Threads

Chapter 4

From Cooperating Processes to Threads

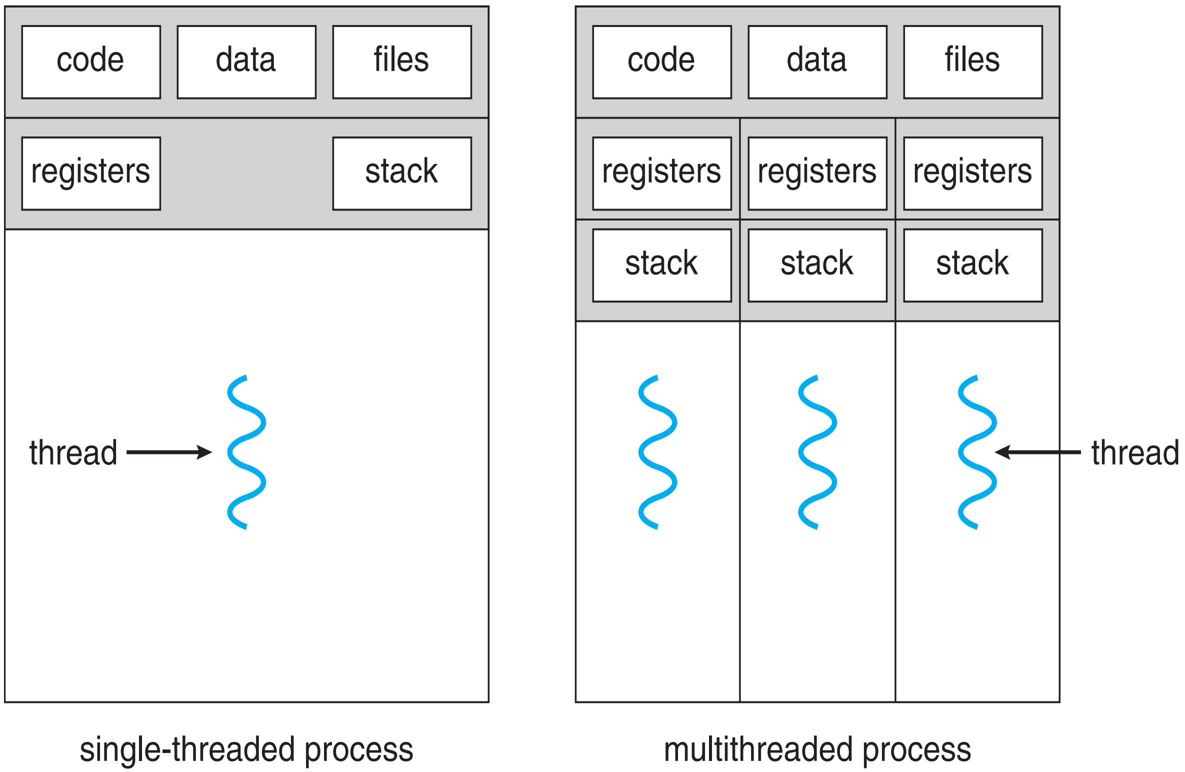
- Lots of value in cooperating processes
  - Computational speedup, modularity, etc.
- What if a few processes wanted to share **everything?** (well, almost everything)
  - e.g., memory, open files, executable code
  - Still independently schedulable, able to run concurrently
- Then, my friend, you'd have yourself multiple threads (in a single process)

Threads

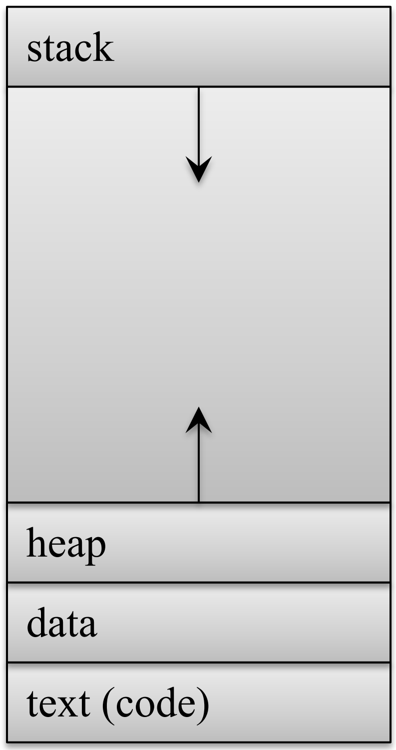
- A Multi-Threaded Process
  - Each using the same code, data, allocated resources
  - Each doing its own thing
    - Running in a different section of the code
    - Calling different subroutines
  - So, each thread needs:
    - Its own set of CPU registers
    - Its own region of memory as a runtime stack

Threads

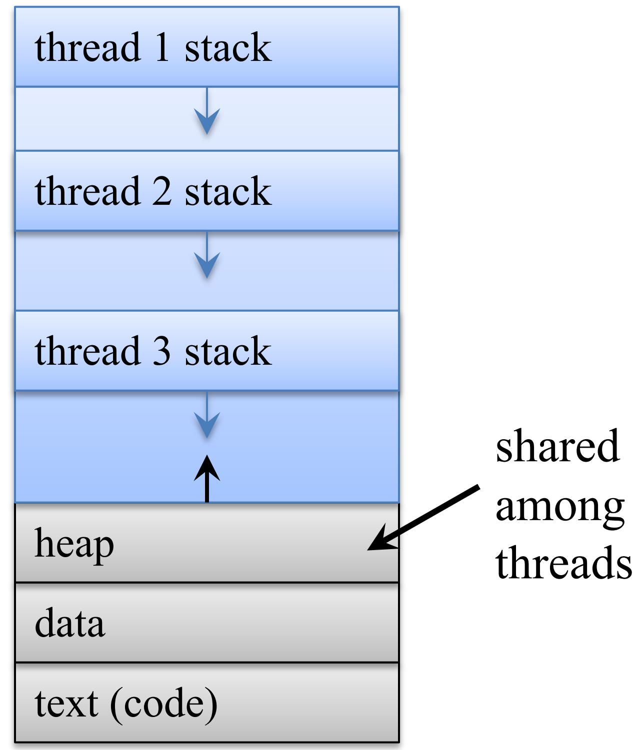
- An abstraction for the ability to execute on a CPU core



Process Address Space



single threaded address space



multi-threaded address space

POSIX Thread Example

- *Start routine* for each new thread
- Include pthread.h
- Compile with **-lpthread**
- System Calls

```
pthread_create( &thread, NULL, startRoutine, argPtr );
pthread_exit( returnPtr );
pthread_join( tid, &returnPtr );
```

- helloThreads.c

Single- vs Multi-Threaded Processes

- Without multi-threading, the process is:
  - The owner of resources (memory, files, etc.)
  - The unit of scheduling/execution
    - Including execution state, occupant of scheduling queues, etc.
- With multi-threading, we have a distinction
  - A thread is the unit of scheduling/execution
  - A process is a container for resources
  - A process has a collection of threads

Implementing Threads in the OS

- Within a process, threads share as much as they can
  - Memory: code, global data, heap
  - Process id
  - Open files, IPC facilities, other OS resources
- Keep as little per-thread state as possible
  - Execution state (running, ready, waiting, etc.)
  - Execution context (program counter, stack pointer, CPU registers)

存在这里

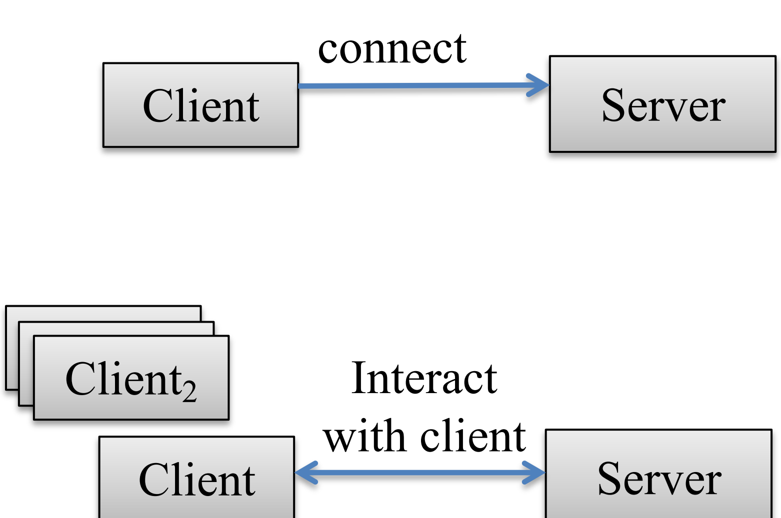
  - Per-thread stack
  - Keep the per-thread cost low
- A Thread Control Block for each thread in a process

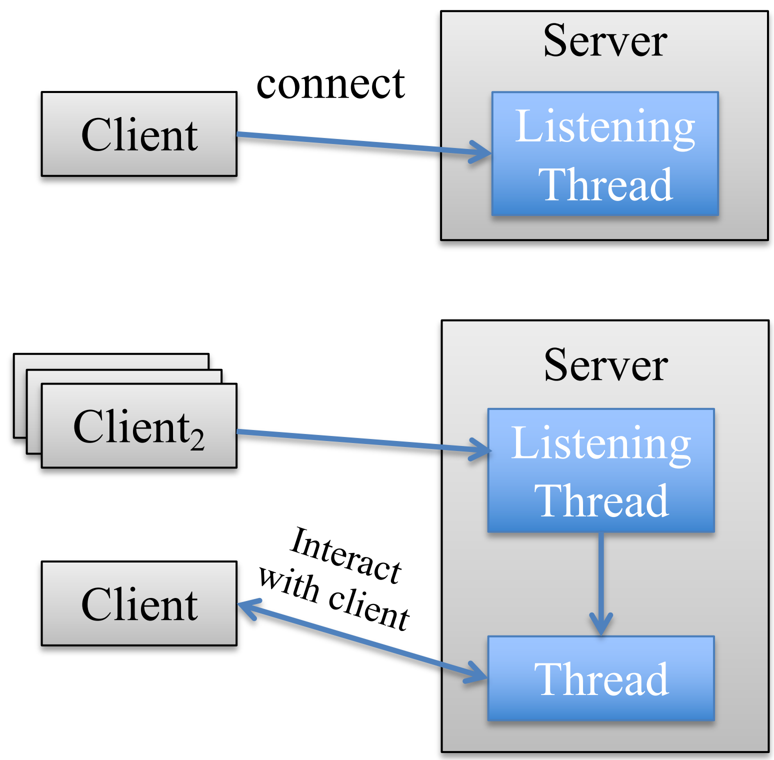
Threads, Who Needs Them

- Do we really need multiple threads, could we just use cooperating processes instead?
- Multi-threading has some advantages
  - Much lower cost (to create/terminate, per-thread overhead), **Example**
  - Faster context switch (between threads in the same process), as it does not have to change protection information (e.g., memory bounds)
  - Efficient thread communication within a process
  - Simplified programming structure
    - Automatic sharing of memory
    - Easy to create a thread for concurrent jobs
    - Easy to keep making progress even if one thread has to block

A Multi-Threaded Server

- Single-threaded server : one thread to accept connections and interact with each client
- Multi-threaded server : a new thread to handle interaction with each client





A Multi-Threaded Server

Single-Threaded

```
main()
{
  sock = socket( … );
  listen( … );
  while (1) {
    new_sock = accept(sock);

    handleRequest(new_sock);
  }
}

void handleRequest(int s){
  /*do something then return*/
  …
}
```

Multi-Threaded

```
main()
{
  sock = socket( … );
  listen( … );
  while (1) {
    new_sock = accept(sock);
    args = ...;

    pthread_create(&tid, NULL,
           handleRequest, args );
  }
}

void *handleRequest(void *args ){
  /*do something then exit*/
  …
}
```

Threads and Context Switching

- Context switching among threads
  - Like switching between processes
  - Asynchronous events (e.g, I/O interrupt from a device, a timer interrupt)
  - Synchronous events (e.g., thread makes a blocking system call, voluntarily gives up the CPU with sched_yield())
- Context switch is simplified when switching among threads in the same process
  - Don't have to change protection information (e.g., memory bounds)
  - So easy, a process could do it without help from the kernel
  - And maybe a process would want to …

Threads Implementation Alternatives

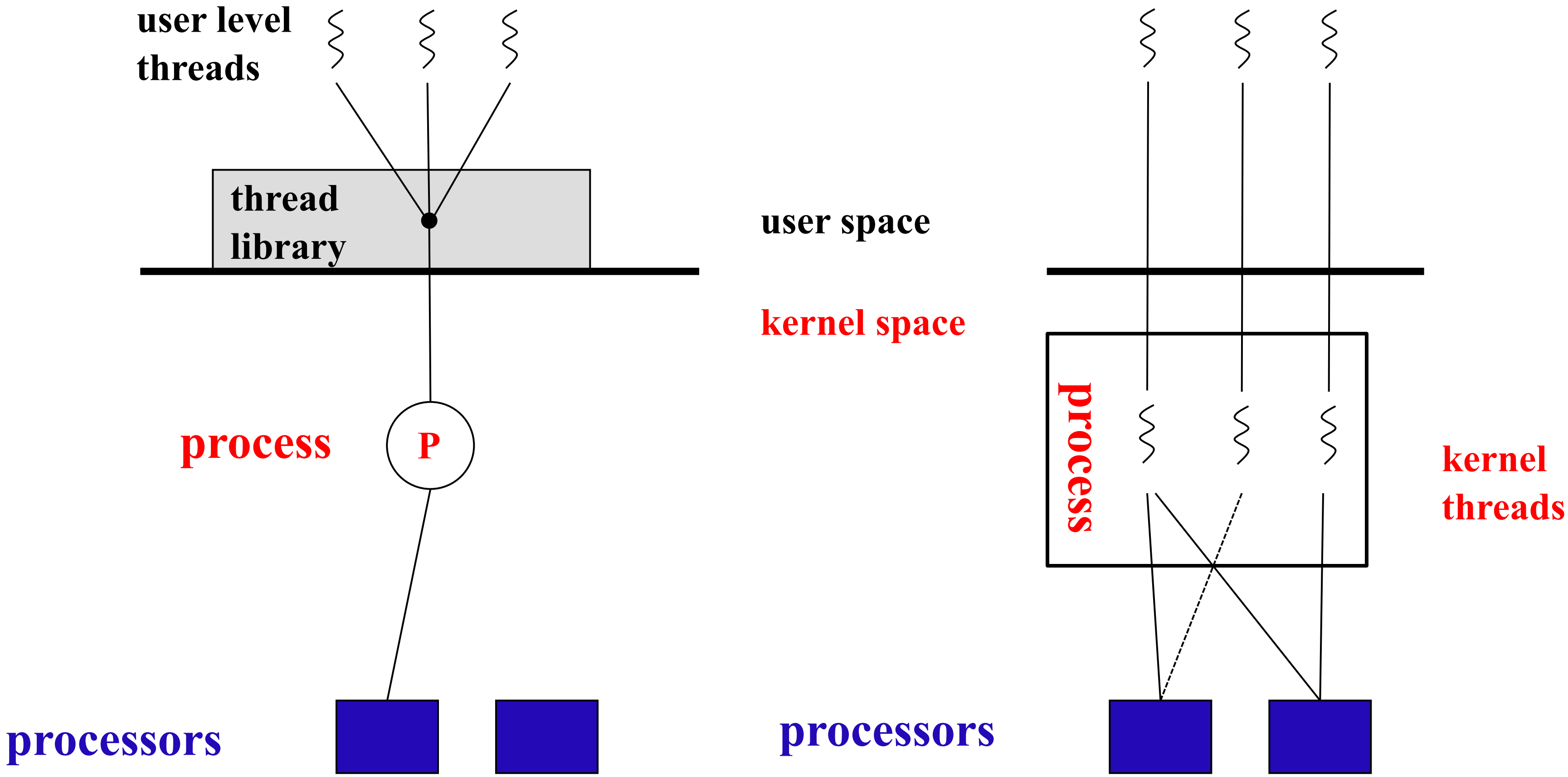
- Kernel-level threads
  - Kernel is aware of the multiple threads in a process
  - Maintains a Thread Control Block for each thread
  - Independently schedules each thread
  - AKA *Lightweight processes*, less expensive than a whole process

Threads Implementation Alternatives

- User-level threads
  - User-space code implements multi-threading
  - Kernel unaware of multi-threading
  - Typically implemented via a reusable library
  - Like a little part of the OS duplicated in user space
    - Time-slices CPU time among threads
    - User-space data structures for thread state

User- vs Kernel-level threads

K–L



Thread Implementation Tradeoffs

- Some advantages of (exclusively) user-level threads
  - Even lower context-switch overhead
  - Lower cost per thread
  - User-space code can make application-specific scheduling choices
  - Reduced consumption of kernel resources.
  - Portability
- Some disadvantages
  - Reduced concurrency, OS will only run the process on one core at a time
  - Maybe reduced share of CPU time, OS just sees one thread

Thread Implementation Tradeoffs

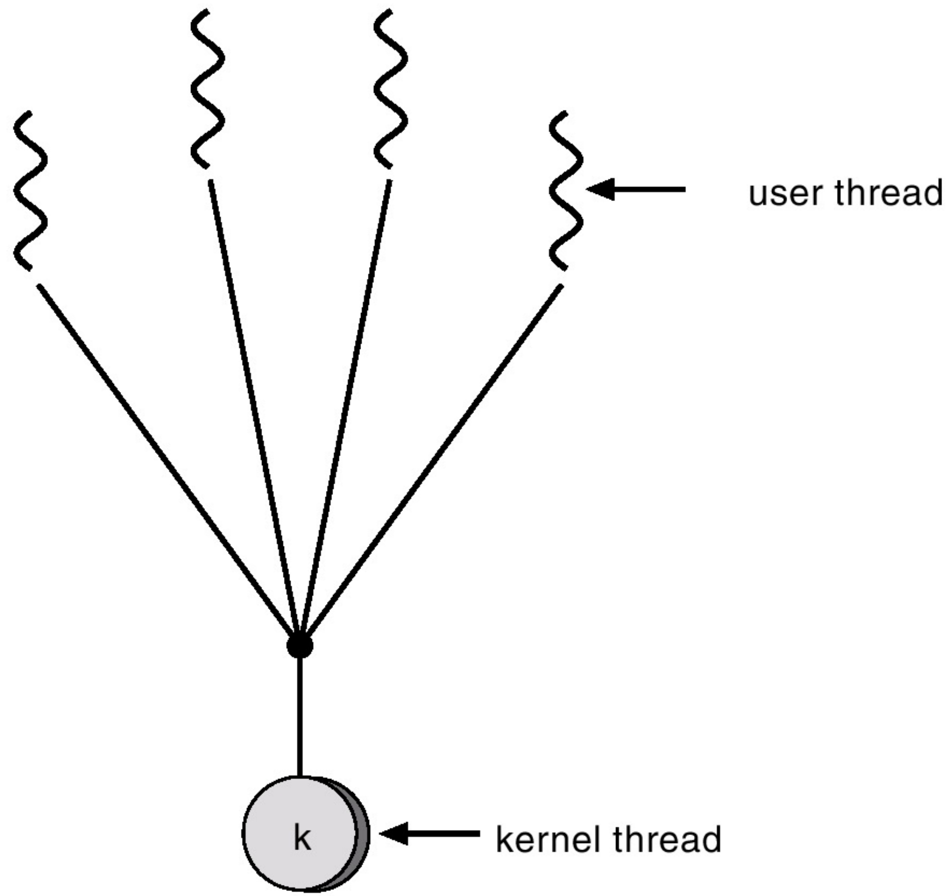
- Blocking behavior, a big disadvantage of user-level threads
  - You can only time-slice CPU time when the OS runs your process
  - What if one thread makes a blocking system call?
  - You guessed it, the OS won't give you any CPU time to subdivide until the call is complete
  - So, when one user-level thread blocks, all user-level threads in that process will be unable to run (even if they didn't actually make the call that blocked)
- The kernel can independently schedule kernel-level threads
  - But, it can't do this for threads it doesn't even know about

Can’t we Have Both?

- We’re programmers, shouldn’t we be able to decide:
  - Which type of threads we use?
  - In general, how kernel-threads are mapped to threads in the application?

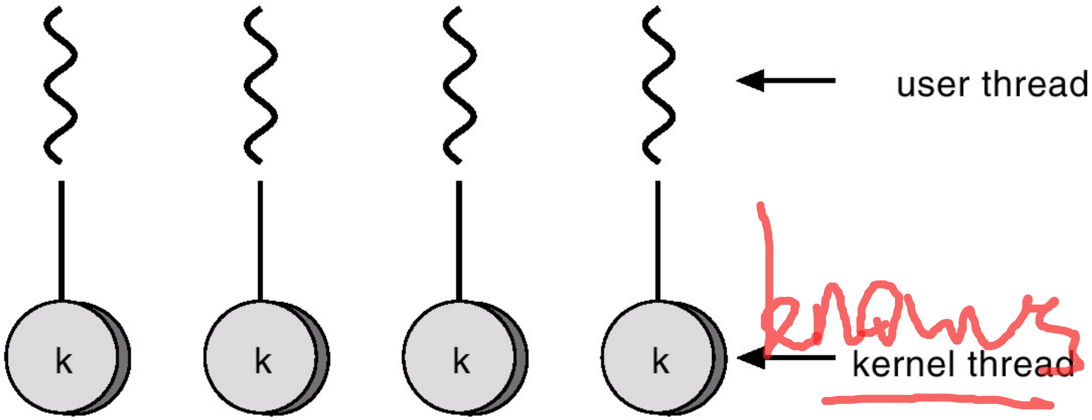
Many-to-One Model

- Many user level threads
- Executed via a single kernel thread
- Examples
  - Solaris Green Threads
  - GNU Portable Threads



One-to-One Model

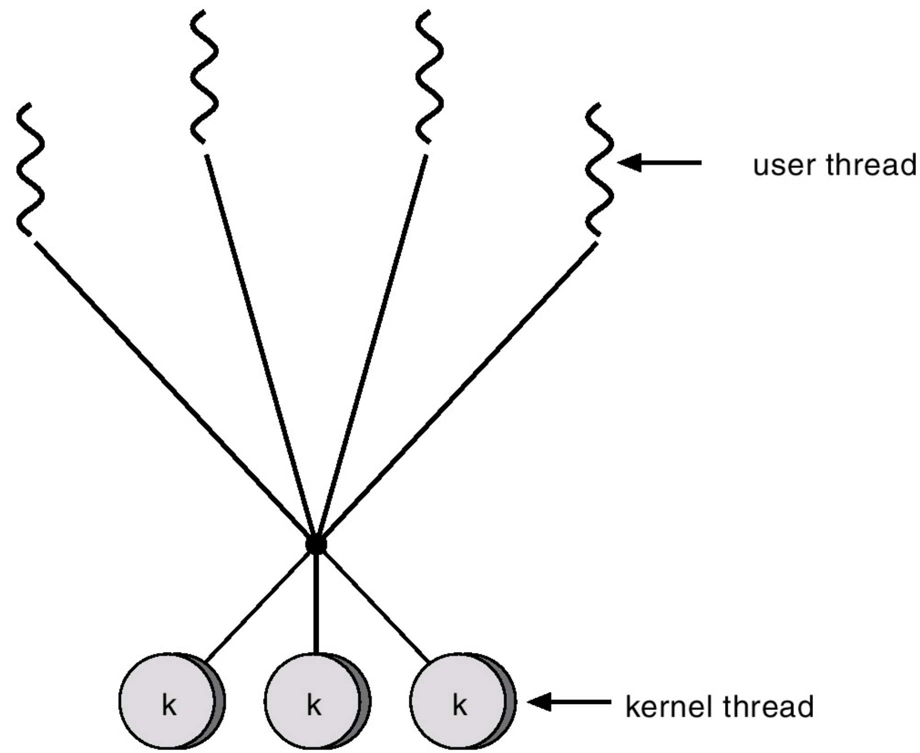
- Each thread maps to a kernel-level thread
- Examples
  - Windows NT/XP/2000/ …
  - Pthreads under Linux
  - Solaris 9 and later



knows every thread

Many-to-Many Model

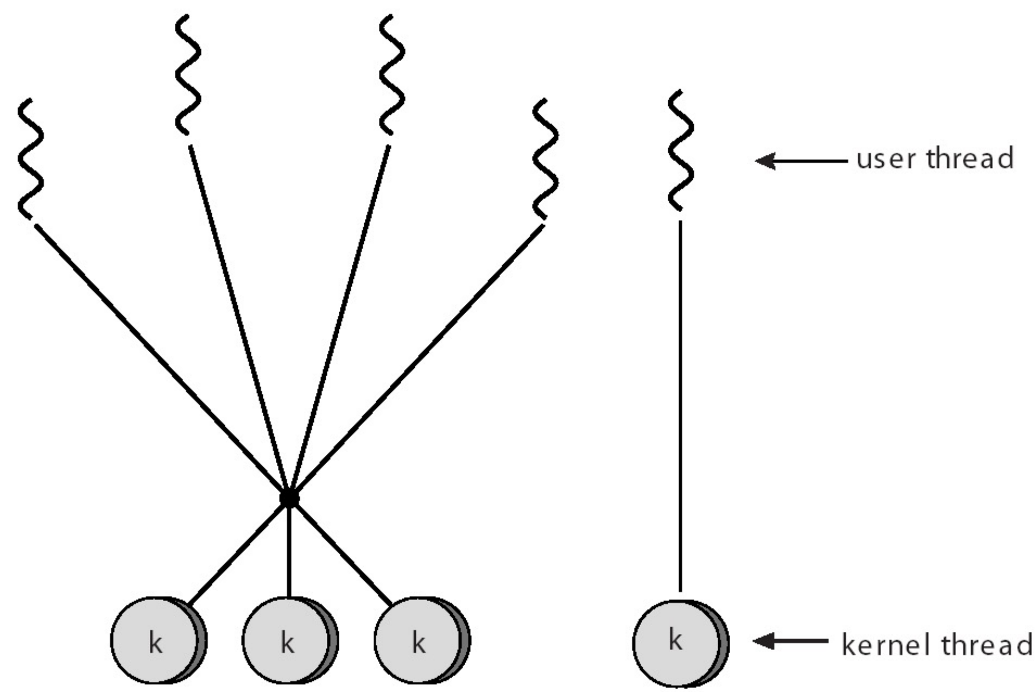
- Independent mapping of threads to kernel threads
- Examples
  - Before Solaris 9
  - Windows *ThreadFiber* package
- Scheduling
  - OS schedules kernel threads on a CPU
  - Process schedules threads on a kernel thread
  - Dispatch decision made in kernel space and in user space, maybe with the help of an upcall.



Two-Level Model

- Many-to-Many and
- … ability to bind some threads to individual kernel threads

- Examples
  - IRIX
  - HP-UX
  - Tru64 Unix
  - Solaris 8 and older



Signal Handling with Multiple Threads

- Claim: new OS features can strain existing mechanisms/abstractions
- POSIX Signals
  - Simple, one process can send a signal to another
  - Vector of signal handler functions in each process
  - System call: void kill(pid_t pid, int sig)
  - This is how the kill command works (surprised?)
  - Signal handler dispatched when a signal received (an upcall)
- Simple for single-threaded processes
- For multi-threaded processes, who gets the signal?

Signal Handling with Multiple Threads

- Different signals should be handled differently
  - Maybe some should go to all threads (e.g., SIGINT)
  - Maybe some should go to the thread that caused it (e.g., SIGFPE)
  - Maybe some should go to a particular thread (e.g., SIGIO)
- A “Solution”
  - Individual threads can block particular signals
  - Deliver the signal to a thread that hasn’t blocked it.
- More examples of stress on existing mechanisms
  - What happens to threads when we call fork()?
  - What happens to threads when we call exec()?

Thread Cancellation

- One thread may want to terminate other threads
  - Say we have a parallel search, and a different thread already found a solution
- Two ways to deal with this
  - *Asynchronous cancellation*
    - Something like kill() to terminate a thread
    - Makes sense for processes, more problematic with threads, why?
  - *Deferred cancellation*
    - Define / create *cancellation* points, where the thread can terminate.
    - Maybe, have target thread check to see if it should terminate
  - Deferred cancellation is generally the way to go

What we Learned

- Definition of a Thread and how they work
- Pthread APIs for using threads
- Alternatives in how threads are provided to the process
- Special considerations in writing multi-threaded programs, fork(), exec(), signals and cancellation