Thread

Yajin Zhou (http://yajin.org)

Zhejiang University

Acknowledgement: some pages are based on the slides from Zhi Wang(fsu).

Review

- Process
  - Multiple parts: text, CPU state, types of memory - stack, data, heap
  - State: new, running, waiting, ready, terminated
  - PCB: process control block - Linux/task_struct
- Context switch: save and restore context
- System calls: fork, exec, wait
- IPC: shared memory, message passing
- Message passing: blocking/non-blocking
- Pipe: ordinary pipe, named pipe

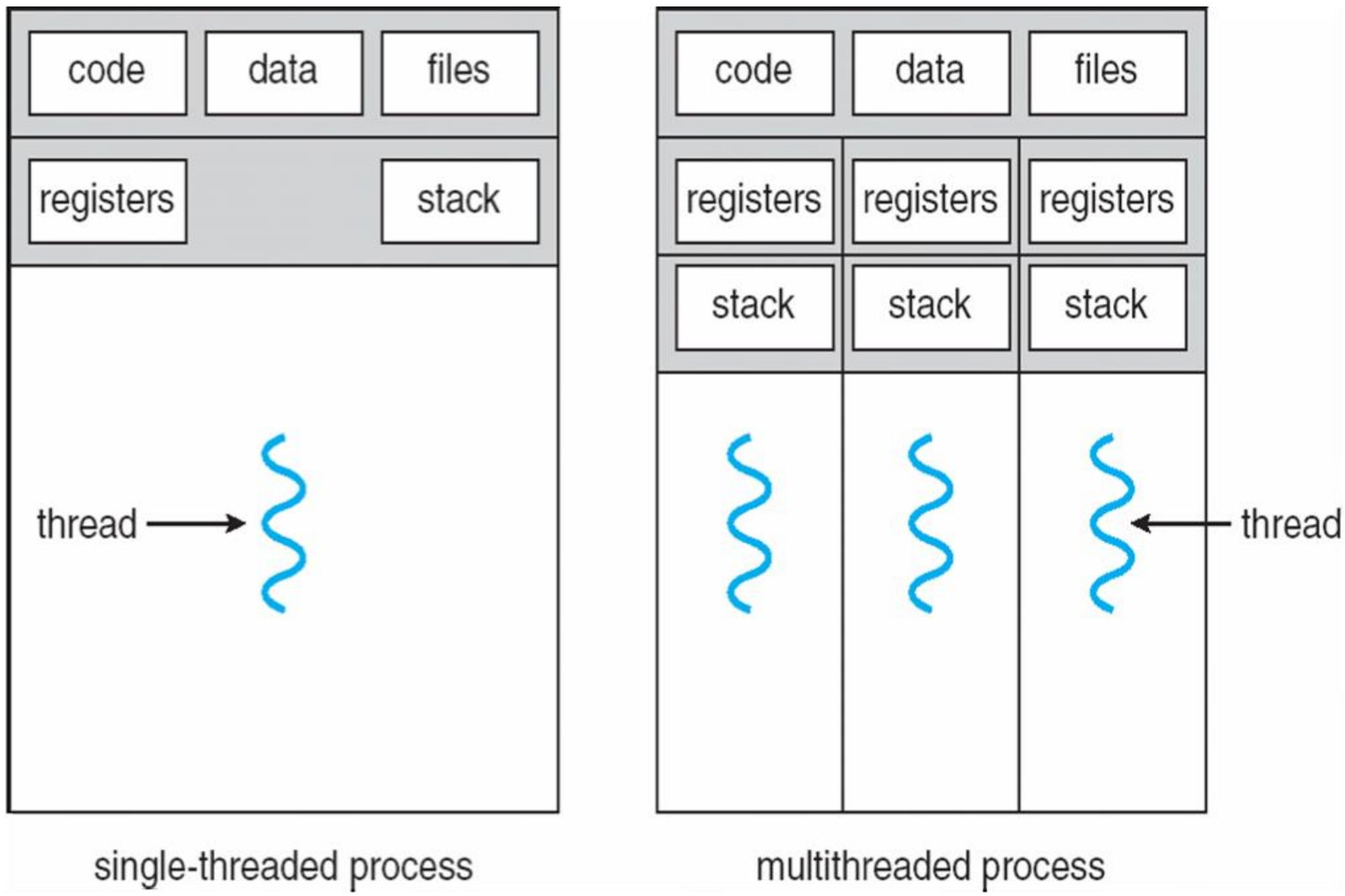
Motivation

- Why threads?
  - multiple tasks of an application can be implemented by threads
    - e.g., update display, fetch data, spell checking, answer a network request
  - process creation is heavy-weight while thread creation is light-weight - **why**?
  - threads can simplify code, increase efficiency
- Kernels are generally multithreaded
  - Why and how to understand?

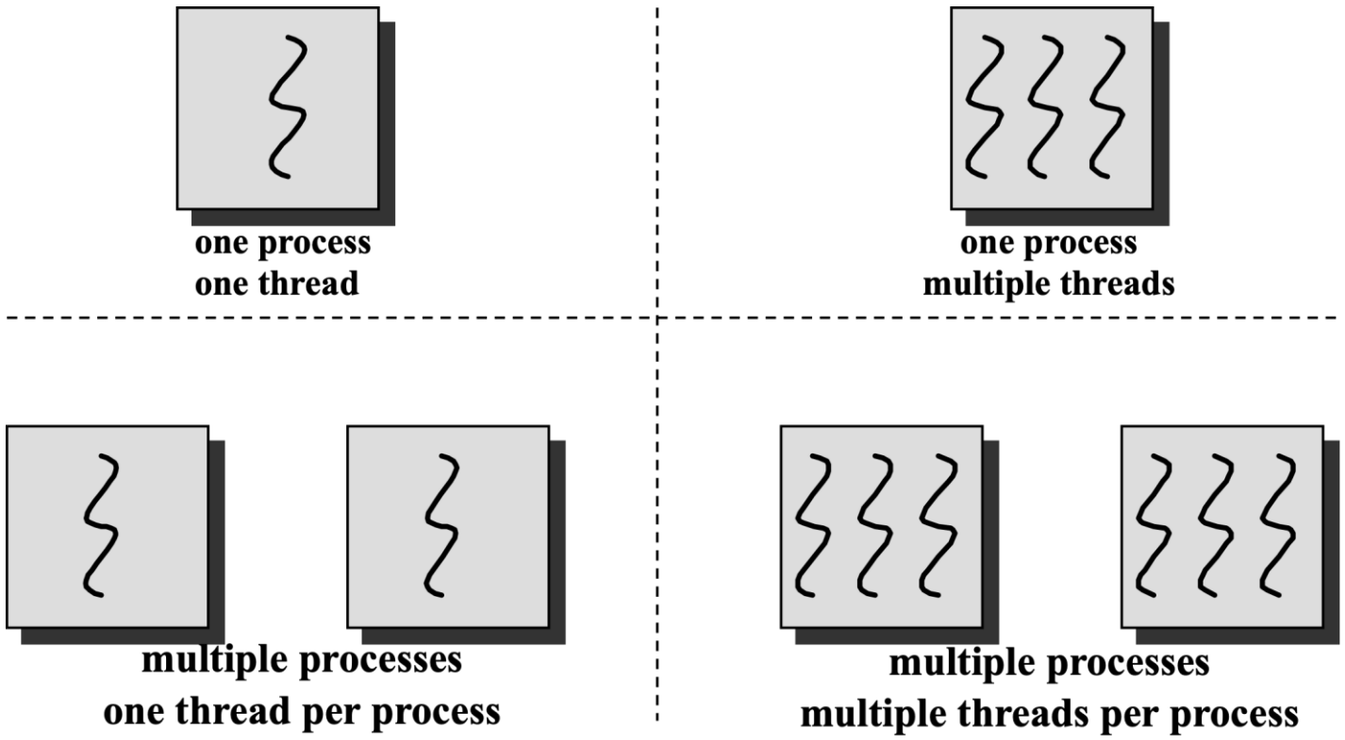
What is Thread

- A thread is an **independent stream of instructions that can be scheduled to run as such by the kernel**
- Process contains many states and resources
  - code, heap, data, file handlers (including socket), IPC
  - process ID, process group ID, user ID
  - stack, registers, and program counter (PC)
- **Threads exist within the process, and shares its resources**
  - each thread has its own essential resources (**per-thread resources**): **stack**, **registers**, **program counter**, **thread-specific data**…
  - access to shared resources need to be **synchronized, why?**
- Threads are individually scheduled by the kernel
  - each thread has its own **independent flow of control**
  - each thread can be in any of the **scheduling states**

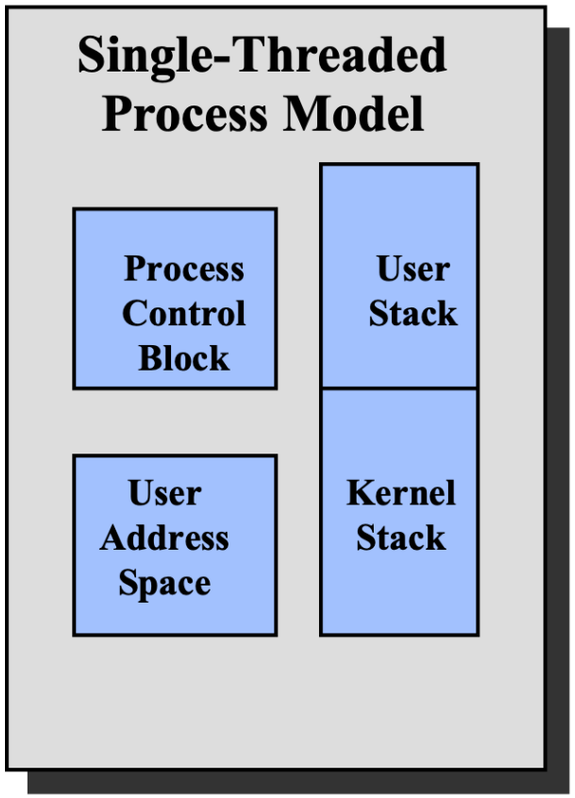
Single and Multithreaded Processes

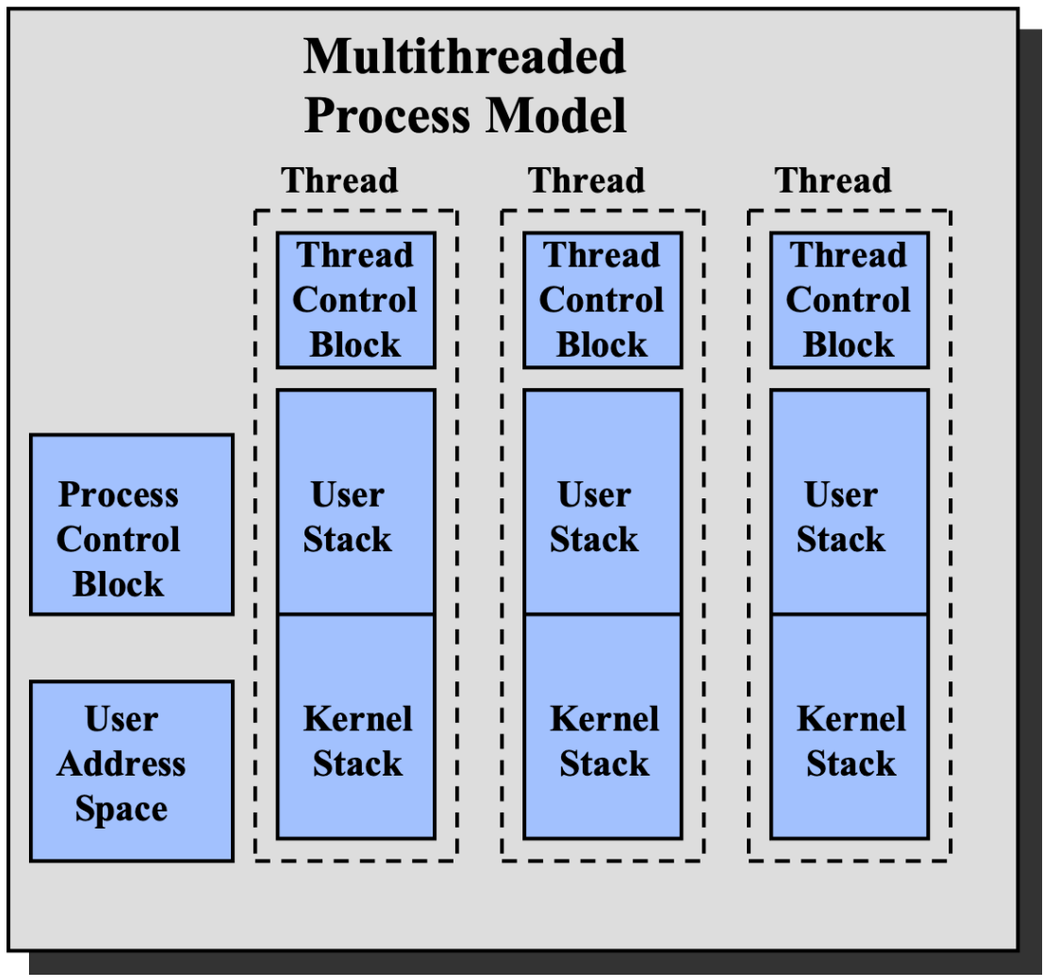


Thread and Process



Thread and Process

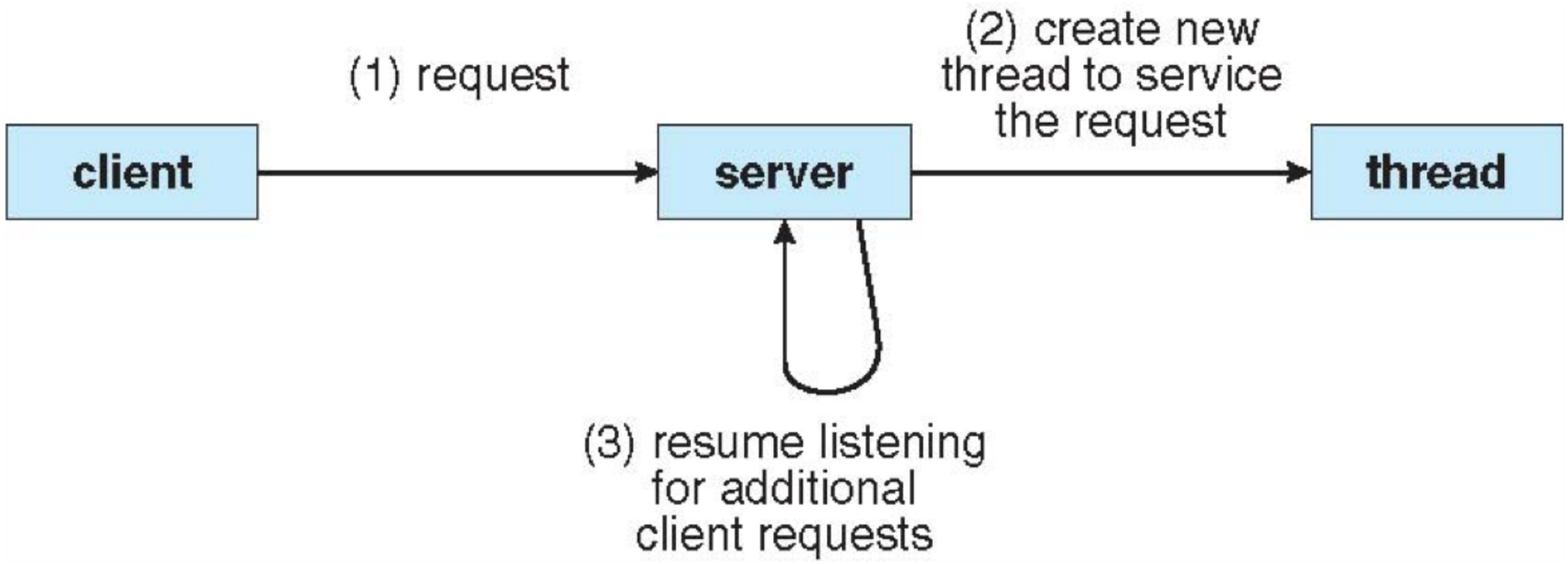




Thread Benefits

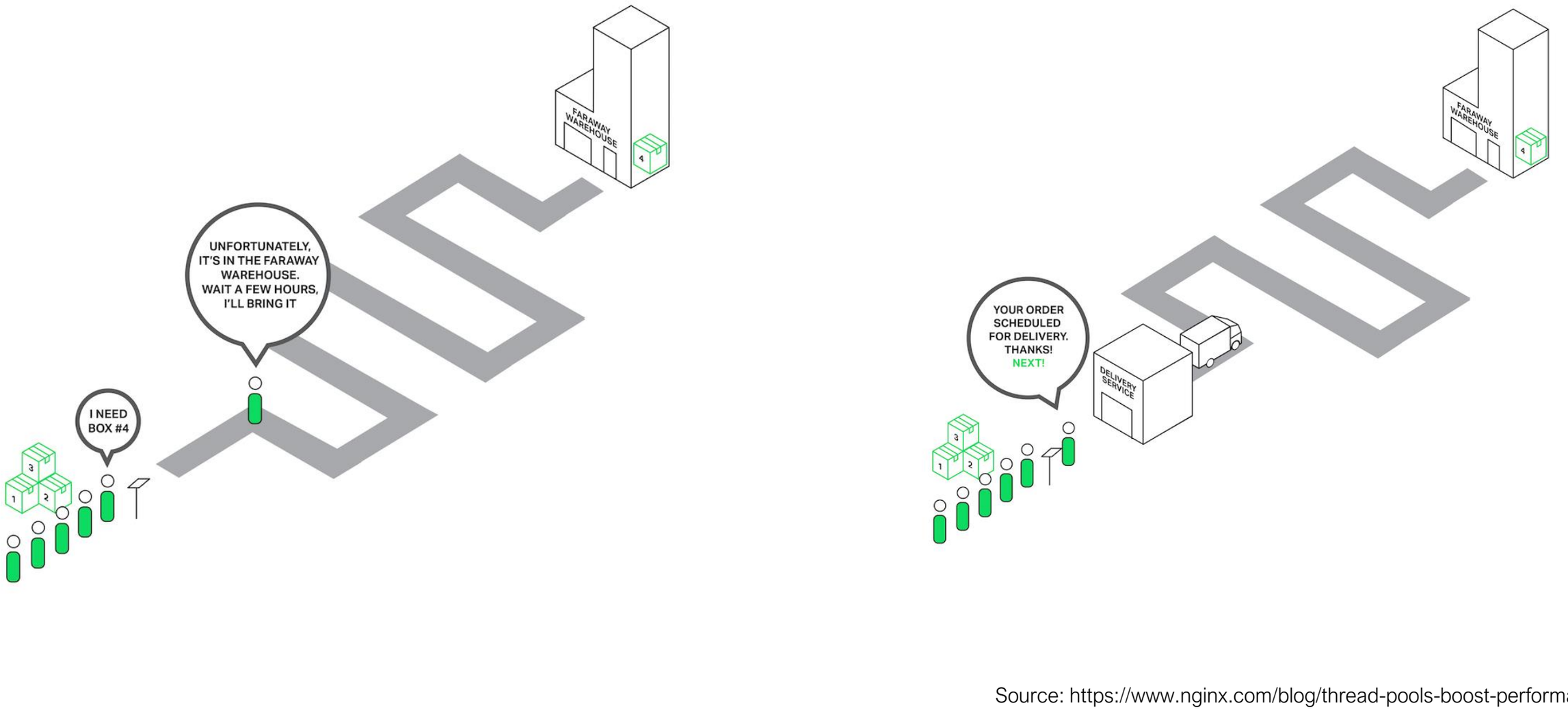
- **Responsiveness**
  - multithreading an interactive application allows a program to continue running even part of it is blocked or performing a lengthy operation
- **Resource sharing**
  - sharing resources may result in efficient communication and high degree of cooperation. Threads share the resources and memory of the process by default.
- **Economy**
  - thread is more lightweight than processes: create and context switch
- **Scalability**
  - better utilization of multiprocessor architectures: running in parallel

Multithreaded Server Architecture



NGINX Example

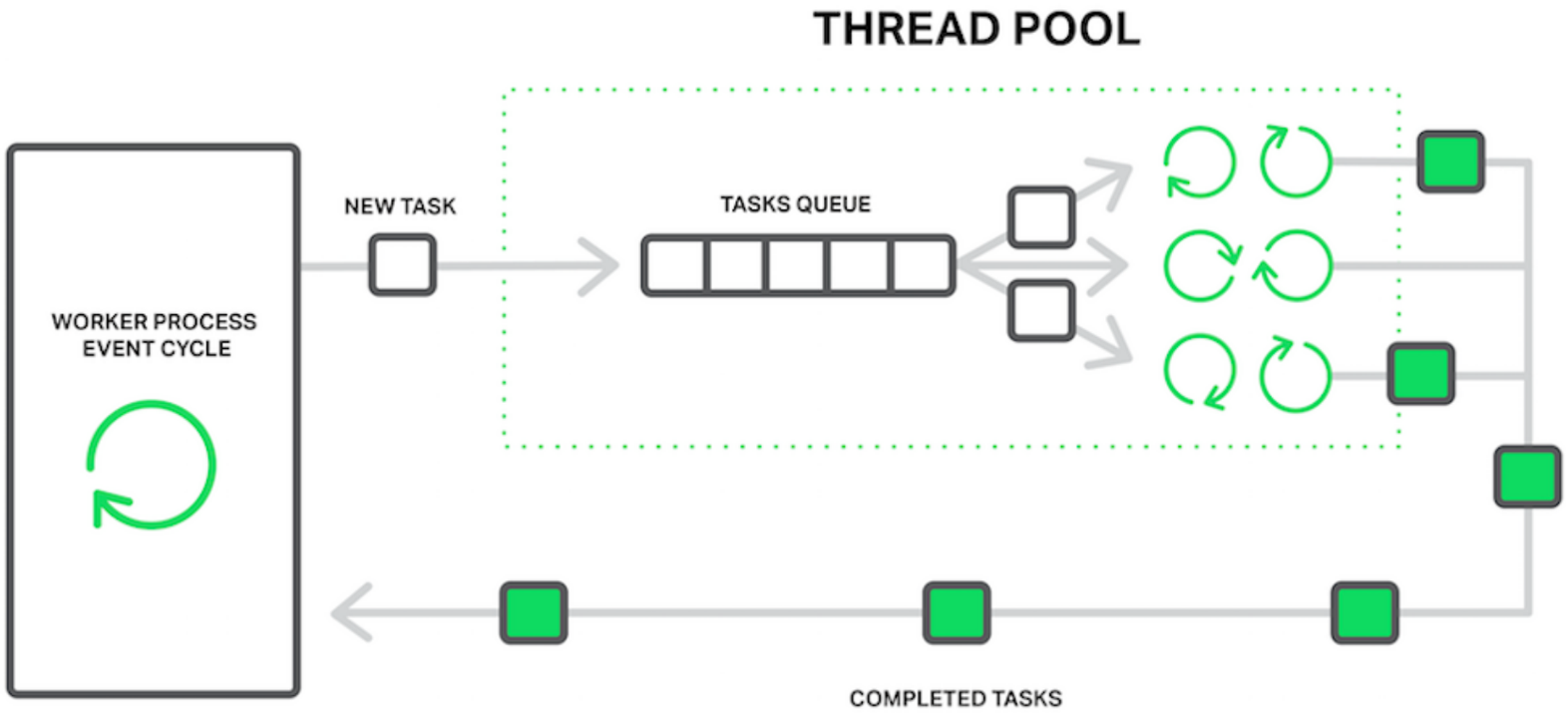
- Thread Pools in NGINX Boost Performance 9x
  - nginx : master process + worker process



Source: https://www.nginx.com/blog/thread-pools-boost-performance-9x/

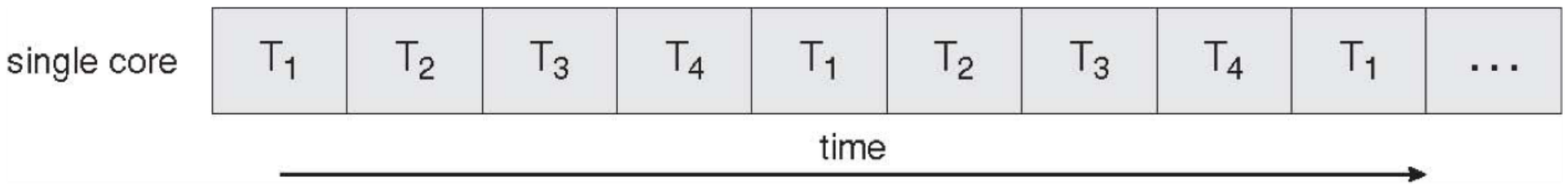
NGINX Example

- Thread Pools in NGINX Boost Performance 9x
  - nginx : master process + worker process

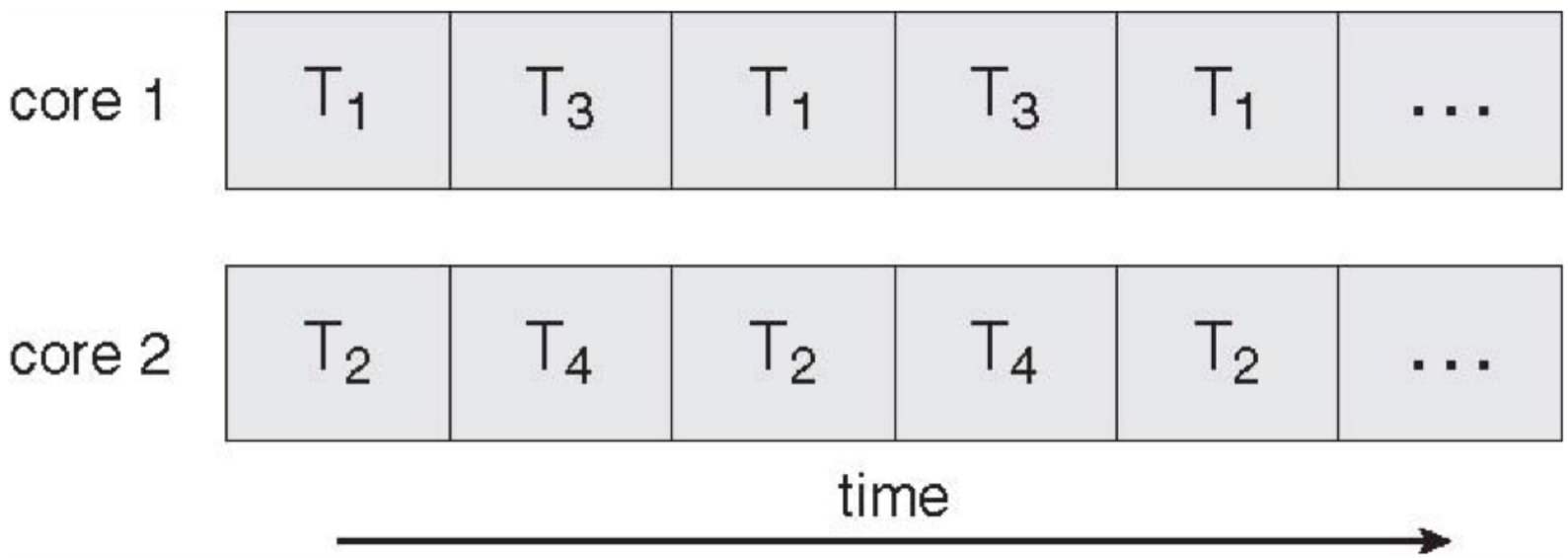


Source: https://www.nginx.com/blog/thread-pools-boost-performance-9x/

Concurrent Execution on a Single-core System



Parallel Execution on a Multicore System



Concurrency vs Parallelism

- Concurrency: 并发
- Parallelism: 并行

Concurrency

- Concurrent computing is a form of computing in which programs are designed as collections of **interacting computational processes** (注意：这里不是os中的进程的概念) that **may be executed in parallel**. Concurrent programs (processes or threads) can be executed **on a single processor** by interleaving the execution steps of each in a **time-slicing way**, or can be **executed in parallel** by assigning each computational process to one of a set of processors that may be close or distributed across a network.

- Programming as the composition of independently executing processes. These processes are **communicating** with each other. (Processes in the general sense, not Linux processes. Famously hard to define.)

Source: http://talks.golang.org/2012/waza.slide

Parallelism

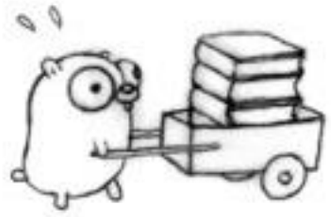
- Parallel computing is a form of computation in which many calculations are carried out **simultaneously**, operating on the principle that large problems can often be **divided into smaller ones**, which are then **solved "in parallel"**.

- Programming as the simultaneous execution of (possibly related) computations.

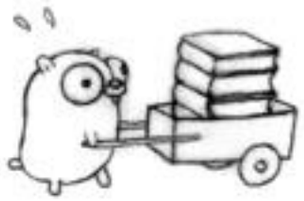
Concurrency is about **structure**, parallelism is about **execution**.

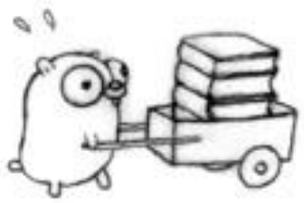
Concurrency provides a way to structure a solution to solve a problem that may (but not necessarily) be parallelizable.

Source: http://talks.golang.org/2012/waza.slide



Move a pile of obsolete language manuals to the incinerator.





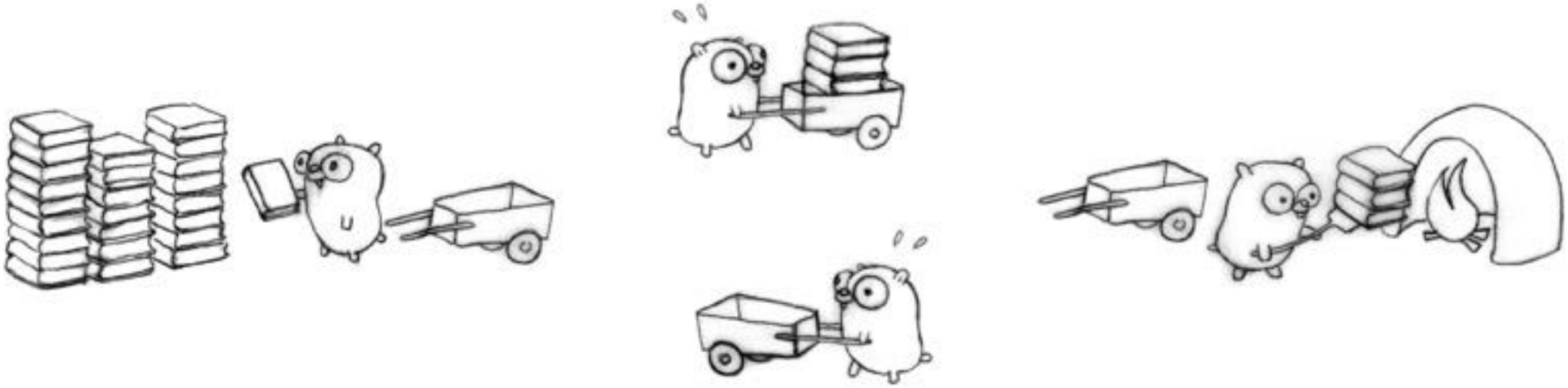
More gophers and more carts

Source: http://talks.golang.org/2012/waza.slide

Three gophers in action, but with likely delays.

Each gopher is an independently executing procedure,

plus coordination (communication).



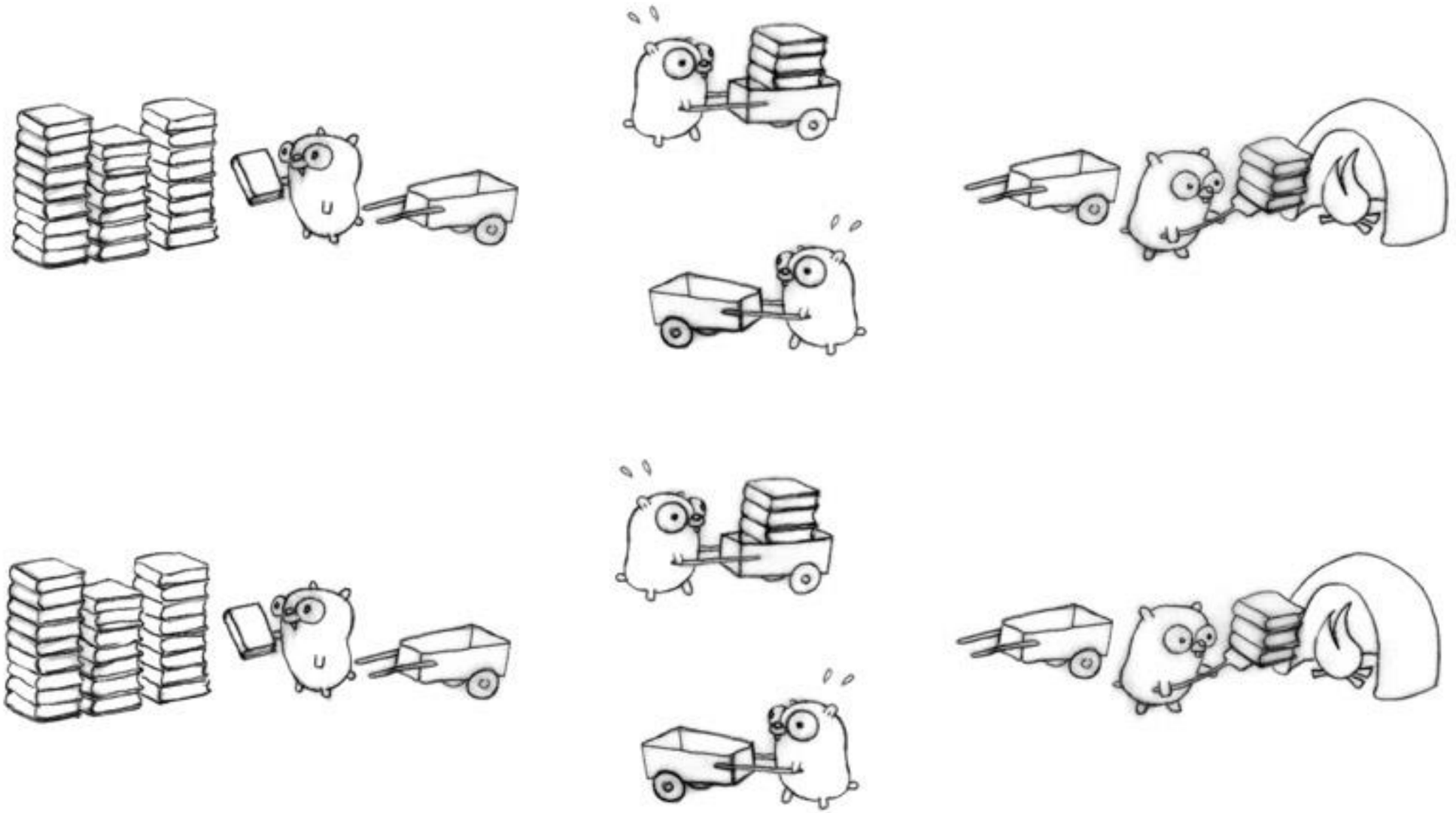
Four gophers in action for better flow, each doing one simple task.
If we arrange everything right (implausible but not impossible),
that's four times faster than our original one-gopher design.

Source: http://talks.golang.org/2012/waza.slide

Concurrency

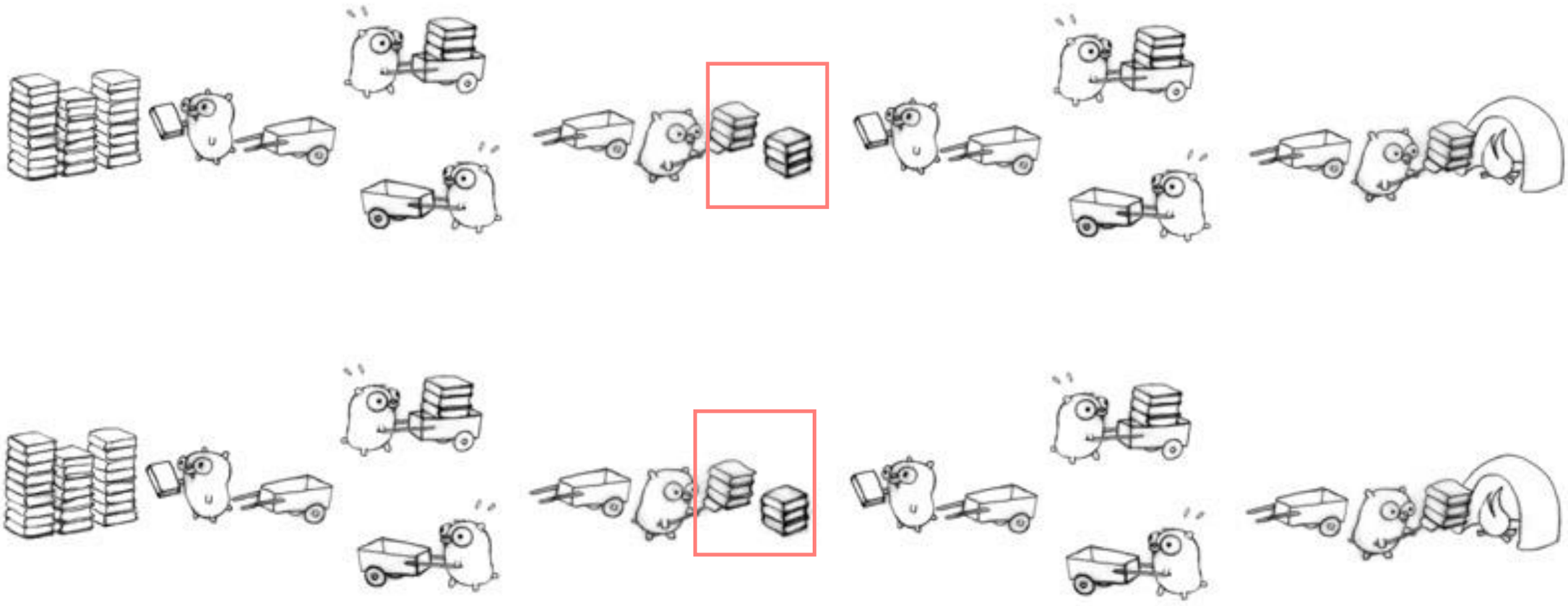
- We improved performance by adding a concurrent procedure to the existing design.
- More gophers doing more work; it runs better. This is a deeper insight than mere parallelism.
- Four distinct gopher procedures
  - load books onto cart
  - move cart to incinerator
  - unload cart into incinerator
  - return empty cart
- Different concurrent designs enable different ways to parallelize

Source: http://talks.golang.org/2012/waza.slide



We can now parallelize on the other axis; the concurrent design makes it easy.
Eight gophers, all busy

Source: http://talks.golang.org/2012/waza.slide



Another better design with a staging pile.

Source: http://talks.golang.org/2012/waza.slide

Implementing Threads

- Thread may be provided either at the user level, or by the kernel
  - **user threads** are supported above the kernel and managed without kernel support
    - three thread libraries: POSIX Pthreads, Win32 threads, and Java threads
  - **kernel threads** are supported and managed directly by the kernel
    - all contemporary OS supports kernel threads

Kernel-Level Threads

- To make concurrency cheaper, the execution aspect of process is separated out into threads. As such, the OS now manages **threads and processes**. All thread operations are implemented **in the kernel and the OS schedules all threads** in the system. OS managed threads are called **kernel-level threads**

- In this method, the kernel **knows about and manages the threads**. No runtime system is needed in this case. Instead of thread table in each process, the kernel has a **thread table** that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes. Operating Systems kernel provides system call to create and manage threads.

Source: http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh's%20files/User%20and%20Kernel%20Level%20Threads.html

Kernel-Level Threads

- Advantages
  - Because kernel has full knowledge of all threads, scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
  - Kernel-level threads are especially good for applications that frequently block.
- Disadvantages
  - The kernel-level threads are slow and inefficient. For instance, threads operations are hundreds of times slower than that of user-level threads.
  - Since kernel must manage and schedule threads as well as processes. It require a full **thread control block (TCB)** for each thread to maintain information about threads. As a result there is significant overhead and increased in kernel complexity.

Source: http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh's%20files/User%20and%20Kernel%20Level%20Threads.html

User-Level Threads

- Kernel-Level threads make concurrency **much cheaper than process because, much less state to allocate and initialize**. However, for fine-grained concurrency, kernel-level threads still suffer from too much overhead. Thread operations still require system calls. **Ideally, we require thread operations to be as fast as a procedure call**. Kernel-Level threads have to be general to support the needs of all programmers, languages, runtimes, etc. For such fine grained concurrency we need still "cheaper" threads.

- To make threads cheap and fast, they need to be implemented at user level. **User-Level threads are managed entirely by the run-time system (user-level library)**.**The kernel knows nothing about user-level threads and manages them as if they were single-threaded processes**.User-Level threads are small and fast, each thread is represented by a PC, register, stack, and small thread control block. Creating a new thread, switching between threads, and synchronizing threads are done via procedure call. i.e no kernel involvement. User-Level threads are hundred times faster than Kernel-Level threads.

Source: http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh's%20files/User%20and%20Kernel%20Level%20Threads.html

User-Level Threads

- Advantages:
  - The most obvious advantage of this technique is that a user-level threads package can be implemented on an Operating System that does not support threads.
  - User-level threads does not require modification to operating systems.
  - Simple Representation: each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
  - Simple Management: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without intervention of the kernel.
  - Fast and Efficient: Thread switching is not much more expensive than a procedure call.

Source: http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh's%20files/User%20and%20Kernel%20Level%20Threads.html

User-Level Threads

- Disadvantages
  - User-Level threads are not a perfect solution as with everything else, they are a trade off. Since, User-Level threads are **invisible to the OS they are not well integrated with the OS**. As a result, Os can make poor decisions like scheduling a process with idle threads, blocking a process whose thread initiated an I/O even though the process has other threads that can run and unscheduling a process with a thread holding a lock. Solving this requires **communication between between kernel and user-level thread** manager.
  - There is a lack of coordination between threads and operating system kernel. Therefore, process as whole gets one time slice irrespect of whether process has one thread or 1000 threads within. It is up to each thread to relinquish control to other threads.
  - User-level threads requires non-blocking systems call i.e., a multithreaded kernel. Otherwise, entire process will blocked in the kernel, even if there are runnable threads left in the processes. For example, **if one thread causes a page fault, the process blocks**.

Source: http://www.cs.iit.edu/~cs561/cs450/ChilkuriDineshThreads/dinesh's%20files/User%20and%20Kernel%20Level%20Threads.html

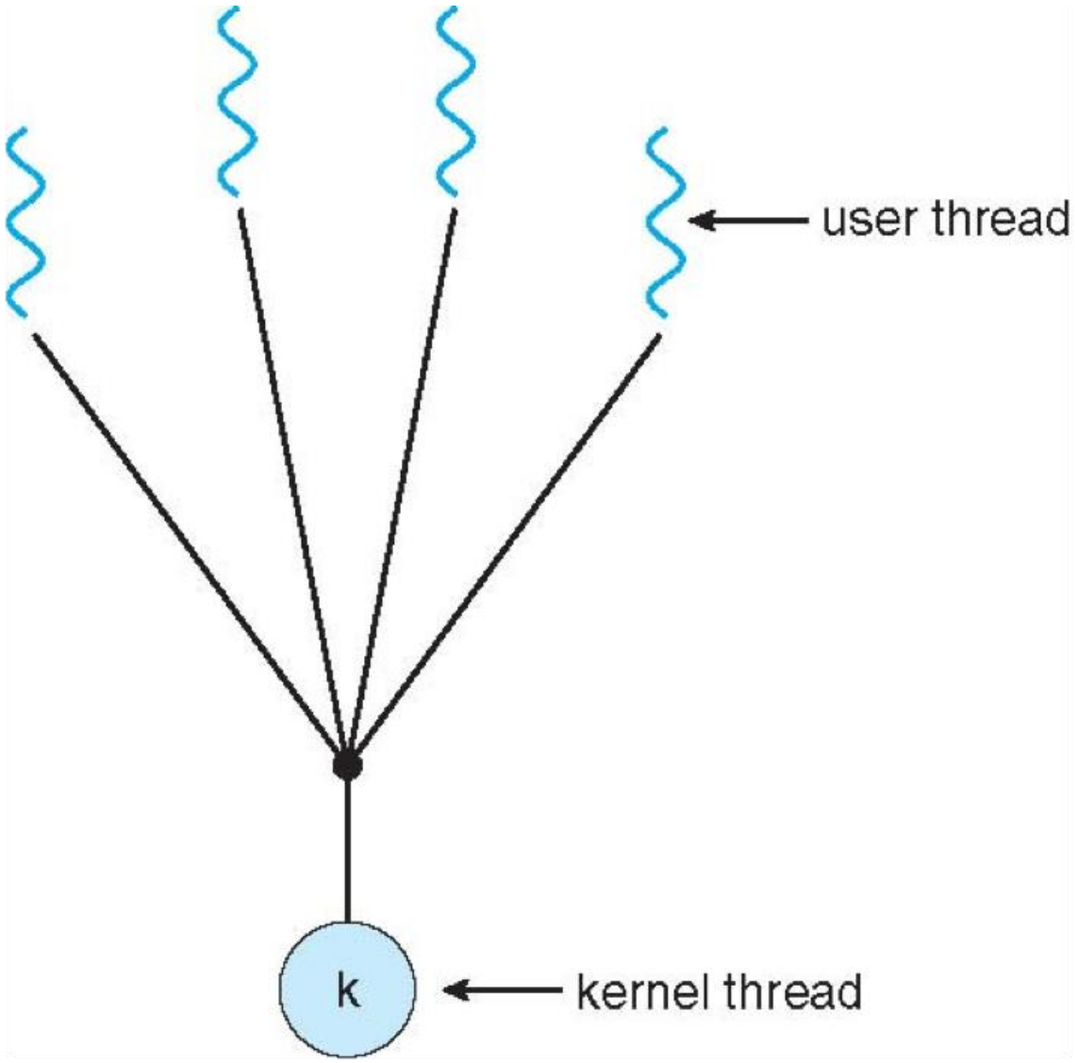
Multithreading Models

- A relationship **must exist** between user threads and kernel threads
  - Kernel threads are the real threads in the system, so for a user thread to make progress the user program has to have its scheduler take a user thread and then run it on a kernel thread.

Many-to-One

- Many user-level threads mapped to a single kernel thread
  - thread management is done by the thread library in **user space** (efficient)
  - entire process will block if a thread makes a blocking system call
    - convert blocking system call to non-blocking (e.g., select in Unix)?
  - multiple threads are unable to run in parallel on multi-processors
- Examples:
  - Solaris green threads

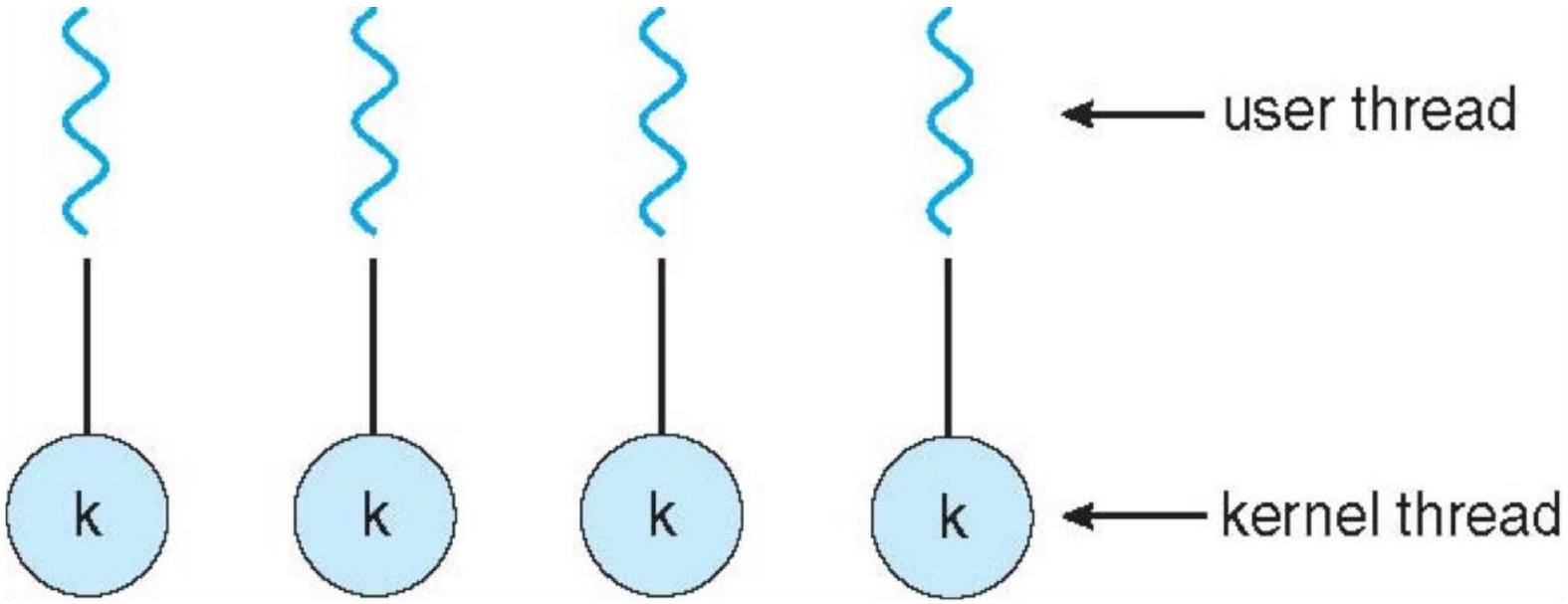
Many-to-One Model



One-to-One

- Each user-level thread maps to one kernel thread
  - it allows other threads to run when a thread blocks
  - multiple threads can run in parallel on multiprocessors
  - creating a user thread requires creating a corresponding kernel thread
    - it leads to overhead
  - most operating systems implementing this model limit the number of threads
- Examples
  - Windows NT/XP/2000
  - Linux

One-to-one Model

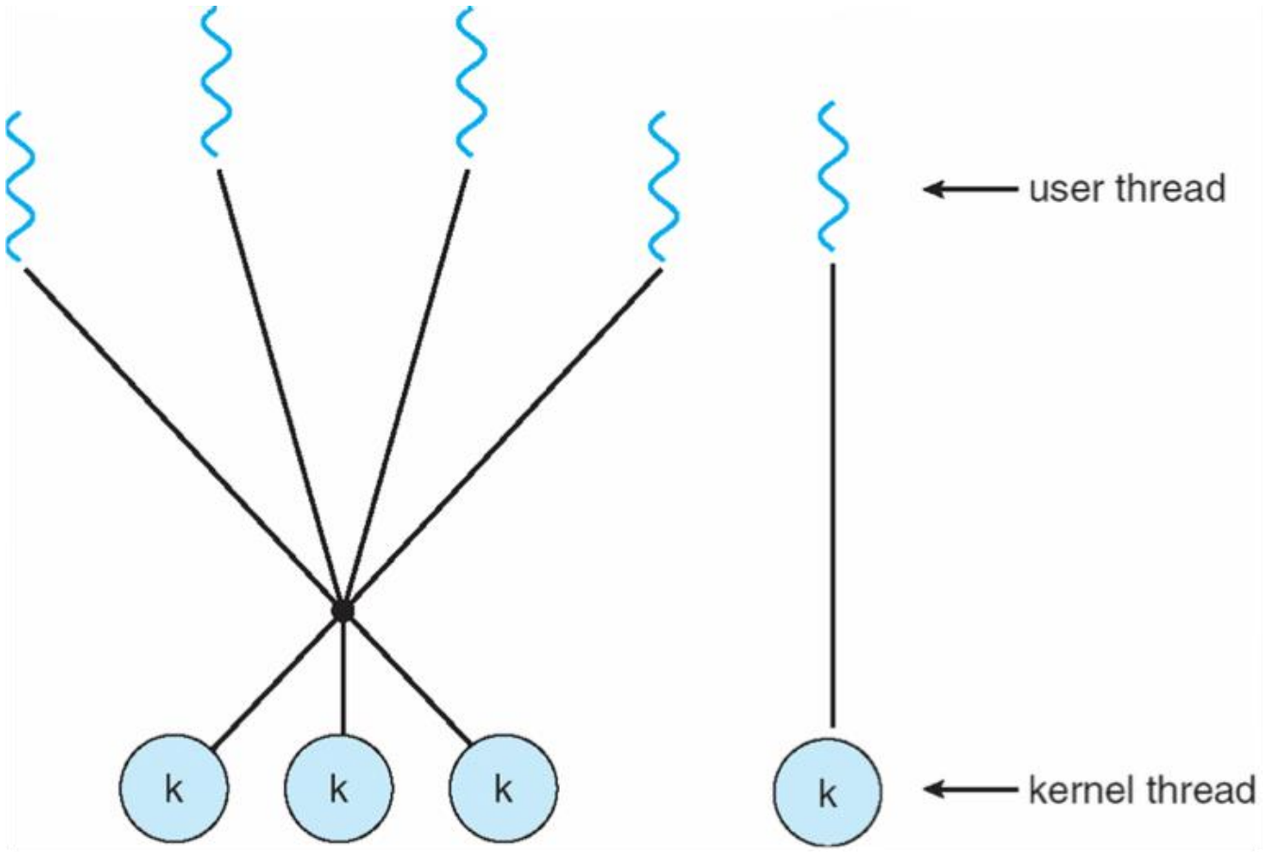


Many-to-Many Model

- Many user level threads are mapped to many kernel threads
  - it solves the shortcomings of 1:1 and m:1 model
  - developers can create as many user threads as necessary
    - corresponding kernel threads can run in parallel on a multiprocessor
- Examples
  - Solaris prior to version 9
  - Windows NT/2000 with the ThreadFiber package

Two-level Model

- Similar to many-to-many model, except that it allows a user thread to be **bound** to kernel thread



Threading Issues

- Semantics of **fork** and **exec** system calls
- Signal handling
- Thread cancellation of target thread
- Thread-specific data
- Scheduler activations

Semantics of Fork and Exec

- **Fork** duplicates the whole single-threaded process
- Does fork duplicate only the **calling** thread or **all** threads for multi-threaded process?
  - some UNIX systems have two versions of fork, one for each semantic
- **Exec** typically replaces the **entire** process, multithreaded or not
  - use “fork the calling thread” if calling exec soon after fork
- Which version of fork to use depends on the application
  - Exec is called immediately after forking: duplicating all threads is not necessary
  - Exec is not called: duplicating all threads

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred. It follows the same pattern
  - a signal is generated by the occurrence of a particular event
  - a signal is delivered to a process
  - once delivered, the signal must be handled
- 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

- A signal can be **synchronous** (exceptions) or **asynchronous** (e.g., I/O)
  - **synchronous signals are delivered to the same thread causing the signal**
- Asynchronous signals can be delivered to:
  - the thread to which the signal applies
  - every thread in the process
  - certain threads in the process (signal masks)
  - a specific thread to receive all signals for the process

Thread Cancellation

- **Thread cancellation**: terminating a (target) thread before it has finished
  - does it cancel the target thread immediately or later?
- Two general approaches:
  - **asynchronous cancellation**: terminates the target thread ***immediately***
    - what if the target thread is in critical section requesting resources?
  - **deferred cancellation**: allows the target thread to periodically check if it should be cancelled
    - Pthreads: cancellation point

Thread Cancellation

- 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);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Thread Cancellation

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state (mode/state can be set using Pthread API)

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

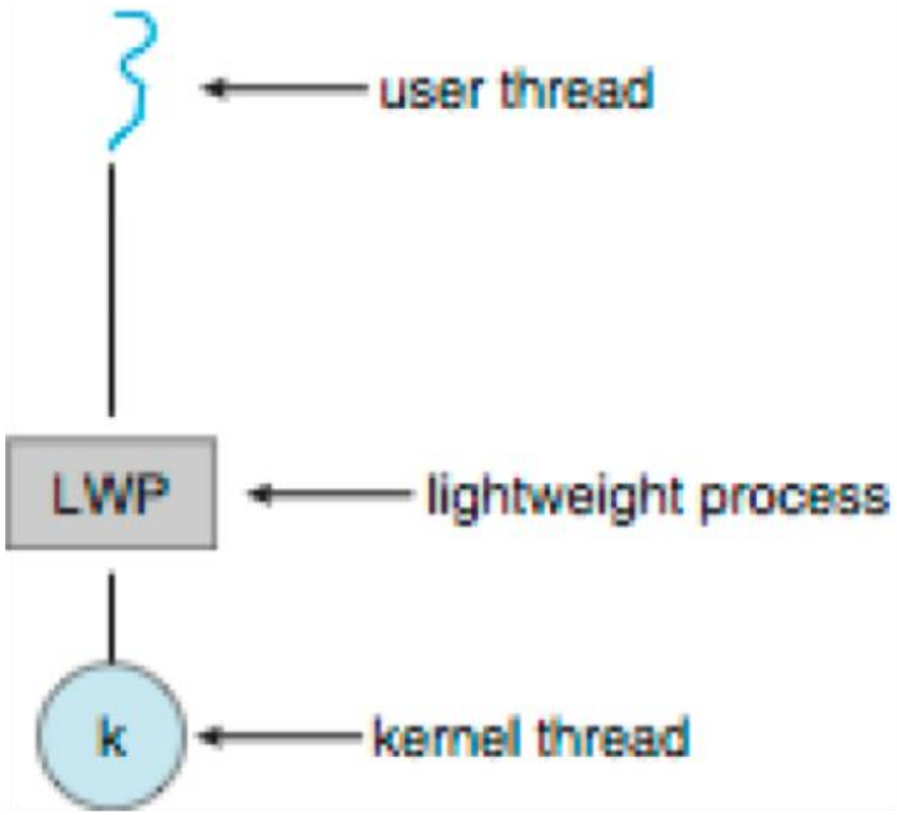
Thread Specific Data

- **Thread-local storage** (TLS) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to **static data**
  - TLS is unique to each thread

Lightweight Process & Scheduler Activations

- Lightweight process (LWP) is an intermediate data structure between the user and kernel thread in **many-to-many** and **two level** models
  - to the user-thread library, it appears as **virtual processors** to schedule user threads on
  - each LWP is attached to a **kernel thread**
    - kernel thread blocks —> LWP blocks —> user threads block
    - kernel schedules the kernel thread, thread library schedules user threads
      - thread library may make sub-optimal scheduling decision
      - solution: **let the kernel notify the library of important scheduling events**
- **Scheduler activation** notifies the library via **upcalls**
  - upcall: the kernel call a upcall handler in the thread library (similar to signal)
  - e.g., when a thread is about to block, the library can pause the thread, and schedule another one onto the virtual processor

Lightweight Processes



Lightweight Processes

- In computer operating systems, a light-weight process (LWP) is a means of achieving multitasking. In the traditional meaning of the term, as used in Unix System V and Solaris, a LWP runs in user space on top of a single kernel thread and shares its address space and system resources with other LWPs within the same process. **Multiple user level threads, managed by a thread library, can be placed on top of one or many LWPs - allowing multitasking to be done at the user level, which can have some performance benefits**

- In some operating systems there is no separate LWP layer between kernel threads and user threads. **This means that user threads are implemented directly on top of kernel threads. In those contexts, the term "light-weight process" typically refers to kernel threads and the term "threads" can refer to user threads**. **On Linux, user threads are implemented by allowing certain processes to share resources, which sometimes leads to these processes to be called "light weight processes"**

Source: https://en.wikipedia.org/wiki/Light-weight_process

Review

- The motivation of using thread
  - Responsiveness, resource sharing, economy, scalability
- Concurrency vs Parallelism
- Implementing Threads: kernel-thread, user-thread
- Thread Models
- Thread related issues
  - fork/exec, signal handling, thread cancellation, thread specific data
- Lightweight process

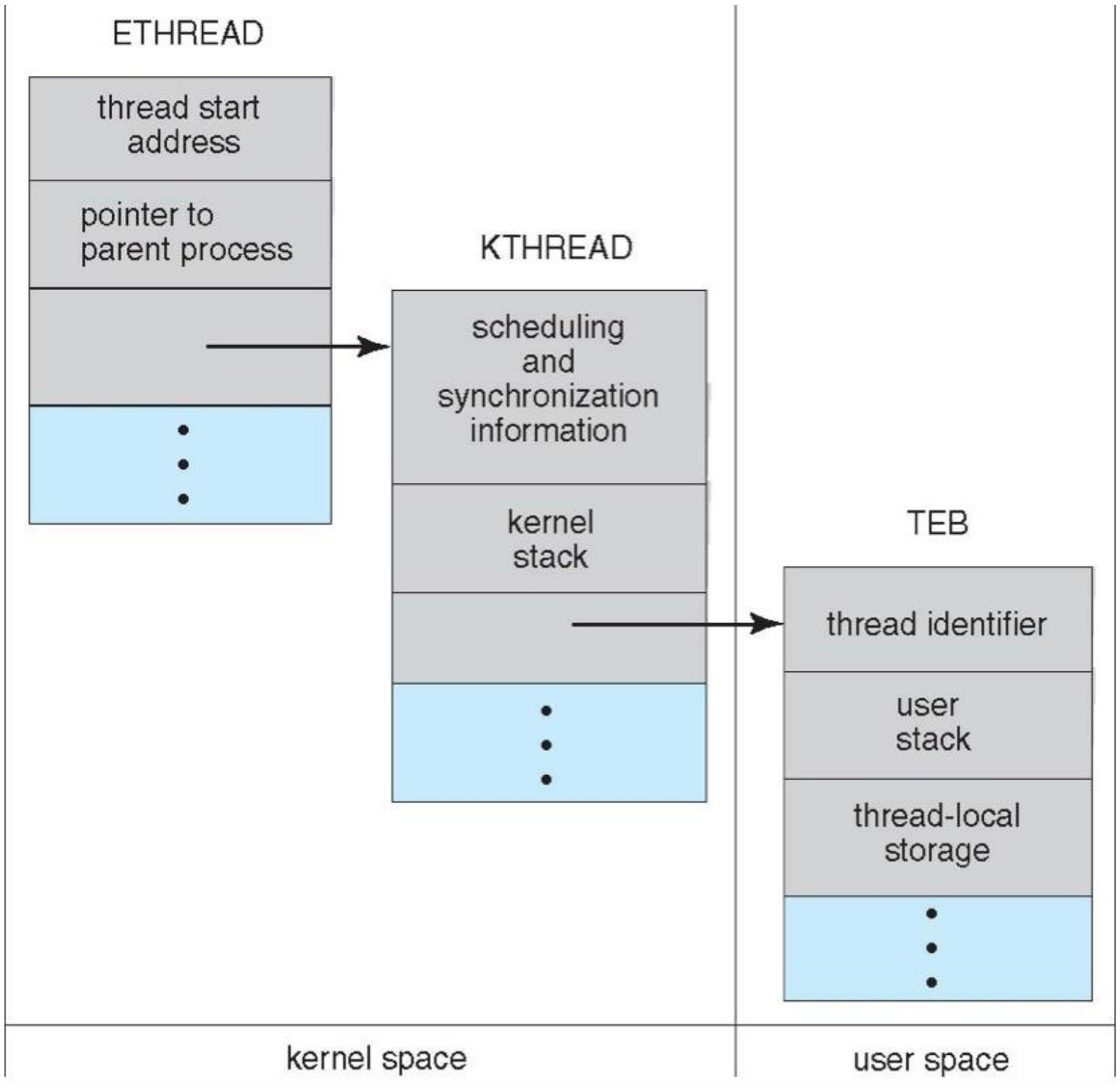
Operating System Examples

- Windows Threads
- Linux Threads

Windows XP Threads

- Win XP implements the one-to-one mapping thread model
  - each thread contains
    - a thread id
    - a register set for the status of the processor
    - a separate user stack and a kernel stack
    - a private data storage area
- The primary data structures of a thread include:
  - **ETHREAD**: executive thread block (kernel space)
  - **KTHREAD**: kernel thread block (kernel space)
  - **TEB**: thread environment block (user space)

Windows XP Threads



Linux Threads

- Linux has both fork and clone system call
- Clone accepts a set of **flags which determine sharing between the parent and children**
  - **FS/VM/SIGHAND/FILES** —> equivalent to **thread creation**
  - **no flag set no sharing** (copy) —> equivalent to **fork**
  - Linux doesn't distinguish between process and thread, uses term **task** rather than thread

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.

Linux Threads

```
os@os:~$ ps -efL
UID          PID    PPID     LWP  C NLWP STIME TTY          TIME CMD
root           1       0       1  0    1 Oct13 ?        00:00:05 /sbin/init text
root           2       0       2  0    1 Oct13 ?        00:00:00 [kthreadd]
root           4       2       4  0    1 Oct13 ?        00:00:00 [kworker/0:0H]
root           6       2       6  0    1 Oct13 ?        00:00:00 [mm_percpu_wq]
root           7       2       7  0    1 Oct13 ?        00:00:00 [ksoftirqd/0]
root           8       2       8  0    1 Oct13 ?        00:00:02 [rcu_sched]
root           9       2       9  0    1 Oct13 ?        00:00:00 [rcu_bh]
root          10       2      10  0    1 Oct13 ?        00:00:00 [migration/0]
root          11       2      11  0    1 Oct13 ?        00:00:00 [watchdog/0]
root          12       2      12  0    1 Oct13 ?        00:00:00 [cpuhp/0]
root          13       2      13  0    1 Oct13 ?        00:00:00 [cpuhp/1]
root          14       2      14  0    1 Oct13 ?        00:00:00 [watchdog/1]
root          15       2      15  0    1 Oct13 ?        00:00:00 [migration/1]
root          16       2      16  0    1 Oct13 ?        00:00:00 [ksoftirqd/1]
root          18       2      18  0    1 Oct13 ?        00:00:00 [kworker/1:0H]
root         761       1     761  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root         761       1     806  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root         761       1     807  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root         761       1     808  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root         761       1     822  0    8 Oct13 ?        00:00:01 /usr/lib/snapd/snapd
root         761       1     823  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root         761       1     824  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
root         761       1    4293  0    8 Oct13 ?        00:00:00 /usr/lib/snapd/snapd
```

Linux Threads

Linux implementations of POSIX threads

Over time, two threading implementations have been provided by the GNU C library on Linux:

LinuxThreads

This is the original Pthreads implementation. Since glibc 2.4, this implementation is no longer supported.

NPTL (Native POSIX Threads Library)

This is the modern Pthreads implementation. By comparison with LinuxThreads, NPTL provides closer conformance to the requirements of the POSIX.1 specification and better performance when creating large numbers of threads. NPTL is available since glibc 2.3.2, and requires features that are present in the Linux 2.6 kernel.

Both of these are so-called 1:1 implementations, meaning that each thread maps to a kernel scheduling entity. Both threading implementations employ the Linux clone(2) system call. In NPTL, thread synchronization primitives (mutexes, thread joining, and so on) are implemented using the Linux futex(2) system call.

Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - library entirely in user space with no kernel support
  - kernel-level library supported by the OS
- Three main thread libraries:
  - POSIX **Pthreads**
  - **Win32**
  - **Java**

Pthreads

- A POSIX standard API for thread **creation** and **synchronization**
  - common in UNIX operating systems (Solaris, Linux, Mac OS X)
- Pthread is a specification for thread behavior
  - implementation is up to developer of the library
  - e.g., Pthreads may be provided either as user-level or kernel-level

Pthreads APIs

pthread_create	create a new thread
pthread_exit	terminate the calling thread
pthread_join	join with a terminated thread
pthread_kill	send a signal to a thread
pthread_yield	yield the processor
pthread_cancel	send a cancellation request to a thread
pthread_mutex_init	initialize a mutex
pthread_mutex_destroy	destroy a mutex
pthread_mutex_lock	lock a mutex
pthread_mutex_unlock	unlock a mutex
pthread_key_create	create a thread-specific data key
pthread_key_delete	delete a thread-specific data key
pthread_setspecific	set value for the thread-specific data key
pthread_getspecific	get value for the thread-specific data key

Pthreads Example

```
struct thread_info {       /* Used as argument to thread_start() */
   pthread_t thread_id;    /* ID returned by pthread_create() */
   int       thread_num;   /* Application-defined thread # */
   char     *argv_string;  /* From command-line argument */
};

static void *thread_start(void *arg)
{   struct thread_info *tinfo = (struct thread_info *) arg;
   char *uargv, *p;

   printf("Thread %d: top of stack near %p; argv_string=%s\n",
          tinfo->thread_num, &p, tinfo->argv_string);
   uargv = strdup(tinfo->argv_string);
   for (p = uargv; *p != '\0'; p++) {
       *p = toupper(*p);
   }
   return uargv;
}
```

Pthreads Example

```
int main(int argc, char *argv[])
{   …
    pthread_attr_init(&attr);
    pthread_attr_setstacksize(&attr, stack_size);

    /* Allocate memory for pthread_create() arguments */
    tinfo = calloc(num_threads, sizeof(struct thread_info));

    /* Create one thread for each command-line argument */
    for (tnum = 0; tnum < num_threads; tnum++) {
        tinfo[tnum].thread_num = tnum + 1;
        tinfo[tnum].argv_string = argv[optind + tnum];

        /* The pthread_create() call stores the thread ID into
           corresponding element of tinfo[] */
        pthread_create(&tinfo[tnum].thread_id, &attr,
                  &thread_start, &tinfo[tnum]);
    }

    pthread_attr_destroy(&attr);

    for (tnum = 0; tnum < num_threads; tnum++) {
        pthread_join(tinfo[tnum].thread_id, &res);
        printf("Joined with thread %d; returned value was %s\n",
              tinfo[tnum].thread_num, (char *) res);
        free(res);      /* Free memory allocated by thread */
    }

    free(tinfo);
    exit(EXIT_SUCCESS);
}
```

Win32 API Multithreaded C Program

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */

/* The thread will execute in this function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 1; i <= Upper; i++)
      Sum += i;
   return 0;
}
```

Win32 API Multithreaded C Program

```
int main(int argc, char *argv[])
{
   DWORD ThreadId;
   HANDLE ThreadHandle;
   int Param;

   Param = atoi(argv[1]);
   /* create the thread */
   ThreadHandle = CreateThread(
    NULL, /* default security attributes */
    0, /* default stack size */
    Summation, /* thread function */
    &Param, /* parameter to thread function */
    0, /* default creation flags */
    &ThreadId); /* returns the thread identifier */

  /* now wait for the thread to finish */
  WaitForSingleObject(ThreadHandle,INFINITE);

  /* close the thread handle */
  CloseHandle(ThreadHandle);

  printf("sum = %d\n",Sum);
}
```

Java Threads

- Java threads are managed by the Java VM
  - it is implemented using the threads model provided by underlying OS
- Java threads may be created by:
  - extending the **java.lang.Thread** class
  - then implement the **java.lang.Runnable** interface

HW4 is out!