PART 2. PROCESS MANAGEMENT

Chapter 4: Multithreaded Programming

WHAT'S AHEAD:

- Overview
- Multithreading Models
- Thread Libraries
- Implicit Threading
- Threading Issues
- Operating System Examples

WE AIM:

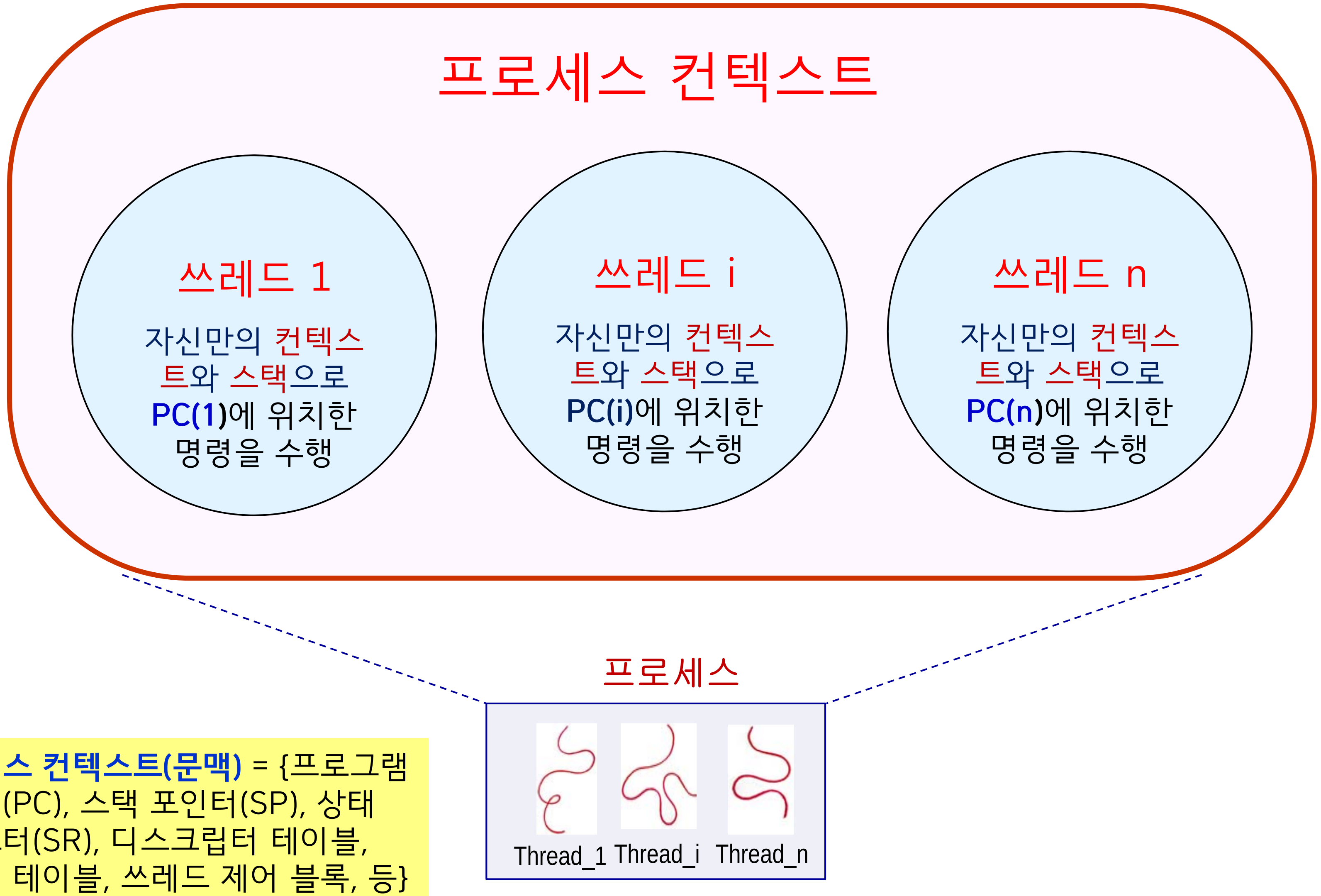
- To introduce the notion of a thread
- To discuss the APIs for multithreading
- To explore implicit threading and other issues
- To cover OS support for threads in Windows, Linux

Note: These lecture materials are based on the lecture notes prepared by the authors of the book titled *Operating System Concepts*, 9e (Wiley)

핵심요점

쓰레드, 쓰레드로 처리하기

(Thread, Threading)



프로세스 컨텍스트(문맥) = {프로그램 카운터(PC), 스택 포인터(SP), 상태 레지스터(SR), 디스크립터 테이블, 페이지 테이블, 쓰레드 제어 블록, 등}

Core Ideas Threads, Threading

Process context

Thread *1*

Instructions at **PC(*1*)** are executed here with its own context and stack

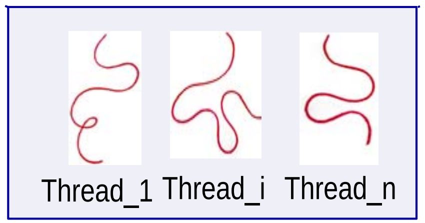
Thread *i*

Instructions at **PC(*i*)** are executed here with its own context and stack

Thread *n*

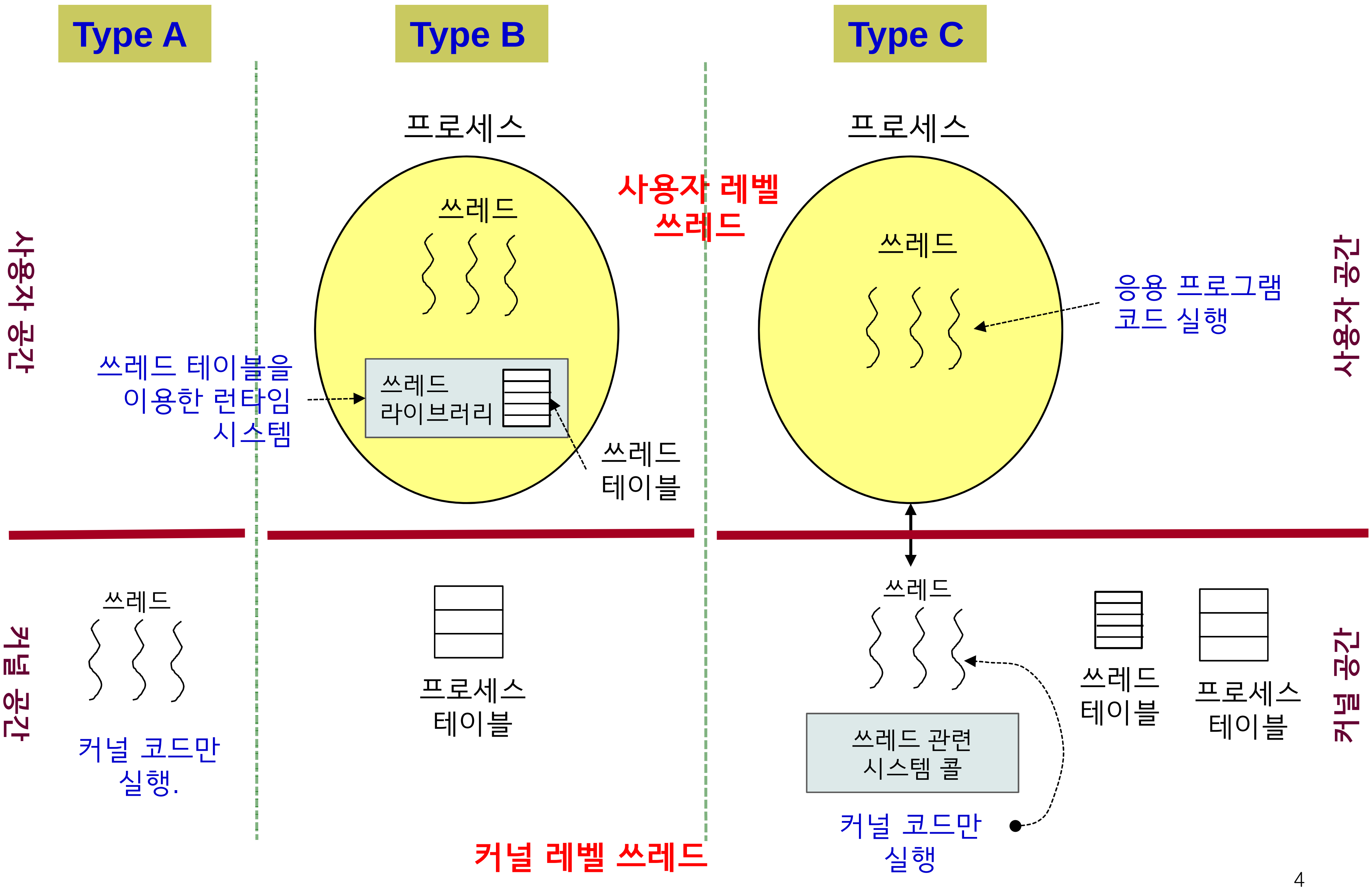
Instructions at **PC(*n*)** are executed here with its own context and stack

Process

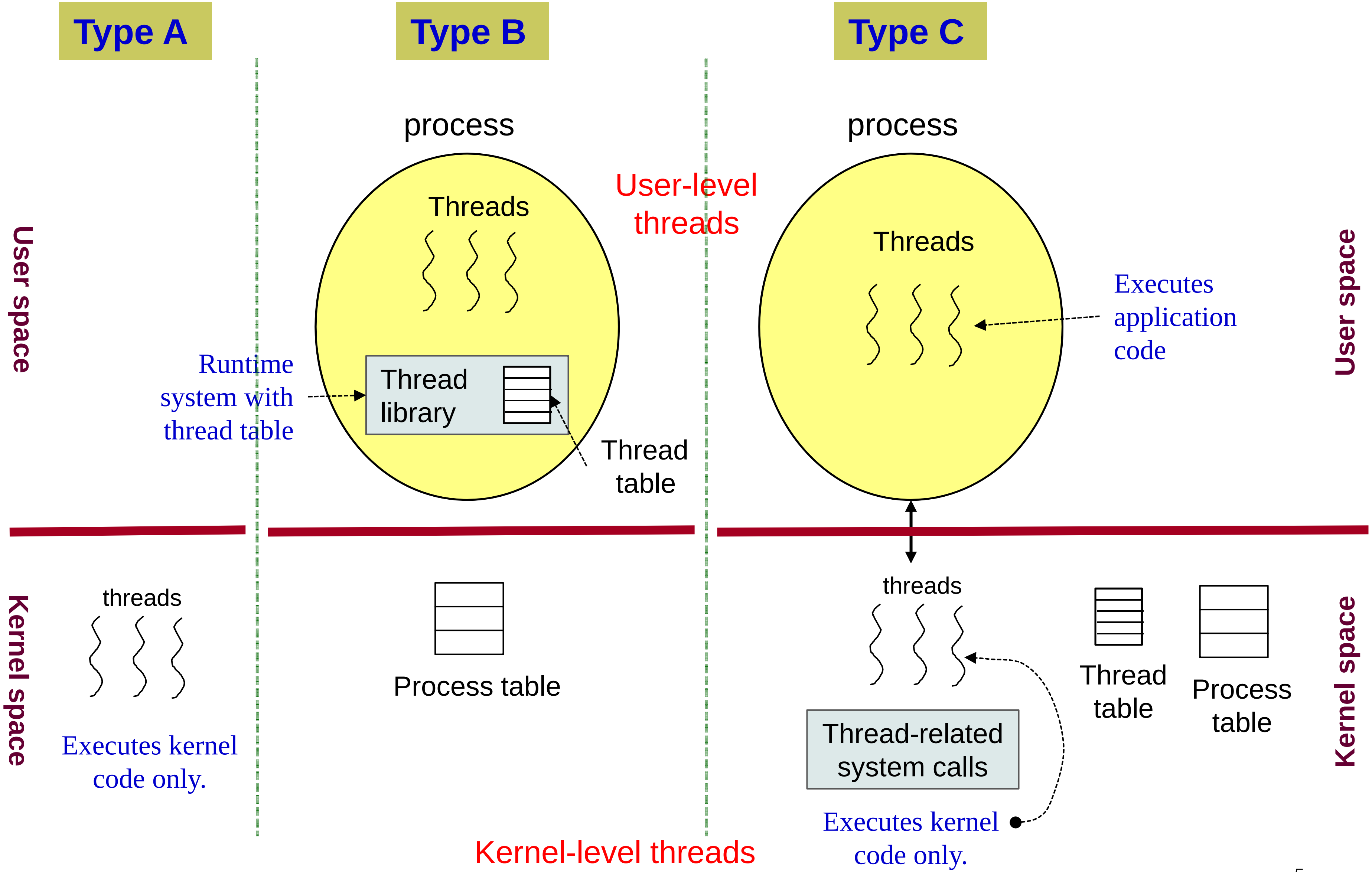


Process context = {program counter(PC), stack pointer(SP), status register(SR), descriptor tables, page table, thread control block, etc.}

핵심요점 쓰레드의 구현


Type A
Type B
Type C
프로세스
쓰레드
사용자 레벨 쓰레드
프로세스
쓰레드
응용 프로그램 코드 실행
사용자 공간
사용자 공간
쓰레드 테이블을 이용한 런타임 시스템
쓰레드 라이브러리
쓰레드 테이블
쓰레드
커널 공간
쓰레드
프로세스 테이블
쓰레드 테이블
프로세스 테이블
커널 공간
커널 코드만 실행.
쓰레드 관련 시스템 콜
커널 코드만 실행
커널 레벨 쓰레드

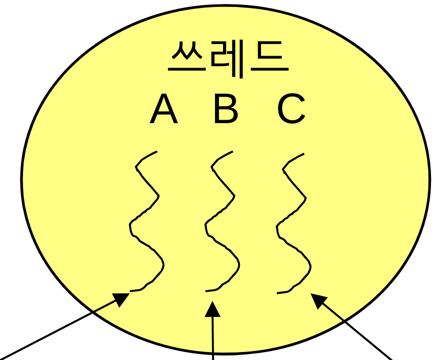
Core Ideas
How to Implement Threads



핵심요점 쓰레드 이용의 장점

전자 표 계산과 같은 응용을 수행할 때 성능향상 방법은?

"Exel" 프로세스



데이터 수정시 다시 계산

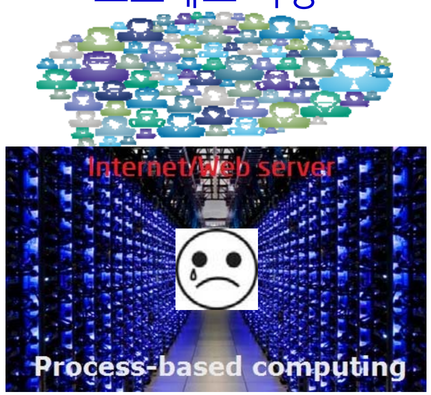
디스크에 주기적으로 백업

사용자 인터페이스 (입출력 연산)

병행계산이나 병렬**계산 가능**

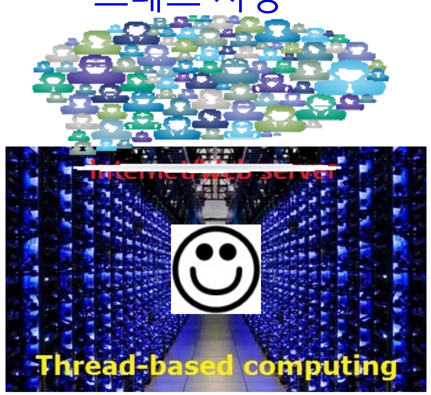
수천 명의 클라이언트가 동시에 인터넷 서비스를 요청한다면?

프로세스 사용



vs.

쓰레드 사용



프로세스 처리 - 어려움

느림. 더 많은 자원 요구.
코드/데이터 공유가 어려움

쓰레드 처리 - 쉬움

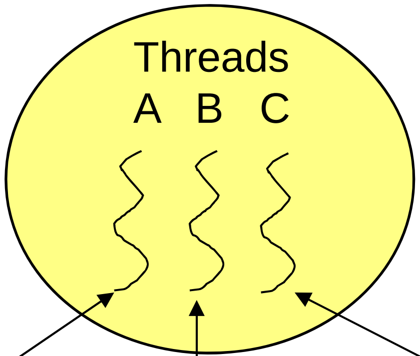
빠름. 적은 자원 요구
코드/데이터 공유가 용이함

코드 작성 및 테스트가 어려움
쓰레드 간의 병행성 제어가 어려움

Core Ideas The Virtue of Threading

How can we speed up the application, such as electronic spreadsheet?

"**Exel**" process



Recomputation upon modification of data

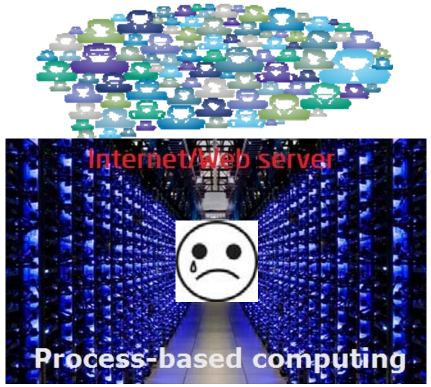
Periodic backups to disk

User interactions (I/O operations)

Concurrency or ***parallelism*** possible

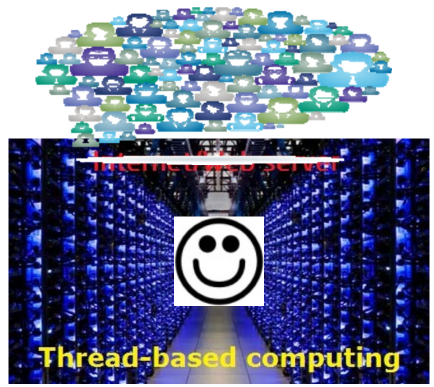
What would happen if thousands of clients request Internet services simultaneously?

Process-based



vs.

Thread-based



Hard to handle processes

Slow, more resources, hard to share code/data

Easy to handle threads

Fast, less resources, easy to share code/data

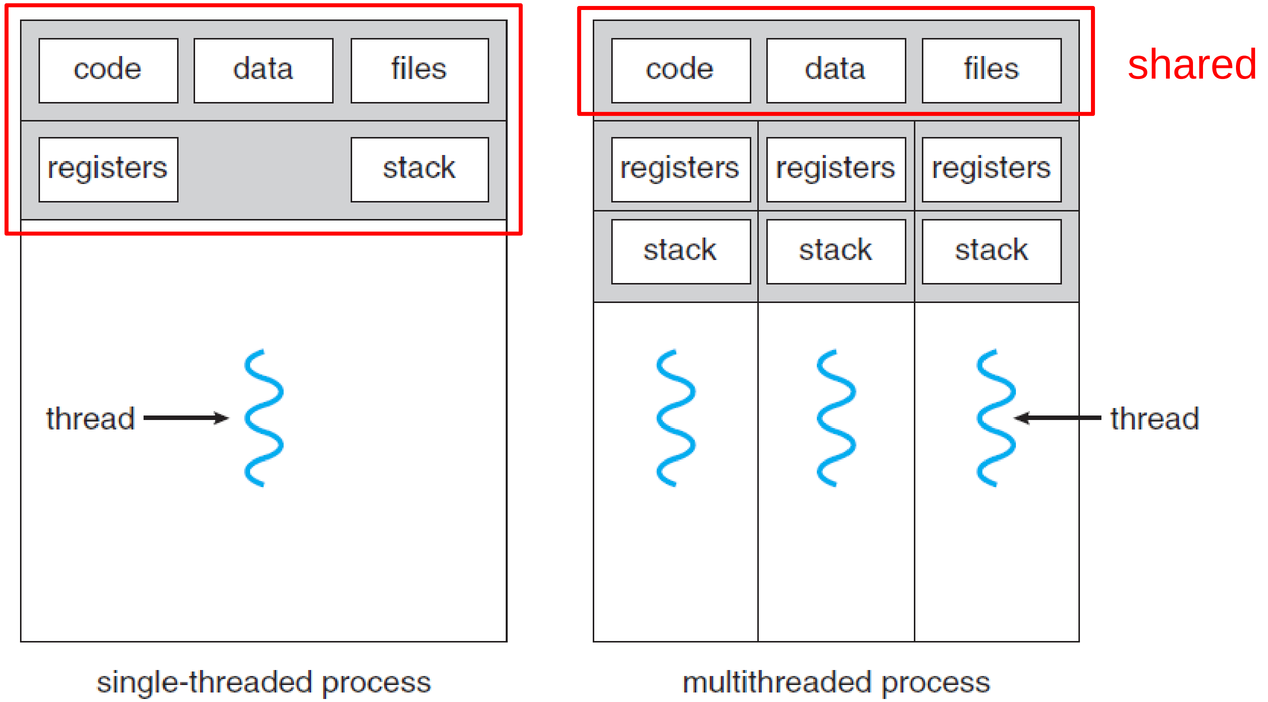
Hard to write/test code
Hard to manage concurrency among threads

Overview - Motivation

- Most modern applications are multithreaded
  - Threads run within an application process
- Multiple tasks (operations) with the application can be implemented by separate threads: e.g.,
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
  - also, process switching (or context switching) is costly because the PCB and other data which need swapping is large
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Single- and Multi-threaded Processes

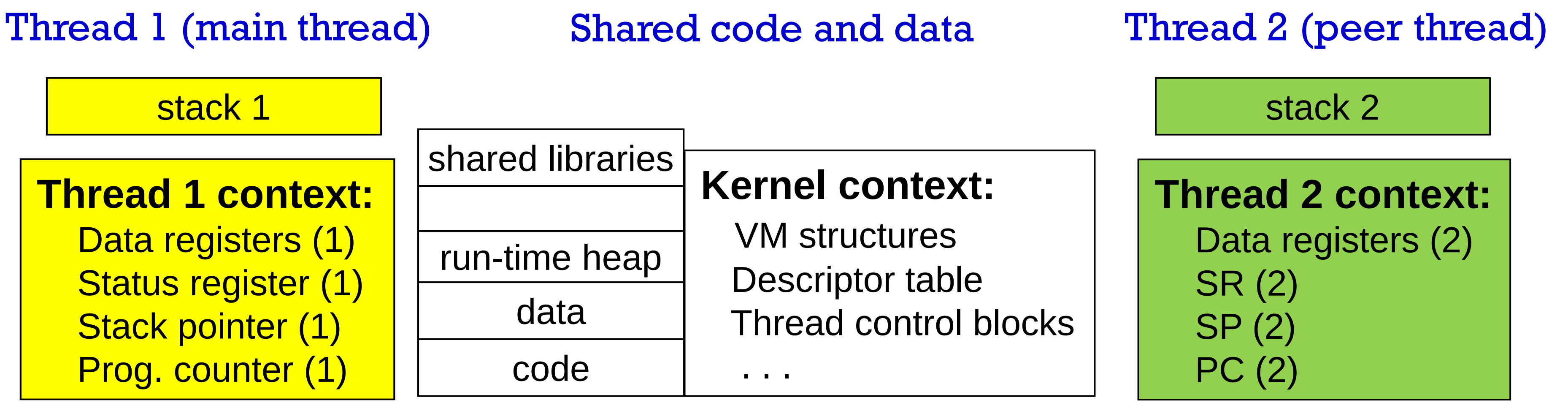
- A "thread" of execution
  - May be considered a locus of execution, that is, a trace of execution of instructions within a process
  - An entity that can be independently scheduled
  - A process has one or more threads, sharing the same address space



Thread vs. Process

- Case 1: No thread implementation, process only
  - Process = {process context, code, data, stack}
    - process context = {program context, kernel context}
    - program context = {PC, SP, status register(SR), registers … }
    - kernel context = {VM structures, descriptor table, … }
- Case 2: Process with threads
  - Process = {thread(s), code, data, kernel context}
    - thread = {thread context, stack}
    - thread context = {PC, SP, SR, registers} of its own

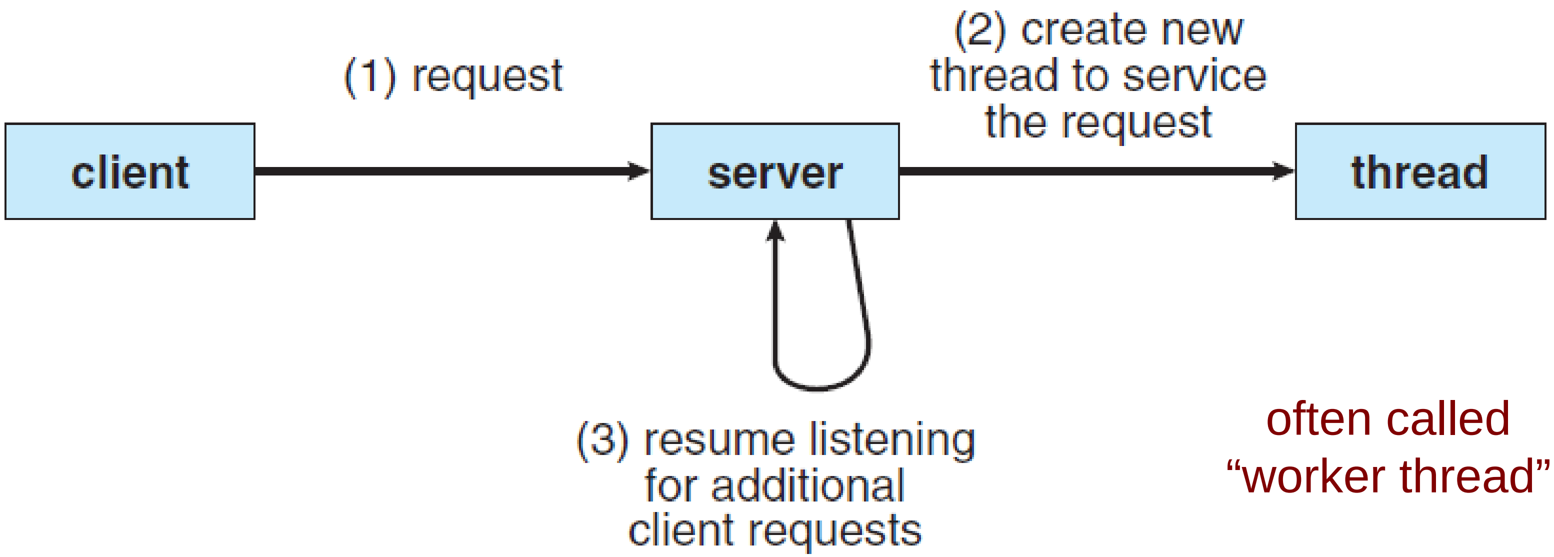
VM: Virtual memory



Benefits of Using Threads

- **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
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multiprocessor architectures

Application 1: Multithreaded Server

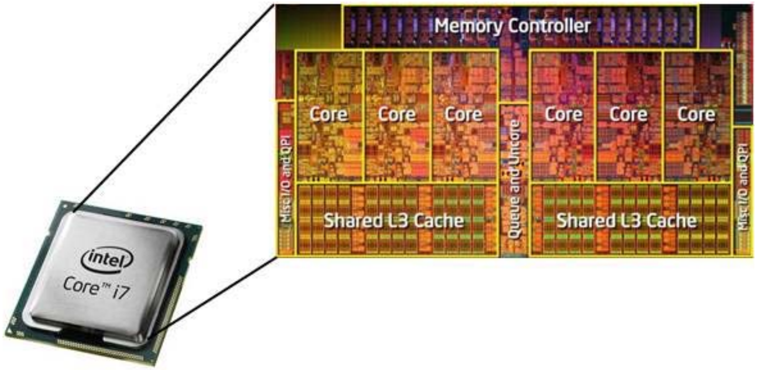


often called "worker thread"

Instead of creating a new process to service a client's request, the OS creates a new thread within the server process, called worker thread. This incurs much less overhead than in the case of creating a new process. We may craft a group of threads in advance so that one of them may be assigned to an incoming request upon their arrival. This approach will be discussed in the section on "thread pools".

Application 2: Multicore Programming

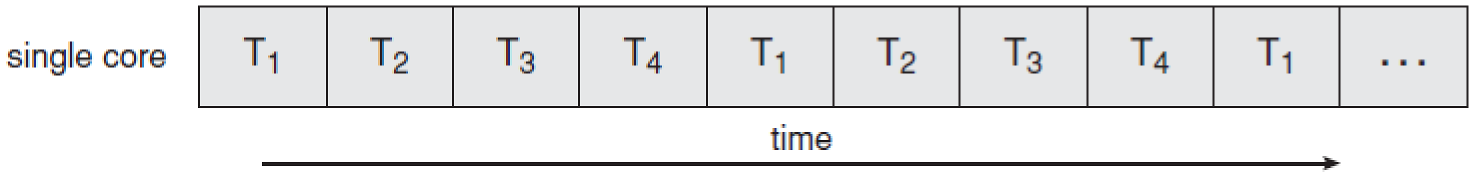
- Multicore microprocessors
  - Multiple cores (or CPUs) on a single microprocessor
  - Homogeneous cores → SMP (symmetric multiprocessors)
  - Heterogeneous cores → e.g., general-purpose, network, graphics, etc.
- Challenges to programmers
  - Dividing activities and balancing loads across processors
  - Data splitting and data dependency
  - Testing and debugging



- Parallelism
  - Implies a system can perform more than one task simultaneously – needs a separate processor for each task
  - In contrast, **concurrency** means execution periods of multiple tasks are overlapped on a single processor

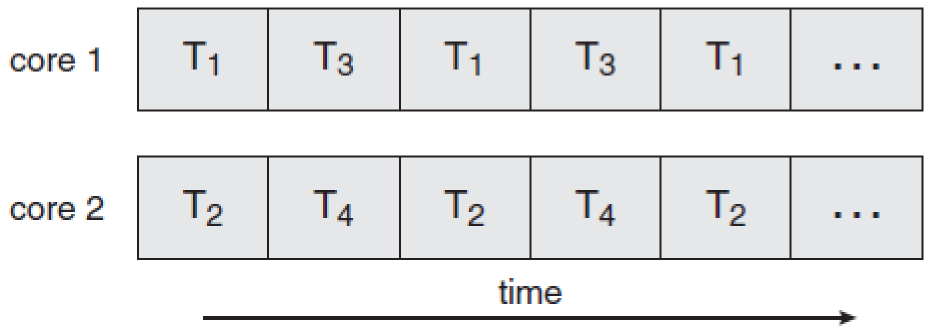
Concurrency vs. Parallelism

- Concurrent execution on a single-core system:



As long as T1, T2, T3 and T4 have not completed execution their executions are overlapped and interleaved on a single core.

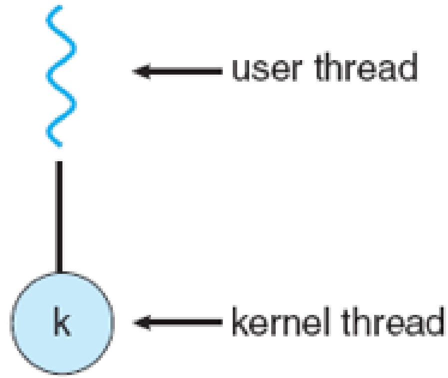
- Parallel execution on a multicore system:



There may be concurrent execution on a single core.
Note that each core runs a different set of processes.

Multithreading Models

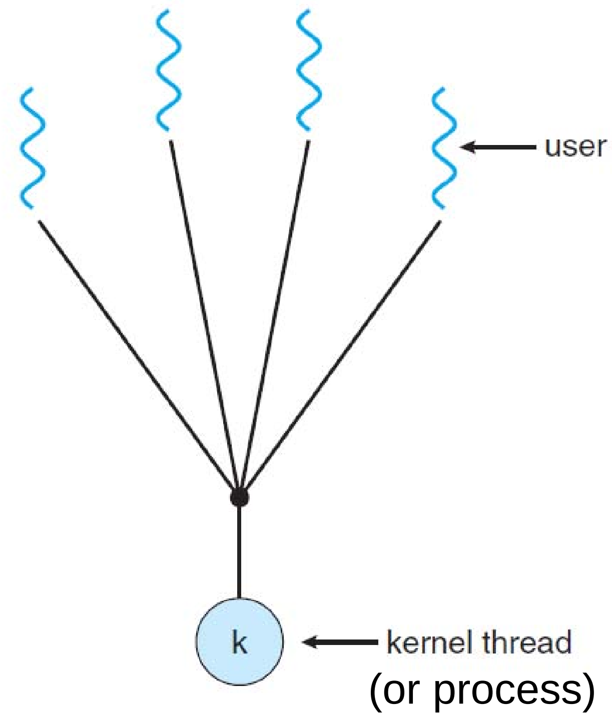
- Multithreading models
  - Many-to-One
  - One-to-One
  - Many-to-Many



- User threads vs. Kernel threads
  - User threads
    - Management done by user-level threads library.
    - The kernel is not aware of user threads.
    - Major thread libraries: POSIX Pthreads, Win32 threads
  - Kernel threads
    - Supported by the kernel – The kernel is aware of threads
    - Scheduled and maintained by the kernel
    - Examples – virtually all general purpose operating systems, including Windows, Solaris, Linux, Tru64 UNIX, Mac OS X

Many-to-One

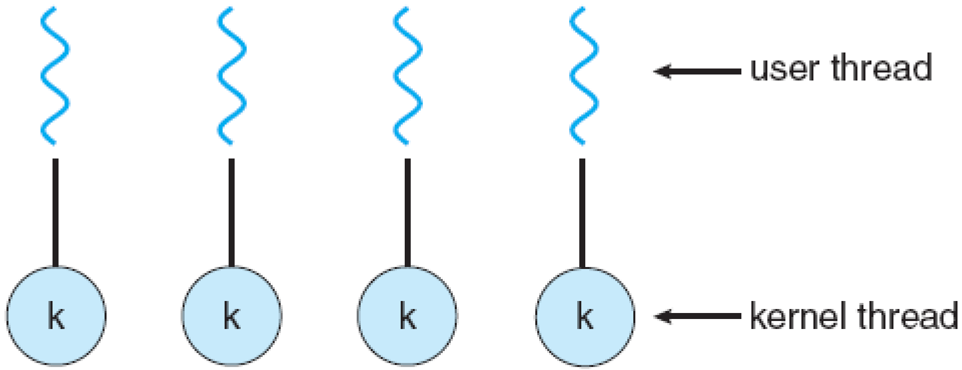
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



One-to-One

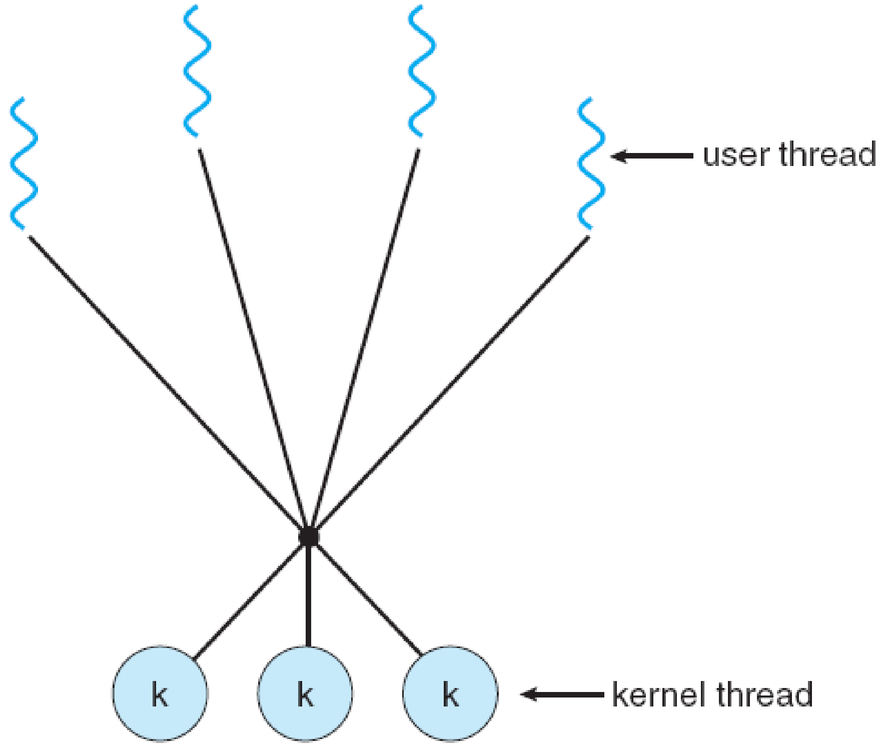
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



Many-to-Many Model

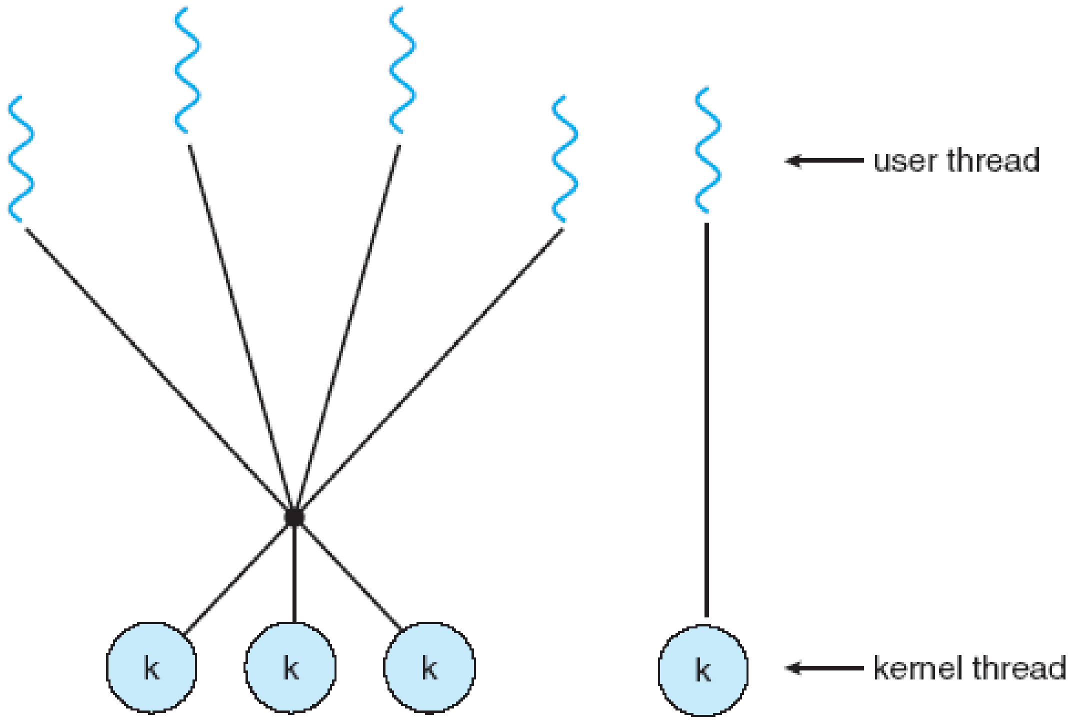
- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows NT/2000 with the *ThreadFiber* package



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



Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads

- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Pthreads Example

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

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */

int main(int argc, char *argv[])
{
   pthread_t tid; /* the thread identifier */
   pthread_attr_t attr; /* set of thread attributes */

   if (argc != 2) {
      fprintf(stderr,"usage: a.out <integer value>\n");
      return -1;
   }
   if (atoi(argv[1]) < 0) {
      fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
      return -1;
   }

   /* get the default attributes */
   pthread_attr_init(&attr);
   /* create the thread */
   pthread_create(&tid,&attr,runner,argv[1]);
   /* wait for the thread to exit */
   pthread_join(tid,NULL);

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

Pthreads Example (Cont.)

```
/* The thread will begin control in this function */
void *runner(void *param)
{
   int i, upper = atoi(param);
   sum = 0;

   for (i = 1; i <= upper; i++)
      sum += i;

   pthread_exit(0);
}
```

Two threads are created:
1. `main()` 2. `runner()`

Figure 4.9 Multithreaded C program using the Pthreads API.

Pthreads Code for Joining 10 Threads

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Figure 4.10 Pthread code for joining ten threads.

Win32 API Multithreaded C Program

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

/* the thread runs in this separate function */
DWORD WINAPI Summation(LPVOID Param)
{
   DWORD Upper = *(DWORD*)Param;
   for (DWORD i = 0; i <= Upper; i++)
      Sum += i;
   return 0;
}

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

   if (argc != 2) {
      fprintf(stderr,"An integer parameter is required\n");
      return -1;
   }
   Param = atoi(argv[1]);
   if (Param < 0) {
      fprintf(stderr,"An integer >= 0 is required\n");
      return -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 */

   if (ThreadHandle != NULL) {
      /* now wait for the thread to finish */
      WaitForSingleObject(ThreadHandle,INFINITE);

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

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

Figure 4.11 Multithreaded C program using the Windows API.

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch

Thread Pools

- Create a number of threads in a pool where they await work, called "prethreading"

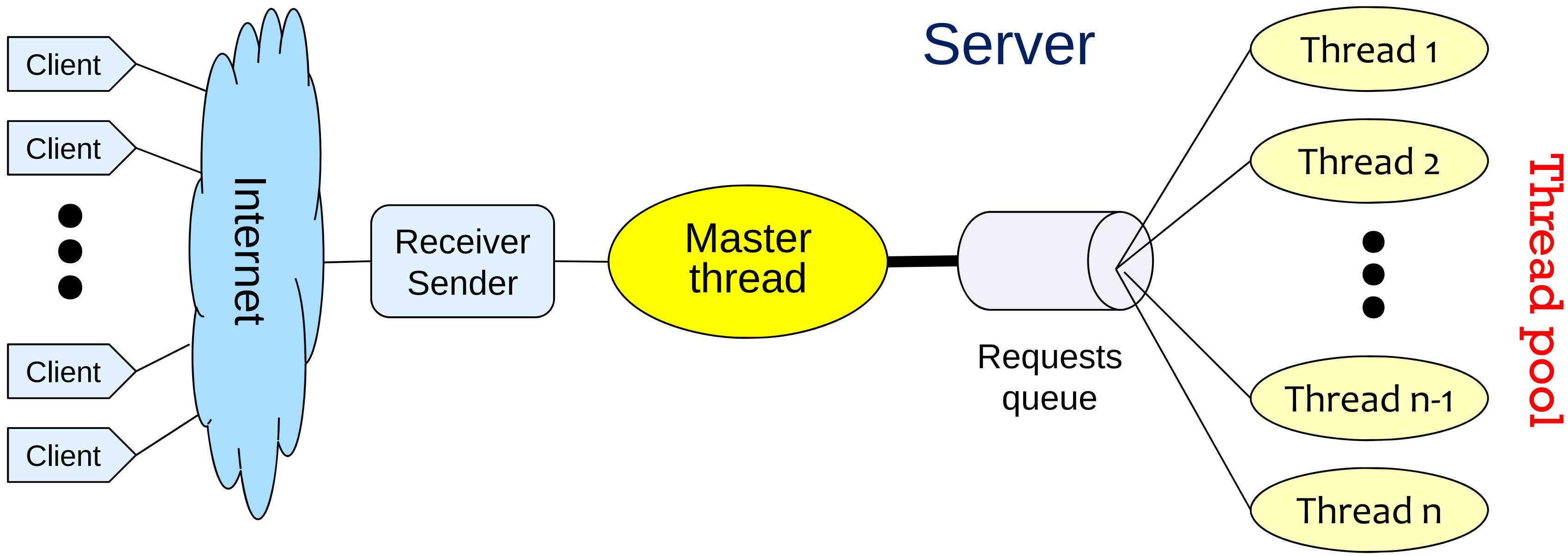
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e.Tasks could be scheduled to run periodically

- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {
   /*
    * this function runs as a separate thread.
   */
}
```

Thread Pools – An Example

A thread pool in Internet-based client-server systems




```
/* Create n worker threads */
    for (i = 0; i < n; i++)
      pthread_create(&tid, NULL, thread, NULL);

/* Send requests to a thread pool */
    while (1) {
      /* Pointers to the tasks to be executed
          are inserted into the queue */
    }
```

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** - blocks of code that can run in parallel
- Specialized for parallel programs in parallel processing environment

```
#pragma omp parallel
```

Create as many threads as there are cores

```
#pragma omp parallel for
  for(i=0;i<N;i++) {
    c[i] = a[i] + b[i];
}
```

Run for loop in parallel

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

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in "^{ }" - `^{ printf("I am a block"); }`
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue
- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, queue is per process, called **main queue**
    - Programmers can create additional serial queues within program
  - concurrent – removed in FIFO order but several may be removed at a time
    - Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
  (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

Threading Issues

- Semantics of **`fork()`** and **`exec()`** system calls
- Signal handling
  - Synchronous and asynchronous
- Thread cancellation of target thread
  - Asynchronous or deferred
- Thread-local storage
- Scheduler Activations

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

Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
  - a communication mechanism between the kernel and processes
- 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

Signal Handling (Cont.)

- 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

- 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

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 should 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

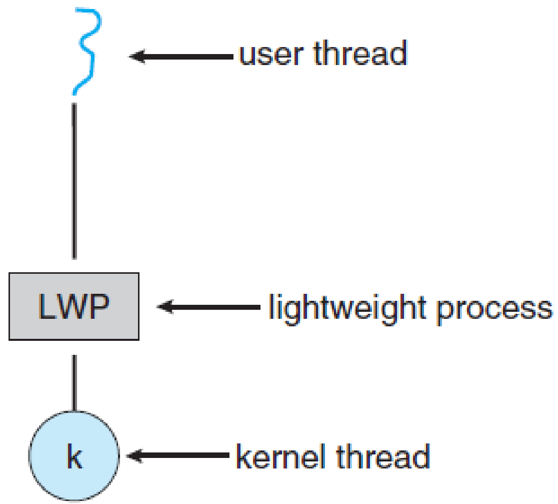
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is *deferred*
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. `pthread_testcancel()`
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation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

Thread-Local Storage

- **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

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Typically use an intermediate data structure between user and kernel threads – **lightweight process** (**LWP**) [Solaris]
  - Appears to be a virtual processor on which process can schedule user thread to run
  - Each LWP attached to kernel thread
  - How many LWPs to create?
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the **upcall handler** in the thread library
- This communication allows an application to maintain the correct number of kernel threads



OS Example – Windows Threads

- Windows implements the Windows API – primary API for Win 98, Win NT, Win 2000, Win XP, and Win 7

- Implements the one-to-one mapping, kernel-level

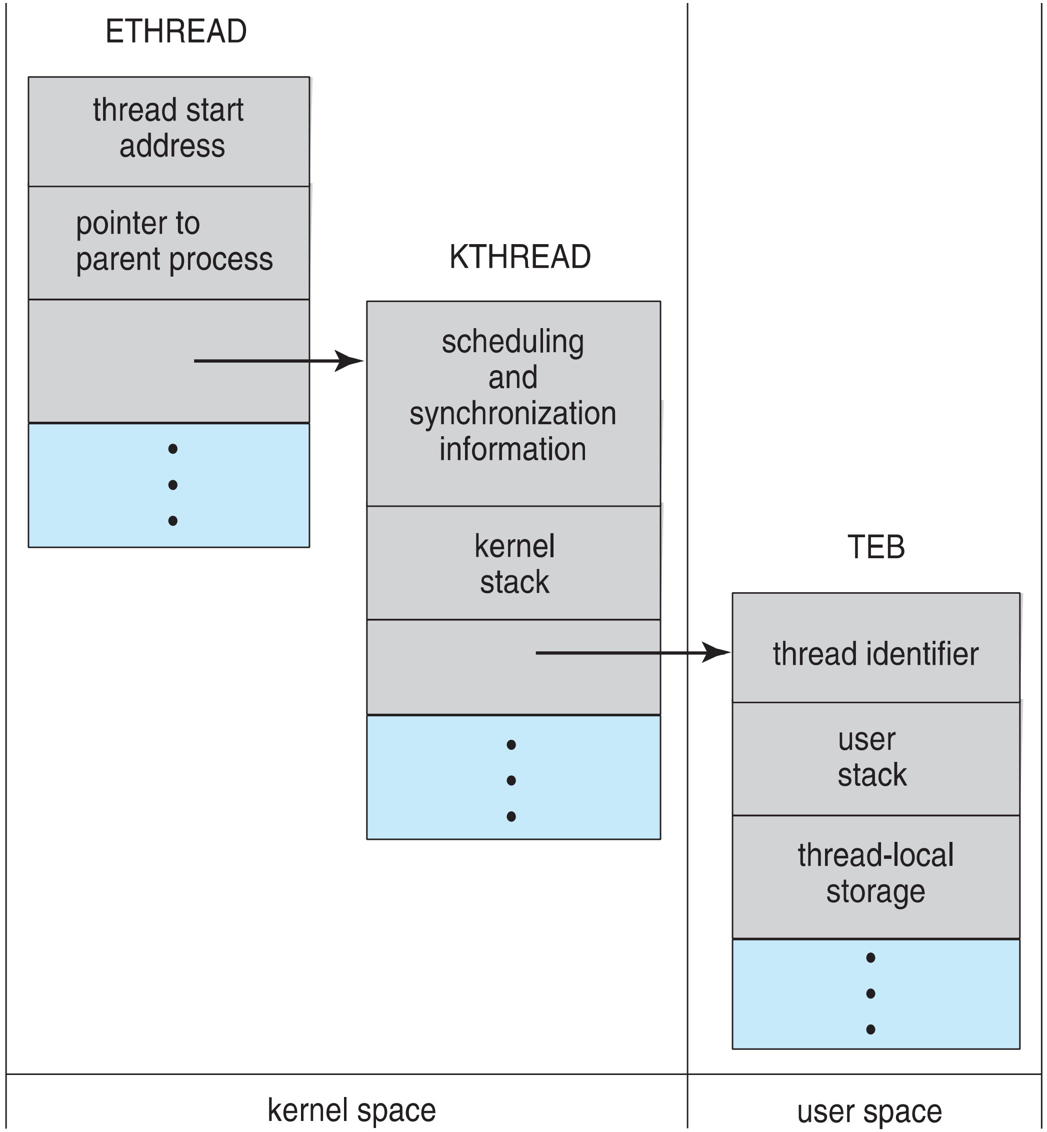
- Each thread contains
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)

Windows Threads (Cont.)

- The register set, stacks, and private storage area are known as the **context** of the thread

- The primary data structures of a thread include:
  - ETHREAD (executive thread block) – includes pointer to process to which thread belongs and to KTHREAD, in kernel space
  - KTHREAD (kernel thread block) – scheduling and synchronization info, kernel-mode stack, pointer to TEB, in kernel space
  - TEB (thread environment block) – thread id, user-mode stack, thread-local storage, in user space

Windows XP Threads Data Structures



OS Example - Linux Threads

- Linux refers to them as ***tasks*** rather than ***threads***
  - Linux does not recognize a distinction between threads and processes (or tasks)
  - User-level threads are mapped onto kernel processes
- **`clone()`** system call
  - Creates a process or task, which may behave as thread
  - Copies the attributes of the current process
  - Allocates separate stack for a new process
  - Allows a child process to share the address space of the parent process
  - The argument "flags" specifies what is shared between the calling process and the child process (next slide)
- **`task_struct`** data structure represents a process or task
  - It points to process/task data structures (shared or unique)

Linux Threads (Cont.)

`clone()` flags – *partial list*

Flags	Meaning
CLONE_CHILD_CLEARTID	Erase child thread ID at location *ctid* in child memory when the child exits
CLONE_FILES	the calling process and the child process share the same file descriptor table
CLONE_FS	the caller and the child process share the same filesystem information
CLONE_IO	the new process shares an I/O context with the calling process
CLONE_NEWPID	create the process in a new PID namespace
CLONE_PARENT	the parent of the new child (as returned by `getpid()`) will be the same as that of the calling process
CLONE_SETTLS	The *newtls* argument is the new TLS (Thread Local Storage) descriptor
CLONE_SIGHAND	the calling process and the child process share the same table of signal handlers
CLONE_STOPPED	the child is initially stopped (as though it was sent a SIGSTOP signal), and must be resumed by sending it a SIGCONT signal
CLONE_THREAD	the child is placed in the same thread group as the calling process
CLONE_VFORK	the execution of the calling process is suspended until the child releases its virtual memory resources via a call to `execve()` or `exit()`
CLONE_VM	the calling process and the child process run in the same memory space

http://man7.org/linux/man-pages/man2/clone.2.html

Summary

- 쓰레드(thread)란 어떤 한 프로세스의 실행환경에서 코드가 실행되는 과정 혹은 그 주체를 일컬음
- 멀티쓰레딩(multithreading)은 한 프로세스에서 복수의 쓰레드가 공유하는 코드를 실행함을 의미함. 그러면, 그 이점은?
- 쓰레드와 프로세스의 구조적 차이점
- 사용자 vs. 커널 쓰레드
- POSIX *Pthreads*
- 쓰레드 풀 (thread pools)
- 묵시적 쓰레딩(implicit threading)
  - 프로그래머가 직접 쓰레드를 생성/관리하지 않고 컴파일러나 런타임 라이브러리가 하게 함
- 쓰레드 사용시 문제점
  - fork(), exec(), 시그널 처리문제, 목표 쓰레드의 취소, 쓰레드에 한정된 데이터, 스케줄러의 활성화
- 쓰레드 구현 사례: Windows, Linux

- A thread refers to an entity or procedure that runs the program in a process execution environment.
- Multithreading means that multiple threads execute shared code in the process context. Then, what are the benefits?
- Structural difference between thread and process
- User-level vs. kernel-level thread
- POSIX *Pthreads*
- Thread pools
- Implicit threading
  - Responsibility of creation and management of threading is transferred to compilers and run-time libraries
- Threading issues
  - fork(), exec(), signal handling, thread cancellation of target thread, thread-local storage, scheduler activation
- OS examples: Windows, Linux