Chapter 4: Threads (Part 2)

Chapter 4: Threads

- Overview
- Multicore Programming
- User-level and Kernel-level Threads
- **Multithreading Models**
- Thread Libraries
- Implicit threads
- Threading Issues

Multithreading Models

- A relationship must exist between user threads and kernel threads

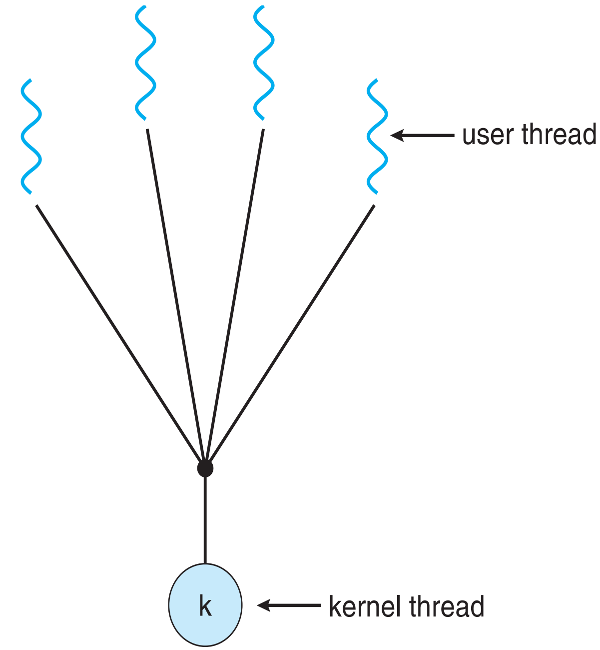
- Many-to-One
  - Many user-level threads mapped to a single kernel thread

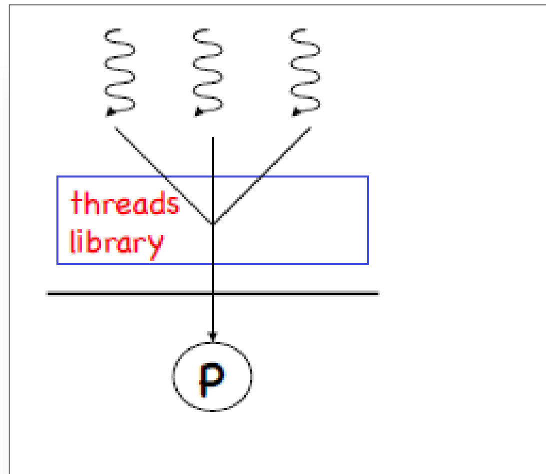
- One-to-One
  - Each user-level thread maps to one kernel thread

- Many-to-Many
  - Allows many user-level threads to be mapped to many kernel threads

Many-to-One

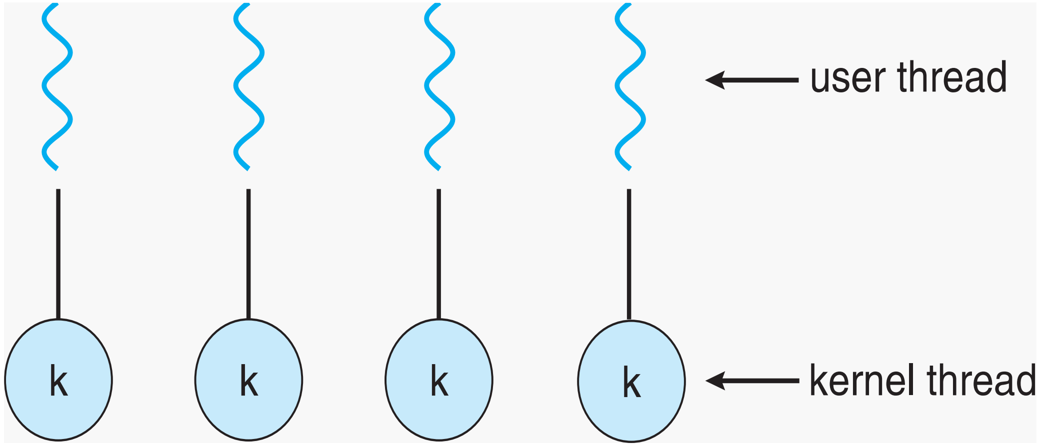
- Many user-level threads mapped to single kernel-level thread
  - Efficiently managed by the thread library
- Thus, this is a user-level thread model.
- Few systems currently use this model
- Examples of many-to-one models:
  - **Green Threads** (adopted in early versions of Java thread library)
  - **GNU Portable Threads**
- **Drawbacks:**
  - One thread blocking causes all to block If the thread makes a blocking system-call
  - Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time and therefore does not benefit from multiple cores





One-to-One

- Each user-level thread maps to one kernel thread.
- Each thread is seen by the kernel as a separately schedulable entity.
- Problem: Creating a user-level thread creates a kernel thread
  - Thread creations burden the performance of an application; **an overhead**
- Advantages:
  - It provides the greatest possible concurrency because it can use as many processors as are available, up to the number of threads, and
  - One thread's blocking does not block other threads.
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Many-to-Many Model

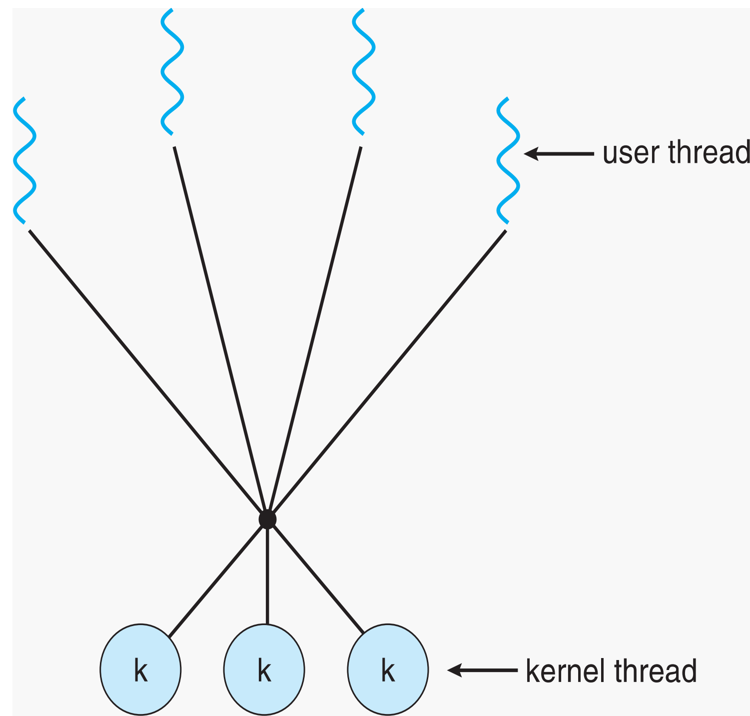
- The **Many-to-Many model** is the most flexible of these models.
- It allows many user level threads to be mapped to many kernel threads
  - ***m*** *user-level threads to be mapped to* ***n*** *kernel threads;* ***n ≤ m***

- User can create as many user threads as wished

- Allows the operating system to create a sufficient number of kernel threads *to be allocated to applications*

- Does not have the problems of other models

- Solaris prior to version 9
- Windows with the *ThreadFiber* package



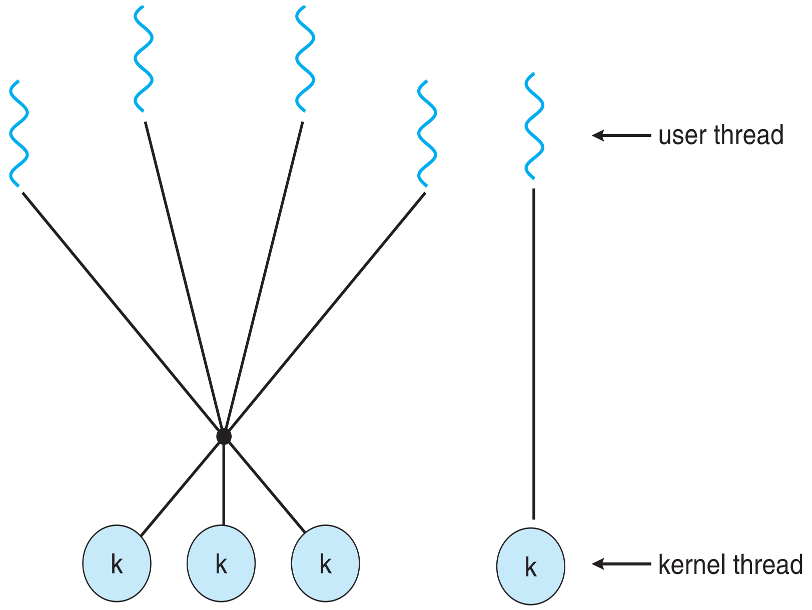
Many-to-Many Model

- This model has several ***advantages***. The most significant include:
  - It does not use kernel resources for user level threads that are not actually runnable.
  - The library-level scheduler can switch between threads much faster because it does not make system calls.
  - It performs better than the others when user level threads synchronize with each other.
- The ***disadvantages*** of this model include:
  - More overhead due to scheduling takes place in both the kernel among the kernel level threads and in the user space for the user level threads.
  - User level threads that are bound to the same kernel level thread can still be blocked when the thread that is running makes a blocking system call.

Two-level (threading) Model

- The two-level model is similar to the many-to-many model but also allows for certain user-level threads to be bound to a single kernel-level thread.
- This is useful when certain threads should not be prevented from running because a thread that is sharing its kernel level thread blocks.

- Examples
  - IRIX
  - HP-UX (Hewlett Packard Unix)
  - Tru64 UNIX
  - Solaris 8 and earlier



Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- **Thread Libraries**
- Implicit threading
- Threading Issues

Thread Libraries

- Thread libraries provide programmers with an API for creating and managing threads.

- There are **three** main thread libraries in use today:

  - **POSIX Pthreads** - may be provided as either a user or kernel library, as an extension to the POSIX standard. Further details in the coming slides.

  - **Win32 threads** - provided as a kernel-level library on Windows systems.

  - **Java threads** - Since Java generally runs on a Java Virtual Machine, the implementation of threads is based upon whatever OS and hardware the JVM is running on.

POSIX Threads (Pthreads)

- A POSIX standard(IEEE 1003.1c) API for thread creation and synchronization
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)
- The POSIX APIs for dealing with threads
  - Declared in pthread.h
  - Not part of the C/C++ language
  - To enable support for multithreading, must include -pthread flag when compiling and linking with gcc command such as
  - gcc –pthread –o mythreadprogram mythreadprogram.c

Pthread Operations

POSIX function	description
pthread_create	create a thread
pthread_detach	set thread to release resources
pthread_equal	test two thread IDs for equality
pthread_exit	exit a thread without exiting process
pthread_kill	send a signal to a thread
pthread_join	wait for a thread
pthread_self	find out own thread ID

Creating a thread with pthread_create

A thread is created with **pthread_create** which takes four arguments

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

i. Creates a new thread, whose identifier is placed in *thread,

ii. attributes *attr (NULL means default attributes)

iii. The new thread runs start_routine(arg).

iv. The last parameter (void *arg) is the sole argument passed to created thread.

- Returns 0 on success and an error number on error.

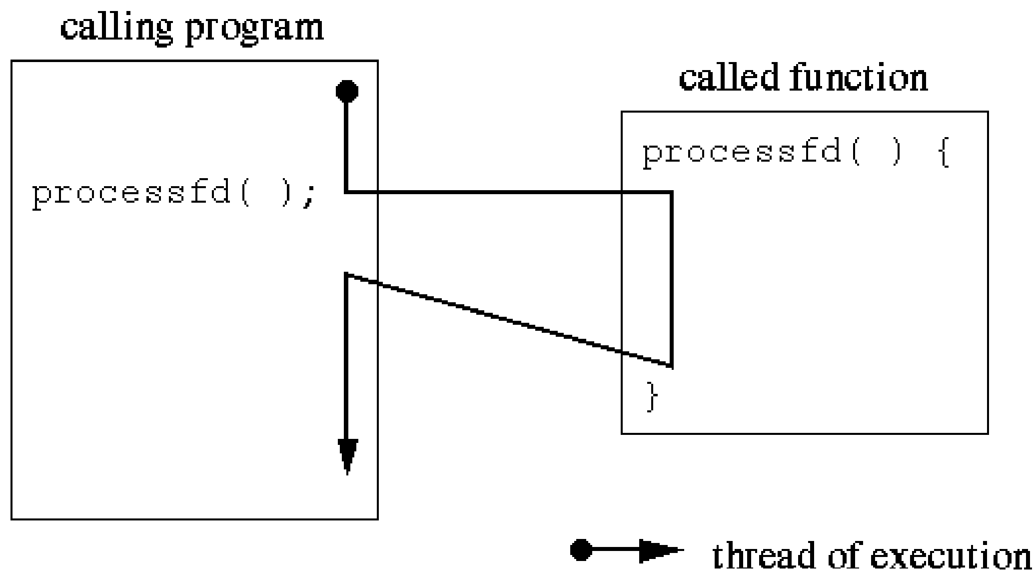
Example 1: Thread Creation

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

void *PrintHello(void *threadid) {
 int tid;
 tid = (int)threadid;
 printf("Hello World! It's me, thread #%d!\n", tid)
 pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
        pthread_t threads[NUM_THREADS];
        int rc, t;
        for(t=0; t<NUM_THREADS; t++){
                printf("In main: creating thread %d\n", t);
                rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
                if (rc) {
                        printf("ERROR code is %d\n", rc);
                        exit(1);
                }
        }

}
```

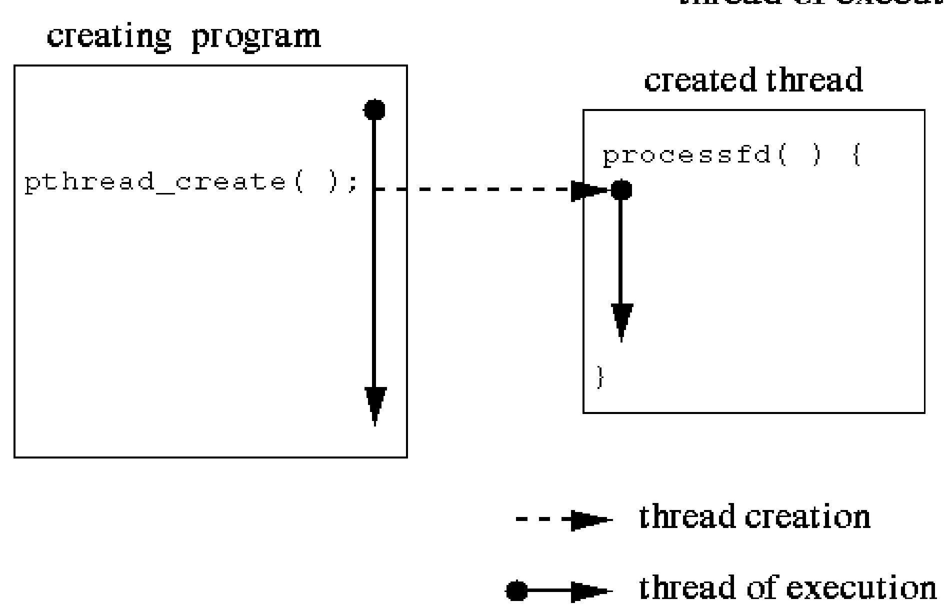
```
One possible output:

In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In main: creating thread 3
Hello World! It's me, thread #0!
In main: creating thread 4
Hello World! It's me, thread #1!
Hello World! It's me, thread #3!
Hello World! It's me, thread #2!
Hello World! It's me, thread #4!
```

Normal vs. Threaded function call

Normal function call



Threaded function call



Terminating Threads

pthread_exit is used to explicitly exit a thread

– Called after a thread has completed its work and is no longer

required to exist

• If main() finishes before the child threads it has created

– If exits with pthread_exit(), the other threads will continue to execute

– Otherwise, they will be automatically terminated when main() finishes

Pthreads Example 2

A separate worker thread executes the runner for calculating the sum of integers from 0 to N, and storing the result in a variable "sum".

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

int sum; /* this data is shared by the threads */
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 */

int s;

s = 5;

printf ("creating thread\n");
/* get the default attributes */
pthread_attr_init(&attr);
/*create the thread */
pthread_create(&tid, &attr, runner, (void *)s );
/*wait for the thread to exit */
printf ("After thread is created\n");

pthread_join(tid, NULL);

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

Pthreads Example 2 (Cont.)

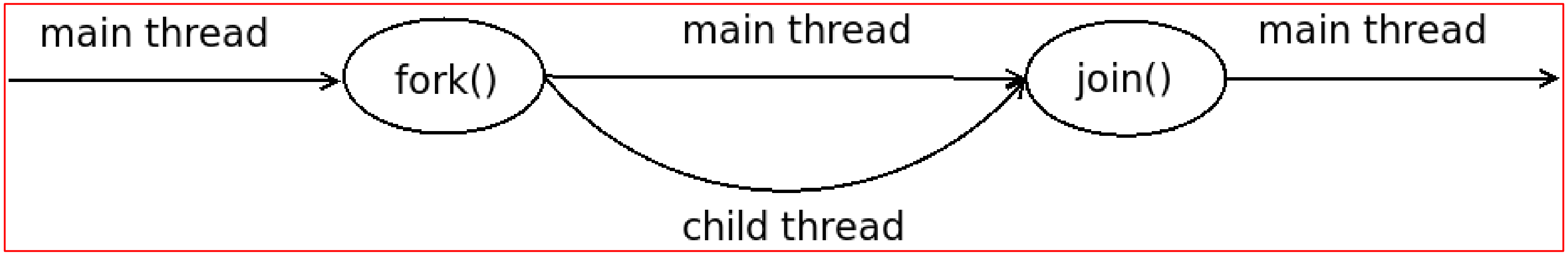
A separate worker thread executes the runner for calculating the sum of integers from 0 to N, and storing the result in a variable "sum".

```
/* 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);
}
```

Waiting for thread termination (Fork-Join Flow)

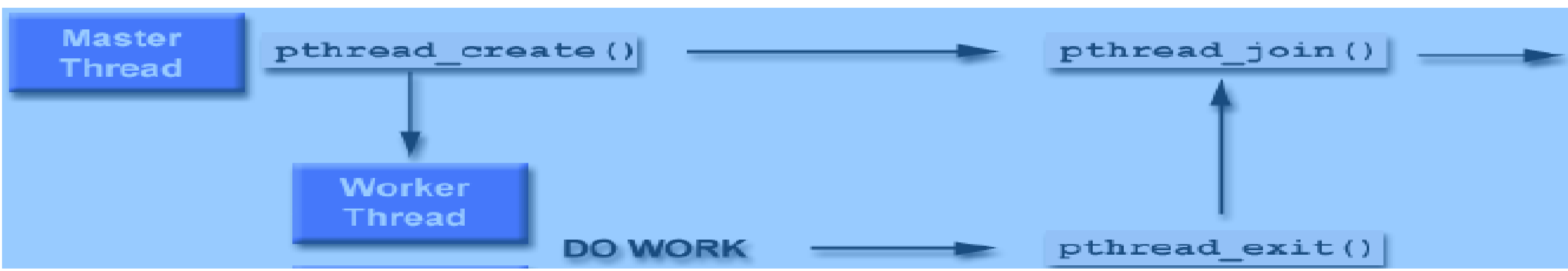
int pthread_join(pthread_t thread, void** retval);

- The **pthread_create()** call creates a new thread, called the child thread. The caller is the **parent thread**.
- The **pthread_join()** call makes the main program wait until the **child thread** makes a call to pthread_exit().



Waiting for thread termination (Fork-Join Flow)

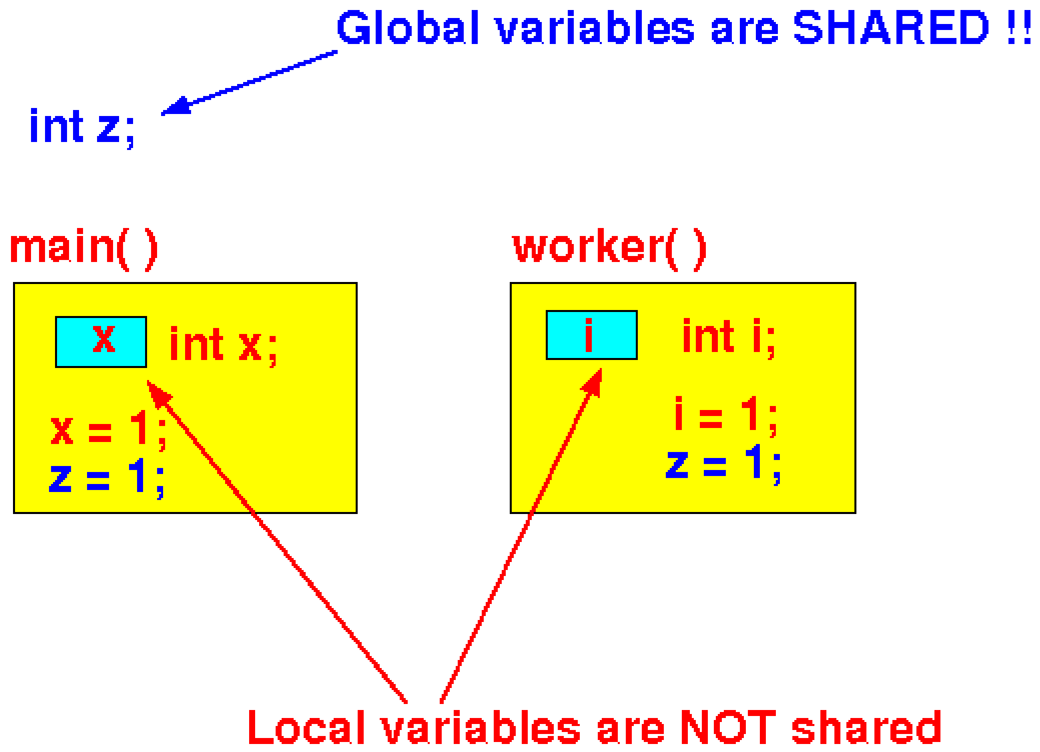
- Similarly, **pthread_join (thread_id, retval)** waits for the thread specified by thread to terminate
  - The thread equivalent of waitpid() for process.
- The exit status of the terminated thread is placed in **retval



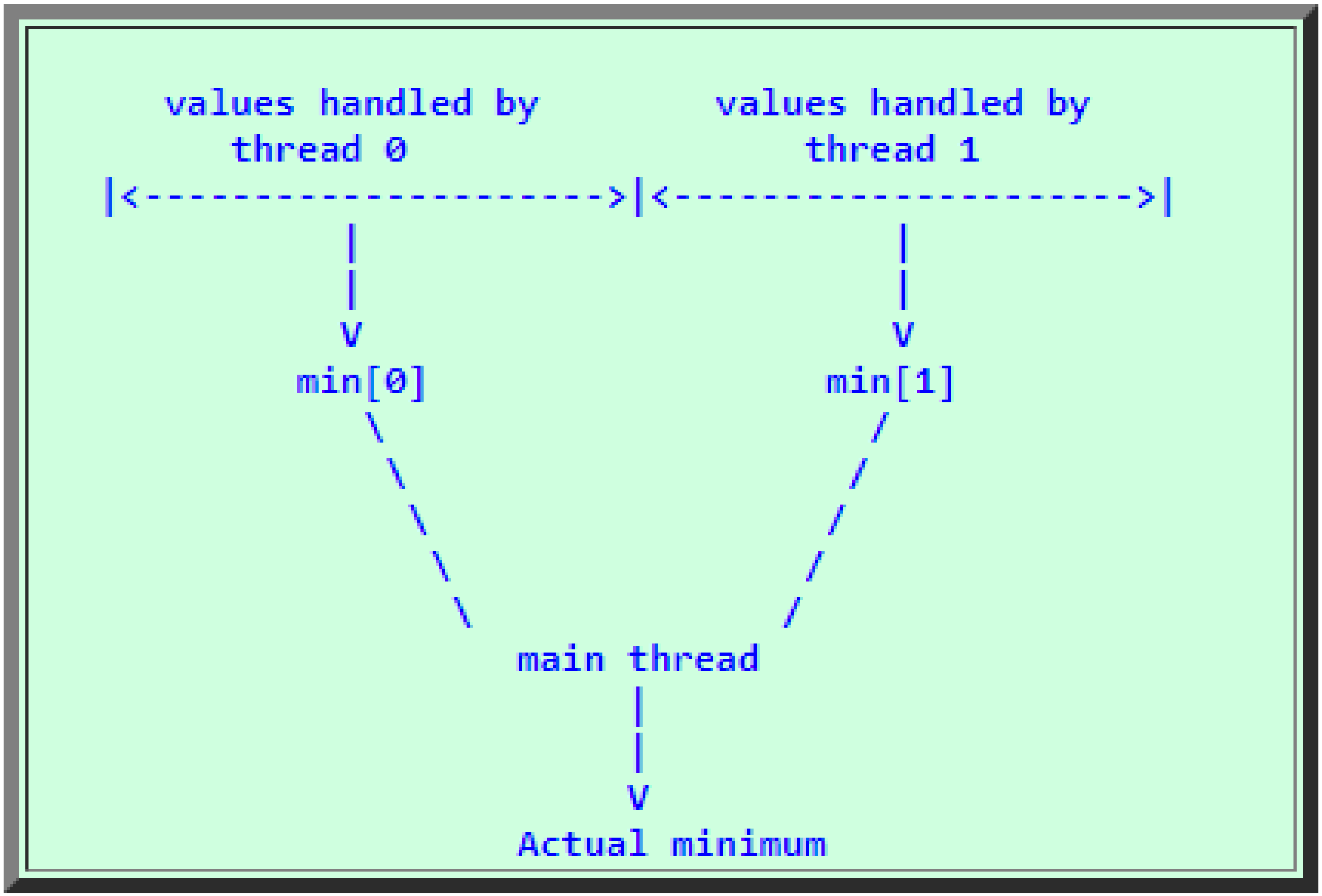
Pthreads Code for Joining 10 Threads

```
#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);
```

Local vs. Global Variables in Threads

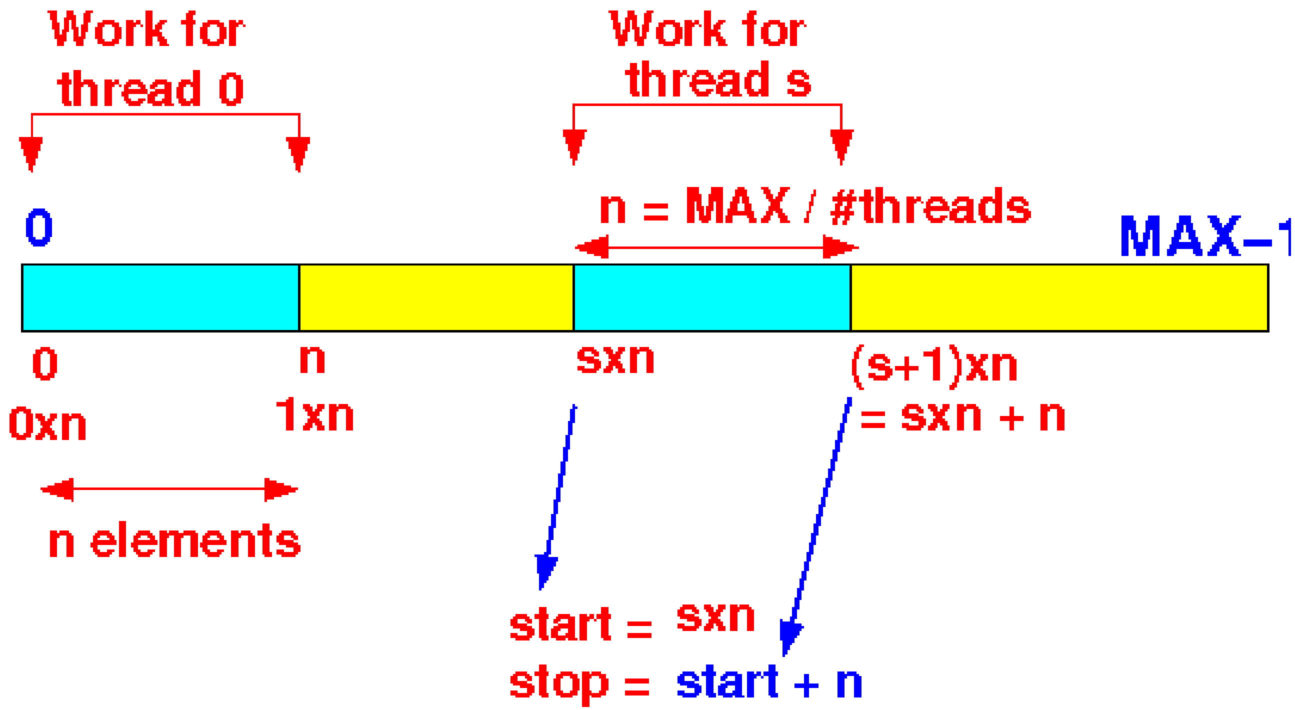


Finding Minimum in an array using Pthreads (1/2)



Finding Minimum in an array using Pthreads (2/2)

- MAX: Total number of array elements
- #threads: Total number of threads
- n: number of elements each thread has to deal with
- If s^{th} is the thread id then it has find minimum in range [sxn) to (s+1)xn] elements



Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries (Posix Threads)
- **Implicit Threading**
- Threading Issues

Implicit Threading

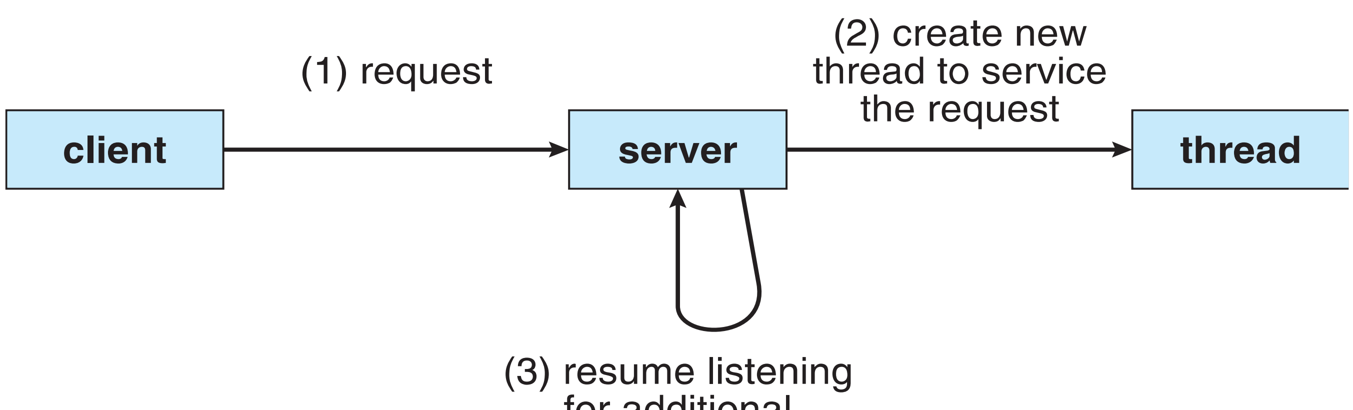
- It is difficult for programmers to write concurrent programs in general, and writing multithreaded programs is among the hardest of tasks.
- Program correctness more difficult with explicit threads (e.g., posix threads). Some of the reasons are:
  - Defining the individual tasks, determining how to distribute data to them
  - Handling coordination and communication among threads is error-prone.
  - Debugging an application containing many threads ?
- Solution = Implicit Threading: Let compilers and runtime libraries create and manage threads rather than programmers and applications developers
- Implicit threading which aims to hide the management of threads as much as possible.
- The basic idea of *implicit threading* is *automation*

Implicit Threading

- Using ***Implicit threading*** compilers and/or libraries create and manage concurrent threads with little or no explicit guidance from the programmer.
- Some well-known implicit threading systems include:
  - ***OpenMP*** (short for Open Multi-Processing) is an API for programs written in C/C++ and FORTRAN that may be used to explicitly specify multithreaded, shared-memory parallelism.
  - ***Grand Central Dispatch*** (GCD) is a technology developed by Apple for its macOS and iOS operating systems. Like OpenMP, it includes a run-time library, an API, and language extensions that allow developers to identify sections of code to run in parallel.
  - Thread Pool:

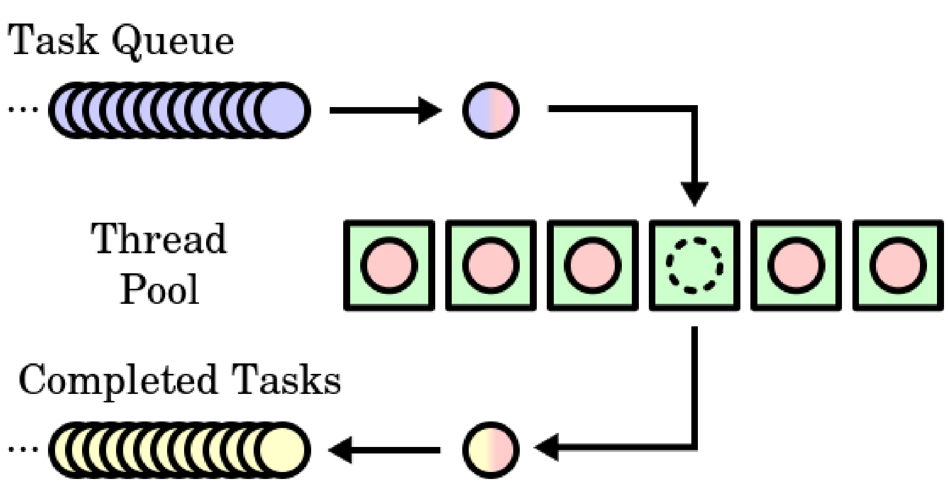
Implicit Threading 1: Thread Pools

- Consider a multithreaded web server. When the server receives a request, it creates a separate thread to service the request.
- When the request has been serviced, the thread is deleted.
- There are two problems with this:
  - Threads are constantly being created and destroyed.
  - There is no bound on how many threads can exist at any time.
- The first problem leads to poor CPU utilization and wasted memory resources. The second could lead to system degradation.



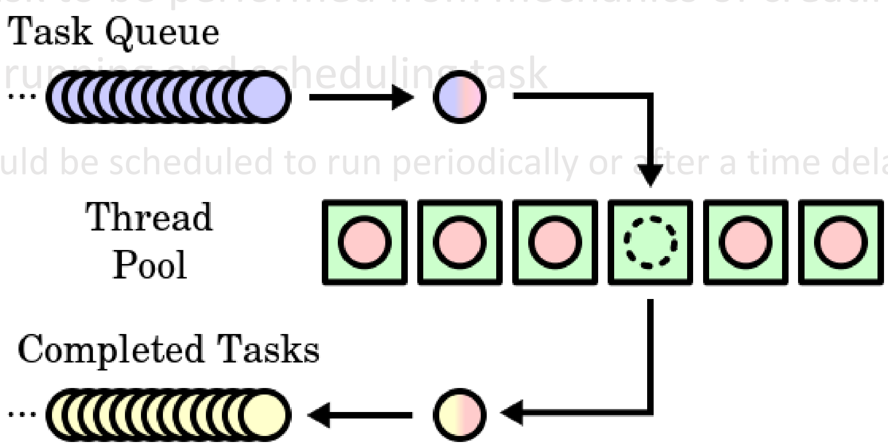
Implicit Threading 1: Thread Pools

- To solve this, rather than constantly ***creating and deleting*** threads, the implementation can maintain a pool of threads,. When work comes in, the **worker thread** is assigned to it. When it finishes, it goes back to the waiting room.
  - A thread pool is initialized with a number of threads, where they await work
  - When a process needs a new thread to perform a task, it requests one from the thread pool.
  - If there is an available thread in the pool, it is awakened and assigned to the process to execute the task.
  - If the pool contains no available threads, the task is queued until one becomes free.
  - Once a thread is available, it returns to the pool and awaits more work.



Implicit Threading 1: Thread Pools

- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread. A thread returns to pool once it completes servicing a request.
  - Allows the number of threads in the application(s) to be bound to the size of the pool. Limits the number of threads that exist at any one point.
  - *Separating* task to be performed from mechanics of creating task allows different strategies for running and scheduling task
    - i.e. Tasks could be scheduled to run periodically or after a time delay



Implicit Threading 2: OpenMP

- The most common implicit threading library is OpenMP.
- OpenMP is a portable, parallel programming model for shared memory multiprocessor architectures, developed in collaboration with a number of computer vendors.
- OpenMP API are available for C/C++ and Fortran
- **OpenMP programs** are **C/C++ programs** that include ***compiler instructions*** **("directives")** to tell the **Fortran** or **C/C++** compiler to generate parallel execution code (using ***threads***)

Implicit Threading 2: OpenMP

- **Parallel regions** are specified using compiler directives, known as **pragmas**. For example, a simple directive is

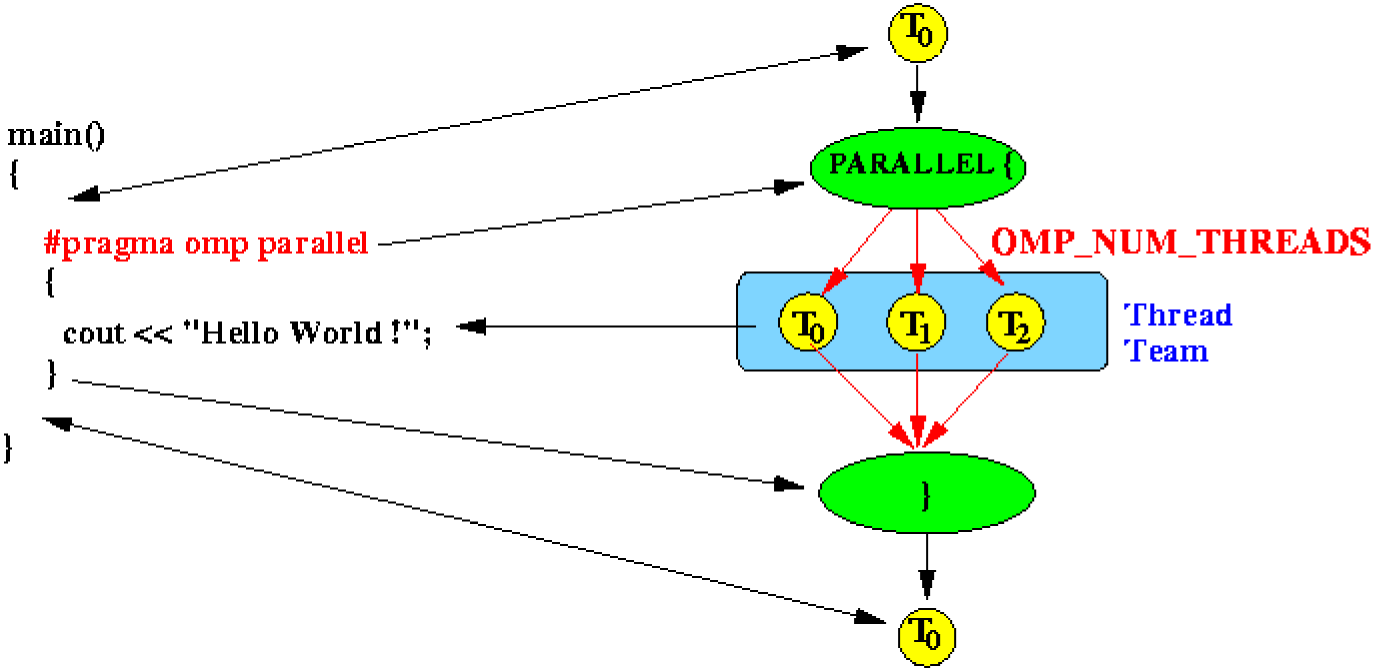
```
#pragma omp parallel
```

- By default, creates as many threads as there are cores
- Consider introductory OpenMP Program :

```
#include <opm.h>
int main(int argc, char *argv[])
 {
int tid;
  #pragma omp parallel
  {
tid = omp_get_thread_num();
printf("Hello World from thread = %d\n", tid);
   }
nthreads = omp_get_num_threads();
printf("Number of threads = %d\n", nthreads);
}
```

```
Hello World from thread = 2
Hello World from thread = 0
Hello World from thread = 3
Hello World from thread = 1
Number of threads = 4
```

openMP



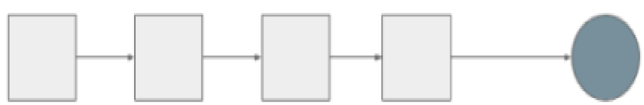
main()
{
#pragma omp parallel
{
cout << "Hello World !";
}
}
T_0
PARALLEL {
OMP_NUM_THREADS
T_0 T_1 T_2
Thread
Team
}
T_0

Implicit Threading 3: Grand Central Dispatch

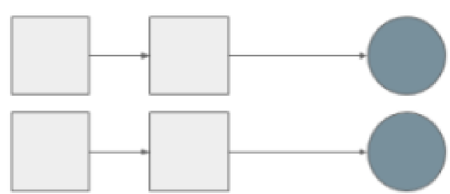
- Apple technology for **Mac OS X** and **iOS** operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Like OpenMP, it provides parallel processing,
- *Block* specified by is in “^{ }” - `^{ printf("I am a block"); }`
  - Block = self-contained unit of work identified by the programmer as above
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue

Grand Central Dispatch

- GCD schedules by placing blocks on dispatch queue. Two types of 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



  - T w, default, high



Chapter 4: Threads

- Overview
- Multicore Programming
- Multithreading Models
- Thread Libraries
- Implicit Threading
- **Threading Issues**

Threading Issues

- Here are some of the issues that arise in the design and implementation of multithreaded programs, whether they are in user level libraries or kernels.

1. Semantics of **fork()** and **exec()** system calls
2. Signal handling
   - Synchronous and asynchronous
3. Thread cancellation of target thread
   - Asynchronous or deferred

1. Issues with fork()

- When a thread that is part of a process issues a fork() system call, a new process is created. The question is, **Should just the calling thread be duplicated in the child process or should all threads be duplicated?**
- Most implementations duplicate just the calling thread. Duplicating all threads is more complex and costly.
  - Some UNIXes have two versions of fork
    - One that duplicates all threads (forkall())
    - And another that duplicates only the thread that invoked fork() system call (fork1())
  - Similarly, **Oracle Solaris**'s fork() duplicates all threads but its fork1() duplicates just the calling thread.

1. Issues with exec()

- The exec() system call for processes replace the process's address space entirely, giving it a new program to execute. For example, the call

```
execv("/bin/ls", argv);
```

would cause the calling process to execute the `/bin/ls` program.

- With multithreaded programs, the question is, when a thread makes this call, **should the entire process be replaced, including all threads?**

  - Normally If a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process
  - In other words, exec() system call from any thread in a multithreaded process causes all other threads in that process to terminate and the calling thread completes the exec().

2. Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred. Examples are: pressing ctrl+c, illegal memory access, divide by zero etc.
- 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 is straightforward

2. Signal Handling (Cont.)

- For multithreaded process, the question is, **where should a signal be delivered**?
- Different systems have solved these problems in different ways. Various possibilities are:
  - 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

3. Thread Cancellation (1/2)

- Sometimes a thread might need to be terminated by another thread before it has finished its work.
- **Thread cancellation** is the act of terminating a thread that has not yet terminated itself. With thread cancellation, one thread can **try to terminate** another.
- There are many reasons to allow thread cancellation.
  - Example 1: if multiple threads are concurrently searching through a database and one thread return results, the remaining threads might be cancelled.
  - Example 2: A user presses a button on a web browser that stops a web page from loading any further. A webpage loads using several threads—each image is loaded in a separate thread. When a user presses the stop button on the browser, all threads loading the page are cancelled.
- Thread to be canceled is **target thread**

3. Thread Cancellation (2/2)

- The ***difficulty with cancellation*** occurs in situations where a thread is cancelled while in the middle of updating data its sharing with other threads.
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled. This allows the cancellation to happen in an orderly manner
  - Since problems could be caused by instantly cancelling a thread in a task that is in the middle of doing some work, the implementation of cancellation typically includes ways for threads to ***defer their cancellation*** so that they have time to ***'clean up'*** first - for example to ***deallocate resources*** they are holding, or to ***finish updating shared data***.

End of Chapter 4