3.4.2 Message passing systems

- Another mechanism for processes to communicate and to synchronize their actions

- Message system – processes communicate with each other without resorting to shared variables

- IPC facility (implemented by OS) provides (two theoretical) operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)

- The *message* size is either fixed or variable

Message Passing (Cont.)

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via `send(message)` and `receive(message)`
- Design choices:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - Is the size of a message that the link can accommodate fixed or variable?
  - What is the capacity of a link?
  - Is a link unidirectional or bi-directional?

Message Passing (Cont.)

- Implementation of communication link
  - On the physical level:
    - Via shared memory
    - Hardware bus or a communication network.
  - On the logical level:
    - A. Direct or indirect
    - B. Synchronous or asynchronous
    - C. Automatic or explicit buffering

A . Naming: Direct Communication

- Symmetric schemes: Processes must specify each other explicitly (via process identifier):
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*Q, message*) – receive a message from process Q
- Asymmetric schemes: Only sender names the recipient
  - **`send`** (*P, message*) – send a message to process P
  - **`receive`**(*&ID, message*) – receive a message from any process and when you receive a message indicate the name of the sender (in the variable *`ID`*).
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link is usually bi-directional
- Process identifiers may be names or integer numbers.
- Disadvantage is that a process identifier needs to be hardcoded, which thus requires recompilation if the identifiers change.

Naming: Indirect Communication

- Messages are directed to, and received from **ports or mailboxes**. Primitives are defined as:
  - **`send`**(*A, message*) – send a message to mailbox A
  - **`receive`**(*A, message*) – receive a message from mailbox A
- Properties of communication link
  - Each mailbox has a unique identifier. (name or number)
  - Link established only if processes share a common mailbox
  - A link **may** be associated with many processes
  - Each pair of processes **may** share several communication links
  - Link may be unidirectional or bi-directional

Naming: Indirect Communication – cont.

- A port (or mailbox) may be **owned by a process.** The port/mailbox is attached to a process and implemented inside its address space.
  - If the process exits, the mailbox is destroyed.
  - Unidirectional:
    - Owner (server) can only receive messages via the mailbox
    - User (client) sends message to the mailbox.
- Alternatively, a port may be **owned by the operating systems**, and thus the OS may need to provide operations such as:
  - Create a new port/mailbox
  - Send and receive messages through the port
  - Delete the port.
  - Unix implements two such ports;
    - pipes (unidirectional) and
    - TCP/IP ports (bidirectional)

Naming: Indirect Communication – cont.

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1 sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

B. Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** -- the sender is blocked until the message is received
  - **Blocking receive** -- the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send** -- the sender sends the message and continue
  - **Non-blocking receive** -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking and the link has zero buffering, we have a **rendezvous**
- NOTE: Throughout the course:

BLOCKING = SYNCHRONOUS

NON-BLOCKING = ASYNCHRONOUS

Synchronization (Cont.)

- Producer-consumer becomes trivial (i.e. no concern about how to manage a circular buffer

```
message next_produced;
while (true) {
  /* produce an item into
  next produced */
  send(next_produced);
}
```

```
message next_consumed;
while (true) {
 /* consume the item into
  next consumed */
  receive(next_consumed);
}
```

C. Buffering

- Queue of messages attached to the link.
- implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
  Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Shared memory vs message passing

- Message passing may be advantageous for exchanging smaller amounts of data since the synchronization overhead is avoided.
- Shared memory can be faster than message passing, particularly for larger amounts of data since no copying is involved. This is true only in cases where the synchronization overhead can be minimized.
- However, in multi-processing (or multicore) systems, research has shown that message passing is more efficient, even for larger blocks of data, due to the cache coherency overhead.

3.5 Examples of IPC Systems – POSIX shared memory

- A process first creates shared memory segment using **`shm_open`**

```
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

- Processes needing to communicate via shared memory must know the name of this memory-mapped object. The last parameter is the file permissions. The function returns a file descriptor.
- Same function is also used to open an existing segment to share it
- A process may then use **`ftruncate`** to set the size of the object (in bytes).
- **`mmap`** may then be used to map and obtain a pointer to the shared memory.
- Now the process could write to the shared memory, e.g.

```
   sprintf(shared_mem_ptr, "Writing to shared memory");
```

POSIX – cont.

- In Unix/Linux, the shared memory is abstracted as a file system located at /dev/shm
- shm (or shmfs) is also known as tmpfs. tmpfs means temporary file storage facility. It is intended to appear as a mounted file system, but one which uses memory instead of a persistent storage device.

- NOTE: It may be useful to install the man pages for POSIX

```
sudo apt-get install manpages-dev     // you probably have this already
sudo apt-get install manpages-posix-dev
```

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

IPC POSIX Consumer

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

3.6 Communications in Client-Server Systems

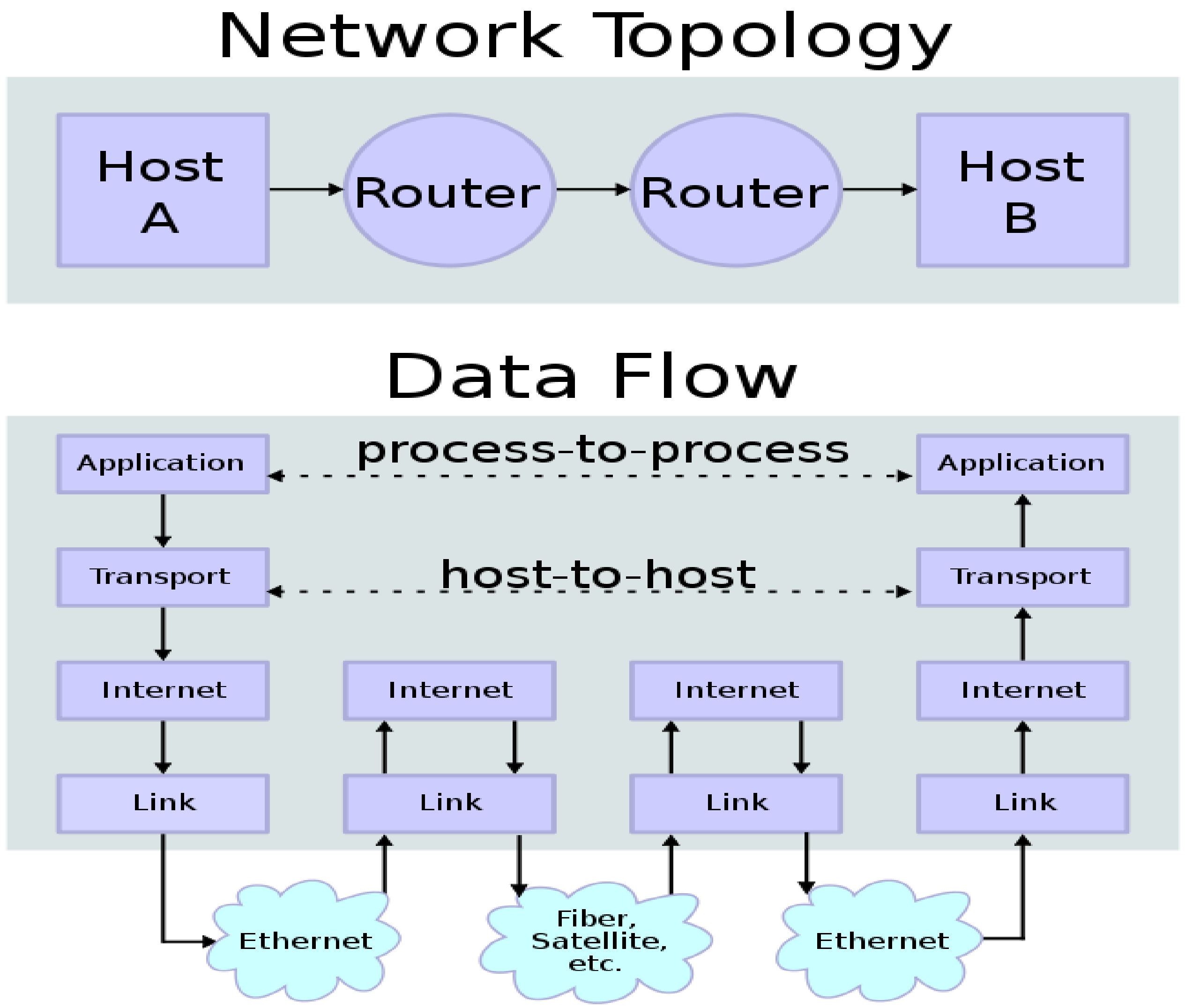
- Sockets
- Remote Procedure Calls
- Pipes

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

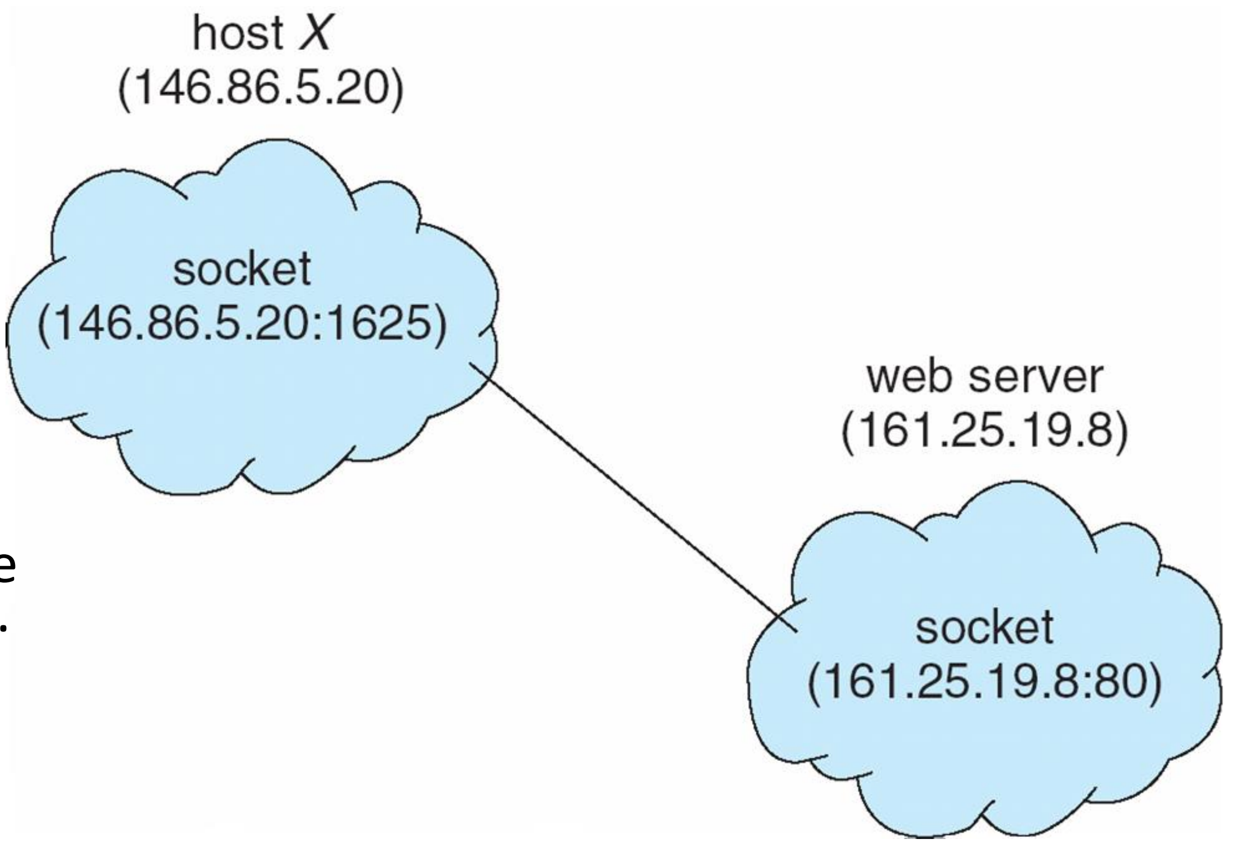


3.6.1 Sockets

- A **socket** is defined as an endpoint for communication
- A socket is the concatenation of an IP address and 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.
- 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 the IP address of the system and an arbitrary port number (above 1024, e.g. 1625), and thus 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.
- e.g. an http/web servers listens to port 80, while an ftp server listens to port 21.



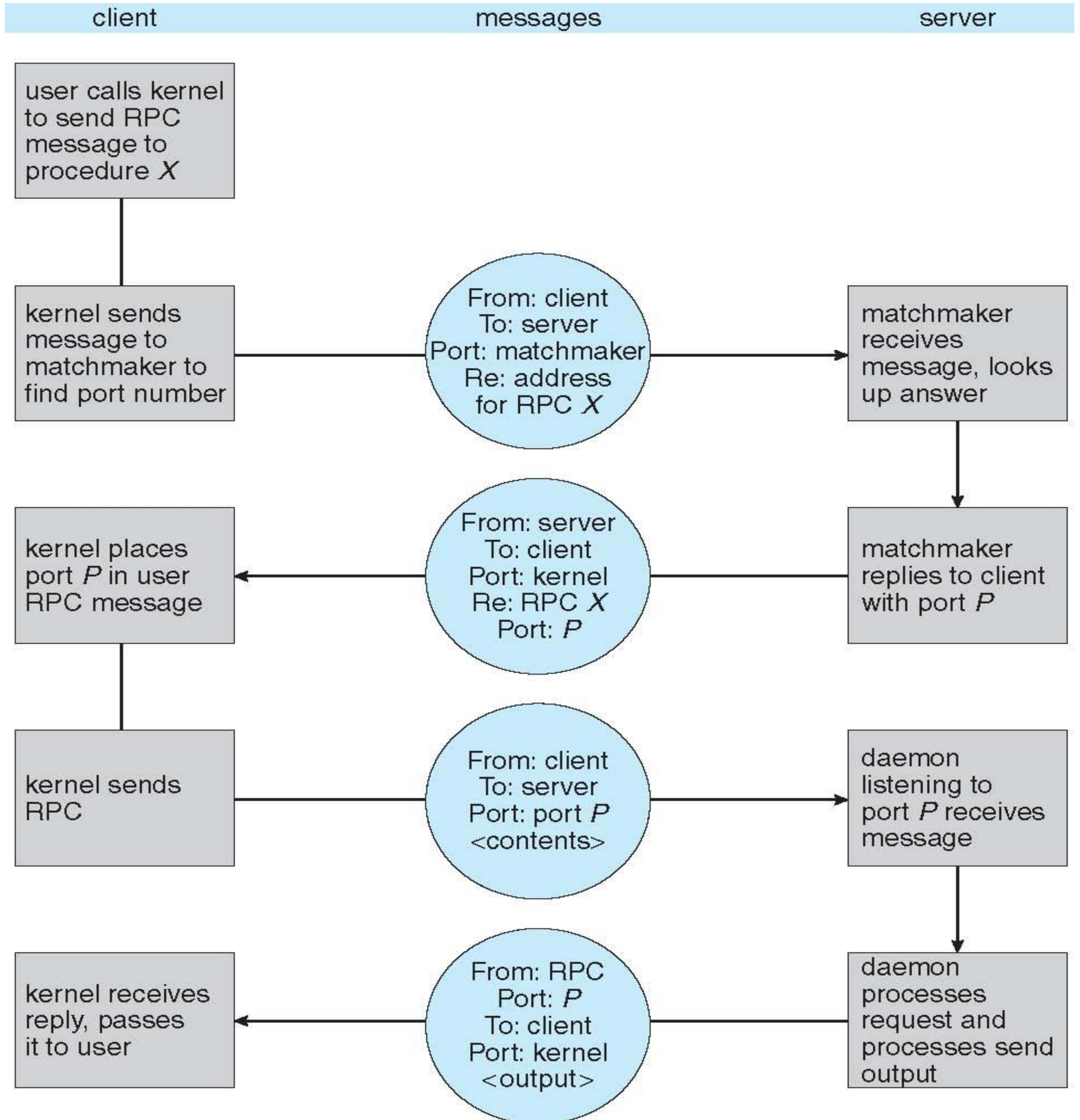
3.6.2 Remote Procedure Calls

- 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
- On Windows, stub code compile from specification written in **Microsoft Interface Definition Language** (**MIDL**)

Remote Procedure Calls (Cont.)

- Data representation handled via **External Data Representation** (**XDL**) format to account for 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

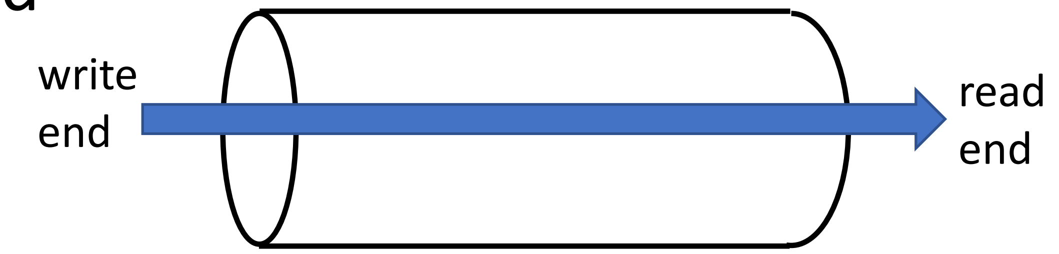

client
messages
server
user calls kernel to send RPC message to procedure *X*
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC *X*
matchmaker receives message, looks up answer
kernel places port *P* in user RPC message
From: server To: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

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

```
#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;
   }
```

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

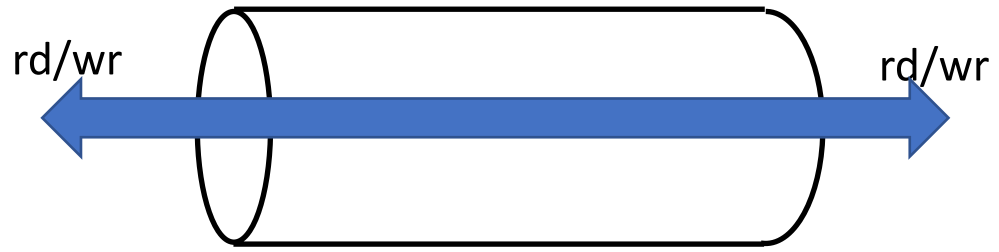
```
    else { /* child process*/
       /* close the write end since we are the consumer */
       close(fd[WRITE_END]);

       /* 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

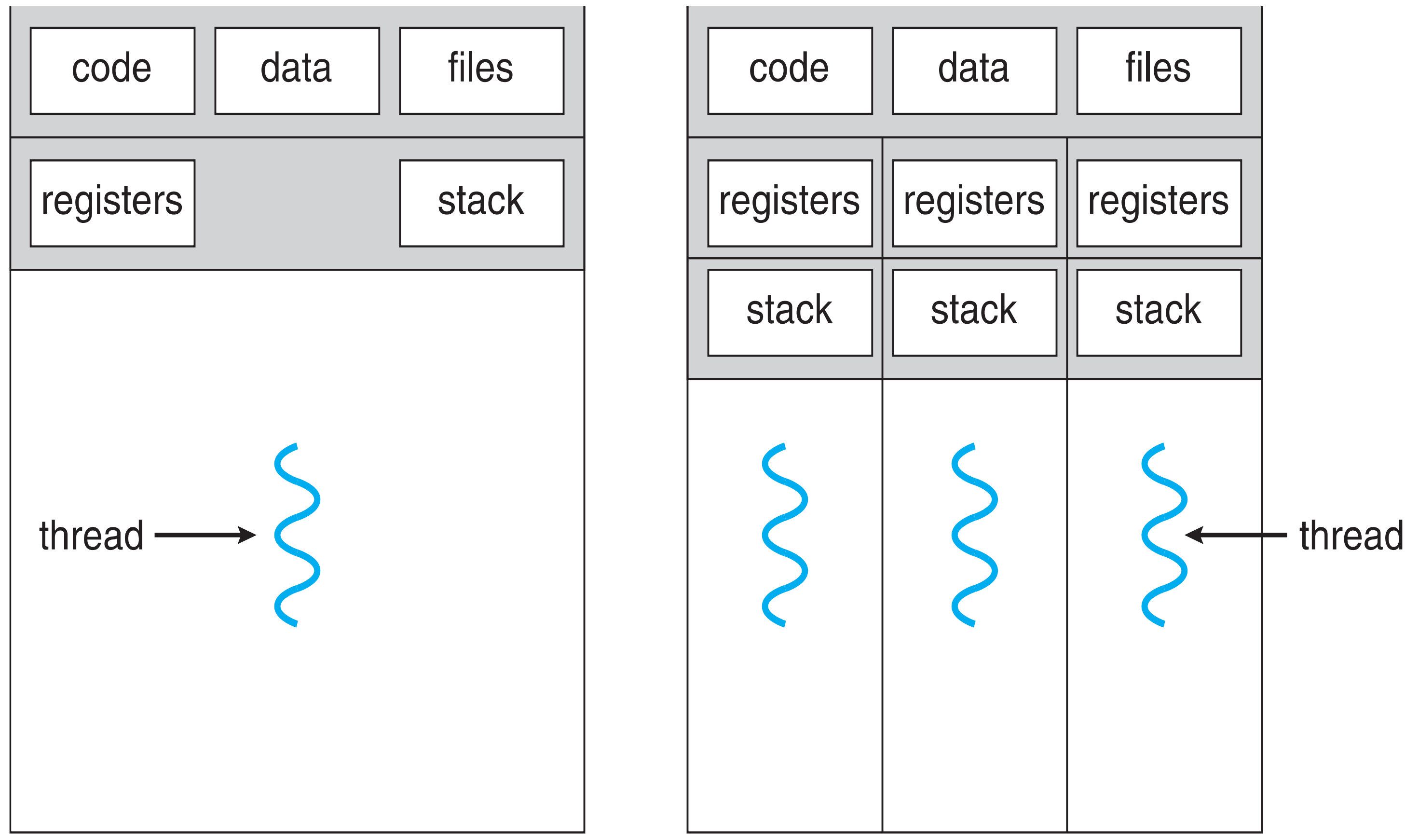


Named Pipes in Unix-like systems

- 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.)

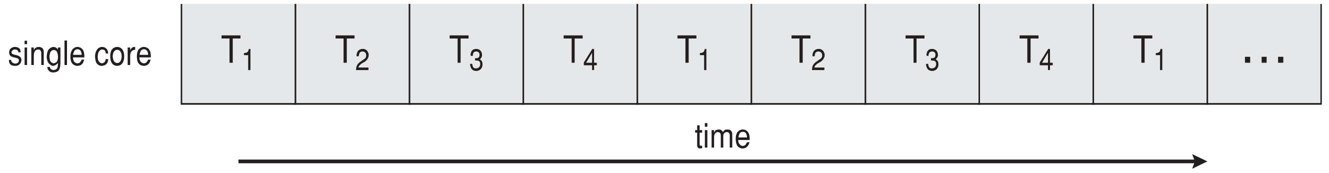
- ***Parallelism*** implies a system can perform more than one task/thread simultaneously (on multiple CPUs)

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

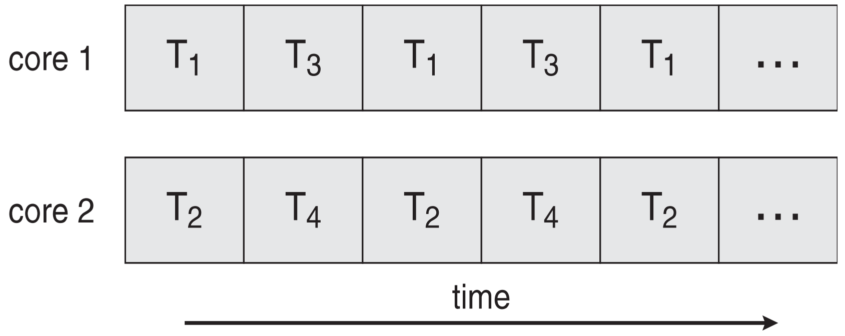
- In a single processor core: The scheduler provides time-sharing, aka concurrency
- In multiple CPU cores, the scheduler provides concurrency and parallelism.

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

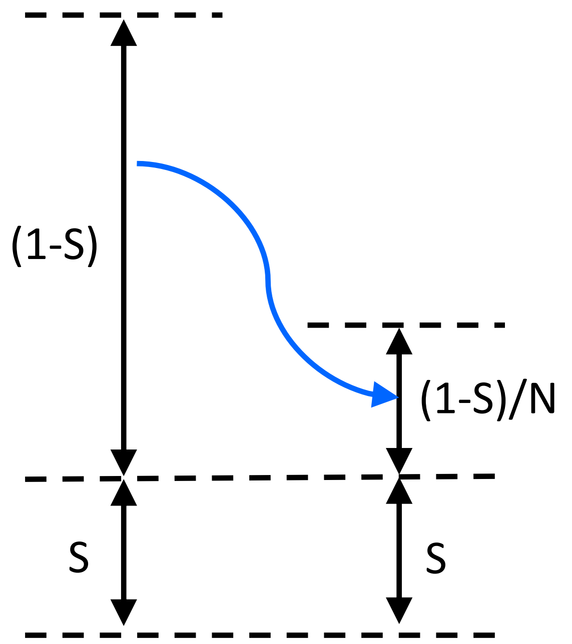
Amdahl's Law

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

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

- 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

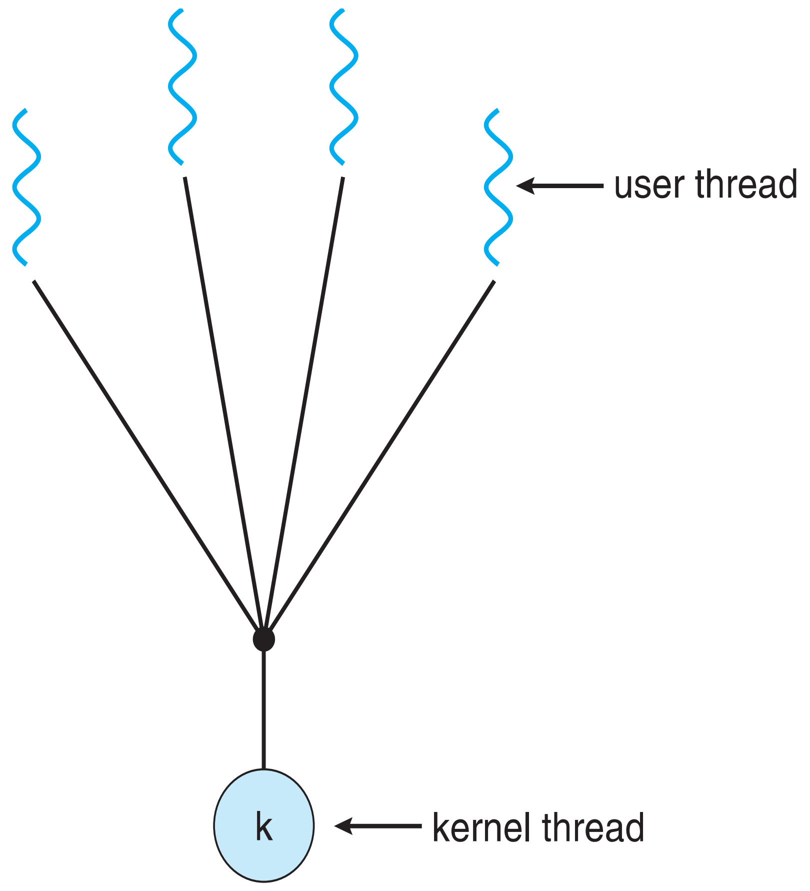
- **User threads** – Thread management takes place using a threads library without OS support.
  - Kernel would treat the process as single-threaded and any blocking system call by one of the threads would end up blocking all the threads of that process.
- **Kernel threads** - Supported and managed by the OS Kernel. Kernel support exists for most well-known Oses, e.g.:
  - Windows
  - Solaris
  - 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

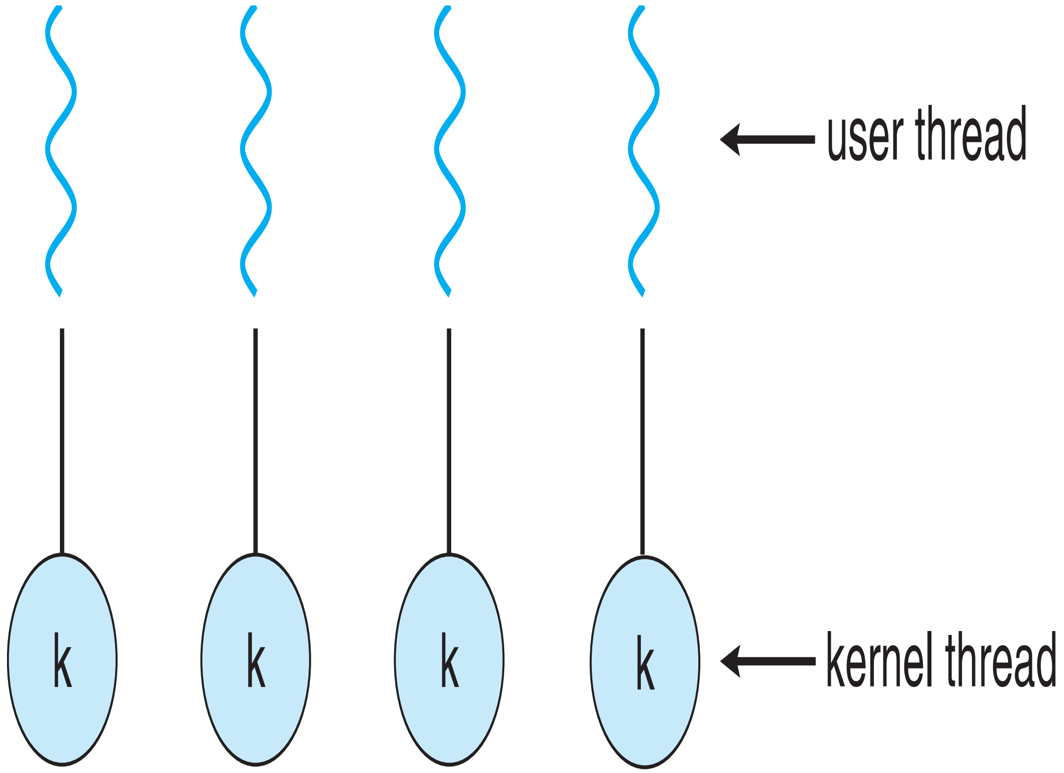
Many-to-One

- 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

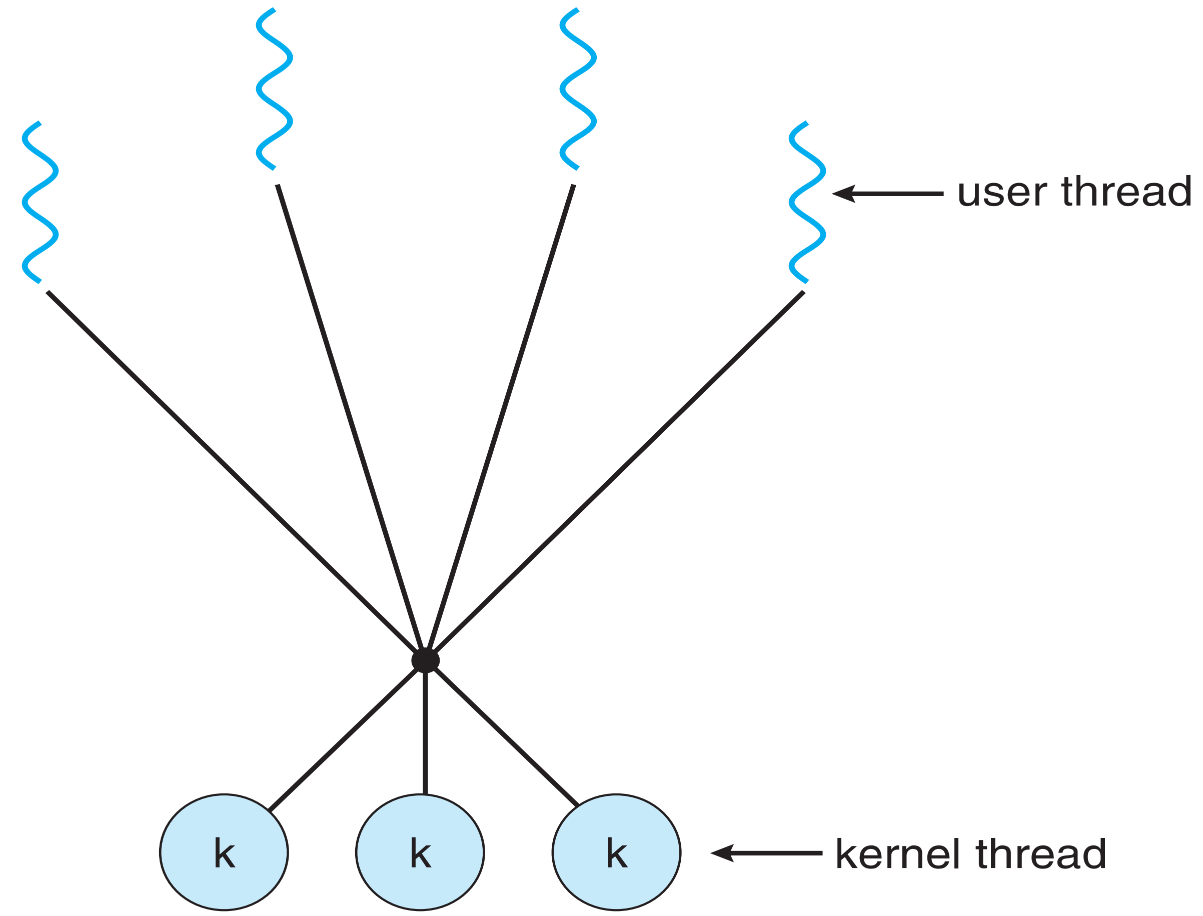


Many-to-Many Model

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

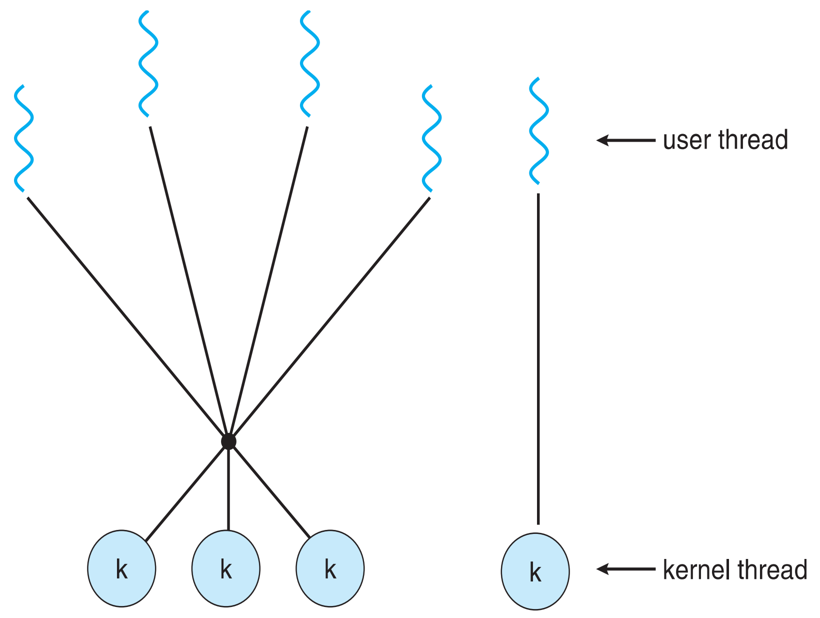
  # kernel threads ≤ # user threads

- 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-UX
  - 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)
    - Windows threads (kernel threads)
    - Java threads (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)

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);
    }
}
```