Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

*Simple OS Definition:

→ A special peice of software that

- abstracts and
  ↳ to simplify what the hardware looks like
- ~~arbitra~~ arbitrates ← the hardware
  ↳ to manage, oversee, control

the use of a computer system.

*Visual metaphore:

OS is like a toy shop manger

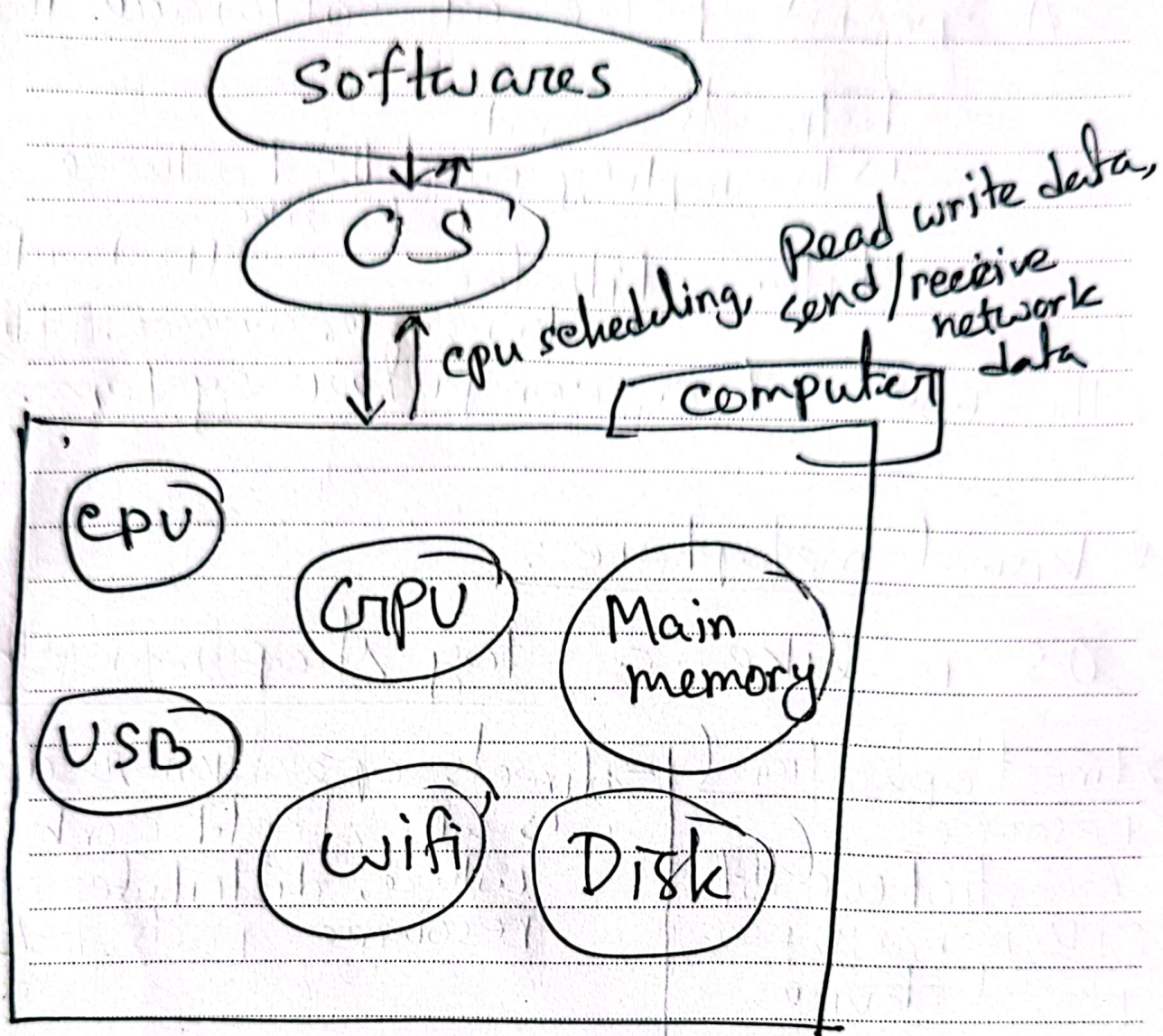
→ direct operational resources ↳ Control use of CPU, memory, peripheral devices	→ directs operation resources ↳ who should work where, distribute resource, parts, tools
→ Enforce working policies ↳ fair resource access, limits to resource usages	→ fairness, safety, cleanu
↳ abstract hardware details.	→ Mitigate difficulties of complex tasks ↳ simplifies operation, optimize performance

(System call)

Sun	Mon	Tue	Wed	Thu	Fri	Sat

Date .....

* What is OS?



→ hide ~~software~~ hardware complexity

→ resource management

→ Provide isolation & protection
(e.g. one software can't access another's memory)

Sun	Mon	Tue	Wed	Thu	Fri	Sat

Date .....

So, An OS is a layer of system software that :

(i) directly has privileged access to the underlying hardware.

(ii) hides hardware complexity

(iii) manages hardware on behalf of one or more applications according to some predefined policie

(iv) In addition, it ensures that applications are isolated and protecte from one another.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date OS Elements

(i) Abstractions (os executes)

eg: Process, Thread, file, socket, memory page

(ii) Mechanisms: (run on the cpu)

eg: create, schedule, open, write, allocate

(iii) Policies (~~memory~~ manage underlying hardware)

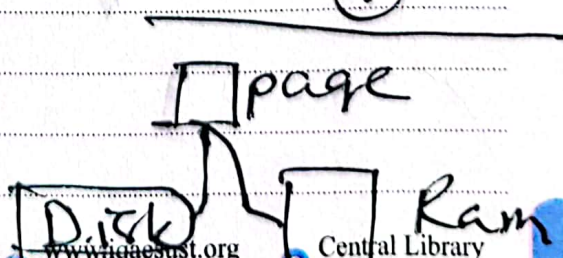
eg: least-recently used, earliest deadline first (EDF)

Memory management example:

Abstraction: memory page

Mechanisms: allocate, map to a process

Policies: LRU - cache



Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

OS Design Principles

(1) Separation of mechanism & policy

→ implement flexible mechanisms to support wide range of policy like. LRU, LFU, Random

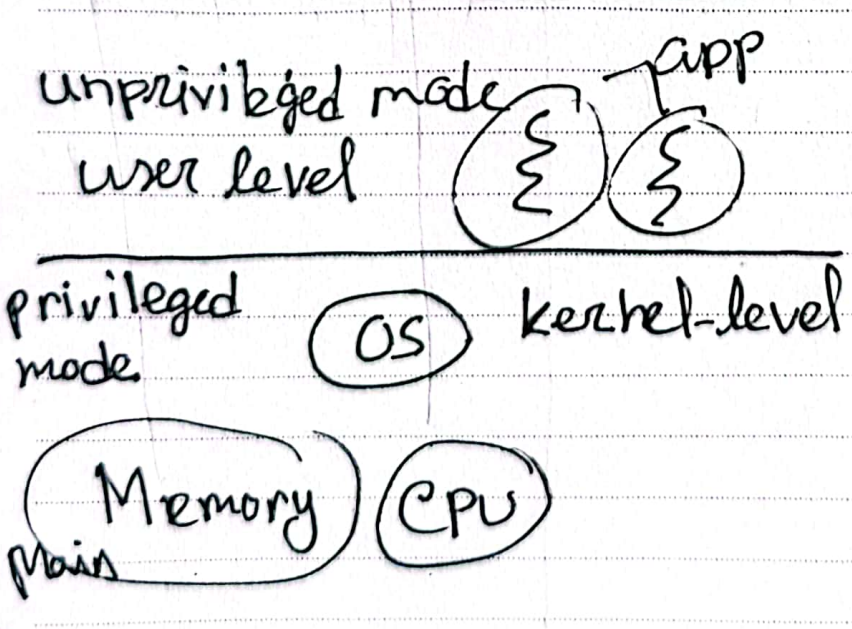
(2) Optimize for common case

→ where the OS will be used

→ What will the user want to execute on that machine

→ what are the workload requi.

User-kernel proteetion Boundary



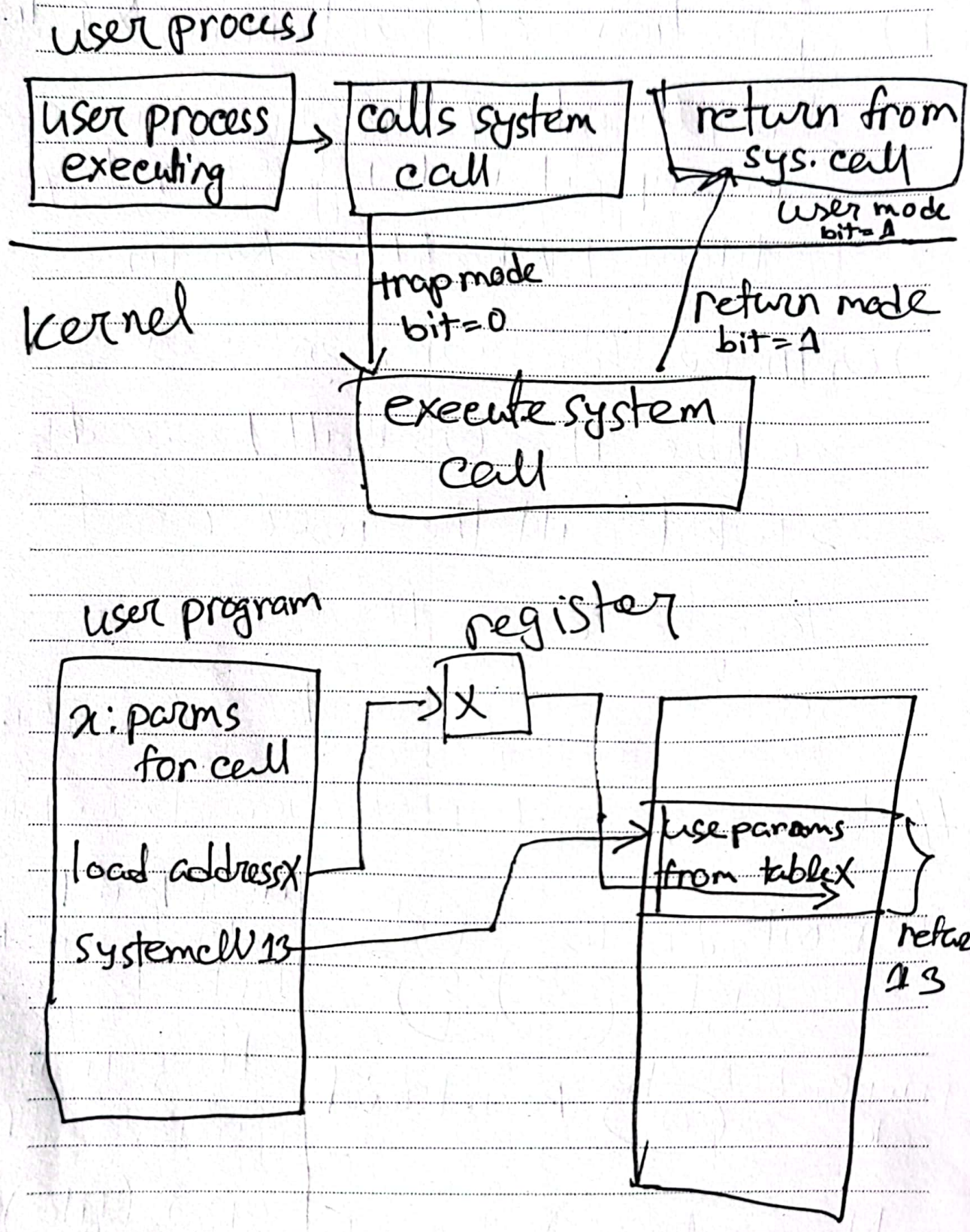
user-kernel switch is supported by hardware:

- trap instruct
- system call
- open (file), send (socket), malloc (memo

Sun Mon Tue Wed Thu Fri Sat

Date .....

System Call flowchart



Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

Basic OS Services

→ process management
→ file management
→ device management
→ memory management
→ storage management
→ security . . .

for 64 bit linux:

→ send signal to a process → KILL
→ set group identity of a process → SETGID
→ mount a filesystem → Mount
→ read/write params → sysctl

Sun | Mon | Tue | Wed | Thu | Fri | Sat

OS organizations

* Monolithic OS

→ every possible service that any one of the application can require or any type of hardware will demand is already part of the OS.

Pros: everything included
inlining, compile time optimization

Cons: customization, portability, managability, memory footprint performance

* Modular OS (associated with linux)

→ has number of basic services and apis already part of it but everything can be added as modular has a module interface that any module must implement in order to become part of OS

pros: maintainibility
smaller footprint
less resource needed

cons: indirection can affect performance
maintainance can still be a issue.

Microkernel: require the most basic primitives at ~~sy~~ OS level. such as: address space, thread.
everything else will run at user level like: filesystem, drivers etc.

pros: size, verifiability.

cons: cost of user/kernel crossing
complexity software development
portability.

Institutional Quality Assurance Cell (IQAC)
Shahjalal University of Science and Technology
Sylhet-3114, Bangladesh

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

* A process is like an order of toys

↳ program counter, stack

→ Parts & temp. holding area
↳ data, register state occupies state in memory

↳ I/O devices

→ State of execution
↳ completed toys, waiting to be built.

↳ plastic pieces, containers

→ may require special hardware
↳ glue, swing machine

⇒ Basically, process represents the execution state of an active application.

⇒ What does a Process look like?

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

types of state

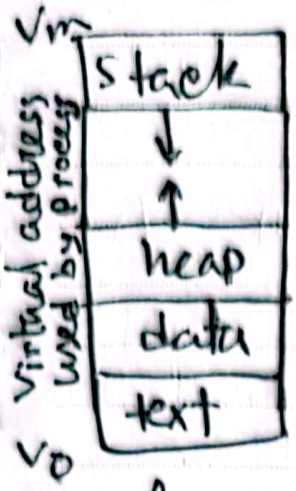


fig: 1

→ text & data
- static state when process first load

→ heap: dynamically created during execution

→ stack: grows & shrinks in LIFO order
eg: call-stack in Java

address space == "in memory" representation of a process

page table == mapping of virtual to physical address

* swapping some virtual address to disk when that's not needed.

example:

if two processes, P1 & P2 are running at the same time, what are the virtual address space ranges they will have

→ both will have access to all the virtual range

fig 1

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

⊛ Process Execution state.

⇒ How does a OS know what a process is doing?

→ Program Counter (PC): At any given point of time, the CPU needs to know where in the instruction sequence of the process is. (assembly code)

→ CPU registers: PC is ~~stored~~ maintained in CPU while the process is executing in a register. It is necessary to as they may store info like address for data

→ Stack pointer: the last state of the process. Need to know the top of stack.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

⊛ Process Control Block (PCB):

A data structure that the OS maintain for every one of the processes that is manage.

process state
process number
pc
registers
memory limits
list of open fiels
priority
signal mask
cpu scheduling info

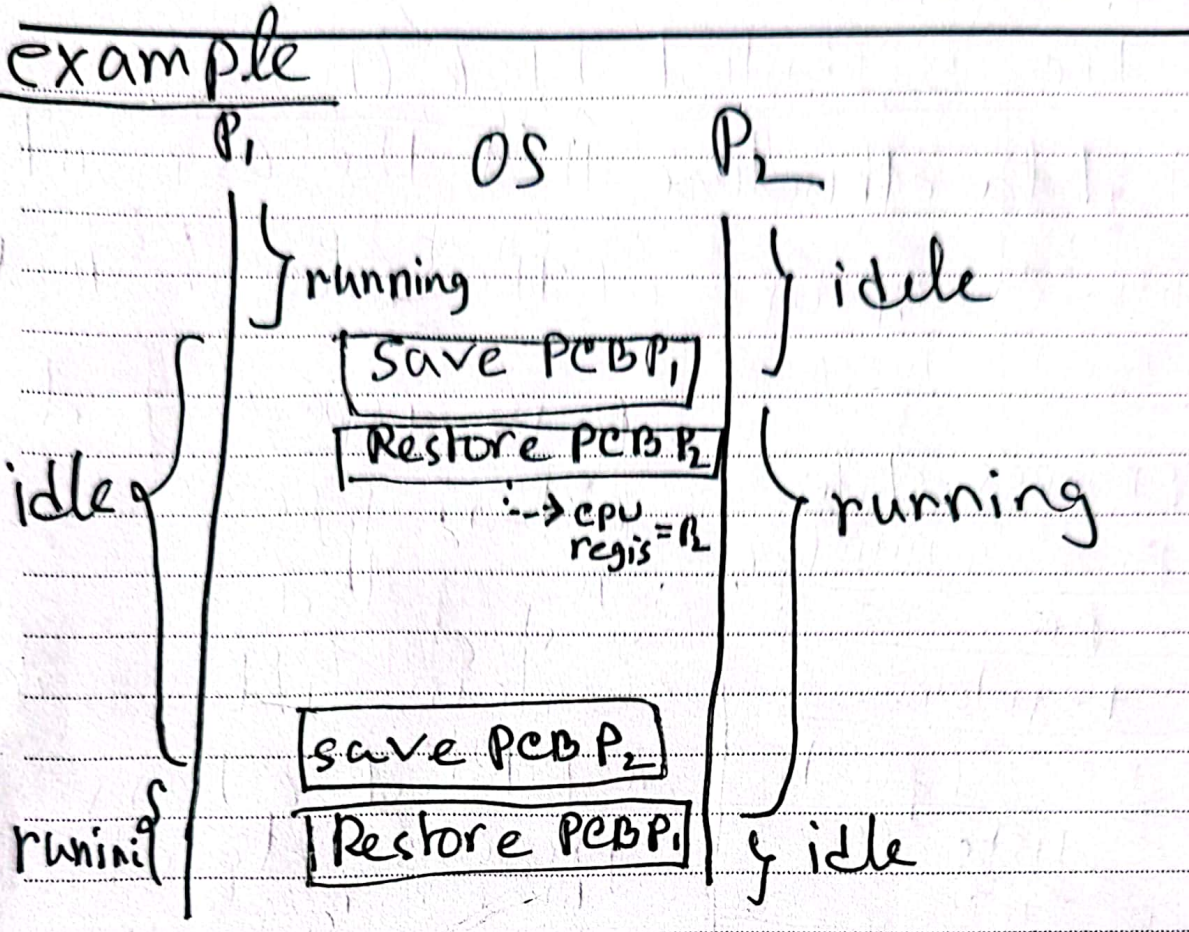
- PCB is created when process is created
- Certain fields are updated when process state change like mem. increase
- Othe fields change too frequently.

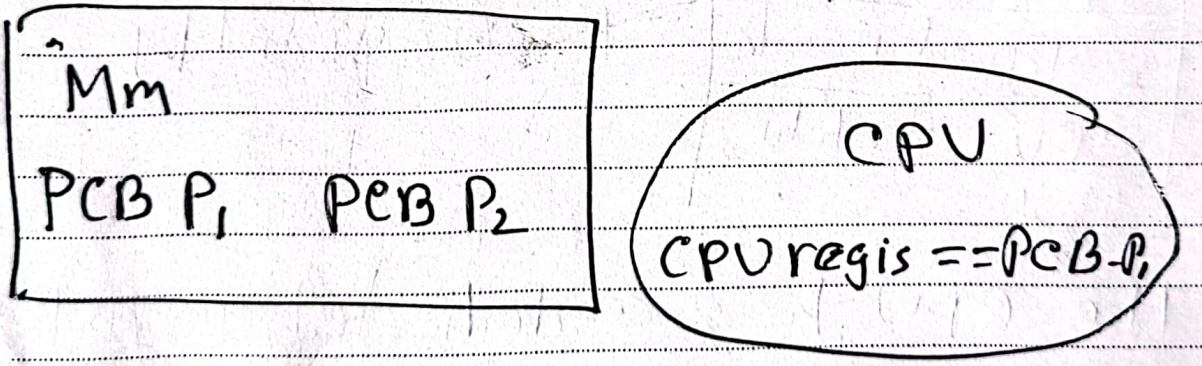
The CPU has a dedicated register which it usages to track the current PC for the currently executing proces The PC register will auto. ~~upgra~~ updated by CPU or every new instruction.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

example





Sun | Mon | Tue | Wed | Thu | Fri | Sat

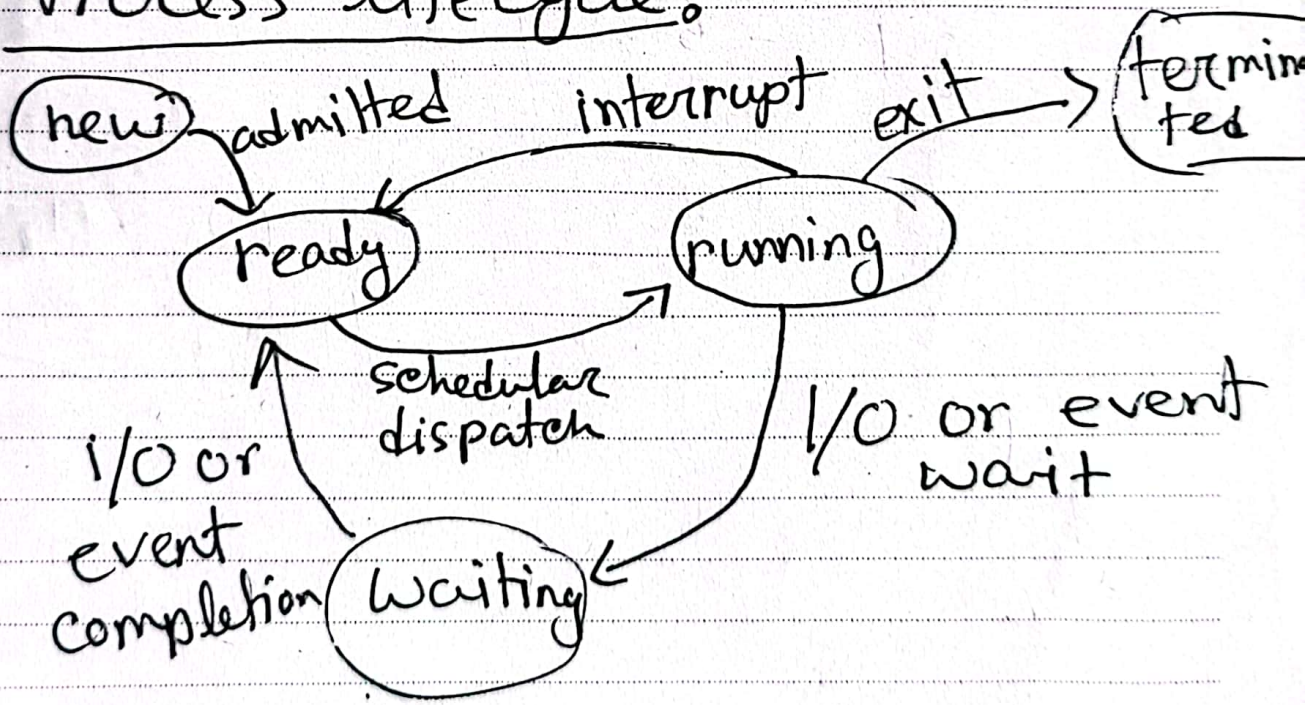
Date .....

⊛ Context Switch

⇒ A mechanism for switching the CPU from the context of one process to context of another.

- they are expensive!
  - direct costs: number of cycles for load & store instructions
  - indirect cost: COLD cache! cache miss! processor cache: store running processes info

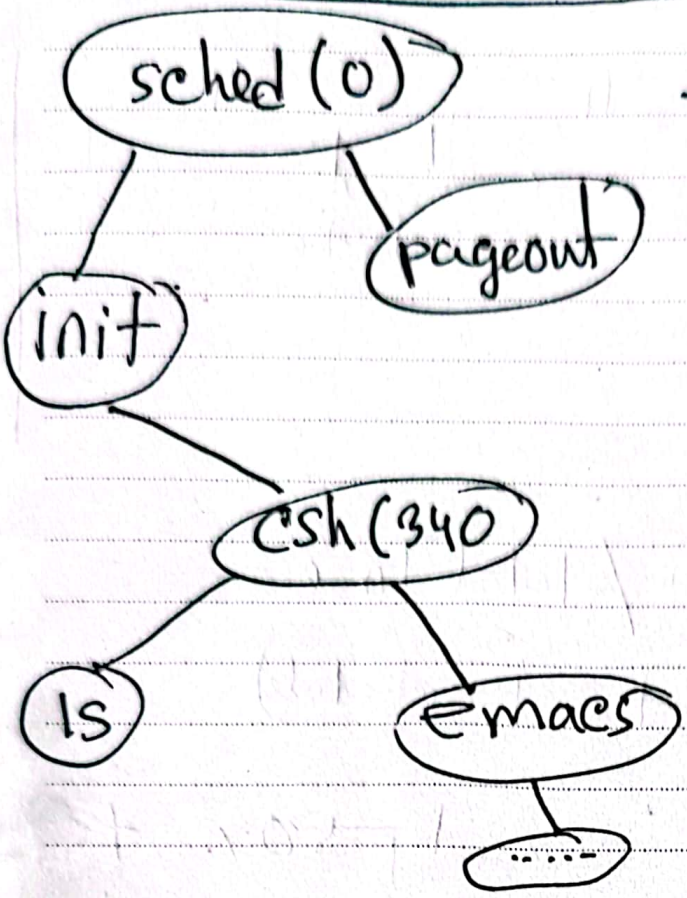
* Process lifecycle:



Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

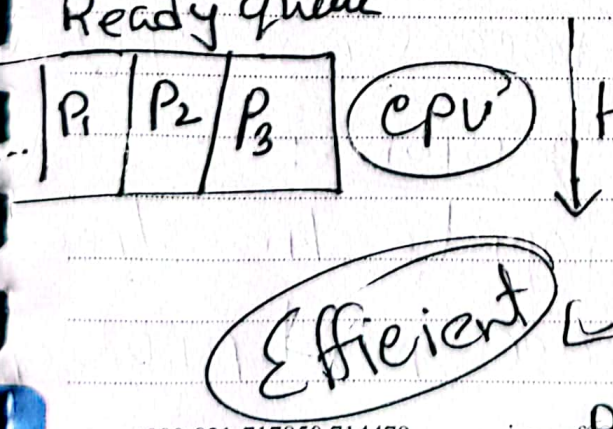
Process Creation



- fork: copies the parent PCB into new child PCB child continues execution at instru after fork
- exec: replace child image load new prog. & starts from first

*Role of CPU Scheduler:

It determines which one of the currently ready processes will be dispatched to the CPU to start running and how long it should run for.



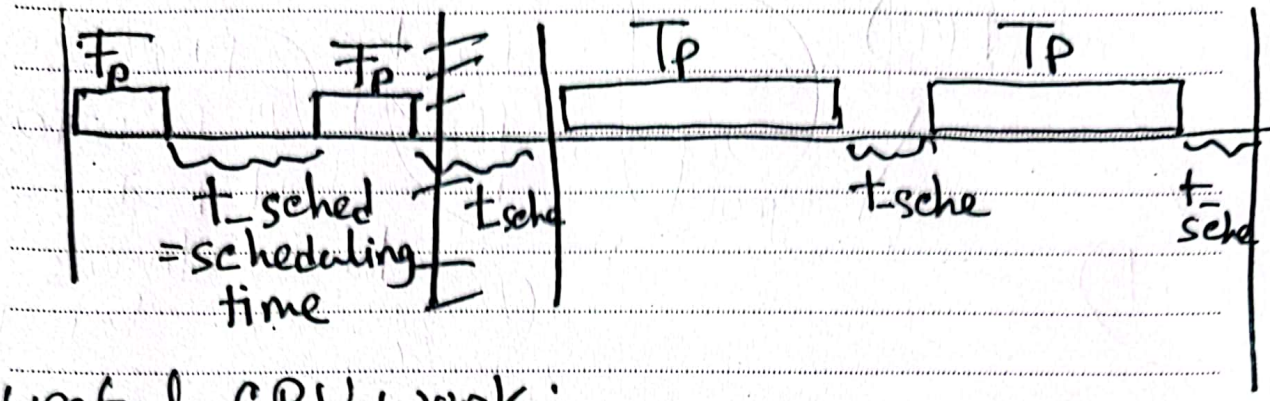
OS must
- preempt = intera and save current cont
- scheduler: run this to choose next proc
- Dispatch: dispatch process & switch into its context

Efficient

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

* Length of a process



Useful CPU work:

= Total processing time / total time

= (2 x Tp) / (2 x Tp + 2 x t_schedule)

if Tp = = t_sche ⇒ only 50% of cpu time spent on useful work

if Tp = = 10 x t_sche

⇒ 91% of cpu useful work

Timeslice = time Tp allocated to a process on CPU

Scheduling Design Decisions:

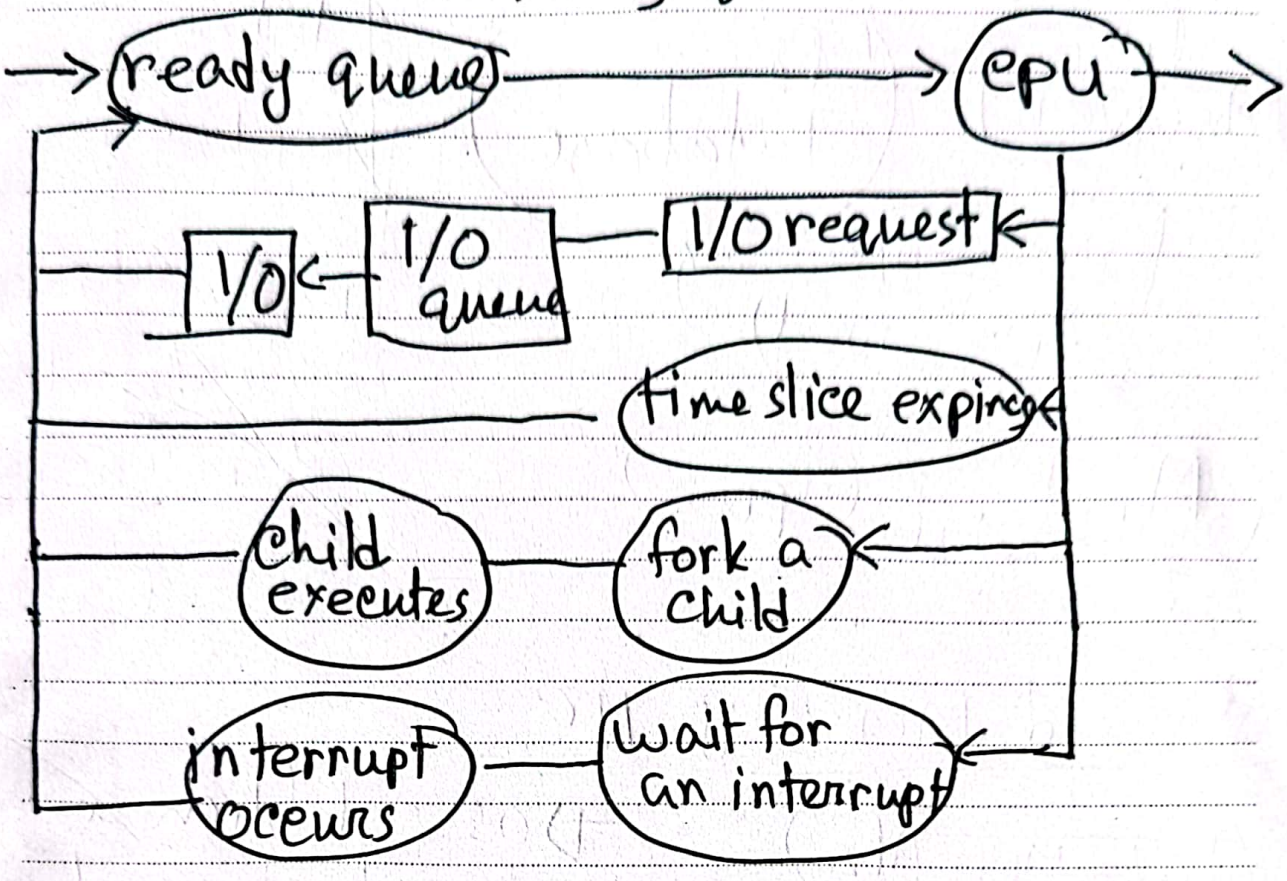
→ finding out appropiate timeslice

→ metrics to choose next process to run.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date

I/O | How process makes its way to a ready queue.



Inter process Communication (IPC)

→ transfer data/info between address spaces

→ maintain protection & isolation

→ provide flexibility & performance

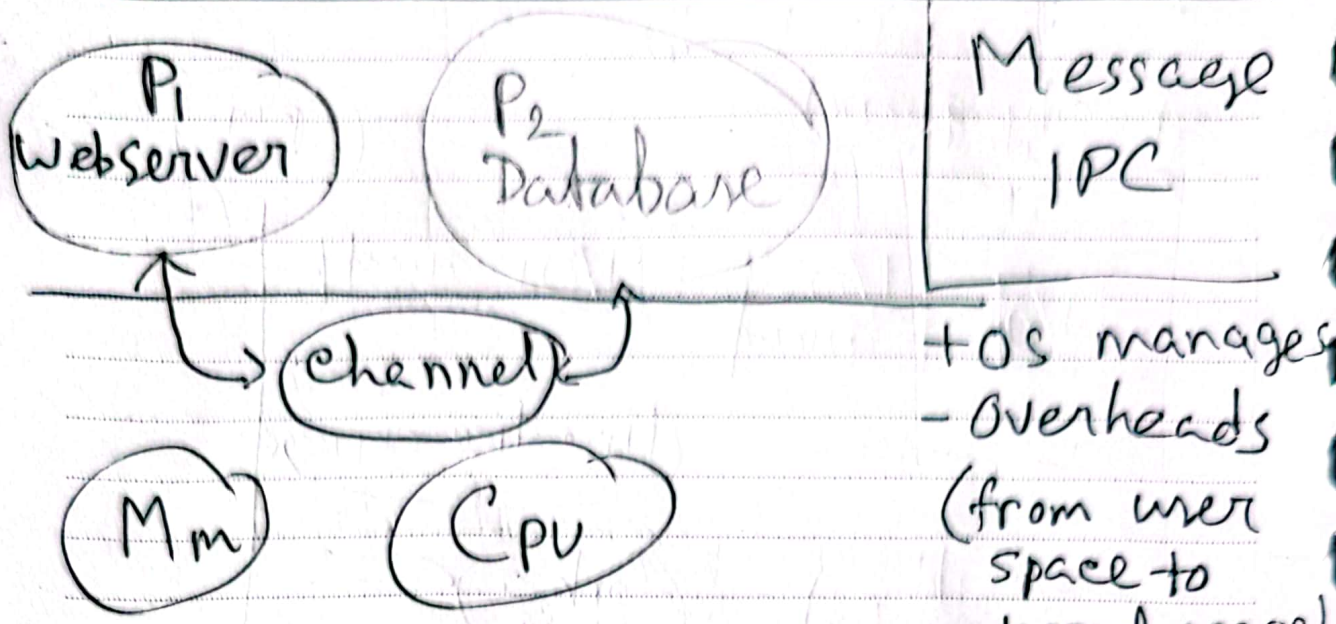
* Message-Passing IPC

→ OS provides communication channel like shared buffer

→ Processes read/write to/from channel

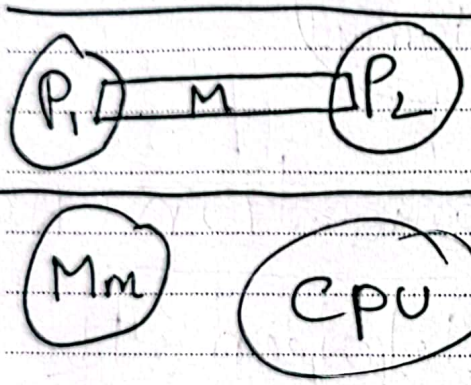
Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....



+ OS manages
- Overheads (from user space to kernel space)

*Shared Memory IPC



→ OS establish a shared channel and maps it into each process address space

→ processes directly read/write from memory

+ OS is out of the way (same space)

- (re) implement code

THREAD

Institutional Quality Assurance Cell (IQAC)
Shahjalal University of Science and Technology
Sylhet-3114, Bangladesh

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

* A thread is like a worker in a toy shop.

↳ executing unit of a process

↳ many threads executing

→ requires co-ordination

↳ sharing of I/O device, CPUs, memory

→ is active Entity

↳ executing unit of toy order.

→ work simultaneously with others

↳ many workers complete toy order

↳ sharing tools, parts, workstation

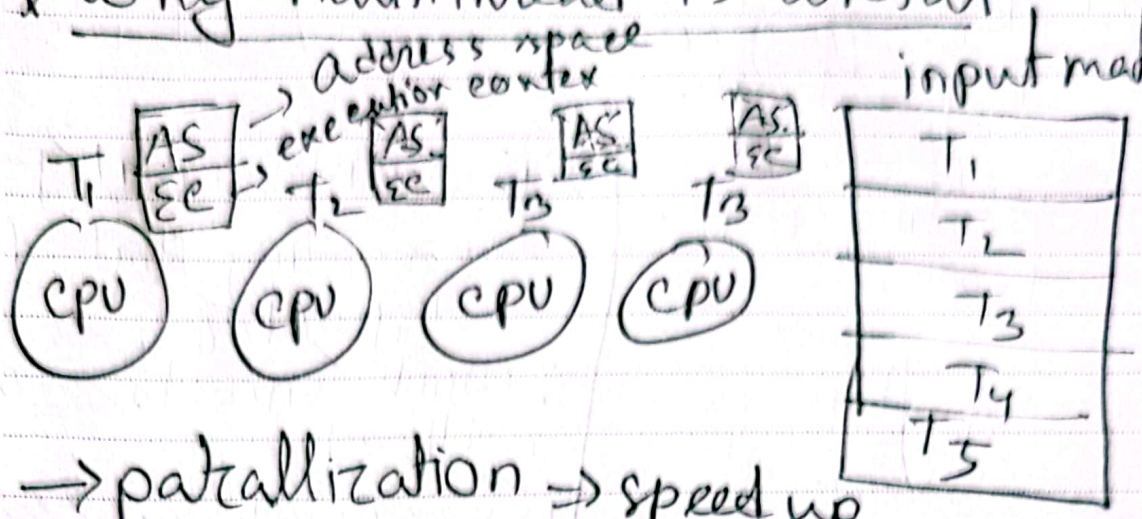
* difference between thread & Process

Process	Thread
→ Process means a program in execution	→ thread means a segment of a process
→ NOT Lightweight	→ Lightweight
→ takes more time to terminate/create	→ less time
→ takes more time to contex switching	→ less time
→ more resource	→ less
→ Different process are treated differently	→ All the level peer threads are treated as a single task by OS
→ most of is isolated	→ share memory

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

* Why multithread is useful



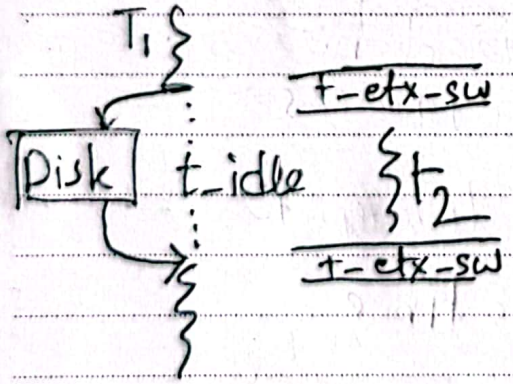
→ parallization ⇒ speed up

→ specialization ⇒ hot cache

(as mvall thread share same memory)

~~Extention of why multithread is useful~~

* Are Threads useful on a single CPU



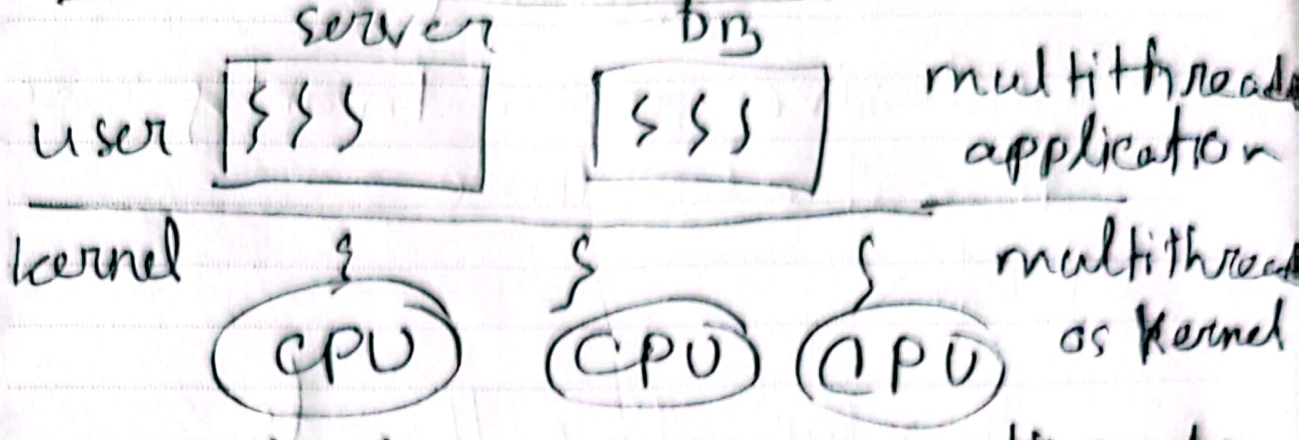
→ if t_idle > 2*2(t_ctx)
the contex switching
to hide idleing time

→ t_ctx_switch threads
< t_ctx_switch process
↳ hide latency

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

* Benefits to application & OS



- os-level services like daemons or drivers
- threads working on behalf of apps

* Basic Thread Mechanisms

⇒ Threads & Concurrency

→ Threads share the same virtual-to-physical address mapping

→ it's possible to access same memory space from different threads (not possible for process)

→ violates consistency

Sun	Mon	Tue	Wed	Thu	Fri	Sat

Date .....

* mutual exclusion (sync. mecha.)

⇒ exclusive access to only one thread at a time

→ mutex

⇒ waiting for other threads

- specific condition before proceeding
- condition variable

* Thread Creation (accor. Bruell)

Thread type + type

- Thread data structure — id, PC, SP, register, stack, attributes

Fork (proc, args)

- create a thread
- not unix thread

T1 DS
PC = proc
stack = args

Join (thread)

- terminate a thread

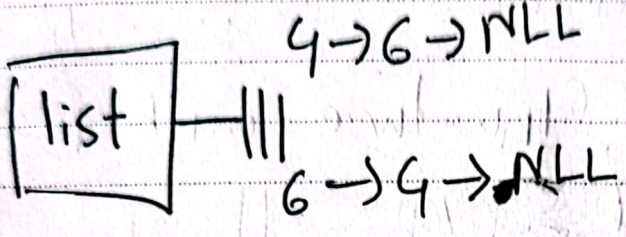
To
t1 = Fork(proc, args)
↳ T1
child_result
— join(t1)
return result
end

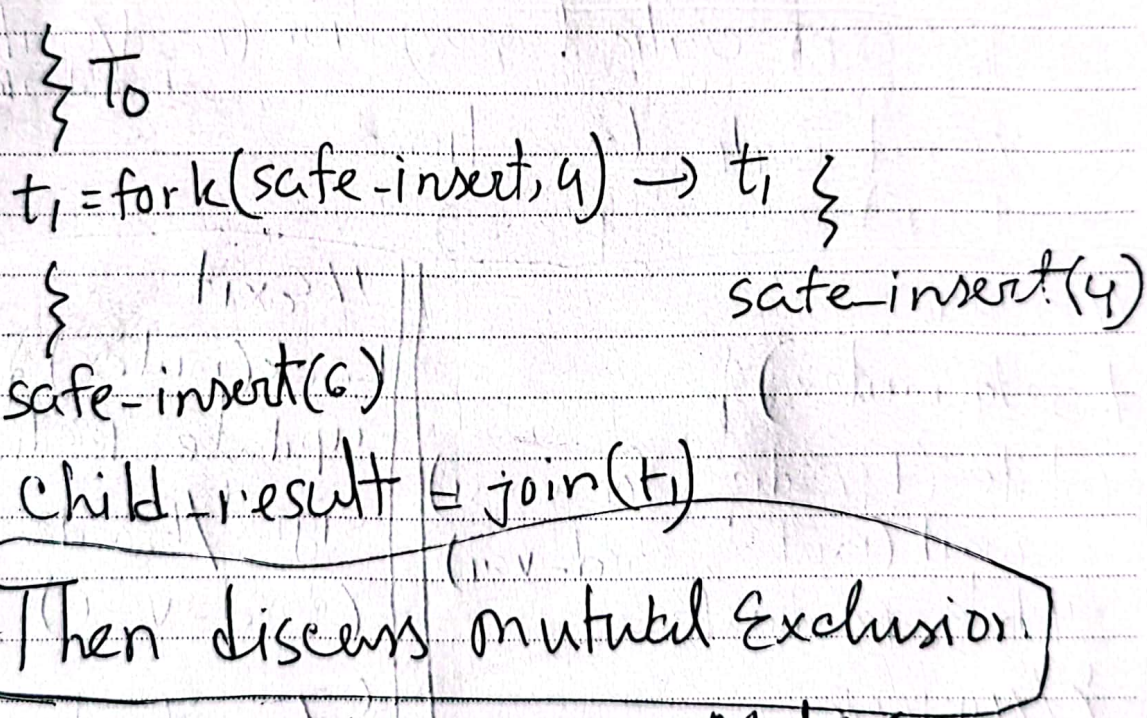
Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

Example of Thread creation

→
```
Thread t1;
Shared_list list;
t1 = fork(safe_insert, 4);
safe_insert(6);
join(t1);
```





* Then discuss mutual Exclusion

```
Lock(mutex){
critical section
```

Mutex
locked?
owner
blocked-threads

* Mutex lock with conditional variable

* Combine Mutex lock & conditional variable for Read/write Problem
→ multiple read at the the same time.

→ Critical Section Structure with proxy

```
// Enter Critical Section
perform critical operation (read/write shared file)
// Exit Critical Section
```

```
//Enter
Lock(mutex) {
while (! predicat_foraccess)
    wait(mutex, cond-var)
update-predicate
} //unlock
```

```
//Exit
Lock(mutex) {
update predicate;
signal/broadcast
(cond-var);
} //unlock
```

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

* Avoid Common ~~mutex~~ mistakes in mutex

→ keep track of mutex/cond. variables used with a resource (comment)

→ Check that you are always using lock & unlock

→ use a single mutex to access a single resource.

→ check that you are signaling correct condition

→ Check that you are not using signal when broadcast is needed

→ deadlock

→ spurious wake up → (unnecessary wake up)

```
// writer
Lock(counter_mutex){
  r_c = 0;
  Broadcast(read_phase);
  signal(write_phase);
}
```

```
// Readers
wait(counter_mutex
read/write phase);
```

Sun	Mon	Tue	Wed	Thu	Fri	Sat

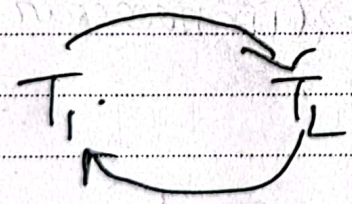
Date .....

* Deadlock: Two or more competing threads are waiting on each other to complete, but none of them ever do.

```
T1                    T2
{                     {
lock(m-A)             lock(m-B)
{                     {
lock(m-B)             lock(m-A)
{                     {
foo(A,B)              foo1(A,B)
```

How to avoid this

maintain lock order



Sun | Mon | Tue | Wed | Thu | Fri | Sat

* Kernel & User level Thread

To execute a user-level thread, first it must be associated with kernel level thread. Then, the OS level scheduler must schedule that kernel level thread onto a CPU.

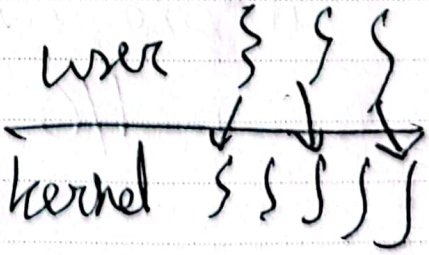
① One-to-one model: Each user thread is associated with one new/free kernel thread

⊕ OS understands threads, sync, blocking

⊖ must go to OS for all operations

⊖ OS may have limits on policies

⊖ portability.



(11) many-to-one model: User level thread are mapped onto a singal kernel level thread.

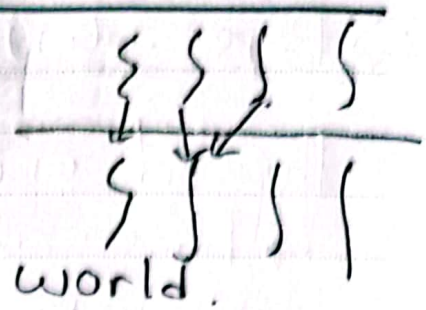
In the user level a thread management library that decides which user-level thread will be mapped onto the kernel level thread at any given point of time.

⊕ totally portable, doesn't depent OS limits.

⊖ OS has no insigihts into application needs (will see the system as singal threaded)

⊖ OS may block entire process if one user thread is blocks on I/O.

(iii) many-to-many:



(+) can be best of both world.

(+) can have bound or unbounded thread

(-) requires coordination between user level and kernel level threads.

⇒ Process scope: user-level library manages thread within a singal process

⇒ System scope: system wide thread management by OS-level thread mangers

(*) multithreading pattern

(i) Boss-workers (ii) Pipeline

(iii) Layered.

Seenario: for a toy shop.

(i) accept the order
(ii) parse the order
(iii) Cut wood parts
(iv) paint & decorate
(v) assemble the toys
(vi) ship the order.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

⇒ Boss - Worker Pattern: One boss thread and multiple worker thread.

→ boss: assigns work to worker
→ worker: perfors entire task.

boss: step 1, worker: 2-6 step.

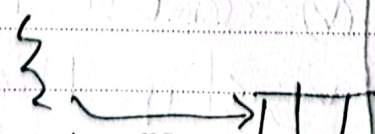
Throughput of the system limited ~~to~~ by boss thread ⇒ must keep boss efficient

$$\text{Throughput} = \frac{1}{\text{boss\_time\_per\_order}}$$

bossi - directly signalling specific worker

⊕ workers don't need to sync.
⊖ boss must keep track what each worker will be doing
⊖ throughput will go down.

- placing work in producer/cosumer queue.



Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

⊕ Boss doesn't need to know the details about workers.

⊖ queue sync. needed.

⊕ better throughput.

⇒ ~~Hog~~ How many workers?
— on demand - pool of workers.

Boss + worker variants:

→ all worker created equal

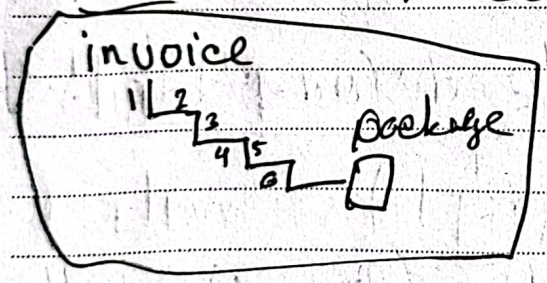
→ worker specialized for certain task
↳ better locality, less managed
↳ load balancing (how many threads should we assign a particular task)

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

⇒ Pipeline Pattern:

- threads are assigned subtask in the system
- entire tasks == pipeline of threads
- multiple tasks concurrently in the system, in different pipeline stages
- Throughput == weakest link
  ⇒ pipeline stage == thread pool.
- shared buffer based comm.



- sequence of stages
- stage == subtask
- each stage == thread pool
- buffer based comm.

⊕ specialization & locality

⊖ balancing and sync. overheads.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

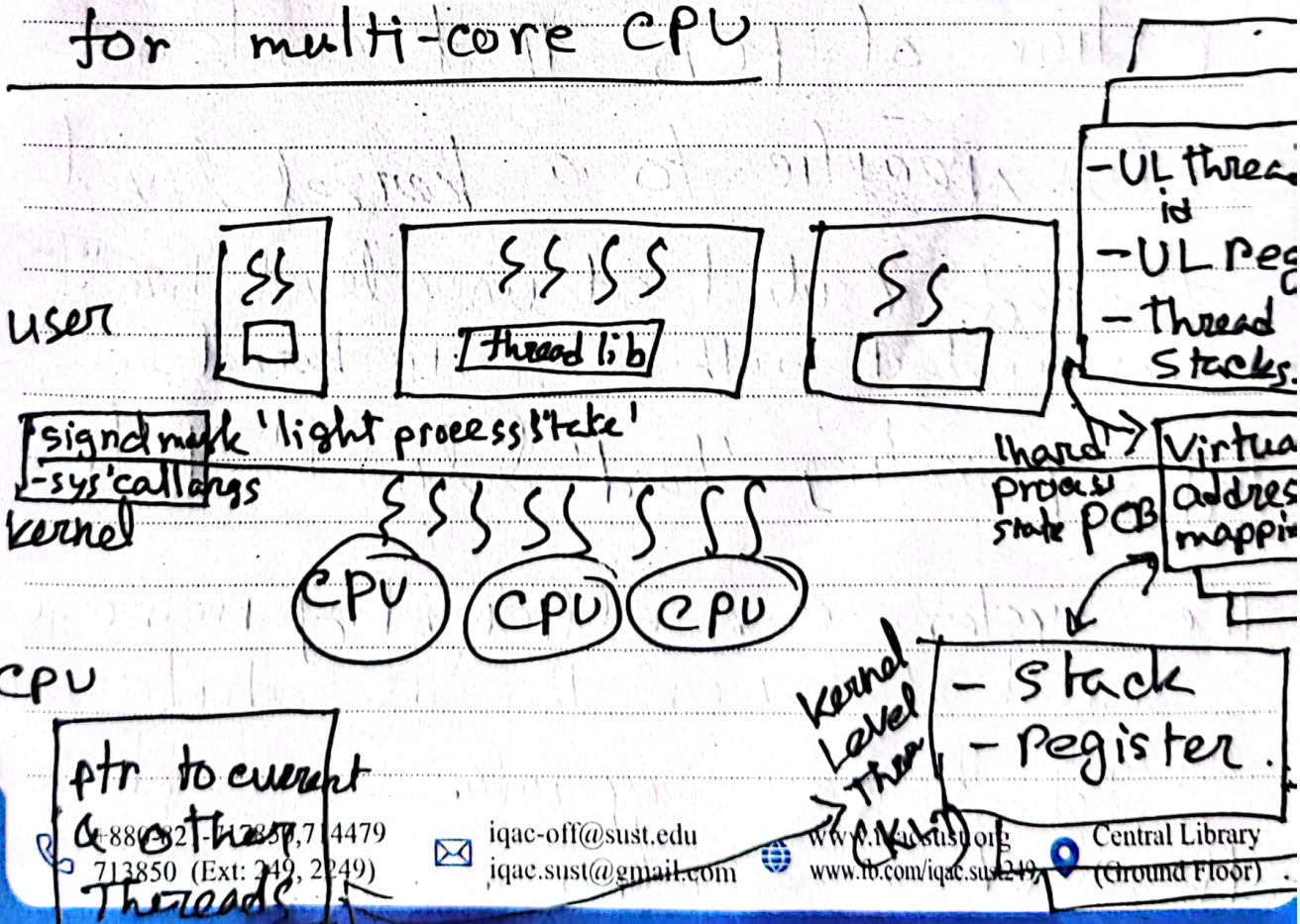
⇒ Layered Pattern:

→ each layer group of related subtasks
→ end-to-end task must pass up & down through all layer.

⊕ specialization ⊕ less-fine-grained than pipeline
⊖ not suitable for all applications
⊖ sync.

Steps
- Decorating (4)
- Assembly (3,5)
- processing (1,2)

* Thread Related Data Structure: for multi-core CPU



Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

→ 'hard' process state: refers to the portion of the PCB that contains information relevent for all user-level threads within a process. This includes virtual address mapping, process id, status etc. It includes info that needs to be preserved during context-switching.

→ 'light' process state: refers to the portion of PCB that contains info. specific to a kernel level thread. (subset of user-level thread associated with a specific kernel level th)

By seperating 'hard' & 'light' state the system can efficiently manage & switch between threads while preserving the necessary info. for each thread execution.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date

* Rational for multiple Data structure

⇒ Single PCB : large continuous DS, private for each entity, saved & restored on each context switch, update for any changes.

⊖ Scalability, ⊖ Overheads, ⊖ performance
⊖ flexibility.

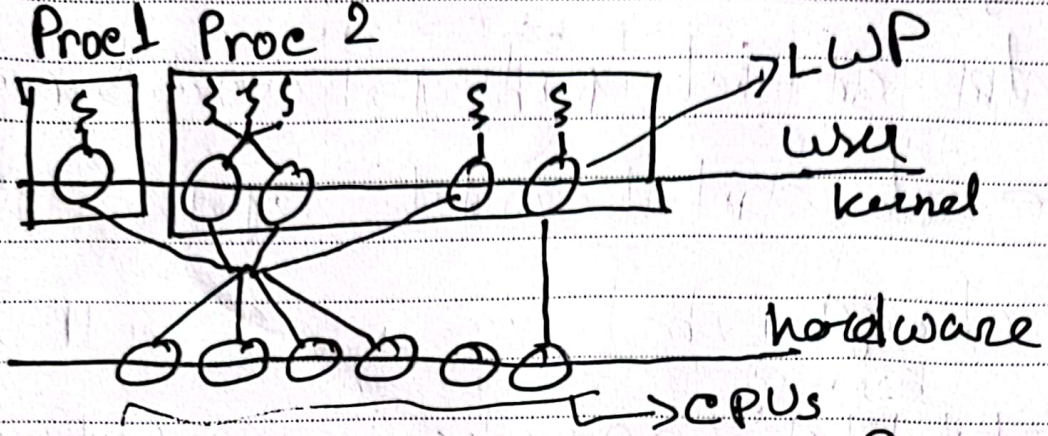
⇒ Multiple DataStructure : smaller DS, easier to share, or context switch only saved and restore what needs to change, user-level library need only update portion of the state.

→ ⊕

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

⊛ SunOS 5.0 Threading model / Solaris 2.0



→ both many to many & one to one model supported.

→ Each kernel-level thread that executing a user-level thread has a light-weight process Ds associated with it.

→ from User-level perspective, these LWP represents the virtual CPUs onto which its going to be scheduling the user-level threads.

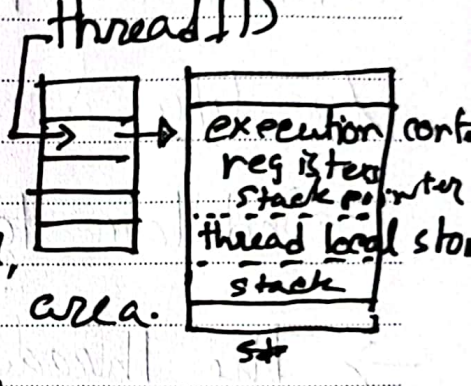
Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

→ At the kernel level, there will be a kernel-level ~~manager~~ scheduler that will be managing the KLT and scheduling them into physical CPUs.

⊛ User-level thread DS

→ Contains various fields such as. registers, signal mask, priority, stack pointer, and thread local storage area.



→ When a thread is created threading libr returns a thread_ID which is an index in a table of pointers rather than direct pointer to a thread DS.

→ This indexing approach allows for meaningful feedback or error message if there is any issue with thread, as opposed to pointing to a corrupt

→ This also includes the stack, which can be of defined size by the library or the user can provide it.

→ This Ds is created in a ~~manner~~ & layered continuous manner, enabling better locality & making it easier for the schedular to find next thread

→ The threading library does-not control stack growth, the OS is unaware of multiple ~~the~~ user-level thread, which can lead to overwriting another's DS

→ To mitigate this, a 'red zone' is introduced, which refers to a portion of the Virtual add. space that is not allocated. If a thread tries to write

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

in this zone, the OS will cause a fault, making it easier to identify and debug the problem.

* Kernel-level DS (multiple DS)

Process
- list of kernel-level thread
- virtual address space
- user credentials
- signal handarels

Light-weight Process (LWP)
- user level registers
- system call argument
- resource usages info
- signal mask

similar to ULT, but visible to kernel. not needed when process not running

kernel-level threads
- kernel-level registers
- stack pointer
- scheduling info
- pointers to associated LWP, Process, CPU structure.
- not swapable. always needed even when the process is not running

CPU
- current thread
- list of kernel-level threads
- dispatching & interrupt handling info.

On sparc dedicated reg == current thread

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

⊛ Basic Thread Management Interaction

→ In a multithreaded process. There can be multiple user-level threads that executes concurrently.

→ The actual level of concurrency may be limited by the factors such as I/O operations, where some thread may be waiting while others are executing

→ The OS has the limit on the number of kernel-Thread it can support

→ To efficienty manage threads, the user-level ~~threads~~ process can request a specific number of kernel-level thread using system called ~~the~~ set-concurrency.

→ The kernel responds to this system calls by creating additional thre. & allocate them to process

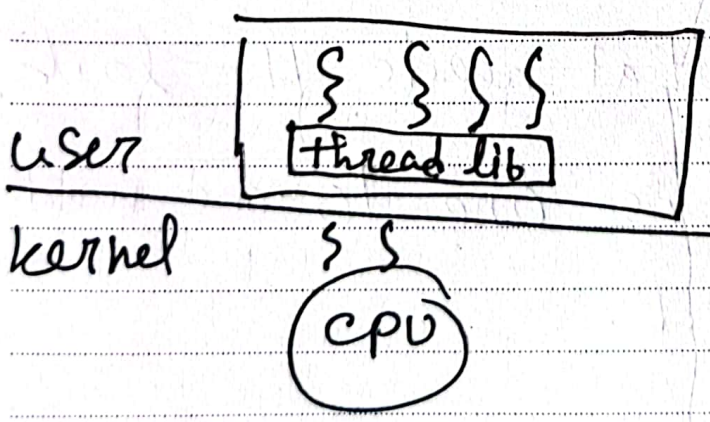
Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

→ If the user-level threads are blocke on I/O operation, the cooros ponding kernel-level thread are also blocked.

→ The user-level library do not know the blocking status of kernel-level thre. which leads to a situation where the whole process is blocked.

→ To address this, the kernel can notify the user-level library before blocking the KL threads, allowing the library to reques more KL threads or lightweight process.

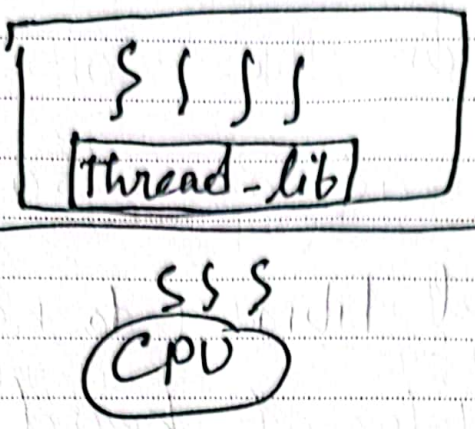


*→ assume the process request 2 kernel level Threads

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

* Thread management Visibility & Design



UL library sees -
- uTs
- available KLTs

kernel sees
- KLTs.
- CPUs
- KL scheduler

This cause,

* lets assume, if a kLT supporting the execution of a critical section preemted by the kernel The execution of that critical section can not continue until the kernel schedules that thread again. This may lead to delays in the execution of other ULTs that needs the lock.

→ To address this one-to-one model are often used.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

Process jumps to U Library scheduler when:

- ULTs explicitely yield
- Timer set by ULibrary expires
- ULTs call library fuction like lock/unlock
- blocked threads become runnable.

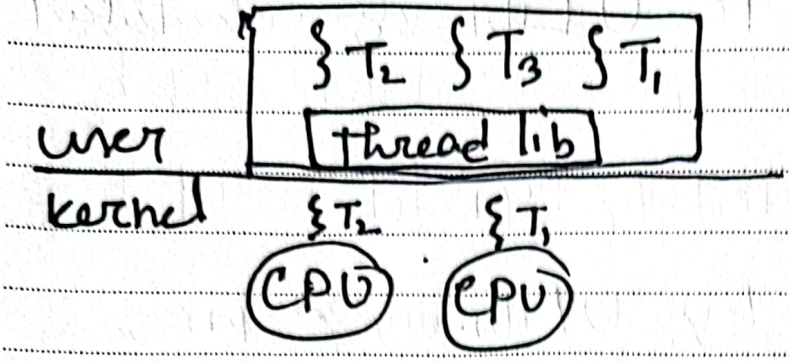
* Issues on multiple CPUs

This arise when dealing with multiple CPUs in a system. In a single CPU system user threads run on top of that CPU and any scheduling changs are immediately reflected on that CPU. However, in a multi-CPU system, KLTs support a single process may be running on multiple CPUs concurrently. This introduce complexities in managing user-level threads across different CPUs

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

For example



Priority $T_3 > T_2 > T_1$

T_2 running on one CPUs and holds a mutex while T_3 waiting on that mutex and are on block. (T_1 is running in another cpu.) When T_2 release the mutex, T_3 become runnable and all 3 threads are now runnable. Now, to ensure the priority. T_1 needs to preemted. As T_1 is running on another CPU, we need to notify CPU to update its registers & PC.

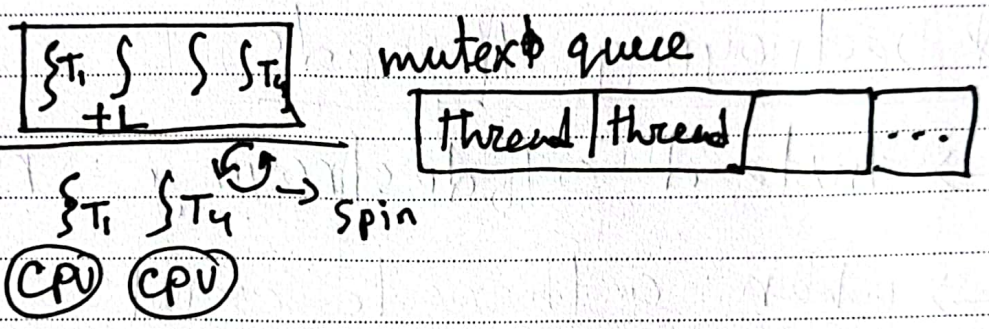
This is done by sending a signal or interrupt from one CPU to the other. instructing it to execute necessary library code for scheduling decisions.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

(*) Syncronization-related issue:

In a multi-CPU system, when multiple threads needs to access a shared resource protected by a mutex, it is possible that the owner of the mutex to be running on different CPU.



In such case it may be more efficient for the waiting thread to spin instead of blocking and queued up on the mutex. This is effective if CS is short.

→ Adaptive mutex are the type of mutex that dyramically decide. wheather a thread should spin or block based on the current state of the mutex and

They are designed for multi-CPU system and provide balance between spinning and blocking. depending on the characteristics of the cs. For long critical section apply default blocking behaviour.

*Destroying Threads

→ Instead of destroying - reuse.

→ when a threads exist.

- put on a death row
- periodically destroyed by reaper thread
- otherwise Thread structure / stacks are reused - peformance gain.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

Interrupts & Signals

Interrupts	signals
→ events generated externally by components other than CPU (I/O device, timers)	→ events triggred by the CPU & software running on it.
→ determined based on physical platform	→ based on the OS.
→ appear asynchro.	→ appear sync or async.

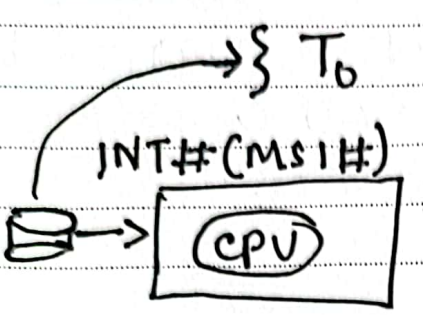
Both have

- unique ID depending on the hardware (intru). or OS.(sig)
- Can be masked and disabled/suspended via corresponding mask
  → per-CPU interrupt mask, per-process signal mask
- If enabled, trigger corresponding handler
  → interrupt handler set for the entire system by OS
  → signal handlers set on per process basis, by process.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

⊛ Interrupt handling



interrupt handler table

INT-1	handler 1 start add.
INT-2	" 2 start add.
⋮	⋮
INT-N	" N start-add

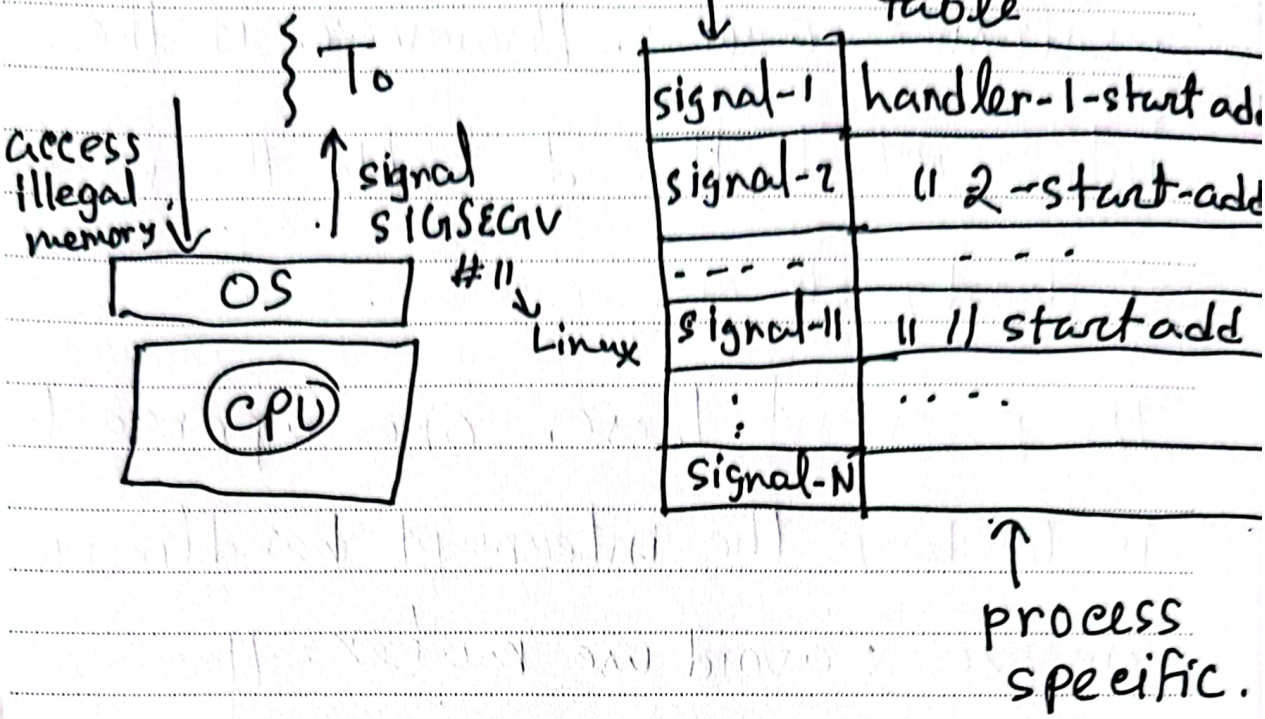
when a device, such as disks, wants to send a notification to the CPU, it sends an interrupt that connect the device to the CPU. In modern system this can be done using a message-signal-interrupter carried on the same interconnect.

This interrupts the execution of the thread that was running on the CPU. If the interrupt is enabled, a table is referenced based on the interrupt number

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

⊛ Signal Handeling



⊛ Why disable Interrupts or signals

→ When the interrupt or signal occurs, it is executed in the context of thread that was interrupted, & this can lead to ✗ issue.

→ One example is that, when the interrupt handling code needs to access shared state that other threads in the system may also be accessing.

In such cases mutex are used for exclusive access. However, if the interrupted thread holds the mutex, a deadlock can occur.

To prevent these, one approach is to keep the interrupt handling code simple & avoid use mutex. However, these can impose limitation on handling. Here masks can be used. These mask allow dynamic enabling or disabling of interrupts or signals during specific sections of code.

* Interrupt mask are per CPU

If masks disables interrupt ⇒ hardware interrupted routing mechanism will not deliver interrupt to CPU.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

⇒ Signal masks are per execution context (ULT on top of KLT)

If masks disables signal ⇒ kernel sees the mask and won't interrupt corresponding thread.

⊛ Interrupts on Multicore System

- Interrupts can be directed to any CPU that has them enabled.
- may set interrupt on just a single core → avoids overheads & perturbations related to interrupt handling from any other CPU/cores

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

* Types of Signals

(i) One-shot signals: These are standard signals. There are designed to be handled at least once when multiple instances of the same signal occurs. The handling routine for one-shot must be re-enabled every time this is invoked.

(ii) Real-time signals: Every signal will be handelled.
n signals → n handler calls.

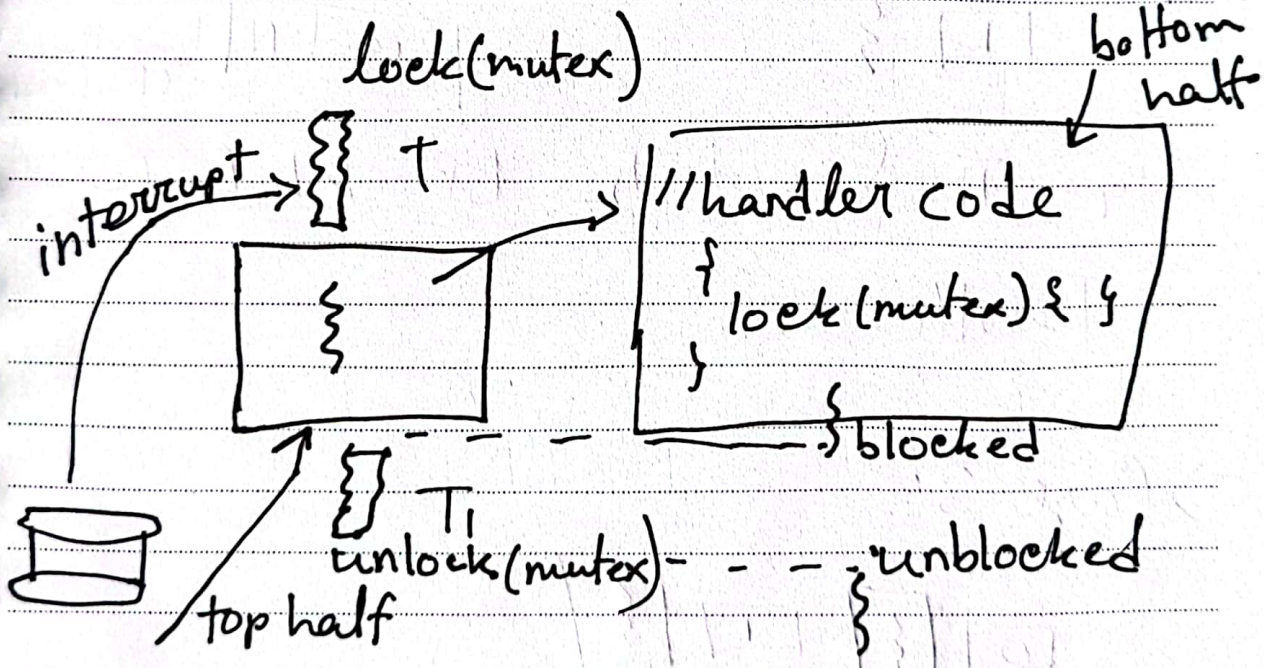
* Top half & bottom half: refers to a technique used in OS to handle interrupts efficiently & avoid potential issues like deadlock.

Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

The interrupt handling process is divided into two parts: the top half & bottom half.

The top half of the interrupt handling code is immediately occurs & the bottom half is a separate thread or task that is responsible for perform complex & time consuming operation.



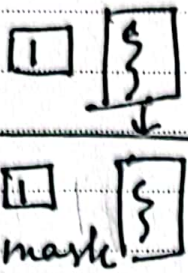
Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date

* Threads & signal Handling

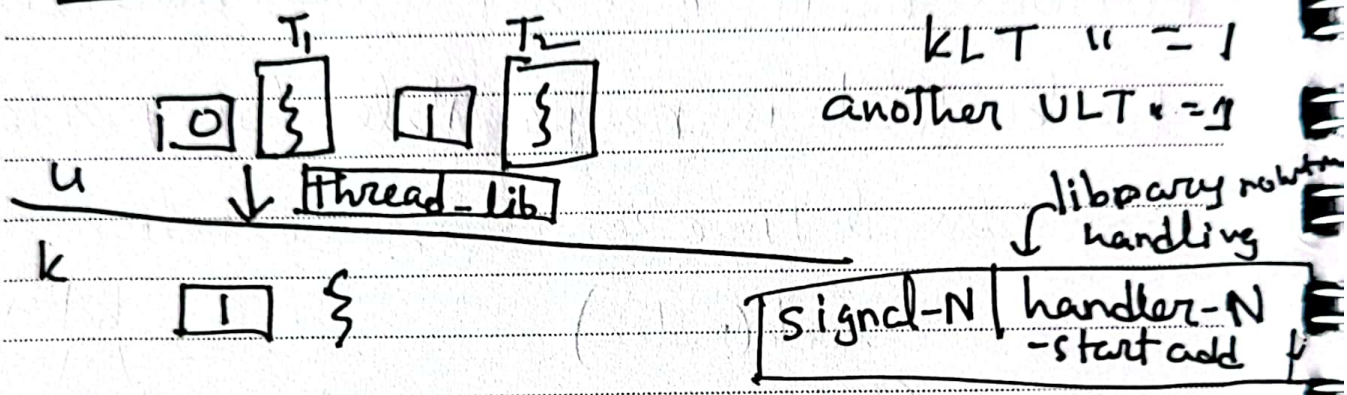
⇒ Case 1 : (both mask is one) so will interrupt The process & complete the interrupt.

ULT mask = 1
KLT " = 1



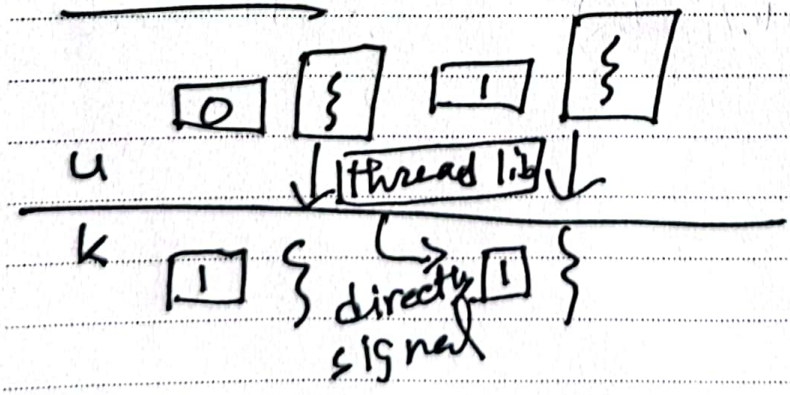
⇒ Case 2 :

ULT mask = 0
KLT " = 1
another ULT " = 1



stops T_1 & exit T_2 on KLT to handle signal.

⇒ Case 3 :



Sun | Mon | Tue | Wed | Thu | Fri | Sat

Date .....

Case 4: