Chapter 4: Threads

Chapter 4: Threads

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

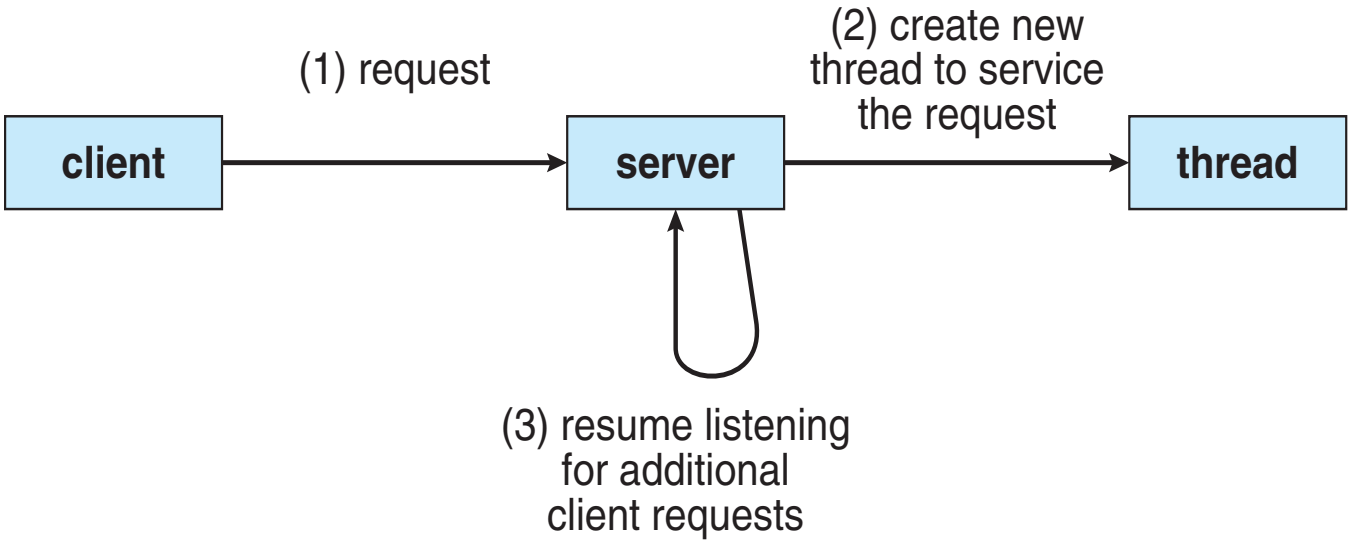
Objectives

- **To introduce the notion of a thread**—a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- **To discuss the APIs for the Pthreads, Windows, and Java thread libraries**
- **To explore several strategies that provide implicit threading**
- **To examine issues related to multithreaded programming**
- **To cover operating system support for threads in Windows and Linux**

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

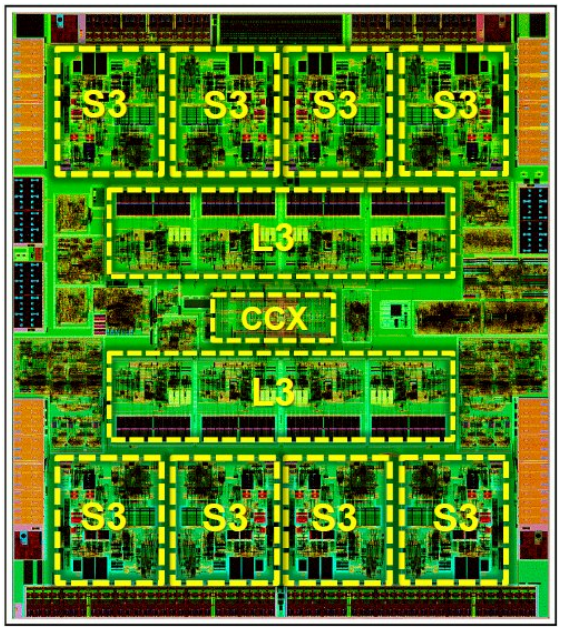
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability – A single-threaded** process can can run on only one processor, regardless how many processors are available

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.)

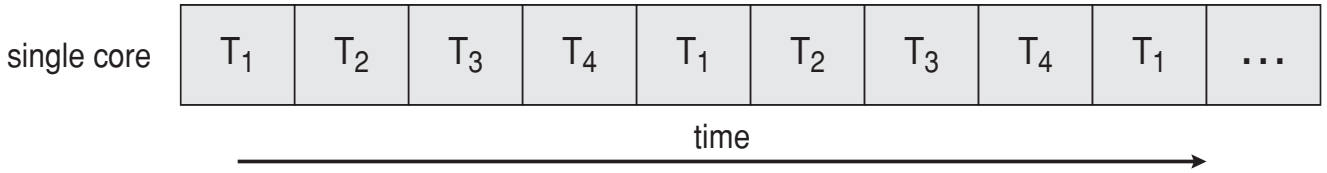
- **Types of parallelism**
  - **Data parallelism –** distributes subsets of the same data across multiple cores, same operation on each
  - **Task parallelism –** distributing threads across cores, each thread performing unique operation
- **As # of threads grows, so does architectural support for threading**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



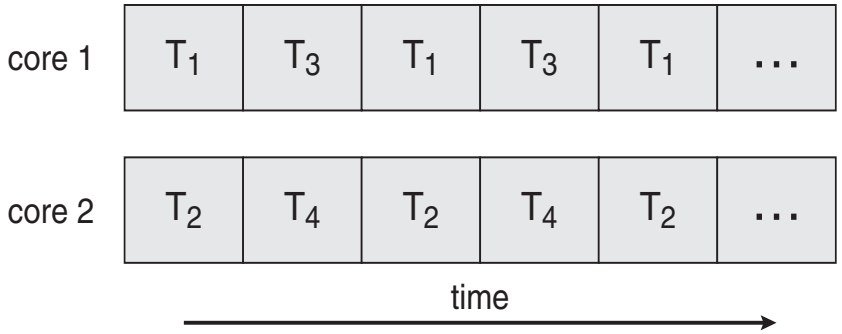
- **8개의 S3 core가짐**
- **S3 core:**
- Out-of-order, Dual-issue
- Dynamically threaded
- Balanced pipeline design
- Single thread performance :
  - Estimate ~5X S2's SPECint2006* performance
  - Estimate ~7X S2's SPECfp2006* performance
- Throughput performance : ~2X S2's per thread throughput performance
- High frequency, deep pipeline: 16 stage integer pipe, 3+ GHz

Concurrency vs. Parallelism

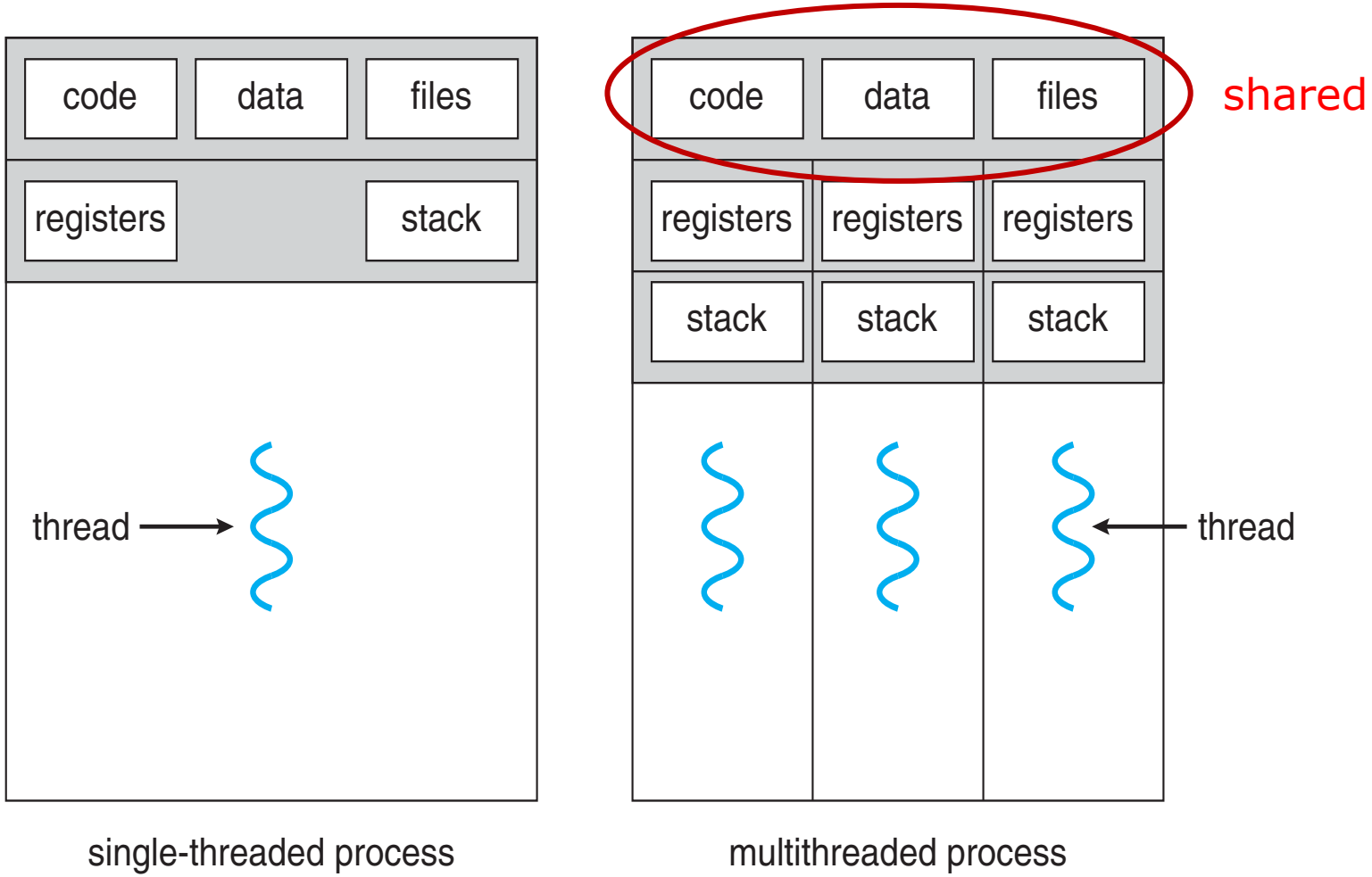
- **Concurrent execution on single-core system:**
  - 4개 task의 동시(concurrent) 수행



- **Parallelism on a multi-core system:**
  - 2개의 core 병렬성 및 4개 task concurrency 제공



Single and Multithreaded Processes



Amdahl's Law

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

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

- **That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times**
- **As *N* approaches infinity, speedup approaches 1 / *S***

Serial portion of an application has disproportionate effect on performance gained by adding additional cores

- **But does the law take into account contemporary multicore systems?**

User Threads and Kernel Threads

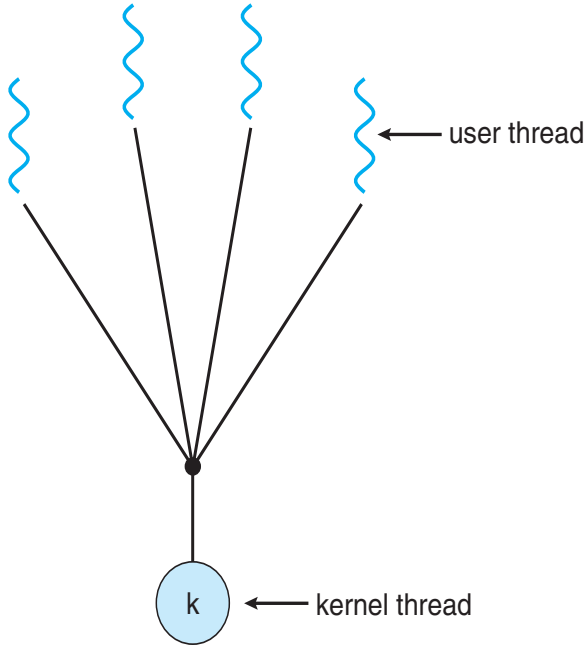
- 사용자 스레드와 커널 스레드
- **User threads - management done by user-level threads library**
- **Three primary thread libraries:**
  - POSIX Pthreads
  - Windows threads
  - Java threads
- **Kernel threads - Supported by the Kernel**
- **Examples – virtually all general purpose operating systems, including:**
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

- 사용자 스레드와 커널스레드는 어떤 연관 관계 존재함
  - Many-to-One
  - One-to-One
  - Many-to-Many

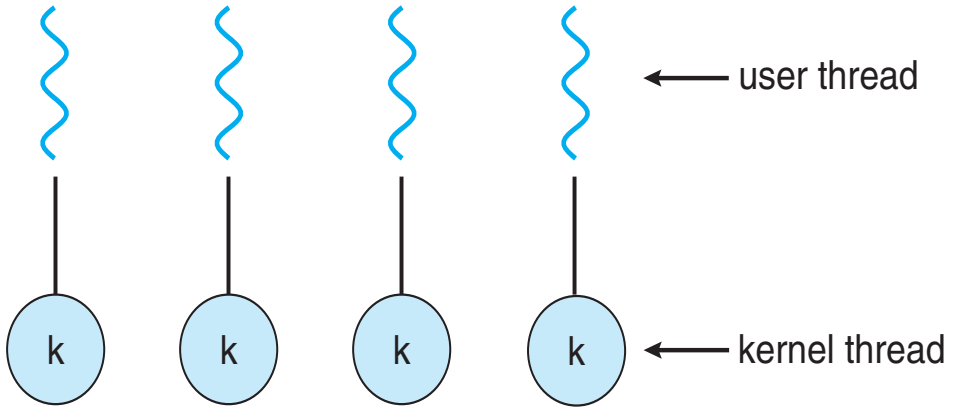
Many-to-One(다대일 모델)

- Many user-level threads mapped to single kernel thread
- 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
- Examples:
  - Solaris Green Threads
  - GNU Portable Threads



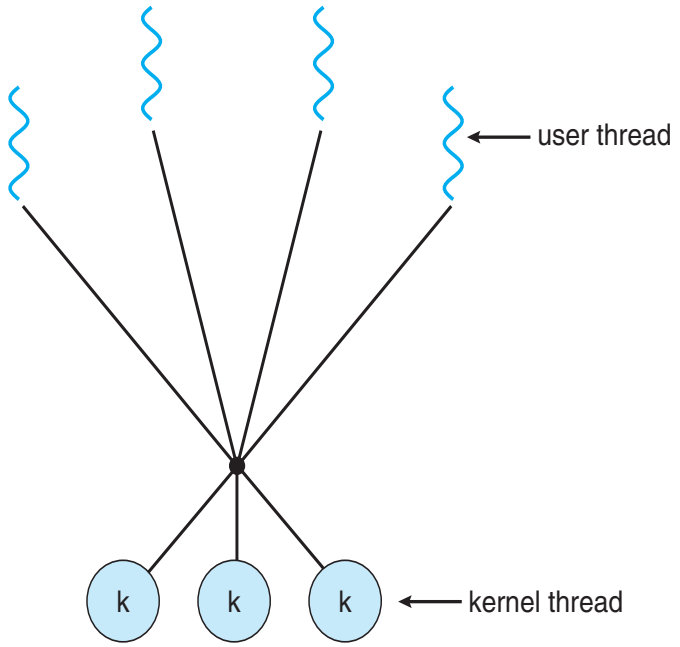
One-to-One (일대일 모델)

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
  - 이에, 사용자가 너무 많은 thread를 생성하지 않도록 주의할 필요 있음
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



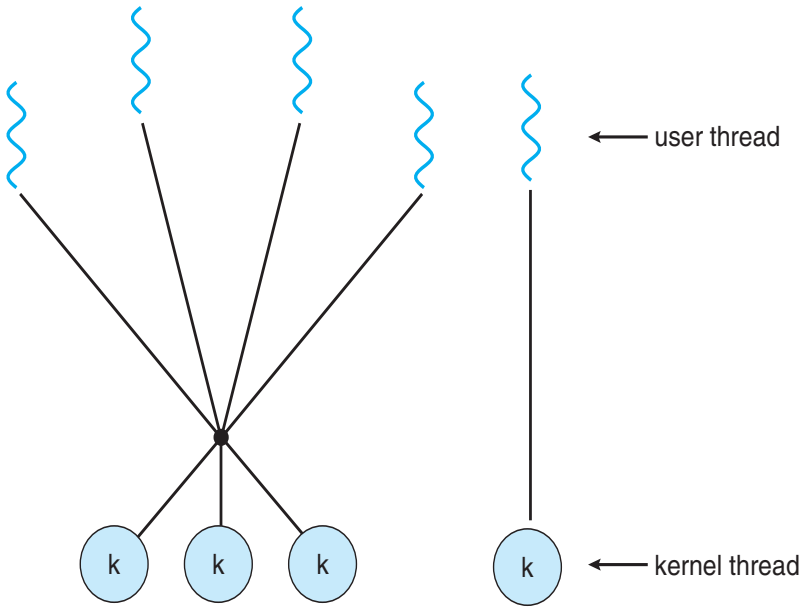
Many-to-Many Model (다대다 모델)

- 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 with the *ThreadFiber* package



Two-level Model (두 수준 모델)

- Similar to M:M, except that it allows a user thread to be bound to kernel thread
- 즉, 많은 사용자 thread를 사용자 thread 개수 이하의 kernel thread로 multiplexing 하고, 동시에 user thread 가 하나의 kernel thread에만 연관되는 것을 허용함
- 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 only in user space**
    - 이에, thread library 호출시, system 호출이 아니라, 사용자 공간 지역 함수 호출을 의미함
  - **Kernel-level library supported by the OS**
    - 이에, thread library 호출은 kernel system call을 의미함

- **POSIX Pthreads(user level or kernel level), Windows thread(kernel level), Java thread가 많이 사용됨**
  - Pthreads는 user level 혹은 kernel level로 제공될 수 있음
  - Windows thread library는 Windows kernel level library일 것임
  - Windows 상에서의 Java thread라고 함은 결국 JVM이 Windows 상에서 동작하므로 결국 Windows API 사용함을 의미
  - UNIX, Linux는 주로 Pthreads 사용

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

$$sum = \sum_{i=0}^{N} i$$

```
#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;
   }
```

thread간 공유하는 변수: 전역변수로 선언됨

```
   /* 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);
}

/* 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);
}
```

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

Compile and link with *-pthread*.

Description

The **pthread_create**() function starts a new thread in the calling process. The new thread starts execution by invoking *start_routine*(); *arg* is passed as the sole argument of *start_routine*().

```
% gcc test.c -lpthread
```

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);
```

Windows Threads

- The technique for creating threads using the Windows thread library is similar to the Pthreads technique in several ways
- For using the Windows threads, we should include the header file, “windows.h”
- An example of C program using Windows API…

Windows Multithreaded C Program

```
#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;
   }
```

thread간 공유하는 변수: 전역변수로 선언됨

각 thread에서 실행될 함수

Thread 생성(여러 attribute이 전달됨)

```
   /* 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);
   }
}
```

Java Threads

- In Java, when we execute application,
  - the JVM creates a thread object whose task is defined by the main( ) method
  - it starts a thread…
  - That is, in a Java program, it turns out that every Java program has more than one thread…즉, Thread는 Java에서 program실행의 근본모델.
- Java threads are managed by the JVM
- Typically implemented using the threads model provided by underlying OS
- Java threads may be created by:

```
public interface Runnable
{
    public abstract void run();
}
```

  - Extending Thread class
  - Implementing the Runnable interface

Java Multithreaded Program

```
class Sum
{
   private int sum;

   public int getSum() {
     return sum;
   }

   public void setSum(int sum) {
     this.sum = sum;
   }
}

class Summation implements Runnable
{
   private int upper;
   private Sum sumValue;

   public Summation(int upper, Sum sumValue) {
     this.upper = upper;
     this.sumValue = sumValue;
   }

   public void run() {
     int sum = 0;
     for (int i = 0; i <= upper; i++)
        sum += i;
     sumValue.setSum(sum);
   }
}
```

Java Multithreaded Program (Cont.)

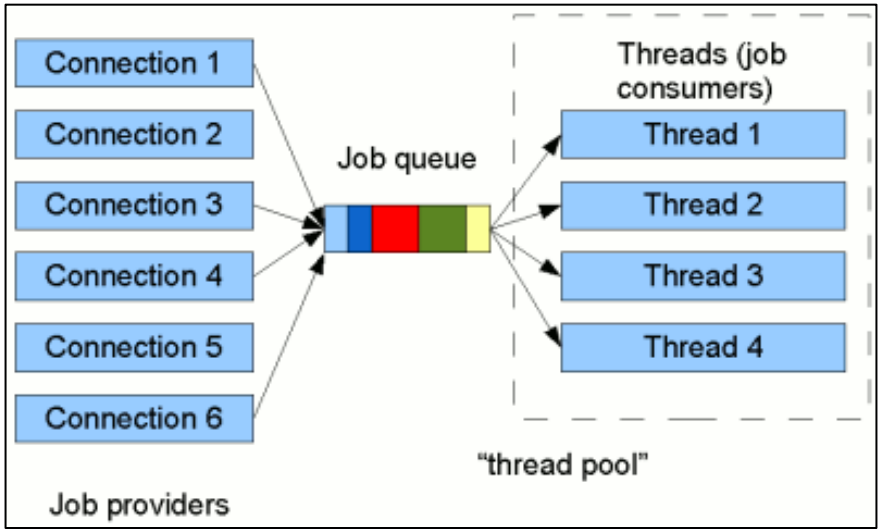
```
public class Driver
{
   public static void main(String[] args) {
     if (args.length > 0) {
       if (Integer.parseInt(args[0]) < 0)
         System.err.println(args[0] + " must be >= 0.");
       else {
         Sum sumObject = new Sum();
         int upper = Integer.parseInt(args[0]);
         Thread thrd = new Thread(new Summation(upper, sumObject));
         thrd.start();
         try {
            thrd.join();
            System.out.println
                    ("The sum of "+upper+" is "+sumObject.getSum());
       } catch (InterruptedException ie) { }
       }
     }
     else
       System.err.println("Usage: Summation <integer value>"); }
}
```

Implicit Threading

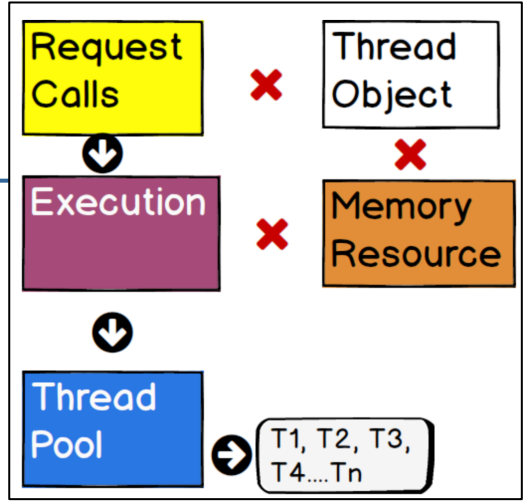
- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- 암묵적 스레딩 기반 (다중코어 처리기를 활용하는) 다중 스레드 프로그램을 만드는 세가지 방법
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch (Mac OS X and iOS)
- Other methods include Microsoft Threading Building Blocks (TBB), `java.util.concurrent` package

Thread Pools

- Create a number of threads at start-up & place them into a pool, where they sit and wait for 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



Thread Pools



- Windows supports thread pools:

- Steps to Create Thread Pooling in C#.NET

1. We need to use **Threading** namespace as shown below:

   using System.**Threading**;

< Thread pool life cycle in Windows >

2. After using **Threading** namespace, we need to call **ThreadPool** class. Using **ThreadPool** object we call method, "QueueUserWorkItem" - which queues a function for an execution and the function executes when a thread becomes available from thread pool.

   ThreadPool.QueueUserWorkItem takes overloaded method called WaitCallback, which represents a callback function to be executed by a **ThreadPool** thread. If no thread is available, it will wait until one becomes free.

   ThreadPool.QueueUserWorkItem(new WaitCallback(Run));

   Here Run( ) is the method to be executed by a thread in **ThreadPool**.

http://www.learncsharptutorial.com/threadpooling-csharp-example.php

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

```
#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

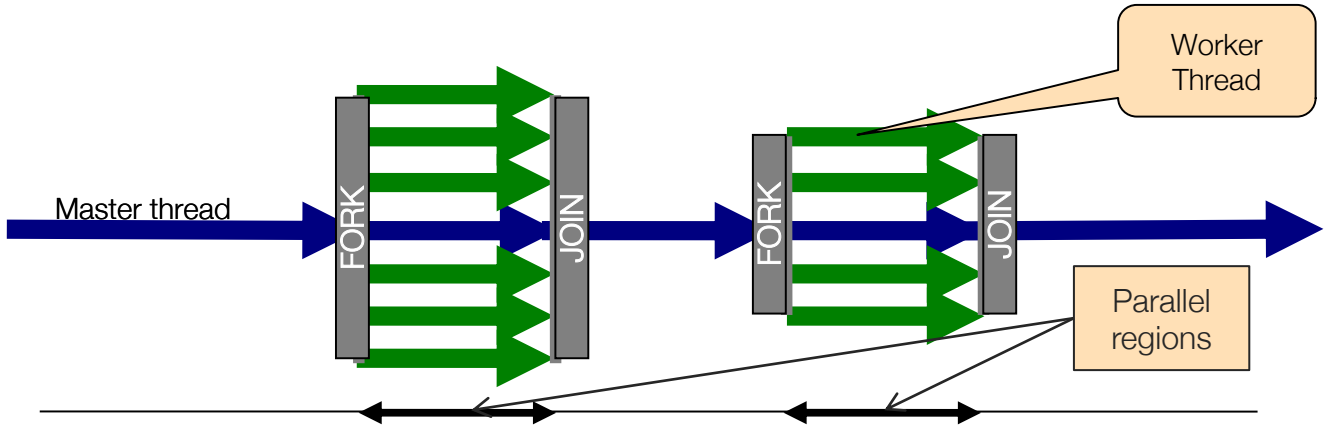
```
#include <omp.h>
#include <stdio.h>

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

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

   /* sequential code */

   return 0;
}
```

OpenMP

- **Motivation of OpenMP**
- **Multicore CPUs are everywhere:**
  - Servers with over 100 cores today
  - Even smartphone CPUs have 8 cores
- **Multithreading, natural programming model**
  - All processors share the same memory
  - Threads in a process see same address space
  - Many shared-memory algorithms developed
- **But…Multithreading is hard**
  - Lots of expertise necessary
  - Deadlocks and race conditions
  - Non-deterministic behavior makes it hard to debug

OpenMP

- **So.. OpenMP..**
- **A language extension with constructs for parallel programming:**
  - Critical sections, atomic access, private variables, barriers
- **Parallelization is orthogonal to functionality**
  - If the compiler does not recognize OpenMP directives, the code remains functional (albeit single-threaded)
- **Industry standard: supported by Intel, Microsoft, IBM, HP**
- **OpenMP execution model:**
  - Fork and Join: Master thread spawns a team of threads as needed



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"); }`
- GCD(Grand Central Dispatch) schedules tasks for run-time execution by placing them on a dispatch queue
  - There are two types of dispatch queue: serial, concurrent queue
  - When it removes a task from a queue, it assigns the task to an available thread from a pool of threads that it manages

Grand Central Dispatch

- **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

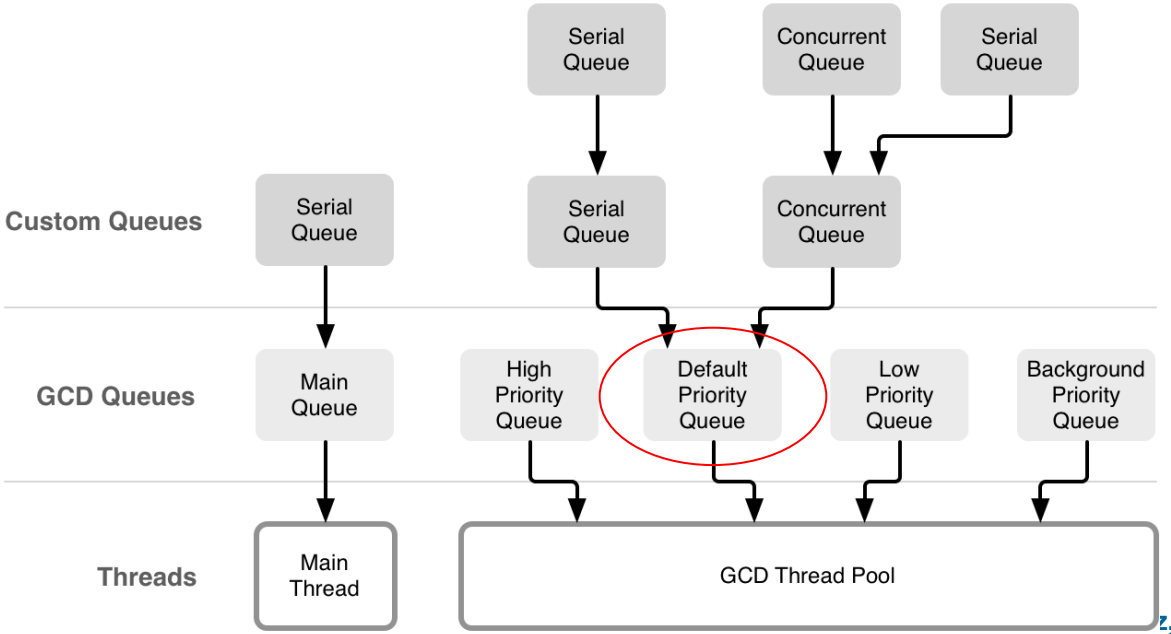
```
dispatch_queue_t queue = dispatch_get_global_queue
  (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

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

Grand Central Dispatch

- **GCD was introduced in OS X 10.6 & iOS 4 in order to make it easier for developers to take advantage of the increasing number of CPU cores.**
  - with GCD, you don't interact with threads directly anymore, Instead you add blocks of code to queues, and GCD manages a thread pool behind the scenes.
  - GCD decides on which particular thread your code blocks are going to be executed on, and it manages these threads according to the available system resource.
  - This alleviates the proble of too many threads being created, because the threads are now centrally managed and abstracted away from application developers.
  - The other important change with GCD is that you as a developer think about work items in a queue rather than threads. This new mental model of concurrency is easier to work with

Grand Central Dispatch

- **GCD exposes 5 different queues:**
  - main queue running on the main thread, three background queues with different priorities, and one background queue with an even lower priority, which is I/O throttled.
- Making use of several queues with different priorities sounds pretty straightforward at first.
  - However, we strongly recommend that you use the default priority queue in almost all cases. Scheduling tasks on queues with different priorities can quickly result in unexpected behavior if these tasks access shared resources.



Issues in designing multi-threading programs

- 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?
- The semantics of the fork() and exec() system calls change in a multithreaded program.
- If one thread in a program calls fork(), **does the new process duplicate all threads, or is the new process single-threaded?**
- Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call.
  - Some UNIXes have two versions of fork
  - 모든 thread를 복사하는 것도 있고 fork()를 호출한 thread 만 복제하는 것도 있음
- **`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 signal handler is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
     1. default
     2. 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

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
  - Thread 취소에서 어려운 문제는 "thread에 할당된 자원 문제때문임". OS는 취소된 thread로부터 자원을 회수하는 경우도 있지만, 회수 못하는 경우도 있음 → 즉, 비동기식 thread 취소는 시스템 자원을 모두 사용 가능한 상태로 만들지 못하는 경우 있음
- 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

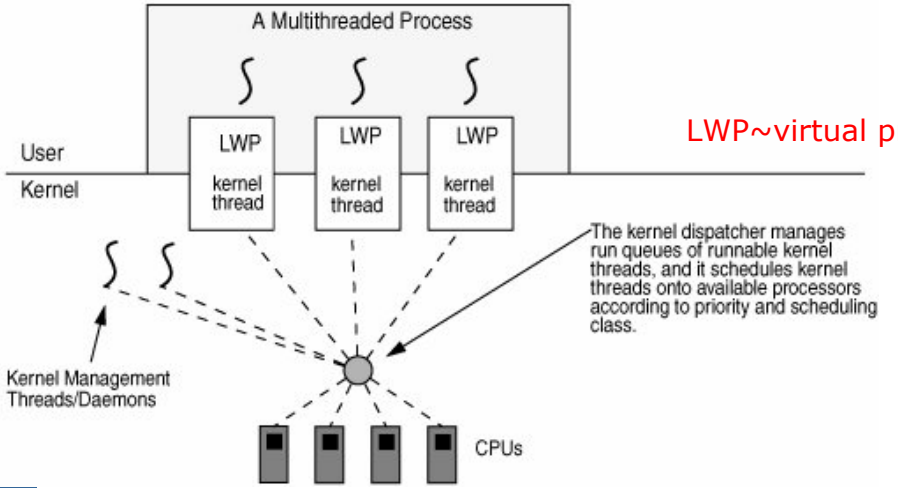
- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches cancellation point
    - I.e. `pthread_testcancel()`
    - Then cleanup handler is invoked
- On Linux systems, thread cancellation is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

Thread-Local Storage

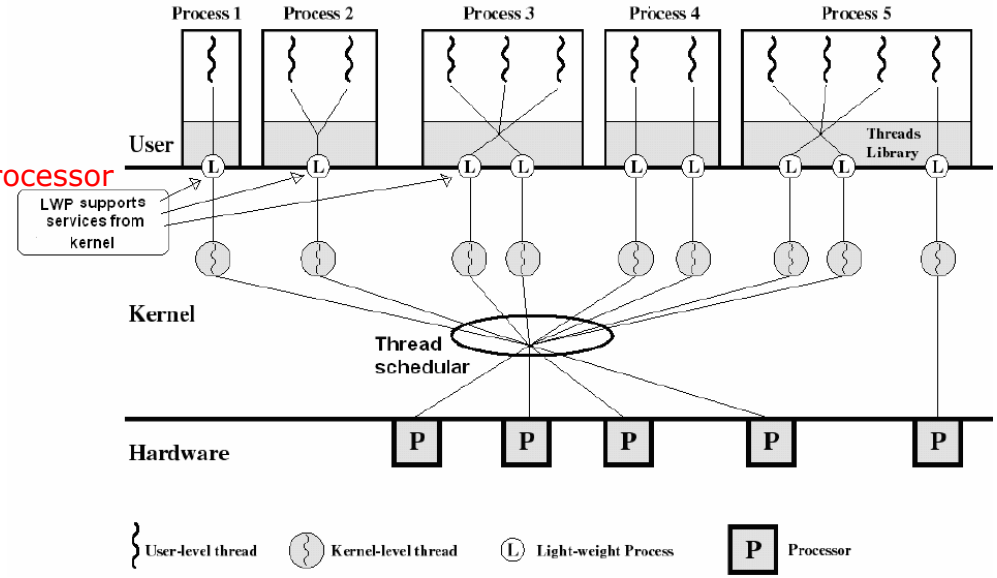
- 동일 proces에 속하는 thread 는 기본적으로 데이터를 모두 공유함. 하지만, 상황에 따라 각 thread는 자기만 access하는 데이터 가질 필요 있음 → Thread 국지 저장소 !
- Thread-local storage (TLS) allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different 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

LWP

- **LWP(Lightweight Process)**
  - A lightweight process is defined as a data structure which is located between user threads and kernel threads.
  - User-thread library considers each lightweight process as a virtual processor on which an application can schedule a user thread to run
  - Each lightweight process is attached to one kernel thread



LWP~virtual processor

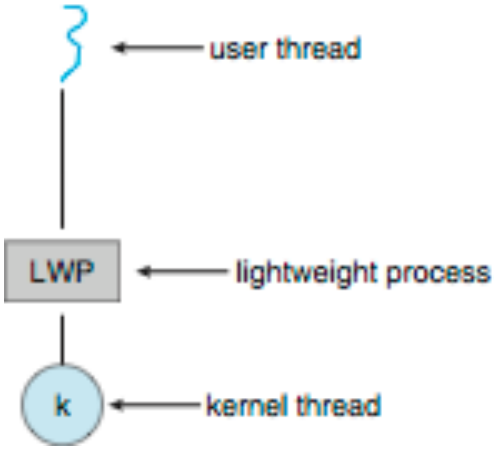


https://flylib.com/books/en/2.830.1.14/1/

<Solaris LWPs>

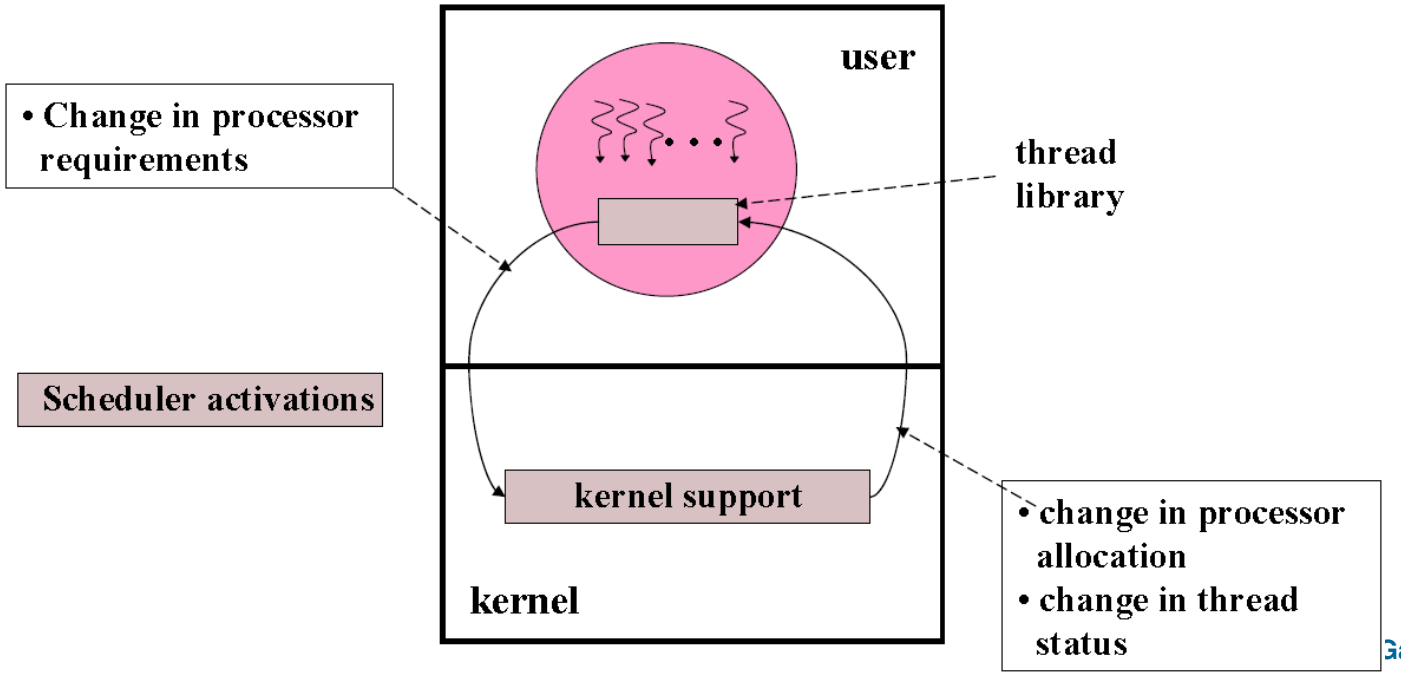
Scheduler Activations

- 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)
  - 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 kernel threads



Scheduler Activations

- **Problems:**
  - Application has knowledge of the user-level thread state but has little knowledge of or influence over critical kernel-level events
  - Kernel has inadequate knowledge of user-level thread state to make optimal scheduling decisions
- **Solutions :**
  - a mechanism is necessary which facilitates exchange of information between user-level and kernel-level space. → Scheduler Activation Mechanism



Scheduler Activations

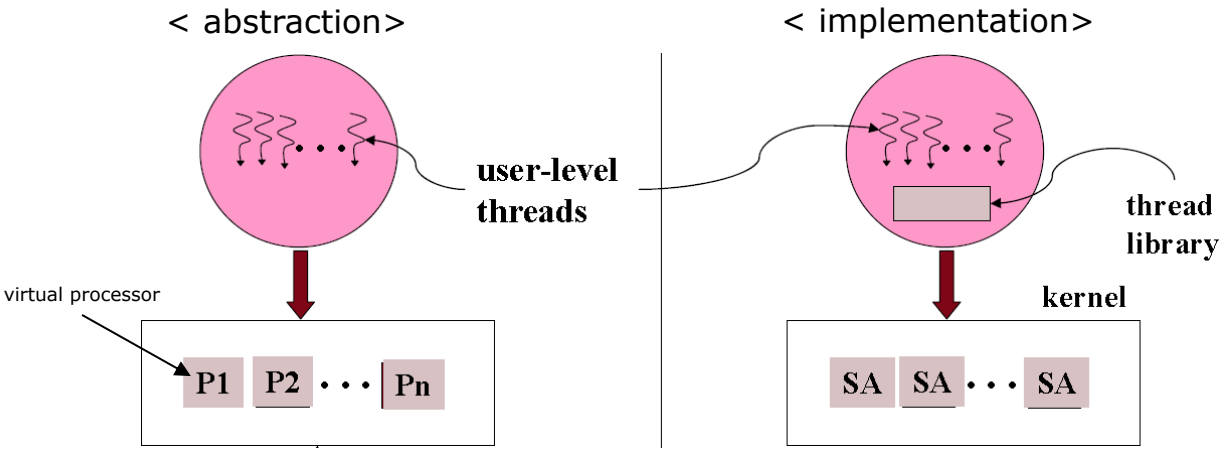
- **Scheduler activation** : A mechanism for kernel & user level to cooperate.
  - Kernel makes an **upcall** with **a scheduler activation**
  - You can either keep the activation, or perform the task informed by the activation.
  - User level → kernel is still **system call**

- It works as follows:
  - The kernel provides an application with a set of virtual processors (LWPs),
    - So, the application can schedule user threads onto an available virtual processor.
  - Furthermore, the kernel must inform an application about certain events.
  - This procedure is known as an upcall. Upcalls are handled by the thread library with an upcall handler, and upcall handlers must run on a virtual processor.

Communication via Upcalls

- The kernel-level scheduler activation mechanism communicates with the user-level thread library by a set of upcalls:

Add this processor (processor #)
Processor has been preempted (preempted activation #, machine state)
Scheduler activation has blocked (blocked activation #)
Scheduler activation has unblocked (unblocked activation #, machine state)



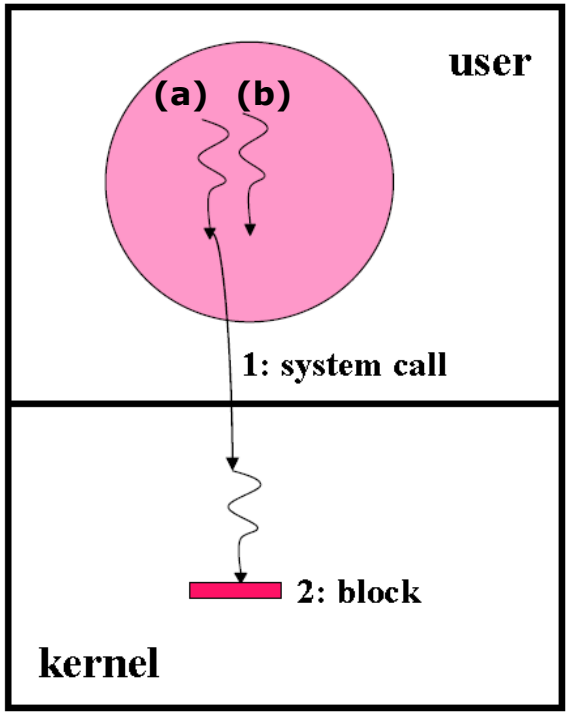
The **thread library** must maintain the association between a thread's identity and thread's scheduler activation number.

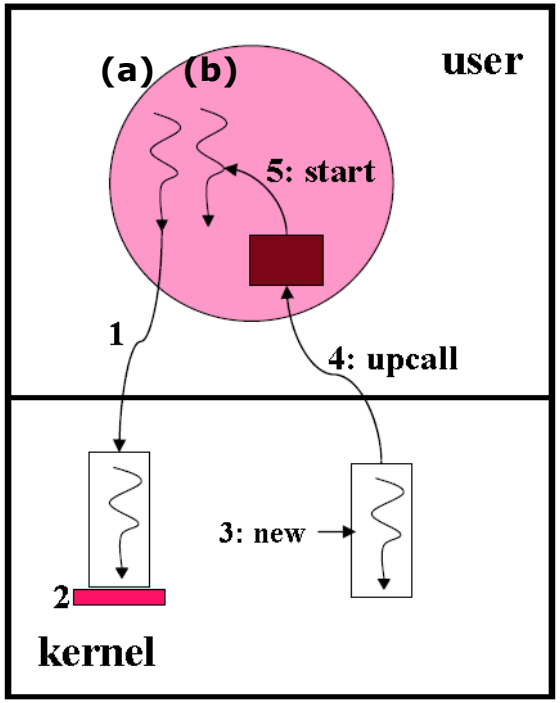
there is **one running scheduler activation (SA)** for each processor assigned to the user process.

Role of Scheduler ActivationsUpcalls

- Scheduler Activation 동작 예

1. user level thread (a)에서 I/O 동작을 한다고 가정함
2. LWP가 할당되어 system call되지만, kernel thread에서 block 됨



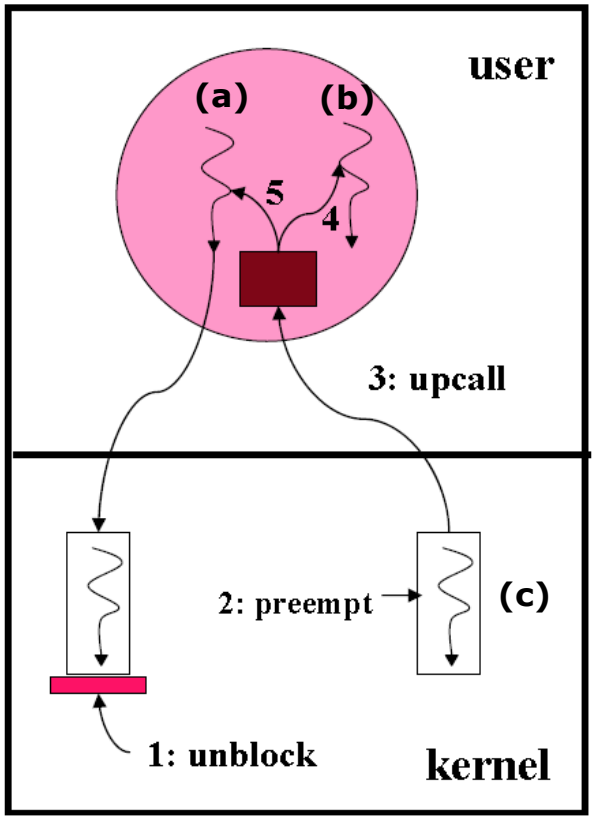


3. (Scheduler Activation 메커니즘 동작함)
   Kernel에서는 새로운 thread 생성되어,
4. upcall 이뤄짐
5. 대신, 다른 user level thread (b)를
   동작시킴.

Role of Scheduler ActivationsUpcalls

- Unblocking 되는 경우..

1. Block된 kernel thread가 풀리는 경우,
2. 대신 실행되던 user thread (b)에 대응되는 kernel thread (c)가 선점됨(user thread (a)실행위해)
3. SA 메커니즘이 동작되어, 이러한 정보가 upcall 됨.
4. user thread (a)가 block되어 대신 실행되던 user thread (b)를 선점(preemption) 시키고
5. 다시 (a)를 실행시킴 !!



Operating System Examples

- Windows Threads
- Linux Threads

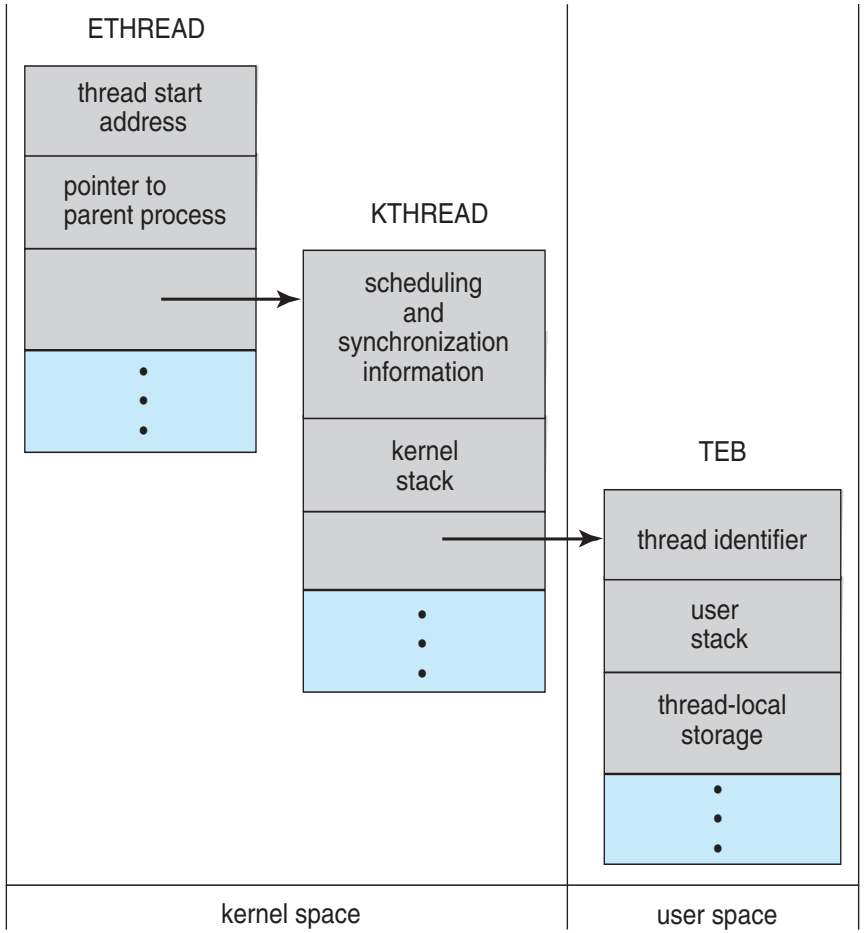
Windows Threads

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

Windows Threads (Cont.)

- 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 Threads Data Structures



Linux Threads

- **Linux refers to them as *tasks* rather than *threads***
  - Linux에서는 task와 process 용어를 거의 구분하지 않고 사용하고 있으며, Thread에 대해서도 task라는 용어를 자주 사용함
- **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)**

- Fork is used to create new processes.
- Clone is a Linux specific low level system call and can be used to either create processes and threads.

End of Chapter 4