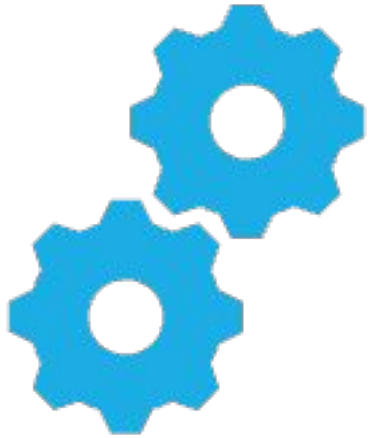
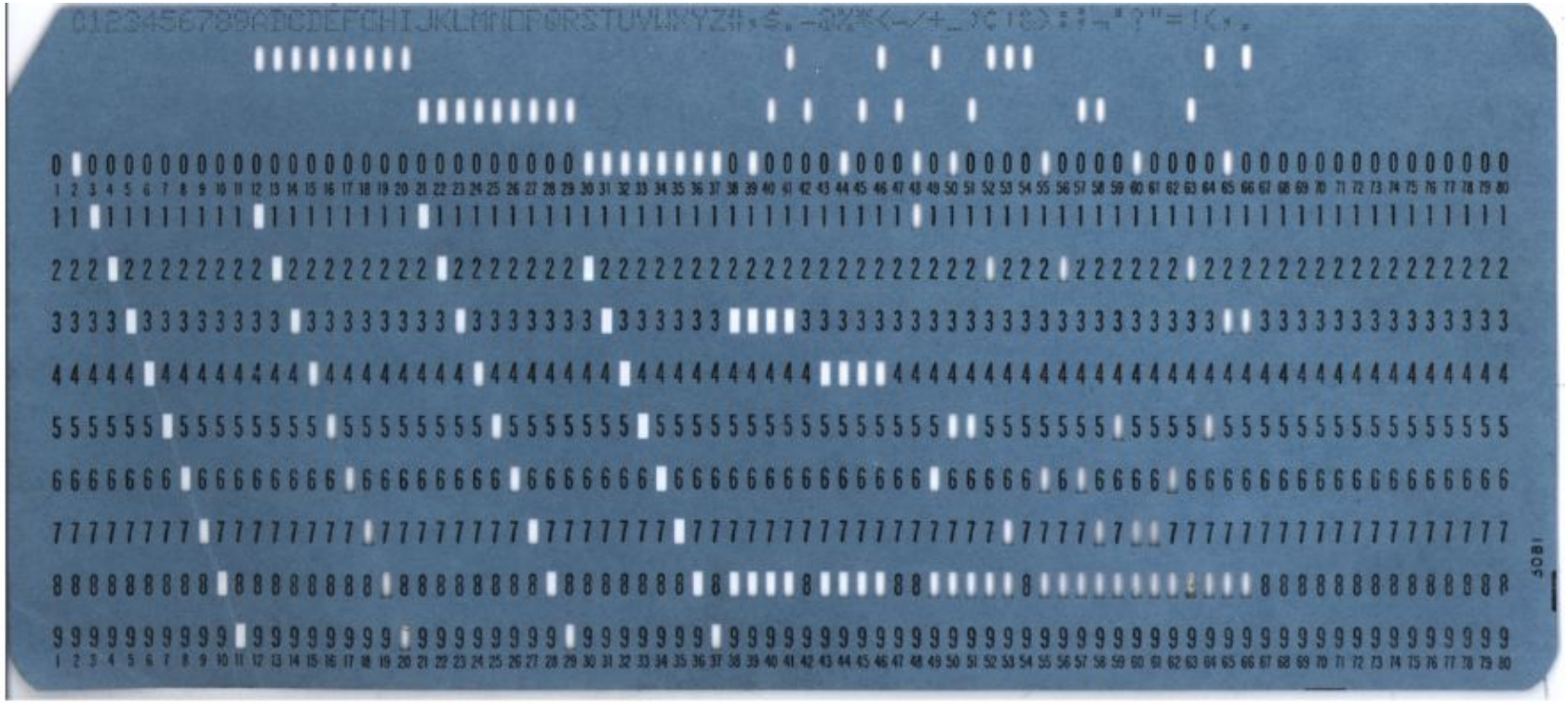


Processes and threads



A glimpse of the past

What is a program?

A program is a set of instructions that tells the computer how to do something, usually in secondary storage.

A program passes through multiple states to transform source code into an executable file.

There are multiple types of executables, depending on the OS:

Windows uses the Portable Executable format (PE)

Mac OS/iOS uses Mach object file format (Mach-o)

Linux uses Executable and Linkable Format (ELF)

Compiling, Linking and Loading

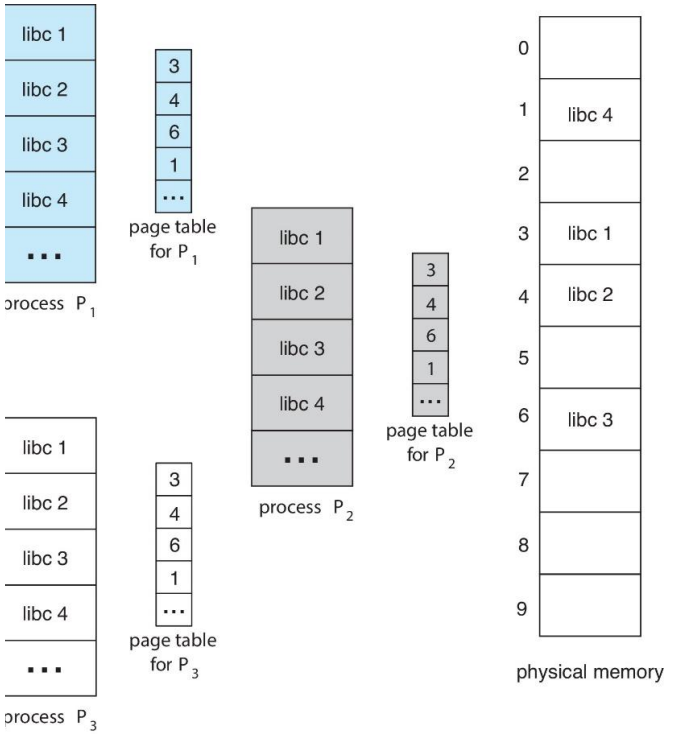
When compiling, the source code is converted to object code.

Linking adds the functions from the libraries and syscalls.

- Static linking copies the libraries into the executable
- Dynamic linking adds a stub for calling the library. The stub is changed by the function address

An interpreted language has a compiled interpreter, and the interpreter is the one that links all the libraries.

Loading copies the program into memory. A program in memory is called a process.



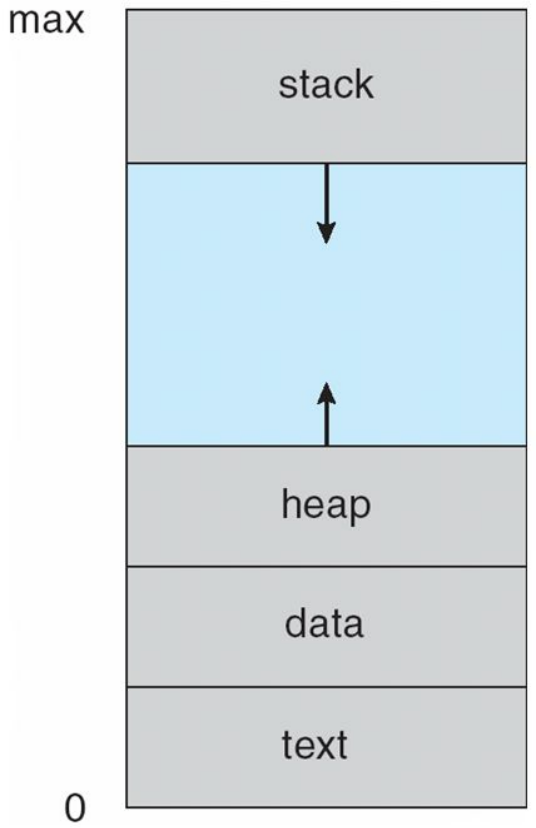
A process in memory

Text: The code of the program

Data: Global and initialized data.

Stack: Temporary data

Heap: Memory allocated dynamically.



Process Control Block

Process state – running, waiting, etc

Program counter – location of instruction to next execute

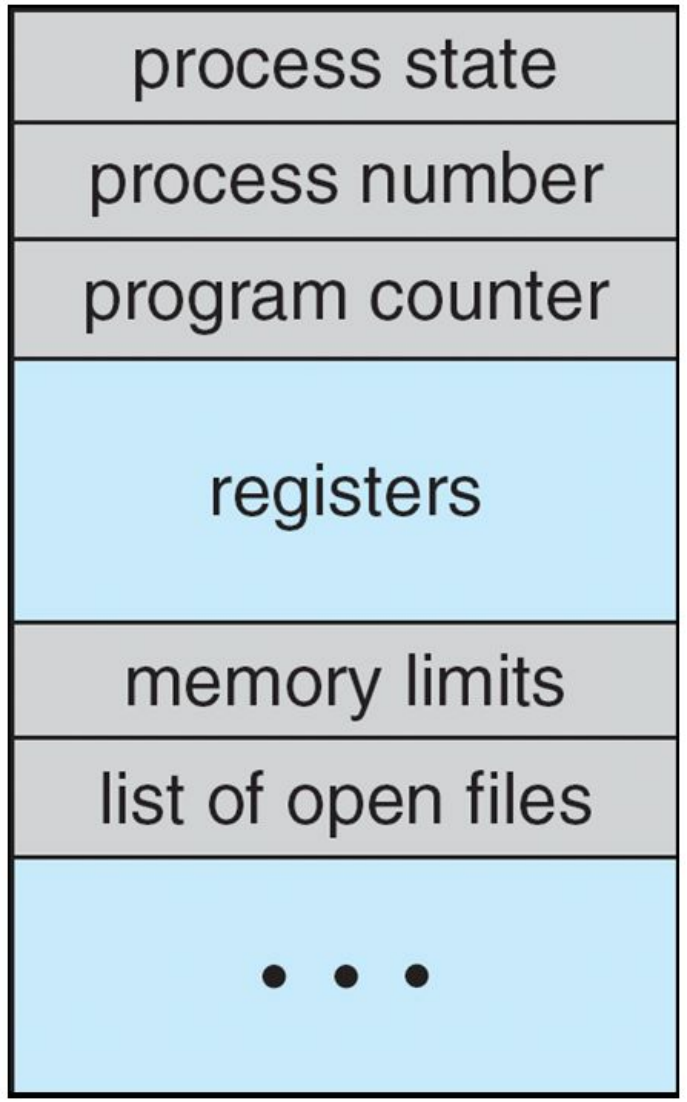
CPU registers – contents of all process-centric registers

CPU scheduling information- priorities, scheduling queue pointers

Memory-management information – memory allocated to the process

Accounting information – CPU used, clock time elapsed since start, time limits

I/O status information – I/O devices allocated to process, list of open files



Process States

New: The process is created.

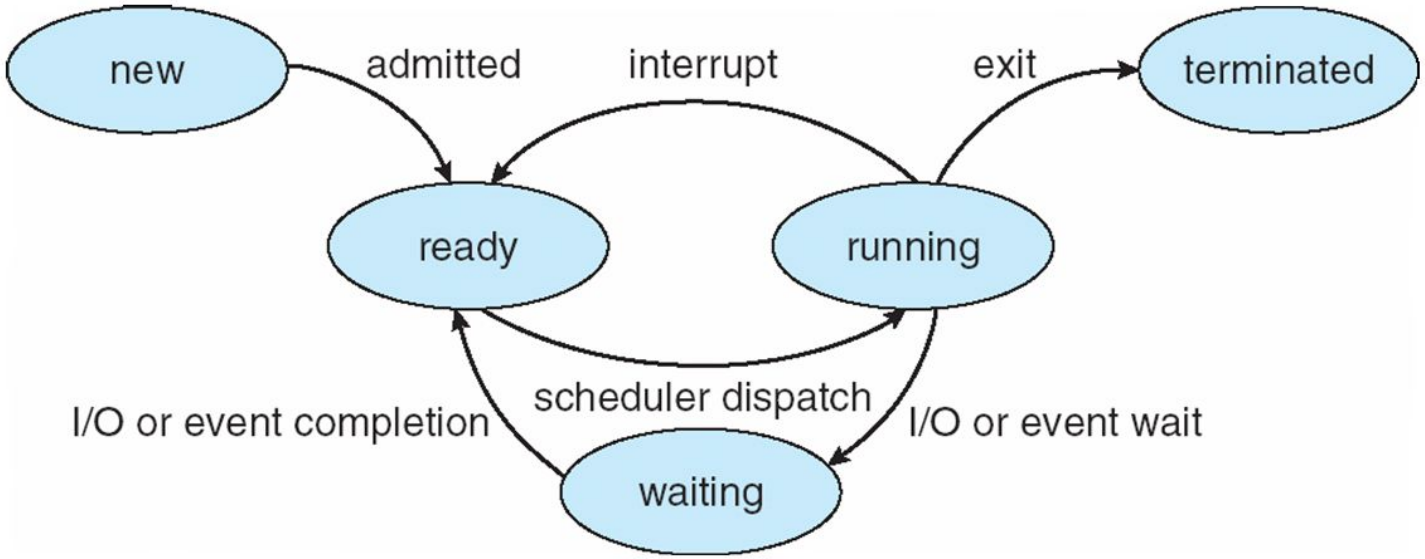
Running: The process is being executed.

Terminated: The process finish.

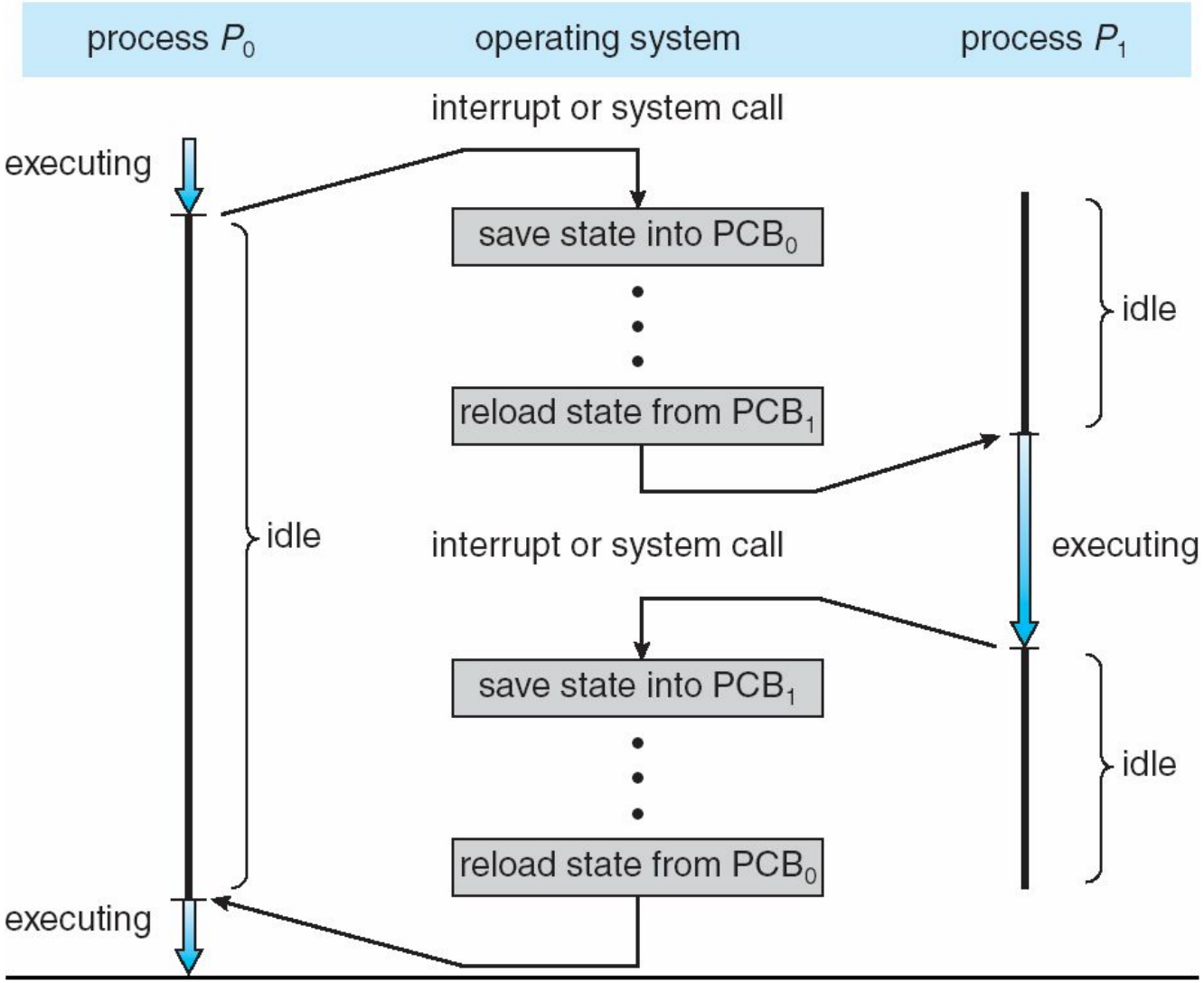
Ready: The process is waiting to execute in the CPU

Waiting: The process is waiting for an event to occur.

Suspended: The process is taken out of memory



Context Change



Process Creation

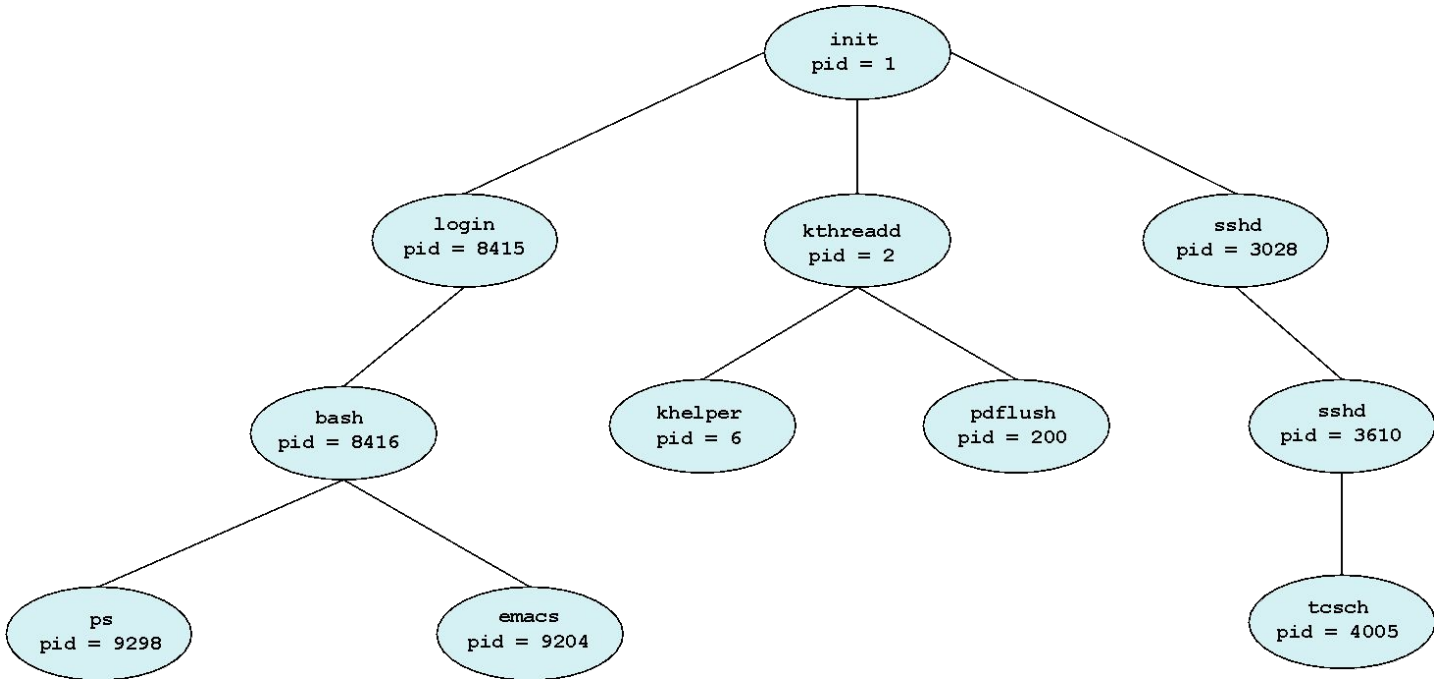
Most processes have a parent process, which creates or invoke them

- Systemd, initd in Linux, System in Windows

Processes are identified by a Process id

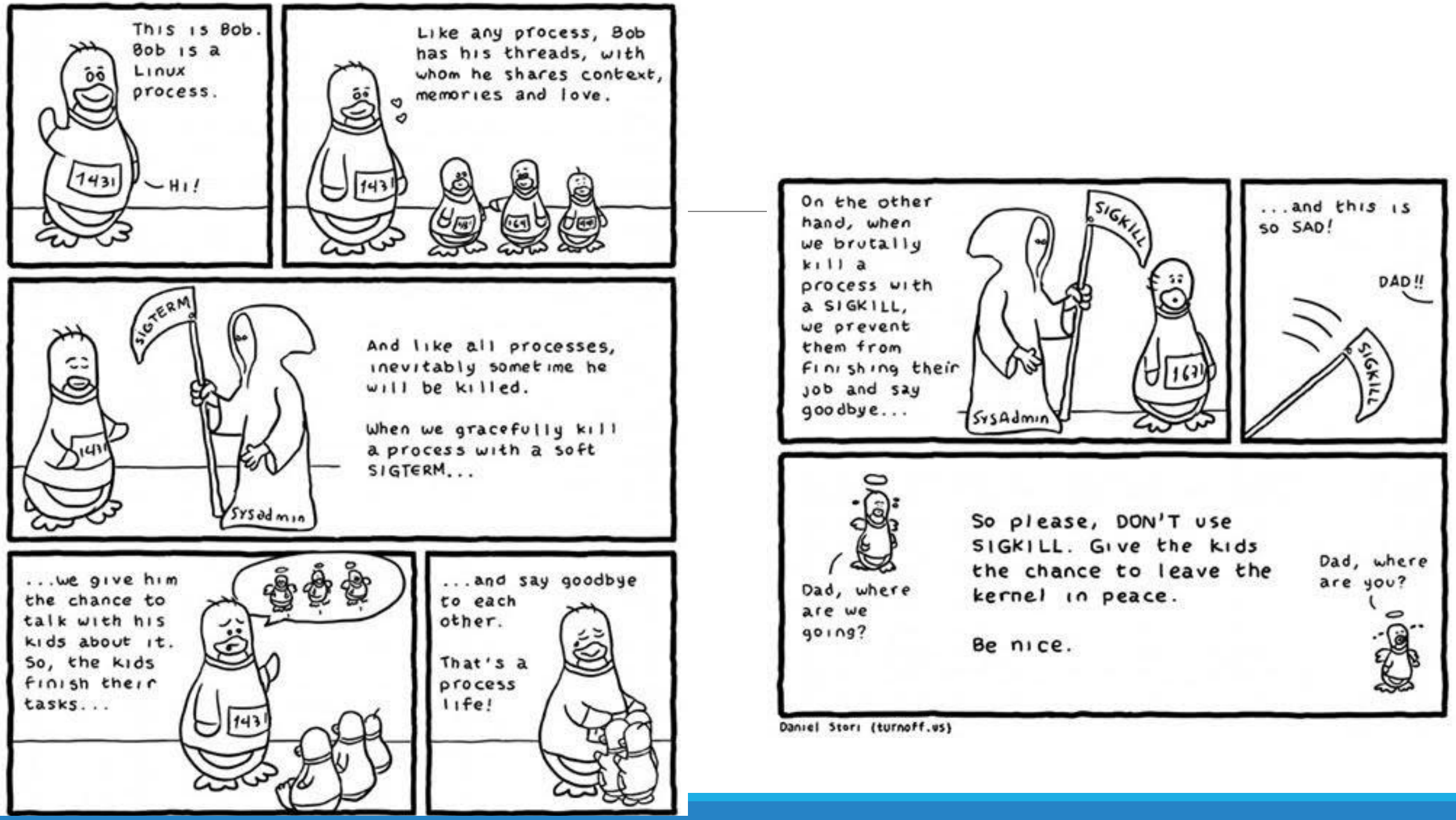
Children share the resources of the parent and the program

- Fork creates a child with the same address space of the parent
- Exec replaces the program and address space of the child



Process Termination

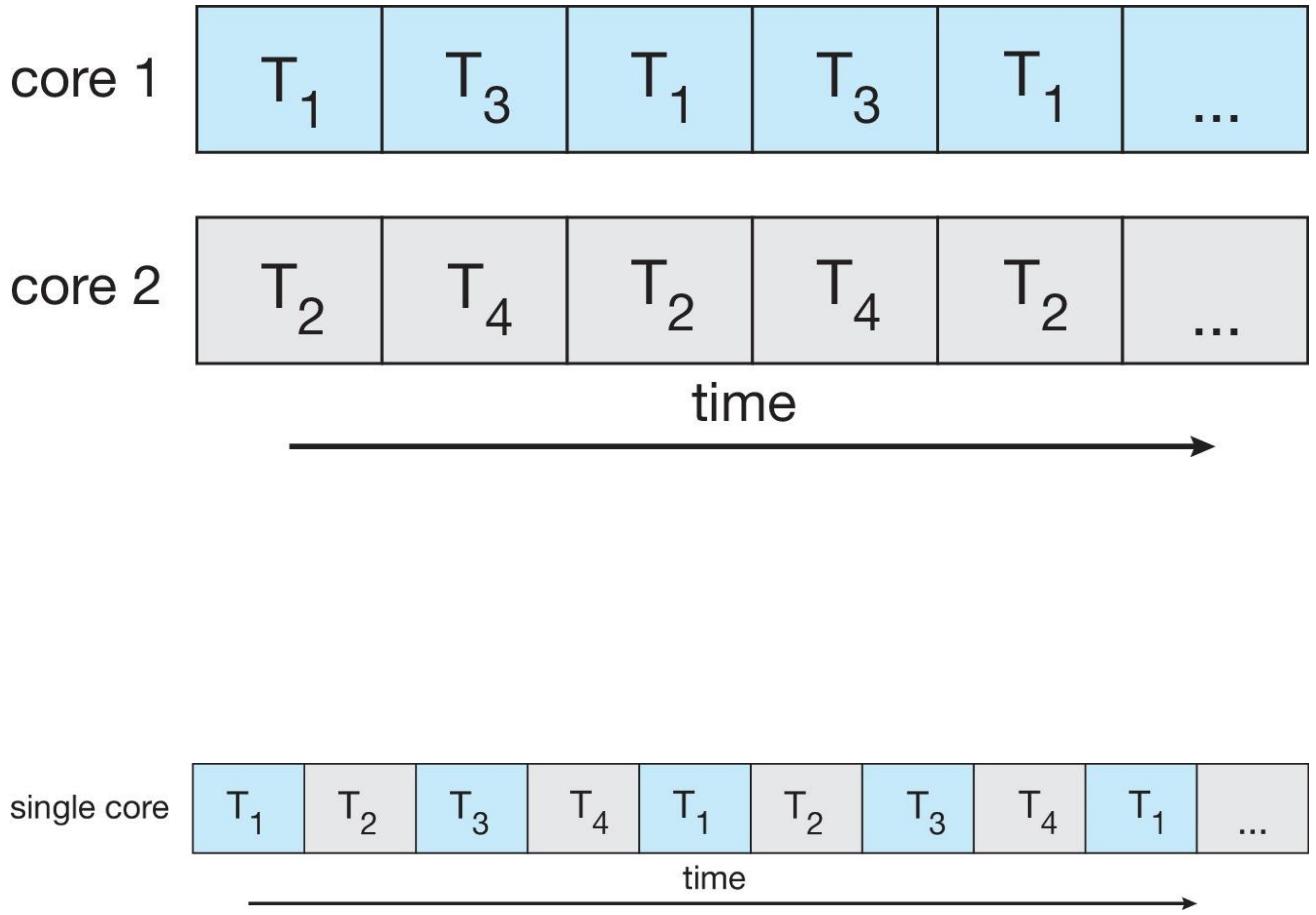
- A process ends by calling exit, and allow the OS to deallocate their resources.
- When a process exits, it notifies its parent
- Parents should be waiting for their child processes to finish.
- If a parent finishes, some operating systems enforce that all the children finish too
- If no one waits for the children to exit, it becomes a zombie.
- If the parent ends, and it didn’t issued the wait call, the child becomes an orphan.

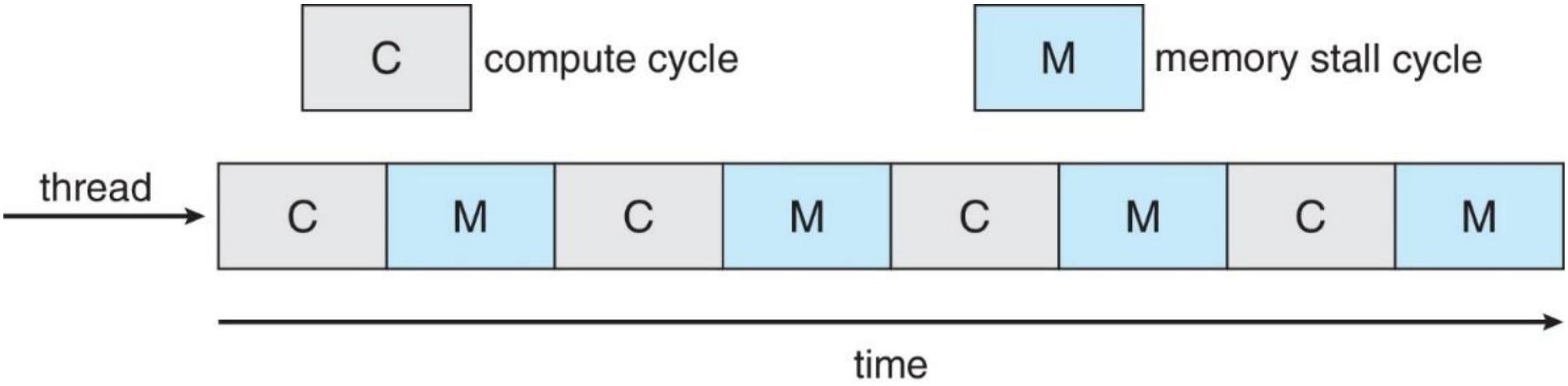

This is Bob. Bob is a Linux process.
1431
Hi!
Like any process, Bob has his threads, with whom he shares context, memories and love.
1431
SIGTERM
1431
SysAdmin
And like all processes, inevitably sometime he will be killed.
When we gracefully kill a process with a soft SIGTERM...
...we give him the chance to talk with his kids about it. So, the kids finish their tasks...
1431
...and say goodbye to each other.
That's a process life!
On the other hand, when we brutally kill a process with a SIGKILL, we prevent them from finishing their job and say goodbye...
SIGKILL
SysAdmin
1431
...and this is so SAD!
DAD!!
SIGKILL
Dad, where are we going?
So please, DON'T use SIGKILL. Give the kids the chance to leave the kernel in peace.
Be nice.
Dad, where are you?
Daniel Stori (turnoff.us)

Parallelism vs concurrency

Concurrency: more than one task/process advances in its execution

Parallelism: two or more processes are running simultaneously.

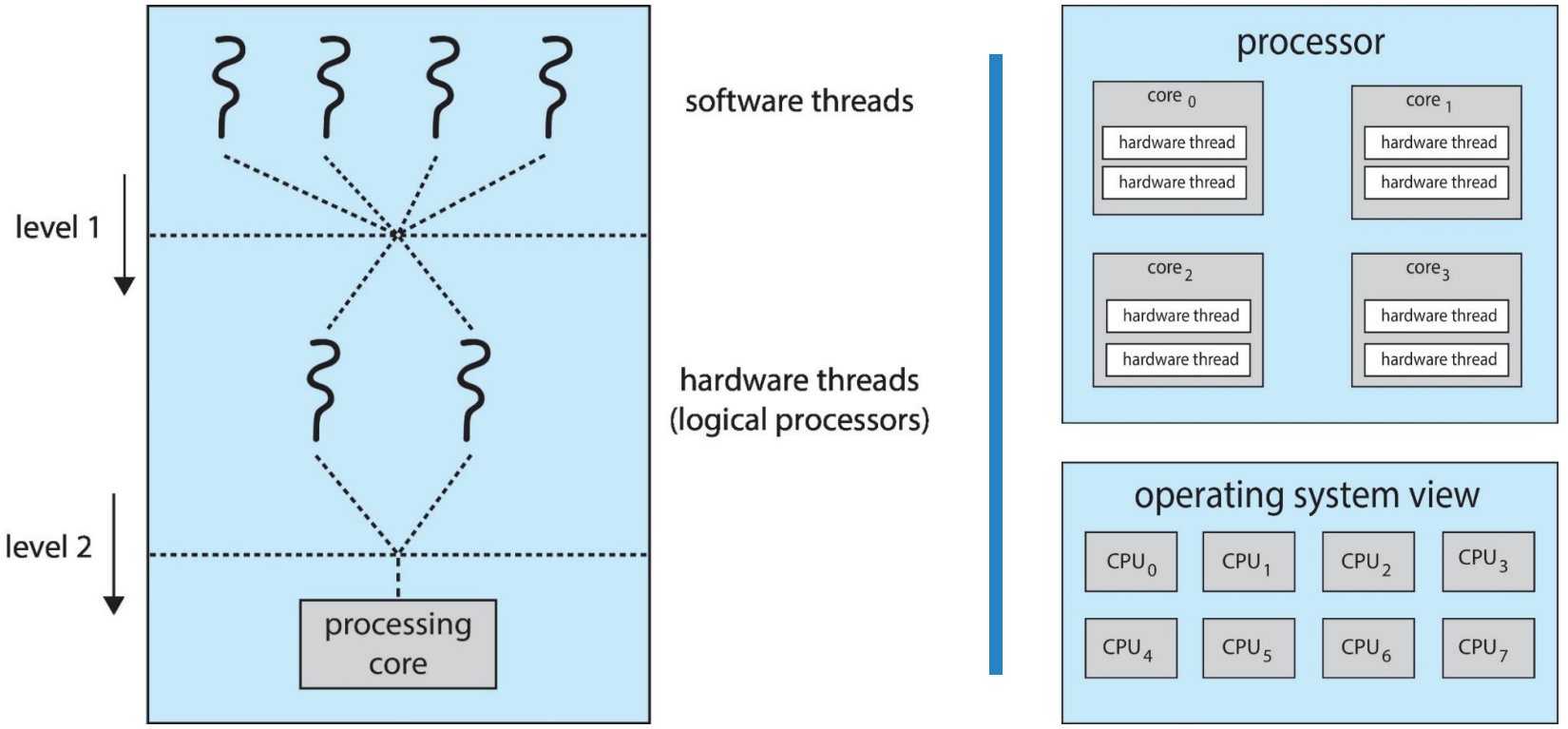




Multithreading processors

Core Access is faster than memory Access, so it can happen that processors have to wait for a memory or cache operation.

A processor can be multithreaded, that is, run multiple processes per core using memory waiting time (memory stall).



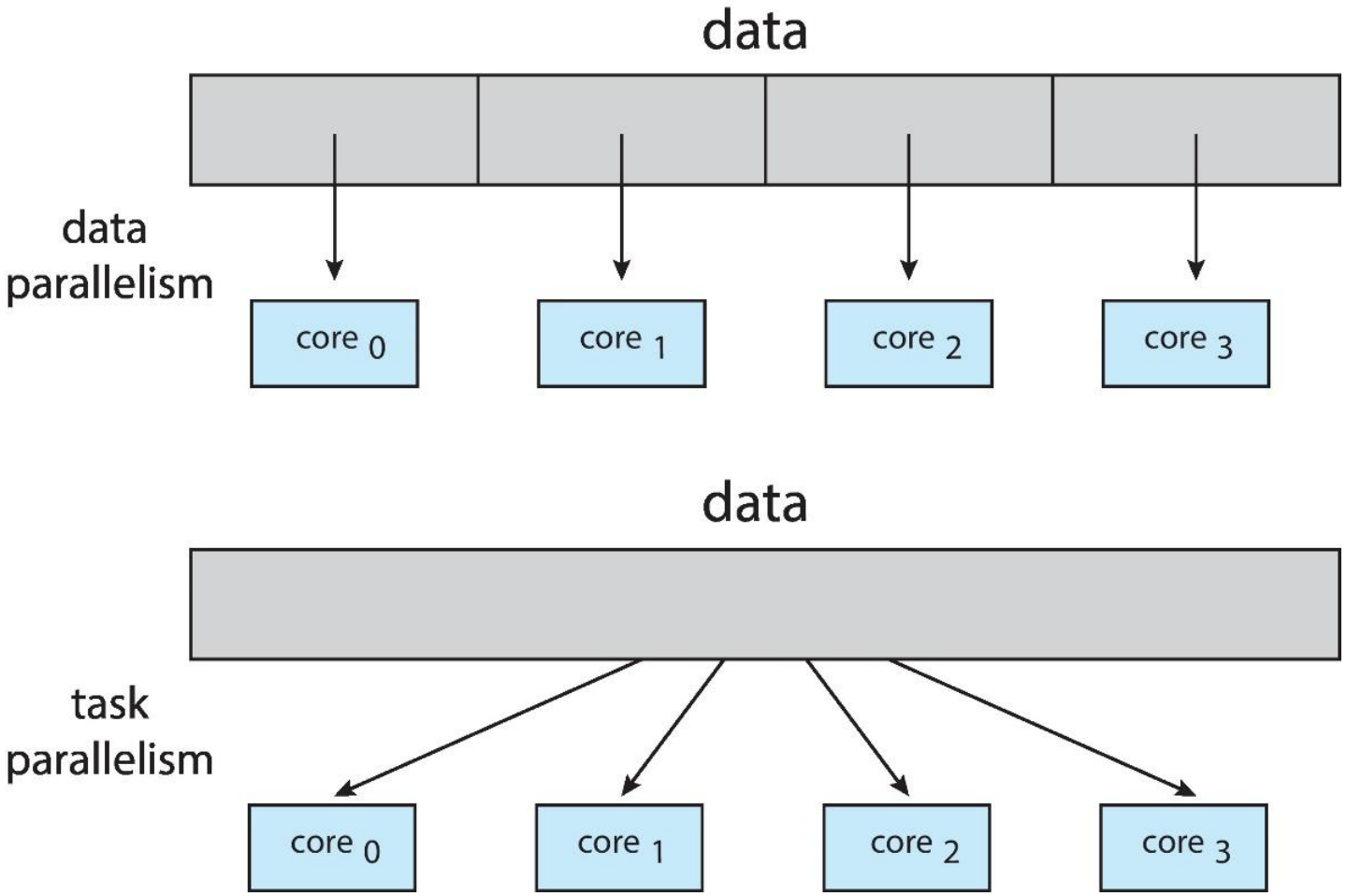
Multi Threading (Concurrency)

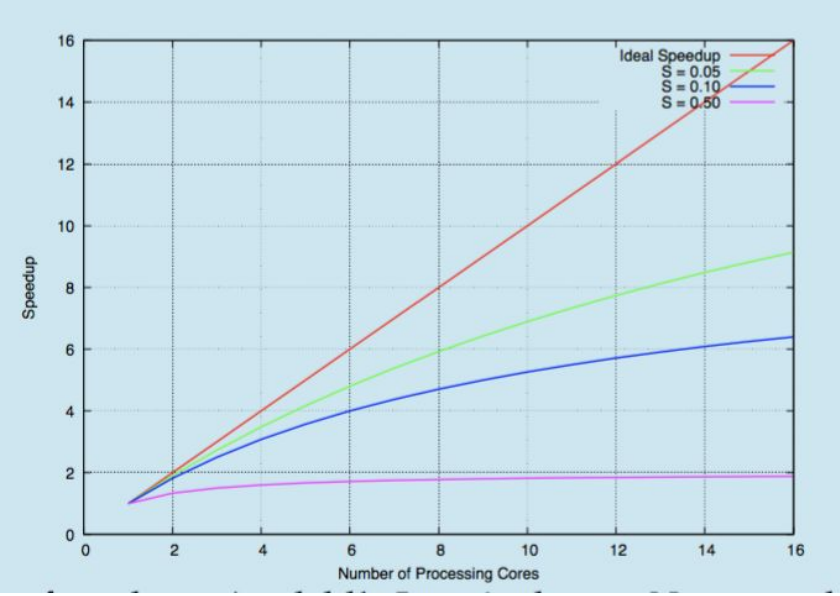
Types of parallelism

Processes and threads can do two types of parallelism.

Data parallelism separates the data but executes the same instruction.

Code parallelism runs different instructions at the same time.





Ahmdal's law

Why having multiple processors does not result in a lineal speed up.

$$speedup \leq \frac{1}{S+\frac{1-S}{n}}$$

S is the serial percentage of the code, and n the number of processors.

As n approaches infinity, the speedup approaches 1/S

A speedup of 1x means the same time is needed to execute the program

To obtain the time, divide the original time between the speedup.

E.g.: A 100% serial program takes 5 hours. How much time it would take if it became a 30% serial code and you had 8 processors

Threads

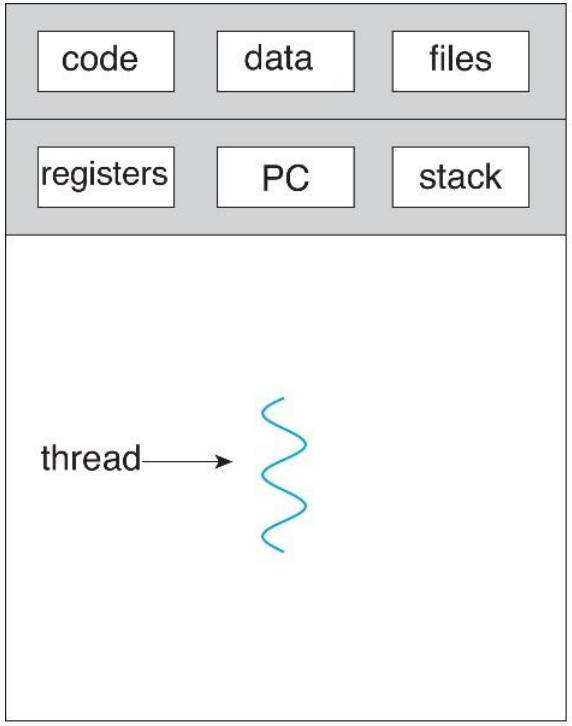
Run multiple things in the same application

Threads run within application

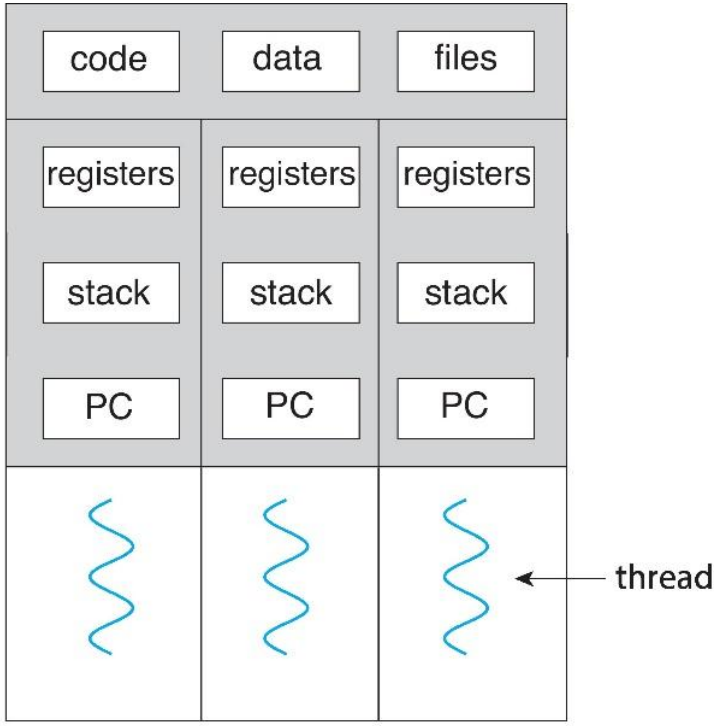
Multiple tasks like

- Update display
- Fetch data
- Spell checking
- Answer a network request

Process creation is heavy-weight while thread creation is light-weight



single-threaded process



multithreaded process

Advantages of threads

Responsiveness

Waiting part of the application

Resource Sharing

Easier than shared memory

Economy

Lightweighter than a fork

Scalability

Allow to use multicore systems

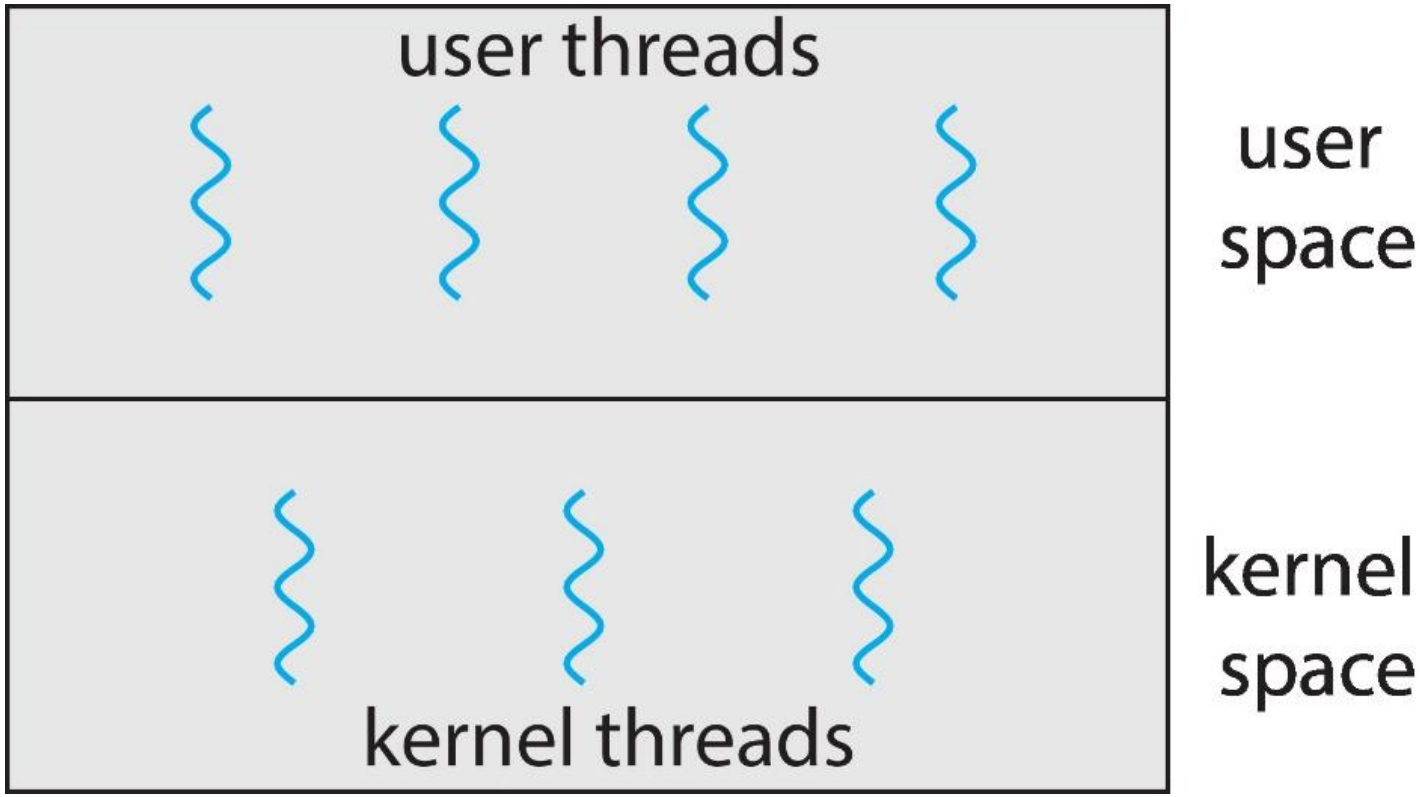
Types of threads

User threads - management done by user-level threads library

- POSIX Pthreads
- Windows threads
- Java threads

Kernel threads - Supported by the Kernel

- Windows
- Linux
- Mac OS X
- iOS
- Android

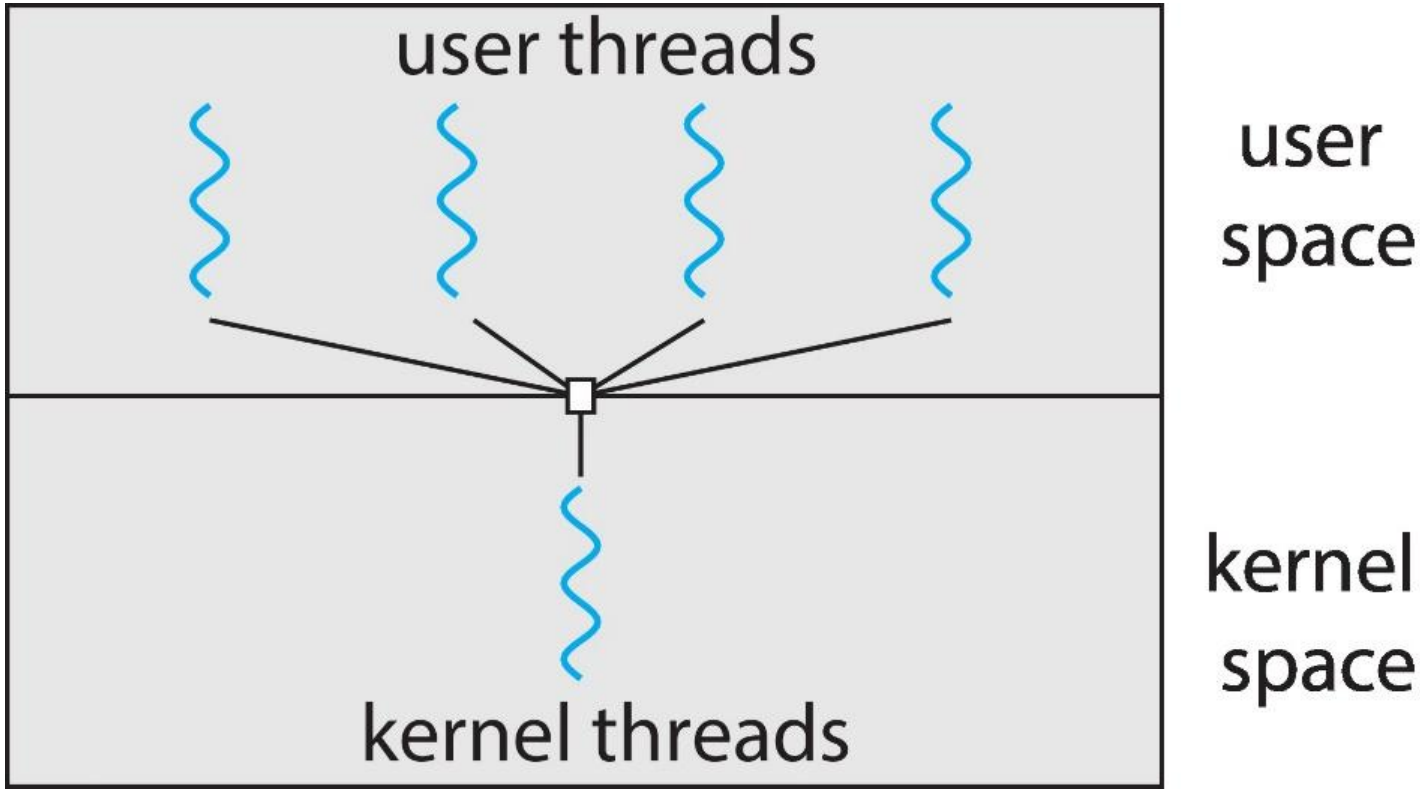


Many-to-one

One kernel thread for all user threads

Creates a bottleneck if the system call has to wait

Not really used



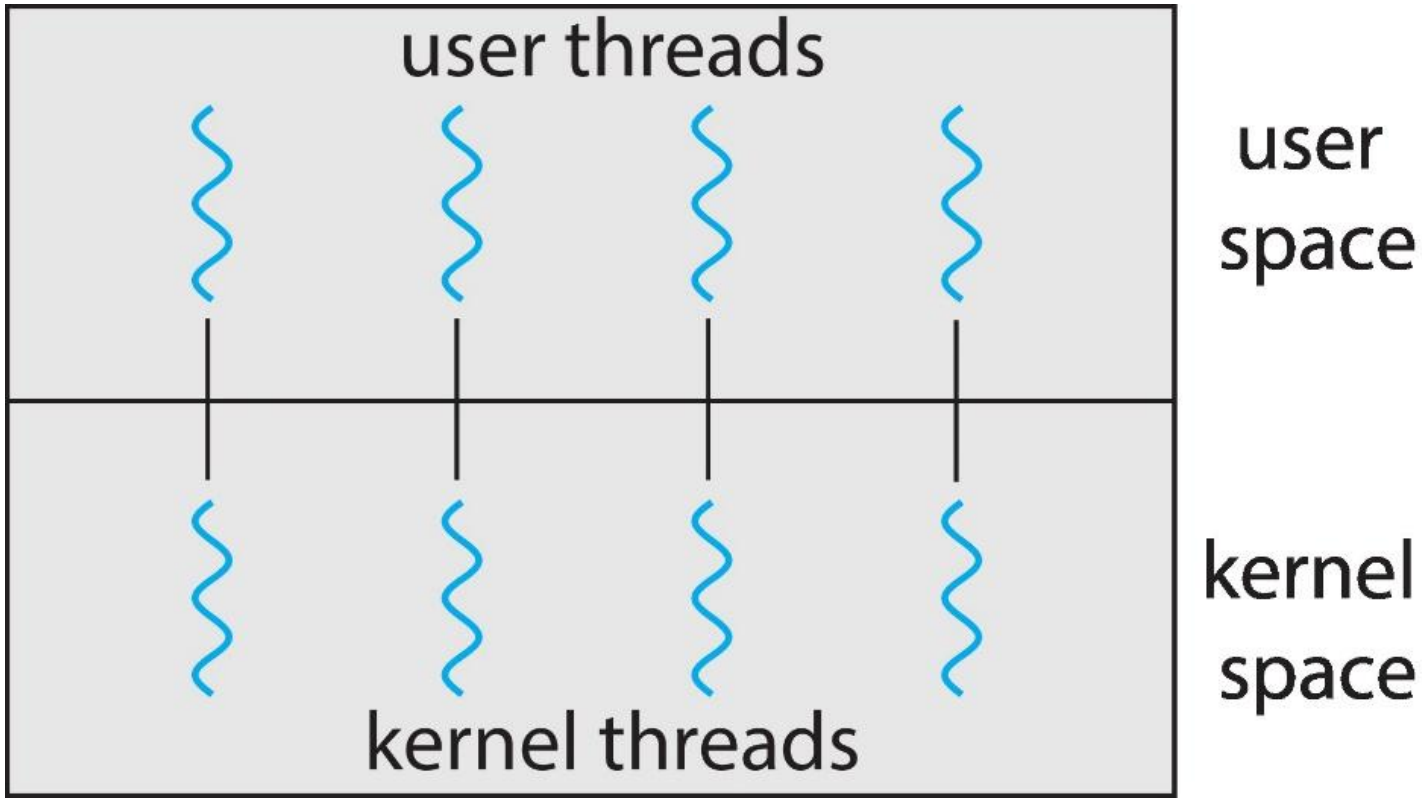
One-to-one

For each user thread, a kernel thread is created.

Allows for greater concurrency of al thread models.

Thread number may be limited by the OS.

This is how Linux and Windows work

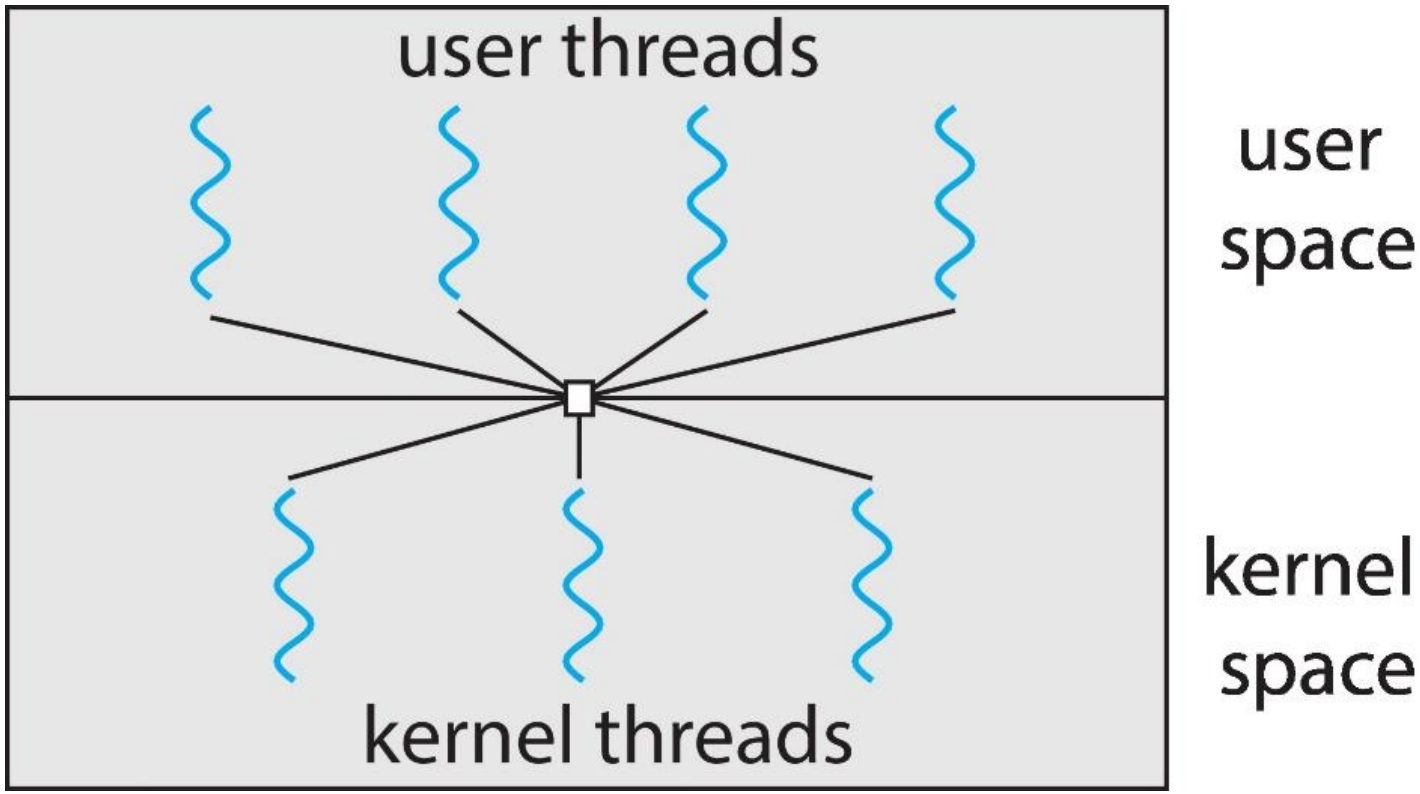


Many-to-many

Has a pool of available kernel threads that get shared.

If a process needs a new system thread and there is not one available, enters to a waiting queue.

Not really used.



Solve the following

1. If a program has a 20% of instructions that can be parallelized, how much will the speed up be with 2 processors? With 4?

2. How much time a program with 10% parallelizable code would take in a machine with 8 cores, if it took 10 hours without parallelism?