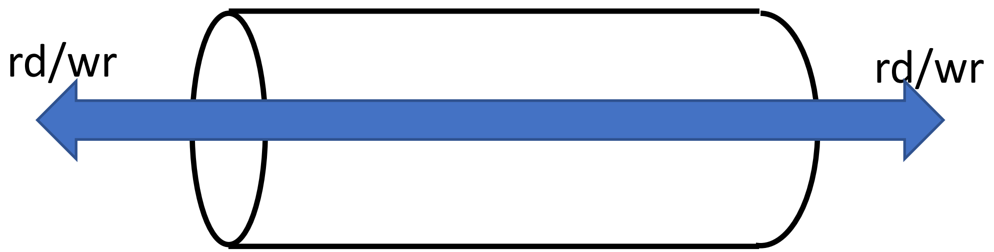
Named Pipes

- 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.

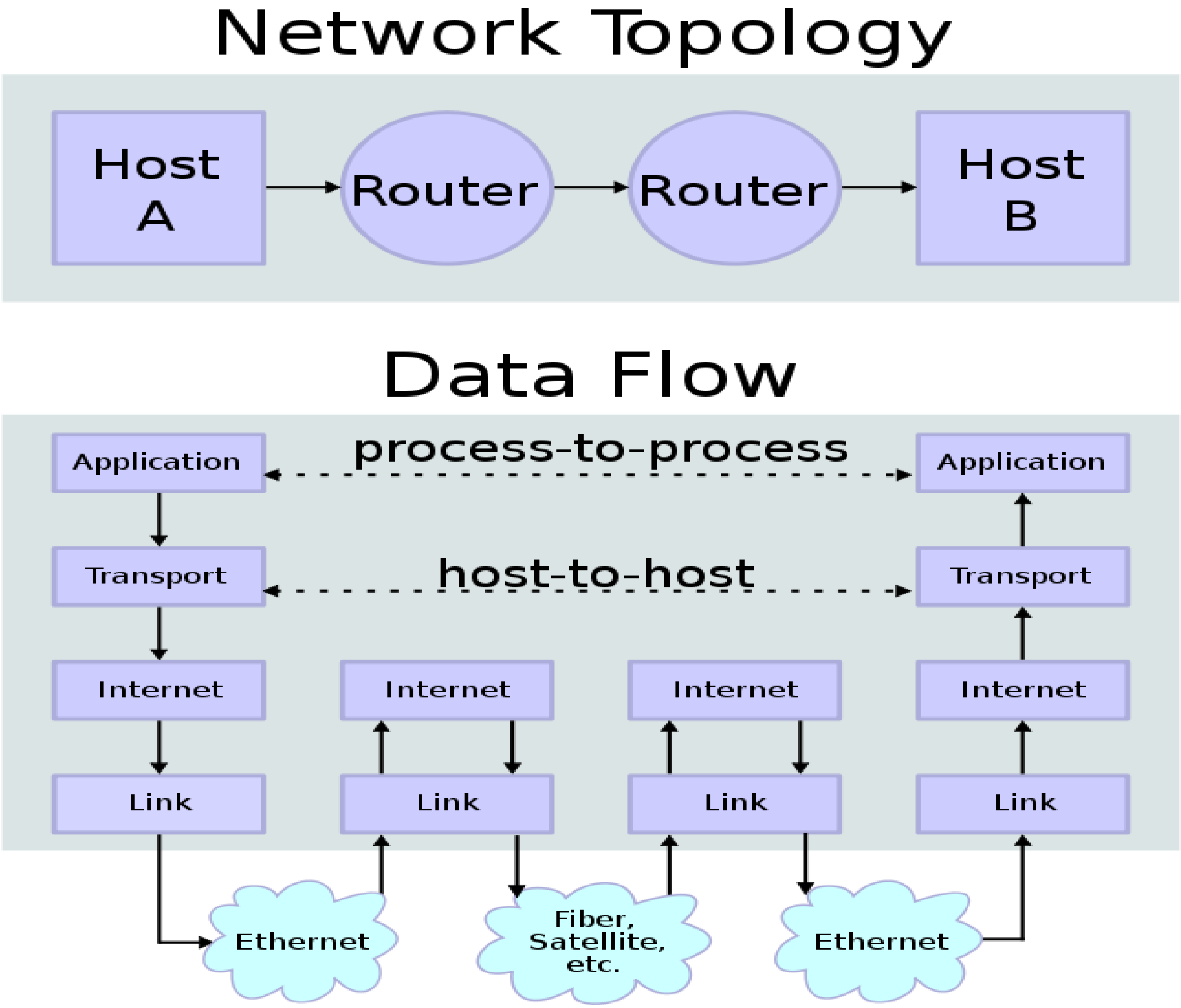
```
int mkfifo(const char *pathname, mode_t mode);
```

The internet layered communication model model

There are two main

communication models:

- The internet model
  - 4 layers
  - The most popular
- The Open system interconnect model
  - 7 layers
  - Not as popular today



3.6.1 Sockets

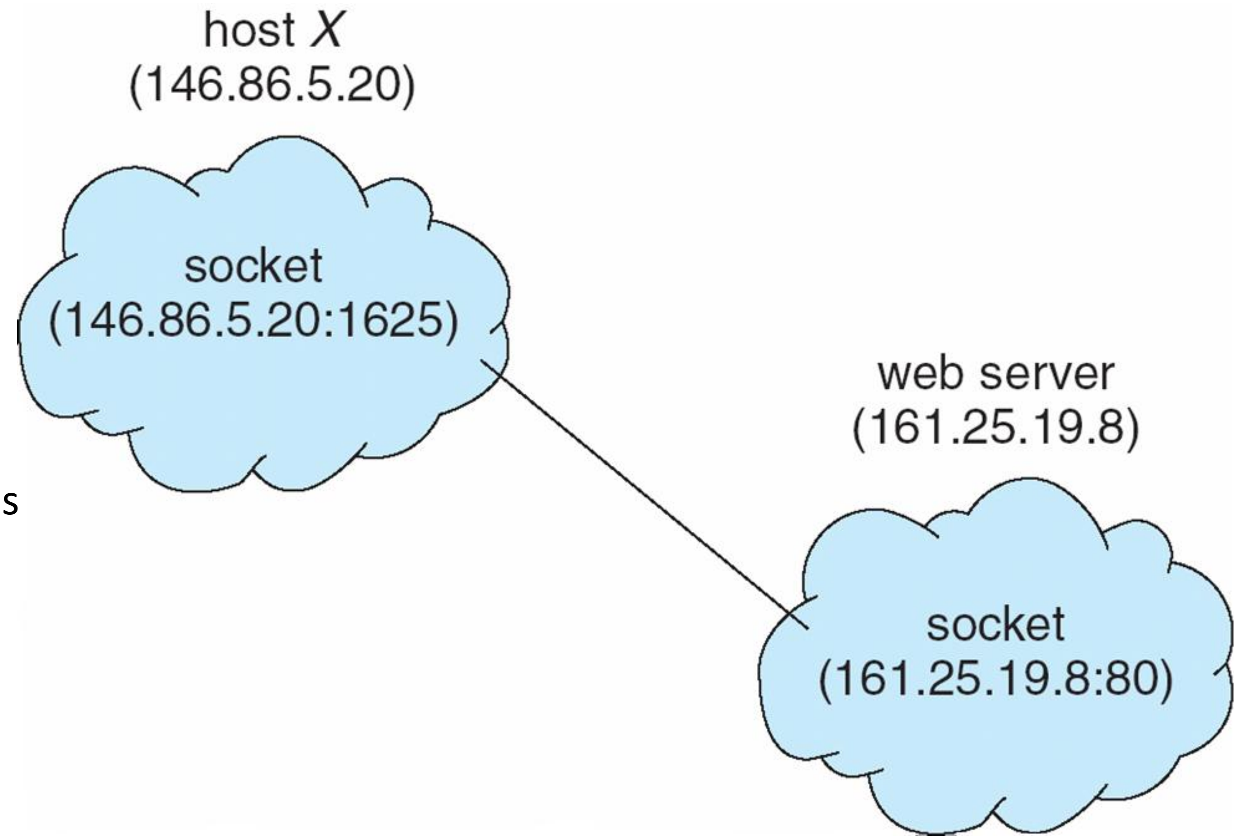
- A **socket** is defined as an endpoint for communication
- A socket is the concatenation of an **IP address + a port number** (a number included at start of message packet to differentiate network services on a host).
- Communication consists between a pair of sockets, to form a virtual circuit.

You need to memorize this

- Port numbers are a **16-bit values**
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Socket Communication

- A client process may open a socket by specifying its protocol type. The system assigns the socket:
  - An IP address (equals IP of the client or host)
  - An arbitrary port number (above 1024, e.g. 1625)
  - e.g. the socket or end point is **146.86.5.20:1625**
- The protocol may be:
  - Transmission control protocol **(TCP)**: connection oriented
  - User datagram protocol **(UDP)**: connectionless
- The client process may then connect the socket to a server at **161.25.19.8:80,** i.e. at port 80 of the server, thus forming a virtual circuit or connection.
- Port number examples:
  - http/web servers listens on port 80,
  - Secure web server (SSL) listens on 443
  - ftp server listens on port 21.
  - Un-encrypted smtp mail server: 25
  - Encrypted smtp mail server: 465



3.6.2 Remote Procedure Calls

- Sockets are a form of low-level communication since they do not specify the data format. Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
    - Again uses ports for service differentiation (e.g. Oracle/Sun Microsystems RPC uses port 111 (may use TCP or UDP)

- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshals** the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server

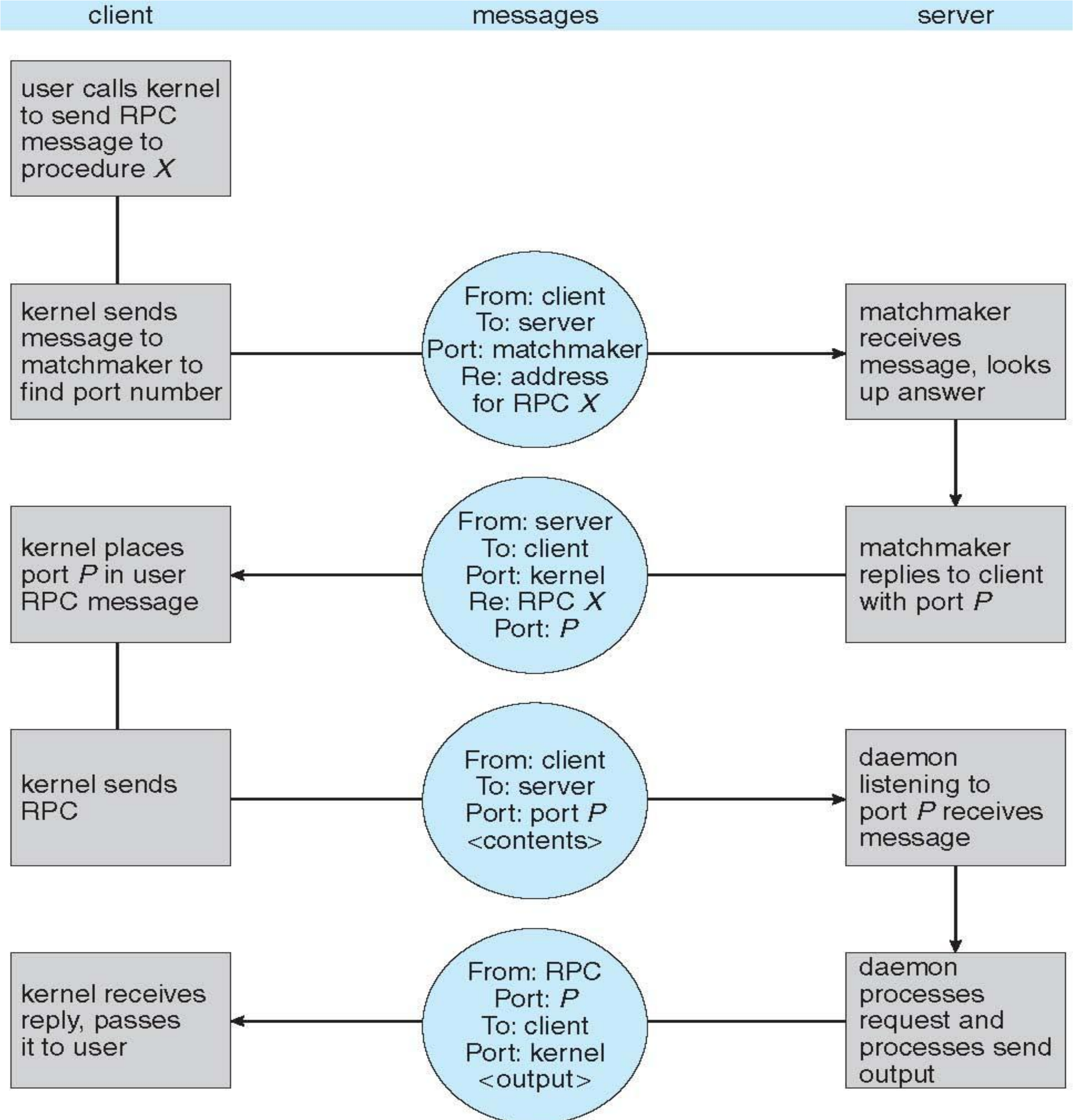
Remote Procedure Calls (Cont.)

- Stubs need to address:
  - Different data representations (i.e. **Big-endian vs little-endian)**
  - Remote communication has more **failure scenarios** than local procedure calls due to communication errors (lost or duplicated messages)
  - OS must ensure messages are delivered ***exactly once.***

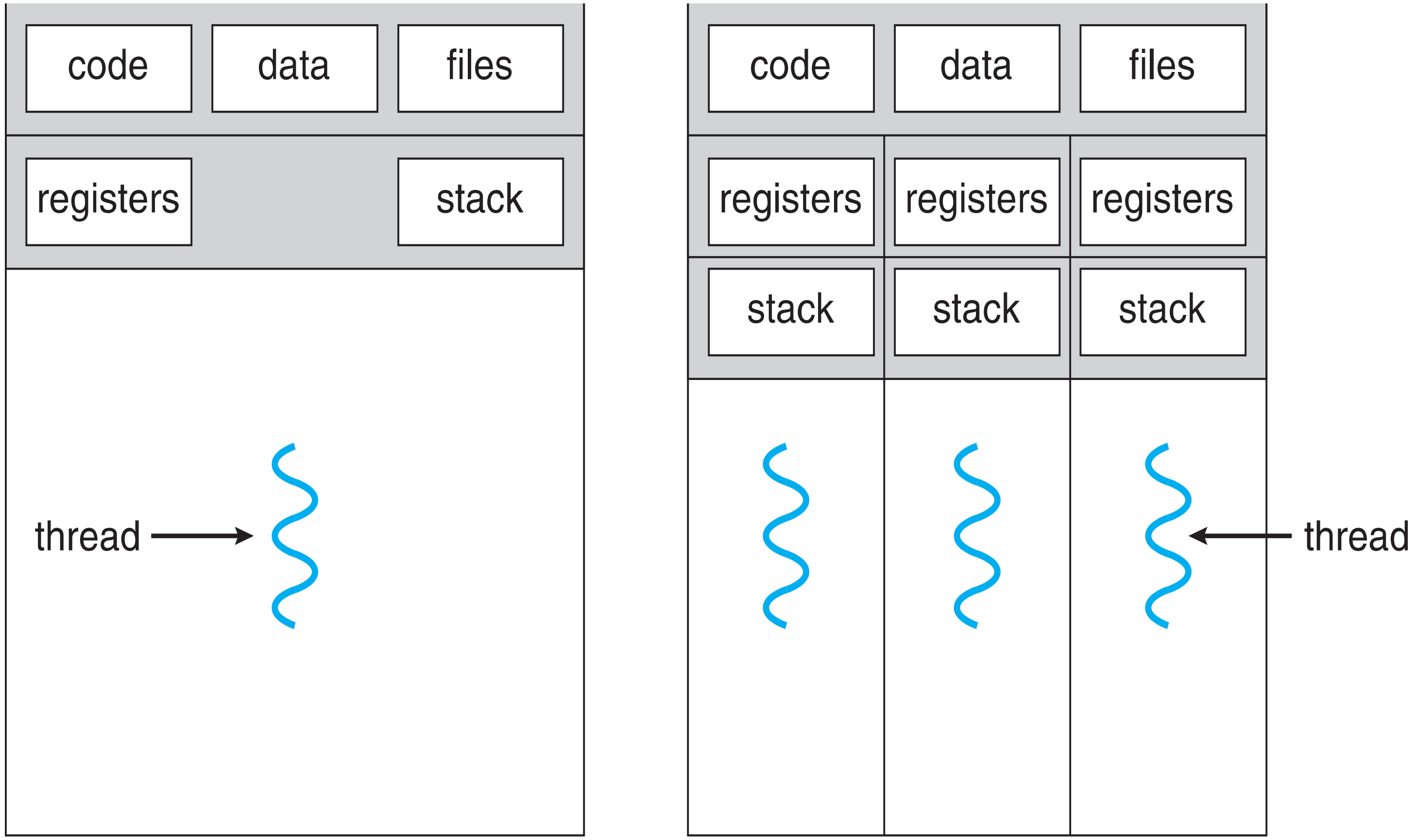
- There are many different, and **often incompatible, RPC standards** (e.g. Oracle RPC, used for network file systems, Microsoft DCOM remoting, Google Web ToolKit, etc.)

- OS typically provides a rendezvous (or **matchmaker**) service to connect client and server

Execution of RPC


client
messages
server
user calls kernel to send RPC message to procedure *X*
kernel sends message to matchmaker to find port number
From: client
To: server
Port: matchmaker
Re: address for RPC *X*
matchmaker receives message, looks up answer
matchmaker replies to client with port *P*
From: server
To: client
Port: kernel
Re: RPC *X*
Port: *P*
kernel places port *P* in user RPC message
kernel sends RPC
From: client
To: server
Port: port *P*
<contents>
daemon listening to port *P* receives message
daemon processes request and processes send output
From: RPC
Port: *P*
To: client
Port: kernel
<output>
kernel receives reply, passes it to user

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.
- Kernels are generally multithreaded

Benefits

- **General Benefits:**
  - **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
  - **Speedup –** a process can take advantage of multiprocessor architectures by using multiple parallel threads

- **Advantages over processes:**
  - **Resource Sharing –** threads within a process share **address space** and **all resources**, including filesystem objects (files, I/O's, etc.) or descriptors within the OS.
    - Can simplify code and increase efficiency (programmer doesn't have to deal with **IPC** – message passing or shared memory)
  - **Economy**
    - Threads within a process share the same address space: No need to copy memory content from parent to child during **creation**. Thus, Process creation is heavy-weight while thread creation is **light-weight**.
    - Thread **switching** has lower overhead than process context switching. No memory management information (i.e. page tables and MMU registers) to be created, saved or restored in the Process control block (PCB)

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 later (next week).
  - **Testing and debugging** is more challenging than in single-threaded applications

Multicore Programming (Cont.)

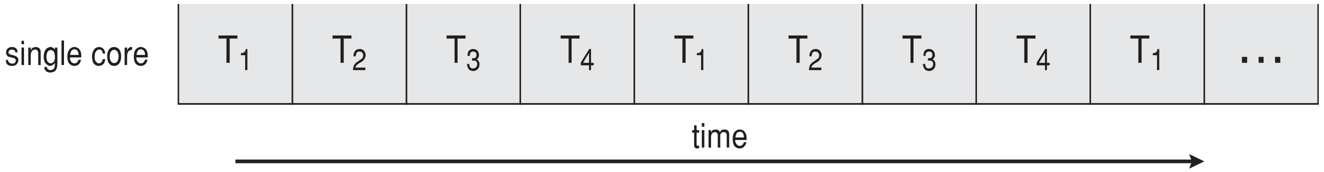
- ***Parallelism*** implies a system can perform more than one task/thread **simultaneously (on multiple CPUs)**

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

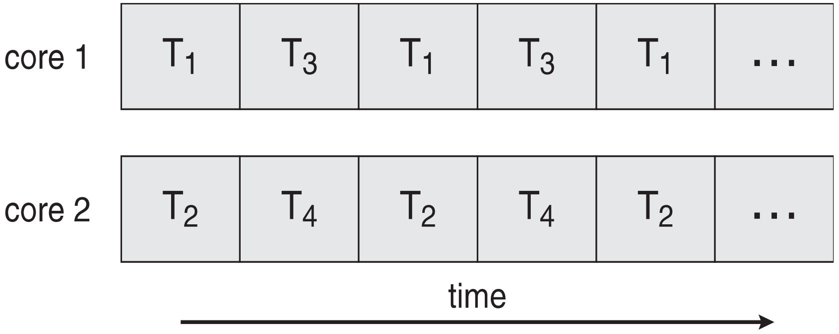
- In a single processor core: The scheduler provides time-sharing, aka concurrency
- In multiple CPU cores, the scheduler provides concurrency and parallelism.

Concurrency vs. Parallelism

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



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


core 1
T_1 T_3 T_1 T_3 T_1 …
core 2
T_2 T_4 T_2 T_4 T_2 …
time

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

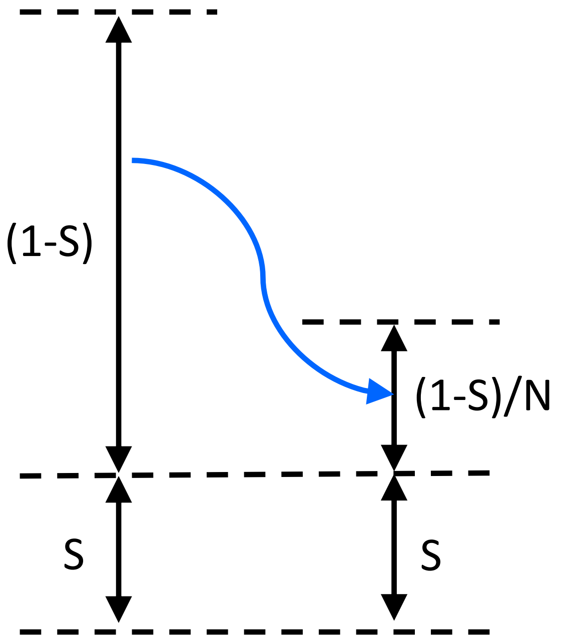
Amdahl’s Law

- 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-managed Threads and Kernel-managed Threads

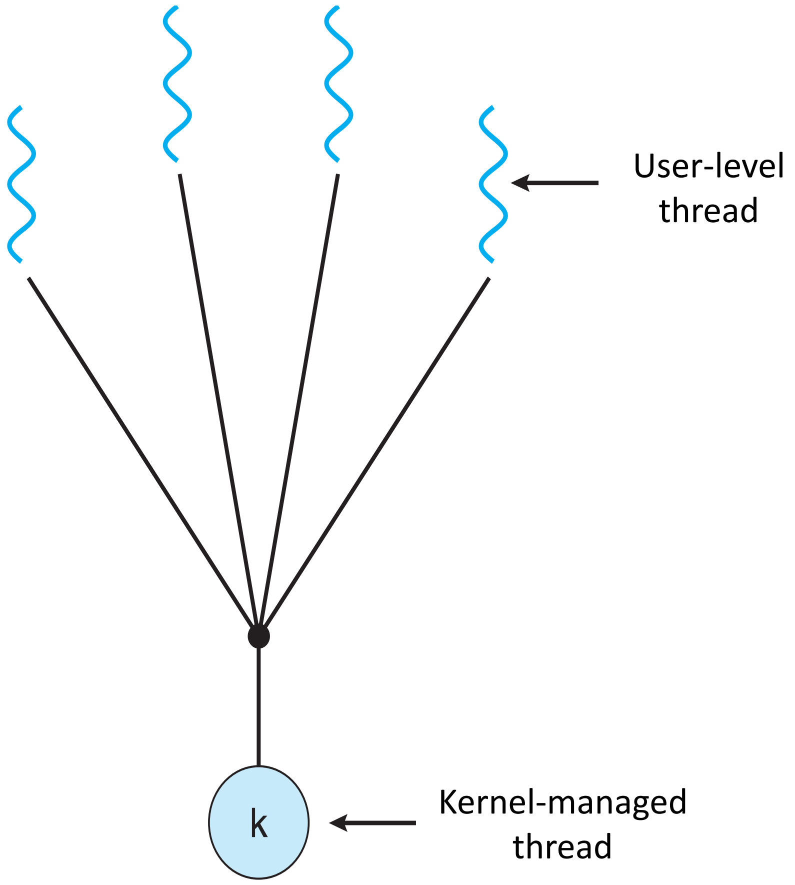
- **User-managed threads** – Thread management takes place using a threads library without OS support.
  - Kernel would treat the process as single-threaded and any blocking system call by one of the threads would end up blocking all the threads of that process.
- **Kernel-managed threads** - Supported 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-managed thread is not meant to indicate a thread executing kernel code, but rather a thread that is created and managed by the kernel.

4.3 Multithreading Models

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

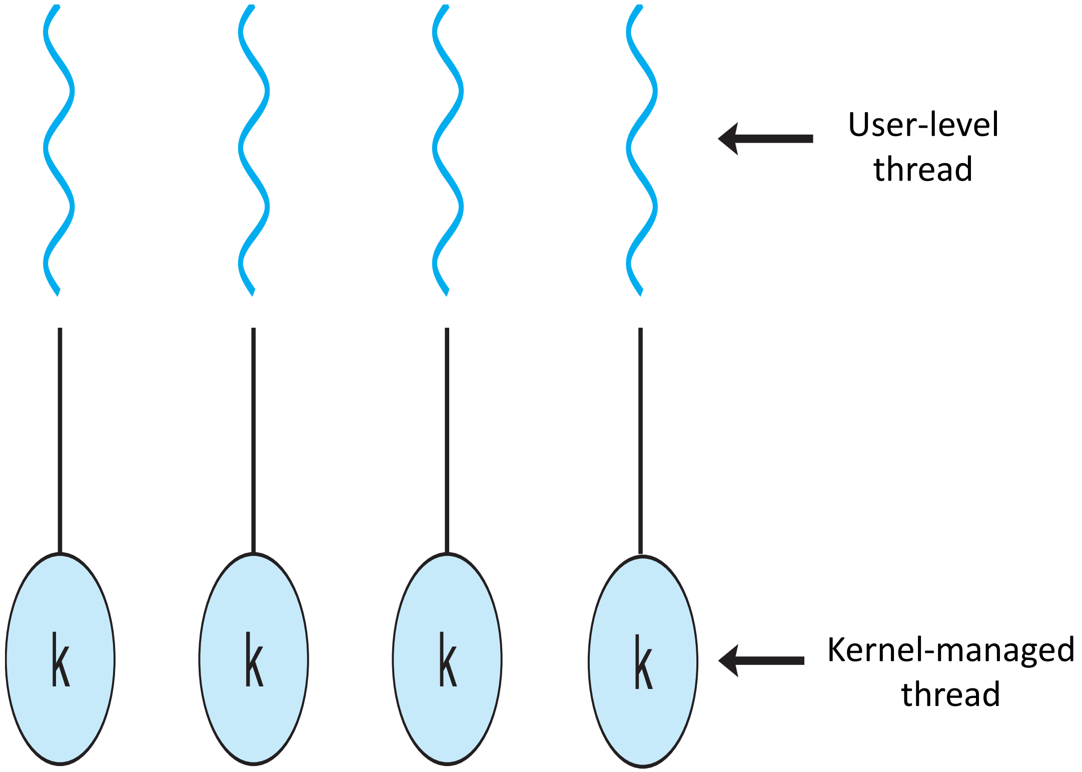
Many-to-One

- Many user-level threads mapped to single kernel-managed thread
- One thread **blocking** causes all to block
- Multiple threads may **not run in parallel** on multicore systems because only one user-level thread may be managed by the 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 a kernel-managed thread
- Creating a user-level thread creates a kernel-managed thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later

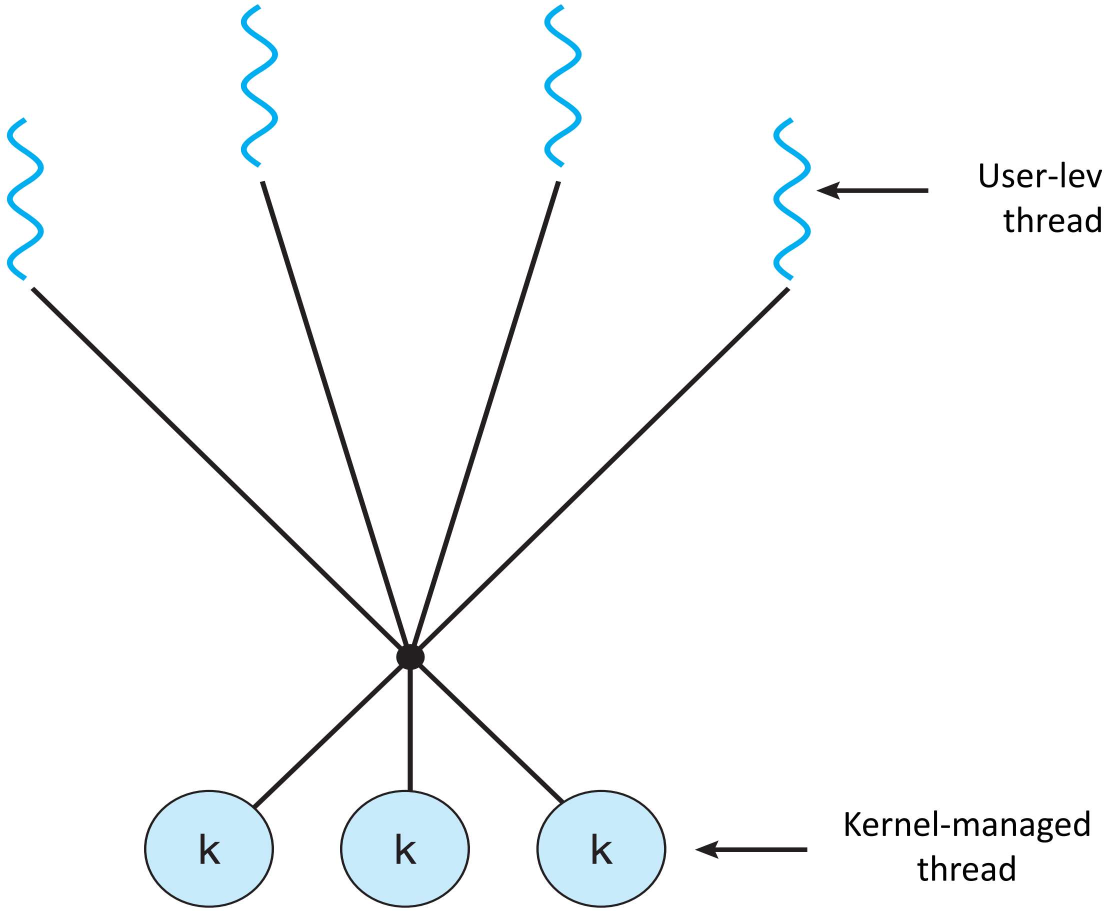


Many-to-Many Model

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

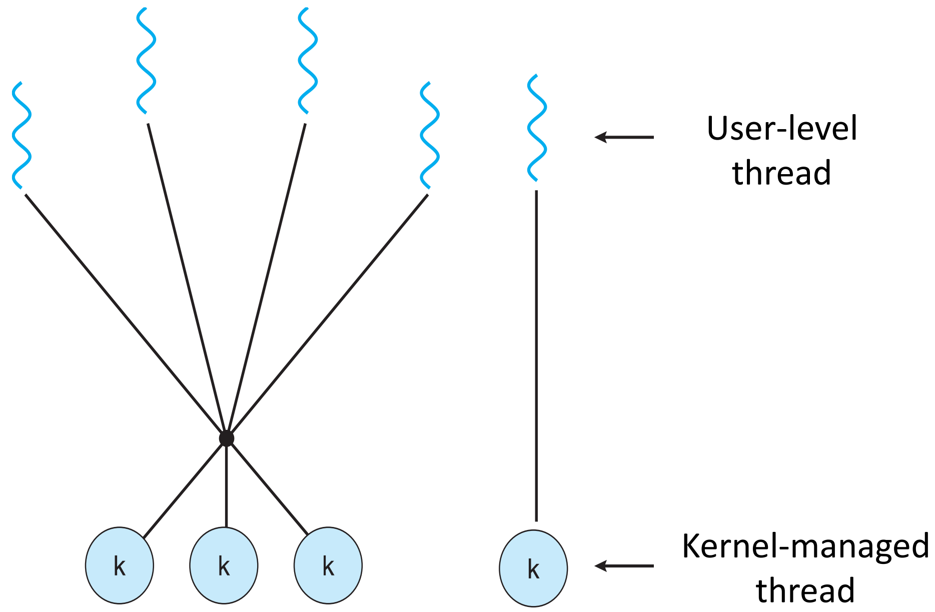
  # kernel-managed threads ≤ # user-level threads

- Allows the operating system to create a sufficient number of kernel-manged 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-level threads to map to many kernel-managed threads.
- But it also allows a one-to-one relationship on some user/kernel-managed thread pairs as shown on the diagram.
- Examples
  - HP-Unix
  - Tru64 Unix
  - Solaris prior to version 8



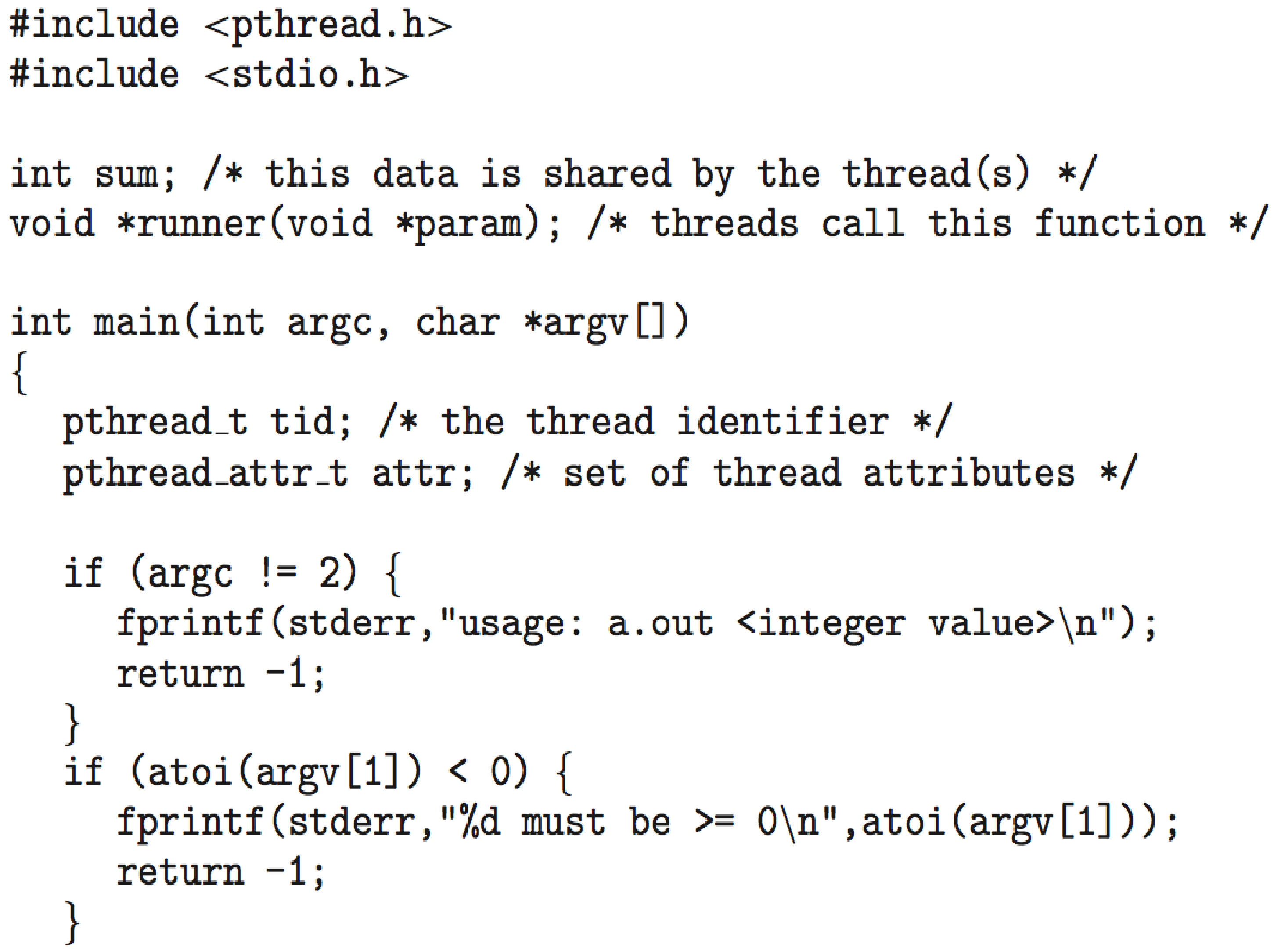
4.4 Thread Libraries

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

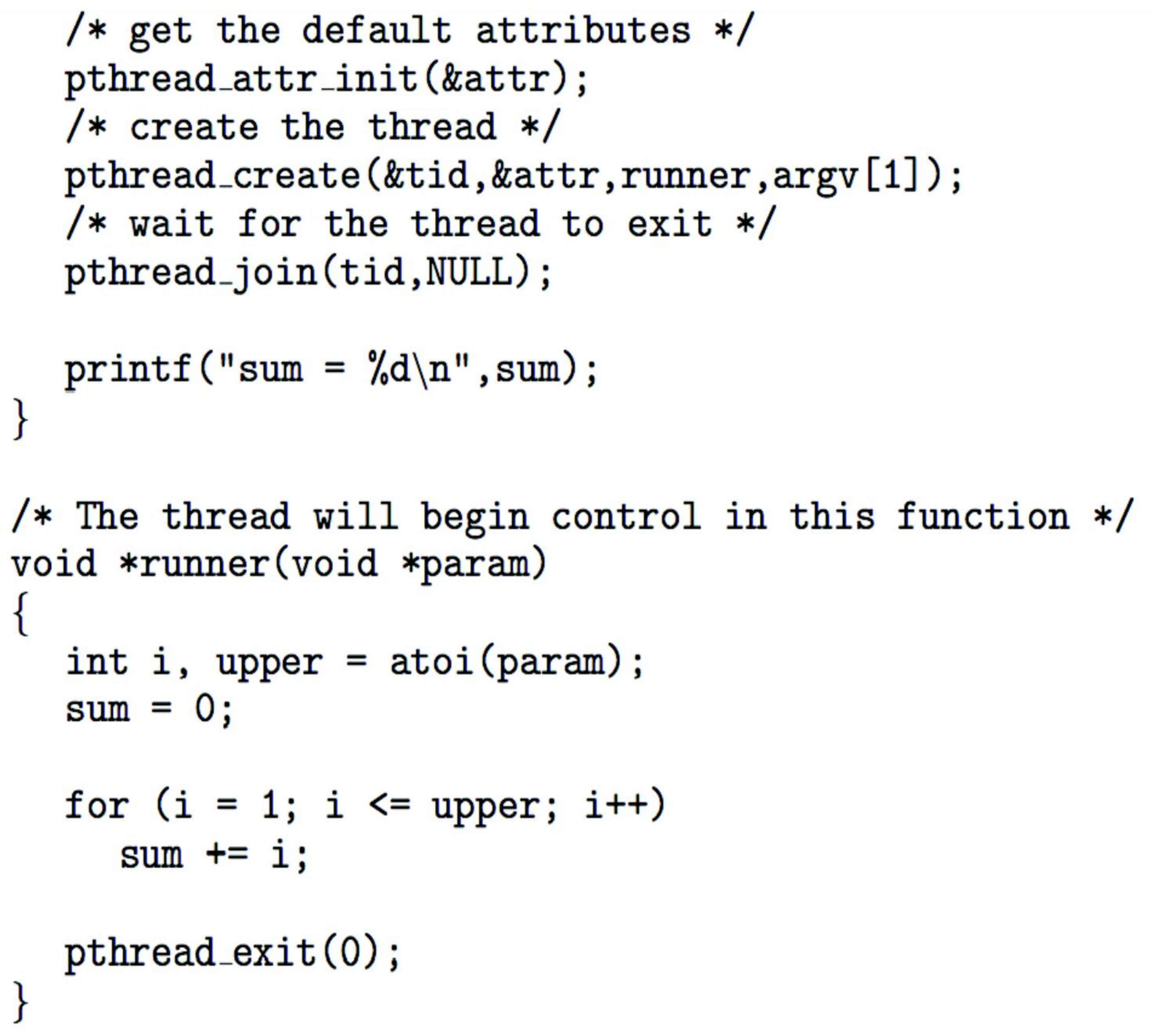
4.4.1 pthreads

- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- May be implemented either as user-managed or kernel-managed.
- ***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-managed threads

pthreads Example

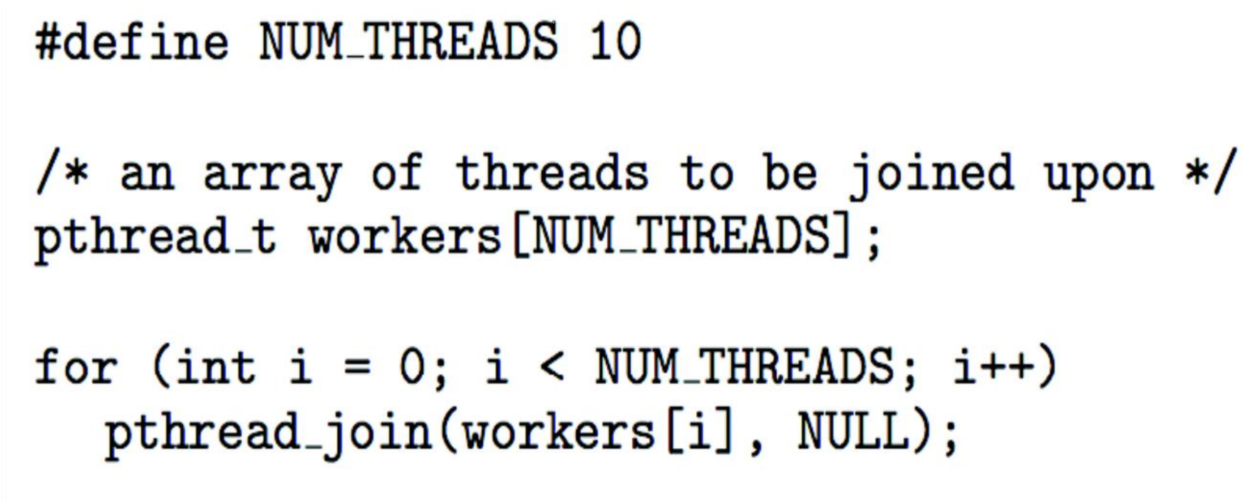


```
#include <pthread.h>
#include <stdio.h>

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);
}
```

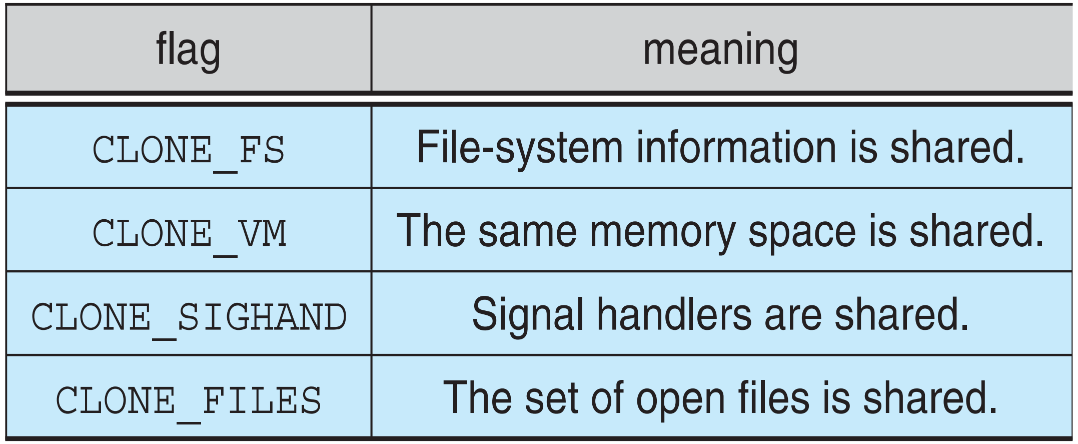
Pthreads Code for Joining 10 Threads



```
#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);
```

OS examples - Linux Threads

- These the **native Linux calls**,
    - pthreads provide a consistent interface across multiple Unix, Linux platforms
    - `pthread_create` calls are library calls that then invoke **clone()**, which is a native system call in Linux
- 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.

```
int clone(int (*fn)(void *), void *child_stack,
                 int flags, void *arg, ...
                 /*pid_t *ptid, struct user_desc *tls, pid_t *ctid*/ );
```

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

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**

Thread Pools – cont.

- **Use case:** Works very well for tasks that have a finite duration, i.e. tasks that start, do some work, then exit.

- **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.**

- **Every thread has a queue** and work is dispatched by enqueueing a function pointer and its parameter into that queue

Thread Pools – cont.

- **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 */
}
```

4.5.3 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
- Block is in “^{ }” – e.g.

```
    ˆ{ 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.

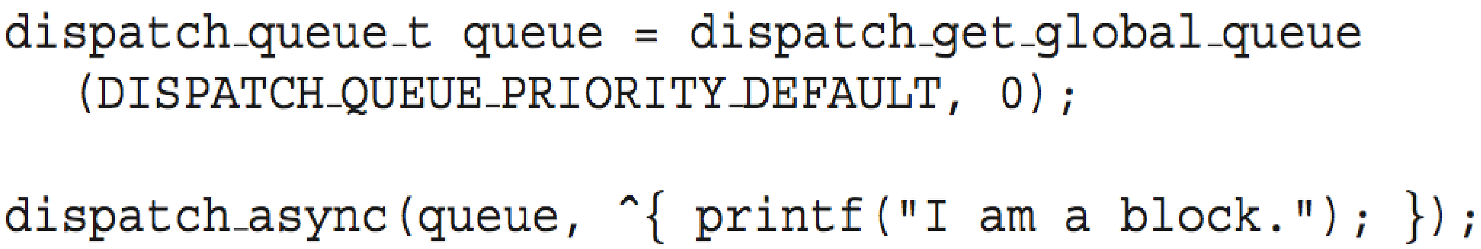
Grand Central Dispatch – cont.

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



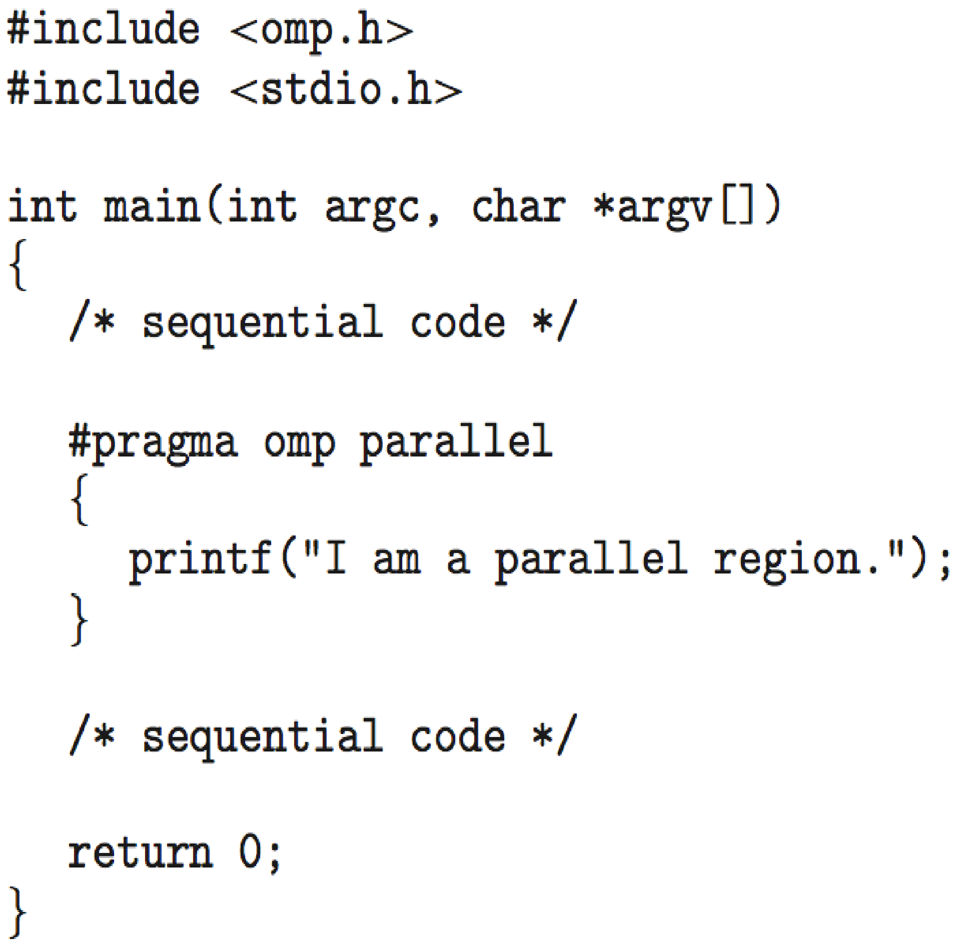
```
dispatch_queue_t queue = dispatch_get_global_queue
   (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

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

4.5.2 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.
- Available in gcc (versions higher than 4.6). You must use a flag to enable it:

```
gcc lab21.c -fopenmp -o lab21
```