OPERATIONS ON PROCESSES

Process creation

- fork()
- exec()
- The argument vector

Process deletion

- kill()
- signal()

PROCESS CREATION

Two basic system calls

- fork() creates a carbon-copy of calling process sharing its opened files
- exec () overwrites the contents of the process address space with the contents of an executable file

FORK()

The first process of a system is created when the system is booted

- e.g., init()

All other processes are forked by another process (parent process)

- They are said to be children of the process that created them.

When a process forks, OS creates an identical copy of forking process with

- a new address space
- a new PCB

The only resources shared by the parent and the child process are the opened files

FORK()

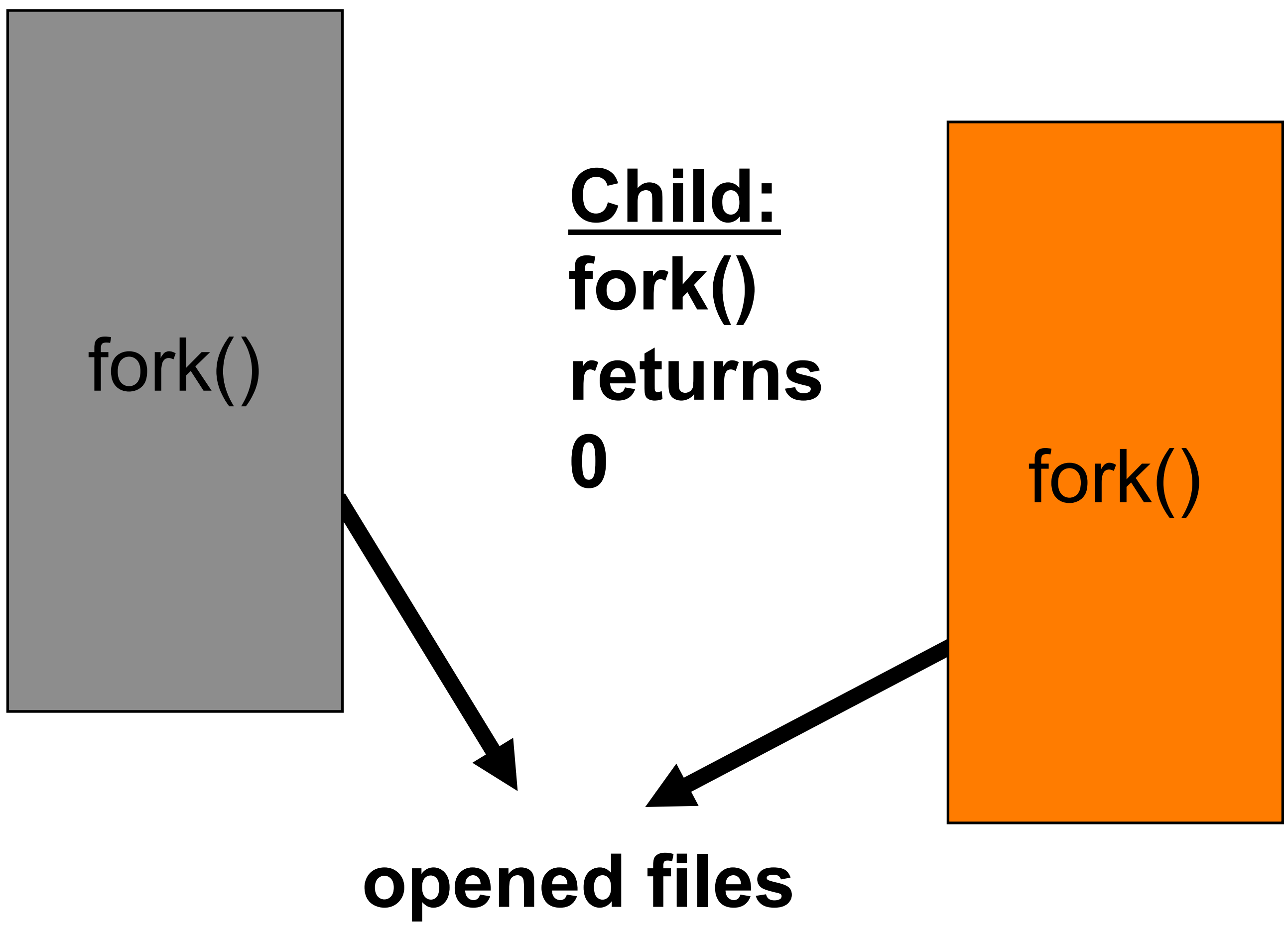
fork() call returns twice!

- Once in address space of child process
  - Function return value is 0 in child
- Once in address space of parent process
  - Function return value is process ID of child in parent

FORK() ILLUSTRATED

Parent:
fork() returns PID of child

Child:
fork() returns 0

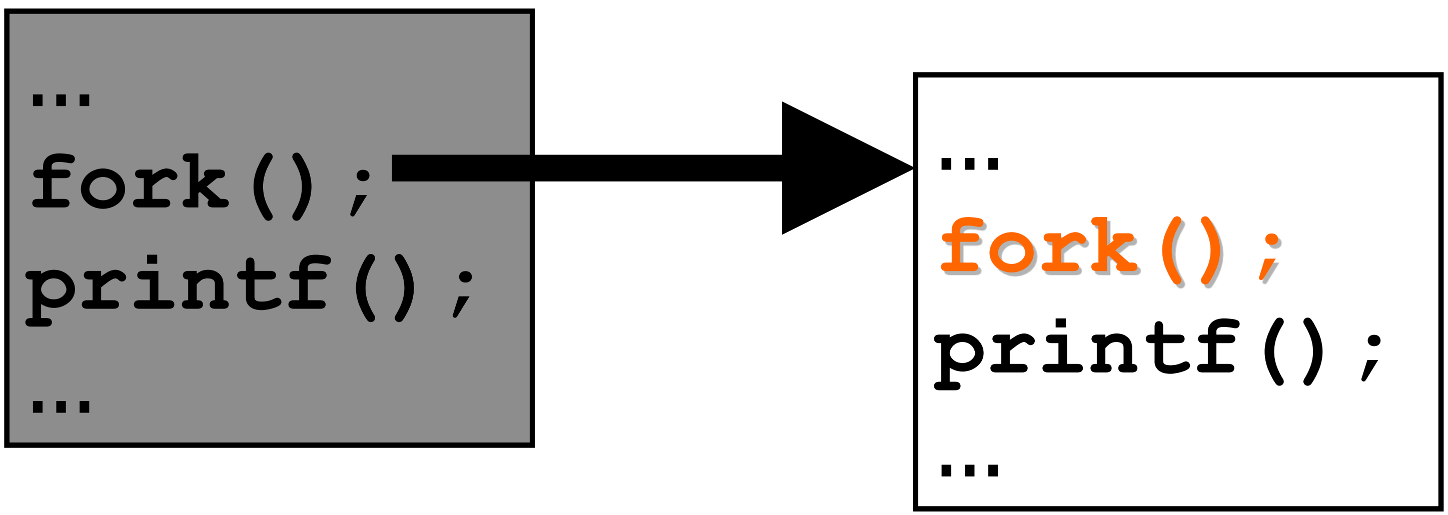


FIRST EXAMPLE

The program

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

int main() {
	pid_t pid = fork();
	if (pid < 0) { /* error occurred */
		fprintf(stderr, "Fork Failed");
		return 1;
	}
	printf("pid = %d, Hello world!\n", pid);
	return 0;
}
```

HOW IT WORKS

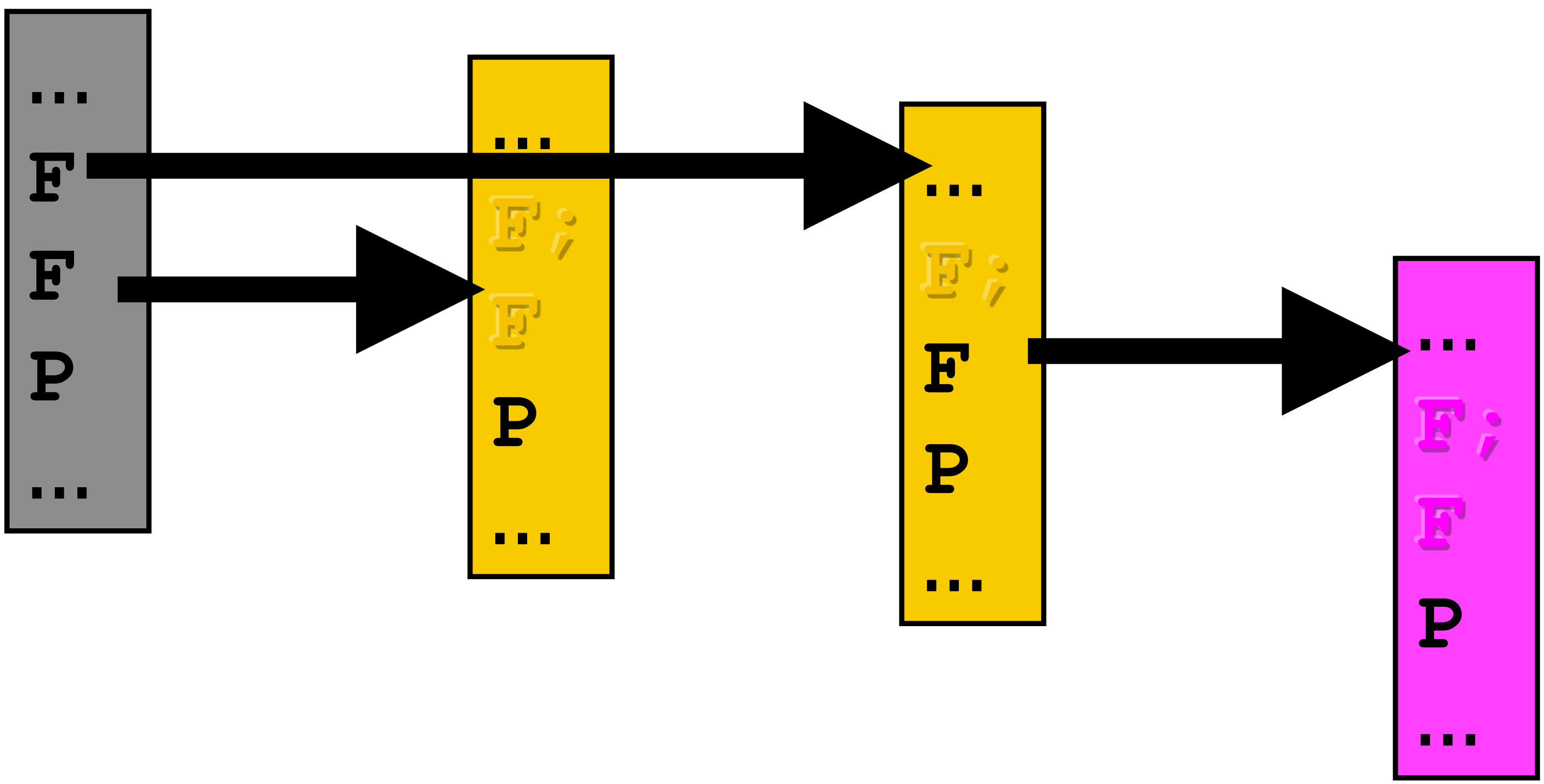


SECOND EXAMPLE

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

int main() {
	pid_t pid = fork();

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

	pid_t pid1 = fork();

	printf("Hello world!\n");
	return 0;
}
```

how many processes?

HOW IT WORKS



DISTINGUISHING CHILD AND PARENT PROCESSES

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main() {
	pid_t pid;
	/* fork a child process */
	pid = fork();
	if (pid < 0) { /* error occurred */
		fprintf(stderr, "Fork Failed");
		return 1;
	} else if (pid == 0) { /* child process */
		printf(“I am a child\n”);
	} else { /* parent process */
		/* parent will wait for the child to complete */
		wait(NULL);
		printf("Child Complete\n");
	}
	return 0;
}
```

fork()
- return 0 in child process; return PID of the child in the parent

wait()
- Waits for the completion of any child

WAIT()

wait() used to wait for the state changes in a child of the calling process

- Blocked until a child changes its status

UNIX keeps in its process table all processes that have terminated but their parents have not yet waited for their termination

- They are called zombie processes

EXEC

Whole set of exec() system calls

Most interesting are

- execv(pathname, argv)
- execve(pathname, argv, envp)
- execvp(filename, argv)

All exec() calls perform the same basic tasks

- Erase current address space of process
- Load specified executable

- execlp(const char *file, const char *arg0, ... /*, (char *)0 */);

PUTTING EVERYTHING TOGETHER

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

int main() {
	pid_t pid;
	/* fork a child process */
	pid = fork();
	if (pid < 0) { /* error occurred */
		fprintf(stderr, "Fork Failed");
		return 1;
	} else if (pid == 0) { /* child process */
		execlp("/bin/ls", "ls", "-l", NULL);
	} else { /* parent process */
		/* parent will wait for the child to complete */
		wait(NULL);
		printf("Child Complete\n");
	}
	return 0;
}
```

OBSERVATIONS

Mechanism is quite costly

- fork() makes a complete copy of parent address space
    - very costly in a virtual memory system
- exec() thrashes that address space

Berkeley UNIX introduced cheaper vfork()

- Shares the parent address space until the child does an exec()

PROCESS TERMINATION

When do processes terminate?

- exit(), "running off end", invalid operations, other process kills it

Resources must be de-allocated

- E.g., PCB, open files
- Memory (address space) that is in use (if no other threads)

What happens when parent dies?

- Children can die ("cascading termination")
- Children can remain executing

What happens when a child terminates

- Parent may be notified

EXAMPLES OF PROCESS TERMINATION

Unix-ish systems (e.g., Mac OS X, Linux)

- E.g., process calls _exit() or exit() itself, or another process calls kill(pid, SIGKILL)
- Parent: Child terminating sends SIGCHLD signal to parent (does not terminate parent)
- Children: of terminating process are inherited by process 1, “init” (BUT, children terminate on Unix!)

Windows system

- e.g., ExitProcess called by the process or another process calls TerminateProcess with a handle to the process
- Children: child processes continue to run

PARENT DIES BEFORE CHILD

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

int main() {
        pid_t pid;
        /* fork a child process */
        pid = fork();
        if (pid < 0) { /* error occurred */
                fprintf(stderr, "Fork Failed");
                return 1;
        } else if (pid == 0) { /* child process */
                sleep(10);
                printf("Child terminating\n");
        } else { /* parent process */
                printf("Press enter to continue in parent\n");
                getchar();
        }
        return 0;
}
```

COOPERATING PROCESSES

Any process that does not share data with any other process is independent"

Processes that share data are cooperating

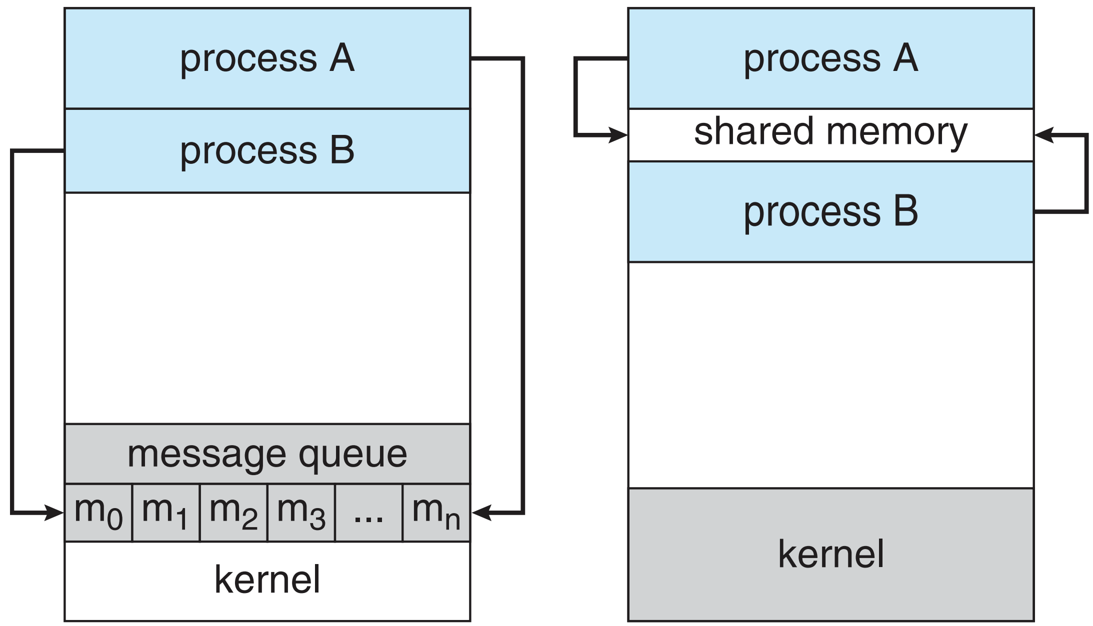
Why cooperation?

- Software engineering issues
  - Sometimes it is natural to divide a problem into multiple processes
  - Often, the parts of a program need to cooperate
  - Modularity
- Run-time issues
  - Computational speedups (e.g., multiple CPU's)
  - Convenience (e.g., printing in background)

MECHANISMS

Shared memory

Message passing



Message passing   Shared memory

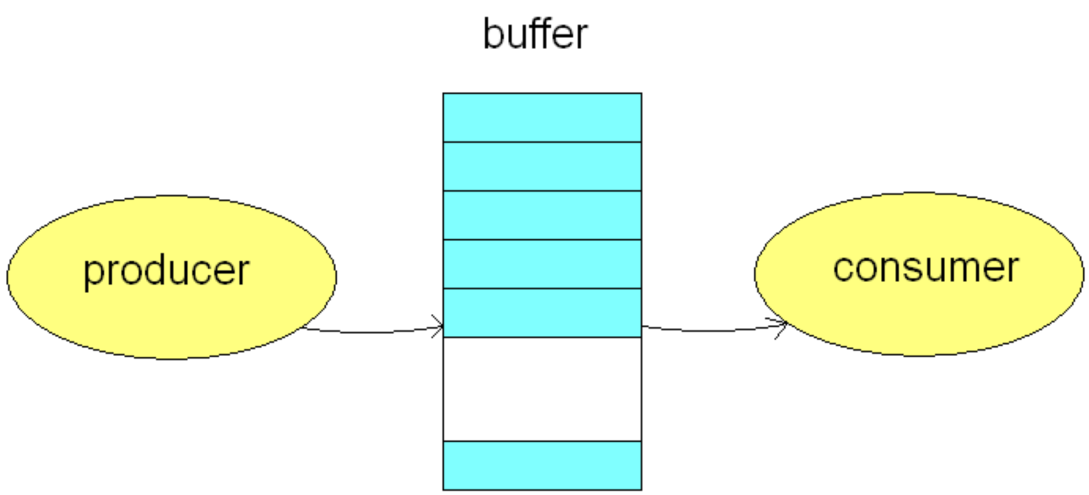
SHARED MEMORY

OS provides the abstraction of shared memory

Programmers need to handle the communication explicitly

Producer-consumer

- Requires synchronization



MESSAGE PASSING

Process 1

```
int main() {
    Message m;
    Send(P2, m);
}
```

Process 2

```
int main() {
    Message m;
    m = Receive(P1, m);
}
```

STYLES OF MESSAGE PASSING

Send/Receive calls

Blocking (synchronous)

- "Rendezvous"
    - Send blocks until receiver executes Receive
    - Receive blocks until there is a message
- Fixed-length queue
    - Sender blocks if queue is full
    - Rendezvous uses a fixed-length queue, length = 0

Non-blocking (asynchronous)

- Send buffers message, so Receiver will pick it up later
    - Needs an 'unbounded' message queue
- Receiver gets a message or an indication of no message (e.g., NULL)

DIRECT MESSAGE PASSING: USE IDENTIFIER OF PROCESS

Direct/symmetric

- Both sender and receiver name a process
- Send(P, message) // send to process P
- Receive(Q, message) // receive from process Q

Direct/asymmetric

- Send(P, message) // send to process P
- Receive(&id, message) // id gets set to sender

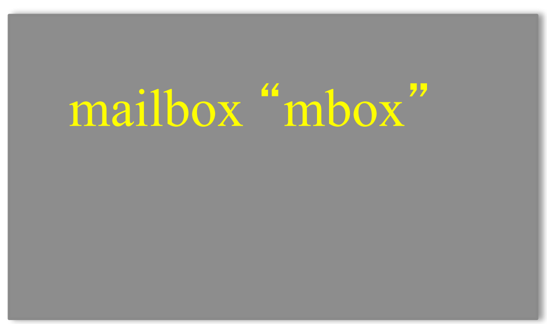
MAILBOXES: INDIRECT MESSAGE PASSING

Process 1

```
int main() {
    Message m;
    Send("mbox", m);
}
```

Process 2

```
int main() {
    Message m;
    m = Receive("mbox");
}
```



MAILBOXES: INDIRECT MESSAGE PASSING

Neither sender or receiver or receiver knows process ID of the other; use a mailbox instead

- E.g., using sockets in UNIX or Windows, and ports in Mach

Mailboxes have names or identifiers

Also have Send/Receive system calls

- Processes Send messages to mailboxes
- Receiver checks mailbox for messages using Receive

Mailboxes have owners

- E.g., owner may be creating process, or O/S
- Pass privileges to other processes
  - e.g., rights to ports in Mach can be sent to other processes
- Children may automatically share privileges

PIPES

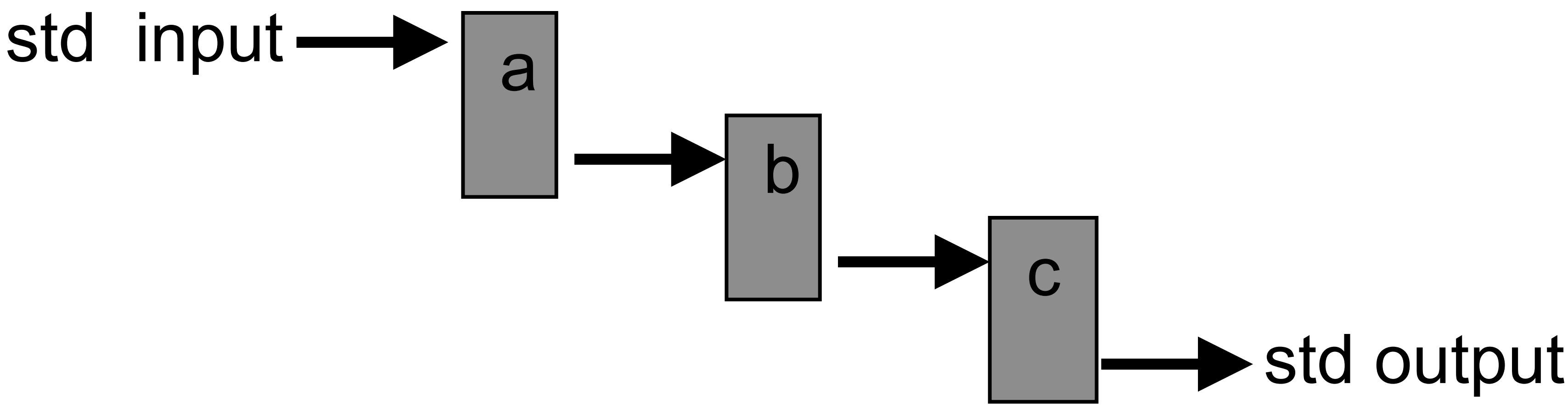
UNIX pipes are a shell construct:

- ls -alg | more

Standard output of program at left (producer) becomes the standard input of program at right (consumer).

HYDRAULIC ANALOGY

a | b | c



THE PIPE() SYSTEM CALL

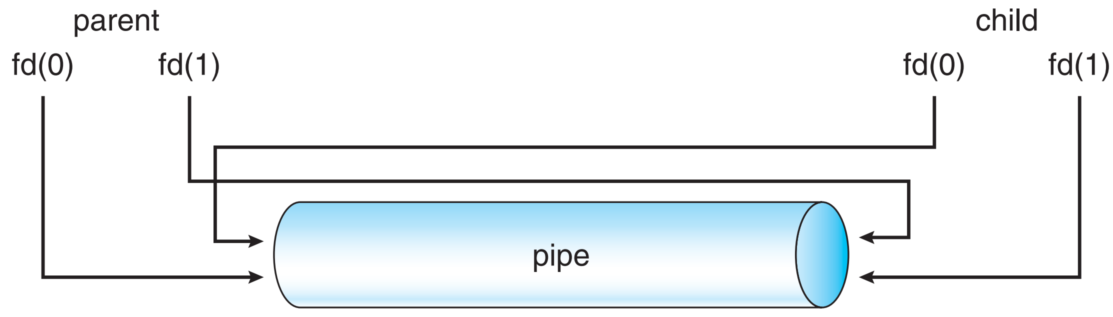
To create a pipe between two processes

```
int pd[2];
pipe(pd);
```

System call creates two new file descriptors:

- pd[0] that can be used to read from pd
- pd[1] that can be used to write to the pd

Also returns an error code



REMOTE PROCEDURE CALL

Process 1

```
int main() {
        // Invoke method
        // on Process 2
        Method1();
}
```

Process 2

Method1()

Method2()

…

REMOTE PROCEDURE CALLS (RPC)

Look like regular function calls to caller

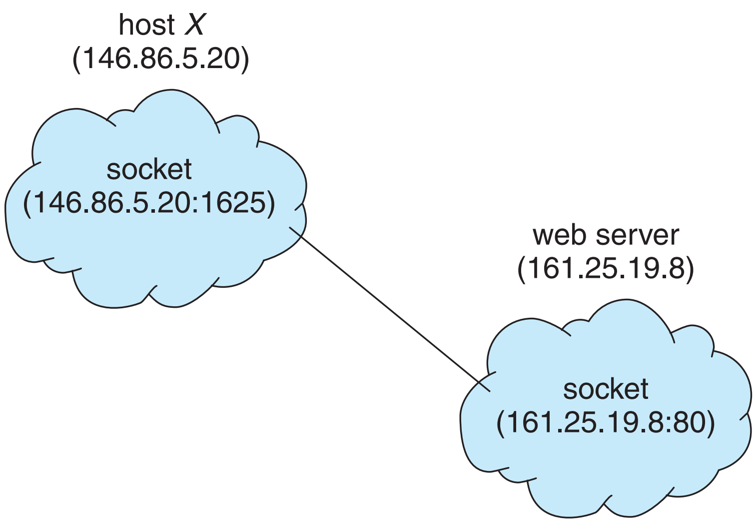
- But, RPC invocation from a ‘client’ causes a method to be invoked on a remote ‘server’
- Remote ‘server’ process provides implementation and processing of method
- Client side interface has to pack (“marshal”) arguments and requested operation type into a message & send to remote server

Can have blocking or non-blocking semantics

CLIENT-SERVER SYSTEMS

Sockets

- **Servers run on well-defined ports**
- **A socket uniquely identified by <src_ip, src_port, dst_ip, dst_port>**



SERVERS

Single threaded server:

- Processes one request at a time

```
for (;;) {
  receive(&client, request);
  process_request(...);
  send (client, reply);
} // for
```

A TRICKY QUESTION

What does a server do when it does not process client requests?

Possible answers:

- Nothing
- It busy waits for client requests
- It sleeps
  - WAITING state is sometimes called sleep state

THE PROBLEM

Most client requests involve disk accesses

- File servers
- Authentications servers

When this happens, the server remains in the WAITING state

- Cannot handle other customers' requests

A FIRST SOLUTION

```
int pid;
for (;;) {
   receive(&client, request);
   if ((pid = fork())== 0) {
       process_request(...);
       send (client, reply);
       _exit(0); // done
   } // if
} // for
```

THE GOOD AND THE BAD NEWS

The good news:

- Server can now handle several user requests in parallel

The bad news:

- fork() is a very expensive system call

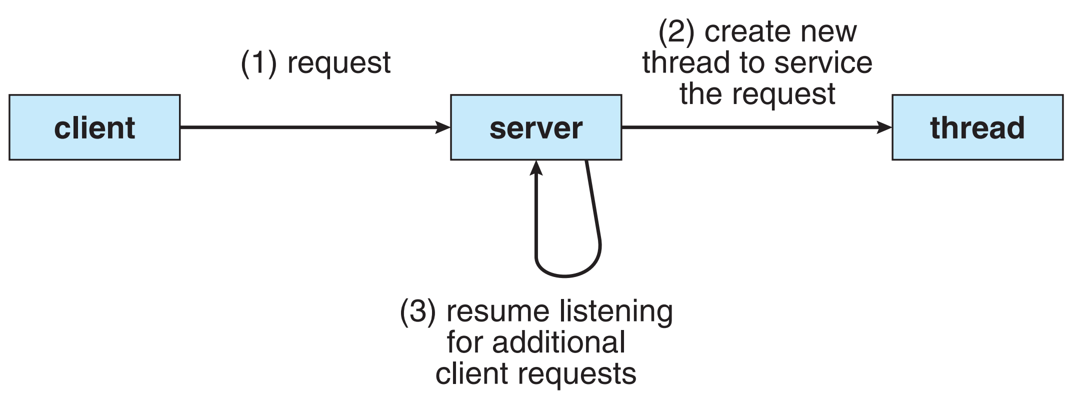
A BETTER SOLUTION

Provide a faster mechanism for creating cheaper processes:

- Threads

Threads share the address space of their parent

- No need to create a new address space
    - Most expensive step of fork() system call



A COMPARISON BETWEEN FORK & PTHREAD_CREATE()

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
Intel 2.6 GHz Xeon E5-2670 (16 cores/node)	8.1	0.1	2.9	0.9	0.2	0.3
Intel 2.8 GHz Xeon 5660 (12 cores/node)	4.4	0.4	4.3	0.7	0.2	0.5
AMD 2.3 GHz Opteron (16 cores/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8 cores/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8 cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8 cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8 cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6

~10 times faster

IS IT NOT DANGEROUS?

To some extent because

- No memory protection inside an address space
- Lightweight processes can now interfere with each other

But

- All lightweight process code is written by the same team
- Synchronization

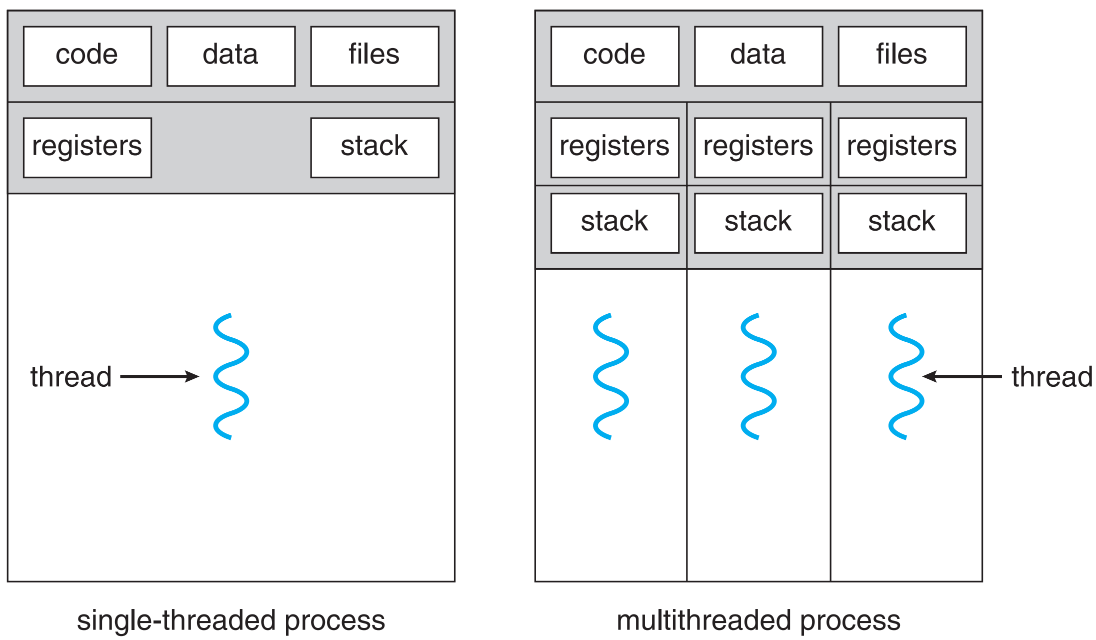
GENERAL CONCEPT

A thread

- Does not have its own address space
- Shares it with its parent and other peer threads in the same address space (task)

Each thread has a program counter, a set of registers and its own stack.

- Everything else is shared



EXAMPLES OF MULTITHREADED PROGRAMS

Embedded systems

- Elevators, Planes, Medical systems, Wristwatches
- Single Program, concurrent operations

Most modern OS kernels

- Internally concurrent because have to deal with concurrent requests by multiple users
- But no protection needed within kernel

Database Servers

- Access to shared data by many concurrent users
- Also background utility processing must be done

EXAMPLES OF MULTITHREADED PROGRAMS (CON'T)

Network Servers

- Concurrent requests from network
- Again, single program, multiple concurrent operations
- File server, Web server, and airline reservation systems

Parallel Programming (More than one physical CPU)

- Split program into multiple threads for parallelism
- This is called Multiprocessing

CLASSIFICATION

Real operating systems have either

- One or many processes
- One or many threads per process

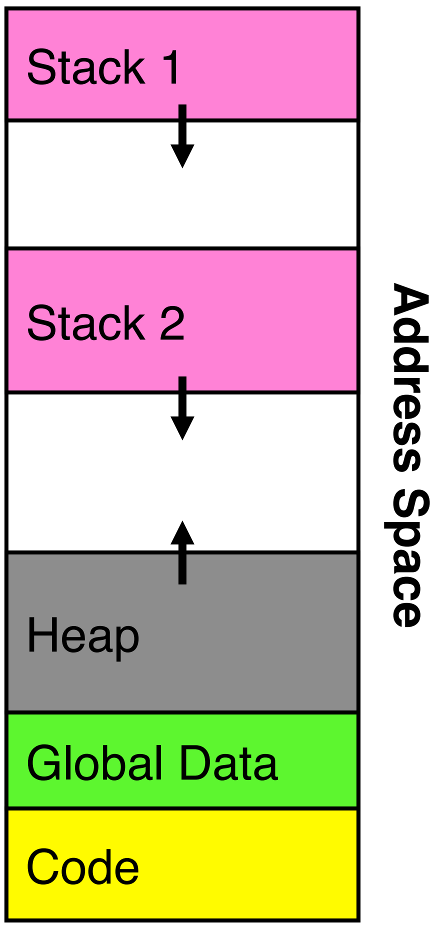
# threads /process: \ # of process:	One	Many
One	MS/DOS, early Macintosh	Traditional UNIX
Many	Embedded systems (Geoworks, VxWorks, etc)	Mach, OS/2, HP-UX, Win NT to 8, Solaris, OS X, Android, iOS

PID ▲	Process Name	User	% CPU	Threads	Real Mem	Kind	Virtual Mem
146	UserEventAgent	rzheng	0.0	3	6.6 MB	Intel (64 bit)	34.3 MB
154	AirPort Base Station Agent	rzheng	0.0	4	5.9 MB	Intel (64 bit)	31.1 MB
155	UFR II BackGrounder	rzheng	0.0	2	2.5 MB	Intel	30.0 MB
156	Canon MF Scan Agent	rzheng	0.0	3	10.5 MB	Intel (64 bit)	33.9 MB
157	Canon CMFP BackGrounder	rzheng	0.0	2	2.3 MB	Intel	30.0 MB
158	TeamViewer_Desktop	rzheng	0.0	3	7.7 MB	Intel	32.7 MB
159	TeamViewer	rzheng	0.0	4	15.1 MB	Intel	35.2 MB
160	LMIGUIAgent	rzheng	0.0	5	3.2 MB	Intel	31.5 MB
161	LogMeIn Menubar	rzheng	0.1	6	4.6 MB	Intel	32.3 MB
163	LogMeIn Hamachi Menubar	rzheng	0.0	3	4.5 MB	Intel	30.8 MB
177	GrowlHelperApp	rzheng	0.0	3	13.3 MB	Intel (64 bit)	30.9 MB
178	Skype	rzheng	0.0	20	55.4 MB	Intel	85.6 MB
180	Dropbox	rzheng	0.0	31	67.4 MB	Intel	174.6 MB
181	Google Drive	rzheng	0.4	22	71.9 MB	Intel	155.6 MB
187	LogMeIn Hamachi	rzheng	2.6	3	6.7 MB	Intel	31.9 MB
241	dbfseventsd	rzheng	0.0	1	168 KB	Intel	28 KB
268	VDCAssistant	rzheng	0.0	4	4.1 MB	Intel (64 bit)	31.3 MB
269	Image Capture Extension	rzheng	0.0	2	5.9 MB	Intel (64 bit)	29.3 MB
289	diskimages-helper	rzheng	0.0	3	10.5 MB	Intel (64 bit)	27.8 MB
724	mdworker	rzheng	0.0	3	44.8 MB	Intel (64 bit)	68.6 MB
735	Terminal	rzheng	0.0	5	14.6 MB	Intel (64 bit)	33.2 MB
738	bash	rzheng	0.0	1	1.1 MB	Intel (64 bit)	17.5 MB
808	Google Chrome	rzheng	0.1	34	99.8 MB	Intel	307.5 MB
812	Google Chrome Helper	rzheng	0.0	8	78.1 MB	Intel	111.4 MB
825	Google Chrome Helper	rzheng	0.0	5	22.9 MB	Intel	54.3 MB
836	Thunderbird	rzheng	0.1	33	262.7 MB	Intel (64 bit)	283.5 MB
850	Microsoft PowerPoint	rzheng	1.2	11	164.5 MB	Intel	149.9 MB
866	Microsoft AU Daemon	rzheng	0.0	2	2.0 MB	Intel	30.0 MB
948	Google Chrome Helper	rzheng	0.3	8	68.1 MB	Intel	89.6 MB

MEMORY FOOTPRINT OF TWO-THREAD EXAMPLE

If we stopped this program and examined it with a debugger, we would see

- Two sets of CPU registers
- Two sets of Stacks



PER THREAD STATE

Each Thread has a Thread Control Block (TCB)

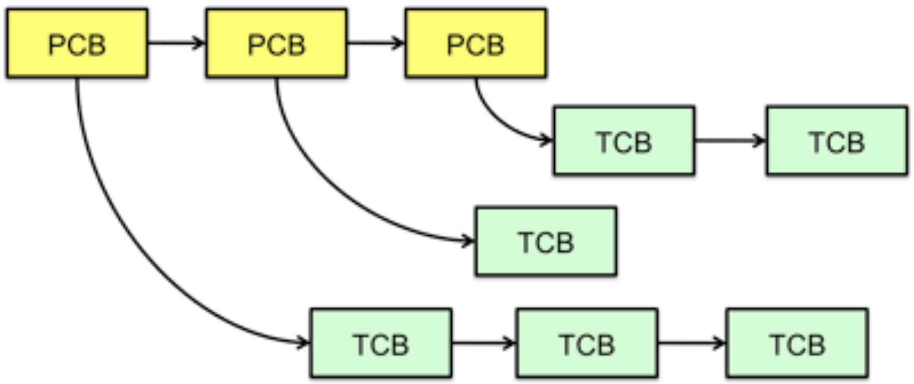
- Execution State: CPU registers, program counter (PC), pointer to stack (SP)
- Scheduling info: state, priority, CPU time
- Various Pointers (for implementing scheduling queues)
- Pointer to enclosing process (PCB)
- Etc (add stuff as you find a need)

OS Keeps track of TCBs in protected memory

- In Array, or Linked List, or …

MULTITHREADED PROCESSES

PCB points to multiple TCBs:



Switching threads within a process is a simple thread switch

Switching threads across processes requires changes to memory and I/O address tables.

THREAD LIFECYCLE

As a thread executes, it changes state:

- new: The thread is being created
- ready: The thread is waiting to run
- running: Instructions are being executed
- waiting: Thread waiting for some event to occur
- terminated: The thread has finished execution

“Active” threads are represented by their TCBs

- TCBs organized into queues based on their state

IMPLEMENTATION

Thread can either be

- Kernel supported:
  - Mach, Linux, Windows NT and after
- User-level:
  - *Pthread library*, Java thread

KERNEL-SUPPORTED THREADS

Managed by the kernel through system calls

One process table entry per thread

Kernel can allocate several processors to a single multithreaded process

Supported by Mach, Linux, Windows NT and more recent systems

Switching between two threads in the same processes involves a system call

- Results in two context switches

USER-LEVEL THREADS

User-level threads are managed by procedures within the task address space

- The thread library

One process table entry per process/address space

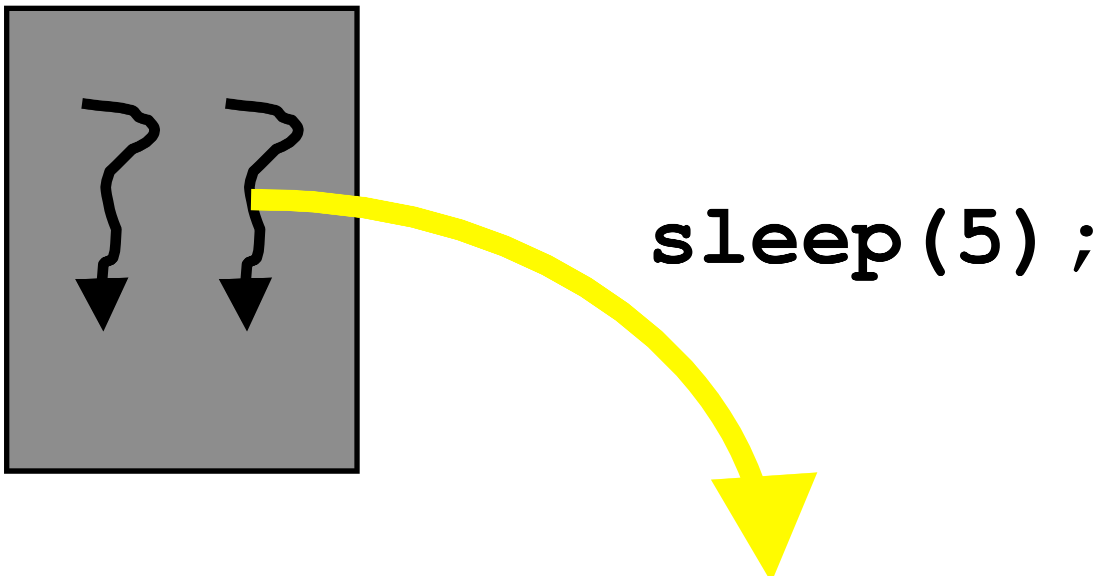
- Kernel is not even aware that process is multithreaded

No performance penalty: Switching between two threads of the same task is done cheaply within the task

Programming issue:

- Each time a thread does a blocking system call, kernel will move the whole process to the waiting state
  - It does not know better
- Programmer must use non-blocking system calls
  - Can be nasty

USER-LEVEL THREADS

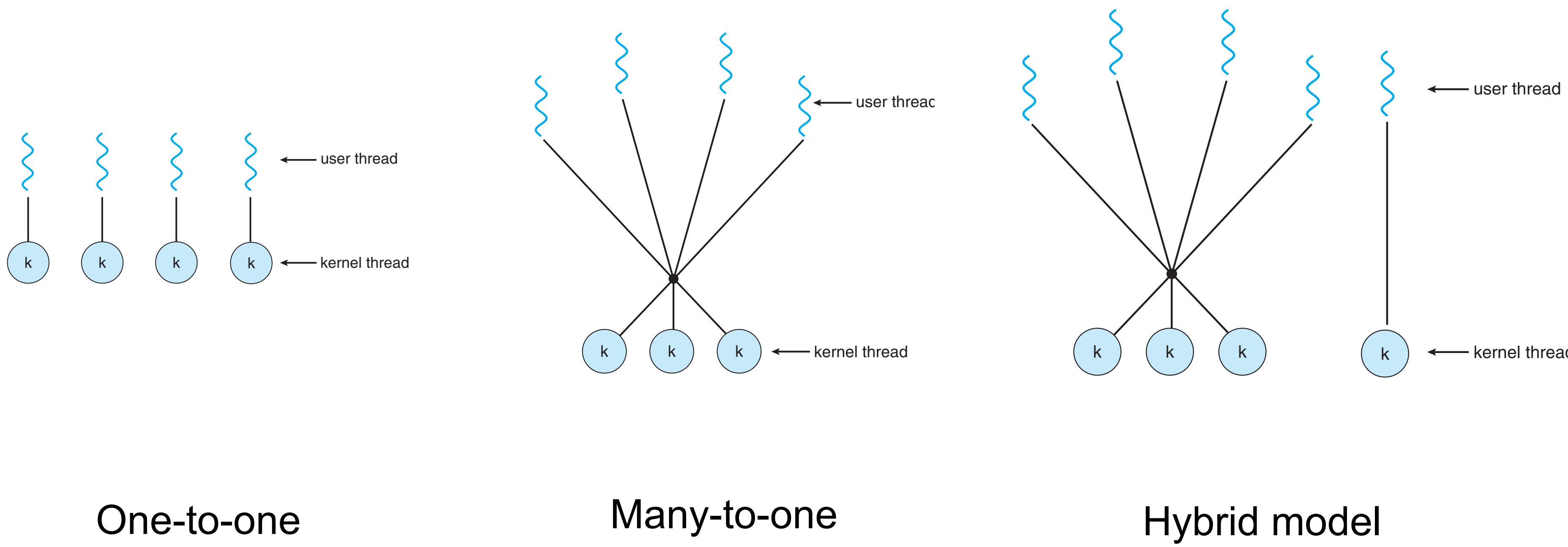


Kernel

Process wants to sleep for 5 seconds:

Let us move it to the waiting state

MAPPING BETWEEN KERNEL AND USER LEVEL THREADS



POSIX THREADS

POSIX threads, or pthreads, are standardized programming interface

Ported to various Unix and Windows systems (Pthreads-win32).

On Linux, pthread library implements the 1:1 model

Function names start with "pthread_"

Calls tend to have a complex syntax : over 100 methods and data types

AN EXAMPLE

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

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

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }
   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

SUMMARY

Goals:

- Multiprogramming: Run multiple applications concurrently
- Protection: Don't want a bad application to crash system!

Solution:

Process: unit of execution and allocation

- Virtual Machine abstraction: give process illusion it owns machine (i.e., CPU, Memory, and IO device multiplexing)

Challenge:

- Process creation & switching expensive
- Need concurrency within same app (e.g., web server)

Solution:

Thread: Decouple allocation and execution

- Run multiple threads within same process