Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Fall 2023

# Threads

# Concurrency

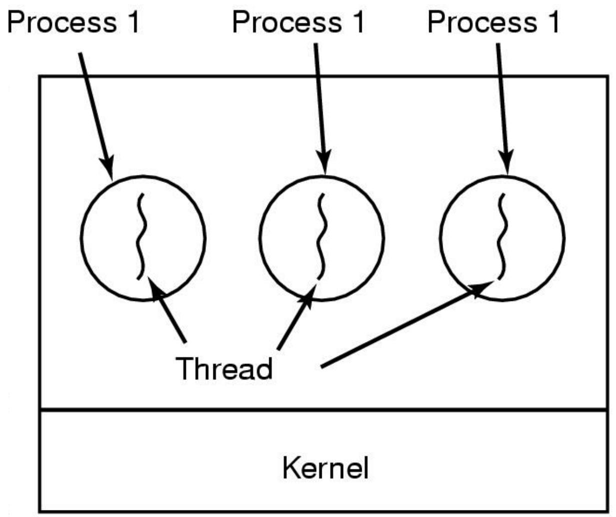
- Virtualization
  - Virtual CPUs
  - Virtual memory
- Concurrency
  - In the user space by running multi-threaded programs
  - In the kernel space too!
- OS Issues
  - How to support multi-threaded programs?
  - How to coordinate accesses to shared resources?

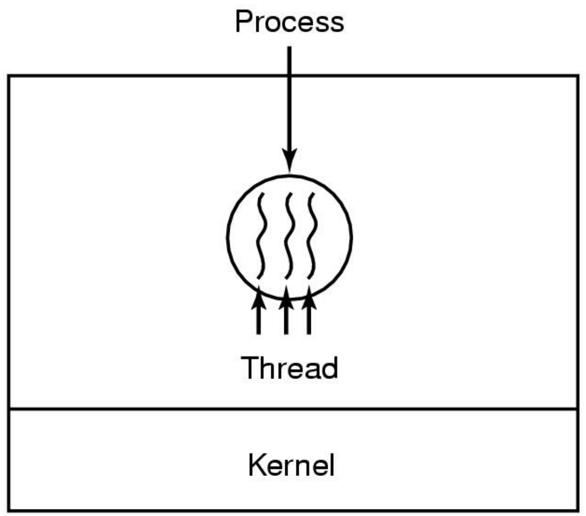
# Motivation

- Process is a cool abstraction to run a new program
  - OS provides protection and isolation among processes
- But, …
  - A single process cannot benefit from multi-cores
  - Very cumbersome to write a program with many cooperating processes
  - Expensive to create a new process
  - High communication overheads between processes
  - Expensive context switching between processes
- How can we increase concurrency within a process cheaply?

# What is a Thread?

- A thread of control:
  a sequence of instructions being executed in a program
- A thread has its own
  - Thread ID
  - Set of registers including PC & SP
  - Stack
- Threads share an address space
- Separate the concept of a process from its execution state



# Using Threads

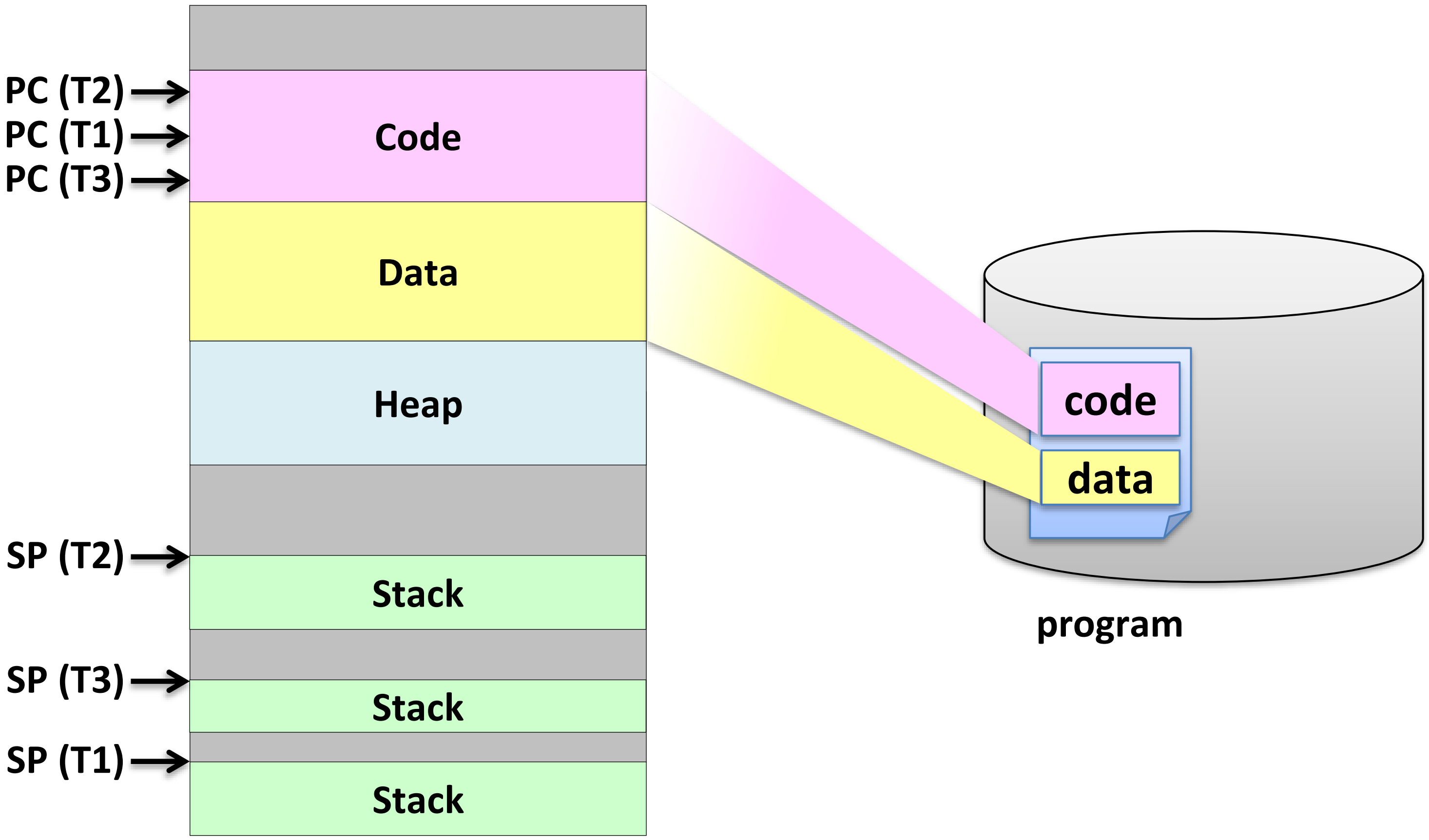
```
#include <stdio.h>
#include <pthread.h>

void *hello(void *arg) {
     printf(“hello, world\n”);
     ...
}

int main() {
     pthread_t tid;

     pthread_create(&tid, NULL, hello, NULL);
     printf(“hello from main thread\n”);
     ...
}
```

# Address Space with Threads



# Processes vs. Threads

- A thread is bound to a single process
- A process, however, can have multiple threads
- Sharing data among threads is cheap; all see the same address space
- Thread is a unit of scheduling
- Processes are containers in which threads execute
  - PID, address space, user and group ID, open file descriptors, current working directory, etc.
- Processes are static, while threads are dynamic entities

Image source: https://dribbble.com/shots/1395795-factory-cross-section-progress-4

# Benefits of Multi-threading

- Creating concurrency is cheap
- Improves program structure
  - Divide large task across several cooperative threads
- Throughput
  - By overlapping computation with I/O operations
- Responsiveness
  - Can handle concurrent events (e.g., web servers)
- Resource sharing
- Utilization of multi-core architectures
  - Allows building parallel programs

# Threads Interface

- Pthreads (POSIX Threads)
  - A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
  - API specifies the behavior of the thread library
  - Implementation is up to the development of the library
  - Common in Unix-like operating systems:
    e.g., Linux, Mac OS X, Solaris, FreeBSD, NetBSD, OpenBSD, etc.

- Microsoft Windows has its own Thread API
  - Win32/Win64 threads

# Pthreads: Thread Creation / Termination

```
int pthread_create (pthread_t *tid,
                    pthread_attr_t *attr,
                    void *(start_routine)(void *),
                    void *arg);
```

```
void pthread_exit  (void *retval);
```

```
int pthread_join   (pthread_t tid,
                    void **thread_return);
```

# Pthreads: Mutexes

```
int pthread_mutex_init
              (pthread_mutex_t *mutex,
              const pthread_mutexattr_t *mattr);
```

```
void pthread_mutex_destroy
              (pthread_mutex_t *mutex);
```

```
void pthread_mutex_lock
              (pthread_mutex_t *mutex);
```

```
void pthread_mutex_unlock
              (pthread_mutex_t *mutex);
```

# Pthreads: Condition Variables

```
int pthread_cond_init
             (pthread_cond_t *cond,
             const pthread_condattr_t *cattr);
```

```
void pthread_cond_destroy
             (pthread_cond_t *cond);
```

```
void pthread_cond_wait
             (pthread_cond_t *cond,
             pthread_mutex_t *mutex);
```

```
void pthread_cond_signal
             (pthread_cond_t *cond);
```

```
void pthread_cond_broadcast
             (pthread_cond_t *cond);
```

# Threading Issue: `fork()` / `exec()`

- When a thread calls `fork()`,
  - Does the new process duplicate all the threads?
  - Is the new process single-threaded?

- In Pthreads, `fork()` duplicates only a calling thread

- In the Unix international standard,
  - `fork()` duplicates all parent threads in the child
  - `fork1()` duplicates only a calling thread

- Normally, `exec()` replaces the entire process

# Threading Issue: Thread Cancellation

- The task of terminating a thread before it has completed
- Asynchronous cancellation
  - Terminates the target thread immediately
  - What happens if the target thread is holding a resource, or it is in the middle of updating shared resources?
- Deferred cancellation
  - The target thread is terminated at the cancellation points
  - The target thread periodically check if it should be cancelled
- Pthreads API supports both asynchronous and deferred cancellation

# Threading Issue: Signal Handling

- Where should a signal be delivered?
- To the thread to which the signal applies
  - For synchronous signals
- To every thread in the process
- To a dedicated thread
  - Solaris 2: Assign a specific thread to receive all signals for the process
- To certain threads in the process
  - Typically, only to a single thread found in a process that is not blocking the signal
  - Pthreads: per-process pending signals, per-thread blocked signal mask

# Threading Issue: Libraries

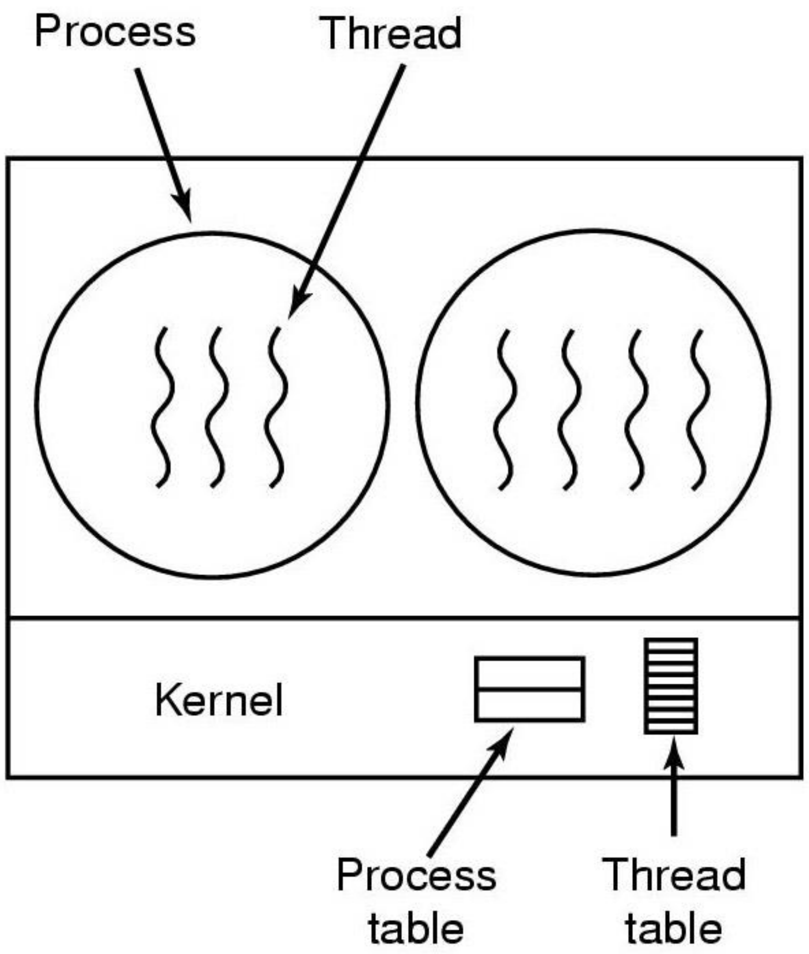
- `errno`
  - Each thread should have its own independent version of the `errno` variable

- Multithread-safe (MT-safe)
  - A set of functions is said to be MT-safe , when the functions may be called by more than one thread at a time without requiring any other action on the caller's part
  - Pure functions that access no global data or access only read-only global data are trivially MT-safe
  - Functions that modify global state must be made MT-safe by synchronizing access to the shared data

# Implementing Threads

# Kernel-level Threads

- OS-managed threads
  - OS manages threads and processes
  - All thread operations are implemented in the kernel
  - Thread creation and management requires system calls
  - OS schedules all the threads
  - Creating threads are cheaper than creating processes
  - Windows, Linux, Solaris, Mac OS X, AIX, HP-UX, …

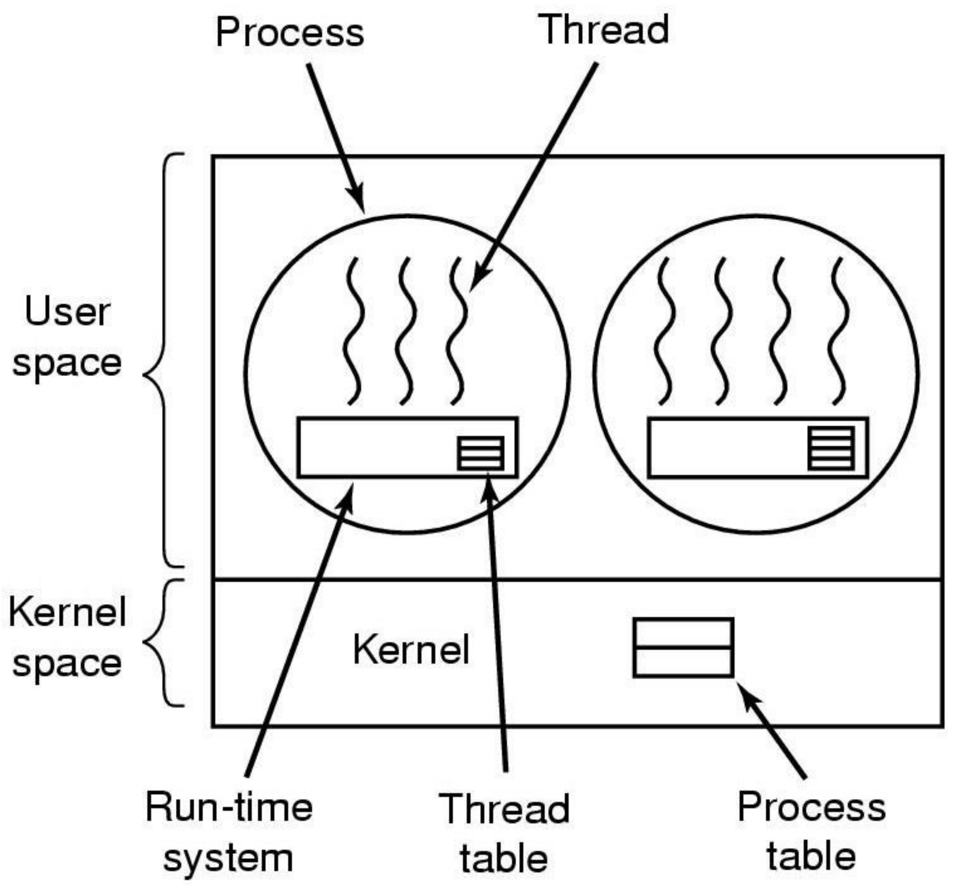


# Kernel-level Threads: Limitations

- They can still be too expensive
- Thread operations are all system calls
- Must maintain kernel state for each thread
  - Can place limit on the number of simultaneous threads
- OS must scale well with increasing number of threads
- Kernel-level threads have to be general to support the needs of all programmers, languages, runtime systems, etc.

# User-level Threads

- Threads are implemented at the user level
  - A library linked into the program manages the threads
  - Threads are invisible to the OS
  - All the thread operations are done via procedure calls (no kernel involvement)
  - Small and fast: 10-100x faster than kernel-level threads
  - Portable
  - Tunable to meet application needs

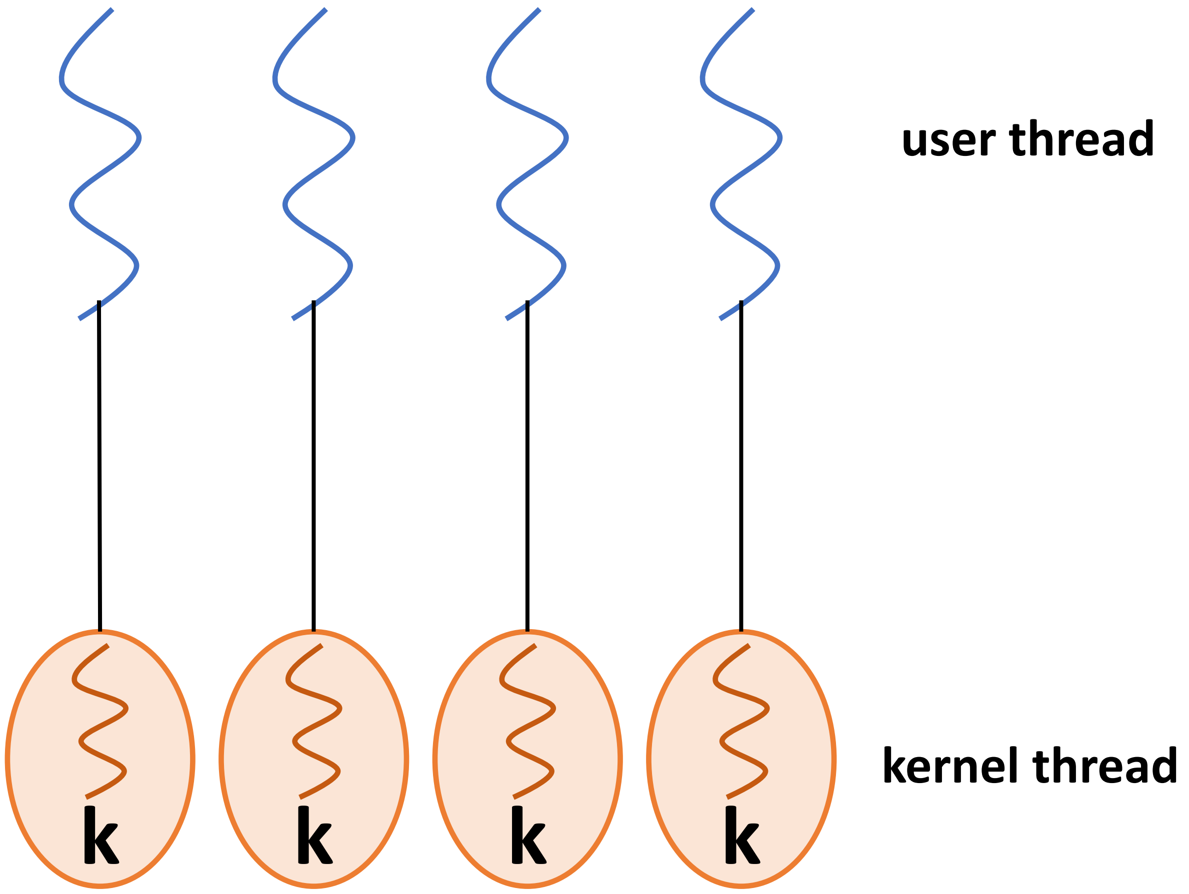


# User-level Threads: Limitations

- Usually, rely on non-preemptive scheduling
  - Preemptive scheduling can be emulated using Unix signals
- OS can make poor decisions as it is not aware of user-level threads
  - Scheduling a process with only idle threads
  - Blocking the entire process when a thread initiates I/O
  - Unscheduling a process with a thread holding a lock
- All blocking system calls should be emulated in the library via non-blocking calls to the kernel
  - Requires coordination between kernel and thread manager
- Cannot leverage multi-core CPUs

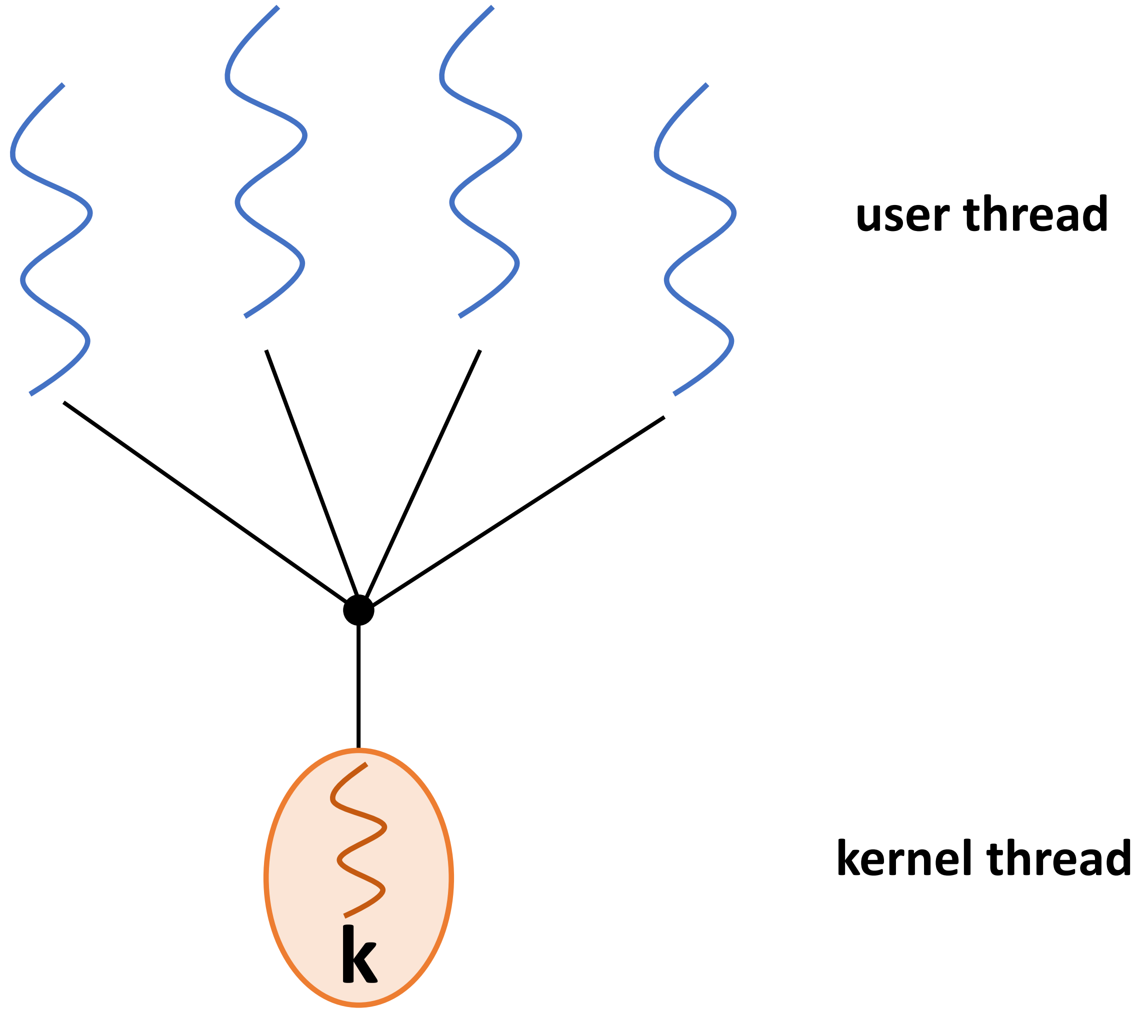
# Threading Model: One-to-One (1:1)

- Each user-level thread maps to a kernel thread
- Most popular
- Windows XP/7/10, OS/2, Linux, Solaris 9+



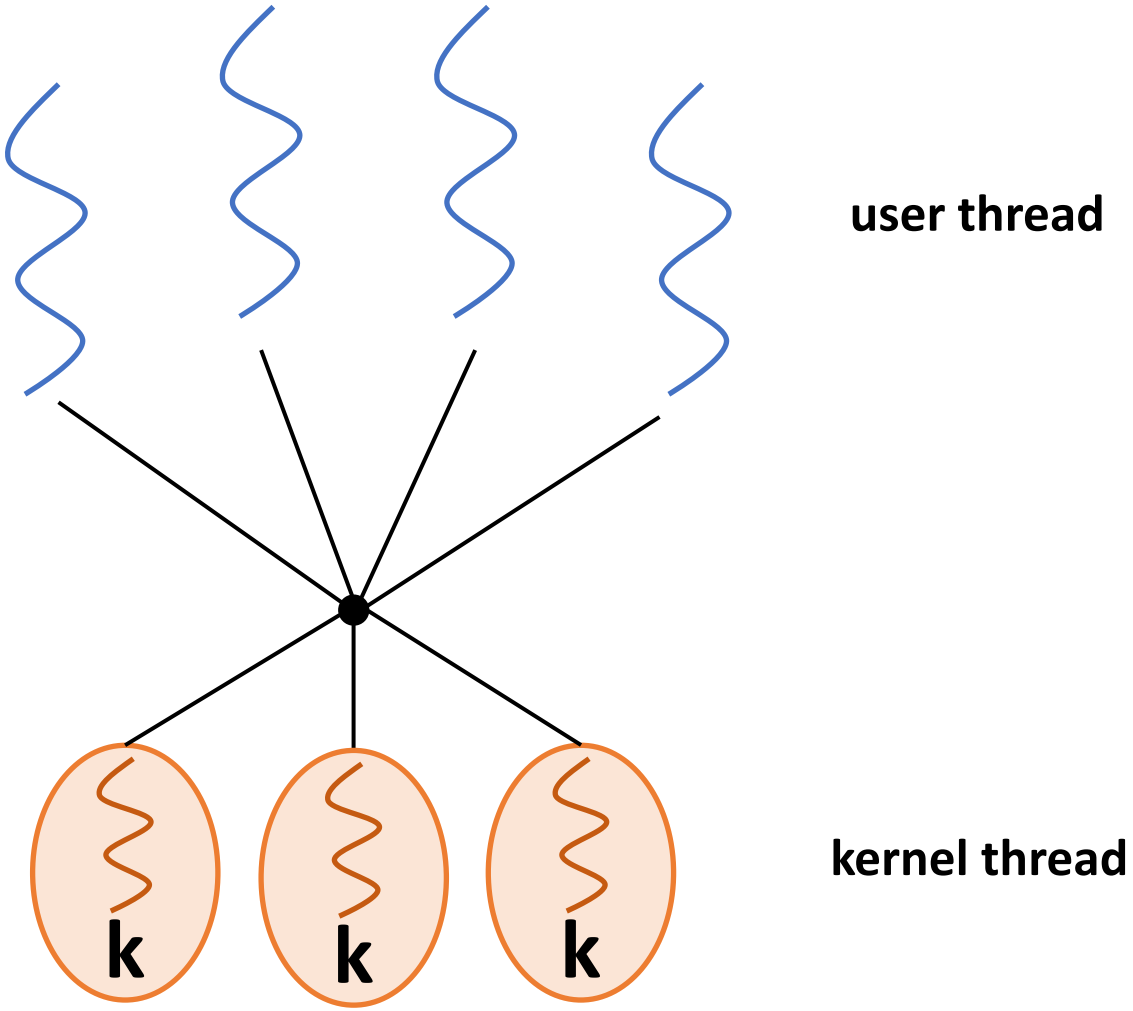
# Threading Model: Many-to-One (N:1)

- Many user-level threads mapped to a single kernel thread
- Used on systems that do not support kernel-level threads
- Solaris Green Threads, GNU Portable Threads



# Threading Model: Many-to-Many (M:N)

- Allows many user-level threads to be mapped to many kernel threads
- Allows the OS to create a sufficient number of kernel threads
- Solaris prior to v9, IRIX, HP-UX, Tru64



# Linux Thread Implementation

- In Linux, the basic unit is a "task"
  - In a program that only calls `fork()` and/or `exec()`, a task is identical to a process
- One-to-one model
  - Linux creates a task for each application thread using `clone()` system call
- Linux threads: separate tasks that may share one or more resources
  - Resources can be shared selectively in `clone()`
  - CLONE_VM, CLONE_FS, CLONE_FILES, CLONE_SIGHAND, etc.
- POSIX threads: a single process that contains one or more threads
  - CPU registers, user stack, and blocked signal mask are specific to a thread, while all other resources are global to a process
- Former POSIX compatibility problems: signal handling, `exit()`, `exec()`, …

# Summary: OS Classification

| # threads per addr space: / # of addr spaces: | One | Many |
|---|---|---|
| One | MS/DOS<br>Early Macintosh | Traditional UNIX<br>Xv6 |
| Many | Many embedded OSes<br>(VxWorks, uClinux, ..) | Mach, OS/2, Linux,<br>Windows, Mac OS X,<br>Solaris, HP-UX |