

Threads, Signals

Abhijit A M
abhijit.comp@coep.ac.in

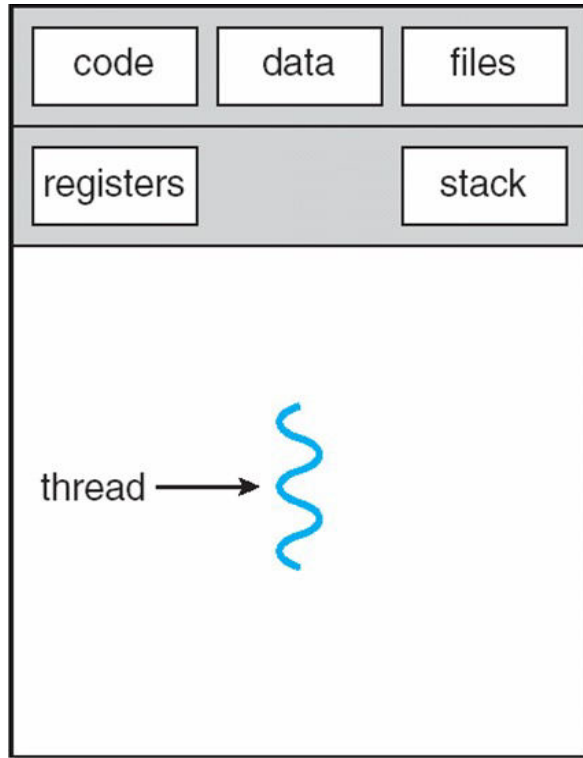
Threads

- **thread — a fundamental unit of CPU utilization**
 - A separate control flow within a program
 - set of instructions that execute “concurrently” with other parts of the code
 - Note the difference: Concurrency: progress at the same time, Parallel: execution at the same time
- **Threads run within application**
 - An application can be divided into multiple parts
 - Each part may be written to execute as a threads
- **Let's see an example**

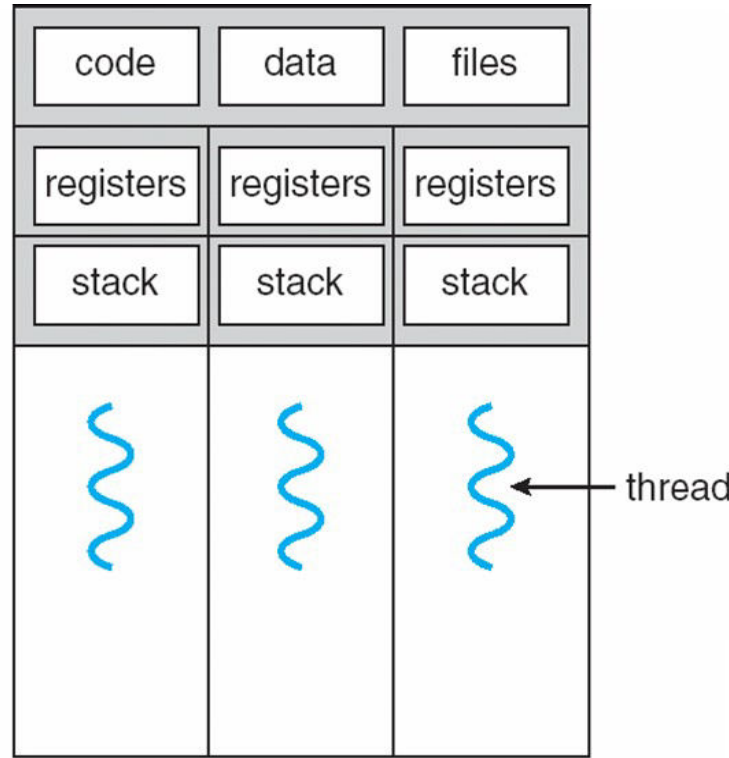
Threads

- **Multiple tasks with the application can be implemented by separate threads**
 - Update display
 - Fetch data
 - Spell checking
 - Answer a network request
- **Process creation is heavy-weight while thread creation is light-weight, due to the very nature of threads**
- **Can simplify code, increase efficiency**
- **Kernels are generally multithreaded**

Single vs Multithreaded process

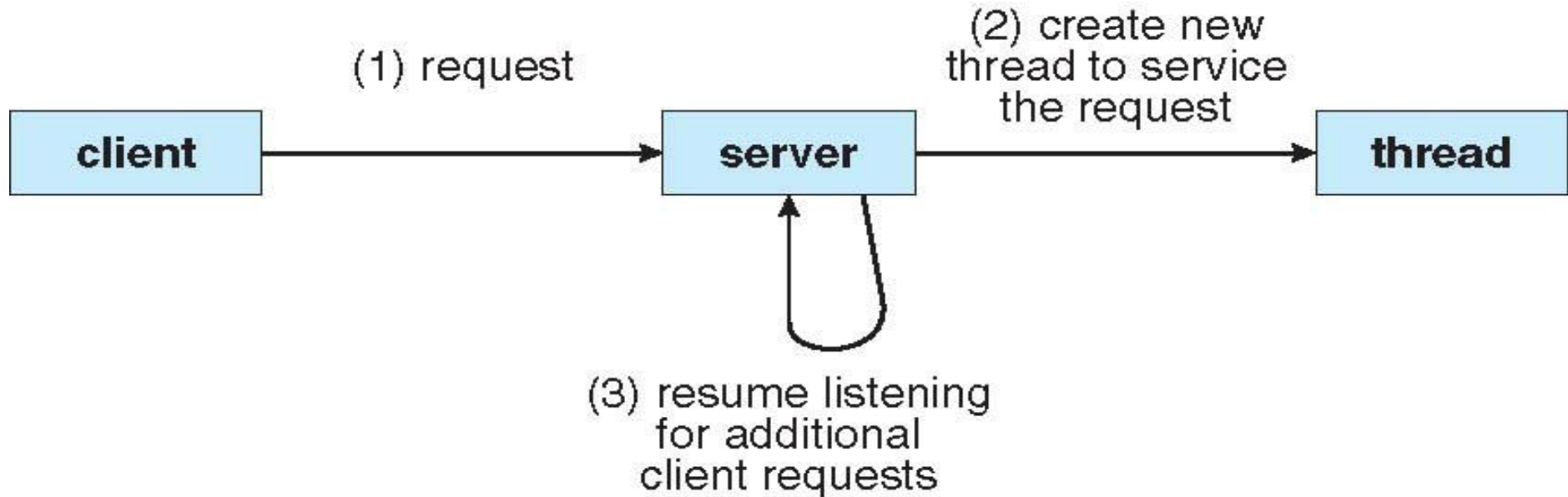


single-threaded process



multithreaded process

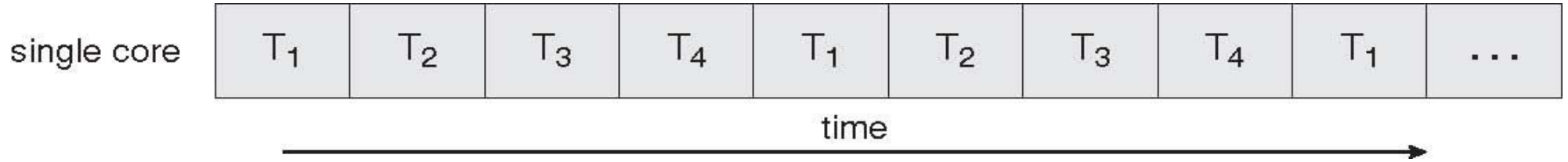
A multithreaded server



Benefits of threads

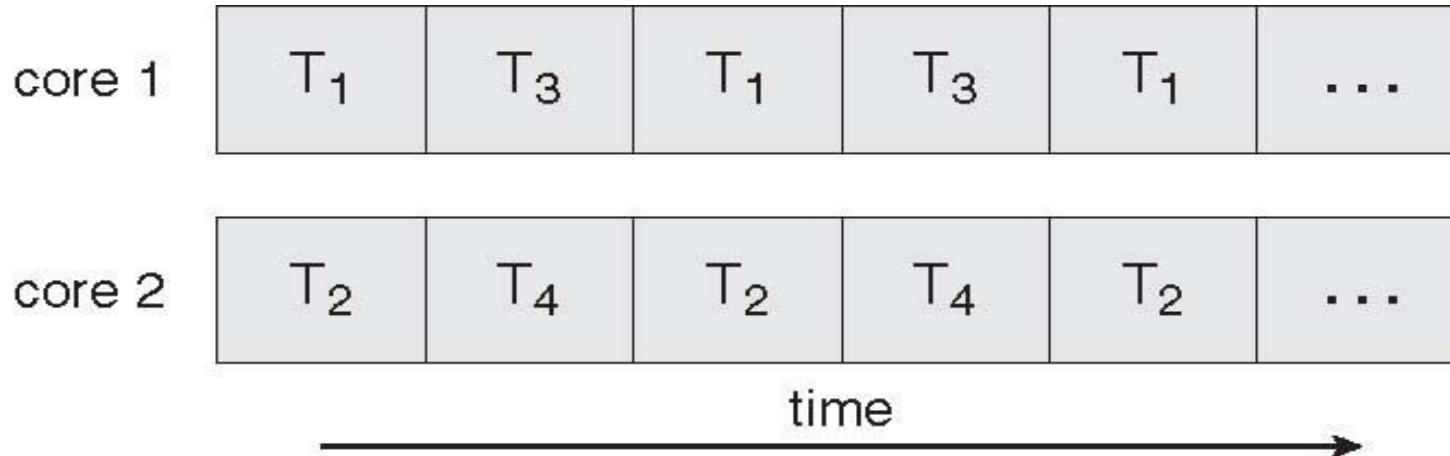
- **Responsiveness**
- **Resource Sharing**
- **Economy**
- **Scalability**

Single vs Multicore systems



Single core : Concurrency possible

Multicore : parallel execution possible



Multicore programming

- **Multicore systems putting pressure on programmers, challenges include:**
 - **Dividing activities**
 - **Balance**
 - **Data splitting**
 - **Data dependency**
 - **Testing and debugging**
 -

User vs Kernel Threads

- **User Threads: Thread management done by user-level threads library**
- **Three primary thread libraries:**
 - **POSIX Pthreads**
 - **Win32 threads**
 - **Java threads**
- **Kernel Threads: Supported by the Kernel**
- **Examples**
 - **Windows XP/2000**
 - **Solaris**
 - **Linux**
 - **Tru64 UNIX**
 - **Mac OS X**

User threads vs Kernel Threads

▪ User threads

- User level library provides a “typedef” called threads
- The scheduling of threads needs to be implemented in the user level library
- Need some type of timer handling functionality at user level execution of CPU
 - OS needs to provide system calls for this
- Kernel does not know that there are threads!

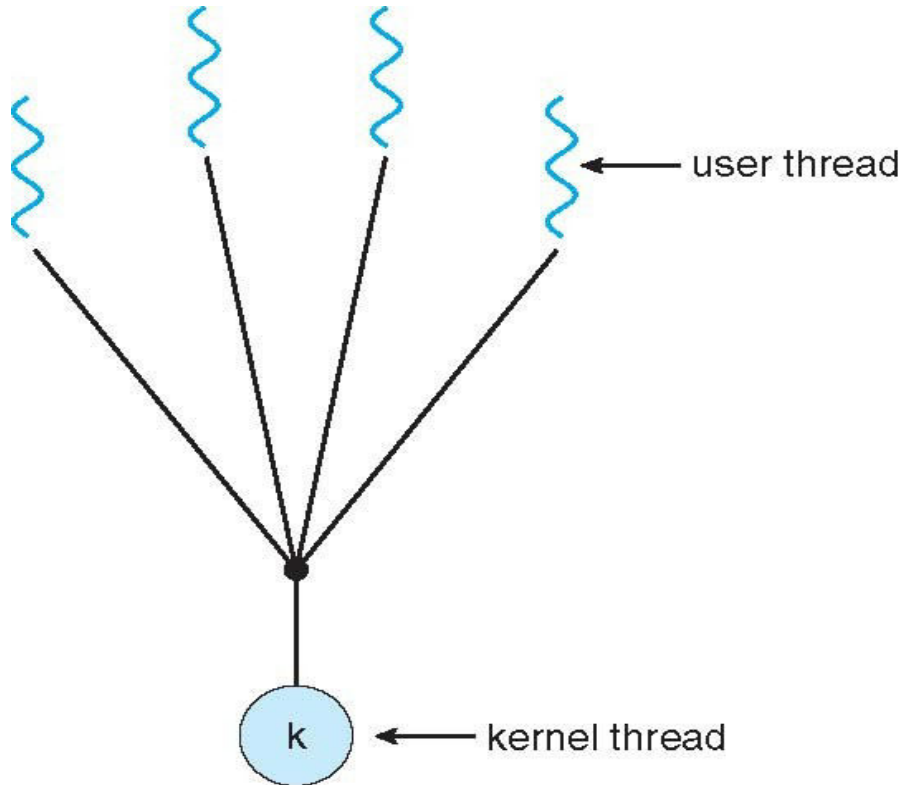
▪ Kernel Threads

- Kernel implements concept of threads
- Still, there may be a user level library, that maps kernel concept of threads to “user concept” since applications link with user level libraries
- Kernel does scheduling!

Multithreading models

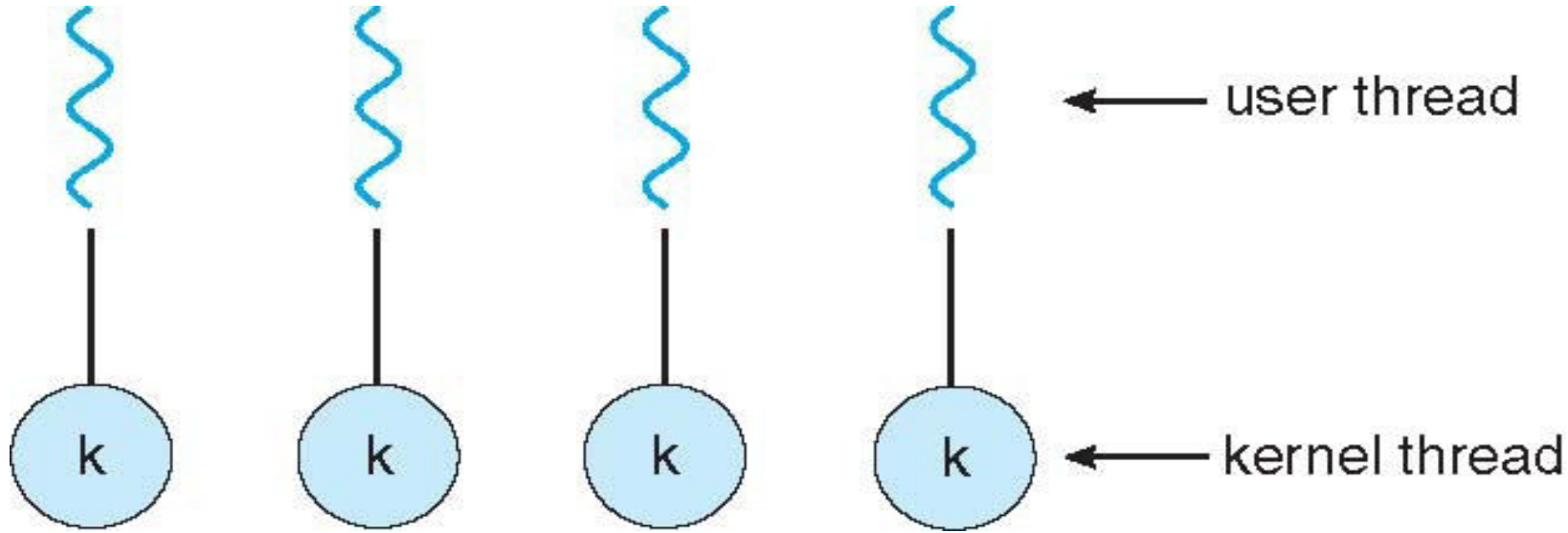
- **How to map user threads to kernel threads?**
 - Many-to-One
 - One-to-One
 - Many-to-Many
- **What if there are no kernel threads?**
 - Then only “one” process. Hence many-one mapping possible, to be done by user level thread library
 - Is One-One possible?

Many-One Model



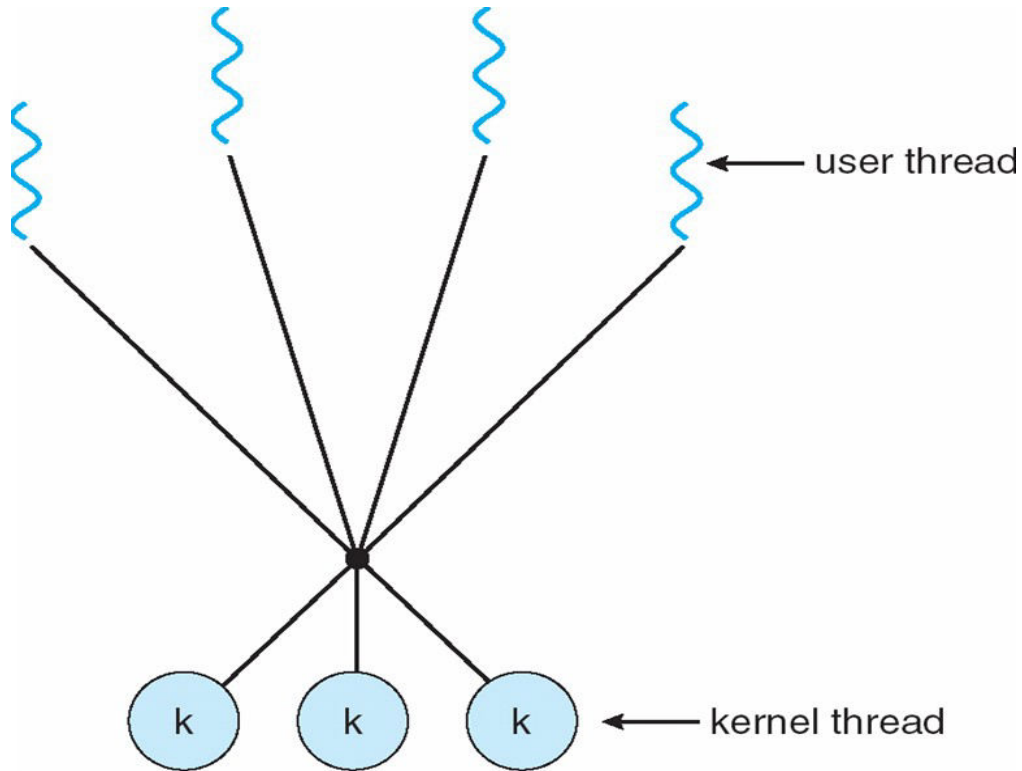
- **Many user-level threads mapped to single kernel thread**
- **Examples:**
 - Solaris Green Threads
 - GNU Portable Threads

One-One Model



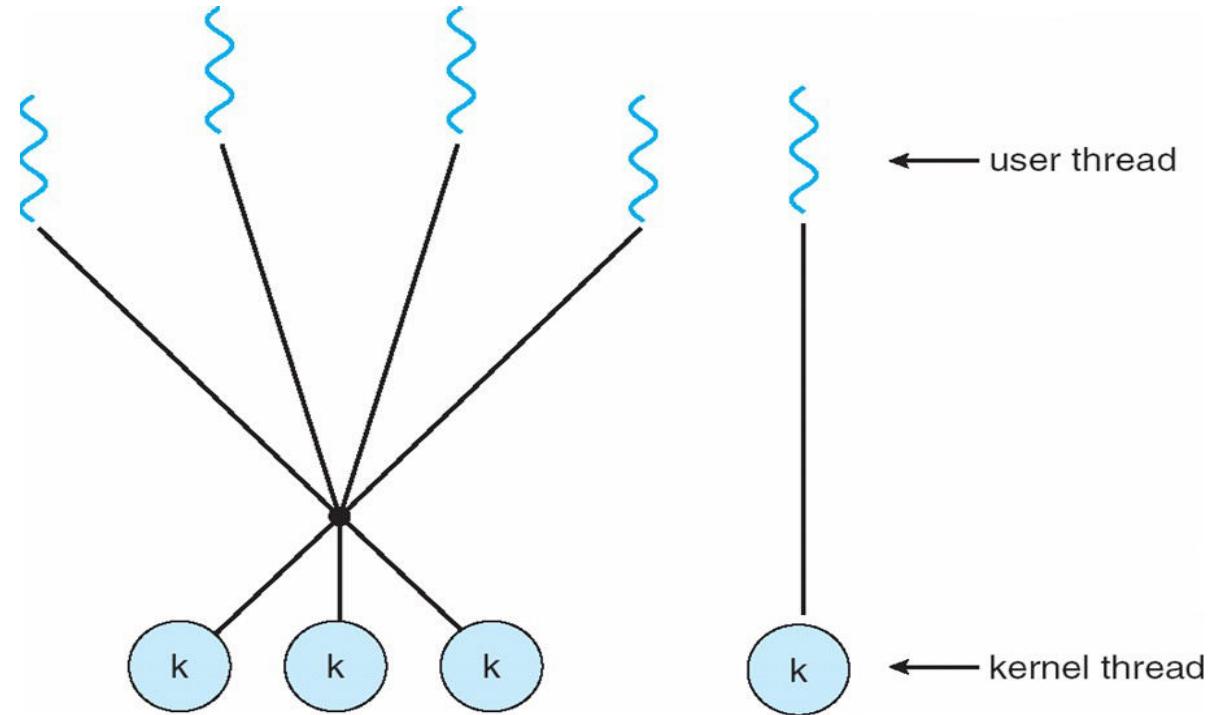
- Each user-level thread maps to kernel thread
- Examples
 - Windows NT/XP/2000
 - Linux
 - Solaris 9 and later

Many-Many Model



- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

Two Level Model



- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- Examples
 - IRIX
 - HP-UX
 - Tru64 UNIX
 - Solaris 8 and earlier

Thread Libraries

Thread libraries

- **Thread library provides programmer with API for creating and managing threads**
- **Two primary ways of implementing**
 - **Library entirely in user space**
 - **Kernel-level library supported by the OS**

pthread

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
-

Demo of pthreads code

**Demonstration on Linux – see the code,
compile and execute it.**

Other libraries

- **Windows threading API**
 - `CreateThread(...)`
 - `WaitForSingleObject(...)`
 - `CloseHandle(...)`
- **Java Threads**
 - The `Threads` class
 - The `Runnable` Interface

Issues with threads

- **Semantics of fork() and exec() system calls**
 - Does fork() duplicate only the calling thread or all threads?
- **Thread cancellation of target thread**
 - Terminating a thread before it has finished
 - Two general approaches:
 - Asynchronous cancellation terminates the target thread immediately.
 - Deferred cancellation allows the target thread to periodically check if it should be cancelled.

More on threads

Thread pools

- **Some kernels/libraries can provide system calls to :
Create a number of threads in a pool where they await work, assign work/function to a waiting thread**
- **Advantages:**
 - **Usually slightly faster to service a request with an existing thread than create a new thread**
 - **Allows the number of threads in the application(s) to be bound to the size of the pool**

Thread Local Storage (TLS)

- Thread-specific data, Thread Local Storage (TLS)
 - Not local, but global kind of data for all functions of a thread, more like “static” data
 - Create Facility needed for data private to thread
 - Allows each thread to have its own copy of data
 - Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
 - gcc compiler provides the storage class keyword `thread` for declaring TLS data

```
static __thread int threadID;
```

```
int arr[16];  
int f() {  
    a(); b(); c();  
}  
int g() {  
    x(); y();  
}  
int main() {  
    th_create(...,f,...);  
    th_create(...,g,...);  
}  
  
//arr is visible to all of them!  
//need data for only f,a,b,c  
//need data for only g,x,y
```


Scheduler activations for threads

Library

--

```
th_setup(int n) {
    max_threads = n;
    curr_threads = 0;
}

th_create(...., fn,....) {
    if(curr_threads < max_threads)
        create kernel thread;
    schedule fn on one of the kernel
    threads;
}
```

application

```
f() {
    scanf();
}

g() {
    recv();
}

h() {...}; i() {...}

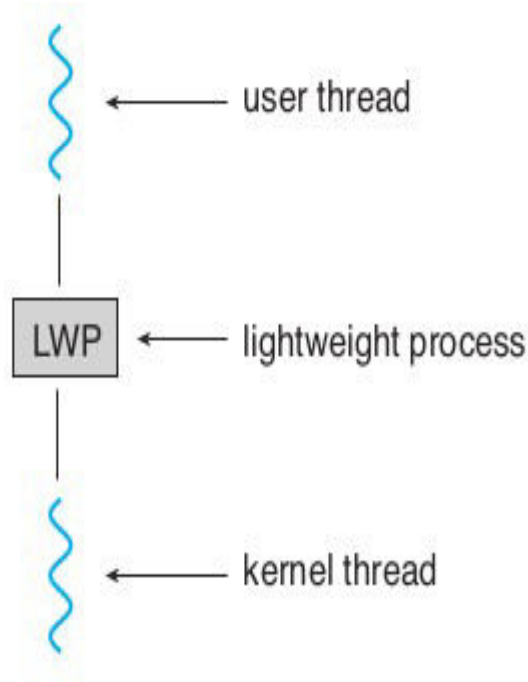
main()
    th_setup(2);
    th_create(...,f,...);
    th_create(...,g,...);
    th_create(...,h,...);
    th_create(...,i,...);
}
```

Scheduler activations for threads

- **Scheduler Activations**

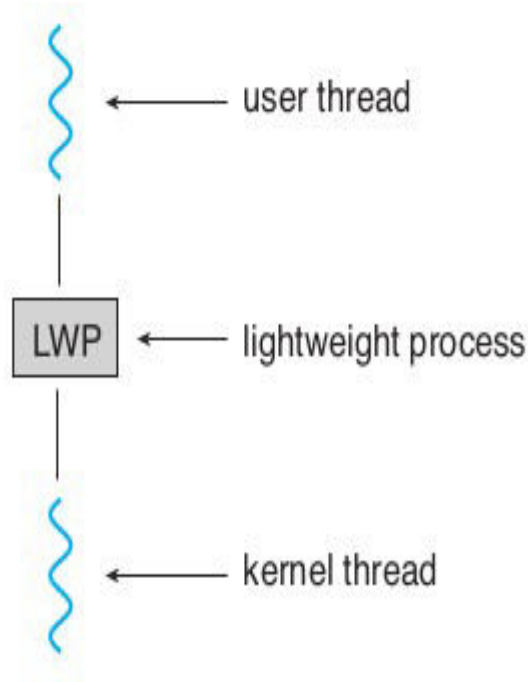
- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

Issues with threads



- **Scheduler Activations: LWP approach**
 - An intermediate data structure LWP
 - appears to be a virtual processor on which the application can schedule a user thread to run.
 - Each LWP attached to a kernel thread
 - Typically one LWP per blocking call, e.g. 5 file I/Os in one process, then 5 LWPs needed

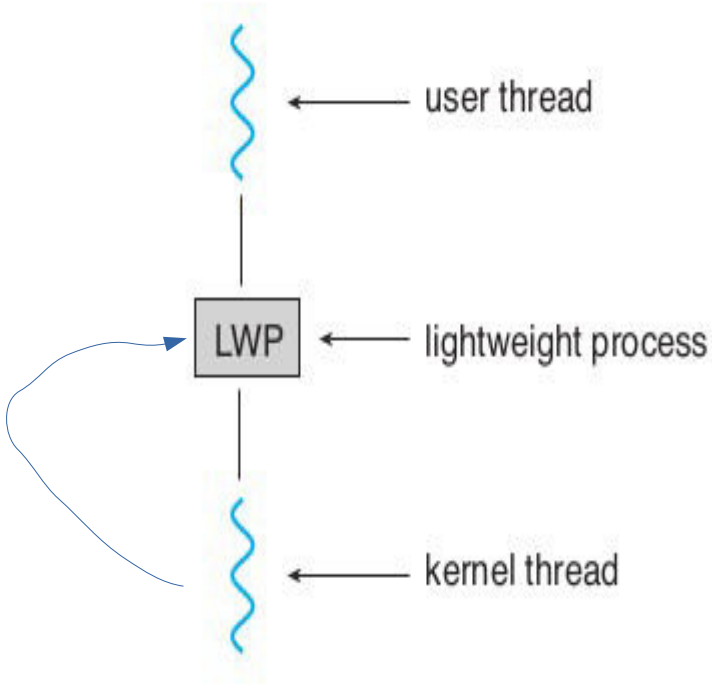
Issues with threads



- **Scheduler Activations: LWP approach**
 - Kernel needs to inform application about events like: a thread is about to block, or wait is over
 - This will help application relinquish the LWP or request a new LWP

Issues with threads

- **The actual upcalls**



Linux threads

- Only threads (called task), no processes!
- Process is a thread that shares many particular resources with the parent thread
- Clone() system call to create a thread

Linux threads

- **clone()** takes options to determine sharing on process create
- **struct task_struct** points to process data structures (shared or unique depending on clone options)
- **fork()** is a wrapper on top of clone()

flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

Issues in implementing threads project

- How to implement a user land library for threads?
- How to handle 1-1, many-one, many-many implementations?
- Identifying the support required from OS and hardware
- Identifying the libraries that will help in implementation

Issues in implementing threads project

- **Understand the clone() system call completely**
 - Try out various possible ways of calling it
 - Passing different options
 - Passing a user-land buffer as stack
- **How to save and restore context?**
 - C: setjmp, longjmp
 - Setcontext, getcontext(), makecontext(), swapcontext() functions
- **Sigaction is more powerful than signal**
 - Learn SIGALRM handling for timer and scheduler, timer_create() & timer_stop() system calls
- **Customized data structure to store threads, and manage thread-lists for scheduling**

Signals

Signals

- **Signals are used in UNIX systems to notify a process that a particular event has occurred.**
- **Signal handling**
 - Synchronous and asynchronous
- **A signal handler (a function) is used to process signals**
 - Signal is generated by particular event
 - Signal is delivered to a process
 - Then, signal is “handled” by the handler

Signals

- **More about signals**
 - Different signals are typically identified as different numbers
 - Operating systems provide system calls like `kill()` and `signal()` to enable processes to deliver and receive signals
 - `Signal()` - is used by a process to specify a “signal handler” – a code that should run on receiving a signal
 - `Kill()` is used by a process to send another process a signal
 - There are restrictions on which process can send which signal to other processes

Demo

- Let's see a demo of signals with respect to processes
- Let's see signal.h
 - /usr/include/signal.h
 - /usr/include/asm-generic/signal.h
 - /usr/include/linux/signal.h
 - /usr/include/sys/signal.h
 - /usr/include/x86_64-linux-gnu/asm/signal.h
 - /usr/include/x86_64-linux-gnu/sys/signal.h
- man 7 signal
- Important signals: SIGKILL, SIGUSR1, SIGSEGV, SIGALRM, SIGCLD, SIGINT, SIGPIPE, ...

Signal handling by OS

```
Process 12323 {  
    signal(19, abcd);  
}
```

```
OS: sys_signal {  
    Note down that process 12323  
    wants to handle signal number  
    19 with function abcd  
}
```

```
Process P1 {  
    kill (12323, 19) ;  
}
```

```
OS: sys_kill {  
    Note down in PCB of process 12323 that  
    signal number 19 is pending for you.  
}
```

When process 12323 is scheduled, at that time the OS will check for pending signals, and invoke the appropriate signal handler for a pending signal.

Threads and Signals

- **Signal handling Options:**
 - Deliver the signal to the thread to which the signal applies
 - Deliver the signal to every thread in the process
 - Deliver the signal to certain threads in the process
 - Assign a specific thread to receive all signals for the process