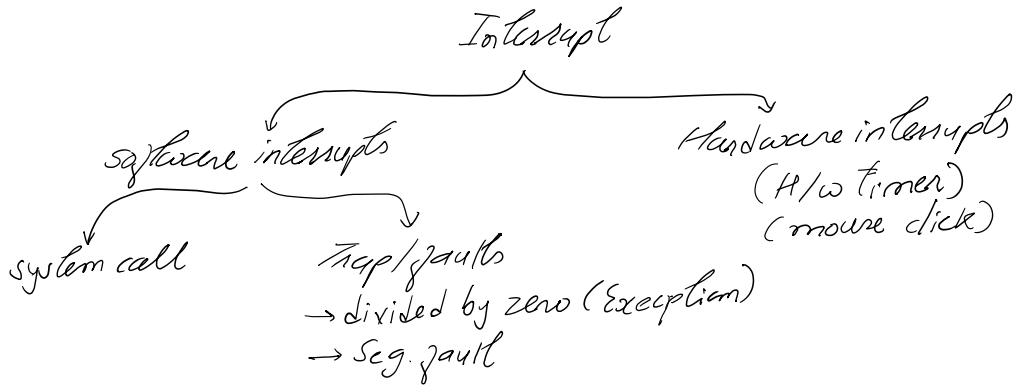


Kernel-level threads advantage : It is fast to create (because no system calls needed), Kernel doesn't have to maintain different states of different threads (user code will do it) hence, context switching is not needed.

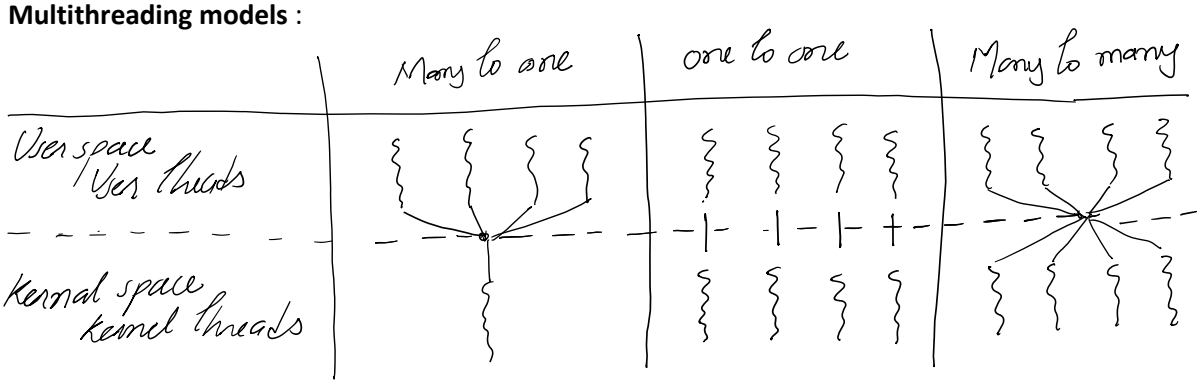
Here in kernel threads kernel is aware of thread so it can run them by some code called **schedular**. If threads are in user space then we cannot take help from kernel schedular to schedule the user level threads because kernel can't see user level threads.

Q : But as a user I might be having my own preferences to schedule so how to schedule user level threads ? – user can also have its own schedular.

Consider one scenario where P1 process is running and P2 process is also there but in waiting list. So, how to schedule the schedular itself ? – we use system call "sched" which also acts as interrupt.



Multithreading models :



Contention scope is a way of scheduling threads. It defines how a user thread is mapped to a kernel thread. There are two types of contention scope:

Process contention scope: A user thread shares a kernel thread with other user threads within the process.

System contention scope: A user thread is directly mapped to one kernel thread.

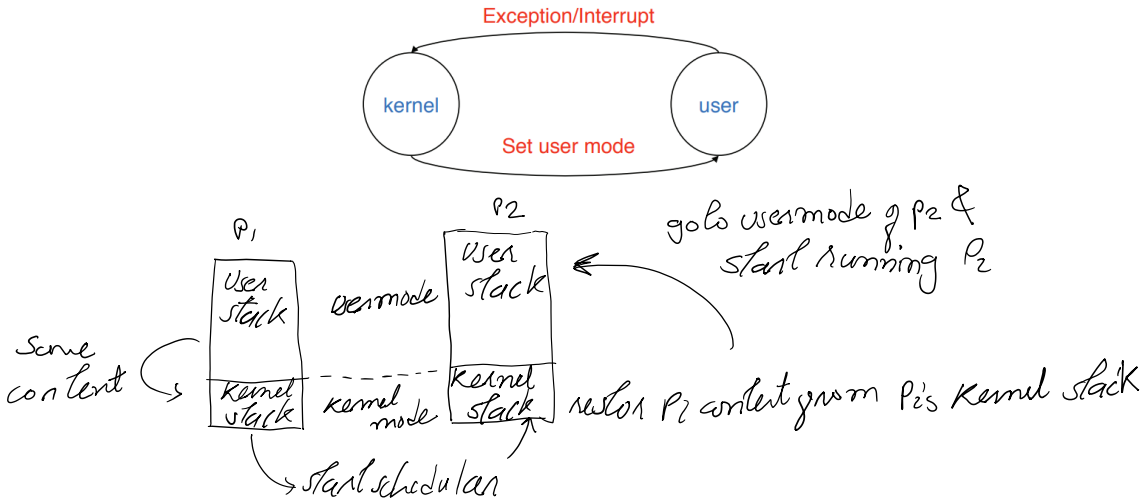
1.2.2) Context switching :

Suppose you have only one CPU and two processes P1, P2 and currently executing P1 process. And after executing few lines P1 becomes busy in taking input from user. P2 gets chance to run so now P2 cannot run in CPU directly because register of CPU may contains already calculated value of P1 so it saves the regs in memory and then after executing P2 it restores the registers from memory.

```
mov r1, #100                    mov r1, #400
mov r2, #200                    mov r2, #500
add r3, r2, r1                  sub r4, r1, r2
mov r1, #111                    mov r4, #600
mov r2, #222                    ...
...
```


This whole procedure is called **context switching**. i.e. save state of a process before a switching to another process and restore original process state when switching back. But along with register we need to store the status of previous process and all.

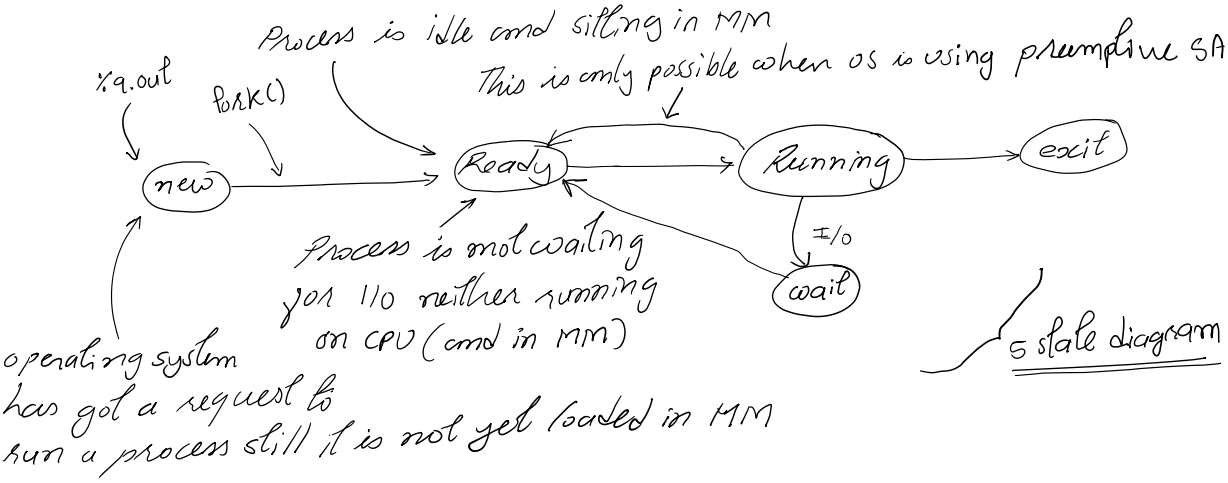
Every process has user stack and kernel stack associated with it. Every process has some information to store like PCB, register in use, we store this information in memory (in privileged area of memory). In that memory there is stack called *kernel stack*.



Context switch only happens when process is already in kernel mode

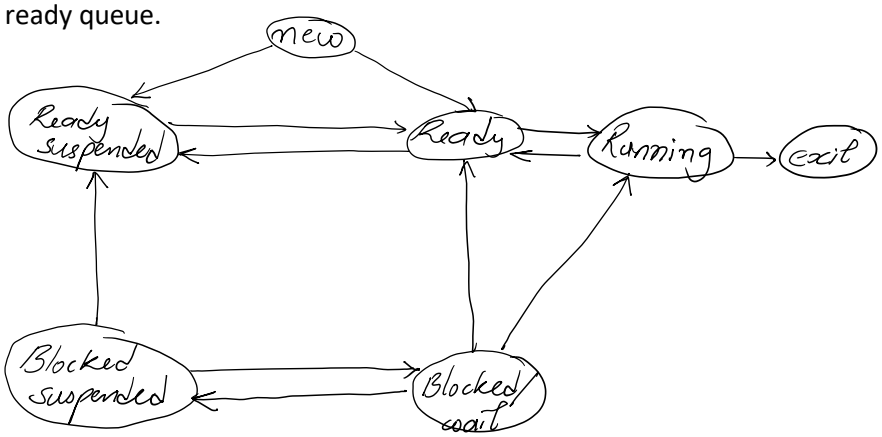
//Lecture 8

1.2.3) Process state diagram :



We can also have 7-state diagram which is basically adding two extra state to 5-state diagram. We first add waiting suspended state in which process is in hard disk and waiting to come in MM. and second state ready suspended state in which process is in hard disk and ready to execute so after acquiring permission process will jump to ready state.

When process is in waiting in hard drive then it can jump to ready suspended state if I/O is over and ready to put into ready queue.



1.3) Process scheduling :

Types of schedulers : long-term schedular, mid-term scheduler, short-term schedular

Long-term schedulers (or job schedular) – select which processes should be brought into the ready queue. It is slow and it controls the degree of multiprogramming meaning number of processes in main memory.

Short-term schedulers (or CPU scheduler) – selects which process should be executed next and allocates CPU. Sometimes the only scheduler in a system. It is frequent.

Mid-term schedulers – it is present in all system with virtual memory, temporarily removes processes from main memory and places them on secondary memory (such as disk drive) or vise versa.

Dispatcher : Another component involved in the CPU-scheduling function is the **dispatcher**. The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

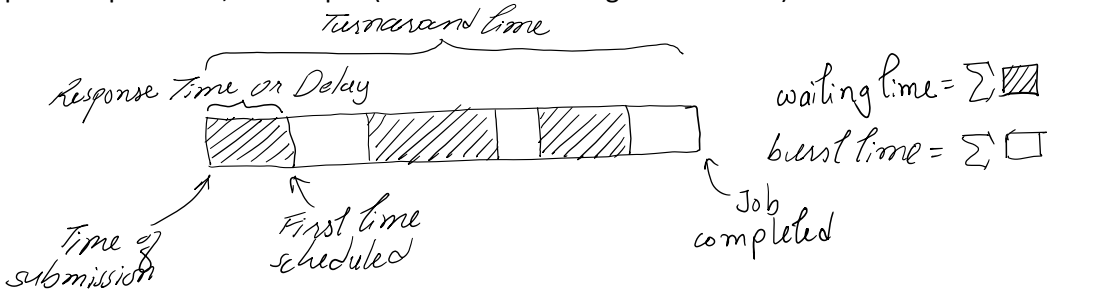
- Switching context
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**

//lecture 9

Scheduling criteria :

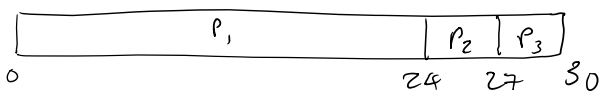
a) ***CPU utilization*** : keep the CPU as busy as possible ← maximize
b) ***Throughput*** : # of processes that complete their execution per time unit ← maximize
c) ***Turnaround time*** : amount of time to execute a particular process ← minimize
d) ***Waiting time*** : amount of time a process has been waiting in the ready queue. ← minimize
e) ***Response time*** : amount of time it takes from when a request was submitted until the first response is produced, not output (for the time-sharing environment) ← minimize



1.3.1) First-come first-serve (FCFS) : by far the simplest CPU scheduling algorithm.

Process	Burst time
P_1	24
P_2	3
P_3	3

Gantt chart:



Process	Response time	Waiting time	Turnaround time
P_1	0	0	24
P_2	24	24	27
P_3	27	27	30
Average	17	17	27

Convoy effect : All the other processes wait for the one big process to get off the CPU.

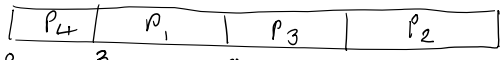
//Lecture 10

1.3.2) Shortest job first (SJF) :

To convoy effect, we can first execute shortest job first. For example,

Process	Burst time(ms)
P_1	6
P_2	8
P_3	7
P_4	3

assume arrival time = 0



Average waiting time = (3+9+16)/4 = 7ms

Comparing to FCFS avg. waiting time is 10.25ms.

Q : but it is just one exmple we can't conclude anything from one example – Moving a short process before a long one decreases the waiting time of the short process more than it increases the waiting time of the long process but this increment is very small as small burst process is running first. Consequently, the average waiting time decreases.

but there are few problems with SJF like the difficulty is knowing the length of the next CPU request. The only logical way seems to use history of previous process.

Suppose we have 5 process out of which we know burst time of first 4 processes so to predict P5 burst time we can use average of previous process. i.e. average of burst time of P1, P2, P3, P4.

$\therefore B_5 = \frac{B_1' + B_2' + B_3' + B_4'}{4}$ & similarly suppose $B_6 = \frac{B_5' + B_4' + B_3' + B_2' + B_1'}{5}$

Actual burst time of P_5 (B_5')

$\therefore B_6 = \frac{B_5' + 4B_5}{5} = \frac{1}{5}B_5' + \frac{4}{5}B_5$

B_5' → Previous prediction

B_5 → Actual time

We are giving more weight to the previous prediction and less weight to the previous real burst time. But it's not always necessary that is why we create generalized formula

Let's say $B_5' = t_5$ $\quad \therefore B_n = \alpha B_{n-1} + (1-\alpha)t_{n-1} \quad 0 \le \alpha \le 1$

Earlier we were rigid in the sense that we are always giving less weight to the recent actual time and more weight to the predicted time.

If we substitute in Bn we get,

$$B_{n+1} = (1-\alpha)t_n + \alpha(1-\alpha)t_{n-1} + \ldots + \alpha^{n+1}B_0$$

Typically, α is less than 1. As a result, (1 − α) is also less than 1, and each successive term has less weight than its predecessor. So, our predicted burst is more dependent upon the recent history than past.

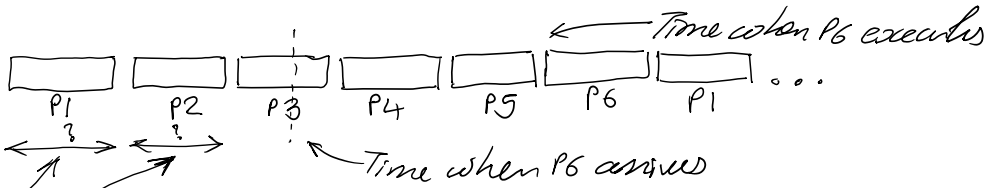
//Lecture 11

The SJF algorithm can be either preemptive or non-preemptive. The choice arises when a new process arrives at the ready queue while a previous process is till executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A preemptive SJF algorithm will preempt the currently executing process. Preemptive SJF scheduling is sometimes called **shortest-remaining time first scheduling**.

1.3.3) Round robin scheduling :

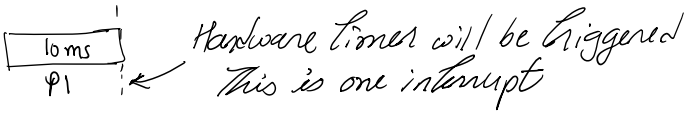
Previous scheduling algo (FCFS and SJF) are non-preemptive in nature. But round robin is preemptive. For preemption, we need to have context switch.

Q : What will happen if while executing P3 new process P6 come (assume initially 5 processes are arrived) ? –



We can say it is fair algorithm as it allows every process to execute.

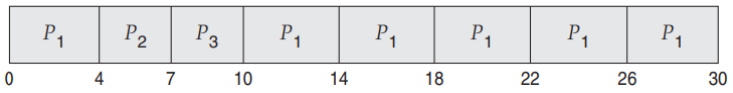
Time quantum :



Interrupt service routine will be executed this in turn will lead to short term schedular code. Generally, very fast because decision about new process to be scheduled is very easy.

Process	Burst Time
P_1	24
P_2	3
P_3	3

Time quatum : 4



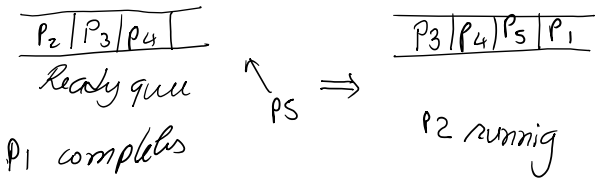
Avg. waiting = (6 + 4 + 7) / 3 = 17/3

Avg. turn around time = (30 + 7 + 10) / 3 = 47/3

Do we need context switch here ?

If ready queue does not have any process then we can bypass (skip) the context switch.

Q : What would happen when one process completes at the same time some new process comes ? –

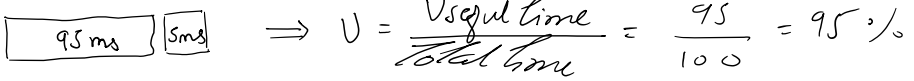


Execution of next process in round robin in nutshell – Suppose P1 is running and P1's scheduling quantum is about to expire and you have P2 process in ready queue.

1) A timer interrupt occurs
2) P1's user-level state is stored into a trap frame (in kernel space)
3) The operating system's scheduler runs

4) There is a context switch from thread T1 (from process P1) to thread T2 (from process P2).
5) P2's user-level state is restored from a trap frame.

Q : A system uses a round robin scheduling algorithm with a time slice of 95 millisecond; the context switch time is 5 milliseconds. What is the average processes utilization ? -

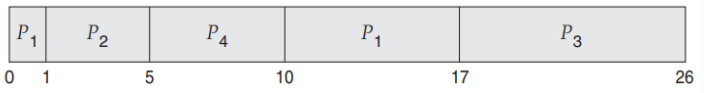


If time quantum (q) is large then scheduling is same as FIFO and if q is small then q must be large with respect to context switch other overhead is too high.

//lecture 12

1.3.4) Shortest remaining time first :

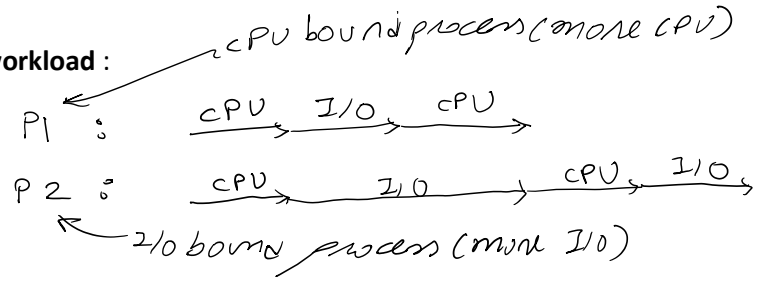
Process	Arrival Time	Burst Time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



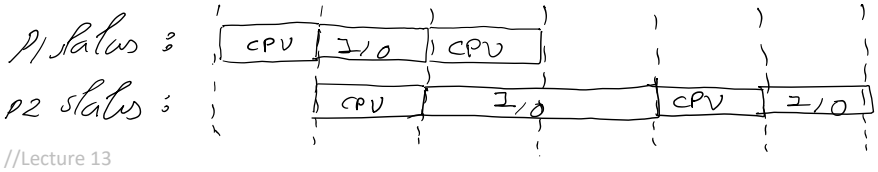
When we are executing P1 at time 1 we realized that there is a process whose time is less than the currently executed P1.

Scheduling with mixed workload :

Suppose two processes :



We run these processes in uniprocessor system.



//Lecture 13

1.3.5) Priority Scheduling :

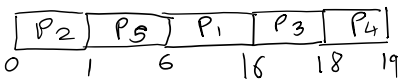
A priority number (integer) is associated with each process. The CPU is allocated to the process with the highest priority (smallest integer = highest priority) it can be preemptive or non-preemptive.

SJF is priority scheduling where priority is the inverse of predicted next CPY burst time

But there is one problem, **starvation** – *low priority processes may never execute*. Solution is **aging** – *as time progresses increase the priority of the process* (highest response ratio next is one of the ways to do it).

Non preemptive priority example :

Process	Burst time	Priority



P1	10	3
P2	1	1
P3	2	4
P4	1	5
P5	5	2

1.3.6) Priority Inversion :

A scheduling challenge arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes. As an example, assume we have three processes—L, M, and H—whose priorities follow the order L < M < H. Assume that process H requires a semaphore S, which is currently being accessed by process L. Ordinarily, process H would wait for L to finish using resource S. However, now suppose that process M becomes runnable, thereby preempting process L. Indirectly, a process with a lower priority—process M—has affected how long process H must wait for L to relinquish resource S.

Typically, priority inversion is avoided by implementing a **priority-inheritance protocol**. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. Inherit the higher priority means leave the higher priority. In above example L and H will interchange the priority. This is somewhat same as **Priority Donation** where - If a thread or process attempts to acquire a resource (lock) that is currently being held, it donates its effective priority to the holder of that resource.

There is other type of scheduling algo which is called **highest response ratio next** scheduling algo which is non-preemptive algo, **response ratio = (waiting time + burst time)/ burst time**. So, after completion of each process calculate this and apply.

Multilevel Queue Scheduling : A multilevel queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm. Each queue has absolute priority over lower-priority queues.

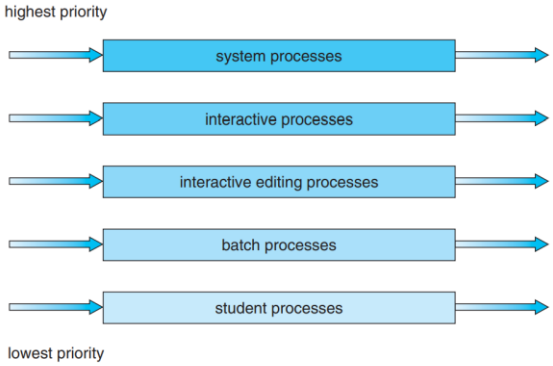


Figure 6.6 Multilevel queue scheduling.

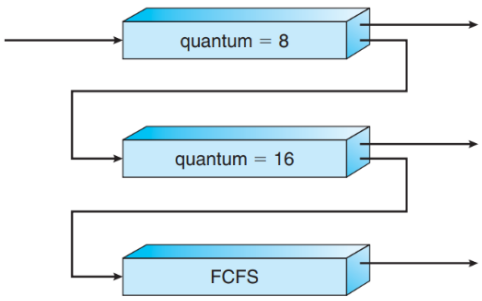


Figure 6.7 Multilevel feedback queues.

But in multilevel queue scheduling there is a possibility of starvation so we have another variation called **multilevel feedback queues scheduling**.

Multilevel feedback Queues Scheduling : The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.

In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue.

//Lecture 14

1.4) Interprocess Communication (IPC) :

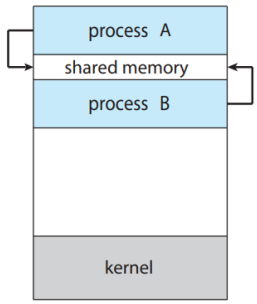
Suppose two process within a system wants to communicate. Then they will take help from OS.

Operating system provide facilities for inter-process communications (IPC), such as

- Shared memory
- Message queues
- Pipes

1.4.1) Shared memory :

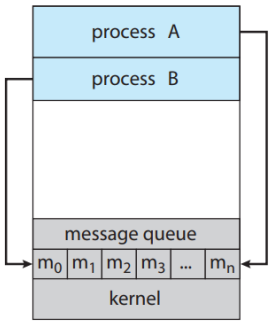
Easiest way for processes to communicate – at least some part of each process's memory region overlaps the other processes. If A wants to communicate to B, A writes into shared memory and B reads from shared memory.



To create shared memory, we need OS support but once we have the shared memory, we do not need OS support (why?) – because this memory is in user space

1.4.2) Message passing :

Shared memory created in the kernel. System calls such as send and receive used for communication.



They do not share any address space for communicating

Pipes : A pipe acts as a conduit allowing two processes to communicate. Difference between Message passing and pipes is that message queue have specific size but in pipe any stream of data having variable size is allowed to pass.

Processor Affinity : Consider what happens to cache memory when a process has been running on a specific processor. The data most recently accessed by the process populate the cache for the processor. As a result, successive memory accesses by the process are often satisfied in cache memory. Now consider what happens if the process migrates to another processor. The contents of cache memory must be invalidated for the first processor, and the cache for the second processor

must be repopulated. This is known as **processor affinity**—that is, a process has an affinity (dependency) for the processor on which it is currently running.

When an operating system has a policy of attempting to keep a process running on the same processor—but not guaranteeing that it will do so—we have a situation known as **soft affinity**. Here, the operating system will attempt to keep a process on a single processor, but it is possible for a process to migrate between processors. In contrast, some systems provide system calls that support **hard affinity**, thereby allowing a process to specify a subset of processors on which it may run.

2. Process Synchronization

//Lecture 15

If multiple processes try to read/write in shared memory then intermixing of operation may create incorrect result which may produce hazardous effect.

Suppose a producer which produces data and consumer process which consumes that data. For storing data we use bounded buffer. We assume that at most BUFFER_SIZE – 1 items or data in the buffer can be allowed. To do this we use integer variable count, initialized to 0.

```
while (true) {
     /* produce an item in next_produced */

     while (count == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     count++;
}
```

Producer code

```
while (true) {
     while (count == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     count--;

     /* consume the item in next_consumed */
}
```

Consumer code

Although the producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently. Suppose count variable has value 5 at some instance.

Note that the statement "count++" may be implemented in machine language (on a typical machine) as follows:

```
register1 = count
register1 = register1 + 1
count = register1
```

where register1 is one of the local CPU registers. Similarly, the statement "count- -" is implemented as follows:

```
register2 = count
register2 = register2 - 1
count = register2
```

where again register2 is one of the local CPU registers.

T_0: *producer* execute $register_1 = \texttt{count}$ $\{register_1 = 5\}$
T_1: *producer* execute $register_1 = register_1 + 1$ $\{register_1 = 6\}$
T_2: *consumer* execute $register_2 = \texttt{count}$ $\{register_2 = 5\}$
T_3: *consumer* execute $register_2 = register_2 - 1$ $\{register_2 = 4\}$
T_4: *producer* execute $\texttt{count} = register_1$ $\{count = 6\}$
T_5: *consumer* execute $\texttt{count} = register_2$ $\{count = 4\}$

A situation like this, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.

//Lecture 16

2.1) The Critical-section problem :

The *critical-section problem* is to design a protocol that the processes can use to synchronize their activity so as to cooperatively share data. Each process must request permission to enter its **critical section**. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

General structure of typical process :

```
while (true) {
    entry section
        critical section
    exit section
        remainder section
}
```

A solution to the critical-section problem must satisfy the following three requirements:

a) **Mutual exclusion**. If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
b) **Progress**. If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in deciding which will enter its critical section next, and this selection cannot be postponed indefinitely (indefinitely = अनिश्चित).

With above two process we can ensure that there will be exactly one process in critical section.

c) **Bounded waiting**. There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

//Lecture 17

So, to satisfy this requirement from now our objective is "for all execution, ME progress and bounded waiting satisfied" we want to come up with some set of rules that covers all possibilities.

Q : How to check these three in questions ? –

Mutual Exclusion.

- One process in critical section, another process tries to enter → show that second process will block in entry code
- Two (or more) processes are in the entry code → show that <u>at most</u> one will enter critical section. Here at most is used meaning if deadlock happens when two process are in entry code then ME will satisfy.

Progress (== absence of deadlock)

- No process in critical section, p1 arrives → p1 enters
- Two (or More) processes are in the entry code → show that <u>at least</u> one will enter critical section.

Bounded waiting (== fairness)

- One process in critical section, another process is waiting to enter → show that if first process exits the critical section and attempts to re-enter, show that waiting process will be get in.

//Lecture 18

2.1.1) Peterson's solution and hardware solutions :

Peterson's solution requires the two processes to share two data items :

```
int turn;          //I might be ready but it's okay if you execute others
boolean flag[2];   //Who's ready to enter its CS

//The structure of process pi in Peterson's solution
while(true){
    flag[i] = true;
    turn = j;
    while(flag[j] && turn == j); //If j is ready and his turn then I will wait

    /*critical section*/

    flag[i] = false;

    /*remainder section*/
}
```

ME ✓
Progress ✓
Bounded waiting ✓

Both processes can be ready to execute in its CS so this will be handled by `flag[2]` but who's going to execute CS is decided by `turn`.

Peterson's solution is software-based solution and software-based solutions are not guaranteed to work on modern computer architectures. As it is code and slow as compared to hardware.

2.1.2) Hardware solutions :

1) **Disabling interrupt** : it could be solved simply in a single-processor environment if we could prevent interrupts from occurring while a shared variable was being modified. If you disable the interrupts before entering into CS, it will disallow any kind of interrupt including context switch.

Unfortunately, this solution is not as feasible in a multiprocessor environment. And Disabling interrupts on a multiprocessor can be time consuming, since the message passed to all the process.

//Lecture 19

2) **Atomic operations** : Processors provide means to execute ready-modify-write operations atomically on a memory location. Typically applies to at most 8 bytes long variables.

Common atomic operations :

- `Test_and_set(type *ptr)` : sets *ptr to 1 and returns its previous value

```
boolean test_and_set(boolean *target) {
   boolean rv = *target;
   *target = true;

   return rv;
}
```

Figure 6.5 The definition of the atomic `test_and_set()` instruction.

```
do {
   while (test_and_set(&lock))
      ; /* do nothing */

      /* critical section */

   lock = false;

      /* remainder section */
} while (true);
```

Figure 6.6 Mutual-exclusion implementation with `test_and_set()`.

- `Fetch_and_add(type *ptr, type val)` : adds val to *ptr and returns its previous value
- `Compare_and_swap(type *ptr, type oldval, type newval)` : if *ptr == oldval , set *ptr to newval and returns true; otherwise return false.

It has number of advantages : It is applicable to any number of processes on either a single processor or multiple processors sharing main memory.

Disadvantages : As you can see in figure 6.6 ME and progress satisfied but busy waiting is employed therefore starvation is possible.

We have seen two main problems in software and hardware solution which is starvation and busy-waiting. To encounter this, we have new solution called **semaphores**.

//Lecture 20

2.2) Semaphores :

Consider Semaphores as integer variable and we can do only two types of (**atomic**) operations with it namely increment or decrement. These operations are

- P(), (or down() or Wait()) – decrement by 1, block until semaphore is open
- V(), (or up() or Signal()) – increment by 1, allow another thread to enter

And there are types of semaphores :

- **Counting semaphore** or **mutex** – can take any integer value (including negative).
- **Binary semaphore** – can only take 0 or 1.

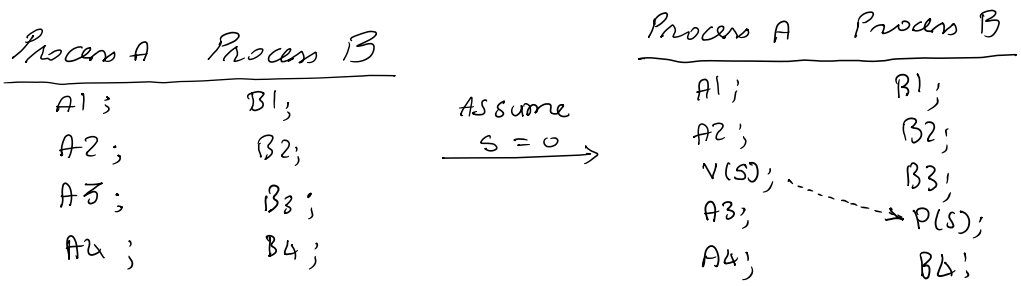
2.2.1) Binary semaphore :

Implementation :

```
wait(BinarySemaphore *S){
    if(S==0) insert calling in wait queue associated with semaphore S. block the process.
    else S = 0;
}
```

```
signal(BinarySemaphore *S){
    if(wait queue associated with S is not empty) wake up one process from the queue
    else S = 1;
}
```

Q : How to use this semaphore ? – suppose we have two processes which includes different statement A1, A2,… and we want statement A2 in process A to complete before statement B4 in process B begins.



2.2.2) Counting semaphore :

Implementation :

```
wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        sleep();
    }
}
```

```
signal(semaphore *S){
    S->value++;
    if(S->value <= 0){
        remove a process P from S->list;
        wakeup(P);
    }
}
```

The `sleep()` operation suspends the process that invokes it. The `wakeup(P)` operation resumes the execution of a suspended process P.

Meaning if S = 2 then 2 process we can allow to enter in CS and if N = -3 there are 3 blocked processes in the queue.

//Lecture 21

Q : How do we make sure that wait and signal are atomic ? – we can use test and set hardware instruction. For example,

```
void sem_wait(semaphore_t* s){
    while(test_and_set(lock));
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        sleep();
    }
    lock = 0;
}
```

critical section of wait()

But say if some process is decreasing S->value and we know that if other process comes it must stay in while loop but this should be counted in busy waiting as well so question is how long does this process2 stay in while loop ? – It is important to admit that we have not completely eliminated busy waiting with this definition of the wait() and signal() operations. Rather, we have moved busy waiting from the entry section to the critical sections of application programs. Furthermore, we have limited busy waiting to the critical sections of the wait() and signal() operations, and these sections are short (if properly coded, they should be no more than about ten instructions). So, we will experience very short amount of busy waiting or almost no waiting.

2.3) Classic Synchronization problems :

There are a number of classic problems that represent a class of synchronization situations.

2.3.1) Producer/Consumer problem :

```
while (true) {
    . . .
   /* produce an item in next_produced */
    . . .
   wait(empty);
   wait(mutex);
    . . .
   /* add next_produced to the buffer */
    . . .
   signal(mutex);
   signal(full);
}
```

Figure 7.1 The structure of the producer process.

```
while (true) {
   wait(full);
   wait(mutex);
    . . .
   /* remove an item from buffer to next_consumed */
    . . .
   signal(mutex);
   signal(empty);
    . . .
   /* consume the item in next_consumed */
    . . .
}
```

Figure 7.2 The structure of the consumer process.

In our problem, the producer and consumer processes share the following data structures:

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

We assume that the pool consists of n buffers, each capable of holding one item. The `mutex` binary semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The `empty` and `full` semaphores count the number of empty and full buffers. The semaphore empty is initialized to the value n; the semaphore `full` is initialized to the value 0.

2.3.2) Reader/writer problem :

Rules : A reader reads the data but won't change it. A writer changes the data.

Goal : Allow multiple concurrent readers but only a single writer at a time, and if a writer is active, readers wait fot it to finish.

In the solution to the first readers–writers problem, the reader processes share the following data structures :

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

```
while (true) {
   wait(rw_mutex);
    . . .
   /* writing is performed */
    . . .
   signal(rw_mutex);
}
```

Figure 7.3 The structure of a writer process.

```
while (true) {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
    . . .
   /* reading is performed */
    . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
}
```

Figure 7.4 The structure of a reader process.

//Lecture 22

2.3.3) Dining philosopher's problem :

Consider five philosophers who spend their lives thinking and eating. The philosophers share a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks (Figure 7.5).

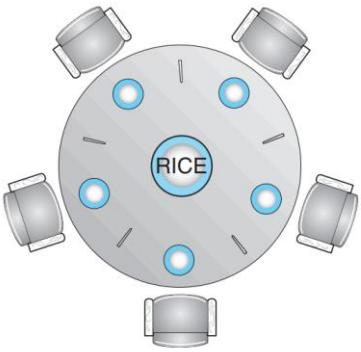


Figure 7.5 The situation of the dining philosophers.

```
while (true) {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for a while */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
}
```

Pick left stick

Pick right stick

Figure 7.6 The structure of philosopher *i*.

A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both her chopsticks at the same time, she eats without releasing the chopsticks. When she is finished eating, she puts down both chopsticks and starts thinking again.

But deadlock may occur as all 5 philosophers may pick left chopstick at the same time.

Some deadlock-free solutions :

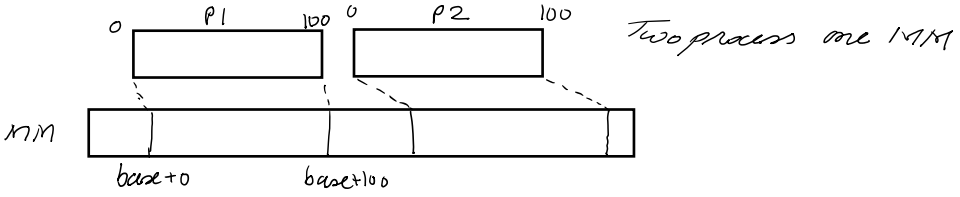
- Allow at most 4 philosophers at the same table when there are 5 resources.
- Odd philosophers pick first left then right, while even philosophers pick first right then left
- Allow a philosopher to pick up chopstick only if both are free. This requires protection of critical sections to test if both chopsticks are free before grabbing them.

3. Memory Management

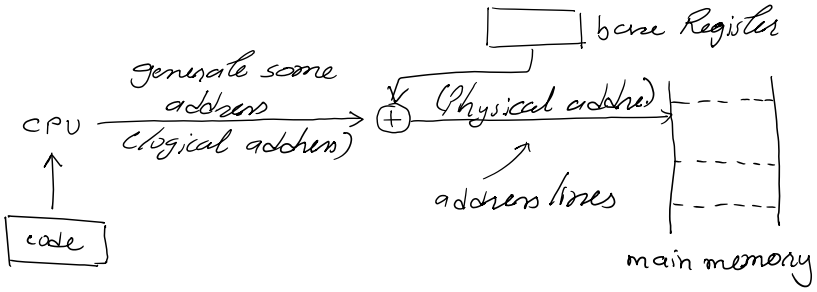
//Lecture 23

Consider a case where two programs request same memory address then we cannot have 2 programs at same time in main memory.

Relocation : on way to achieve this is to relocate program at different addresses. For example,



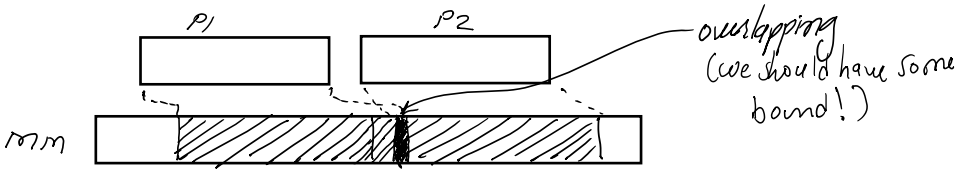
- Process is contiguously stored in main memory.
- Process start from "base" address and each process has its base address.
- Base address is maintained in hardware register called **base register**
- We can store process base address into PCB.
- At time of context switch we will (just like other register) load the base register value corresponding to the particular process.



Logical Vs. Physical address :

- Logical address –
  Generated by the CPU. And a programmer sees this.
- Physical address –
  Address sent to the RAM (or ROM, or IO) for a read or write operation.

Problem with above model what if process 1 generate some illegal address ? -



Solution : 2 hardware registers : **base** and **bound**. A process can only access physical memory in (base, base + bound) and on context switch we need to save these values.

Here in base and bound we do not need any main memory access we can simply check the addressing and allowing valid one's to access memory for use.

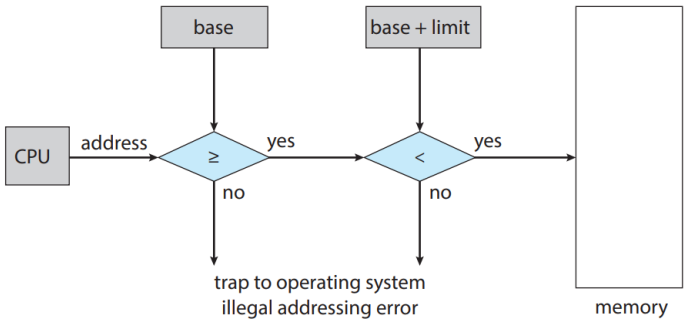
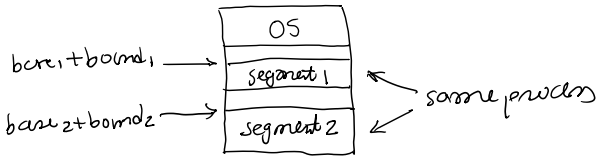


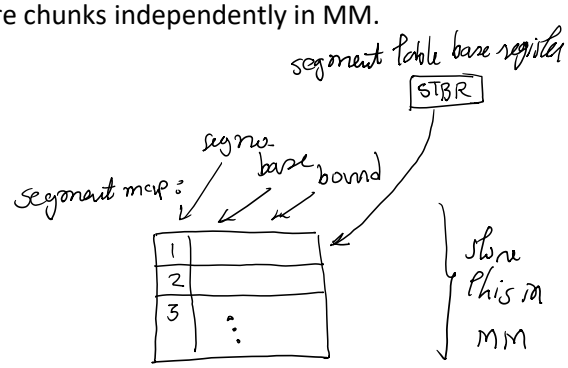
Figure 9.2 Hardware address protection with base and limit registers.

Base + Bound disadvantages : each process must be allocated *contiguously* in physical memory.

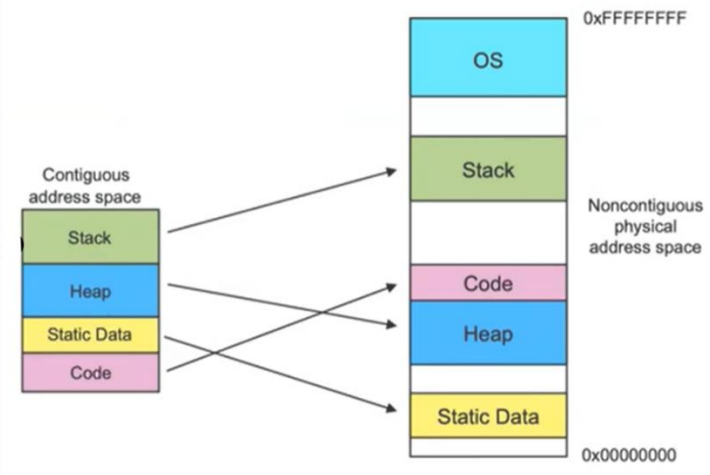
Segmentation :

- We will divide process into some chunks and store chunks independently in MM.
- **Segment** : variable-sized area of memory
- *Natural extension of base and bound* :
  Base and bound : 1 segment per process
  Segmentation : many segments per process



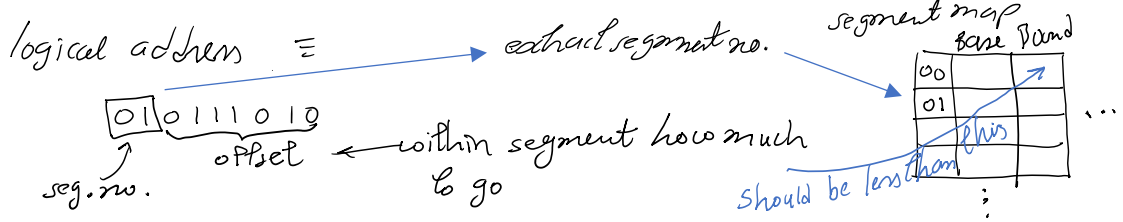


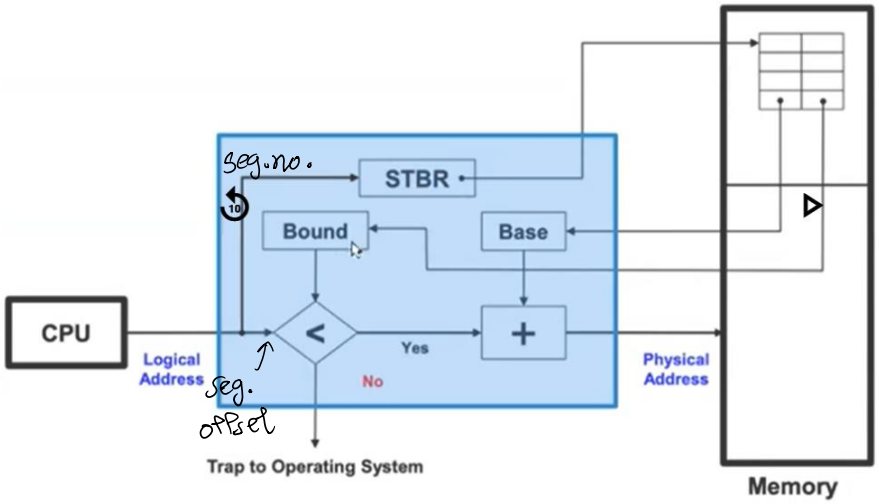
So, we will have one segment map table (one per process) which holds base and bounds registers, permissions for each segment. We also maintain segment table base register which will be different for different processes.

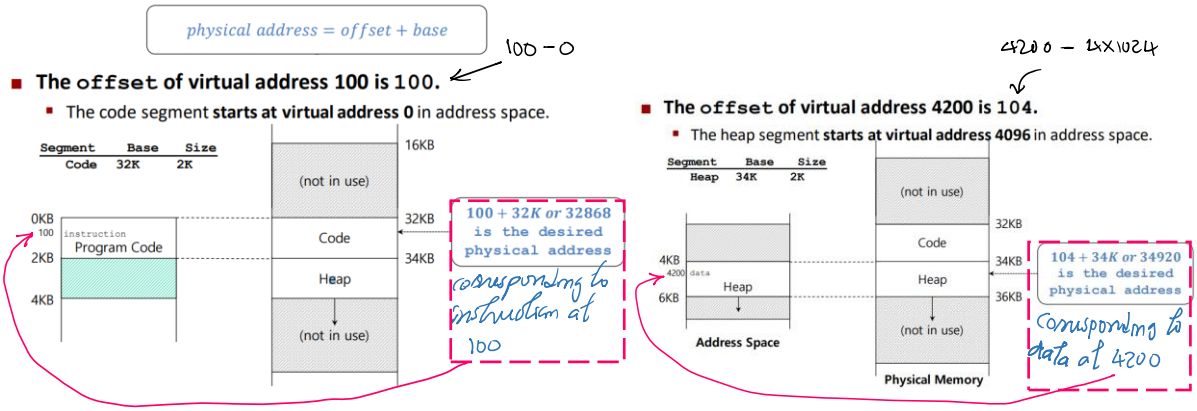


Address translation in segmentation :

We will use logical address and break them into two parts namely : **seg no**. and **offset**







Thus, a process can legally access [base, base + bound) physical address.

Problem with segments :

- Segments are somewhat inconvenient : relocating or swapping the entire process takes time.

//lecture 25

- As processes are loaded and removed from memory, the free memory space is broken into little pieces. This is called **External fragmentation** which exists when there is enough total memory space to satisfy a request but the available spaces are not contiguous.
- Scanning over free chunks to see if we can put segment there
- If the segments are large, it may be inconvenient or even impossible, to keep them in MM.

Solution :

- One solution to the problem of external fragmentation is **compaction**. The goal is to shuffle the memory contents so as to place all free memory together in one large block. Compaction is not always possible, however. If relocation is static and is done at assembly or load time, compaction cannot be done. It is possible only if relocation is dynamic and is done at execution time. If addresses are relocated dynamically, relocation requires only moving the program and data and then changing the base register to reflect the new base address. When compaction is possible, we must determine its cost. The simplest compaction algorithm is to move all processes toward one end of memory; all **holes (free space)** move in the other direction, producing one large hole of available memory. This scheme can be expensive.

- Another possible solution to the external-fragmentation problem is to permit the logical address space of processes to be noncontiguous, thus allowing a process to be allocated in physical memory wherever such memory is available. This is the strategy used in **paging.**

3.1) Paging :

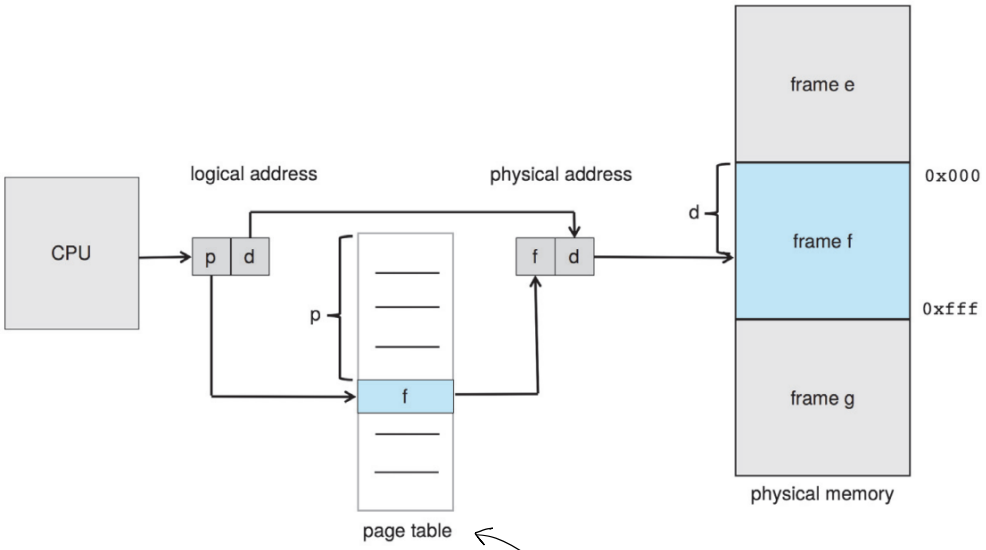
The basic method for implementing paging involves breaking physical memory into fixed-sized blocks called **frames** and breaking logical memory into blocks of the same size called **pages**.

When a process is to be executed, its pages are loaded into any available memory frames from their source (a file system or the backing store). The backing store (your program) is divided into fixed-sized blocks that are the same size as the memory frames or clusters of multiple frames.

For example, As the logical address space is totally separate from the physical address space, so a process can have a logical 64-bit address space even though the system has less than 2^{64} bytes of physical memory.

Every address generated by the CPU is divided into two parts: a page number (p) and a page offset (d):

The **page table** contains the base address of each frame in physical memory, and the offset is the location in the frame being referenced. Thus, the base address of the frame is combined with the page offset to define the physical memory address.



Address translation : page to frame

array

index (page no.)

$$Frame\ Number = Page\ table\left[\frac{Logical\ address}{page\ size}\right]$$

$$offset = logical\ address\ \%\ page\ size$$

$$Physical\ address = frame\ no. \times framesize + offset$$

Some common number used in calculation :

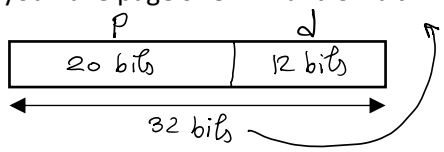
$$1KB = 2^{10}\ Bytes \qquad 1GB = 2^{30}\ Bytes$$

$$1MB = 2^{20}\ Bytes \qquad 1TB = 2^{40}\ Bytes$$

All pages are of same size. In above formula we have divided logical address by page size this is same as making group of page size up to logical address (note that logical address is just a number from 0). And taking mod of logical address by page size gives us relative position of that address in that particular page which is nothing but offset.

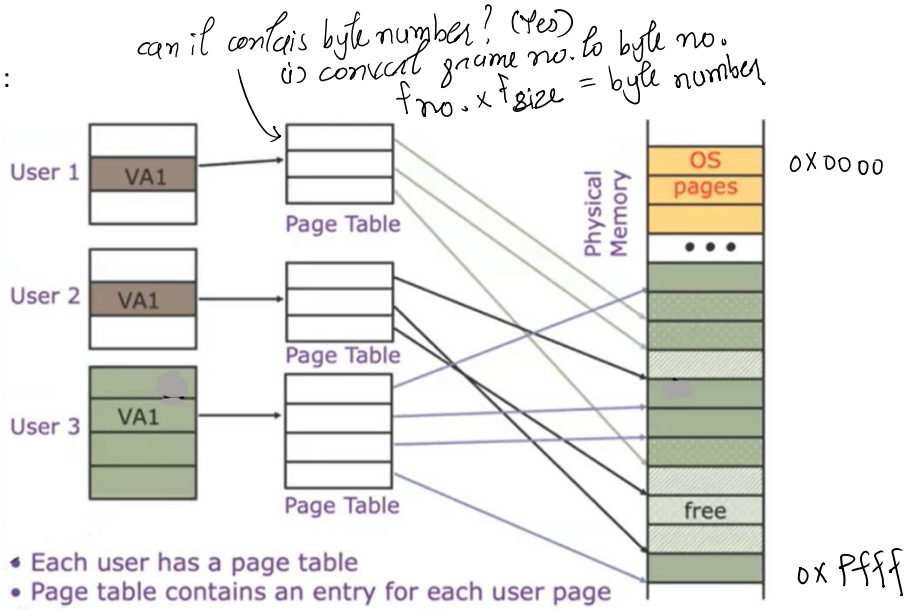
You can observe one thing, suppose you have page size 4KB and 32 bit LA so,

$$\frac{2^{32}}{4KB} = \frac{2^{32}}{2^{12}} = 2^{20}\ pages$$



//Lecture 26

About page table :



Q : What other info is in PTE besides PFN ? –

Valid bit : indicating whether the particular translation is valid.

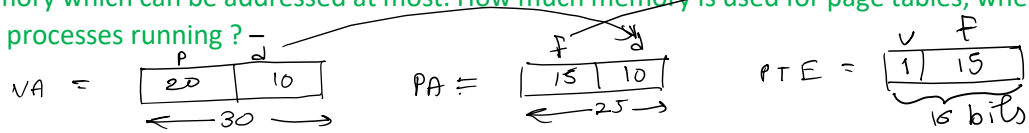
Protection bit : indicating whether the page could be read from, written to, or executed from.

Present bit : indicating whether this page is in a physical memory or on disk (swapped out).

Dirty bit : indicating whether the page has been modified since it was brought into memory.

Reference bit (accessed bit) : indicating that a page has been accessed.

Q : A system uses paging to implement virtual memory. Specifically, it uses a simple linear page table. The virtual address is of size 1GB (30 bits); the page size is 1KB; each page table entry holds only a valid bit and the resulting page frame number; the system has a maximum of 32MB of physical memory which can be addressed at most. How much memory is used for page tables, when there are 100 processes running ? –



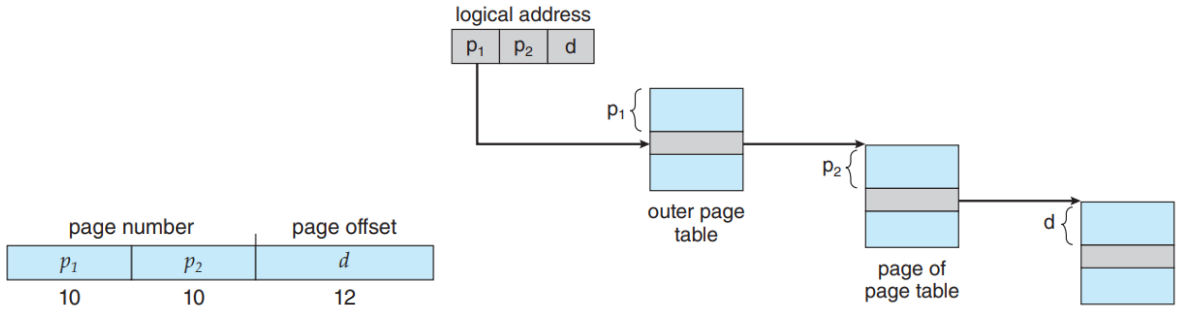
Size of page table is no. of entry × size of PTE = $2^{20} \times 16$ bytes. But this is wrong. No. of entry in PT is just a number and size of PTE is in bit not in bytes so correct answer should be 2^{21} bytes or 2^{24} bits.

Here you can see even if we have 100 processes running, we consider PT size including all pages (2^{20} > 100). Which means we have too much invalid PTE. So, how to avoid storing invalid PTE ?

Multi-level page tables !

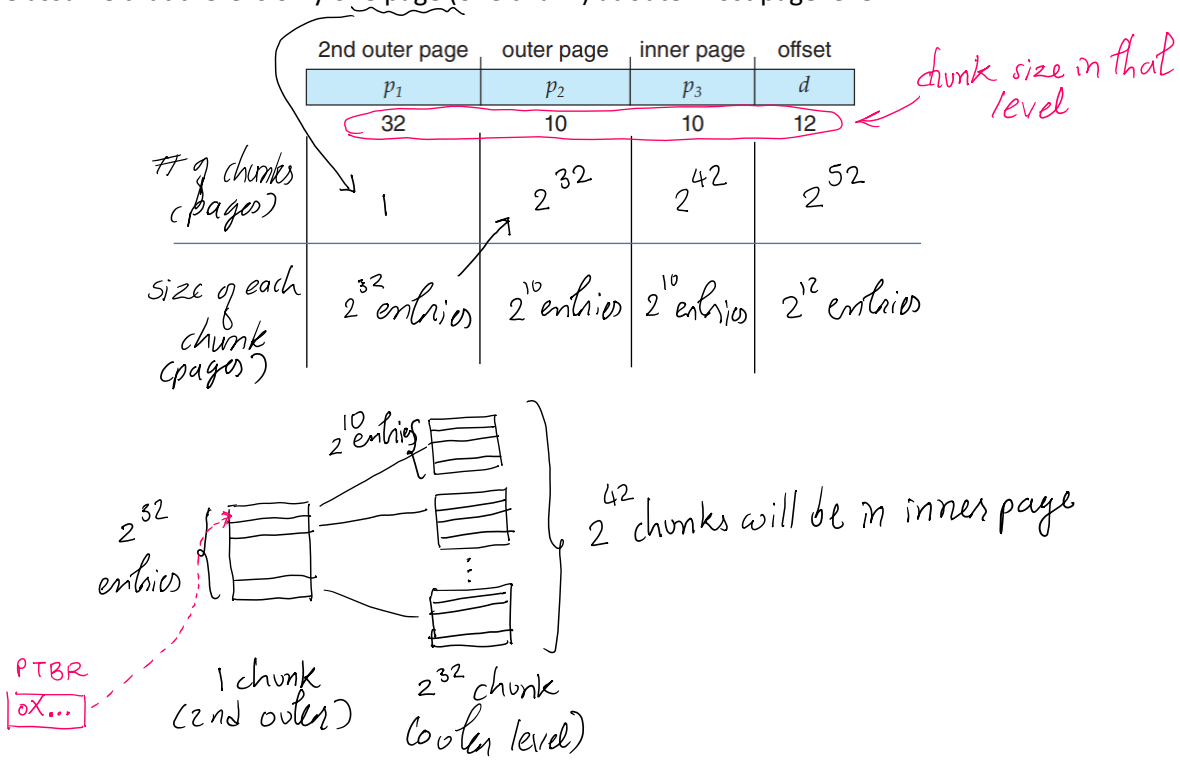
3.1.1) Multilevel paging :

One way is to use a two-level paging algorithm, in which the page table itself is also paged. A logical address is divided into a page number consisting of 20 bits and a page offset consisting of 12 bits. Because we page the page table, the page number is further divided into a 10-bit page number and a 10-bit page offset. Thus, a logical address is as follows:

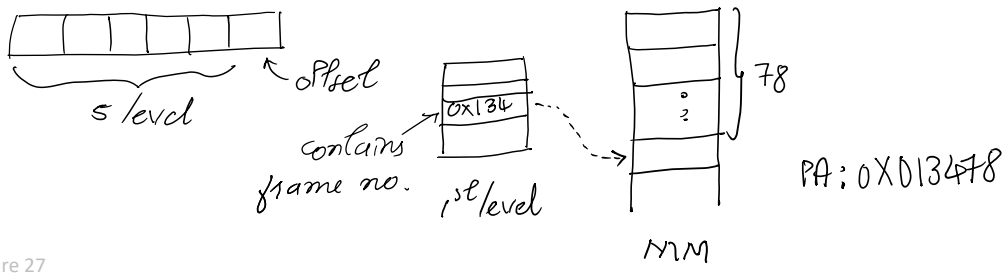


where p1 is an index into the outer page table and p2 is the displacement within the page of the inner page table. Because address translation works from the outer page table inward, this scheme is also known as a **forward-mapped page table**.

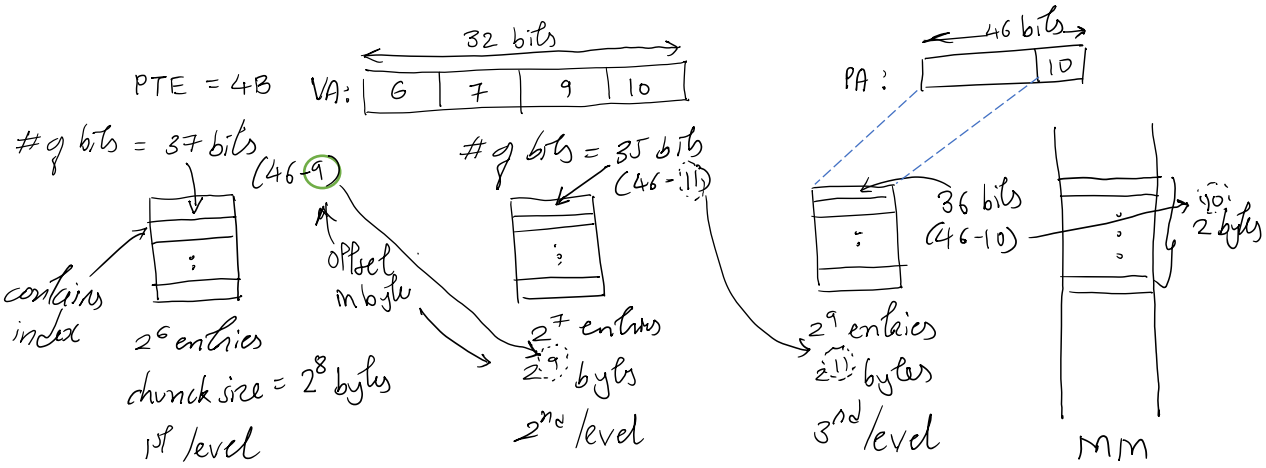
We assume that there is only one page (one chunk) at outermost page level.



Q : Suppose we have 32 bits of VA and value of address is 0x12345678, and we are using 5 level page tables at the innermost page table. The frame no. is 0x0134 then calculate PA if size of page is 2^8 bytes.
– here size of page is 2^8 bytes meaning offset is 2^8 bytes or taking last 2 bytes of VA which is 78.



//Lecture 27

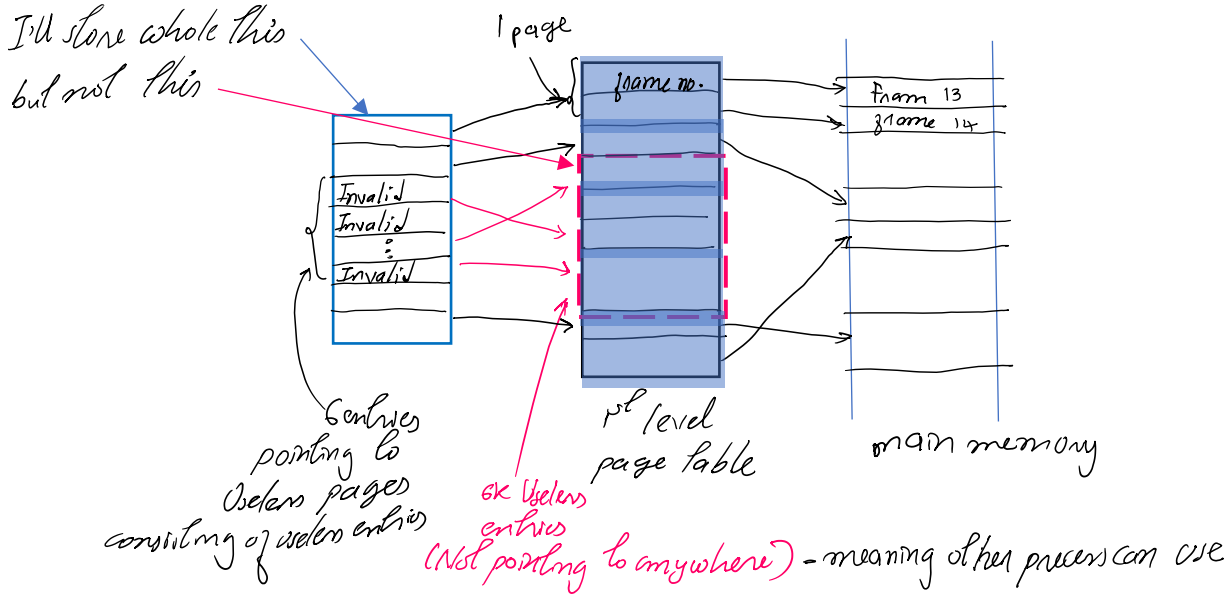


//Lecture 28

We know that we are storing this page table into main memory only. But

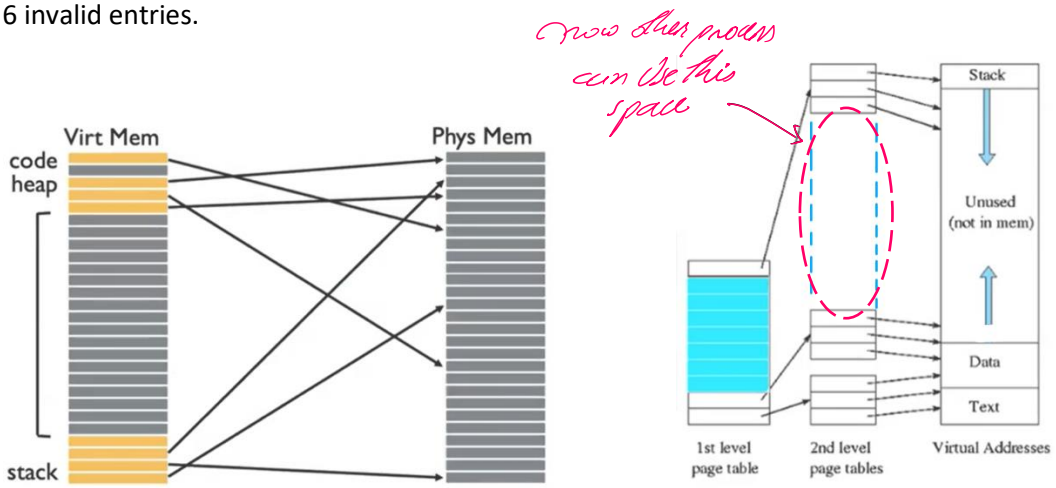
Q : How to store Page table into physical memory if memory itself is divided into frames of larger size ? – we can again divide the page table and then store in main memory.

But where is the saving ? It seems we have added one more page table and not saving any space.



In multilevel paging we have option to not store pink memory. But in single level paging we have to store those useless pages (consists of entries).

In multilevel paging, instead of storing all 6k invalid entries (like in case of single level) we just store outer 6 invalid entries.

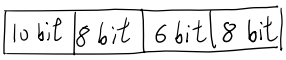


<u>If we do not divide page table (single level page table) then we do not have any choice but to store all entries (including invalid one's). but if we divide the page table (multilevel) into multiple pages then we have choice to save chunks independently hence we can avoid saving invalid entry pages into physical memory</u>.

How much space Im saving with the help of multilevel paging ? to answer this we solve following que.

//Lecture 29

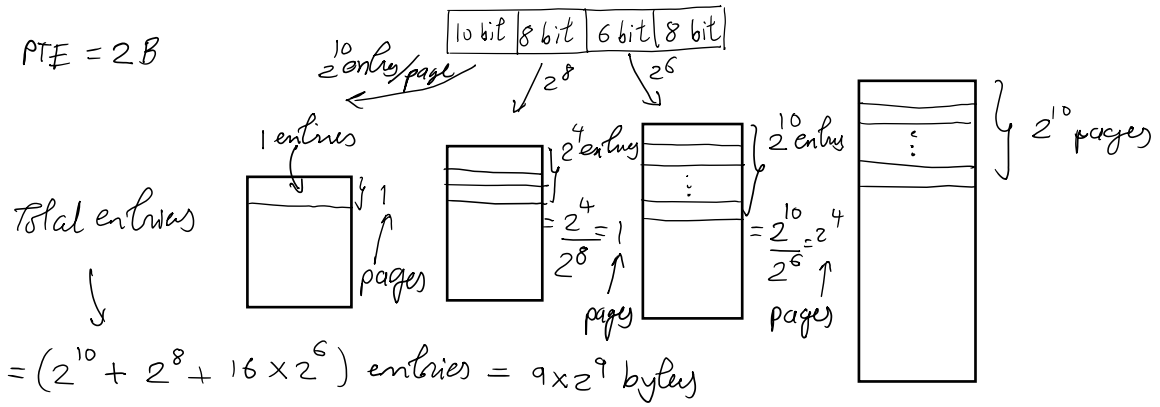
Q : In a 32-bit machine we subdivide the virtual address into 4 segments as follows :



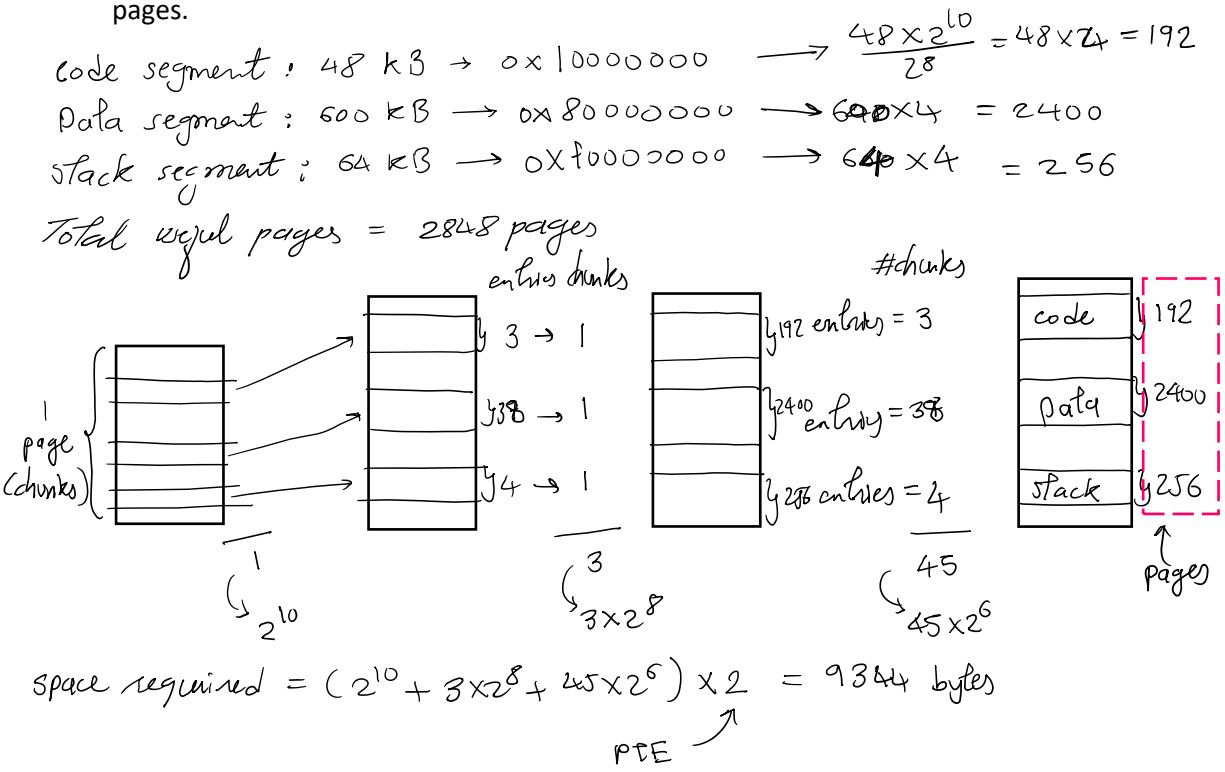
We use 3-level page table, such that the first 10-bit for the first level and so on. What is the page size in such a system ? What is the size of a page table for a process that has 256K of memory starting at address 0 ? What is the size of a page table for a process that has a code segment of 48K starting at address 0x10000000, a data segment of 600K starting at address 0x80000000 and a stack segment of 64K starting at address 0xf00000000 and growing upward ? –

- Size of a page = 2^8 bytes.
- A process has 256K of memory which means it is a final memory which we have to store in main memory. As we are paging here, we divide $\frac{2^{18}}{2^8}$. This much amount of frame is present in main memory. Which means only 2^{10} useful pages or frame in main memory.

Which means there should be 2^{10} entries should be there on 1st level page table. And so on..



- As all sections of code is scattered across the main memory, we need to calculate total no. of pages.



//Lecture 30

Advantages :

- No external fragmentation
- Fast to allocate (no searching for space) and free (no coalescing)
- Simple to adjust what subset is mapped in core (later)

Disadvantages :

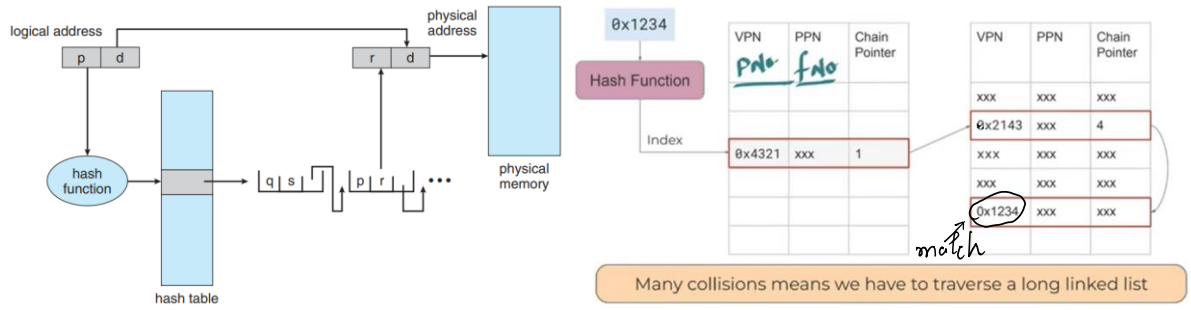
- Additional memory reference to page table (hint : use a cache we will see later)
- Internal fragmentation : unused memory that is internal to a partition.
- Required space for page table may be substantial.

NOTE : Every different process has its own multipage structure. Meaning P1 process has its own second and first level paging and P2 process has its own second as well as first level paging.

3.1.2) Structure of page table :

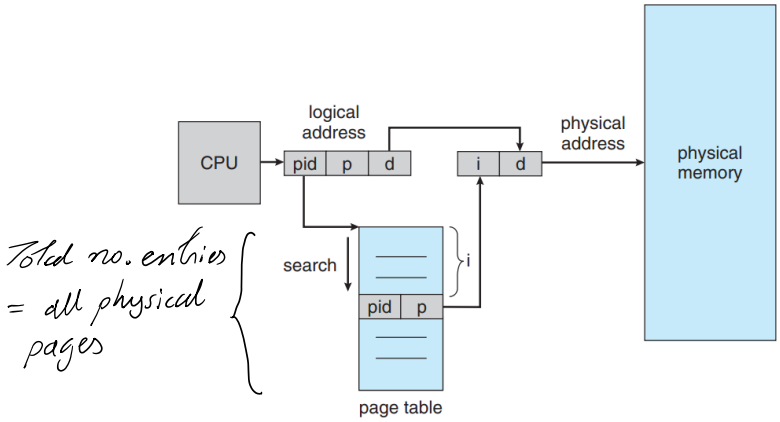
Till now we saw one structure of page table also known as Hierarchy paging (multilevel paging). Now, we will explore some more structure of page table.

1) **Hashed page table** : One approach for handling address spaces larger than 32 bits is to use a hashed page table, with the hash value being the virtual page number.



A variation of this scheme that is useful for 64-bit address spaces has been proposed. This variation uses **clustered page tables**, which are similar to hashed page tables except that each entry in the hash table refers to several pages (such as 16) rather than a single page. Therefore, a single page-table entry can store the mappings for multiple physical-page frames. *Clustered page tables* are particularly useful for **sparse** address spaces, where memory references are *noncontiguous* and *scattered* throughout the address space.

2) Inverted page table :



One global page table : One-page table entry per physical page.

Entries keyed by PID and virtual page number. Physical frame number = index in page table.

Advantages : bounded amount of memory for page table(s)

Disadvantages : Costly translation. To avoid this, we can use hashing $h: page \times pid \rightarrow PT\ entry\ (or\ chain\ of\ entries)$.

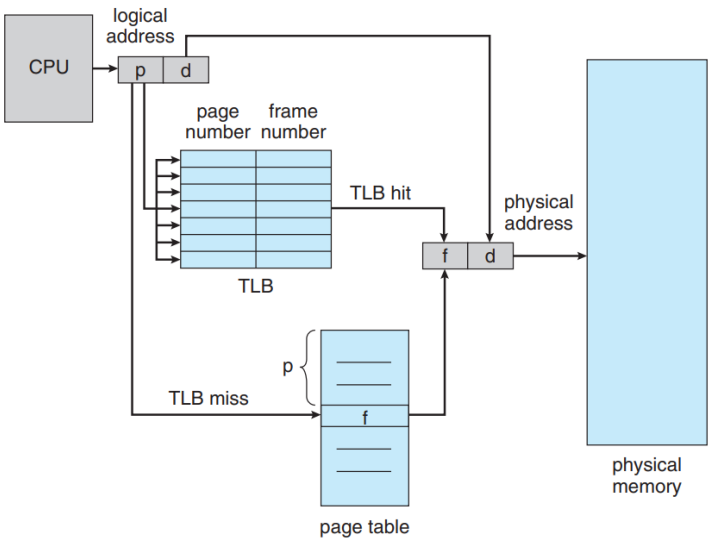
//Lecture 33

3.2) TLB and Dynamic allocation :

3.2.1) Translation lookaside buffer :

With prior scheme, two memory accesses are needed to access data (one for the page-table entry and one for the actual data). Thus, memory access is slowed by a factor of 2, a delay that is considered intolerable under most circumstances. The standard solution to this problem is to use a special, small, fast-lookup hardware cache called a **translation look-aside buffer** (TLB). The TLB is associative, high-speed memory.

Each entry in the TLB consists of two parts: a key (or tag) and a value.



Localities : programs tend to use data and instructions with addresses near or equal to those they have used recently

- ***Temporal locality*** : Recently referenced items are likely to be referenced again in the near future.
- ***Spatial locality*** : Items with nearby addresses tend to be referenced close together in time.

Q : how many TLB lookups ? – Assume 4KB pages

```
int sum = 0;
for(int i = 0;i<1024;i++){
    sum += a[i];
}
```

size of array = $4 \times 1024 = 2^{12}$ bytes

no. of pages (interval) = $\frac{2^{12}}{2^{12}} = 1$

No. of TLB miss = 1 or 2 because if data is in continuous fashion then we need to add just one page but if data are spanned across the two pages then we need two access to main memory.

3.2.2) Dynamic allocation :

In general, as mentioned, the memory blocks available comprise a set of holes of various sizes scattered throughout memory. **Dynamic storage allocation problem**, which concerns how to satisfy a request of size n from a list of free holes. There are many solutions to this problem. The first-fit, best-fit, and worst-fit strategies are the ones most commonly used to select a free hole from the set of available holes.

First fit. Allocate the first hole that is big enough. Searching can start either at the beginning of the set of holes or at the location where the previous first-fit search ended. We can stop searching as soon as we find a free hole that is large enough.

Best fit . Allocate the smallest hole that is big enough. We must search the entire list, unless the list is ordered by size. This strategy produces the smallest leftover hole.

Worst fit. Allocate the largest hole. Again, we must search the entire list, unless it is sorted by size. This strategy produces the largest leftover hole, which may be more useful than the smaller leftover hole from a best-fit approach.

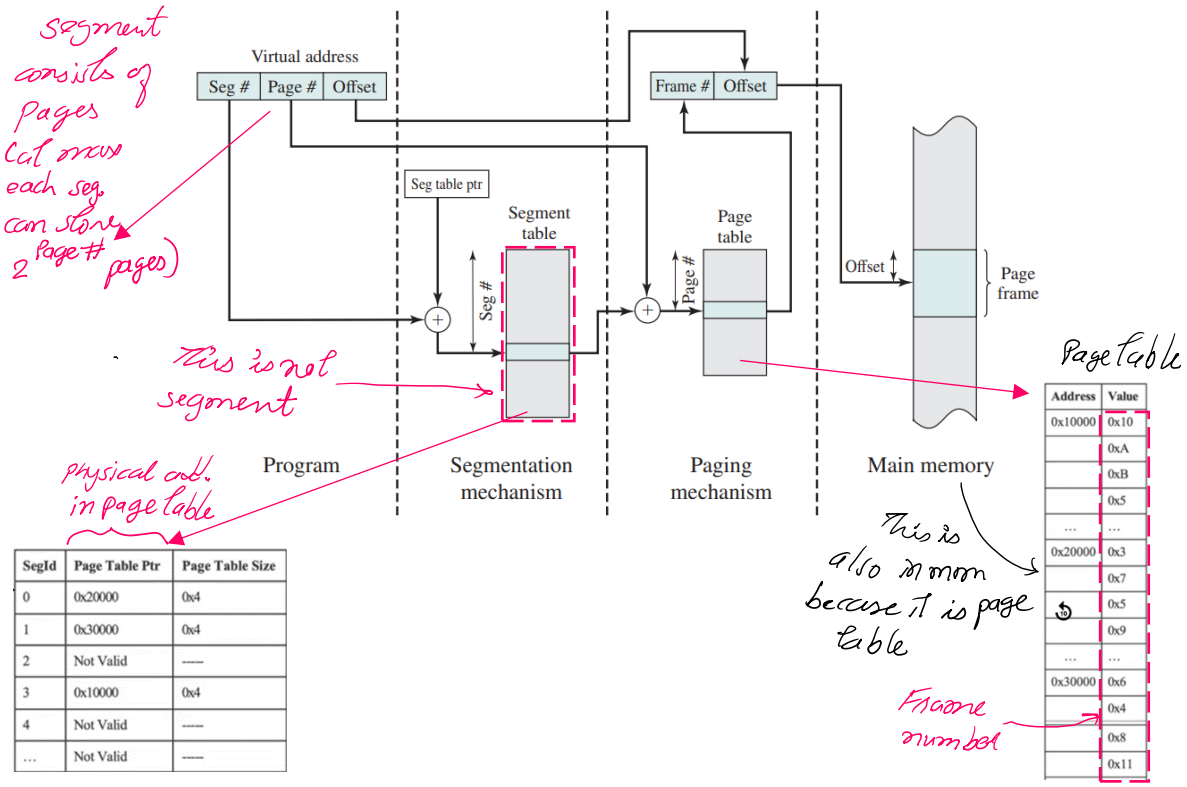
Next fit. Allocate the next hole large enough, searching from the last allocated block.

//Lecture 31

3.3) Segmentation with paging, demand paging and page replacement algo :

3.3.1) Segmentation with paging :

Both paging and segmentation have their strengths. Paging, which is transparent to the programmer, eliminates external fragmentation and thus provides efficient use of main memory. Segmentation, which is visible to the programmer, has the strengths listed earlier, including the ability to handle growing data structures, modularity, and support for sharing and protection.



The segment table entry and page table entry formats. As before, the segment table entry contains the length of the segment. It also contains a base field, which now refers to a page table (which is in main memory so whatever addresses is there in segment table corresponds to page table is physical address). Each page number is mapped into a corresponding frame number if the page is present in main memory.

//Lecture 32 – Virtual memory starts now

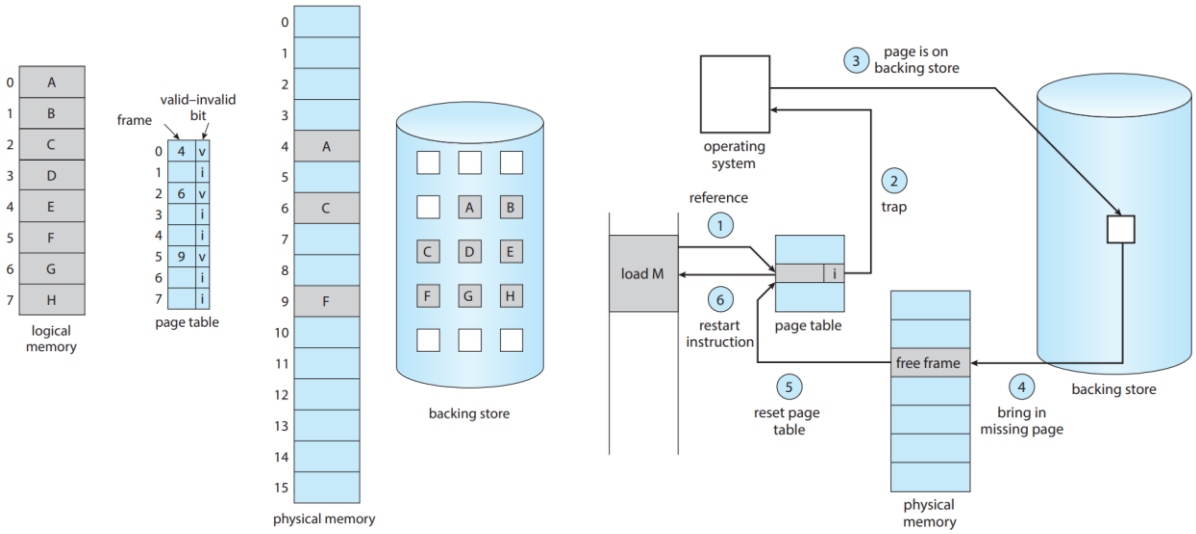
Virtual memory is a technique that allows the execution of processes that are not completely in memory. One major advantage of this scheme is that programs can be larger than physical memory.

3.3.2) Demand paging :

Paging was to divide memory into fixed-sized pages, map to frames. Up till now, we assumed it was all in memory.

So, if we need 50 pages to run the job then we need 50 available frames otherwise we can't run job.

An alternative strategy is to load pages only as they are needed. This technique is known as **demand paging** and is commonly used in virtual memory systems.



While a process is executing, some pages will be in memory, and some will be in secondary storage.

About v/i bit : When the bit is set to "valid," the associated page is both legal and in memory. If the bit is set to "invalid," the page either is not valid (that is, not in the logical address space of the process) or is valid but is currently in secondary storage. The page-table entry for a page that is brought into memory is set as usual, but the page-table entry for a page that is not currently in memory is simply marked invalid.

But what happens if the process tries to access a page that was not brought into memory? Access to a page marked invalid causes a **page fault**.

Demand paging	***Pure demand paging***
Keep some pages in physical memory and some pages in secondary memory. Bring page from SM to MM whenever needed.	Start with 0 pages in MM, keep all pages in SM initially. Bring page from SM to MM whenever needed.

Performance of demand paging :

Effective access time for demand-paged memory can be computed as :

effective access time = (1 – p) × ma + p × page fault time

where, p = probability that page fault will occur

ma = memory access time

Problem with demand paging : what if after we have selected frame from backing store, we come to physical memory and found that there is no free frame in physical memory due to increase in degree of multiprogramming, we are over-allocating memory. In such case we use page replacement algorithm.

3.3.3) Page replacement algorithm :

If there is no free frame, use a page-replacement algorithm to select a **victim frame**.

Page replacement – find some page in memory, but not really in use, page it out.

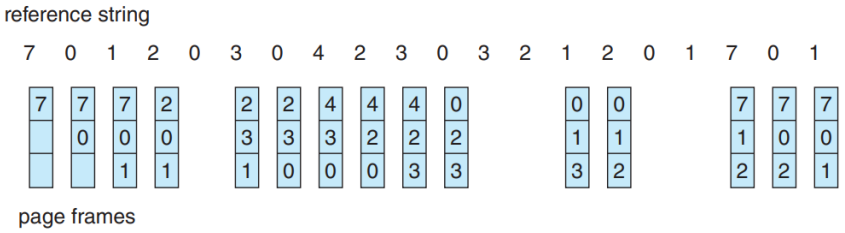
In Johny man implementation, we simply swap victim page with frame which came from SM.

Notice that, if no frames are free, two-page transfers (one for the page-out (MM to SM) and one for the page-in (SM to MM)) are required. This situation effectively doubles the page-fault service time and increases the effective access time accordingly. We can reduce this overhead by using a **modify bit** (or **dirty bit**). When this scheme is used, each page or frame has a modify bit associated with it in the hardware. The modify bit for a page is set by the hardware whenever any byte in the page is written into, indicating that the page has been modified. When we select a page for replacement, we examine its modify bit.

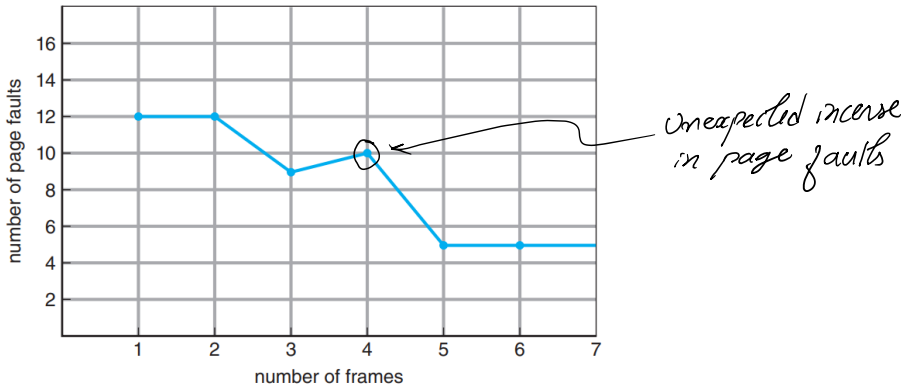
If the bit is set, we know that this page has been modified so we must write it back to SM. But if bit is not set, which means we have not touched page till now so it is safe to replace it without doing (MM to SM) transfer. i.e. we can simply overwrite page from SM.

We evaluate an algorithm by running it on a particular string of memory references and computing the number of page faults. The string of memory references is called a **reference string**.

1) **FIFO page replacement** :



To illustrate the problems that are possible with a FIFO page-replacement algorithm, consider the following reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

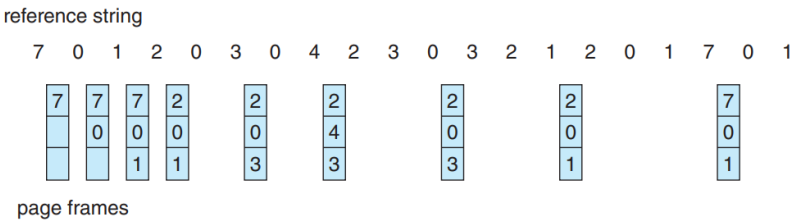


Belady's anomaly: for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.

2) **Optimal page replacement** :

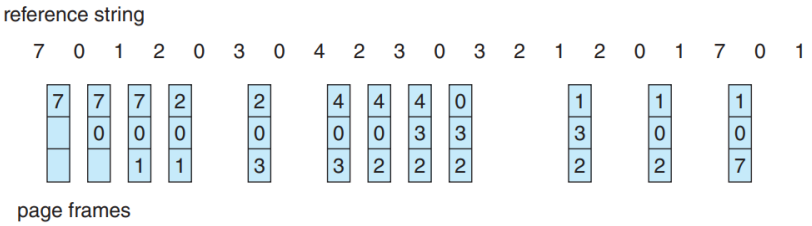
The algorithm that has the lowest page-fault rate of all algorithms and will never suffer from Belady's anomaly.

Replace the page that will not be used for the longest period of time.



3) LIFO page replacement :

If we use the recent past as an approximation of the near future, then we can replace the page that has not been used for the longest period of time. This approach is the least recently used (LRU) algorithm.



4) Counting-based page replacement :

The **least frequently used** (LFU) page-replacement algorithm requires that the page with the smallest count be replaced.

The **most frequently used** (MFU) page-replacement algorithm is based on the argument that the page with the smallest count was probably just brought in and has yet to be used.

The minimum number of page frames that must be allocated to a running process in a virtual memory environment is determined by instruction set architecture and maximum number of page frames that must be allocated to a running process in a virtual memory environment is determined by process's address space.

//Galvin

3.3.4) Thrashing :

If the process does not have the number of frames it needs to support pages in active use. At this point it must replace some page. However, since all its pages are in active use, it must replace a page that will be needed again right away. This high paging activity is called **thrashing**.

A process is thrashing if it is spending more time paging than executing

Working-Set Model : As mentioned, the working-set model is based on the assumption of locality. This model uses a parameter, Δ, to define the **working-set window**. The idea is to examine the most recent Δ page references. The set of pages in the most recent Δ page references is the working set (Figure 9.20). For example, given the sequence of memory references shown in Figure 9.20, if Δ = 10 memory references, then the working set at time t1 is {1, 2, 5, 6, 7}. By time t2, the working set has changed to {3, 4}.

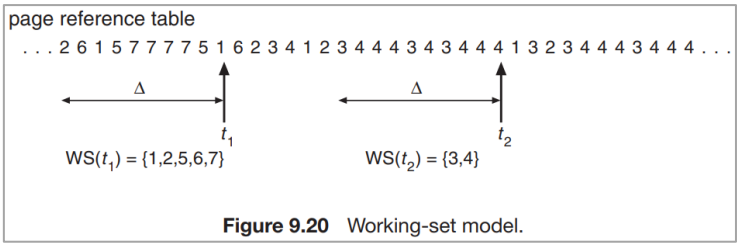


Figure 9.20 Working-set model.

If we compute the working-set size, WSSi , for each process in the system, we can then consider that D = ∑ WSSi , where D is the total demand for frames. Each process is actively using the pages in its working set. Thus, process i needs WSSi frames. If the total demand is greater than the total number of available frames (D > m), thrashing will occur, because some processes will not have enough frames.

//Lecture 34

3.4) Deadlock :

Every process in a set of processes is waiting for an event that can be caused only by another process in the set

Livelock is another form of liveness failure. It is similar to deadlock; deadlock occurs when every thread in a set is blocked waiting for an event that can be caused only by another thread in the set, livelock occurs when a thread continuously attempts an action that fails.

3.4.1) Deadlock characteristics :

1) **Four necessary conditions for deadlock** :

- ***Mutual exclusion***. At least one resource must be held in a non-sharable mode; that is, only one thread at a time can use the resource. If another thread requests that resource, the requesting thread must be delayed until the resource has been released.
- ***Hold and wait***. A thread must be holding at least one resource and waiting to acquire additional resources that are currently being held by other threads.
- ***No preemption***. Resources cannot be preempted; that is, a resource can be released only voluntarily by the thread holding it, after that thread has completed its task.
- ***Circular wait***. Processes waiting for resources in circle.

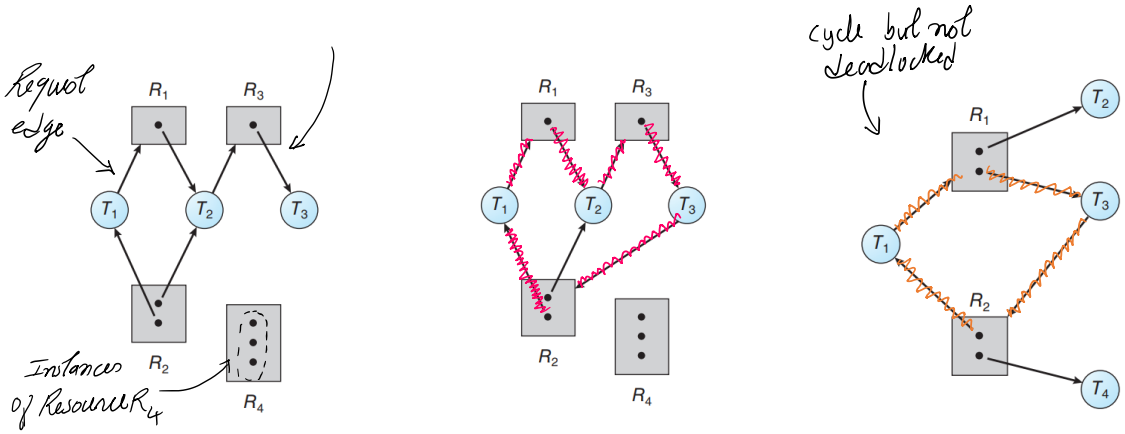
Having these 4 conditions → deadlock may or may not occur

Not having any of the conditions → deadlock will never occur

2) **Resource-allocation graph** :

Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation graph**. This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes: T = {T1, T2, ..., Tn}, the set consisting of all the active threads in the system, and R = {R1, R2, ..., Rm}, the set consisting of all resource types in the system.

A directed edge from thread Ti to resource type Rj is denoted by Ti → Rj; it signifies that thread Ti has requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type Rj to thread Ti is denoted by Rj → Ti ; it signifies that an instance of resource type Rj has been allocated to thread Ti . A directed edge Ti → Rj is called a **request edge**; a directed edge Rj → Ti is called an **assignment edge**.



At this point, two minimal cycles exist in the system: T1 → R1 → T2 → R3 → T3 → R2 → T1

T2 → R3 → T3 → R2 → T2

In second example, we also have a cycle: T1 → R1 → T3 → R2 → T1

In short, *If graph contains no cycles → no deadlock*

If graph contains a cycle → if only one instance per resource type, then deadlock

If graph contains a cycle → if several instances per resource type, possibility of deadlock.

Relationship between number of processes and number of resources for deadlock to not to occur :

Let there are n processes and Si : max resources any ith process need

$$\sum_{i=1}^{n}(S_i - 1) + 1 \leq total\ resource$$

3) Methods of handling deadlock :

Generally speaking, we can deal with the deadlock problem in one of three ways:

- We can ignore the problem altogether and pretend that deadlocks never occur in the system.
- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlocked state.
- We can allow the system to enter a deadlocked state, detect it, and recover.

Deadlock prevention provides a set of methods to ensure that at least one of the necessary conditions cannot hold.

Deadlock avoidance requires that the operating system be given additional information in advance concerning which resources a thread will request and use during its lifetime. With this additional knowledge, the operating system can decide for each request whether or not the thread should wait.

3.4.2) Deadlock prevention :

Mutual exclusion : we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable.

Hold and wait :

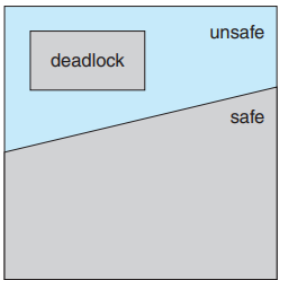
Solution 1 : One protocol that we can use requires each thread to request and be allocated all its resources before it begins execution. This is impractical for most app.

Solution 2 : allow a thread to request resources only when it has none. Meaning before it can request any additional resources, it must release all the resources that it is currently allocated.

No preemption : can't be prevented

Circular wait : One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each thread requests resources in an increasing order of enumeration.

Condition	Negation of condition	Approach to prevent condition	Comments about Practicality of approach
Mutual Exclusion	No Mutual Exclusion	Spool Everything	Only possible for few resources like Printers. Not possible for all resources.
Hold and wait	No Hold and wait	Request all resources initially	Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none. Low resource utilization.
No preemption	Preemption is allowed	Take resources away	This is very inefficient since we may need to restart process every time we take away resource from it.
Circular wait	No Circular wait	Order Resources Numerically	Yes, it is practical and easy to apply.



3.4.3) Deadlock avoidance :

A state is **safe** if the system can allocate resources to each thread (up to its maximum) in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a **safe sequence**.

Structure of Banker's algorithm :

```
int n;              //# of threads
int m;              //# of resources
int avail[m];       //# of available resources of each type
int max[n,m];       //# of each resource that each thread may want
int alloc[n,m];     //# of each resource that each thread is using
int need[n,m];      //# of resources that each thread might still request.
```

Example,

	Allocation	*Max*	*Available*		*Need*
	A B C	*A B C*	*A B C*		*A B C*
T_0	0 1 0	7 5 3	3 3 2	T_0	7 4 3
T_1	2 0 0	3 2 2		T_1	1 2 2
T_2	3 0 2	9 0 2		T_2	6 0 0
T_3	2 1 1	2 2 2		T_3	0 1 1
T_4	0 0 2	4 3 3		T_4	4 3 1

Sequence <T1, T3, T4, T0, T2> is safe sequence.

4. File System Implementation

//Lecture 35

Q : What is file system ? – It provides a way to store files, provides a way to name files and provides a way to store metadata of file. File systems provide efficient and convenient access to the storage device by allowing data to be stored, located, and retrieved easily.

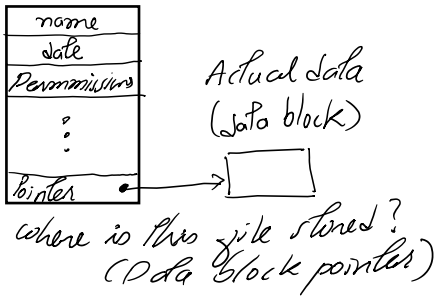
Directory : A file (in UNIX, everything is file) with special bit set in metadata to indicate directory.

File metadata : info related to name, identifier, location, size, protection, time, data, and user identification.

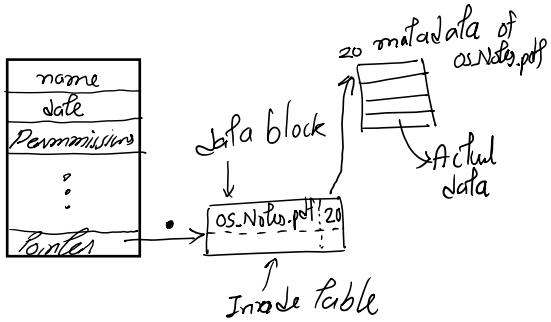
All this information is stored in some data structure called **inode**. One inode represents on file or one directory.

File (or directory) = inode (header/metadata) + data

Inode for a file

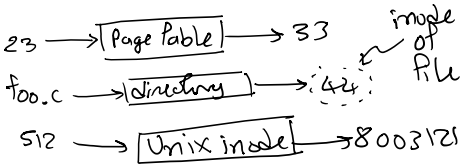


Inode for a directory



Thus, we can say that like page tables, file system metadata are simply data structures used to construct mappings

- Page table : map virtual page # to physical page #
- Directory : map name to disk address or file
- File metadata : map byte offset to disk block address



4.1) Allocation methods : how to allocate disk space to files ?

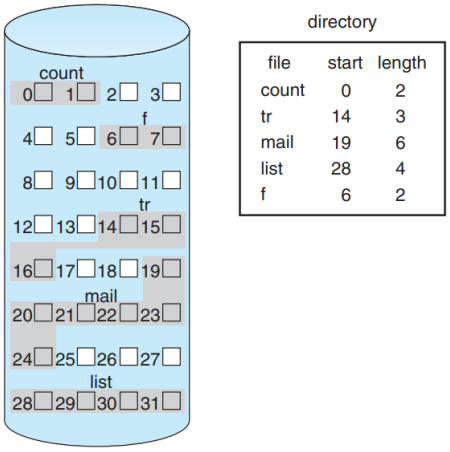
4.1.1) Contiguous allocation :

Contiguous allocation requires that each file occupy a set of contiguous blocks on the device. Device addresses define a linear ordering on the device.

Contiguous allocation of a file is defined by the address of the first block and length (in block units) of the file. If the file is n blocks long and starts at location b, then it occupies blocks b, b + 1, b + 2, ..., b + n − 1.

Pros :

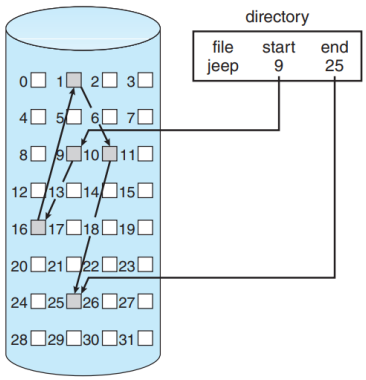
- Easy to implement
- Low storage overhead (two variables to specify disk area for file)
- Fast sequential access since data stored in continuous blocks
- Fast to compute data location for random addresses. Just an array index.



Cons : large external fragmentation, difficult to grow file (possibility of overriding)

4.1.2) Linked Allocation :

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of storage blocks; the blocks may be scattered anywhere on the device. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 (Figure 14.5). Each block contains a pointer to the next block. These pointers are not made available to the user. Thus, if each block is 512 bytes in size, and a block address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.



Pros :

- no external fragmentation
- Files can be easily grown with no limit
- Also, easy to implement, though awkward to spare space for disk pointer per block

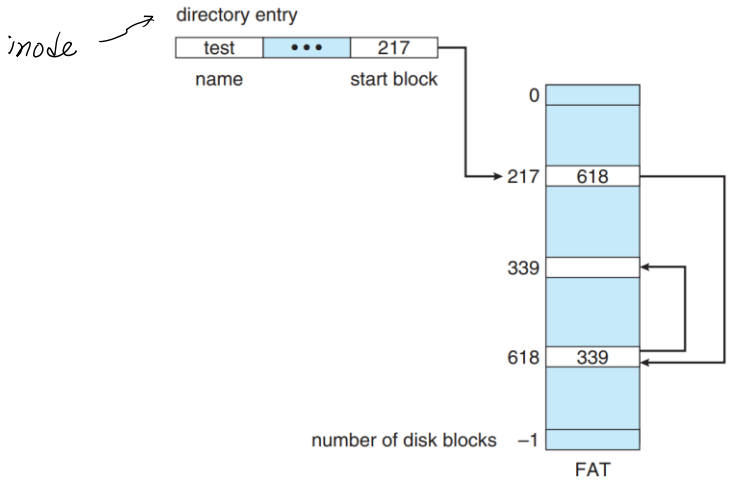
Cons :

- Large storage overhead (one pointer per block)
- Potentially slow sequential access
- Difficult to computer random addresses.

Variation : File allocation table

An important variation on linked allocation is the use of a **file-allocation table** (**FAT**). A section of storage at the beginning of each volume is set aside to contain the table. The table has one entry for

each block and is indexed by block number. The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until it reaches the last block, which has a special end-of-file value as the table entry.



So, FAT still do pointer chasing, but can bring entire FAT in MM so can be cheap compared to disk.

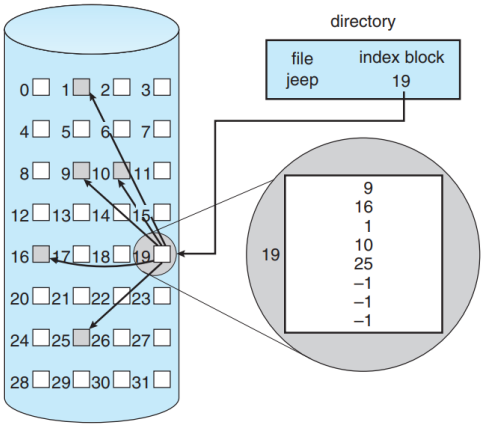
Q : Entry size = 16 bits. What's the maximum size of the FAT given a 512-byte block, what's the maximum size of FS ? – We know that entry size of FAT 16 bits meaning at max it can represent 2^{16} blocks meaning its maximum size is 2^{16} entries. And each entry points to one block meaning maximum size of File system = 2^{16} x 512 bytes = 32 MB.

Pros : fast random access. Only search cached FAT

Cons : large storage overhead for FAT table, potentially slow sequential access.

4.1.3) Indexed allocation :

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order. **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block**.



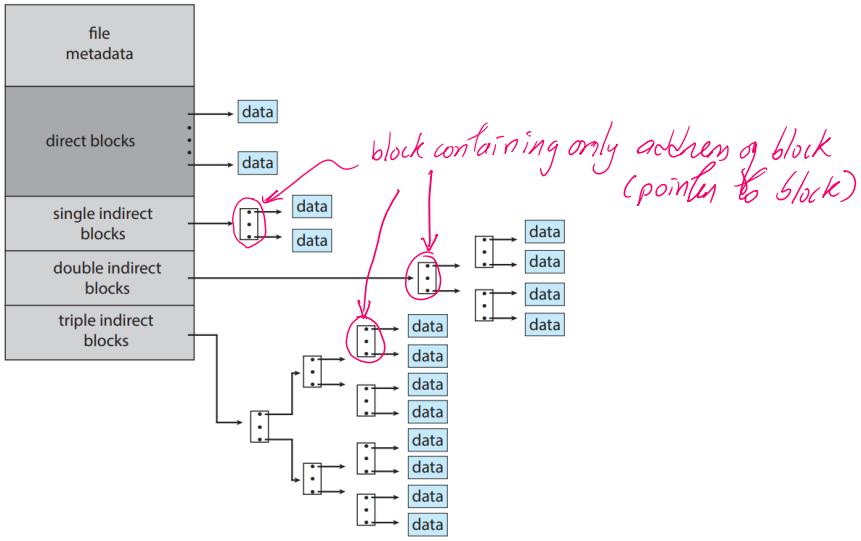
Each file has its own index block, which is an array of storage-block addresses. The ith entry in the index block points to the ith block of the file. The directory contains the address of the index block.

Pros : both sequential and random access easy

Cons : mapping table requires large chunk of contiguous space… same problem we were trying to solve initially.

This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers for a large file.

Q : Ca new have a method by which we can have small inode size and still we can store large file size ? – yes, this is same as page table where we introduced multilevel paging to represent large size data using small table.



15 pointers of the index block in the file's inode.

The first 12 of these pointers point to **direct blocks**; that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly. The next three pointers point to **indirect blocks**. The first points to a **single indirect block**, which is an index block containing not data but the addresses of blocks that do contain data. The second points to a **double indirect block**, which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer contains the address of a **triple indirect block**.

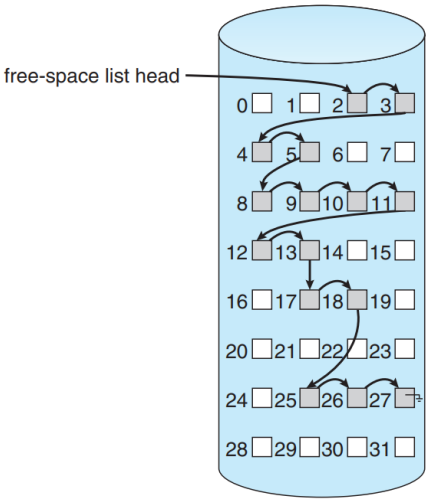
//Lecture 36

4.2) Free space :

Since storage space is limited, we need to reuse the space from deleted files for new files, if possible. (Write-once optical disks allow only one write to any given sector, and thus reuse is not physically possible.) To keep track of free disk space, the system maintains a **free-space list**.

Bit Vector : Frequently, the free-space list is implemented as a bitmap or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0.

Linked List : Another approach to free-space management is to link together all the free blocks, keeping a pointer to the first free block in a special location in the file system and caching it in memory. This first block contains a pointer to the next free block, and so on.



Grouping : Store n free blocks in first free block, last entry points to next block of free blocks.

Counting : Specify start block and number of contiguous free blocks. Rather than keeping a list of n free block addresses, we can keep the address of the first free block and the number (n) of free contiguous blocks that follow the first block.

⋆ ***Special terms*** :

I/O redirection : can be employed to use an existing file as input file for a program

interpretation: A methodology that allows a program in computer language to be either executed in its high-level form or translated to an intermediate form rather than being compiled to native code.