Operating Systems

8. CPU: Thread

Hyunchan, Park

http://oslab.chonbuk.ac.kr

Division of Computer Science and Engineering

Chonbuk National University

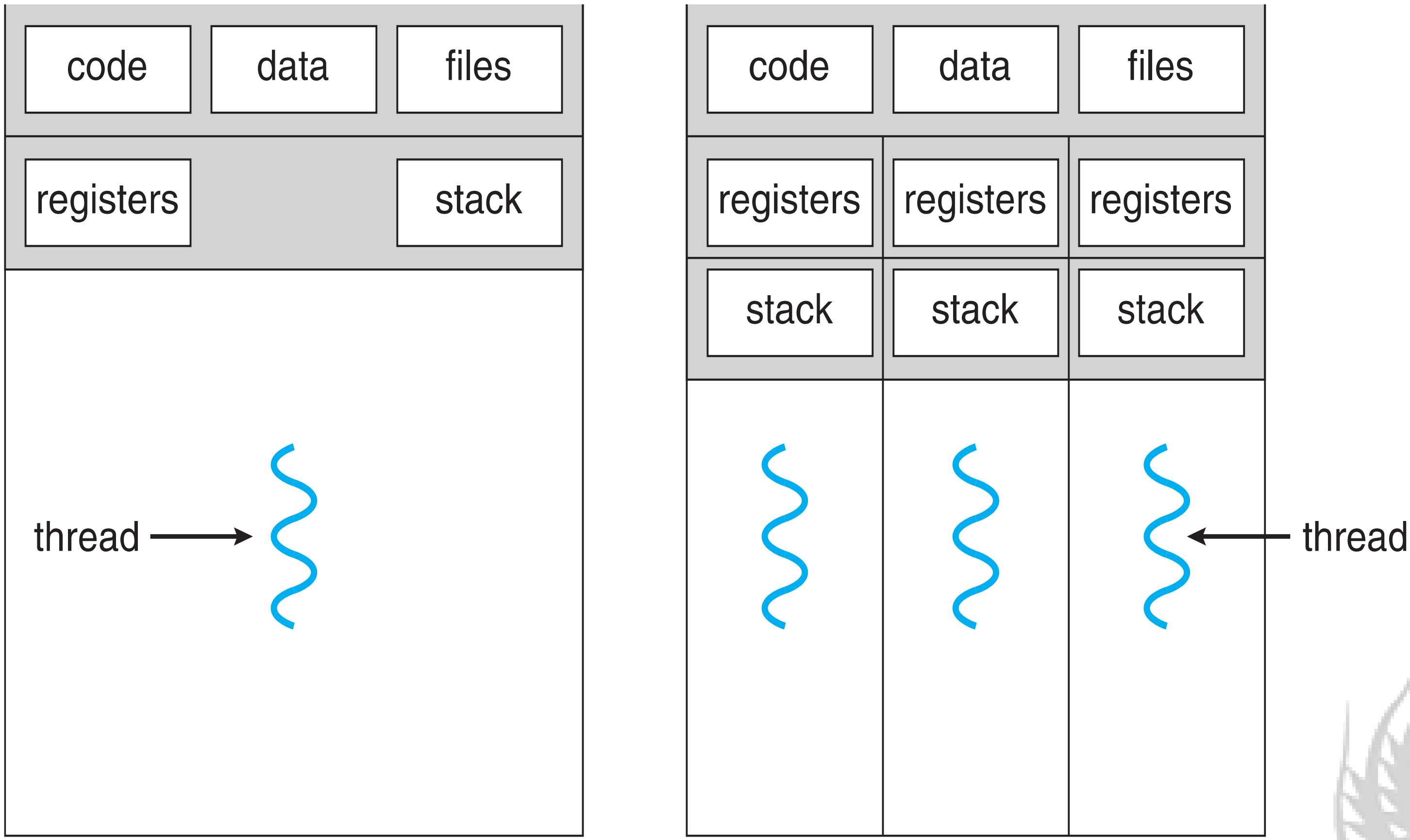
Contents

- Thread
- User and Kernel threads: Multithreading Models
  - Many-to-One
  - One-to-One
  - Many-to-Many
- Issues with threads
  - Creation (fork and exec system calls)
  - Cancellation
  - Thread pools
  - IPC between threads
  - Signal handling
- Implementation: Linux thread

Thread

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - Multiple locations can execute at once
    - Multiple threads of control -> threads
  - Must then have storage for thread details, multiple program counters in PCB
- Thread
  - The execution unit in a process
  - Program counter, register set, stack
    - Code, data, and opened files are shared among threads in a process

Single and Multithreaded Processes



single-threaded process

multithreaded process

Processes vs. Threads

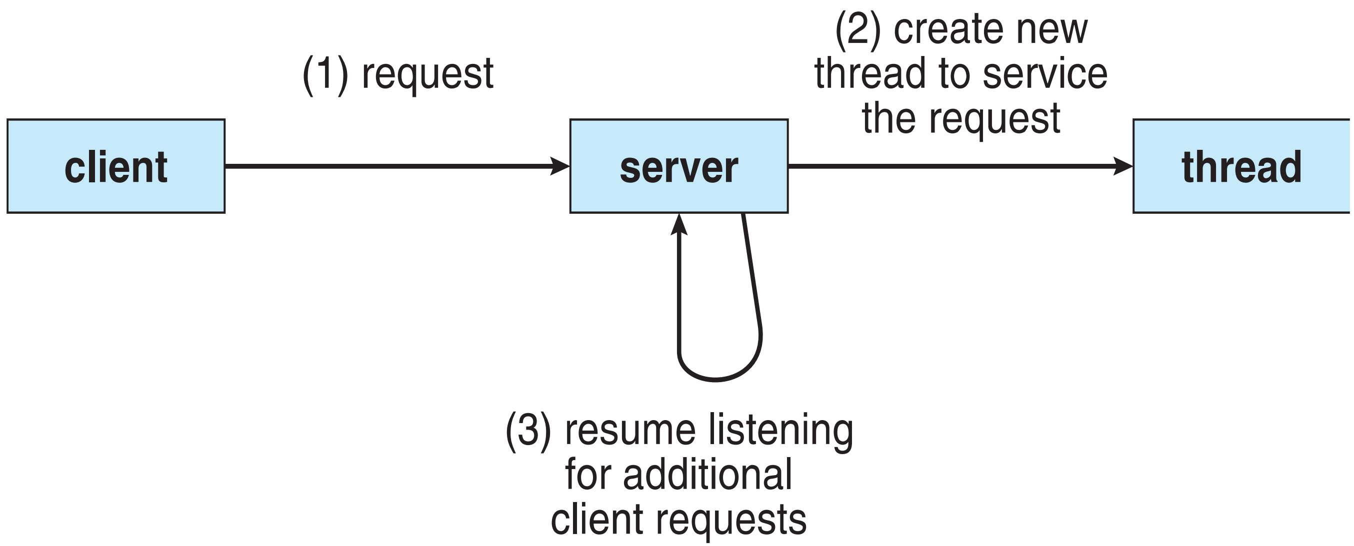
- Processes
  - Protection domain between processes
    - Cannot directly access the other's memory $\Rightarrow$ must use IPC
  - Heavy weight: more operations are needed for context switch

- Threads
  - Code and data sections are shared among threads in a process
  - Light weight: more efficient switching between threads in a process
    - Because they share the same address space
    - No TLB and cache flush (will be explained later)

Motivation

- Most modern applications are multithreaded
- 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



Benefits

- Responsiveness
  - may allow continued execution if part of process is blocked, especially important for user interfaces
- Resource Sharing
  - threads share resources of process, easier than shared memory or message passing
- Performance
  - cheaper than process creation, thread switching lower overhead than context switching
- Scalability
  - process can take advantage of multiprocessor architectures

Amdahl's Law

- Identifies performance gains from adding additional cores to an applic ation that has both serial and parallel components
- S is serial portion
- N processing cores

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

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 c ores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches 1 / S
- Serial portion of an application has disproportionate effect on perfor mance gained by adding additional cores
- But does the law take into account contemporary multicore systems?

User Threads and Kernel Threads

- User threads
  - Management done by user-level threads library
  - Three primary thread libraries:
    - POSIX Pthreads, Windows threads, Java threads
- Pros
  - High performance: only maintains user-mode context
- Cons
  - If a thread executes a system call, entire threads are blocked by kernel
    - Because they are just a sole process for the kernel: kernel doesn’t know about user level threads

User Threads and Kernel Threads

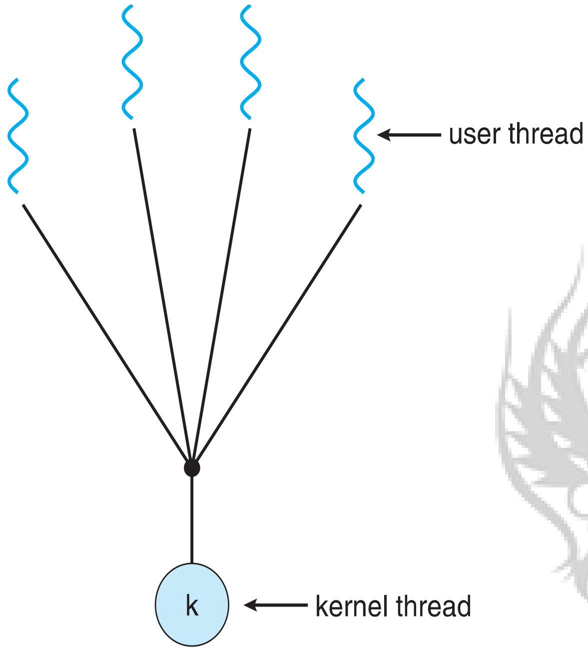
- Kernel threads
  - Management done by the Kernel
  - Examples – virtually all general purpose operating systems, including:
    - Windows, Solaris, Linux, Tru64 UNIX, and Mac OS X
- Pros
  - Support high level of parallelism
    - Each of threads use a system call exclusively; does not blocked by other thread's system call
- Cons
  - High cost on thread creation/management
    - Should maintains kernel level context and metadata
      - E.g. have to enlarge the PCB for the threads

User to Kernel: Multithreading Models

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

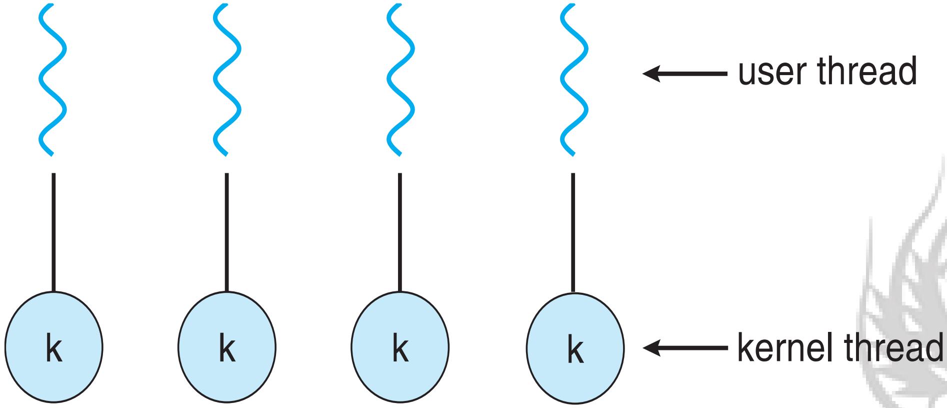
Many-to-One

- Many user-level threads mapped to single kernel thread
- Cons: One thread blocking causes all to block
  - Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
  - Who does not support kernel threads
  - Examples:
    - Solaris Green Threads
    - GNU Portable Threads



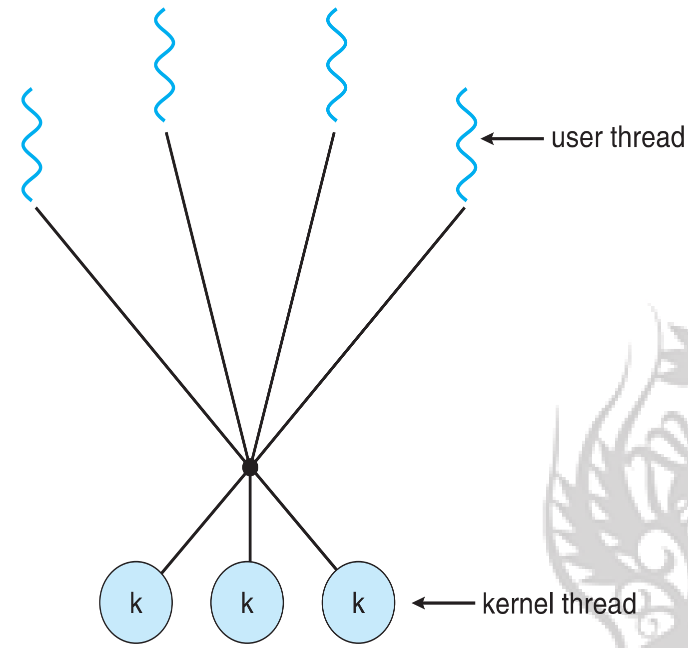
One-to-One

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- Pros: More concurrency than many-to-one
- Cons: Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



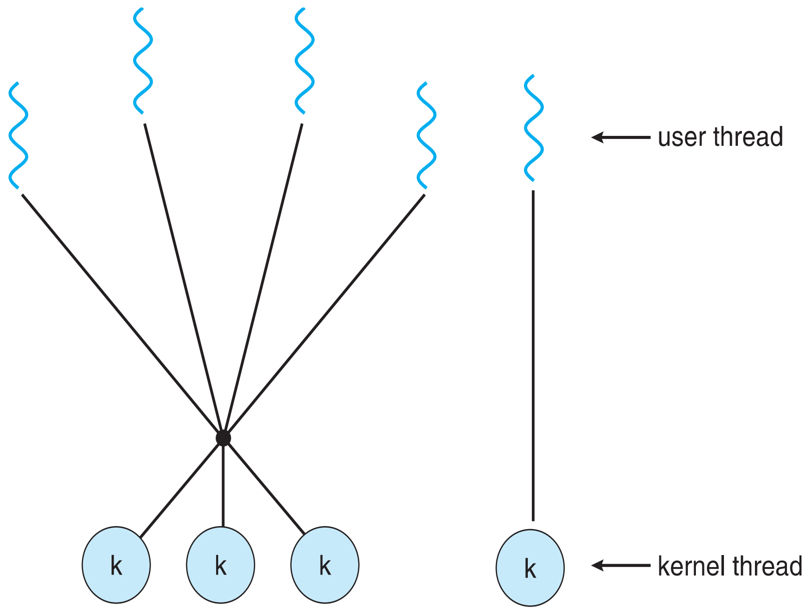
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Pros: Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows 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



Threading Issues

- Semantics of fork() and exec() system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread pools
- IPC between threads
- Signal handling
  - Synchronous and asynchronous

Semantics of fork() and exec()

- Does fork() duplicate only the calling thread or all threads?
  - Some UNIXes have two versions of fork
- exec() usually works as normal
  - Replace the running process including all threads

  - Is it a right policy?
    - How about that the only thread who execute fork() should be replaced by exec()?

Thread Cancellation

- Terminating a thread before it has finished
  - Thread to be canceled is target thread
- Two general approaches:
  - Asynchronous cancellation terminates the target thread immediately
  - Deferred cancellation allows the target thread to periodically check if it s hould be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches cancellation point
    - I.e. pthread_testcancel()
    - Then cleanup handler is invoked
- On Linux systems, thread cancellation is handled through signals

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
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e. Tasks could be scheduled to run periodically

IPC between threads

- Shared memory is most appropriate
  - Because threads share a same address space
  - High performance, but synchronization is required
- Threads naturally decrease the requirements for IPCs
- The IPC between threads in different processes?
  - No differences with IPC between processes
  - Originated in POOR program design

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 by one of two signal handlers:
    - default
    - user-defined
- Every signal has default handler that kernel runs when handling signal
  - User-defined signal handler can override default
  - For single-threaded, signal delivered to process

Signal Handling (Cont.)

- Where should a signal be delivered for multi-threaded?
  - 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

Implementation

- Thread library
- POSIX Pthread
  - IEEE 1003.1c: pthread_create()
- Windows threads API
  - Via system call: CreateThread()
- Linux threads
  - Introduced in version 2.2
  - Via system call: clone()
- Java threads
  - Thread class

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)
  - Flags control behavior

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.

- struct task_struct points to process data structures (shared or unique)

Multicore Programming

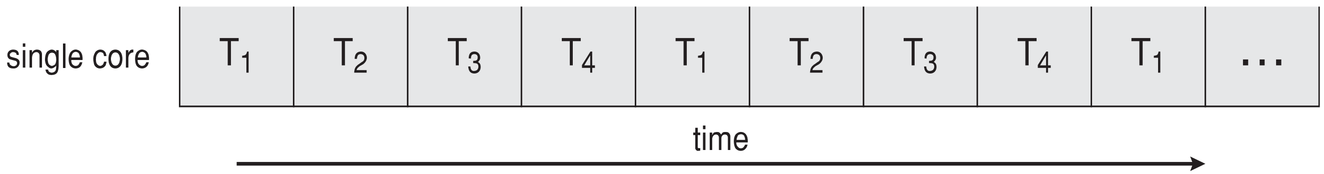
- Multicore or multiprocessor systems putting pressure on programmers, challenges include:
  - Dividing activities
  - Balance
  - Data splitting
  - Data dependency
  - Testing and debugging
- Parallelism implies a system can perform more than one task simultaneously
- Concurrency supports more than one task making progress
  - Single processor / core, scheduler providing concurrency

Multicore Programming (Cont.)

- Types of parallelism
  - Data parallelism – distributes subsets of the same data across multiple cores, same operation on each
  - Task parallelism – distributing threads across cores, each thread performing unique operation
- As # of threads grows, so does architectural support for threading
  - CPUs have cores as well as hardware threads
  - Consider Oracle SPARC T4 with 8 cores, and 8 hardware threads per core

Concurrency vs. Parallelism

- Concurrent execution on single-core system:



- Parallelism on a multi-core system: