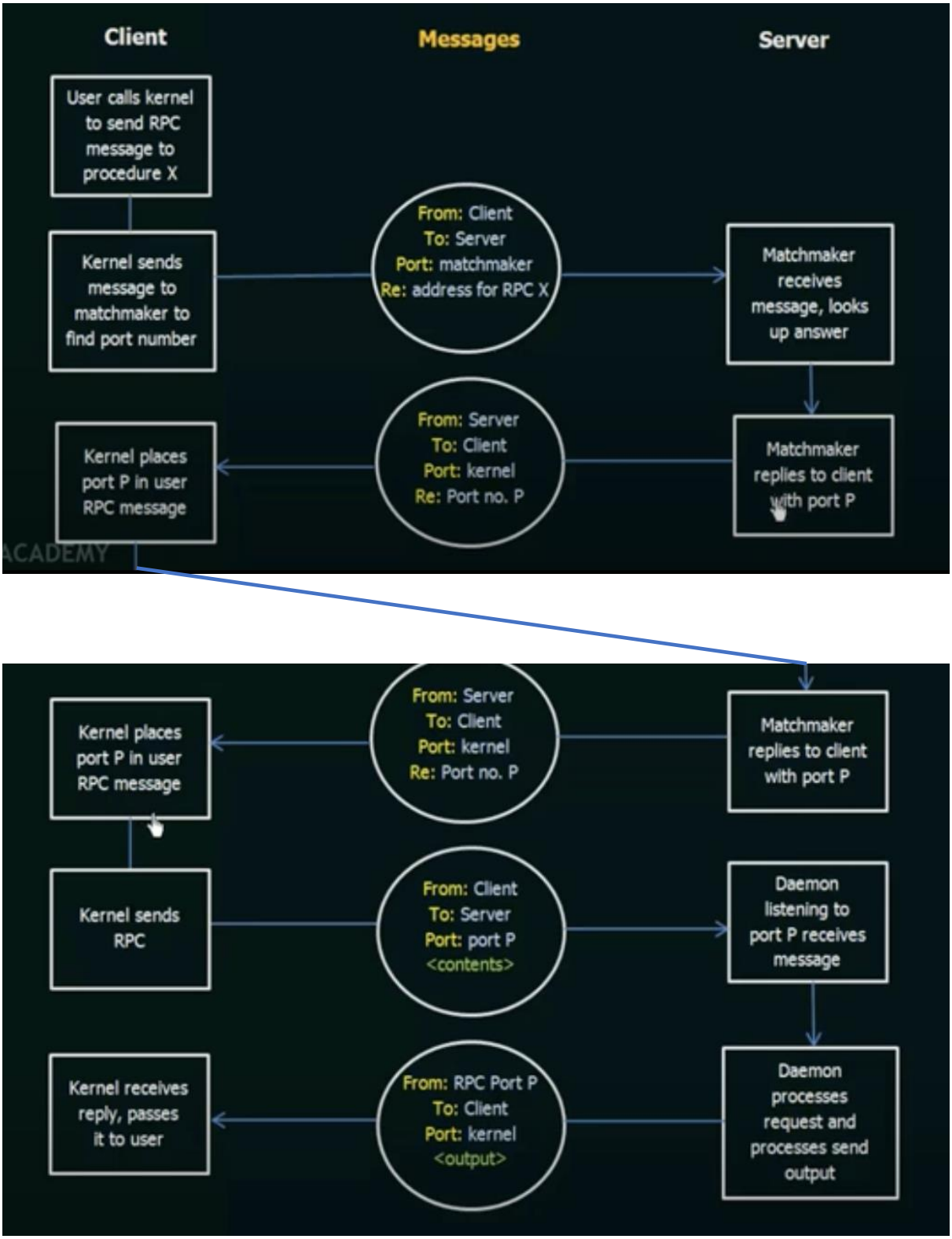
With standard procedure calls, some form of binding takes place during link, load or execution time so that procedure call's name is replaced by the memory address of the procedure call. The RPC scheme requires a similar binding of the client and server port, but how does a client know the port numbers on the server? Neither system has full information about the other because they do not share memory.	1) The binding information may be predetermined, in the form of fixed port addresses. At compile time, an RPC call has a fixed port number associated with it. Once a program is compiled, the server cannot change the port number of the requested service. 2) Binding can be done dynamically by a rendezvous mechanism. Typically, an operating system provides a rendezvous (also called a matchmaker) daemon on a fixed RPC port. A client then sends a message containing the name of the RPC to the rendezvous daemon requesting the port address of the RPC it needs to execute. The port number is returned, and the RPC calls can be sent to that port until the process terminates (or the server crashes)

Execution of a remote procedure call (RPC)



4. Threads

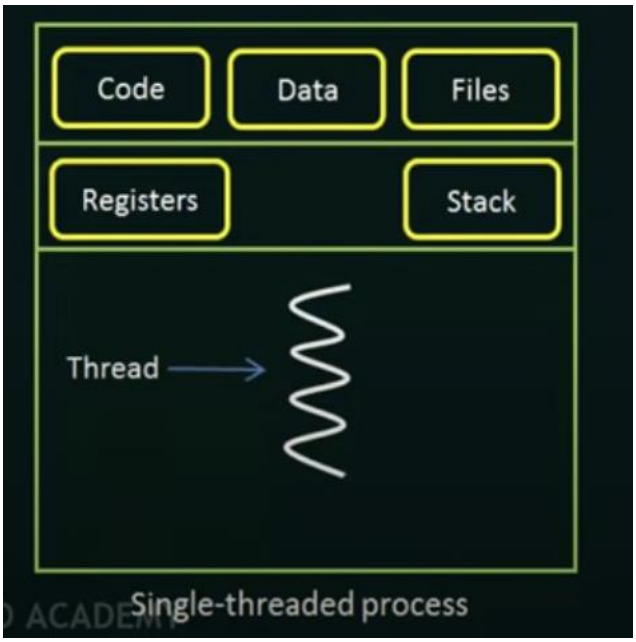
A thread is a basic unit of execution (CPU utilization)

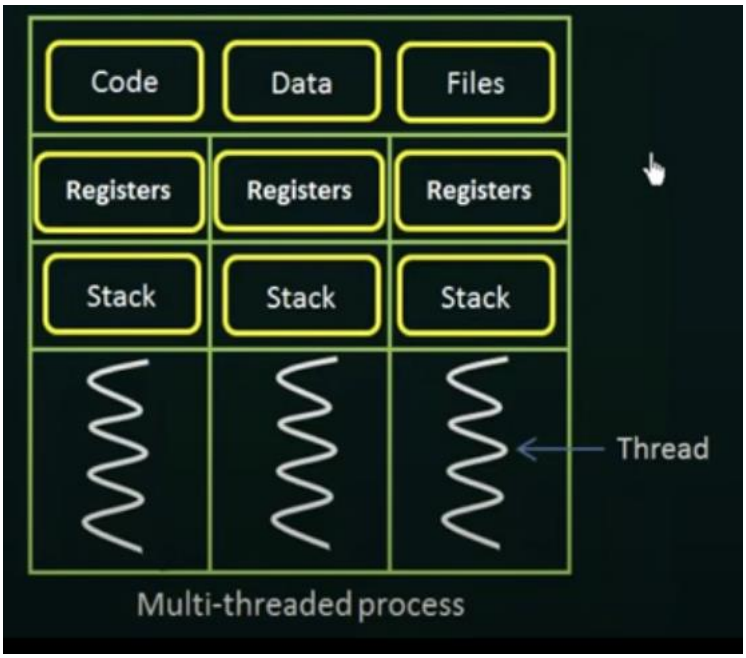
It comprises of,

- **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/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.





The benefits of multithreaded programming can be broken down into four major categories.

- **Responsiveness** – multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

- **Resource sharing –** by default, threads share the memory and the resources of the process to which they belong. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

- **Economy** – allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create a context-switch threads.

- **Utilization of multiprocessor architecture** – treads may be running in parallel on different processors. A single-treaded process can only run on one CPU, no matter how many are available. Multithreading on a multi-CPU machine increases concurrency.

Multithreading Models and Hyperthreading

Types of threads,

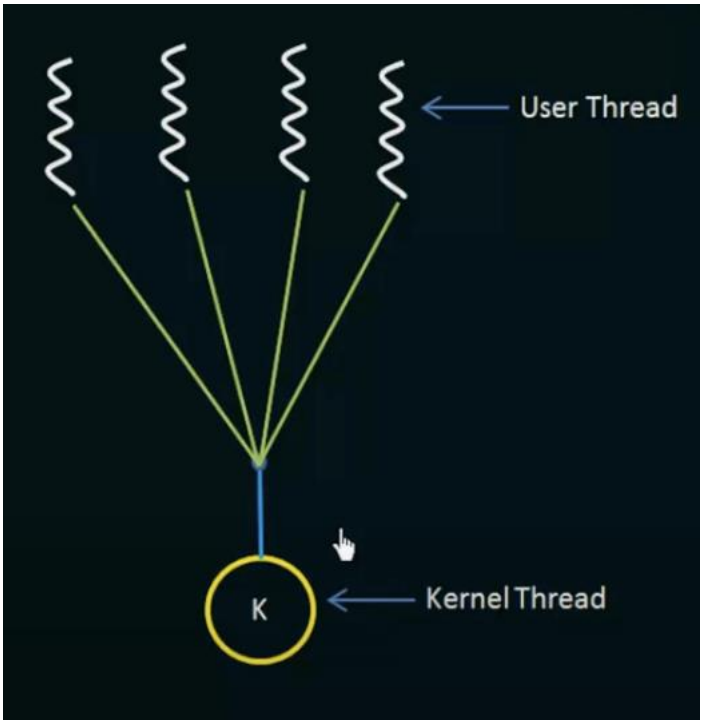
1. User threads – supported above the Kernel and are managed without Kernel support.
2. Kernel threads – supported and managed directly by the operating system.

Ultimately, there must exist a relationship between user threads and Kernel threads.

There are three common ways of establishing this relationship.

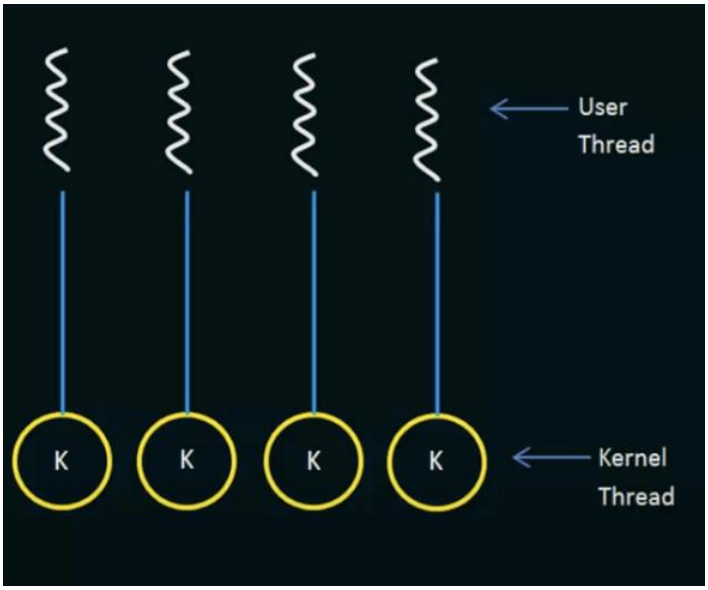
1. Many-to-One model
2. One-to-One model
3. Many-to-many model

Many-to-One model



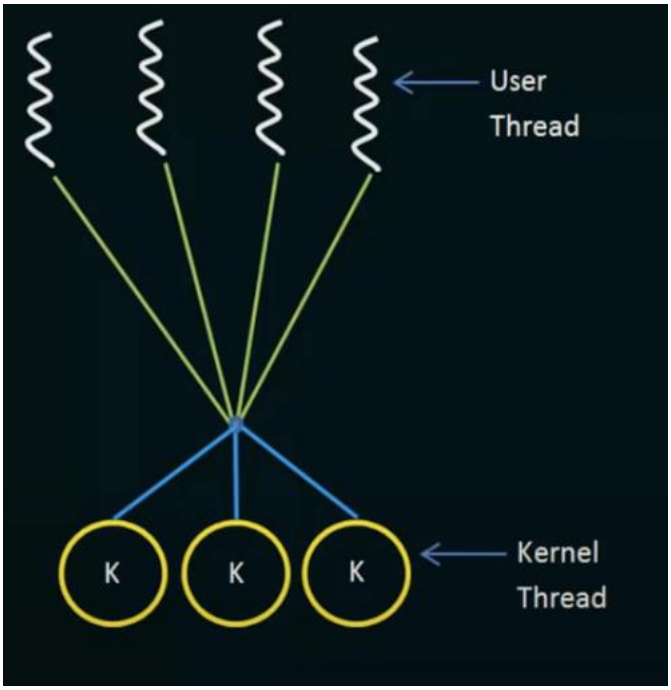
- Maps many user-level threads to one Kernel thread
- Thread management is done by the thread library in user space, so it is efficient.
- The entire process will block if a thread makes a blocking system call
- Because only one thread can access the Kernel at a time, multiple threads are unable to run in parallel on multiprocessor.

One-to-One model



- Maps each user thread to a Kernel thread.
- Provides more concurrency than the many-to-one model by allowing another thread to run when a tread makes a blocking system call.
- Also allows multiple threads to run in parallel on multiprocessors.
- Creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating Kernel level threads can burden the performance of an application, most implementations of this model restrict the # of threads supported by the system.

Many-to-Many model



- Multiplexes many user-level threads to a smaller or equal # of Kernel threads
- The number of kernel threads may be specific to either a particular application or machine.
- Developers 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.

Simultaneous Multithreading (SMT) (Hyperthreading)

Hyperthreaded systems allow their processor cores' resources to become multiple logical processors for performance.

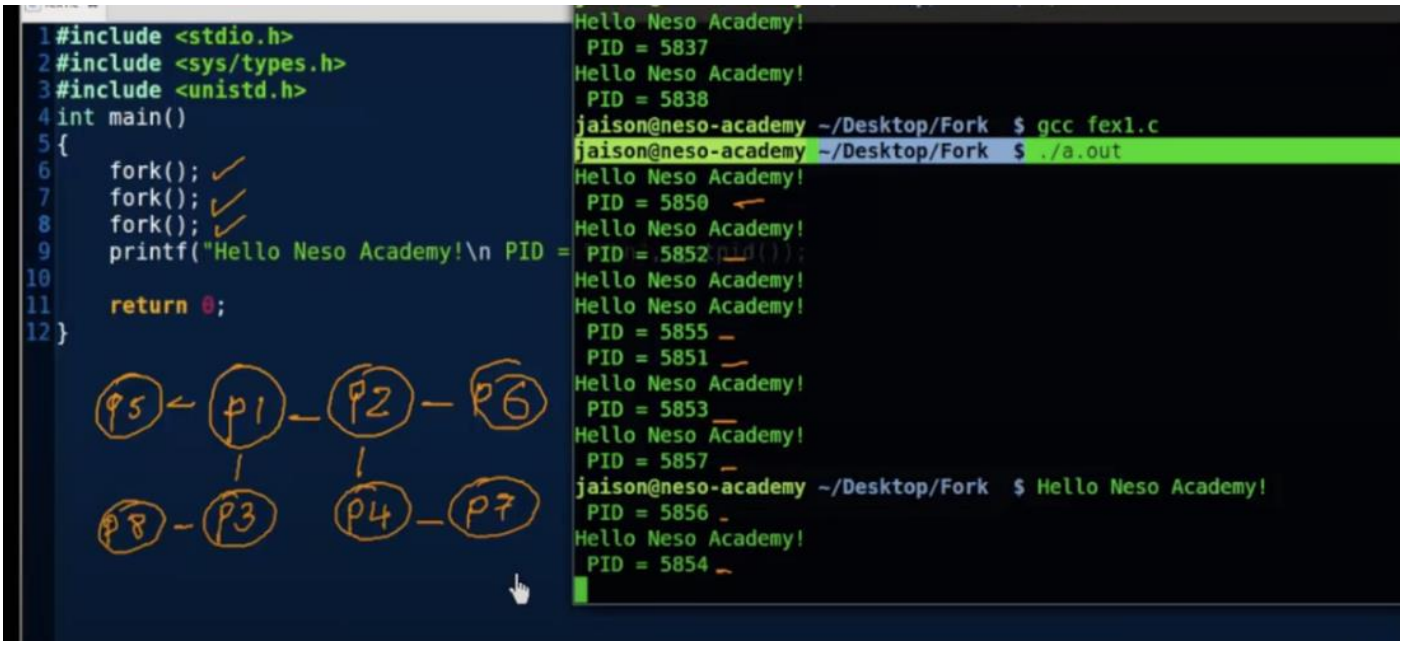
It enables the processor to execute two treads, or sets of instructions, at the same time. Since hyper-threading allows two streams to be executed in parallel, it is almost like having two separate processors working together.

The fork() and exec() system calls

fork() : the fork() system call creates a separate, duplicate process.

exec() : when an exec() system is invoked, the program specified in the parameter to exec() will replace the entire process. – including all threads.
We have to pass a parameter to exec() system call which will be another program, then that program will replace the entire process from which this exec() was called.

fork() system call



Total number of processes = 2^n (n Is the # of fork() system calls)

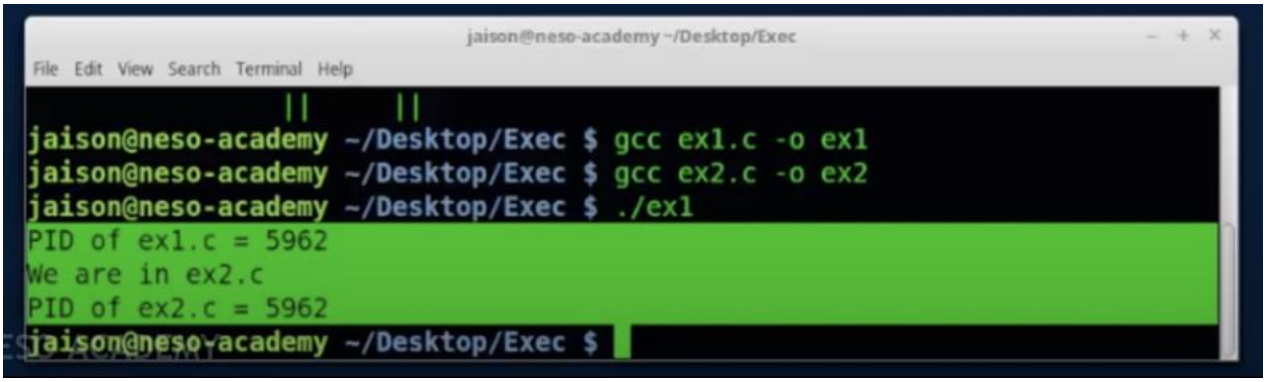
exec() system call

ex1

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("PID of ex1.c = %d\n", getpid());
    char *args[] = {"Hello", "Neso", "Academy", NULL};
    execv("./ex2", args);
    printf("Back to ex1.c");
    return 0;
}
```

Ex2

```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
    printf("We are in ex2.c\n");
    printf("PID of ex2.c = %d\n", getpid());
    return 0;
}
```



Same PIDs…that means exec() system call hasn't created a new process, but replaced the whole ex1 process with the contents of ex2 retaining the same PID

Treading Issues

The fork() and exec() system calls

The semantics of the fork() and exec() system calls change in a multithreaded program.

Issue

If one tread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded ?

Solution

Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the threads that invoked the fork() system call.

But which version of fork() to use and when ?

Also, is a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process – including all the threads.

Which of the two versions of fork() to use depends on the application.

If exec() is called immediately after forking

Then duplicating all threads in unnecessary, as the program specified in the parameters to exec() will replace the process.

In this instance, duplicating only the calling thread is appropriate.

If the separate process does not call exec() after forking

Then the separate process should duplicate all threads.

Tread Cancellation

Thread cancellation is the task of terminating a thread before it had completed.

Examples of thread cancellations,

- If multiple threads are concurrently searching through a database and one thread returns the result, the remaining threads might be canceled.
- When a user presses a button that stops a web page from loading any further, all threads loading the page are canceled.

A thread that is to be canceled is often referred to as the target thread.

Cancellation of a thread may occur in two different scenarios.

1. Asynchronous cancellation – One thread immediately terminates the target thread.
2. Deferred cancellation – The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion

Where the difficulty with cancellation lies:

In situations where,

- Resources have been allocated to a canceled thread
- A thread is canceled while in the midst of updating data it is sharing with other threads.

Often, the OS will reclaim system resources from a canceled thread but will not reclaim all resources.

Therefore, cancelling a thread asynchronously may not free a necessary system-wide resource.

With deferred cancellation:

One thread indicates that a target thread is to be canceled.

But cancellation occurs only after the target thread has checked a flag to determine if it should be canceled or not.

This allows a thread to check whether it should be canceled at a point when it can be canceled safely.