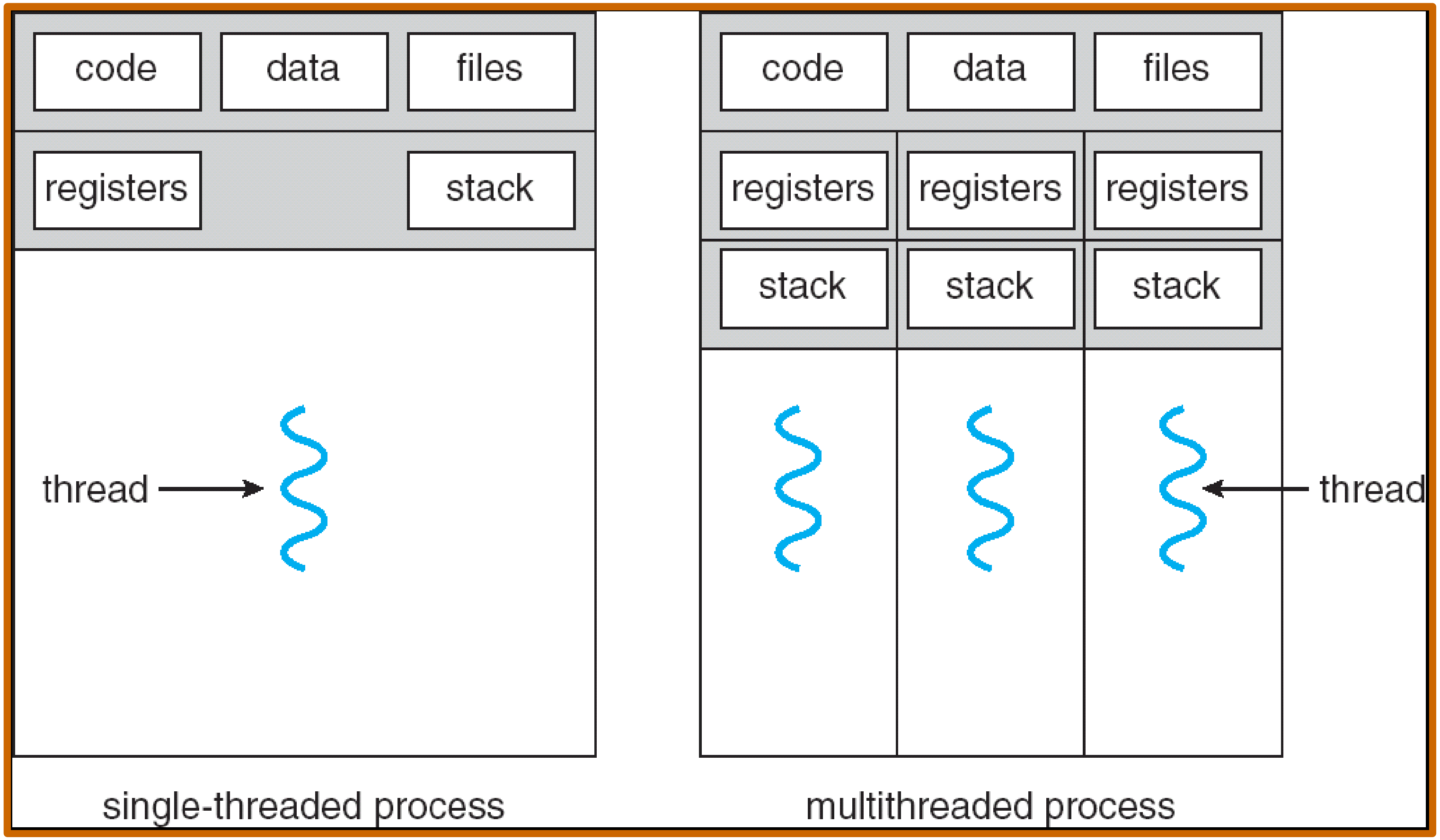
Threads

Threads

- A basic unit of of CPU utilization:
  - Thread ID.
  - Program counter.
  - Register set.
  - Stack.
- There may be multiple threads in a single process. Those threads share:
  - Code.
  - Data.
  - Other OS resources.
- Sometimes referred to as LWP.

Threads


code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

Advantages

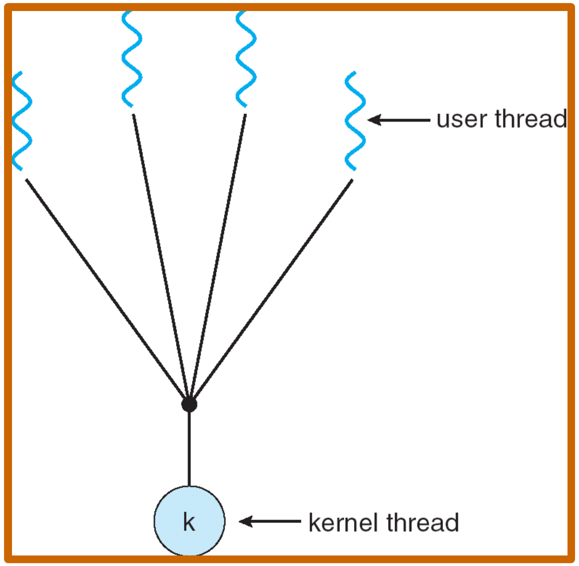
- Responsiveness.
  - When one thread is waiting another can proceed.
- Resource Sharing.
  - Instant, cheap “IPC”.
- Economy.
  - Threads have lower overhead than processes.
- Utilization of MP Architectures.
  - More and more important in an era of multi-core CPU's.

User Threads vs. Kernel Threads

- Who manages the stack, registers, and PC for a thread?
- User-level thread library:
  - Kernel has no concept of threads.
  - User level library represents and schedules multiple threads on top of a single kernel process.
- Kernel threads:
  - Kernel explicitly represents and schedules individual threads.
- Potential confusion:
  - Linux “kernel thread” vs. kernel thread.

Threading Models: Many to One

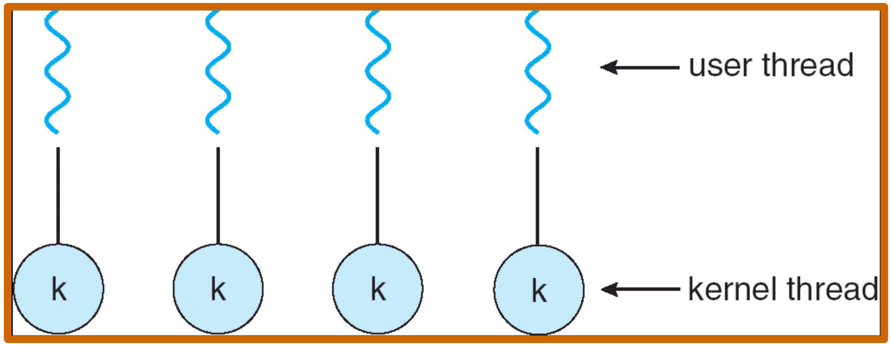
- Multiple user threads mapped to one kernel thread.
  - Multiple stacks etc. handled by user level library.
  - No speedup gained from MP systems.
  - There *are* advantages for responsiveness, resource sharing, and economy.



- Examples:
  - Solaris Green Threads
  - GNU Portable Threads

Threading Models: One to One

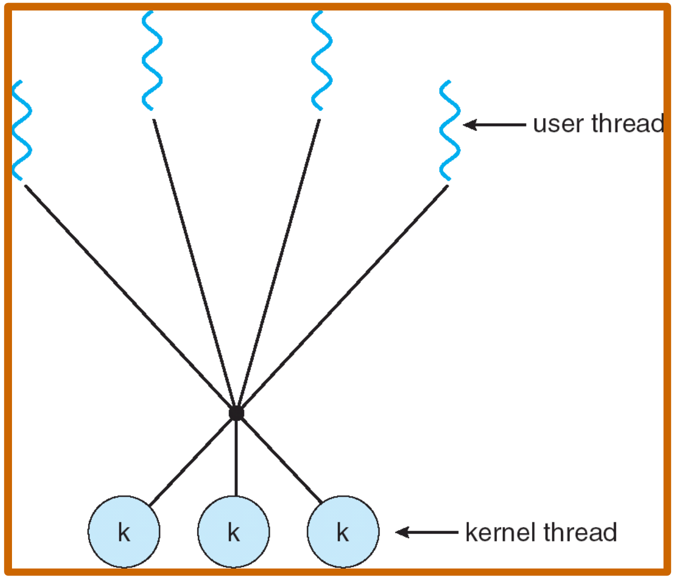
- Exactly one kernel thread for every user thread.
- Examples:
  - Windows NT/XP/2000.
  - Linux. (more in a minute...)
  - Solaris 9 and later.



- Allows better utilization of MP.
- Disadvantage: Requires kernel to maintain a very large set of threads.
- Every entry into kernel space can be costly.

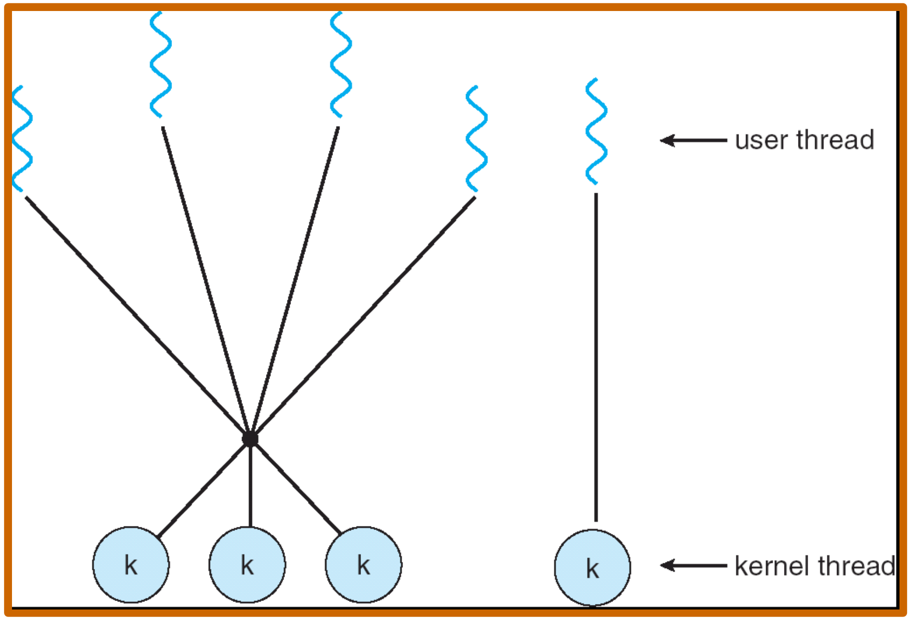
Threading Models: Many to Many

- Many user level threads are mapped to many kernel threads.
- Allows the kernel to create a sufficient, but manageable, number of kernel threads.
  - Solaris prior to version 9
  - Windows NT/2000 with the ThreadFiber package.
- Hybrid of user level threads and kernel level threads.



Threading Model: Two Level

- Variation on M:M, that allows a user thread to be bound to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier
- Barely qualifies as a distinct model.



Example Threading Libraries: pthreads

- POSIX threads.
- Implemented on many OS's.
- Association between user threads and kernel threads depends on implementation.
- Let's look at an example...

Java Threads

- Thread capabilities are usually provided by a library or by the kernel.
- Java provides language level thread support.
- Mapping from Java threads to kernel threads depends on JVM implementation.
- Let's look at an example...

Win32 Threads

- There is some sample code in your book :)

OS Examples: Linux Threads

- The Linux kernel does not distinguish between processes and threads.
- Everything is a “task”. Tasks may or may not share an address space.
- `clone()` system call allows creation of tasks.
- Similar to `fork()`, but accepts parameters that specify the degree of resource sharing.
- Sharing is handled by pointers in the kernel's task control structures.
  - Each thread has it's own struct.
  - The fields can point to shared objects.

Kernel Stacks

- Every Linux task has a small kernel stack in addition to the process stack.
- Why is that???

Windows XP Threads

- Each thread contains:
  - A thread id.
  - Register set.
  - Separate user and kernel stacks.
  - Private data storage area.
- (Not much different from Linux.)
- The register set, stacks, and private storage area are known as the context of the threads.

Windows XP Threads

- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
    - Pointer to the parent process.
    - Address of the starting routine.
    - Pointer to KTHREAD.
  - KTHREAD (kernel thread block)
    - Scheduling and synchronization info.
    - Kernel stack.
    - Pointer to TEB.
  - TEB (thread environment block)
    - Stack.
    - Other thread specific data.

Threading Issues

- Semantics of fork() and exec() system calls.
- Thread cancellation.
- Signal handling.
- Thread pools.
- Thread specific data.

fork/exec

- What happens when a multi-threaded program calls fork()?
  - All threads could be duplicated.

    OR
  - Just the calling thread.
    - POSIX fork works this way.
- Either way exec() replaces the entire process, including threads.

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 canceled
- Relevant POSIX functions:

```
int pthread_cancel(pthread_t thread);

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred.
- A signal handler is used to process signals
  - Signal is generated by particular event
  - Signal is delivered to a process
  - Signal is handled
    - Kernel provides default signal handler.
    - Process may override it.

Signal Example

```
/* the signal handler function */
void handle_SIGINT() {
    write(STDOUT_FILENO,buffer,strlen(buffer));

    exit(0);
}

int main(int argc, char *argv[])
{
    /* set up the signal handler */
    struct sigaction handler;
    handler.sa_handler = handle_SIGINT;
    sigaction(SIGINT, &handler, NULL);

    strcpy(buffer,"Caught <ctrl><c>\n");

    /* wait for <control> <C> */
    while (1)
        ;

    return 0;
}
```

Signal Handling With Threads

- Options:
  - Deliver the signal to the thread to which the signal applies
    - Synchronous signals: signals caused by the thread. E.g. divide by 0.
  - Deliver the signal to every thread in the process.
    - First one willing to take it, gets it.
  - Deliver the signal to certain threads in the process.
    - Only some threads willing to take it.

Sending Signals in Linux

- Some functions:

```
int kill(pid_t pid, int sig);
```

  - “Kill” is a bit of a misnomer.

```
int pthread_kill(pthread_t thread, int sig)
```

  - Sends a signal to a specific thread.

Thread Pools

- Create a number of threads in a pool where they await work.
- Advantages:
  - Faster to use an existing thread than to create a new thread.
  - Allows the number of threads in the application(s) to be bound to the size of the pool.
    - Once a process has exhausted its pool, it waits.

Acknowledgments

- Portions of these slides are taken from Power Point presentations made available along with:
  - Silberschatz, Galvin, and Gagne. Operating System Concepts, Seventh Edition.
- Original versions of those presentations can be found at:
  - http://os-book.com/