BITS, PILANI – K. K. BIRLA GOA CAMPUS

Operating Systems

by

Mrs. Shubhangi Gawali

Dept. of CS and IS

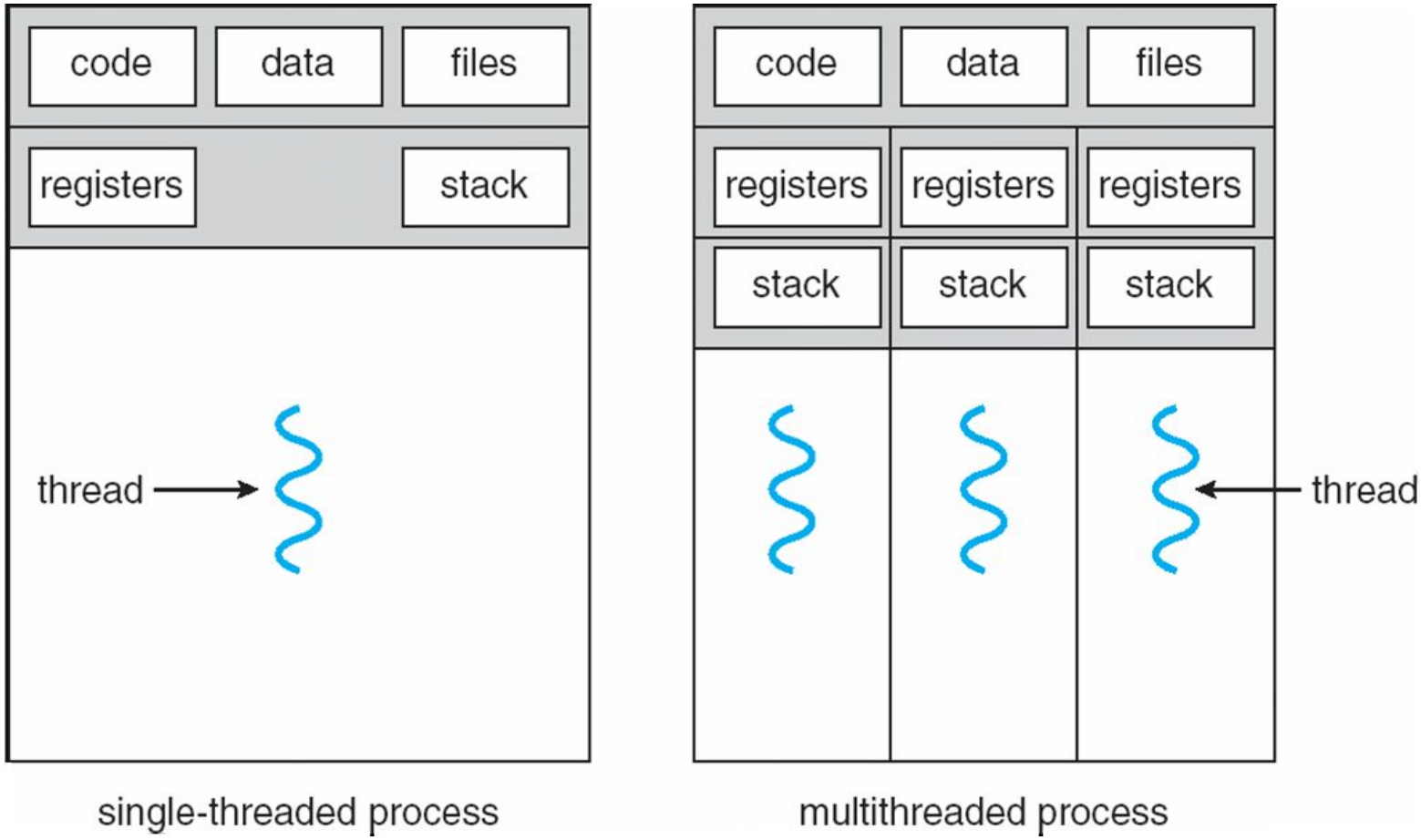
Lecture 17

Multithreaded Programming

Processes and Threads

PROCESSES	THREADS
Multiple simultaneous Programs	Multiple simultaneous Functions
Independent memory space	Shared memory space
Independent open file-descriptors	Shared open file-descriptors

Single and Multithreaded Processes



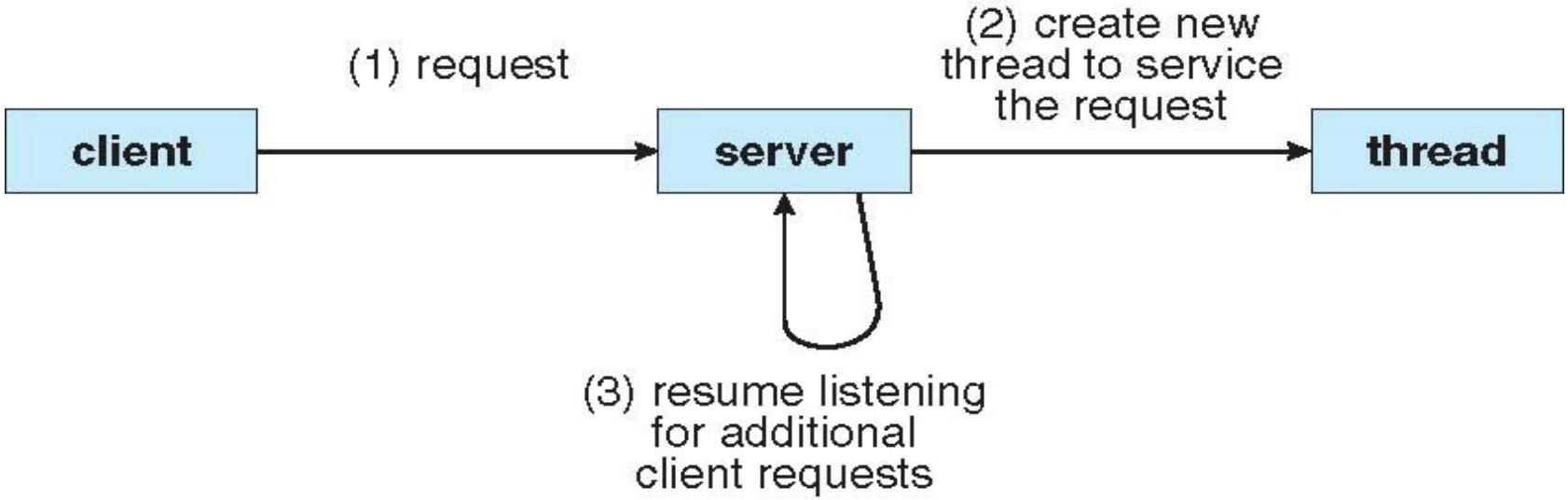
single-threaded process

multithreaded process

Motivation

- Threads run within application
- 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
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Multithreaded Server Architecture



a Thread

- A *thread (or lightweight process) is a basic unit*

of CPU utilization; it consists of:

  - Thread ID
  - program counter
  - register set, stack pointer
  - stack space for local variables and return addresses
  - Signal mask
  - Priority
  - Return value : errno

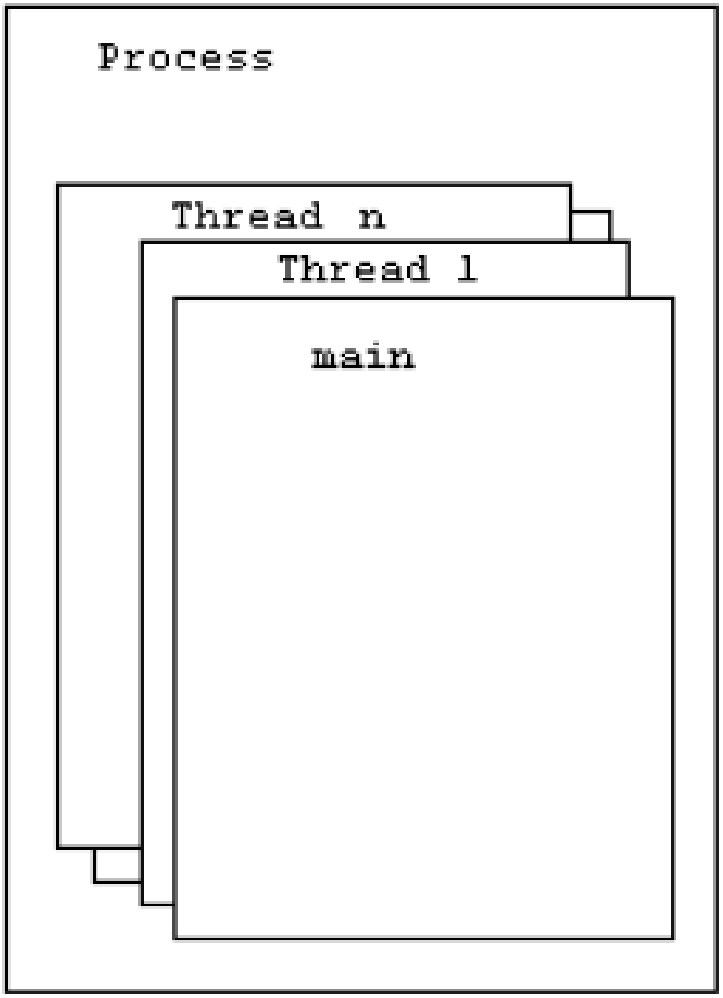
- A thread shares with its peer threads its:
  - Code section
  - Global data section
  - Operating-system resources
  - Process instructions
  - Open files (descriptors)
  - Signal and signal handlers
  - Current working directory
  - User and group id
- And is collectively know as a *task.*

- A traditional or *heavyweight process is equal* to a task with one thread

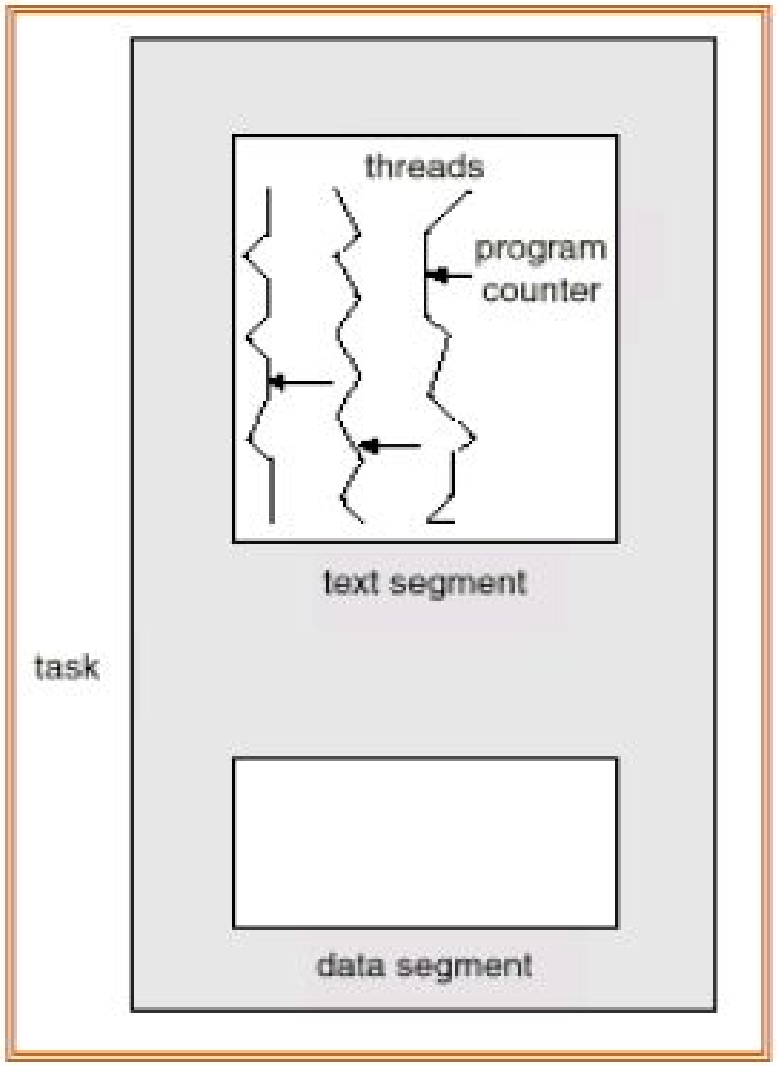
- Unlike process thread does not maintain a list of created threads.

- It does not know the thread created it also

a Process



Multiple threads within a task



Local to a thread

- Thread ID
- program counter
- register set, stack pointer
- stack space for local variables
- return addresses
- Signal mask
- Priority
- Return value : errno

Global to all threads

- Code section
- Global data section
- Operating-system resources
- Process instructions
- Open files (descriptors)
- Signal and signal handlers
- Current working directory
- User and group id

Benefits

- **Responsiveness**
  - Even if one thread is blocked other threads can continue execution
- **Resource Sharing**
  - Sharing memory & other resources of the process it belongs to
- **Economy**
  - It is more economical to create & context switch threads
- **Utilization of multiprocessor architectures**
  - Increases multi threading (threads can run parallel)

Benefits contd…

- Takes less time to create a new thread than a process
- Less time to terminate a thread than a process
- Less time to switch between two threads within the same process
- Since threads within the same process share memory and files, they can communicate with each other without invoking the kernel

pthread

- #include <pthread.h>
- Define a worker function

  void *func(void *args) { }
- Initialize pthread _attr_t

  pthread_attr_t attr;

  pthread _attr _init(attr);
- Create a thread

  pthread _t thread;

  pthread_create(&thread, &attr, worker function, arg);
- Exit current thread

  pthread_exit(status)

Compiling

- Use -pthread

pthread examples

Lecture 18

Multithreaded Programming

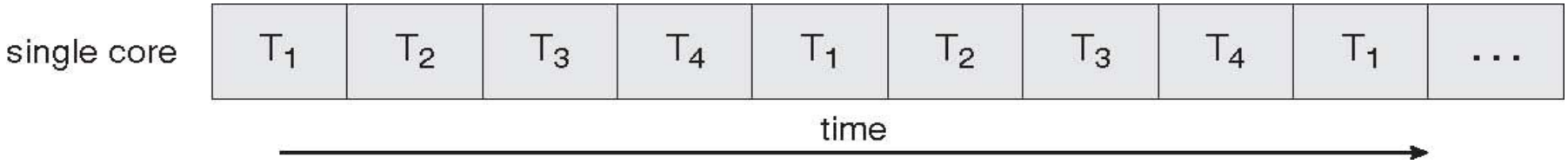
Thread Management

- **int pthread_detach(pthread_t *thread*);**
  - The **pthread_detach**() function marks the thread identified by *thread* as detached. When a detached thread terminates, its resources are automatically released back to the system without the need for another thread to join with the terminated thread.
  - The following statement detaches the calling thread:
    - pthread_detach(pthread_self());

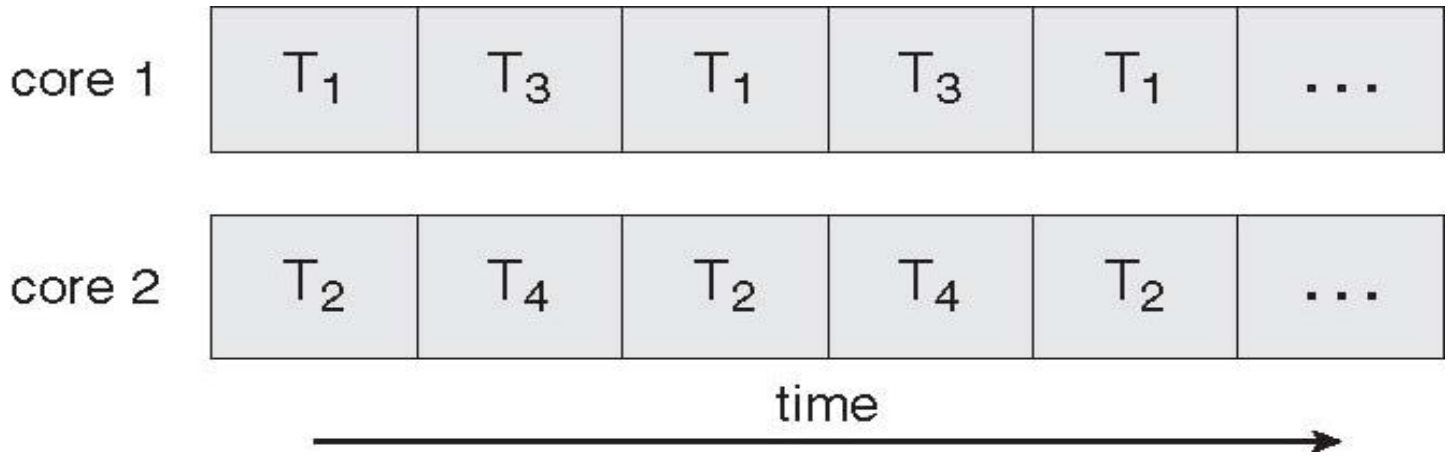
Thread Management

- **int pthread_join(pthread_t** *thread*, **void *****retval*);
  - The **pthread_join**() function waits for the thread specified by *thread* to terminate.
  - If that thread has already terminated, then **pthread_join**() returns immediately. The thread specified by *thread* must be joinable.
  - If *retval* is not NULL, then **pthread_join**() copies the exit status of the target thread (i.e., the value that the target thread supplied to pthread_exit(3)) into the location pointed to by **retval*.
  - If the target thread was canceled, then **PTHREAD_CANCELED** is placed in **retval*.
  - If multiple threads simultaneously try to join with the same thread, the results are undefined. If the thread calling **pthread_join**() is canceled, then the target thread will remain joinable (i.e., it will not be detached).

Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System



Multicore Programming

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

Types of Parallelism

- Data Parallelism
- Task Parallelism

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

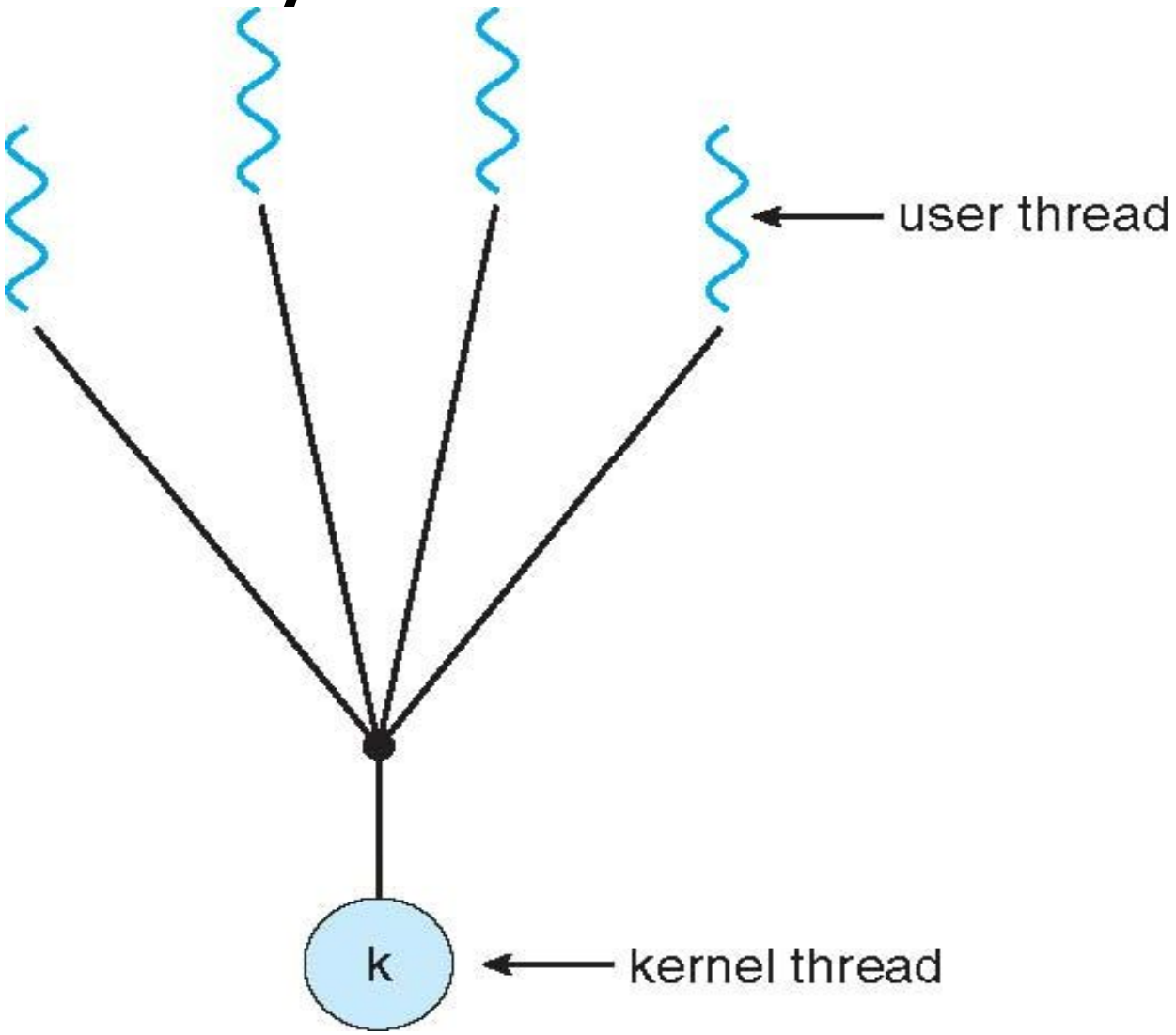
Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Many user-level threads mapped to single kernel thread
- Shortcomings
  - If a thread makes a blocking system call entire process is blocked
  - Multiple threads are unable to read in parallel on MP
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**

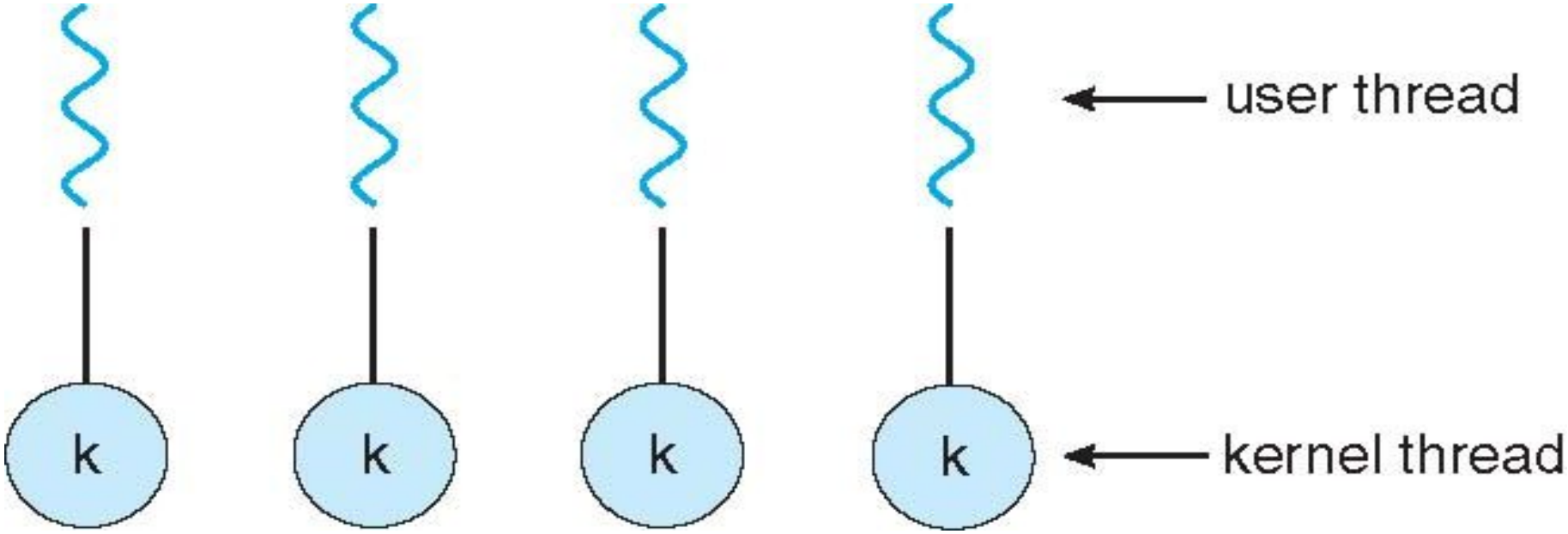
Many-to-One Model



One-to-One

- Each user-level thread maps to kernel thread
- Allow multiple threads to run in parallel on MP, also when a thread makes a blocking system call
- Only drawback is creating a user thread requires creating the corresponding kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later

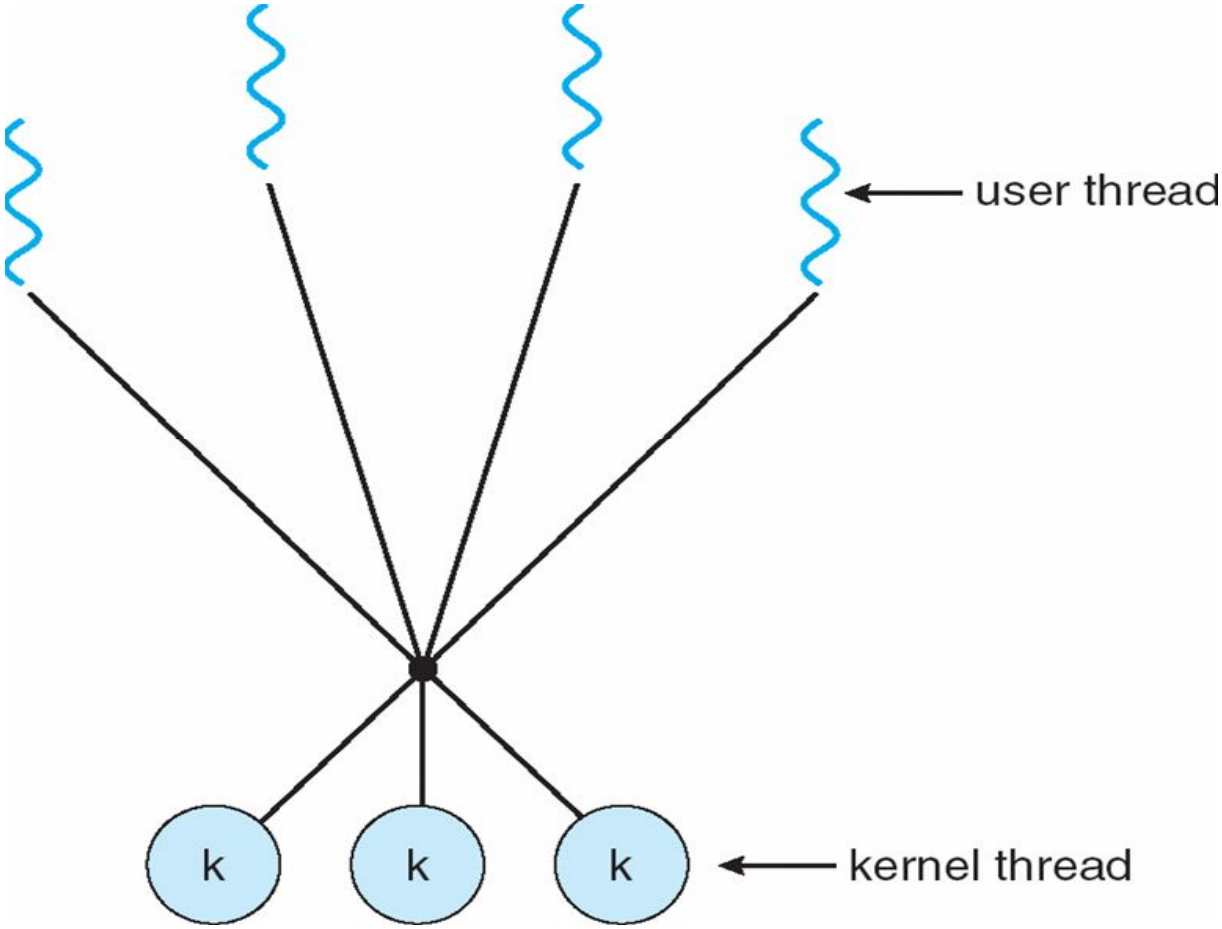
One-to-one Model



Many-to-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
- Kernel threads can run in parallel on MP, also when a thread performs a blocking system call
- Examples
  - Solaris prior to version 9
  - Windows NT/2000 with the *ThreadFiber* package

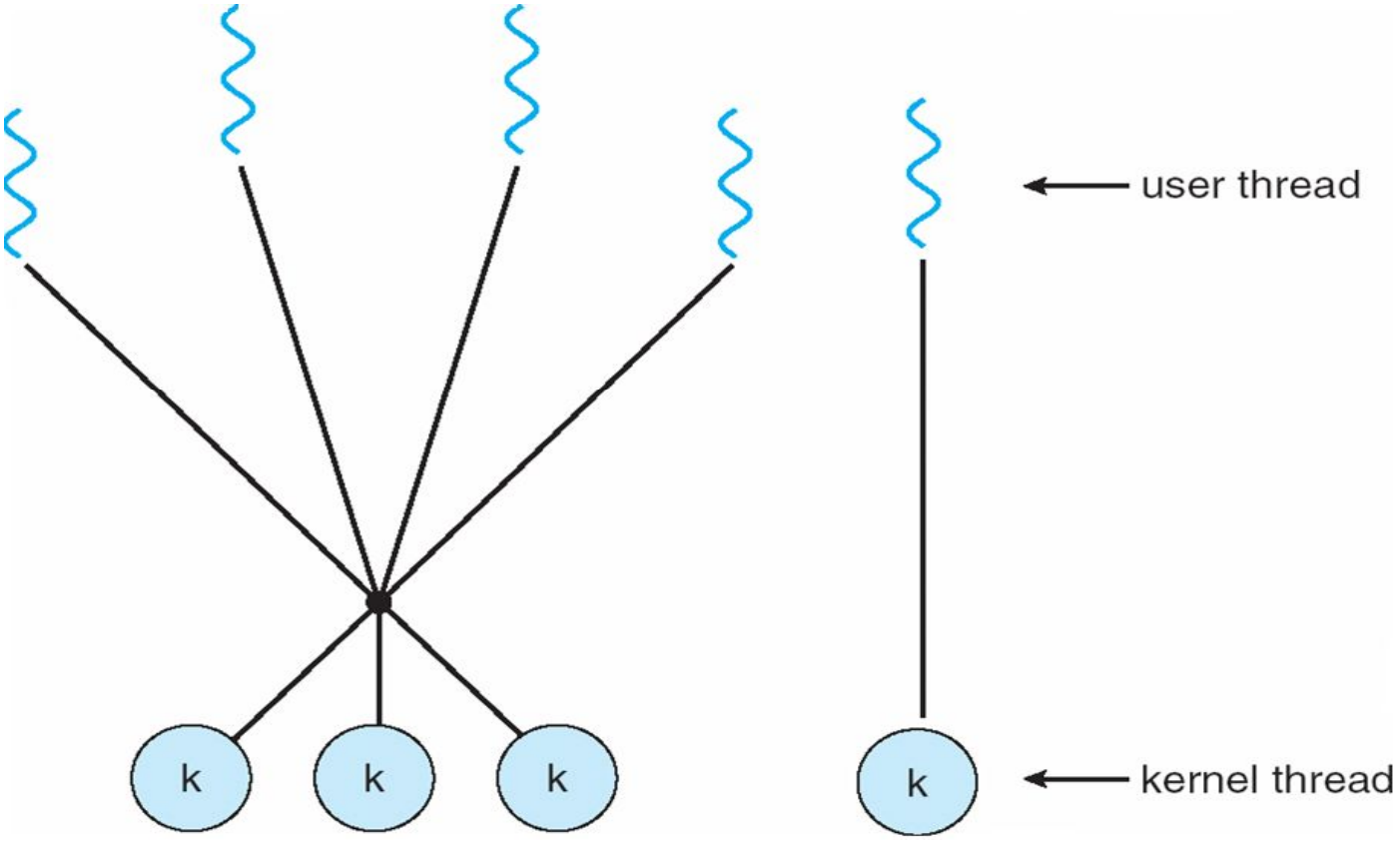
Many-to-Many Model



Two-level Model

- Variation on M:N model,
- Multiplexes many user level thread to a smaller or equal number of kernel threads
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Two-level Model



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads

- Two primary ways of implementing
  - Library entirely in user space (like local function call)
  - Kernel-level library supported by the OS (system call)

Pthreads

- 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)

Threading Issues

- Semantics of **fork()** and **exec()** system calls

Q: If one thread forks, is the entire process copied, or is the new process single-threaded?

A: System dependant.

A: If the new process execs right away, there is no need to copy all the other threads. If it doesn't, then the entire process should be copied.

A: Many versions of UNIX provide multiple versions of the fork call for this purpose.

Thread Cancellation

- 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.
- System-wide resources

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- Signal may be received **Synchronously or Asynchronously**
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled (Default signal handler/user defined signal handler)

Signal Handling

Q: When a multi-threaded process receives a signal, to what thread should that signal be delivered?

A: There are four major 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

- Uses system call kill(pthread_t pid, int signal) and pthread_kill(pthread_t tid, int signal)

Lecture 19

Multithreaded Programming

Thread Pools

- Create a number of threads in a pool where they await work

- 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 Specific Data

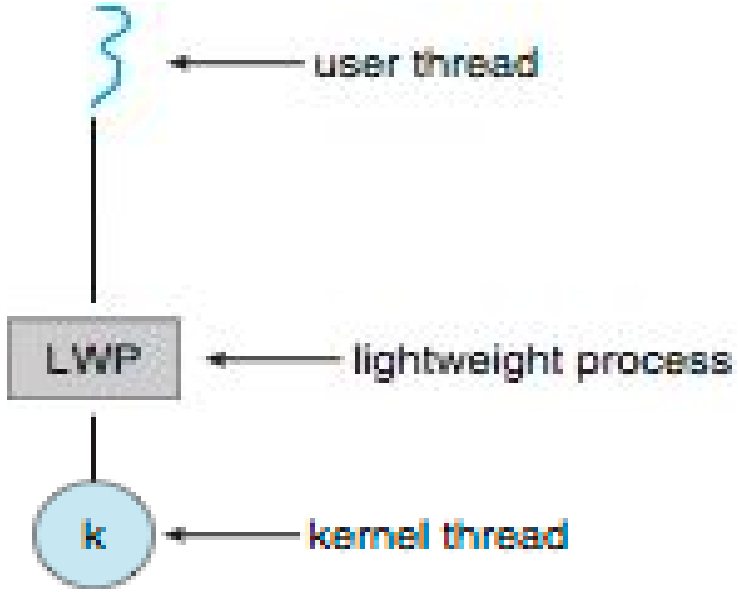
- Allows each thread to have its own copy of data

- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

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 of kernel threads

Lightweight Processes



Linux Threads

- Linux refers to them as *tasks* rather than *threads*
- Thread creation is done through `clone()` system call
- `clone()` allows a child task to share the address space of the parent task (process)
- `struct task_struct` points to process data structures (shared or unique)

Linux Threads

- `fork()` and `clone()` system calls
- Doesn't distinguish between process and thread
  - Uses term ***task*** rather than thread
- `clone()` takes options to determine sharing on process create
- `struct task_struct` points to process data structures (shared or unique)

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.

Thread Scheduling

- Distinction between user-level and kernel-level threads

- Many-to-one and many-to-many models, thread library schedules user-level threads to run on LWP
  - Known as **process-contention scope (PCS)** since scheduling competition is within the process

- Kernel thread scheduled onto available CPU is **system-contention scope (SCS)** – competition among all threads in system

Pthread Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD SCOPE PROCESS schedules threads using PCS scheduling
  - PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

Pthread Scheduling API

```
#include <pthread.h>
#include <stdio.h>
#define NUM THREADS 5
int main(int argc, char *argv[])
{
     int i;
    pthread t tid[NUM THREADS];
    pthread attr t attr;
    /* get the default attributes */
    pthread attr init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread attr setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread attr setschedpolicy(&attr, SCHED_OTHER);
    /* create the threads */
    for (i = 0; i < NUM THREADS; i++)
            pthread create(&tid[i],&attr,runner,NULL);
```

Pthread Scheduling API

```
    /* now join on each thread */
    for (i = 0; i < NUM THREADS; i++)
        pthread join(tid[i], NULL);
}
 /* Each thread will begin control in this function */
void *runner(void *param)
{
    printf("I am a thread\n");
    pthread exit(0);
}
```