Scheduling Criteria.

Different CPU-scheduling algorithms have different properties. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.

Many criteria have been suggested for comparing CPU-scheduling algorithms. The criteria include the following:

CPU utilization:

We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent.

In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system)

Throughput:

If the CPU is busy executing processes, then work is being done.

One measure of work is the number of processes that are completed per time unit, called throughput.

- Turn around time

The interval from the time of submission of a process to the time of completion is the turnaround time.

Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Waiting time

The CPU - scheduling algorithm does not affect the amount of time during which a process executes or does I/O.

It affects only the amount of time that a process spends waiting in the ready queue.

Waiting time is the sum of the periods spent waiting in the ready queue.

Response time

The time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time and response time.

In most case, we optimize the average measure.

However, under some circumstances, we prefer to optimize the minimum or maximum values rather than the average. For example; to guarantee that all users get good service, we may want to minimize the maximum response time.

Scheduling algorithms

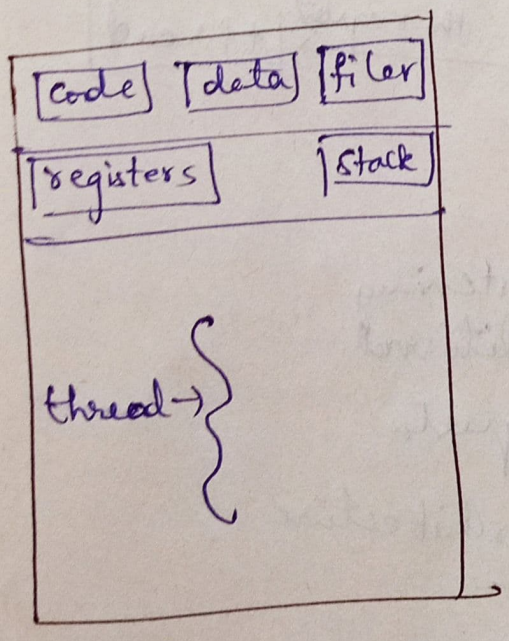
- FCFS scheduling
- SJF scheduling
- Priority
- Round-Robin
- Multilevel Queue
- Multilevel feedback queue

THREADS

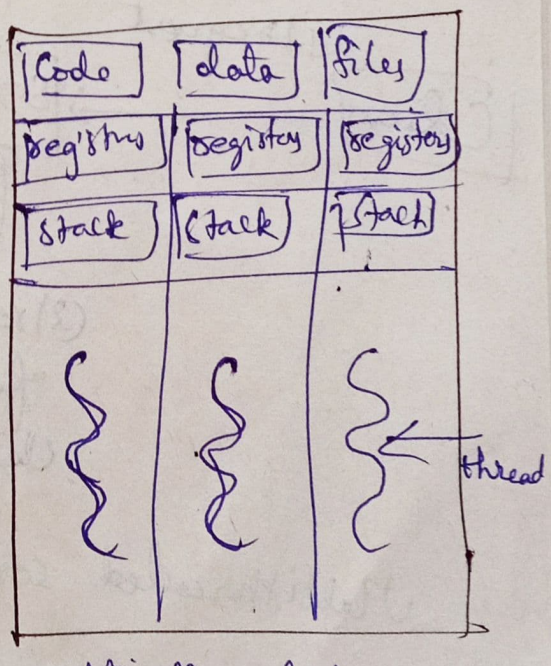
Overview

* A thread is a basic unit of CPU utilization
* It comprises a thread ID, a program counter, a register set and a stack.
* It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.
* A traditional (or heavyweight) process has a single thread of control.

If a process has multiple threads of control, it can perform more than one task at a time.



single-threaded process



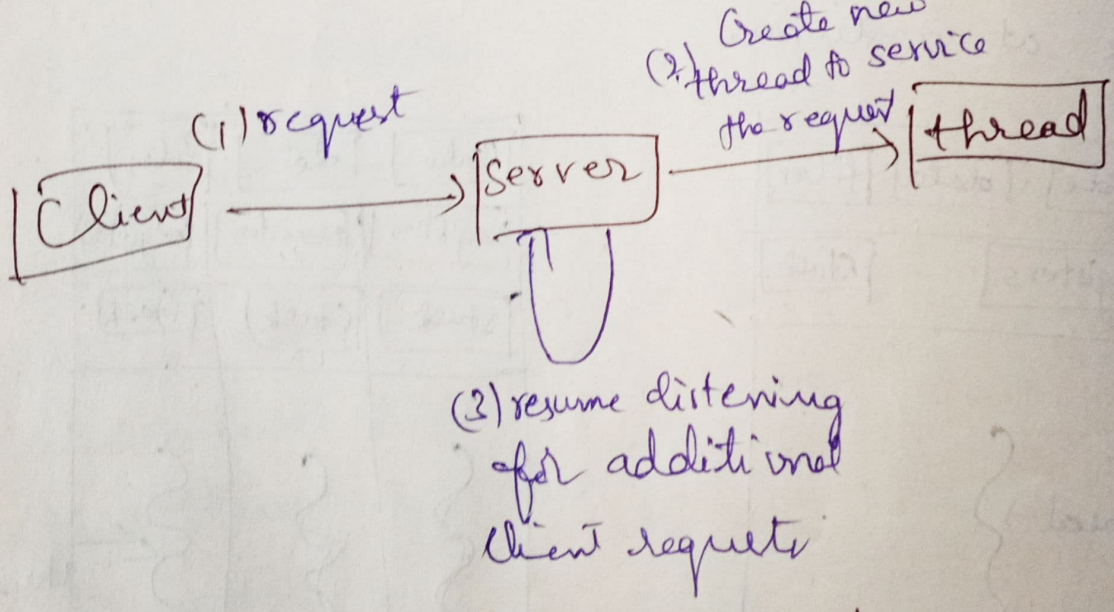
multi-threaded process

Most s/w applications that run on modern computers are multithreaded.

An application typically is implemented as separate process with several threads of control.

For example; A word processor may have a thread for displaying graphics, another thread for responding to keystrokes from the user, and third thread for performing spelling and grammar checking in the background.

Threads also play a vital role in remote procedure call (RPC) systems. ~~Recall~~ Typically RPC servers are multithreaded.



Multithreaded server architecture.

Multithreading Models

Ultimately, a relationship must exist between user threads and kernel threads.

→ Three common ways of establishing such a relationship:

- → Many-to-one model
- → One-to-one model
- → Many-to-many model

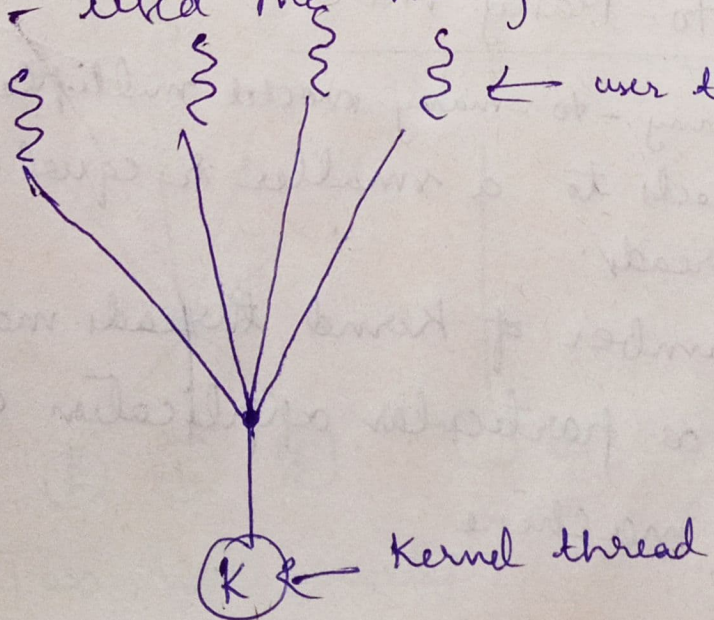
Many-to-one Model

→ The many-to-one model maps many user-level threads to one kernel thread.

→ The entire process will block if a thread makes a blocking system call.

→ Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.

Green threads - A thread library available for Solaris systems and adopted in early versions of Java - used the many-to-one model.



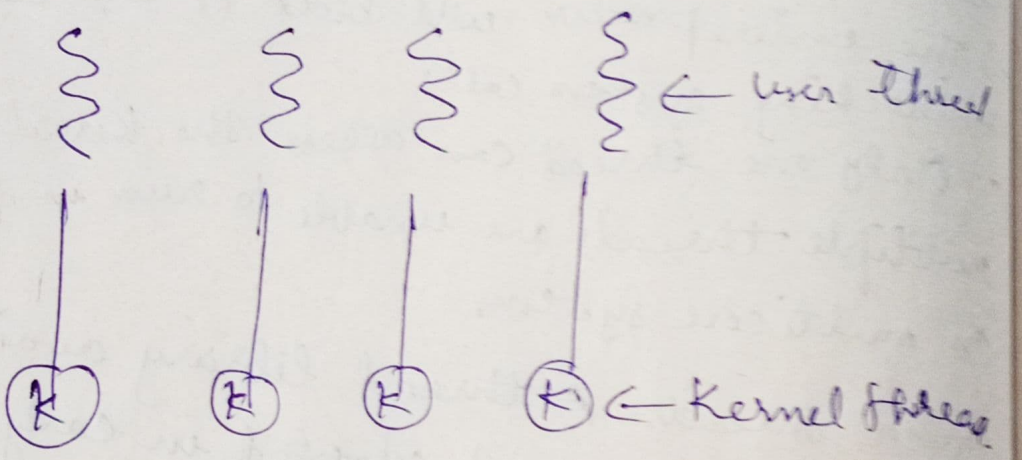
One-to-one model

→ The one-to-one model maps each user thread to a kernel thread.

→ It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.

→ It also allows multiple threads to run in parallel on multiprocessors.

→ The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.

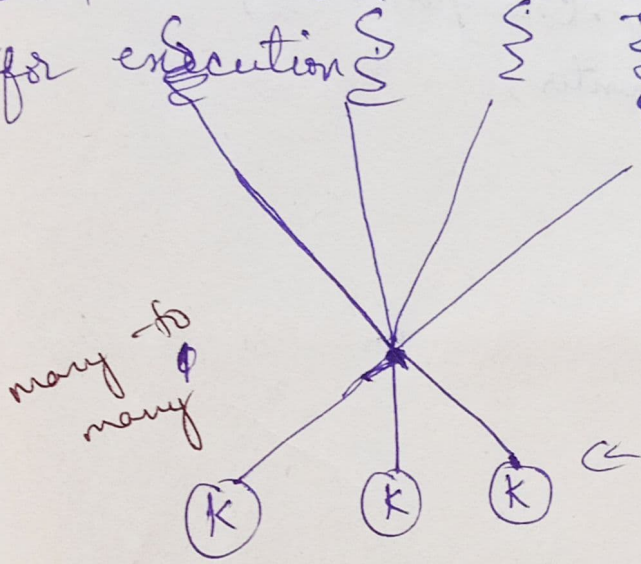


Many-to-Many model

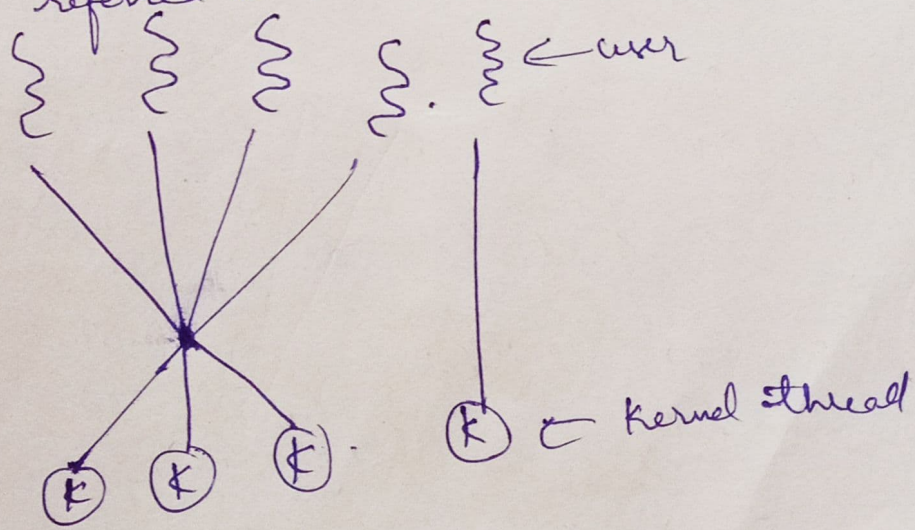
→ The many-to-many model multiplexes many user level threads to a smaller or equal number of kernel threads.

→ The number of kernel threads may be specific to either a particular application or a particular machine.

The many-to-many model suffers from neither of these shortcomings:
developer can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution



One variation on the many-to-many model still multiplexes many user-level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the two-level model



Two-level model