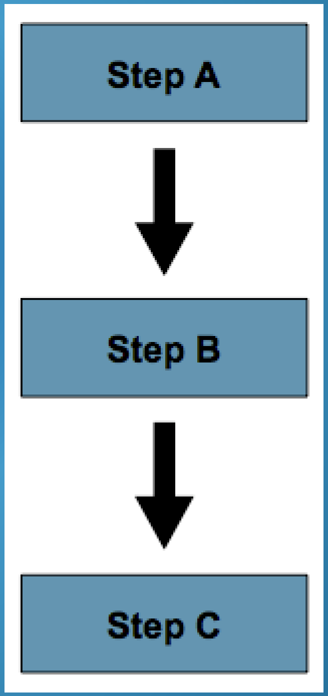
CS240 Operating Systems, Communications and Concurrency

Sequential Process Execution Pattern

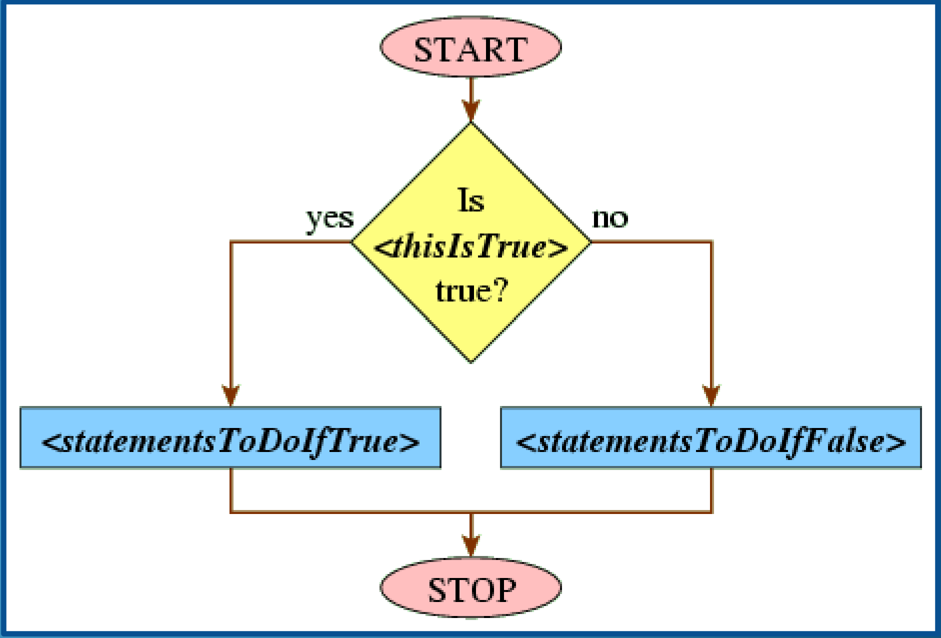
A sequential process has a single thread of execution. That is, the process executes instructions retrieved from only one specific point in the program at a time and executes them in a sequential manner.



CS240 Operating Systems, Communications and Concurrency

Sequential Process Execution Pattern

Flow control instructions like loops, selection statements and procedure calls or external events cause the process to jump to other parts of the code when required, but a single threaded process can only do one thing at a time.



CS240 Operating Systems, Communications and Concurrency

Process Creation Overhead

Recall the Process Control Block Structure

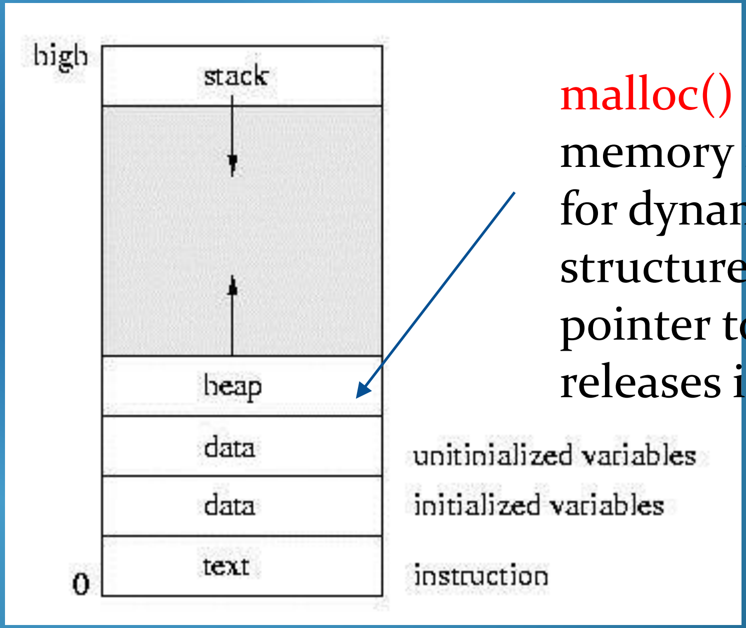
Process Identification Data	Processor State Data	Process Control Data
Unique process identifier	CPU Register State,	Flags, signals and messages.
Owner's user identifier	Pointers to stack and memory space	Pointers to other processes in the same queue
Group identifier	Priority	Parent and Child linkage pointers
	Scheduling Parameters	Access permissions to I/O Objects
	Events awaited	Accounting Information

Overhead Cost: Creating a process using fork() incurs processor time to allocate and initialise many structures and incurs ongoing cost in terms of memory resource usage.

CS240 Operating Systems, Communications and Concurrency

Sequential Process Memory Space Organisation

The run time stack is an area of memory which is intrinsically linked with processor instructions for managing the call/return mechanism and other execution related purposes.



malloc() allocates memory from the heap for dynamic data structures and returns a pointer to it; free() releases it

The stack contains contextual information relevant to the current execution stream.

CS240 Operating Systems, Communications and Concurrency

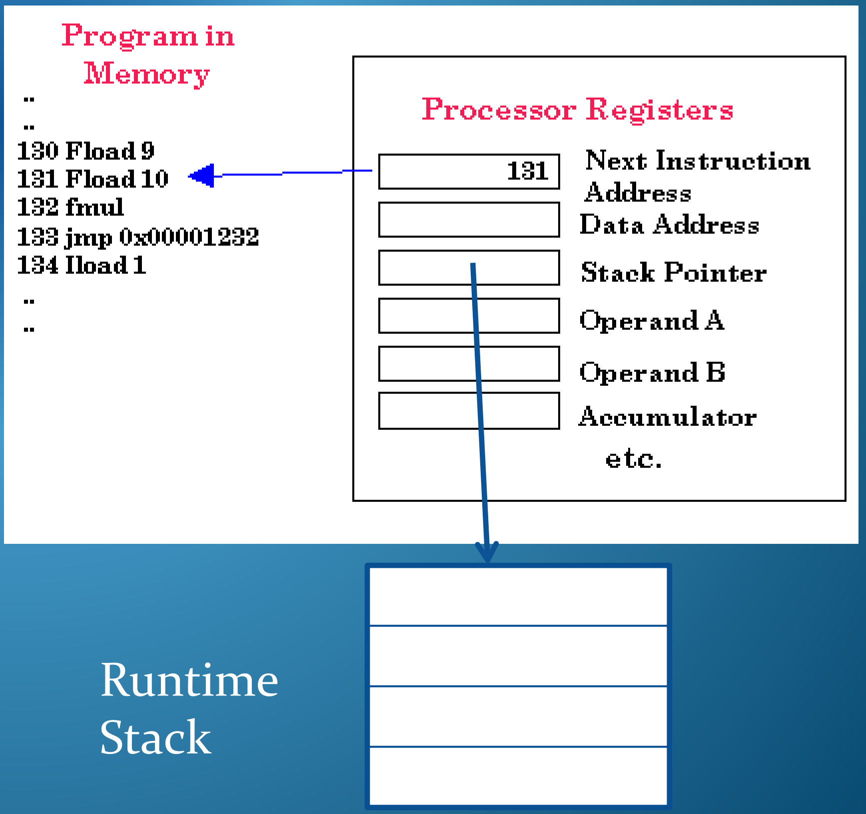
Subset of Data Associated with a Process Execution Context

The current execution context of a process is represented by a small specific subset of information.

The current state of the CPU program counter.

Various CPU registers may be used for caching other data.

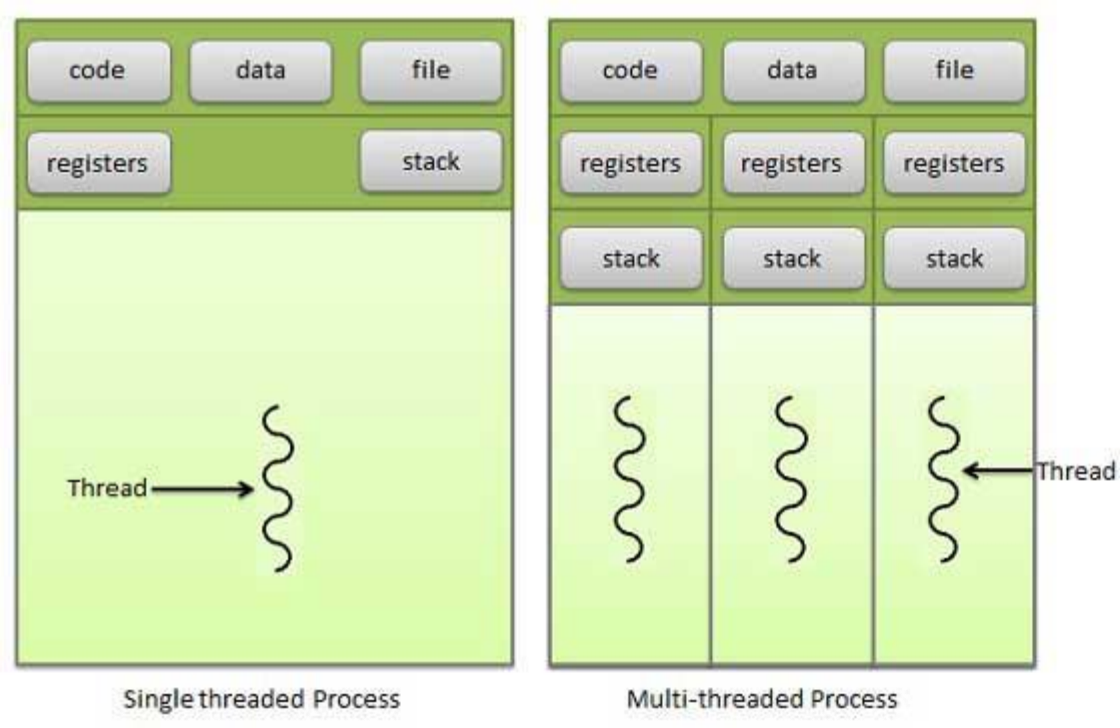
A runtime stack stores parameters and linkage for method calls and other temporary data.



CS240 Operating Systems, Communications and Concurrency

Multi-threaded Processes

Modern operating systems support the model of a single process environment supporting multiple threads of control executing independently at different points within the process's code.

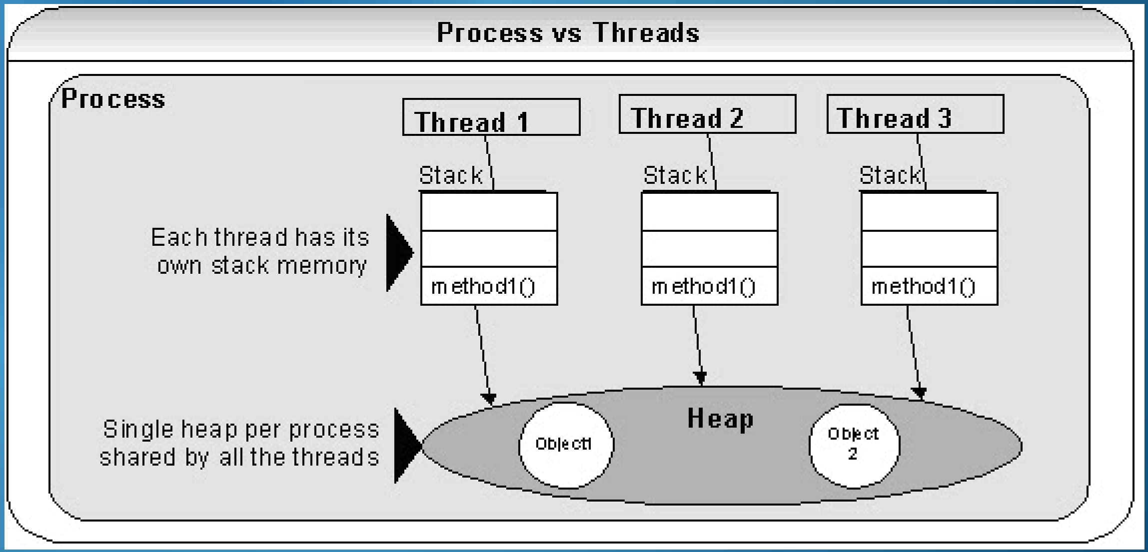


CS240 Operating Systems, Communications and Concurrency

Multi-threaded Processes

Advantages of Threads

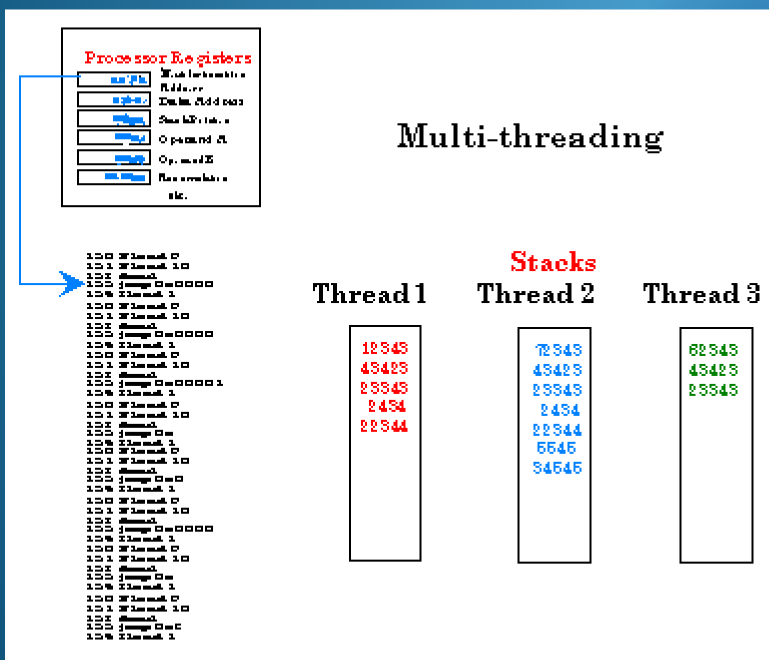
These lightweight thread structures share the environment of the containing process and so are quick to create and can be managed as schedulable entities by the Kernel.



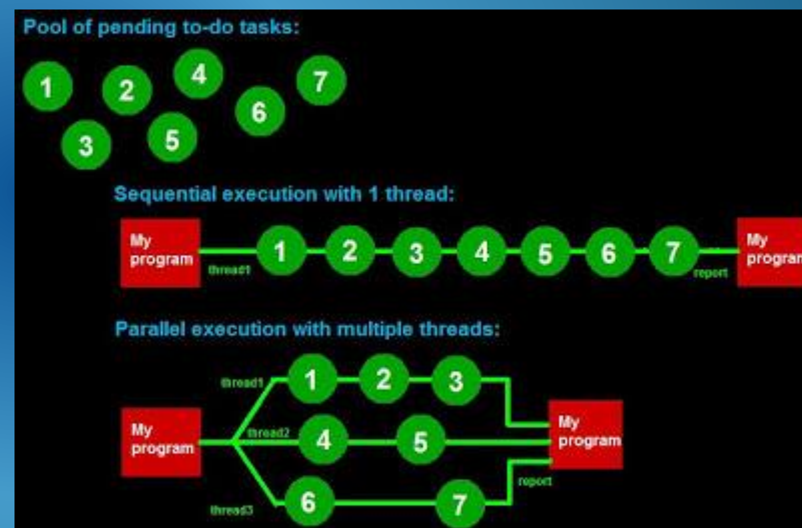
Better Resource Usage / Quicker to Create

CS240 Operating Systems, Communications and Concurrency

Multi-threaded Processes



Advantages of Threads



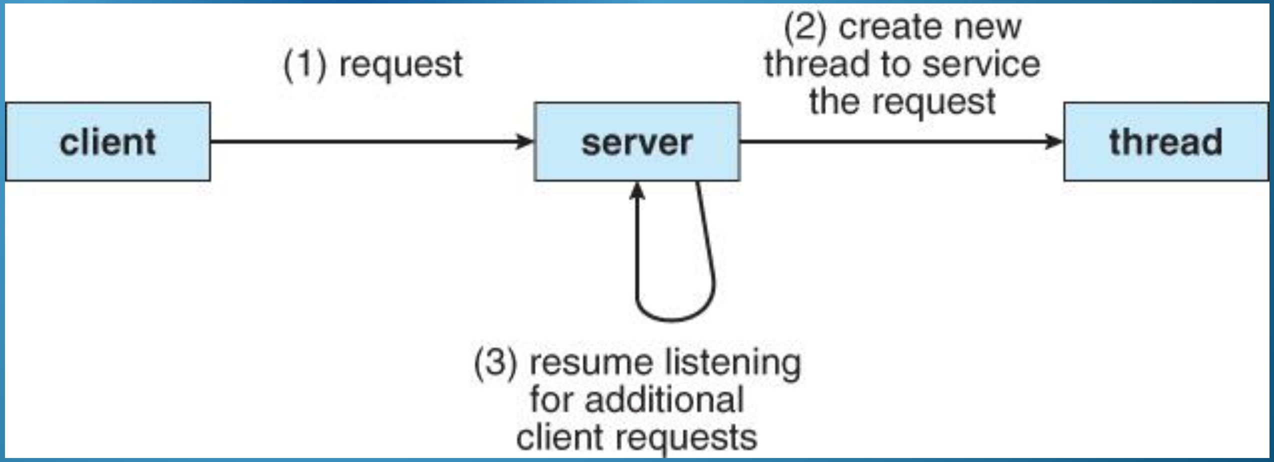
Improved program performance on multicore architectures and improved responsiveness for GUI based applications.

CS240 Operating Systems, Communications and Concurrency

Multi-threaded Processes

Advantages of Threads

Less problems with network protocols due to server responsiveness



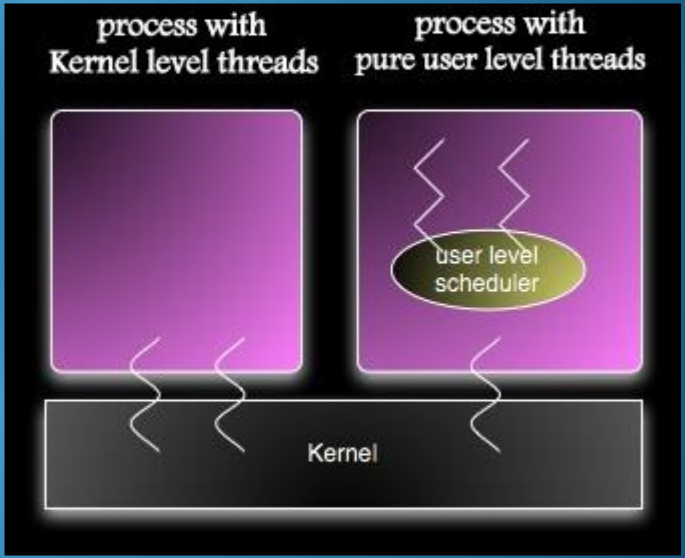
Easy concept of parallelism for programmer to understand, assign threads to modular tasks

CS240 Operating Systems, Communications and Concurrency

Implementing Thread Support

Parallelism on any operating system

A thread mechanism can be implemented within the kernel or within an ordinary user space process (e.g Java VM) or both - where user threads are mapped onto kernel threads.

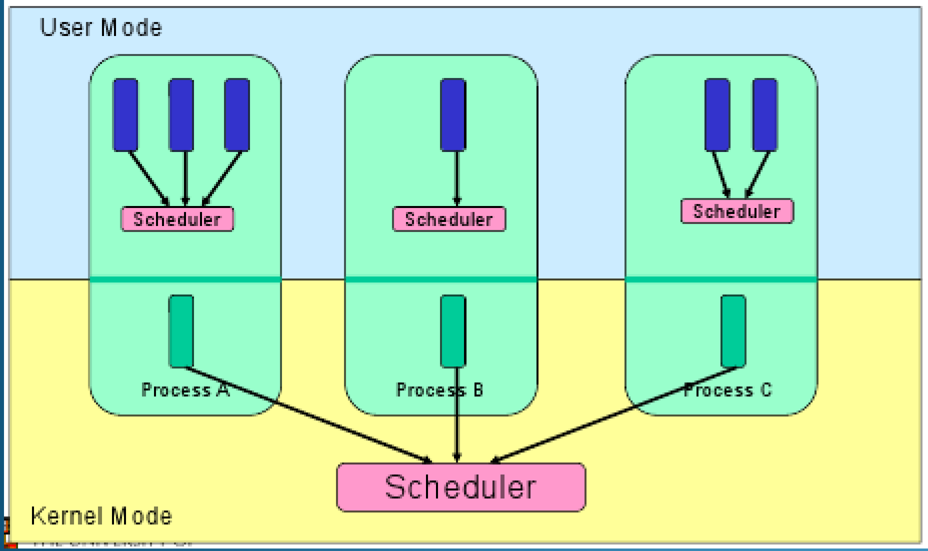


Parallelism in any language

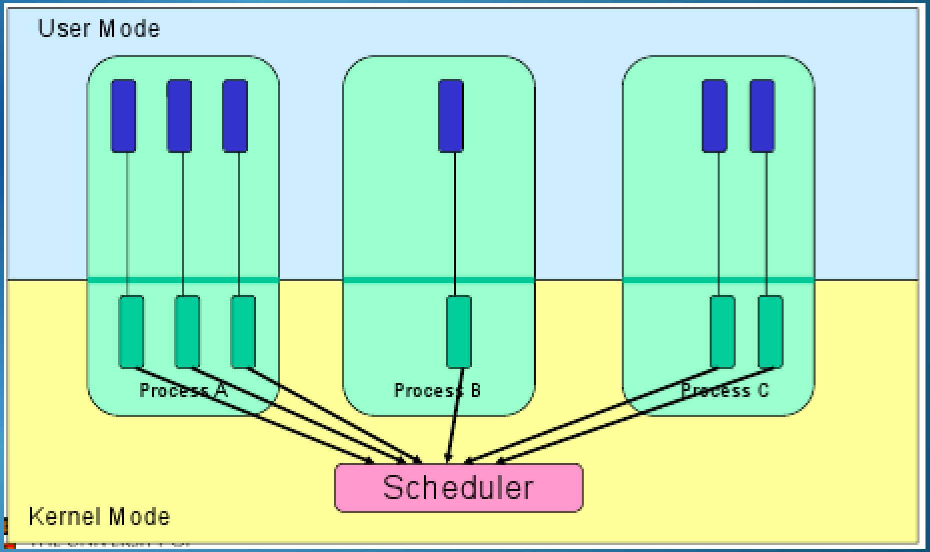
Multithreaded code can be written in any programming language for which there is a thread library and runtime that can be linked with the program.

CS240 Operating Systems, Communications and Concurrency

Multi-threaded Processes



User space package like Java VM manages thread abstraction, creation, communication and scheduling.



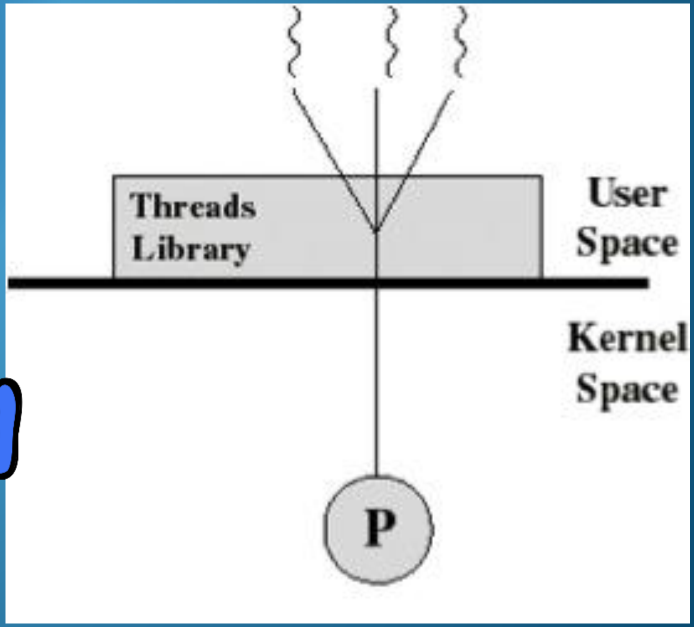
Kernel manages thread abstraction, creation, communication and scheduling.

CS240 Operating Systems, Communications and Concurrency

Multi-threaded Processes

User Space Threads

This implementation does not require any operating system support for threads. Invoking a function in the library to create or destroy a thread or schedule a thread can be handled very fast as a local function call in user space, which manipulates local data structures for controlling user defined threads and what the program is doing at a given moment.

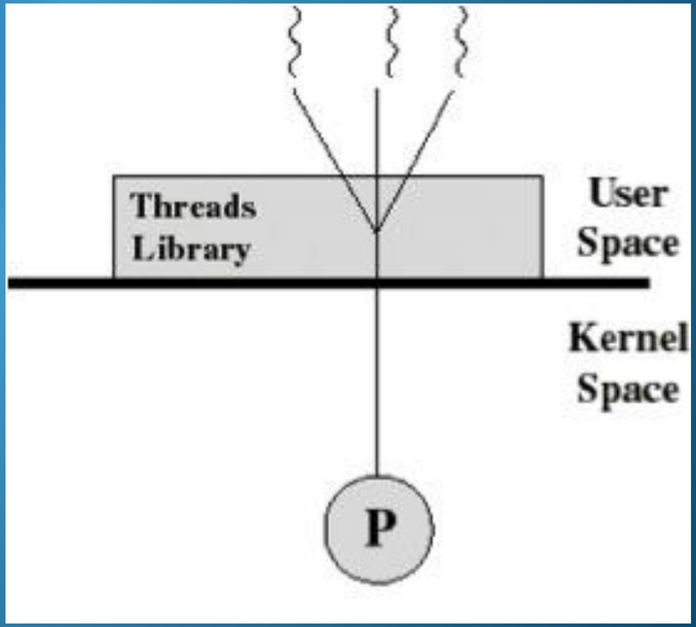


CS240 Operating Systems, Communications and Concurrency

Multi-threaded Processes

User Space Threads

A context switch to the kernel would block all threads in the process, as the kernel is unaware of the existence of separate thread abstractions manipulated within the application context.

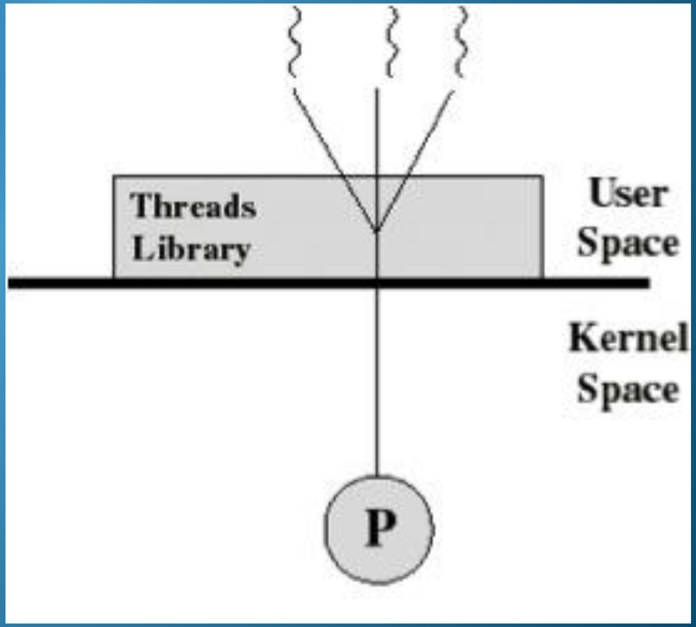


Multi-threaded Processes

User Space Threads

Also, scheduling of threads by the user space thread library would be non-preemptive.

And the application can't benefit from multicore processing.

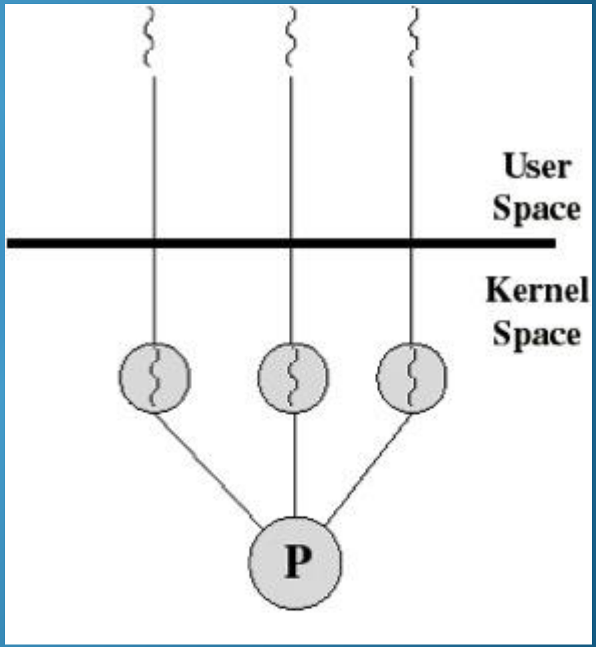


Multi-threaded Processes

Kernel Space Threads

Implementing threads in kernel space offers greater flexibility and efficiency.

A thread issuing a system call does not necessarily block all other threads in that process.



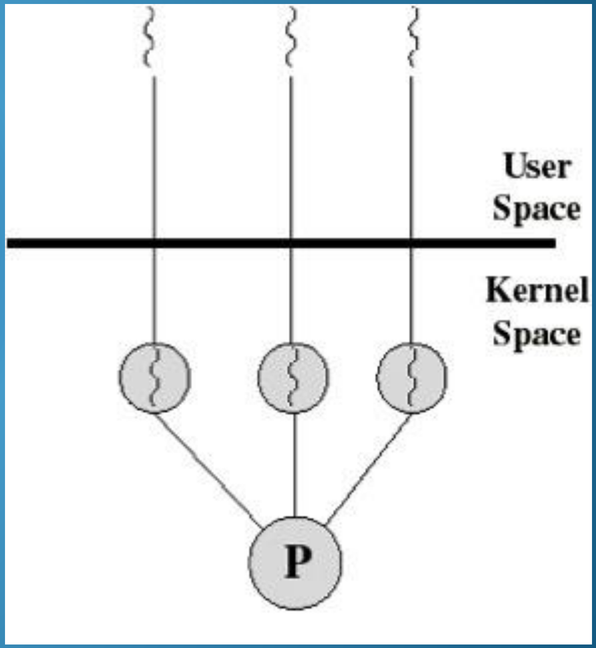
CS240 Operating Systems, Communications and Concurrency

Multi-threaded Processes

Kernel Space Threads

Threads can be scheduled onto separate CPUs on a multiprocessor system offering true application concurrency.

Threads compete on an equal basis for CPU cycles and they may be preempted by hardware timers easily.



Multi-threaded Processes

Kernel Multithreaded Design

To achieve greater responsiveness, the kernel itself could be implemented as a multithreaded process. E.g. The scheduler, the IPC and I/O handling could all be separate threads.

In this way the kernel does not block all processes requiring kernel services while a system call is in progress.

This requires careful design of the kernel data structures and algorithms to allow for synchronisation of multithreaded activity. It is a necessary kernel design especially if the underlying hardware has a multiprocessor architecture.

Thread Management in Unix – POSIX Interface

POSIX Standard was developed to provide a common abstraction or virtualization of operating system services by making system calls standard across all Unix system varieties, enhancing code portability.

The POSIX pthread API is the component of the interface applicable to thread management.

There are also pthread library implementations available for Windows systems.

CS240 Operating Systems, Communications and Concurrency

Thread Management in Unix – POSIX Interface

pthread Thread Creation

A new thread is created by the **`pthread_create()`** function. The function takes four parameters:-

An **identifier for the thread management structure**, of type **pthread_t**

attributes the new thread will have such as stack size and scheduling attributes.

where the new thread will begin execution in the program

list of arguments to be passed to the function the thread will execute

CS240 Operating Systems, Communications and Concurrency

Thread Management in Unix – POSIX Interface

Example of pthread creation and termination

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

void* print_message_function( void* message ) {
   printf("%s \n", (char *)message);
   return NULL;
}

main(int argc, char **argv) {
 pthread_t   thread1, thread2;
```

Thread Management in Unix – POSIX Interface

```
char message1[9] = "Thread 1";
char message2[9] = "Thread 2";
int  iret1, iret2;

/* Create independent threads, each of which will execute the
function - pthread_create returns 0 if successful */

     iret1 = pthread_create( &thread1, NULL,
                            &print_message_function, message1);
     iret2 = pthread_create( &thread2, NULL,
                            &print_message_function, message2);
```

Default stack and scheduling attributes

Thread Management in Unix – POSIX Interface

Thread Termination & Collecting Results

A thread terminates implicitly when it comes to the end of its function.

The thread can also terminate explicitly if it calls **`pthread_exit()`** anywhere in its code.

Thread structure remains in the system until it has been detached.

The value returned by pthread_exit() in the thread's function is saved and can be collected by the caller of a **`pthread_join()`** function.

Thread Management in Unix – POSIX Interface

Detaching a Thread & Cleaning up allocated structures

A thread structure can be detached either by another thread calling **`pthread_detach()`**

or by another thread calling **`pthread_join()`** which allows a return value to be collected.

The **`pthread_join()`** function blocks the caller until the thread terminates.

CS240 Operating Systems, Communications and Concurrency

Thread Management in Unix – POSIX Interface

Example of pthread creation and termination

```
/* Wait till threads are complete before main continues.
Unless we wait we run the risk of executing an exit which
will terminate the process and all threads before the
threads have completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);
     printf("All threads finished\n");

exit(0);
}
```

No return value to be collected

Thread Management in Unix – POSIX Interface

Example of pthread creation and termination

Compile the program using: `cc -lpthread pthread1.c`
Run: `./a.out`
(note the order of output depends on thread scheduling by the operating system)

Output:

```
Thread 1
Thread 2
All threads finished
```