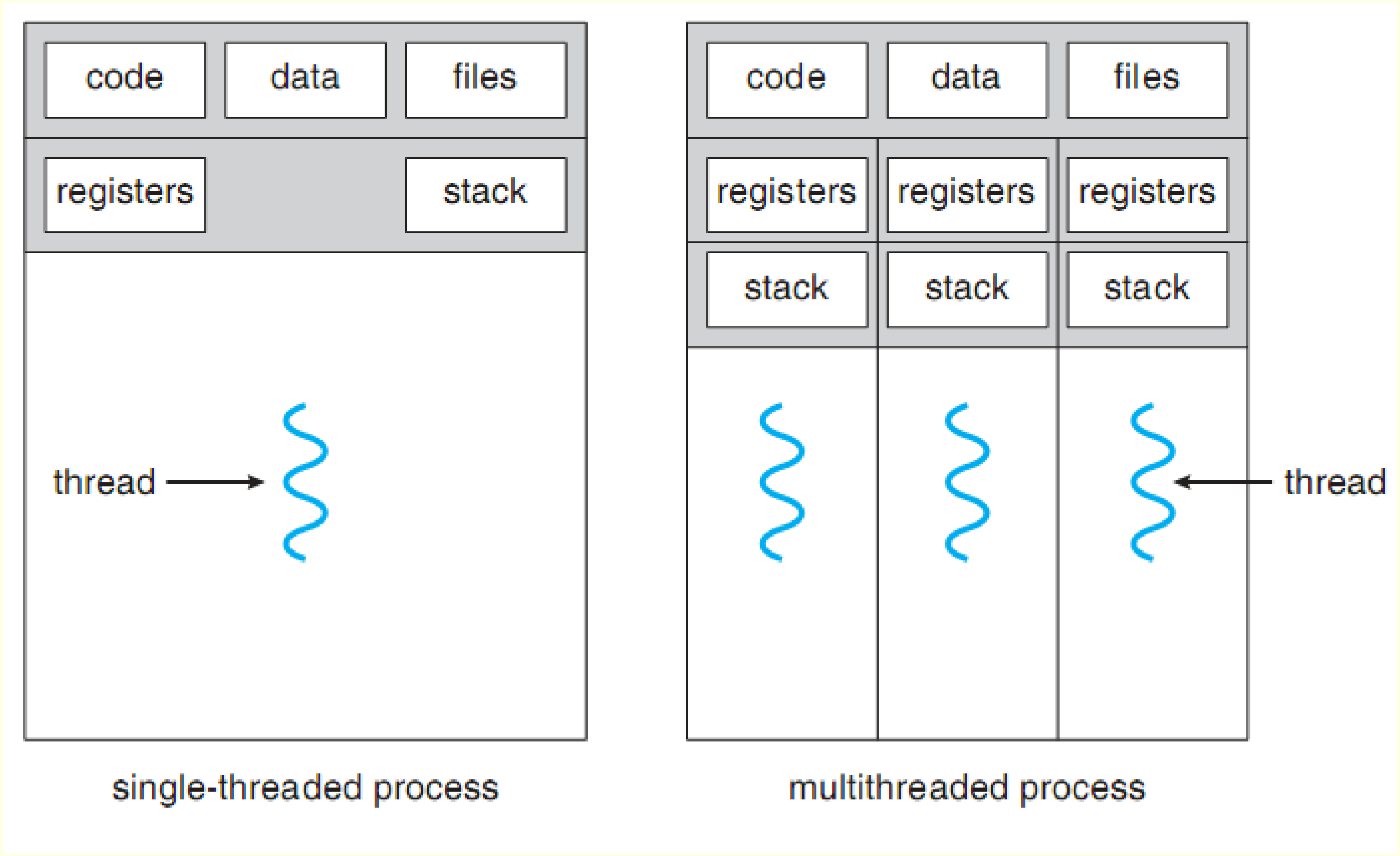
Threads

Mehdi Kargahi
School of ECE
University of Tehran
Summer 2016

Definition

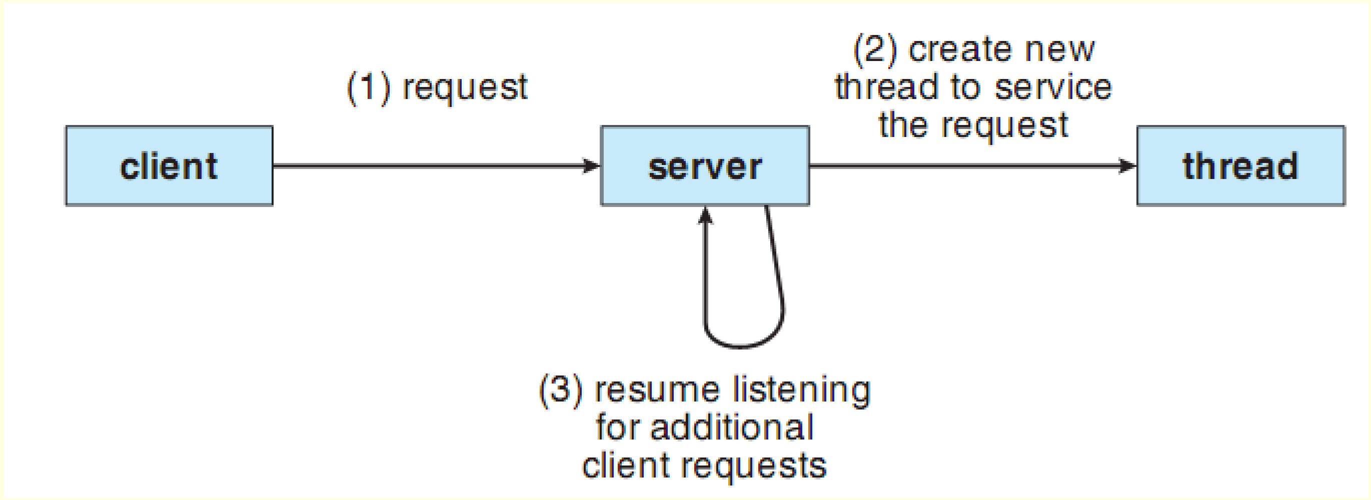
- What's the difference between a thread and a process?
- Heavyweight process: A single thread process
- What a thread has for itself?
  - Thread-Specific Data
    - Thread ID
    - Program Counter
    - A register set
    - A stack

Single-threaded and Multithreaded Processes



Examples

- A Web server
- A RPC server
- Since the requests from these servers are similar, it is better to avoid creating new processes that may be much more costly


client
(1) request
server
(2) create new thread to service the request
thread
(3) resume listening for additional client requests

The Benefits of Multithreaded Programming

- Responsiveness
  - If a thread is blocked, other threads can continue
  - E.g., showing an image while getting text from the user in a web page
- Resource sharing
  - Threads share the memory and resources of the respective process (shared code and data)
- Economy
  - Allocating memory and resource to processes and their respective context-switch is more costly with respect to threads
  - Solaris: speed of process creation=1/30 speed of thread creation
  - Solaris: speed of process CS=1/5 speed of thread CS
- Utilization of multiprocessor architectures
  - No speed-up with a heavyweight process on a multiprocessor system
  - Concurrency with multithreaded processes on multiprocessor systems

Multicore Programming

- Amdahl's Law
  - S: The portion of the application that must be performed serially
  - N: the number of processing cores
    - Suppose that N goes to infinity!

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

Challenges in Programming for Multicore Systems

- **Identifying tasks**: Finding areas that can be divided into separate, concurrent tasks.
- **Balance**: Programmers must ensure that the tasks perform equal work of equal value to assign a core for which.
- **Data splitting**: The data accessed and manipulated by the tasks must also be divided to run on separate cores.
- **Data dependency**: When one task depends on data from another, programmers must ensure that the execution of the tasks is synchronized to accommodate the data dependency.
- **Testing and debugging**: When a program is running in parallel on multiple cores, many different execution paths are possible. Testing and debugging such concurrent programs is inherently more difficult.

Types of Parallelism

- **Data Parallelism**: Distributing subsets of the same data across multiple computing cores and performing the same operation on each core.
  - Example: Summing the contents of an array of size N.
- **Task Parallelism**: Distributing not data but tasks (threads) across multiple computing cores. Each thread is performing a unique operation.

In most instances, applications use a hybrid of these two strategies.

Multithreading Models

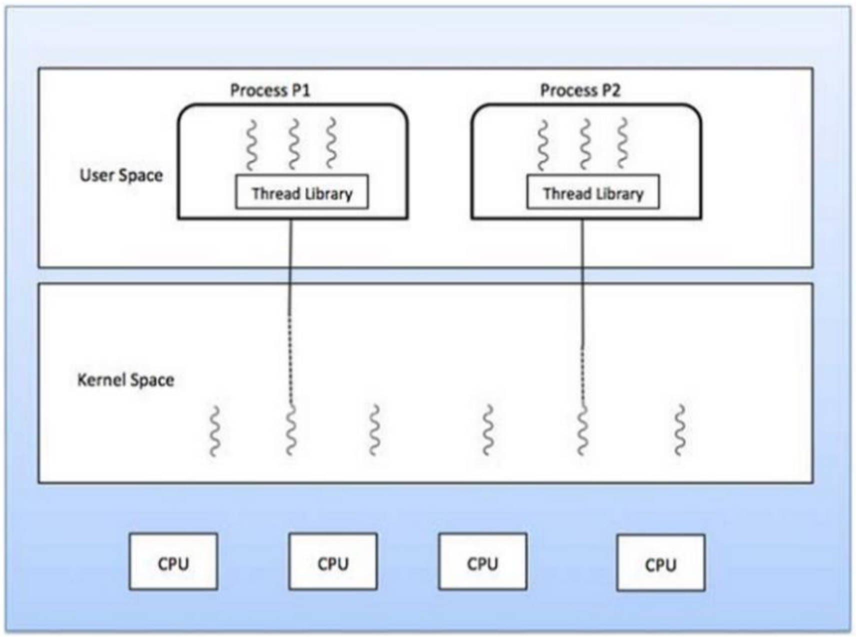
- *User threads* are supported above the kernel
- *Kernel threads* are supported and managed by the OS

Thread Libraries

- Pthreads (Book::P.173)
- Win32 threads
- Java threads

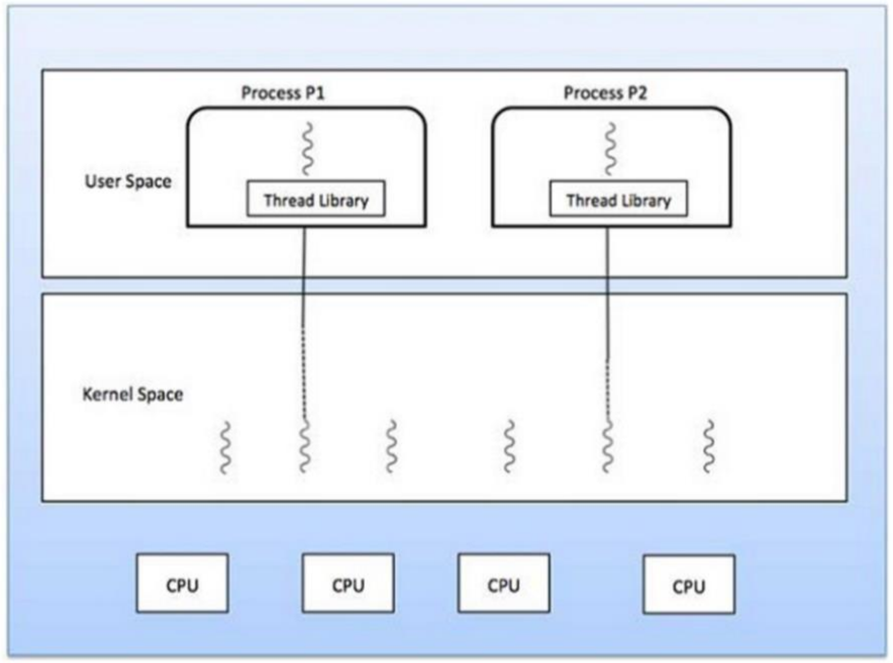
Many-to-One Model

- Benefit
  - Since thread management is in user space, it is efficient
- Disadvantage
  - A blocking system call will block the entire process
  - No benefit of using a multiprocessor system
- Examples
  - Green threads in Solaris
  - GNU Portable threads



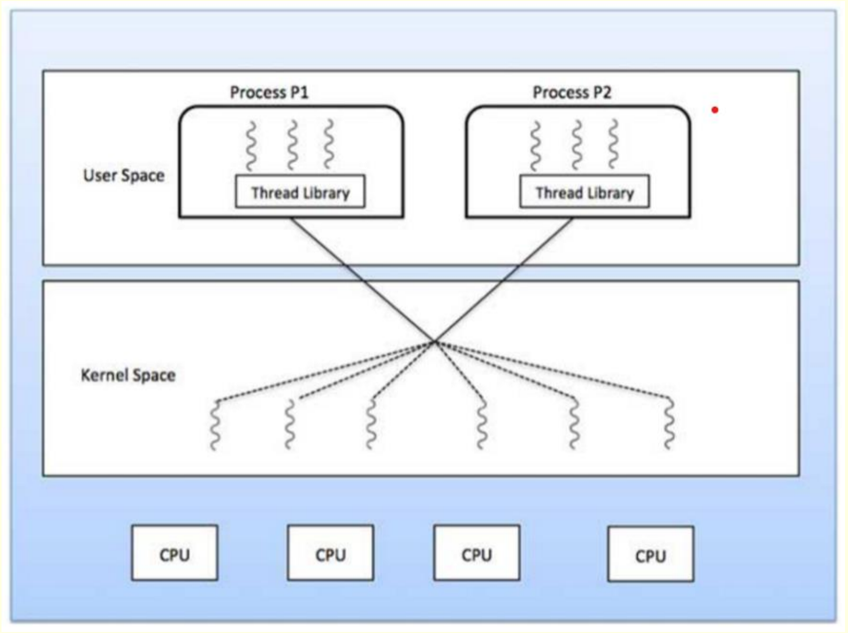
One-to-One Model

- Benefits
  - A blocking system call will not block the entire process
  - Multiple threads can run concurrently on a MP system
- Disadvantage
  - Creating a user thread requires creating a corresponding kernel thread which is costly and usually restricts the number of threads supported by the system
- Examples
  - Linux and the family of Windows

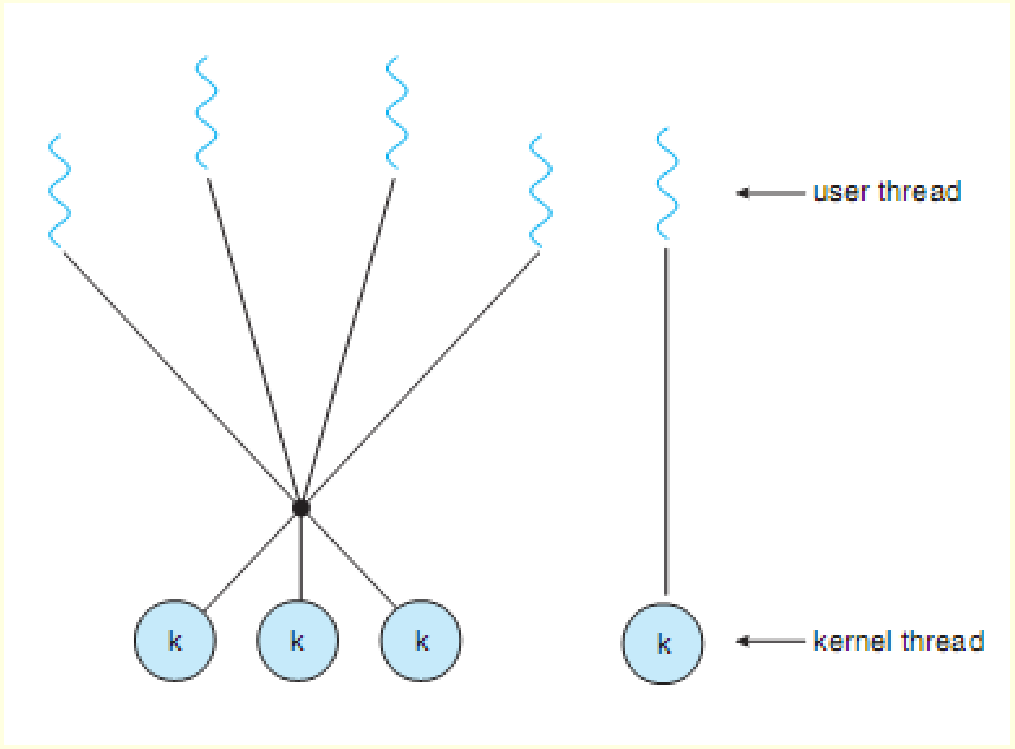


Many-to-Many Model

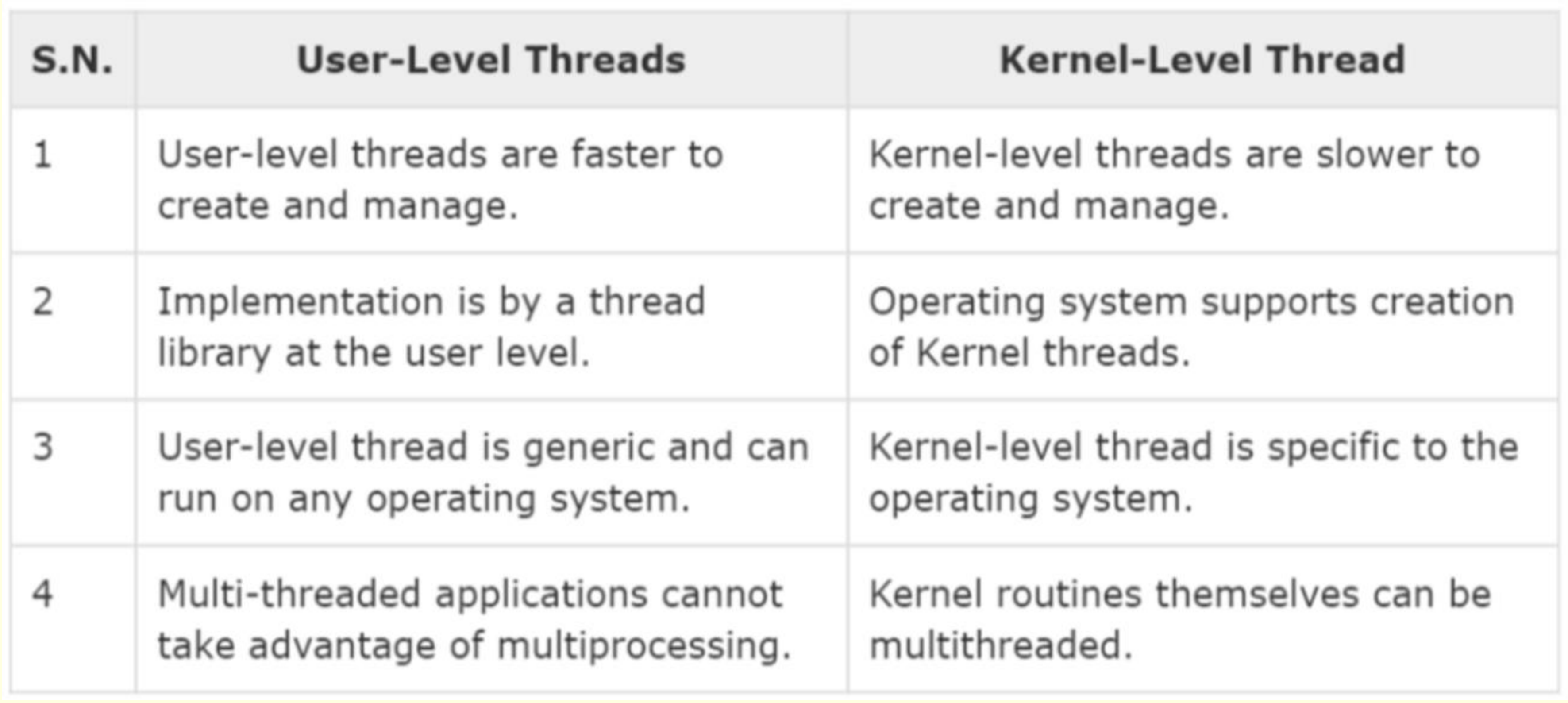
- Multiplexing (Tru64 UNIX)
  - No. of kernel threads ≤ no. of user threads
- Properties
  - As many user threads as necessary can be created
  - The corresponding kernel threads can run in parallel on a multiprocessor
  - When a thread performs a blocking system call, the kernel can schedule another thread



Two-Level Model



Differences Between User-Level and Kernel-Level Threads



S.N.	User-Level Threads	Kernel-Level Thread
1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

Implicit Threading

- To transfer the creation and management of threading from application developers to compilers and run-time libraries.
- Thread Pools
- OpenMP
- GCD: Grand Central Dispatch
- …

Threading Issues

- The fork() and exec() system calls
  - After calling fork(), does the new process duplicates all threads, or is the new process single-threaded?
    - Two variation
  - What about exec()?
  - Which variation should be used if exec() is called immediately after forking?
    - Duplicating is not necessary!

Thread Cancellation

- Asynchronous cancellation
  - One thread immediately cancels the target thread
  - *Problem*: if resources have been allocated to a cancelled thread or is updating some shared data
- Deferred cancellation
  - The target thread periodically checks whether it should terminate
  - Safe cancellation in *cancellation points*

Signal Handling

- Synchronous signal
  - Delivered to the same process that performed the operation that caused the signal (division by 0)
- Asynchronous signal
  - Generated by an external event (e.g., a timer expiration)
- Signal handlers
  - *Default signal handler* is run by the kernel
  - *User-defined signal handler* overrides default handler

Options for Delivering a Signal

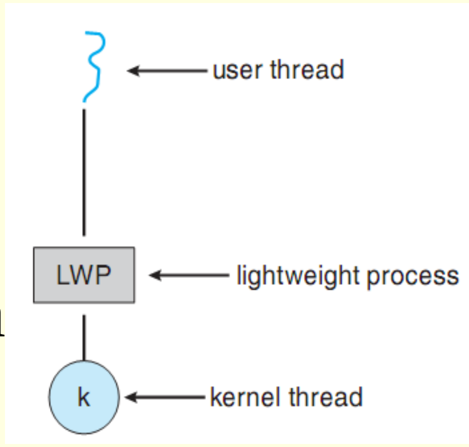
- To the thread to which the signal applies
- To every thread in the process
- To certain threads in the process
- Assign a specific thread to receive all signals to the process

Thread Pools

- Creating a number of threads at process startup and place them into a pool
- Benefits
  - Faster than waiting for creating a thread
  - Limiting the number of threads of each process

Scheduler Activation

- For communication between the kernel and the thread library
  - Many-to-many and two-level models
- LWP: Lightweight Process
  - An intermediate data structure between the user and kernel threads
  - Similar to a virtual processor for the user-thread library
  - Each LWP is attached to a kernel-level thread
  - If the kernel thread blocks, the LWP blocks as well



Number of LWPs

- Assume a CPU-bound application running on a single processor
  - Only one thread can run at once, so one LWP is sufficient
- Assume an IO-bound application
  - One LWP is required for each concurrent blocking system call

How Scheduler Activation Works?

- Kernel informs an application about certain events through *upcalls*
- Upcalls are handled by the thread library with the respective *upcall handler* which is run on a virtual processor
- When a thread is about to block, the kernel makes an upcall and allocates a new virtual processor to the application
- The application runs an upcall handler on this new virtual processor that saves the state of the blocking thread
- The upcall handler schedules another thread on the new virtual processor