CS370 Operating Systems

Colorado State University
Yashwant K Malaiya

Fall 2016 Lecture 10

Slides based on

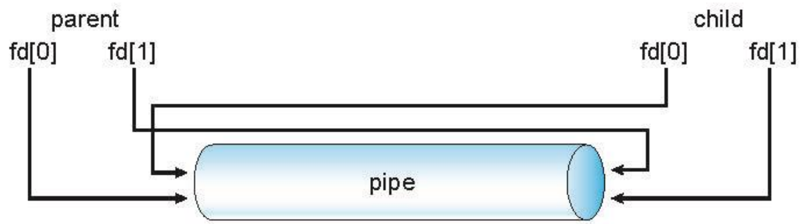
- Text by Silberschatz, Galvin, Gagne
- Various sources

FAQ

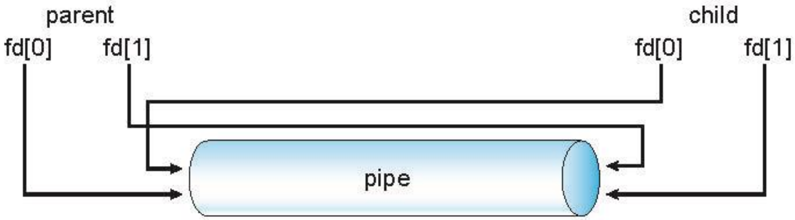
- Context switch in real-time processes (with fixed response times)?
- Logical links? Software abstractions
- POSIX shared memory example: see canvas discussions
- Isn't message passing also shared memory?
- What kind of messages are sent?
- Is a pipe a form of direct communication?
- What are pipes? Functions, arrays, strings? Special kind of files
- Synchronous (blocking) vs asynchronous (non-blocking)
- When to use shared memory instead of message passing?
- Can a pair of processes have multiple links using the same mailbox?
- When are sockets used? all internet connections

Ordinary Pipes

- Pipe is a special type of file.
- Inherited by the child
- Must close unused portions of the the pipe



UNIX pipe example




```
#define READ_END     0
#define WRITE_END    1

          int fd[2];
```

create the pipe:

```
          if (pipe(fd) == -1) {
                    fprintf(stderr,"Pipe failed");
                    return 1;
```

fork a child process:

```
          pid = fork();
```

Child inherits the pipe

parent process:

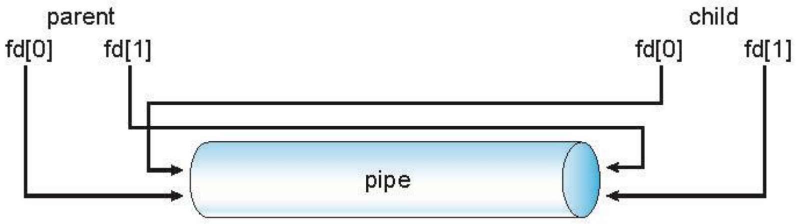
```
                    /* close the unused end of the pipe */
                    close(fd[READ_END]);

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

                    /* close the write end of the pipe */
                    close(fd[WRITE_END]);
```


Colorado State University

UNIX pipe example



child process:

```
/* close the unused end of the pipe */
close(fd[WRITE_END]);

/* read from the pipe */
read(fd[READ_END], read_msg, BUFFER_SIZE);
printf("child read %s\n",read_msg);

/* close the write end of the pipe */
close(fd[READ_END]);
```

Named Pipes

- Named Pipes (termed FIFO) 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

Chapter 4: Threads

Objectives:

- Thread—basis of multithreaded systems
- APIs for the Pthreads and Java thread libraries
- implicit threading, multithreaded programming
- OS support for threads

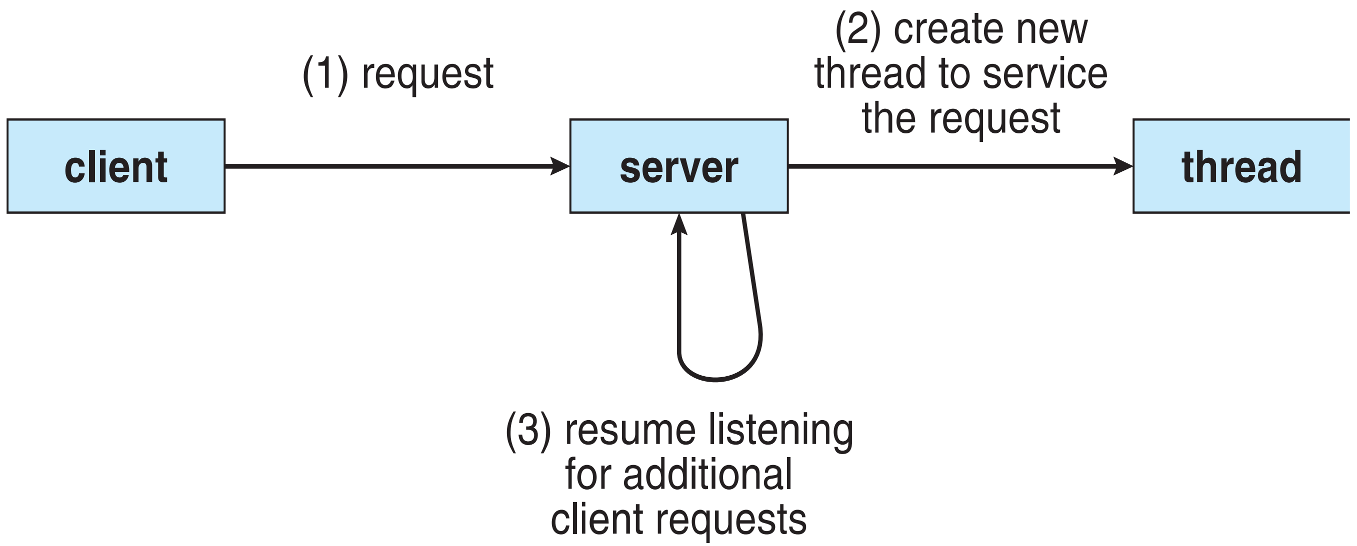
Chapter 4: Threads

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

Modern applications are multithreaded

- Most modern applications are multithreaded
  - Became common with GUI
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

Multithreaded Server Architecture



Benefits

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation (10-100 times), thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multiprocessor architectures

Multicore Programming

- **Multicore** or **multiprocessor** systems putting pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies a system can perform more than one task simultaneously
  - Extra hardware needed for parallel execution
- ***Concurrency*** supports more than one task making progress
  - Single processor / core: scheduler providing concurrency

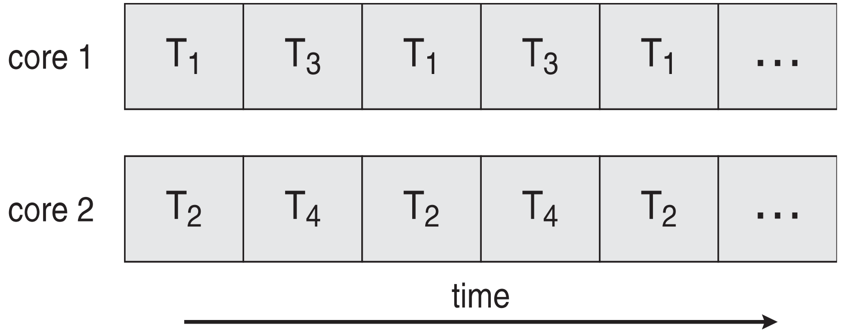
Multicore Programming (Cont.)

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

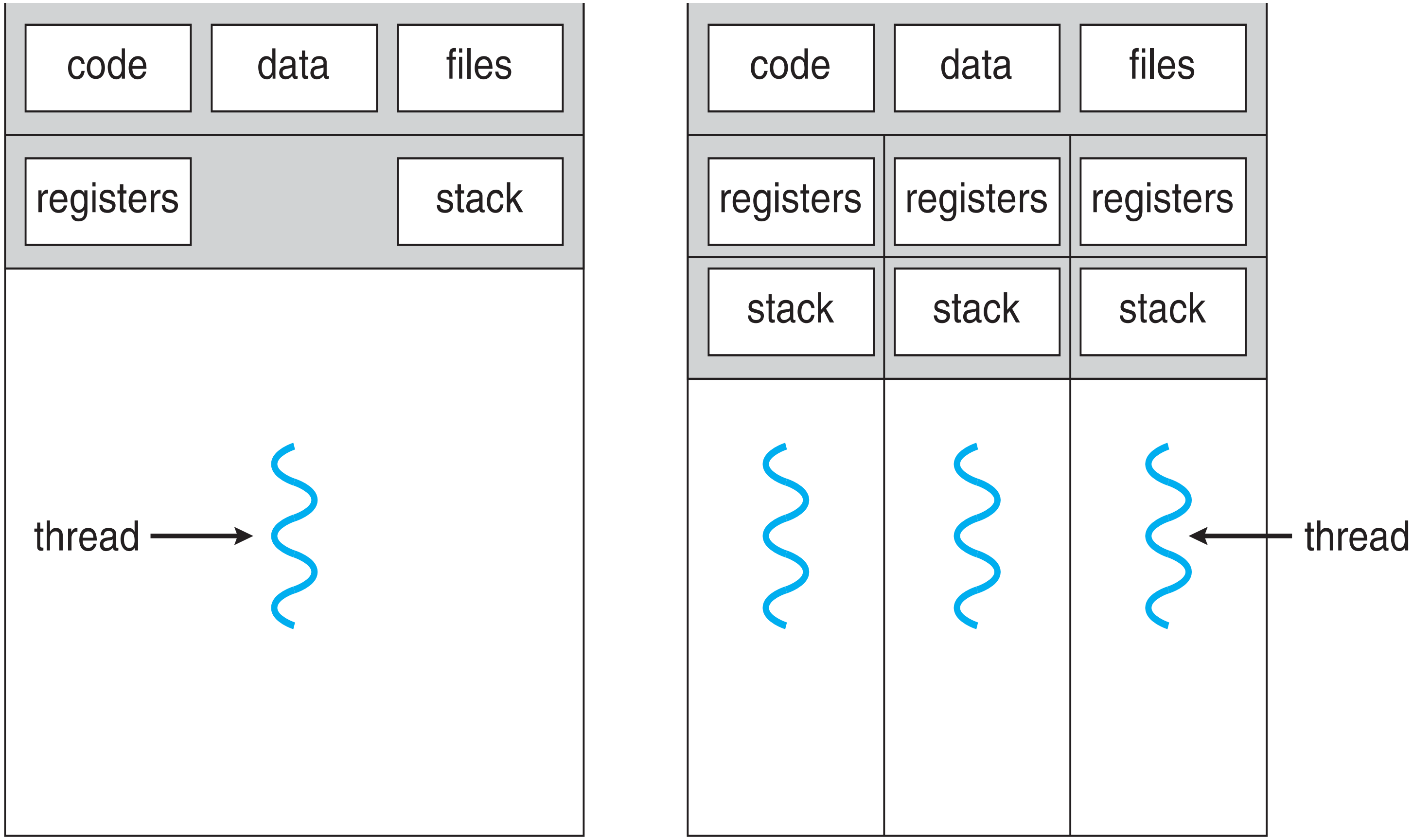
Concurrency vs. Parallelism

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

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



Single and Multithreaded Processes



single-threaded process

multithreaded process

Process vs Thread

- All threads in a process have same address space (text, data, open files, signals etc.), same global variables
- *Each thread has its own*
  - *Thread ID*
  - *Program counter*
  - *Registers*
  - *Stack: execution trail, local variables*
  - *State (running, ready, blocked, terminated)*
- *Thread is a schedulable entity*

Amdahl's Law

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

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

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

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

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

User Threads and Kernel Threads

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

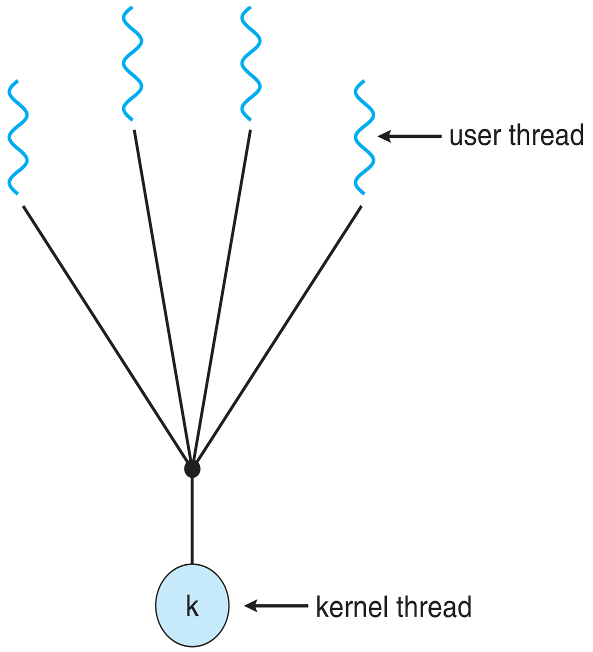
Multithreading Models

How do kernel threads support user process threads?

- Many-to-One
- One-to-One (now common)
- Many-to-Many

Many-to-One

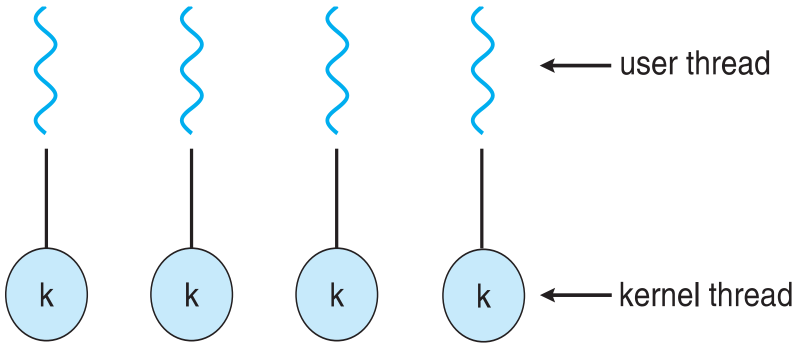
- Many user-level threads mapped to single kernel thread (thread library in user space)
- One thread blocking causes all to block
- Multiple threads may not run in parallel on muticore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads for Java** **1996**
  - **GNU Portable Threads** **2006**



Colorado State University

One-to-One

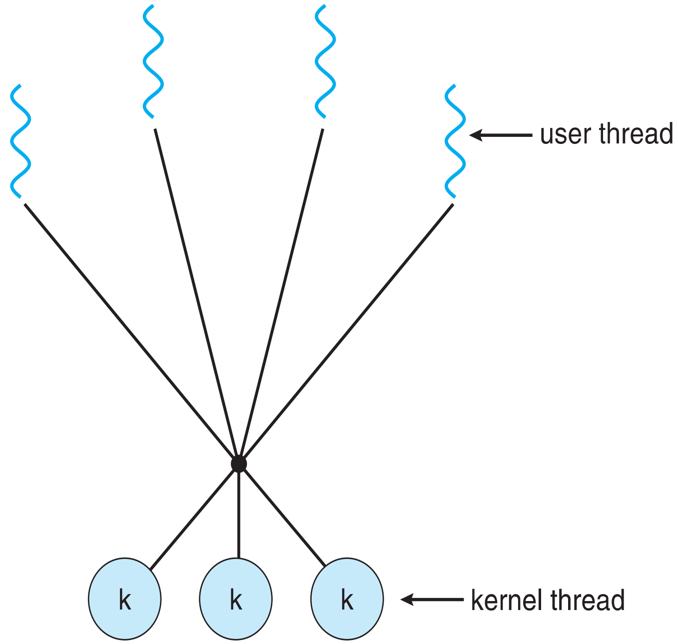
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux
  - Solaris 9 and later



Colorado State University

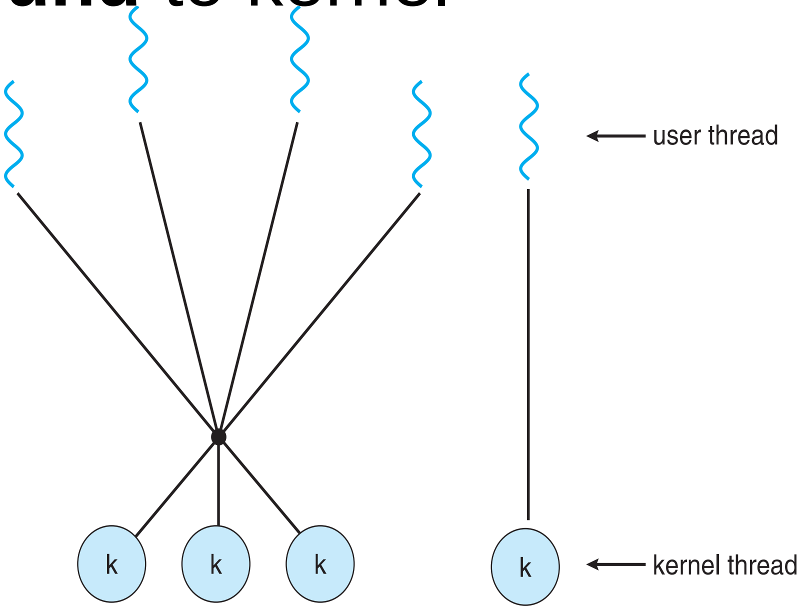
Many-to-Many Model

- Allows many user level threads to be mapped to smaller or equal number of kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9 2002-3
- Windows with the *ThreadFiber* package NT/2000



Two-level Model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX -2006
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier



Colorado State University

Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization 1991
- ***Specification***, not ***implementation***
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Some Pthread management functions

POSIX function	Description
pthread_cancel	Terminate a thread
pthread_create	Create a thread
pthread_detach	Set thread to release resources
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

- Return 0 if successful

POSIX: Thread creation pthread_create()

- Automatically makes the thread runnable without a start operation
- Takes 3 parameters:
  - Points to ID of newly created thread
  - Attributes for the thread
    - Stack size, scheduling information, etc.
  - Name of function that the thread calls when it begins execution

```
/* create the thread */
pthread_create(&tid, &attr, runner, argv[1]);
```

POSIX: Detaching and Joining

- pthread_detach()
  - Sets internal options to specify that storage for thread can be reclaimed when it exits
  - 1 parameter: Thread ID of the thread to detach
- Undetached threads don't release resources until
  - Another thread calls pthread_join for them
  - Process exits
- pthread_join
  - Takes ID of the thread to wait for
  - Suspends calling thread till target terminates
  - Similar to waitpid at the process level

```
pthread_join(tid, NULL);
```

POSIX: Exiting and cancellation

- If a process calls exit, **all** threads terminate
- Call to pthread_exit causes only the calling thread to terminate

pthread_exit(0)

- Threads can force other threads to return through a *cancellation* mechanism
  - pthread_cancel: takes thread ID of target
  - Depends on *type* and *state* of thread

Pthreads Example (next 2 slides)

- This process will have two threads
  - Initial/main thread to execute the main ( ) function. It crates a new thread and waits for it to finish.
  - A new thread that runs function runner ( )
    - It will get a parameter, an integer, and will compute the sum of all integers from 1 to that number.
    - New thread leaves the result in a global variable **sum**.
  - The main thread prints the result.

Pthreads Example Pt 1

```
#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;
  }
```

Pthreads Example (Cont.)

```
   /* 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);
}
```

Compile using
gcc thrd.c -lpthread

Demo here

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);
```