# 3.6.3 Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
    - In the case of bidirectional communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- **Ordinary pipes** – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- **Named pipes** – can be accessed without a parent-child relationship.

# Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore **unidirectional**
- **Require parent-child relationship** between communicating processes
- They are referred to as ordinary pipes in Unix/Linux, but Windows calls them **anonymous pipes.**

# Ordinary Pipes – example

```c
#include <sys/types.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
   char write_msg[BUFFER_SIZE] = "Hello, world!";
   char read_msg[BUFFER_SIZE];
   int fd[2];
   pid_t pid;

   /* create the pipe */
   if (pipe(fd) == -1) {
      fprintf(stderr, "Pipe failed");
      return -1;
   }
```

```c
/* fork a child process */
pid = fork();

if(pid<0){ /* error occured */
   fprintf(stderr, "Fork Failed");
   return 1;
}

else if (pid > 0) {/* parent process* /
   /* close the read end since we are the producer */
   close(fd[READ_END]);

   /* Write to the pipe */
   write(fd[WRITE_END], write_msg, strlen(write_msg) + 1);

   /* close the write end, now that we are done */
   close(fd[WRITE_END]);
}
```

```
    else { /* child process*/
        /* close the write end since we are the consumer */
        close(fd[WRITE_END]);

        /* read from the pipe and print the message to screen */
        read(fd[READ_END], read_msg, BUFFER_SIZE);
        printf("read %s\n", read_msg);

        /* close the read end now that we are done */
        close(fd[READ_END]);S
    }
    return 0;
}
```

# Named Pipes

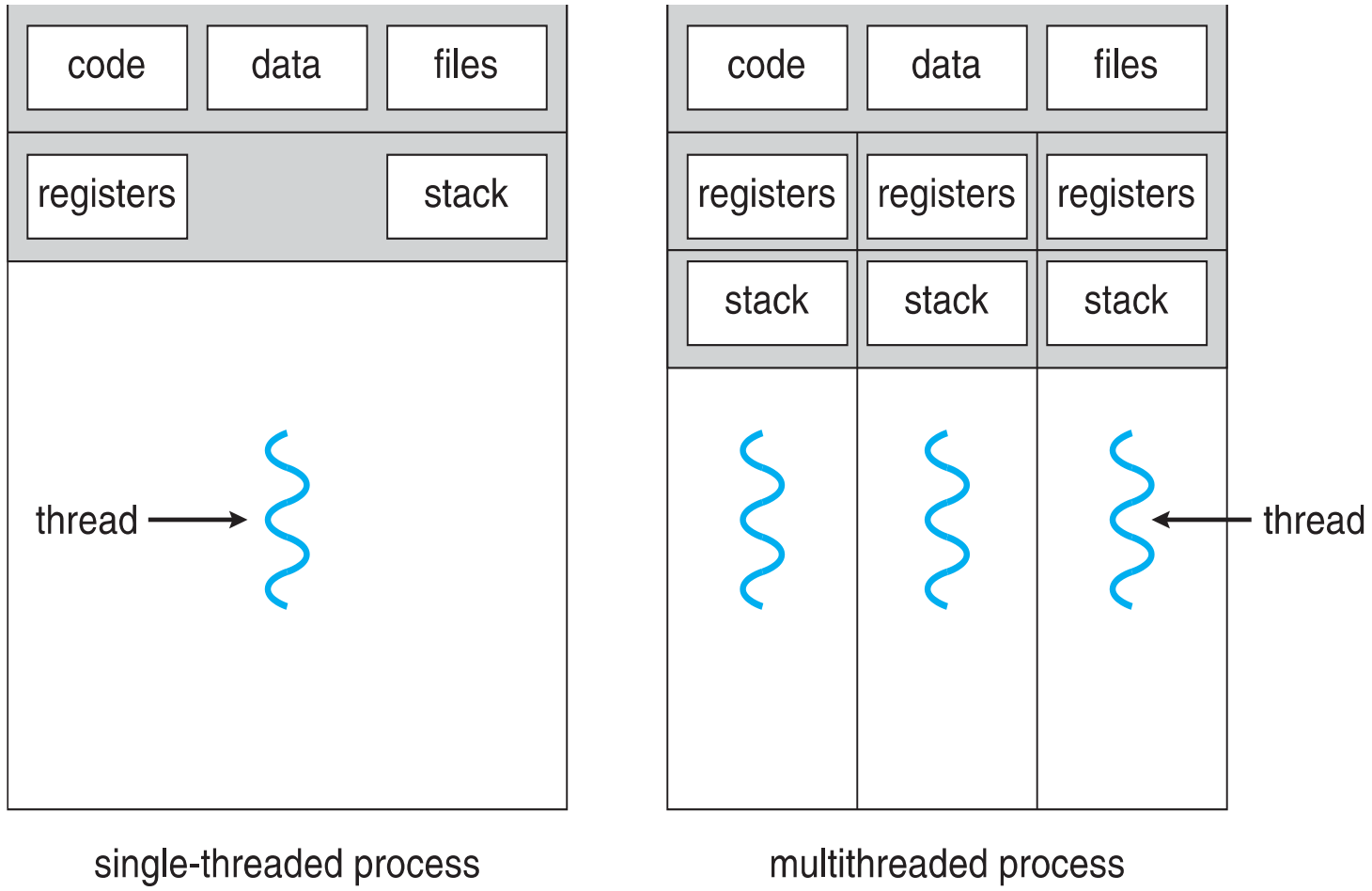
- Named Pipes are more powerful than ordinary pipes
- Communication is **bidirectional**
- **No parent-child relationship is necessary** between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

# Named Pipes in Unix-like systems

- Unix supports named pipes via the **`mkfifo()`** followed by **`open()`** library calls:
  - Note that the two (or more) communicating processes need to use the same pipe name.
  - The default fifo behavior is blocking, however the open flag **O_NONBLOCK** controls whether blocking or non-blocking.
  - In **blocking calls**, both (all) processes are blocked till at least one open for read and one open for write have occurred.
  - **In non-blocking calls**, a process opening the FIFO in read-only always succeeds, whereas a process opening the FIFO in write-only fails if no other process already opened the FIFO for reading.
  - **Whether blocking or non-blocking**, If one process opens the FIFO with read-write mode, then the process will not block or fail.
  - After the FIFO is opened, normal file operations can be used to read and write into the FIFO.

```
int mkfifo(const char *pathname, mode_t mode);
```

# Single and Multithreaded Processes




single-threaded process

multithreaded process

# 4.1 Motivation

- Most modern applications are multithreaded.
- Multiple sub-tasks within an application can be implemented by separate threads, e.g.
  - A word processor may update the display (GUI), save data to disk, spell check
  - A web server may create multiple thread, one for each client request, or else the server may be stuck waiting for one client while other clients are trying to communicate.
- Process creation is heavy-weight while thread creation is light-weight
  - Since threads within a process share the same address space:
    - No need to copy **memory content** from parent to child during creation
    - No memory management information (i.e. **page tables** and **MMU** registers) to be created and stored in the Process control block (PCB)
  - Also threads not only share memory, but also **filesystem objects** (files, I/O's, etc.) or descriptors within the OS → another overhead is reduced.
- Can simplify code and increase efficiency (programmer doesn't have to deal with **IPC** – message passing or shared memory)
- Kernels are generally multithreaded

# 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 (no MMU registers need to be saved/restored)
- **Speedup –** a process can take advantage of multiprocessor architectures by using multiple parallel threads

# 4.2 Multicore Programming

- Taking advantage of **multicore** or **multiprocessor** systems puts pressure on programmers. Challenges include:
  - **Dividing activities:** dividing applications into multiple functions or tasks and identifying which ones can operate in parallel
  - **Balance:** Ensuring that each task or group of tasks running on a CPU core have the same amount of load as others running on different cores.
  - **Data splitting:** Data may also need to be split and distributed amongst the different cores.
  - **Data dependency:** Not all data is split amongst threads. Some data may be shared and thus it is essential to organize how the different threads access them. More about synchronization on Chapter 5.
  - **Testing and debugging** is more challenging than in single-threaded applications

# Multicore Programming (Cont.)

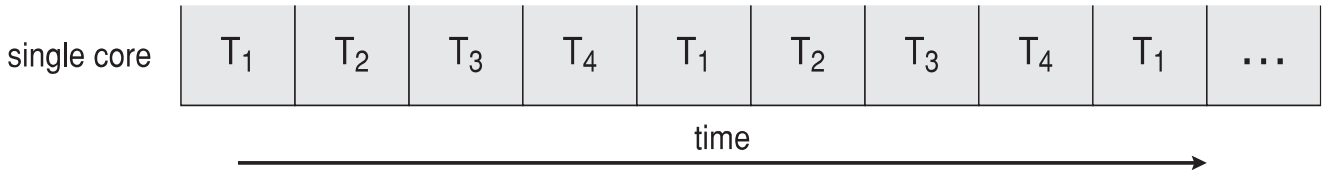
- ***Parallelism*** implies a system can perform more than one task/thread simultaneously (on multiple CPUs)

- ***Concurrency*** implies a system can perform more than one task/thread by time sharing the CPU.

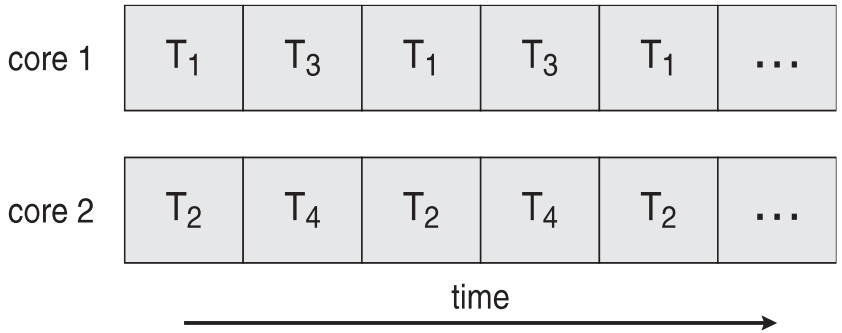
- In a single processor core: The scheduler provides time-sharing, aka concurrency
- In multiple CPU cores, the scheduler provides concurrency and parallelism.

# Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**




- **Parallelism on a multi-core system:**



# Multicore Programming (Cont.)

- Types of parallelism
  - **Data parallelism** – distributes subsets of the data across multiple cores, same operation on each
  - **Task parallelism** – distributing threads across cores, each thread performing unique operation

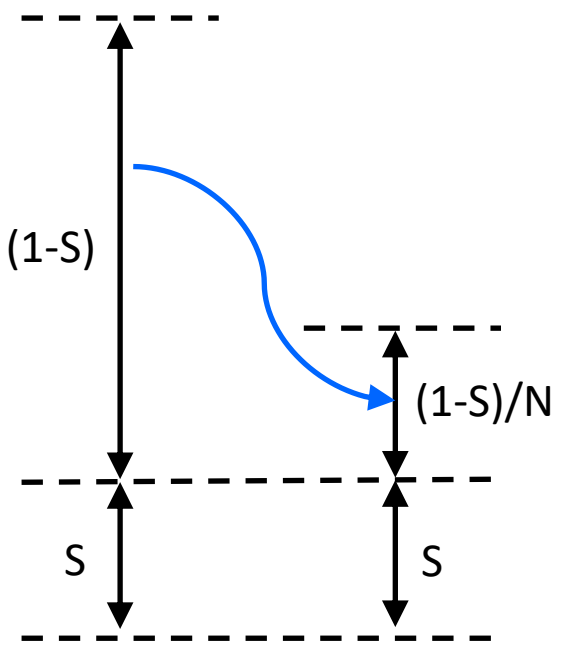
# Amdahl's Law

- 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**



# User Threads and Kernel Threads

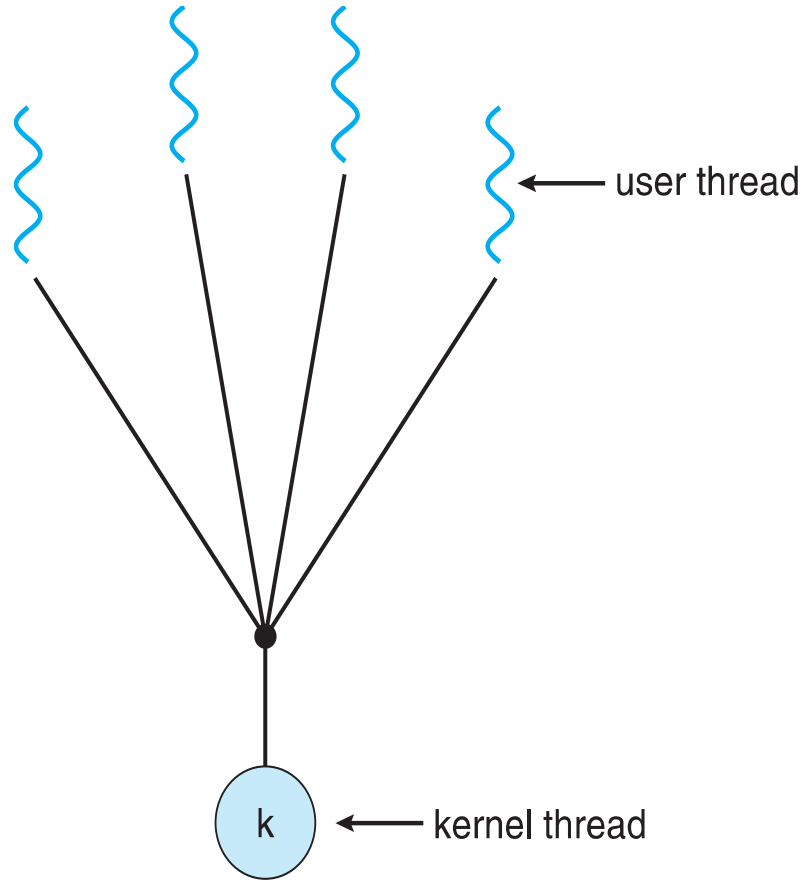
- **User threads** – Thread management takes place using a threads library without OS support.
  - Kernel would treat the process as single-threaded and any blocking system call by one of the threads would end up blocking all the threads of that process.
- **Kernel threads** - Supported and managed by the OS Kernel. Kernel support exists for most well-known Oses, e.g.:
  - Windows
  - Solaris Unix
  - Linux
  - Tru64 Unix
  - Mac OS X
- **NOTE:** A kernel thread is not meant to indicate a thread executing kernel code, but rather a thread that is managed and supported by the kernel.

# 4.3 Multithreading Models

- Several thread management models exist:
  - Many-to-One model
  - One-to-One model
  - Many-to-Many model

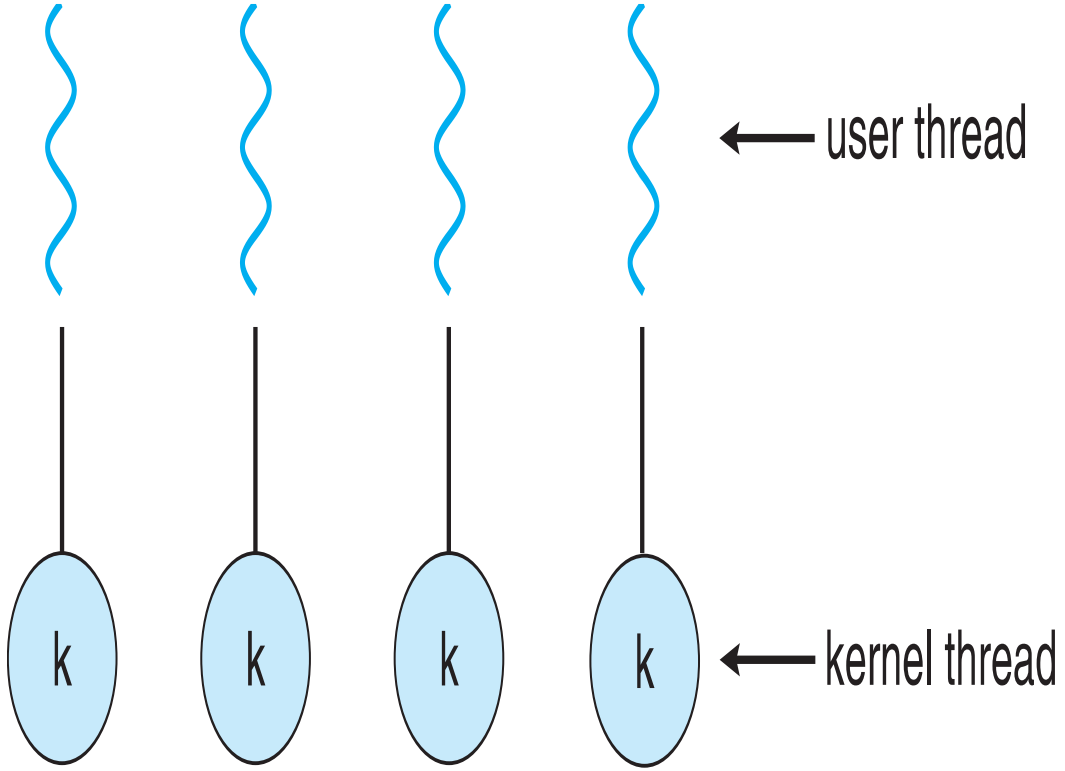
# Many-to-One

- 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 systems 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
  - Linux
  - Solaris 9 and later

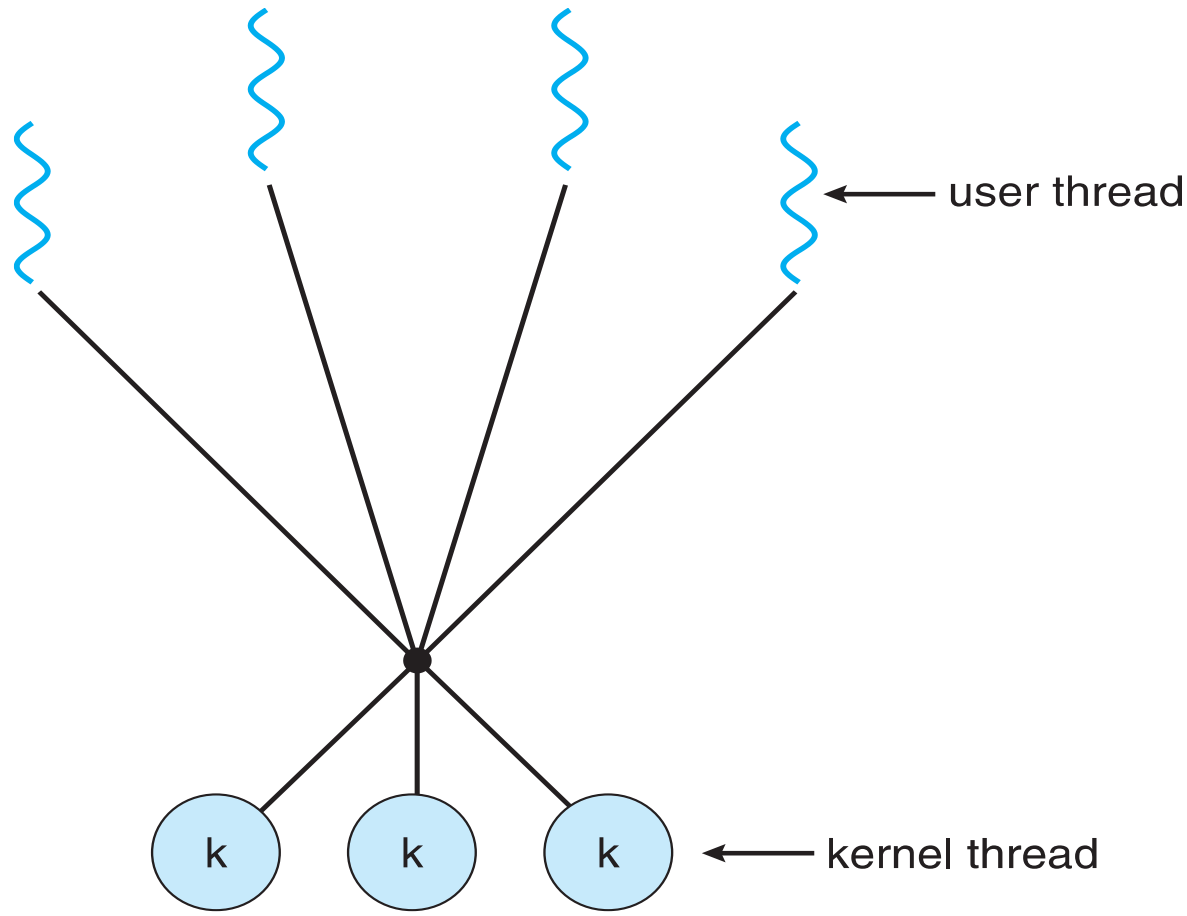


# Many-to-Many Model

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

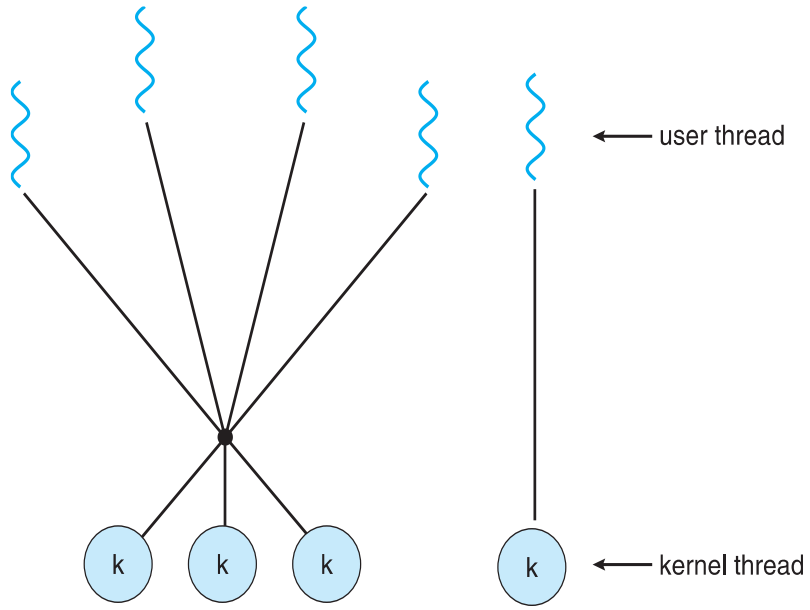
  # kernel threads $\leq$ # user threads

- Allows the operating system to create a sufficient number of kernel threads
- Examples:
  - Windows with the *ThreadFiber* package
  - Solaris prior to version 9



# Many-to-Many variant: The Two-level Model

- Similar to the many-to-many model in that it allows many user threads to map to many kernel threads.
- But it also allows a one-to-one relationship on some user/kernel thread pairs as shown on the diagram.
- Examples
  - HP-Unix
  - Tru64 Unix
  - Solaris prior to version 8



# 4.4 Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads and is responsible for implementing a thread management model
- Three primary thread libraries:
  - POSIX **Pthreads** (whether user/kernel thread is platform dependent, but same interface)
    - Note: You need to use the –pthread option so that your code links with the pthreads library.
  - Windows threads (kernel threads)
  - Java threads (mostly user threads)
- Two primary ways of implementing
  - Library entirely in user space (i.e. with no kernel support)
  - Kernel-level library supported by the OS

# 4.4.1 Pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- May be implemented either as user-level or kernel-level.
- ***Specification***, not ***implementation***
  - Different implementation for different OS platforms, but all have the same interface.
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX-like operating systems (Solaris, Linux, Mac OS X)
  - Note: For Linux, it is implemented using kernel threads

# 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);
}

/* 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);
}
```

# 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);
```

# 4.4.2 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;
   }
```

# Windows Multithreaded C Program (Cont.)

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

# Windows Multithreaded C Program (Cont.)

The creating thread can use the arguments to **CreateThread** to specify the following:

- The security attributes for the handle to the new thread, including:
  - An **inheritance flag** that determines whether the handle can be inherited by child processes.
  - A **security descriptor**, which the system uses to perform access checks on all subsequent uses of the thread's handle before access is granted.
- The initial **stack size** of the new thread.
  - The thread's stack is allocated automatically in the memory space of the process
  - The system increases the stack as needed and frees it when the thread terminates.
- A creation flag that enables you to create the thread in a **suspended** state. When suspended, the thread does not run until the **ResumeThread** function is called.

# 4.5 Implicit Threading

- Growing in popularity; As the number of threads increases, program correctness becomes 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
- Other methods include Microsoft Threading Building Blocks (TBB) and **`java.util.concurrent`** package

# 4.5.1 Thread Pools

- **At process startup**, create a **predefined** number of threads in a pool where they await work.
- Threads wait for work to be dispatched to them. When work is dispatched to the thread it performs it and when done returns back to the pool.
- If more work needs to be dispatched with no threads available in the pool, the main process waits till a thread becomes available to the pool.
- **Advantages:**
  - Servicing a request with an existing thread is **faster than creating** a new thread
  - Allows the number of threads in the application(s) to be **bound** to the size of the pool (thus preventing the creation of too many threads)
  - Separating tasks to be performed **from mechanics of creating task allows** different strategies for running tasks
    - e.g. Tasks may be scheduled to run as **one-shot** after a delay time
    - or may be scheduled to run **periodically**
- **Use case:** Works very well for tasks that have a finite duration, i.e. tasks that start, do some work, then exit.

# Thread Pools – cont.

- **The number of threads** in a pool may be **pre-determined** based on the number of CPUs, memory or the expected number of client requests.
- Alternatively, more sophisticated systems may adjust the number of threads **dynamically.**
- Windows API supports thread pools – The user may call an API for dispatching work to a thread using the function:

```
BOOL QueueUserWorkItem(LPTHREAD_START_ROUTINE
    Function, PVOID Param, ULONG Flags );
```

- The function may take the form:

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

# 4.5.2 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 using:

  **`#pragma omp parallel`**

- Creates as many threads as there are cores

```
#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 cont.

- The following runs a for-loop in parallel

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

- OpenMP allows developers to control the level of parallelism by allowing:
  - Manual setting of the **number of threads**.
  - Specify certain data as **shared** or private
- OpenMP is available for open-source as well as commercial compilers:
  - Linux, Windows and Mac OS X.

# 4.5.3 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)
- Block is in "^{ }" – e.g.

```
^{ printf("I am a parallel block"); }
```

- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue (i.e. for getting serviced)
- Grand central dispatch manages the number of threads dynamically.

# Grand Central Dispatch – cont.

Two types of dispatch queues:

- **serial** – blocks removed in FIFO order, **queue is per process**, called **main queue**
  - A block **must finish before** the next one in the queue can be executed (to ensure serial execution).

```
Dispatch_queue_t queue =
dispatch_get_main_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0)

dispatch_sync(queue, ^{ printf("I am a serial block\n"); });
```

  - If parallel execution is desired, programmers may create additional serial queues within program
- **concurrent** – removed in FIFO order but several may be removed at a time and execute concurrently.
  - 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."); });
```

# 4.6 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

# 4.6.1 Semantics of fork() and exec()

- Does **`fork()`** duplicate only the calling thread or all threads?
  - Some UNIX systems have two versions of fork
  - In Linux, if a process has multiple threads and one of them calls `fork()`, the child thread will have a replica of the parent's code, data, stack, heap, file and other resources BUT **the child will only have one thread** running, the one that called the `fork()` function.
- **`exec()`** usually works as normal – replace the running process **including all its threads**

# 4.6.2 Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
  - Synchronous signals are those caused by the running process (e.g. divide by zero, or memory illegal memory access).
  - Asynchronous signals are caused outside the program (e.g. upon an expiration of a timer after the process issues an `alarm()` call).
- A **signal handler** is a function that is used to process/handle signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Each signal is handled by one of two signal handlers:
     1. default
     2. user-defined (provided by the process)
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, the signal is delivered to process's main thread → no issues

# 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
- The standard linux function for delivering signals is:

```
Kill (pid_t pid, int signal)
```

- Most Unix versions allow a **thread to specify which signals it accepts** (else the signal is blocked), and a signal is **delivered ONLY ONCE to the first** thread that accepts it.
- POSIX allows signal delivery to a specified thread

```
pthread_kill(pthread_t tid, int signal);
```

- Windows (which doesn't have signals) uses Asynchronous procedure calls (APC)
  - Allows a thread X to specify an APC function to another thread Y.

```
DWORD QueueUserAPC(PAPCFUNC pfnAPC, HANDLE hThread, ULONG_PTR dwData );
```

  - Windows calls the APC function once thread Y blocks for an event, semaphore or any synchronization object.

# 4.6.3 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 and thus terminates orderly.
- Difficulty:
  - When a thread is allocated resources or shared data. When canceling a thread asynchronously, the OS may reclaim some, but not all the system resources.
  - Also a thread may be in the middle of updating some shared data.

# Thread Cancellation - cont.

- Pthread code to create and cancel a thread:

```
pthread tid;
/* create the thread */
pthread_create(&tid, 0, worker, NULL);
…
/* cancel the thread */
pthread_cancel(tid);
```

# Thread Cancellation (Cont.)

- In pThreads, 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.
  - `pthread_setcanceltype(..)` – to set thread's mode
  - `pthread_setcancelstate(..)` – to set thread's state
- Default is deferred and enabled
  - Cancellation only occurs when thread reaches **cancellation point,**
    - By calling `pthread_test_cancel()` to establish a cancellation point (after it frees the assigned resources or gracefully stops manipulating shared data).
    - After calling pthread_join()
    - After calling sigwait() or pthread_cond_wait()
- On Linux systems, thread cancellation is handled through signals

# 4.6.4 Thread-Local Storage

- Generally threads within the same process share their data, but sometimes a thread may have some data that is not to be shared (e.g. tabbed webpages each with a thread) -> it needs **Thread-local storage** (**TLS**).
- Local variables within a function do not directly achieve that, since they do not propagate across function calls.
- Most thread libraries (pthreads and Windows API) provide some form of support.
- For pthreads use the `__thread` specifier prior to variable declaration, e.g.

```
__thread int i;
```

  This requires significant support from the compiler (gcc), linker (ld), dynamic linker (ld.so) and system libraries (libc.so and libpthread.so)