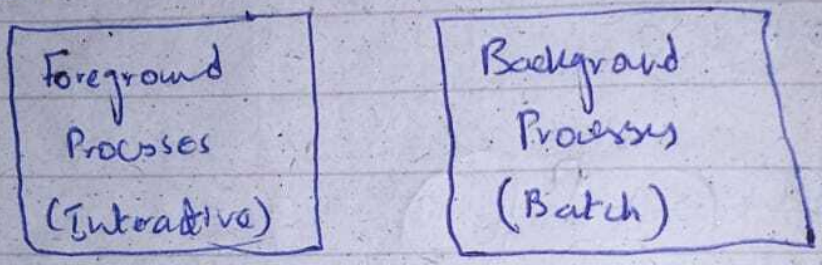
Multilevel Queue Scheduling:

- A class of scheduling Algorithms has been created for situations in which processes are easily classified into different groups.

Example:

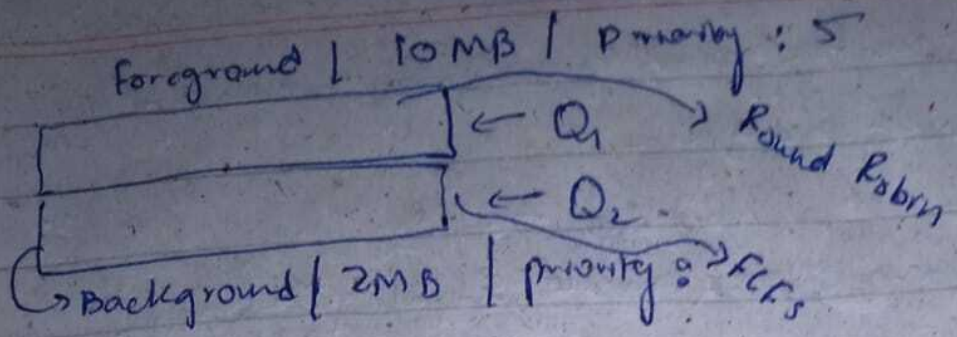



They have:

- Different response-time requirements
- Different Scheduling needs.

In addition, foreground processes may have priority (externally defined) over background processes.

A multilevel queue algorithm partitions ready queue into several seprate queues.




- The prousses are permanantly assigned to one queue, generally based on some propaty of the prass, such as memory size; process priority, or prouss type
- Each queue has its own scheduling algorithm.

Example:

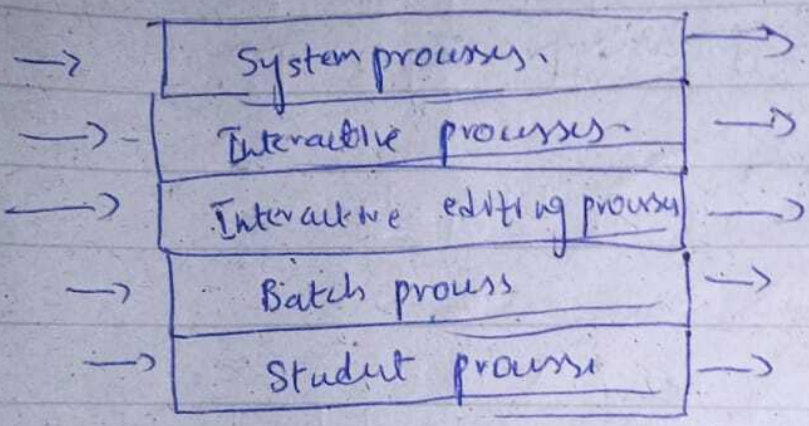
Sprate queus might be used for foreground and background processes.

The foreground queue might be scheduled by an RR algorithm, while background queue is scheduled by FCFS algorithm

In addition, there must be scheduling among queus, which is commonly implemented as fixed priority preemptive scheduling.

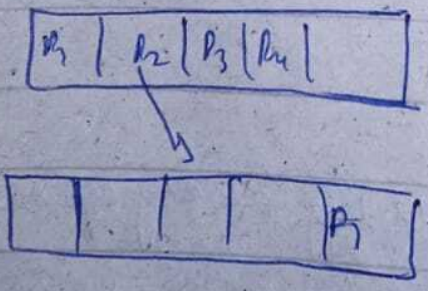
For example, the foreground queue may have absolute priority over background queue.

An example of a multilevel queue scheduling algorithm with five queues listed below in order of priority.




Multilevel feedback - Queue Scheduling;

- Move the processes among the queues




- The idea here is to separate processes according to the characteristics of their CPU bursts

- If a process uses too much ~~queue~~ CPU time, it will be moved to a lower priority queue;
- This scheme leaves I/O - bound and interactive processes in the higher priority queue.
- In addition, a process that waits too long in a lower priority queue may be moved to a higher priority queue.

This form of aging prevents starvation

Multiprocessor Scheduling:

- Multiple CPU's available → load sharing
- Where multiple threads may run in parallel, becomes possibles, however scheduling issues become correspondingly more complex. Many possibilities have been tried; and as we saw with CPU scheduling with a single core CPU there is no one best solution.

Multiple processor means:

- Multicore CPUs
- Multi threaded cores
- NUMA systems
- Hetroogenous multi prarsing.

Asymmetric: All os code runs on just one of the processor, so only one processor has access to system data structures. This avoids synchronization problems.

Virtually all modern ~~system~~ os supporting SMP system code can run on any processor. OS code on each processor schedules that processor. SMP can be used in conjunction with either, a common ready Queue for each processor ~~(configur~~

Access to a common ready queue has to be programmed carefully (synchronization problem).

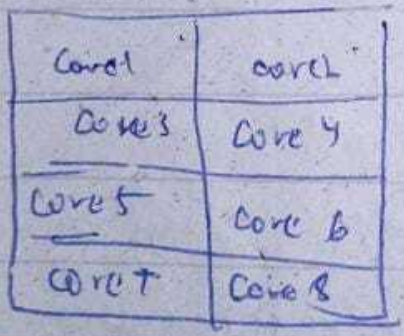
On other hand, load balancing can be problematic if there is a seperate ready queue for each processor what if some queues are empty and others are full.

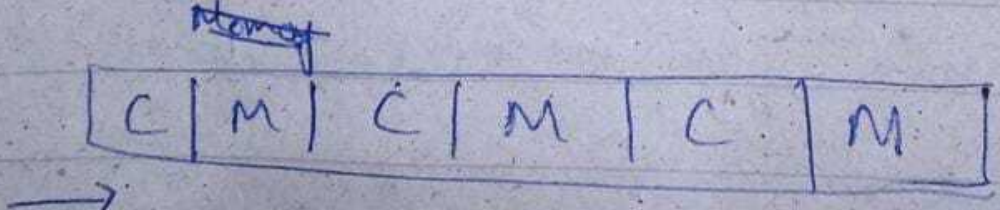
Multi core processors:

- Multicore processors may complicate scheduling issues.
- when a processor accesses memory it spends significant amount of time waiting for the data

to become available..

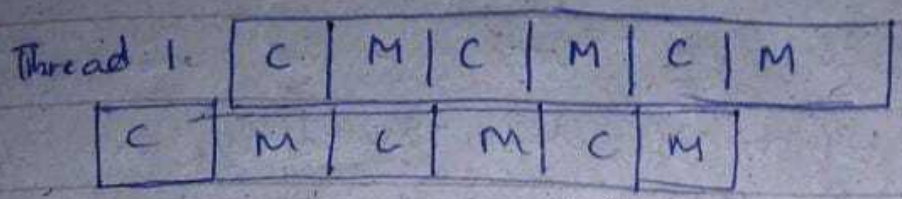
- This situation is known as memory stall; occurs primarily because modern processors operate at much faster speeds than memory.
- However a memory stall can occur because of a cache miss (accessing data that are not in cache memory).
  - Processor can spend upto 50% of time waiting for data to become available from memory




C - Computing cycle

M - Memory stall




Coarse grained: a thread executes on a core until a long latency event such as memory stall occurs. Because of the delay caused by long latency event, the core must switch to another thread to begin execution on the processor core. Once this new thread begins the execution, this it begins filling the pipeline with its instructions.

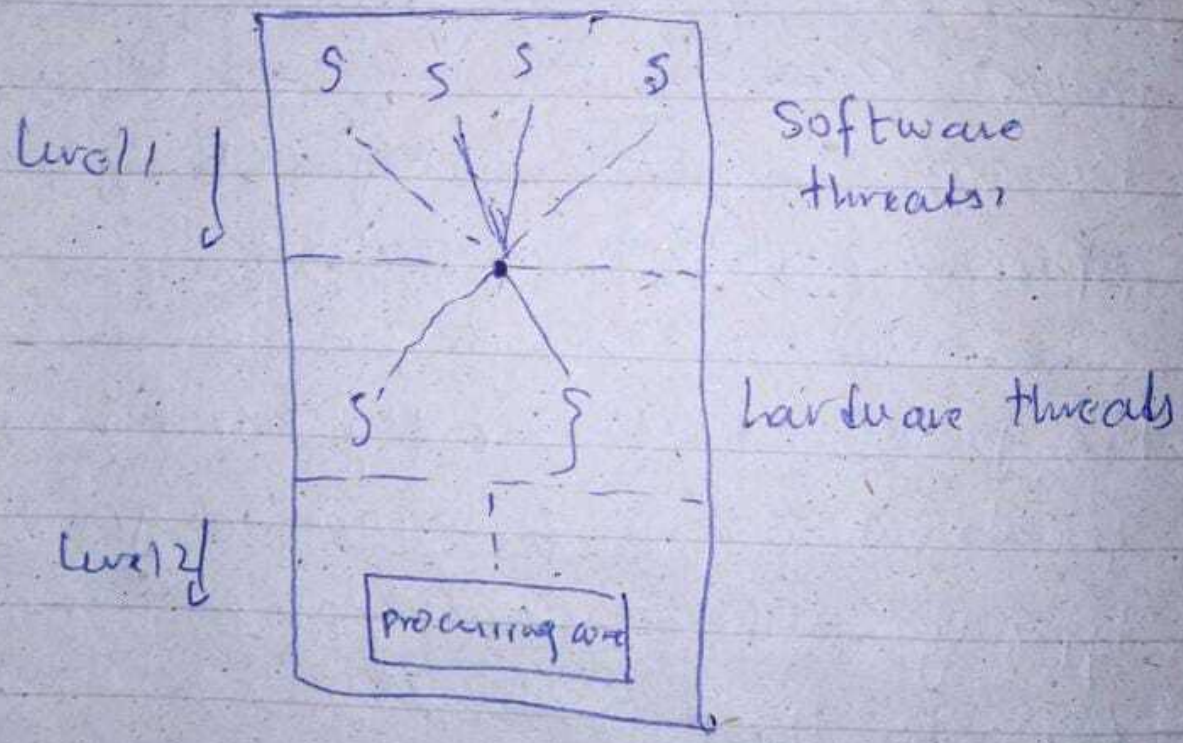
Fine grained: multi threading switches between threads at a much finer level of granularity typically at the boundary of an instruction. Logic for thread switching.

A

Thread scheduling.

Level 1: scheduling decisions that must be made by OS as it chooses which software thread to run on each hardware thread (logical CPU). Any scheduling Algorithm (Round Robin, Priority, SJF, FCFS)

Level # 2: How each core decides which hardware thread to run.




Two levels of scheduling.

LOAD Balancing:

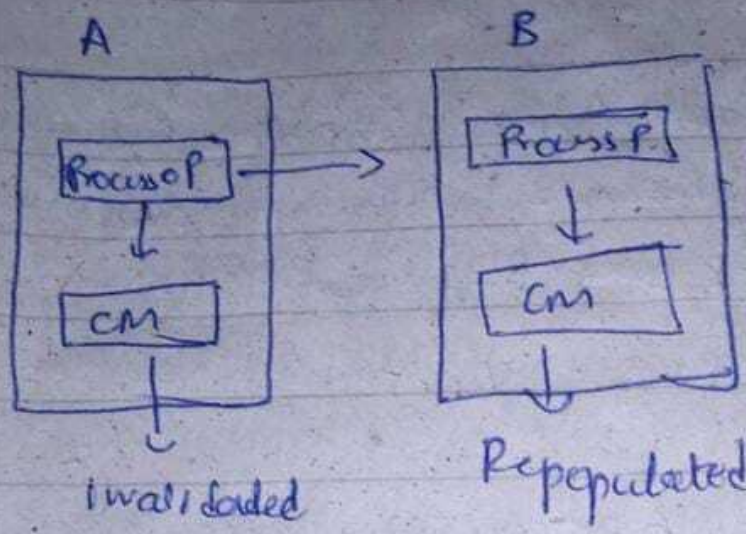
When each processor has a separate ready queue, there can be imbalance in numbers and/or kinds of jobs in the queues. Push and Pull migration are standard approaches to Load Balancing.

✓ Push migration: A process periodically checks ready queues and moves processes to different queues, if need be.

✓ Pull migration: OS code running on each processor notices when and tries to take jobs from another processor queues

Processor Affinity:

If a processor migrates from one instruction to another




(Keep on same processor)

most os with SMP try to avoid migrating thread from one processor

Expensive (Cache Memory of Processor Annialidated)

Hard affinity (Migration not allowed):

allowing processes to identify a subset of processors on which it can run.

Soft affinity (Migration MAY BE allowed):

When OS has a policy of attempting to keep a process running on the same processor — but not guaranteeing that it will do so.

only during load balancing

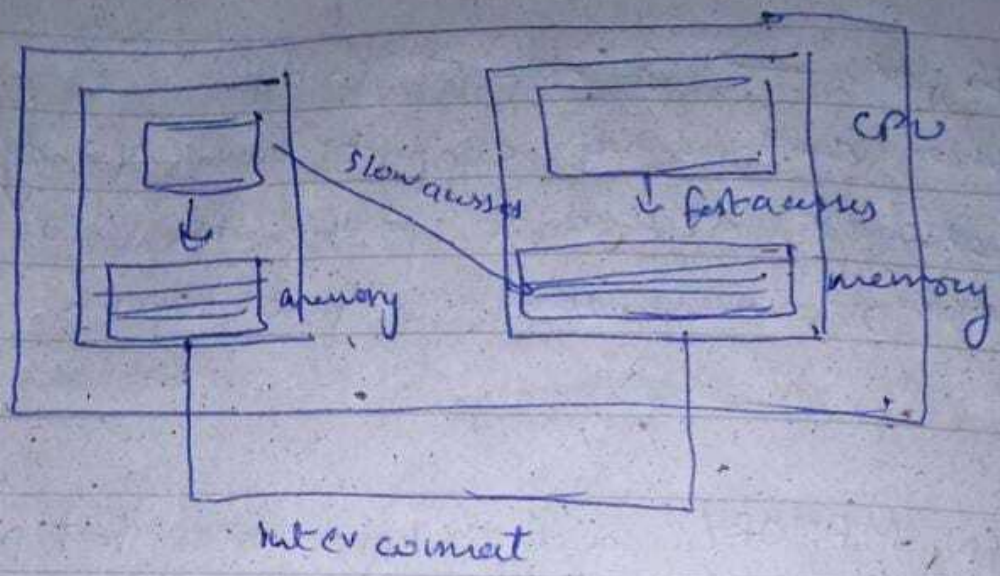
Linux uses ——> sched_set affinity().
system call, which supports hard affinity by allowing a thread to specify the set of CPUs on which it is eligible to run

* MAIN MEMORY architecture can affect processor affinity issues as well.

[Role of Processor affinity when NUMA is used]

NUMA disadvantages when Load Balancing take place.

When load balancing occurs (moving a thread from one CPU to another) the data which needs to be accessed by the thread is on the another memory block where the thread has been migrated from. And it takes a longer time to access the memory from first processor.




Hetrogenous Multiprocessing;

Combining a number of slower cores with faster ones, a CPU scheduler can assign tasks that do not require high performance, but required to run for longer periods, tasks that require little CPU time but high CPU power, are assigned to big cores.

Background tasks ⟶ little cores.

Interactive applications ⟶ Big Cores.

Real time CPU scheduling:

Soft Realtime systems: no guarantee as to when a critical realtime process will be scheduled. They guarantee that only processes will be given prefrence over non-critical processes.

Hard Real time Systems: have stricter requirements. A task must be serviced by its deadline. After the deadline is expired it is same as not service at all.

Interrupt latency: Time required to deal with the interrupt. till it is R starts.

Dispatch Latency: time of context switching.

# chap4

## Threads and Concurrency

- Most software applications that run modern computers and mobile devices are multi threaded.

- An application typically is implemented as a seprate proces with several threads of control. ~~Below we highlight a few examples~~ of

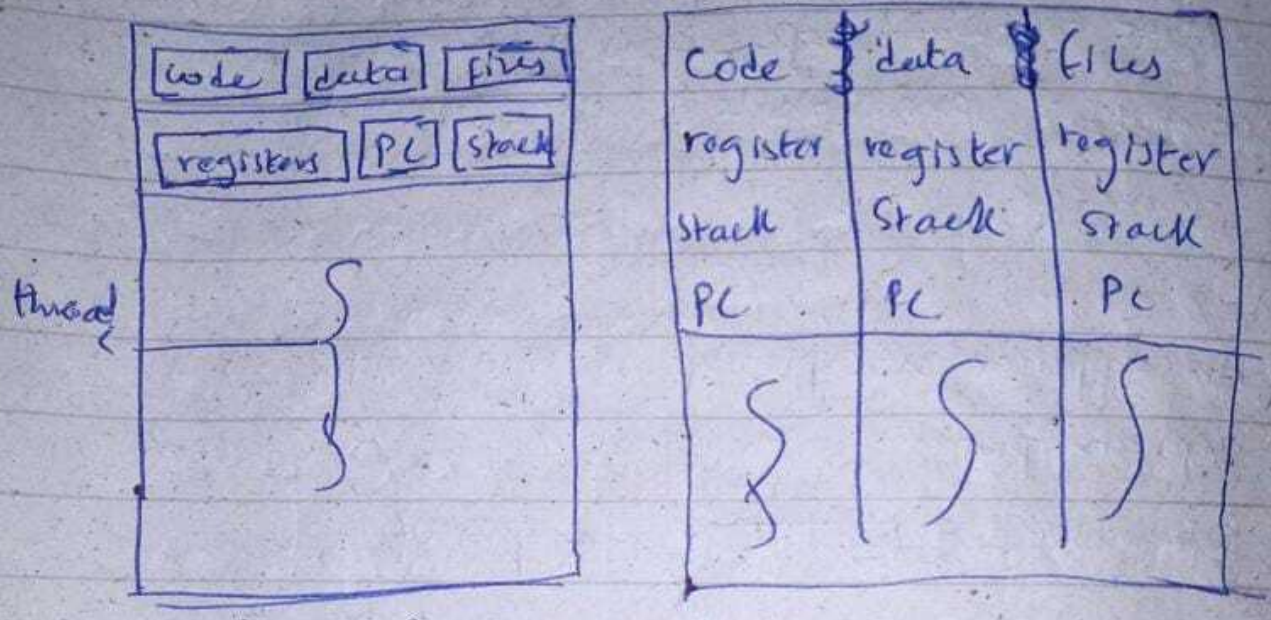
Multi threaded applications:

- An application that creates photo thumbnails from a collection of images. may use a seprate thread to generate a thumbnail from each seprate image.

- web browser

- Word processor.

Single threaded processes  multi threaded processes

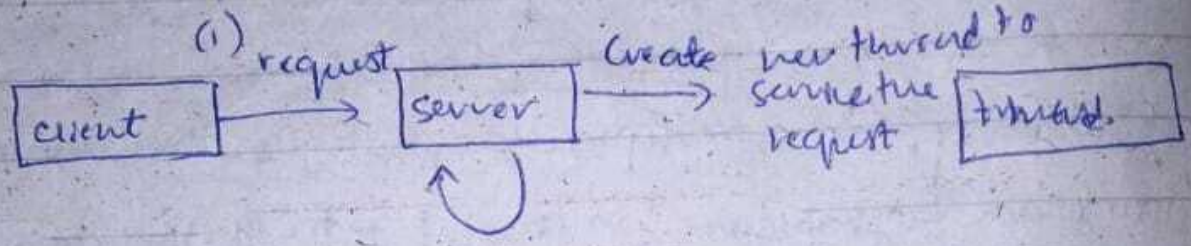



Multi threaded processes can handle multiple processes simultaneously. For example a client may request multiple services from a web server, or multiple clients might want to access a webserver.

Solutions:

1) Single process server. When server receives requests, it creates a separate process to service that request. In fact, this ~~creating~~. Process creation is time consuming, ~~however~~ creates an overhead (creating similar processes again and again)

2) It is better to use multithreaded prosses.
~~Multiple thr~~ Web browser should be multithreaded.
Seprate thread is created everytime user requests a service and resumes listening for additional request.




Benefits;

- Responsiveness; work ~~divided~~ is divided among threadsand some threads continue while others are ~~being~~ blocked (for example, waiting for I/O to complete). Usually used in user interface.

- Resource Sharing; Unlike seprate processes, threads can share memory and other resources by default which makes it easier for them to communicate.

and Cooperate. Also, there is efficient utilization of primary memory when threads share code and data

Economy: Not much needs to be done to add new thread to an existing process, or to perform context switch between different process.

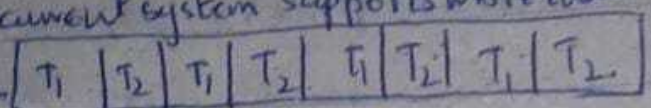
Scalibility: On a multiprocessor, multiple threads can work on a problem in parallel — truly simultaneously. A single threaded process can only run on one CPU at a time, no matter how many CPUs are available in the computer.

## Multi Core Programming:

- Separate computing units on single chip which can run multiple process simultaneously.
- improved Concurrency.
- Separate thread is assigned to each core.
- Each core can run single thread however a core with two threads could perform a efficient and quick context switch.

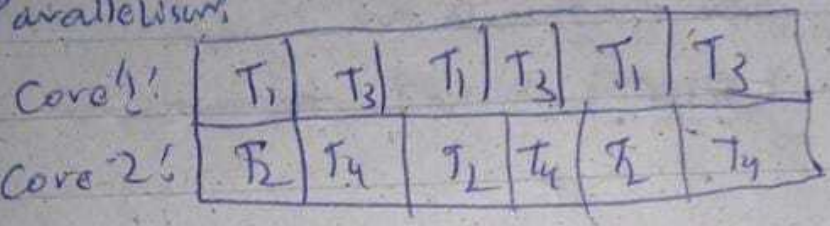
Concurrency:

A concurrent system supports more than one tasks by allowing all tasks to make progress.




Basically multitasking and content switching.
Single Core.

Parallelism:




Parallel system can perform more than one task simultaneously.

- Possible to have concurrency without parallelism.

4.2.1 Programming Challenges:

1) Identifying tasks: This involves examining applications to find areas that can be divided into seprate concurrent tasks.

2) Balance: In some instances task may not contribute as much value to the overall process as other tasks using aseprate execution core to run that may not be worth the cost.

tasks

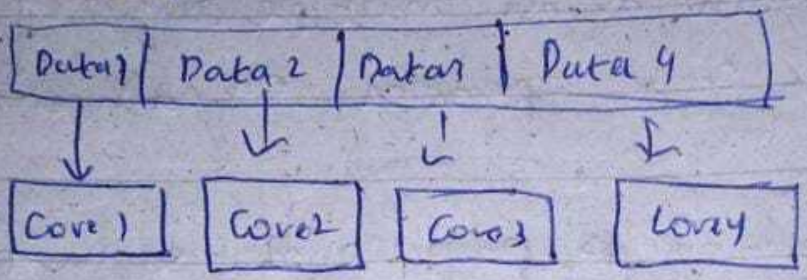
3) Data splitting: Like the ~~threads~~ are divided into seprate threads, data must also be divided. to run on seprate cores.

4) Data dependency: The data accessed by the task must be examined for dependencies between two or more tasks. When one task depends on data from another, programmer must ensure that the execution of the tasks is synchronized.

5) Testing and debugging: When tasks are running on multiple cores ~~concurrently~~ parallely. testing becomes difficult.

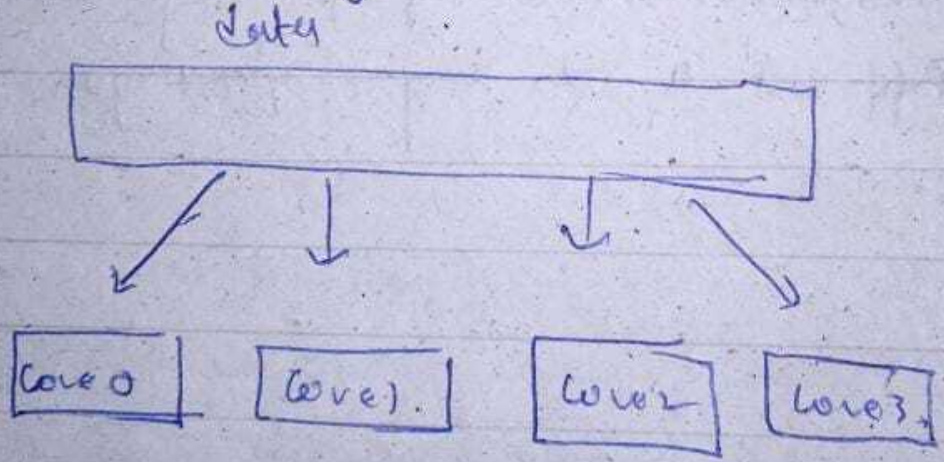
Data parallelism:

Concurrent execution of same task on each multiple computing core




Task Parallelism:

Concurrent execution of different task on multiple computing cores.



Types of Parallelism
(Data and Task parallelism):

Data parallelism: focuses on distributing subsets of the same data across multiple computing cores and performing the same operation on each core.

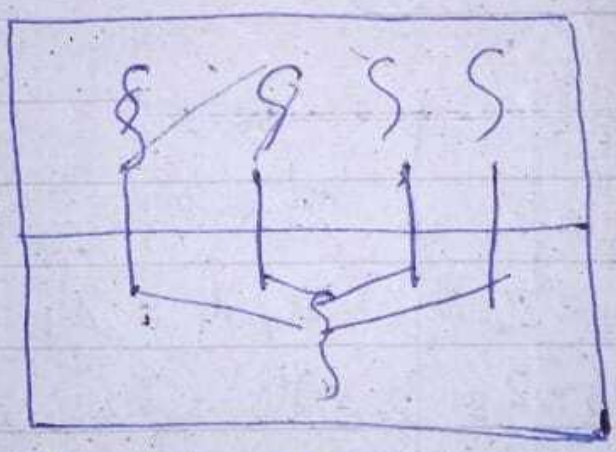
Source collection is partitioned so that multiple threads can operate on different segments concurrently.

Task parallelism: tasks are distributed across multiple computing cores. Each thread is performing a unique operation. ~~Different threads may be operating on~~

Multi threading Models:

- Many to one Model: maps user level threads to one kernel Many to thread. Thread Management is done by thread library in user space.

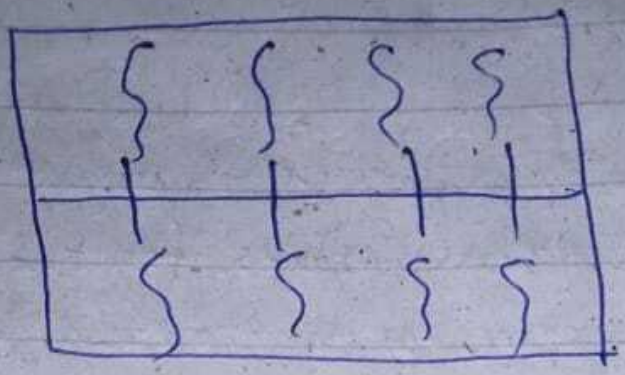
- Multiple threads are unable to run in parallel in multicore systems.



Many to one

One to One:

Maps each user thread to a kernel thread. It provides more concurrency than many to one model by allowing another thread to run when a thread makes blocking system call.



One to one Model.

Many to Many model:

multiplexes many user level threads to a smaller or equal number of kernel threads

- Number of kernel threads can be specific to a particular application or a particular machine (an application may be allocated more kernel threads on a system with eight processing cores than a system with four cores.)

Thread libraries:

- Libraries in userspace with no kernel support. All code and data structures for the library exist in user space with no kernel support. This means invoking function in library results in local function call, in user space and ~~system~~ not system calls.

Kernel level library directly supported by the OS. In this case, code and data structures for the library exist in Kernel space. Invoking a function in API for the library typically results in system call to kernel.

1. Posix Pthreads: It may be as a user or kernel library as an extension to the POSIX standard.

Asynchronous threading: parent creates a child thread. The parent resumes its execution, so that parent and child can execute concurrently and independently. There is very little data sharing between them.

Synchronous threading: occurs when parent threads create one or more children and wait for all of its children before it resumes. Threads created by parent (children threads) work concurrently, but parent cannot continue until its all childs have not completed the execution. Each thread joins itself to parent. Involves data sharing among threads.

P threads:

Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization.

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

void * routine () {
    printf ("Hello from threads \n");
    sleep(3);
    printf ("Ending threads \n");
}

int main() {
    pthread_t t1, t2;
    if (pthread_create(&t1, NULL, &routine, NULL
        perror ("Thread cannot be created");
    }
    if ( pthread_create (&t2, NULL, &routine, NULL)
        perror ("Thread Cannot be created");
    }
    if(pthread_join(t1, NULL)) {
        perror(" ----");}
    if (pthread_join(t2, NULL)) {
        perror (" ----"); }
```

Date

Pthreads:

Posix standard (IEEE 1003.1c) defining an API for thread creation and synchronization. Specification for thread behaviour

pthread_t → used for creating threads
declares the identifier for thread we will create.

Creating a thread t1 syntax:
pthread_t t1;

Pthread_create → function for initializing thread and creating thread.

Syntax P_thread_Create;
p_thread (&t1, NULL, &routine, NULL);
- &t1 → address of t_id.
- NULL → attributes address.
- &routine → reference of the function / task
- NULL → reference of where you want to create threads

pthread_join → joins all threads to parent thread. (basically acts same as "wait(NULL);")
parent process thread waits for its children threads to compute results and parents then combine results and use them collectively.

Syntax:
pthread_join(t1, NULL);

Date

Thread pools:

whenever Server recieves a request, it creates a seprate thread to service the request

Potential problems of multithreaded severs:

- Amount of time required to create the thread
- Threads should finish discard it self after its task has been completed.

→ if we allow each concurrent request to be serviced in a new thread, we have not placed a bound on the number of threads concurrently active in the system. Unlimited threads could exhaust the systems resources such as CPU time or memory.

What is thread pool?

number of threads are created and in start and placed in a pool, where they sit and wait for work.

Instead of creating an new thread an new request is serviced by threadpool. and thread pool keeps waiting for new requests. If there are no available threads tasks are queued. until one becomes free. Once a thread completes its service, it returns to the pool and awaits for more work.

Date

Benifits of thread pools:

1.) Servicing request with an existing thread is often faster than waiting to create a thread.

2.) A thread pool limits the number of threads that exists at any one point.

3).

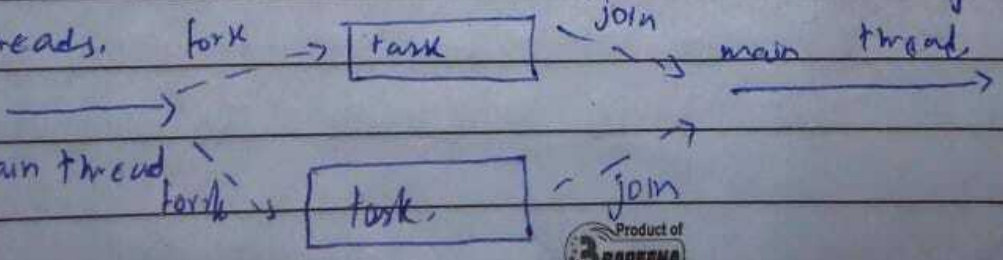
Number of threads in the pool can be heuristically based on factors such as number of CPUs:

- amount of physical memory.
- number of concurrent client requests.
- number of CPUs in the systems

Fork join: (explicit)

Parent thread creates one or more child threads and wait for the children to terminate and join, at which point it can retrieve and combine their results.

(implicit): threads are not constructed directly. during fork stage; rather parallel tasks are designated. A Library manages number of threads that are created and also responsible for assigning tasks to threads.



Date

Signal Handling:

A signal is used in UNIX systems to notify a process that a particular event has occured. A signal may be recieved synchronously or asynchronously.

Synchronous signals:

- illegal memory access
- division by 0

Synchronous signals are delivered to same process that performed the operation that caused the signal (that is the reason they are considered synchronous).

a synchronous signals:

Signal is generated by an event external to running Process, that process recieves asynchronously.

Examples:

- terminating a process (Ctrl+C)
- having timer expire

A signal may be handled by one of the two possible handlers:

- A default signal handler.
- A user defined signal handler.

Date

Default signal Handler can be overridden by a user defined signal handler

Signals delivered in multithreaded programs:

1) thread to which signal applies.
2) to every thread in the process
3) to certain threads in the process.
4) to recieve all signals for the process.

Thread Scheduling:

- The thread that are being scheduled by OS are known as Kernel threads.
- User level threads are managed by thread library and kernel is unaware of them.

- Kernel (OS) donot schedule processes, it schedules threads.
- Thread is a schedulable entity, not a process.

- On systems implementing many to many or ~~one~~ many to one the thread library schedules user level threads to run on available LWP. This scheme is known as Process Contention scope. Since competition for CPU takes place among threads belong to same process.

Date

- To decide which kernel-level thread to schedule onto a CPU, the kernel uses system-contention scope (SCS)

- Systems using one to one model such as windows and Linux schedule threads using SCS.

- Typically, PCS is done according to priority - the scheduler selects the runnable thread with the highest priority to run.

- User-level thread ~~priority~~ priorities are set by programmer and are not adjusted by thread library, although some thread libraries may allow programmer to change priority of a thread.

- PCS will typically preempt the thread currently running in favor of a higher priority thread; however there is no guarantee of time slicing.

OPEN MP:

Set of compiler directives as well as an API. Program written in C, C++, or FORTRAN that provides support for parallel programming in shared memory environments.

- app devs insert compiler directives into their code at parallel regions ~~at~~ parallel regions and these directives instruct the OPEN ~~runtime~~ runtime library to execute the regions in parallel.

Date

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

int main (int argc, char* argv[]){

    #pragma omp parallel {
        printf (" I am a parallel region.");
    }
    /* sequential code */
    return 0;
}
```

- When OPENMP encounters the directive, it creates as many threads as there are processing cores in the system. All the threads then execute the parallel region. As each thread exits the parallel region, it is terminated.

OPEN MP provides several additional directives for running code regions in parallel, including parallelizing loops.

```
#pragma omp parallel for {
for (i=0; i<N; i++) {
    c[i] = a[i] + b[i];
}
```

- Open MP also allows devs to choose among several levels of parallelism. For eg. they can set number of threads manually.

Date

- It also allows devs to identify whether data are shared b/w threads or are private to a thread.

Grand Central Dispatch:

- GCD schedules tasks for runtime execution by placing them on a dispatch queue.

- removes the task from the queue and assign the task to an available thread from a pool of thread.

Types of Dispatch Queues:

Serial: tasks are removed in FIFO order. Once the task has been removed from the queue, it must complete execution before another task is removed. Each process has its own serial queue. Serial queues are useful for ensuring sequential execution of several tasks.

Concurrent: removed in fifo order. But several tasks may be removed at a time, thus allowing multiple tasks to execute in parallel. There are several system wide concurrent queues (also known as global dispatch queues).

Date

Scheduler Activation:

Kernel provides an application with a set of virtual processes (LWPs), and the application can schedule user threads onto an available virtual processor. Furthermore, the kernel must inform an application about certain events. This procedure is known as upcalls. Upcall is handled by the thread library with an upcall handler, and upcall handler must run on a virtual processor.