CSE 380
Computer Operating Systems

Instructor: Insup Lee

University of Pennsylvania
Fall 2003

Lecture Note 2: Processes and Threads
Lecture Note 2.1: Processes and System Calls

Process

- Consider a simple disk operating system (like MS-DOS)
- User types command like “run foo” at Keyboard (I/O device driver for keyboard, screen)
- Command is parsed by command shell
- Executable program file (load module) “foo” is located on disk (file system, I/O device driver for disk)
- Contents are loaded into memory and control transferred ==> process comes alive! (disk device driver, relocating loader, memory management)
- During execution, process may call OS to perform I/O: console, disk, printer, etc. (system call interface, I/O device drivers)
- When process terminates, memory is reclaimed (memory management)

A process is a program in execution with associated data and execution context

Multiprogramming/Timesharing Systems

- Goal: to provide interleaved execution of several processes to give an illusion of many simultaneously executing processes.
- Computer can be a single-processor or multi-processor machine.
- The OS must keep track of the state for each active process and make sure that the correct information is properly installed when a process is given control of the CPU.
- Many resource allocation issues to consider:
  - How to give each process a chance to run?
  - How is main memory allocated to processes?
  - How are I/O devices scheduled among processes?

Keeping track of processes

- For each process, OS maintains a data structure, called the process control block (PCB). The PCB provides a way of accessing all information relevant to a process:
  - This data is either contained directly in the PCB, or else the PCB contains pointers to other system tables.

- Processes (PCBs) are manipulated by two main components of the OS in order to achieve the effects of multiprogramming:
  - **Scheduler**: determines the order in which processes will gain access to the CPU. Efficiency and fair-play are issues here.
  - **Dispatcher**: actually allocates CPU to the process selected by the scheduler.

Process Context

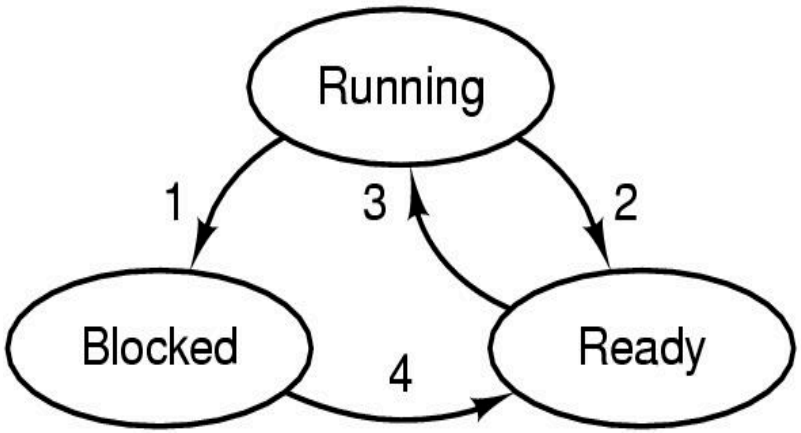
- The context (or image) of a process can be described by
  - contents of main memory
  - contents of CPU registers
  - other info (open files, I/O in progress, etc.)

- Main memory -- three logically distinct regions of memory:
  - text region: contains executable code (typically read-only)
  - data region: storage area for dynamically allocated data structure, e.g., lists, trees (typically heap data structure)
  - stack region: run-time stack of activation records

- Registers: general registers, PC, SP, PSW, segmentation registers

- Other information:
  - open files table, status of ongoing I/O
  - process status (running, ready, blocked), user id, ...

Process States



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process states
  - Running: executing
  - Blocked: waiting for I/O
  - Ready: waiting to be scheduled

When are processes created?

1. System initialization (many **daemons** for processing email, printing, web pages etc.)

2. Execution of a process creation system call by a running process (fork or CreateProcess)

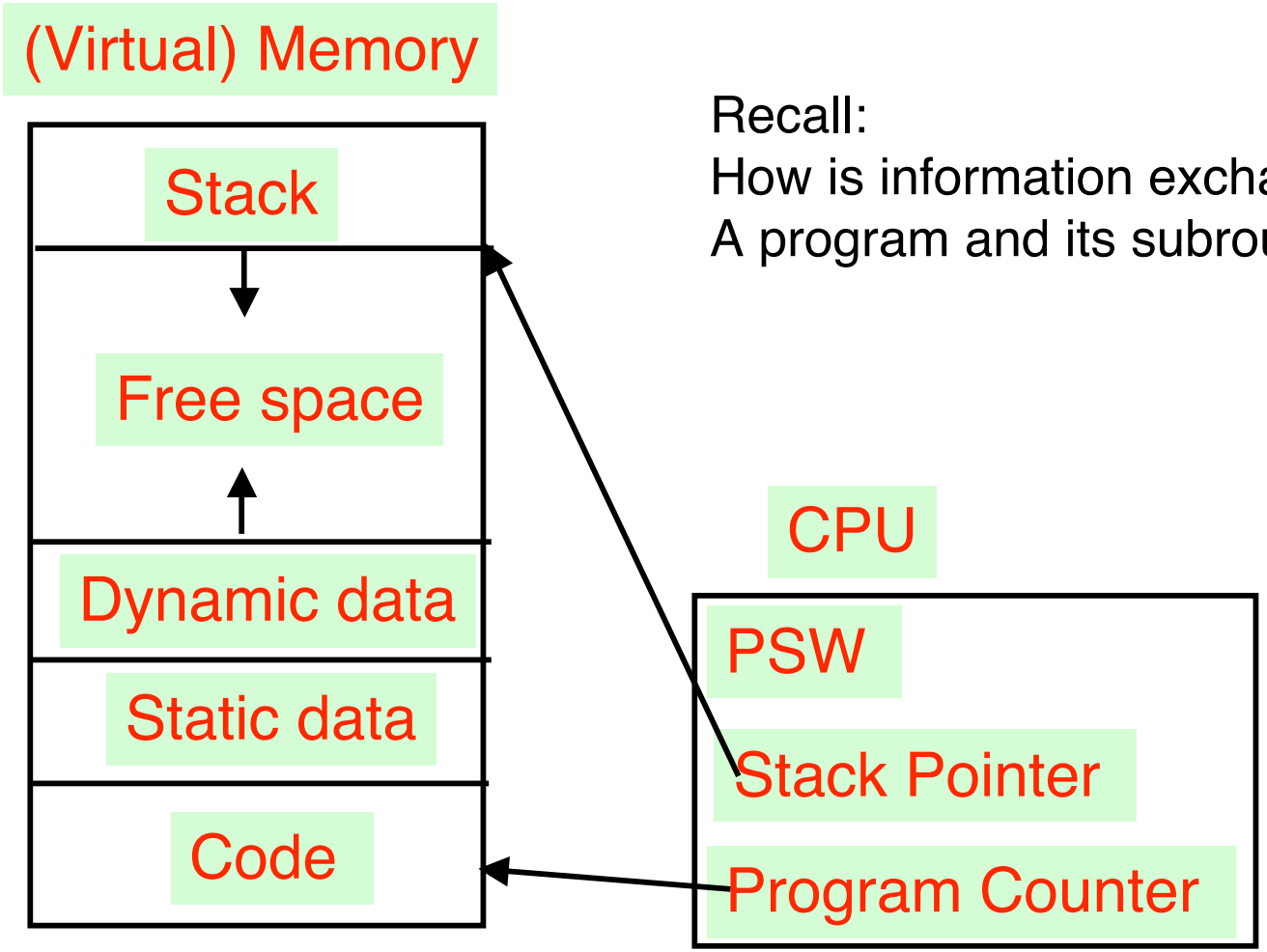
3. User request to create a new process (executing new applications)

List of all active processes: ps (on Unix), Ctl-Alt-Del (on Windows)

When are processes terminated?

1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary), due to bugs
4. Killed by another process (involuntary)

Process Snapshot



Recall:
How is information exchanged between
A program and its subroutines ?

When does OS get invoked?

- Operating system gets control in two ways
  - A user process calls OS by a system call (e.g. executes TRAP)
  - An interrupt aborts the current user process
- System stack can be used to exchange information between OS and user processes
- Recall: Mode bit in PSW is set only when OS is running

Interrupts

- An interruption of the normal processing of processor.
- Interrupts cause the CPU to suspend its current computation and take up some new task. Control may be returned to the original task at some time later.
- Reasons for interrupts and/or traps:
  - control of asynchronous I/O devices
  - CPU scheduling
  - exceptional conditions (e.g., div. by zero, page fault, illegal instruction) arising during execution
- Interrupts are essentially what drives an OS. We can view the OS as an event-driven system, where an interrupt is an event.
- Bounding interrupt handling latency is important for real-time systems.

Interrupt Handling

- The servicing of an interrupt is known as interrupt handling.
- An integer is associated with each type of interrupt. When an interrupt occurs, the corresponding integer is supplied to the OS usually by the hardware (in a register).
- The OS maintains a table, known as the interrupt vector, that associates each interrupt's id with the starting address of its service routine.
- Example interrupt vector:

Interrupt No.	Interrupt Handler
0	clock
1	disk
2	tty
3	dev
4	soft
5	other

Interrupts

On Intel Pentium, hardware interrupts are numbered 0 to 255

- 0: divide error
- 1: debug exception
- 2: null interrupt
- 6: invalid opcode
- 12: stack fault
- 14: page fault
- 16: floating-point error
- 19-31: Intel reserved (not available?)
- 32-255: maskable interrupts (device controllers, software traps)

Issue: can CPU be interrupted while an OS interrupt handler is executing?
Maskable interrupts can be disabled, plus there is a priority among interrupts

Typical interrupt handling sequence

- Interrupt initiated by I/O device signaling CPU, by exceptional condition arising, through execution of special instruction, etc.
- CPU suspends execution of current instruction stream and saves the state of the interrupted process (on system stack).
- State typically refers to contents of registers: PC, PSW, SP, general-purpose registers.
- The cause of the interrupt is determined and the interrupt vector is consulted in order to transfer control to the appropriate interrupt handler.
- Interrupt handler performs whatever processing is necessary to deal with the interrupt.
- Previous CPU state is restored (popped) from system stack, and CPU returns control to interrupted task.

Example: Servicing a Timer Interrupt

- Timer device is used in CPU scheduling to make sure control is returned to system every so often (e.g., 1/60 sec.)
- Typically, timer has a single register that can be loaded with an integer indicating a particular time delay (# of ticks).
- Once loaded, timer counts down and when 0 is reached, an interrupt is generated.
- Interrupt handler might do the following:
  - update time-of-day information
  - signal any processes that are "asleep" and awaiting this alarm
  - call the CPU scheduler
- Control returns to user mode, possibly to a different process than the one executing when the interrupt occurred.

Example: Servicing a Disk Interrupt

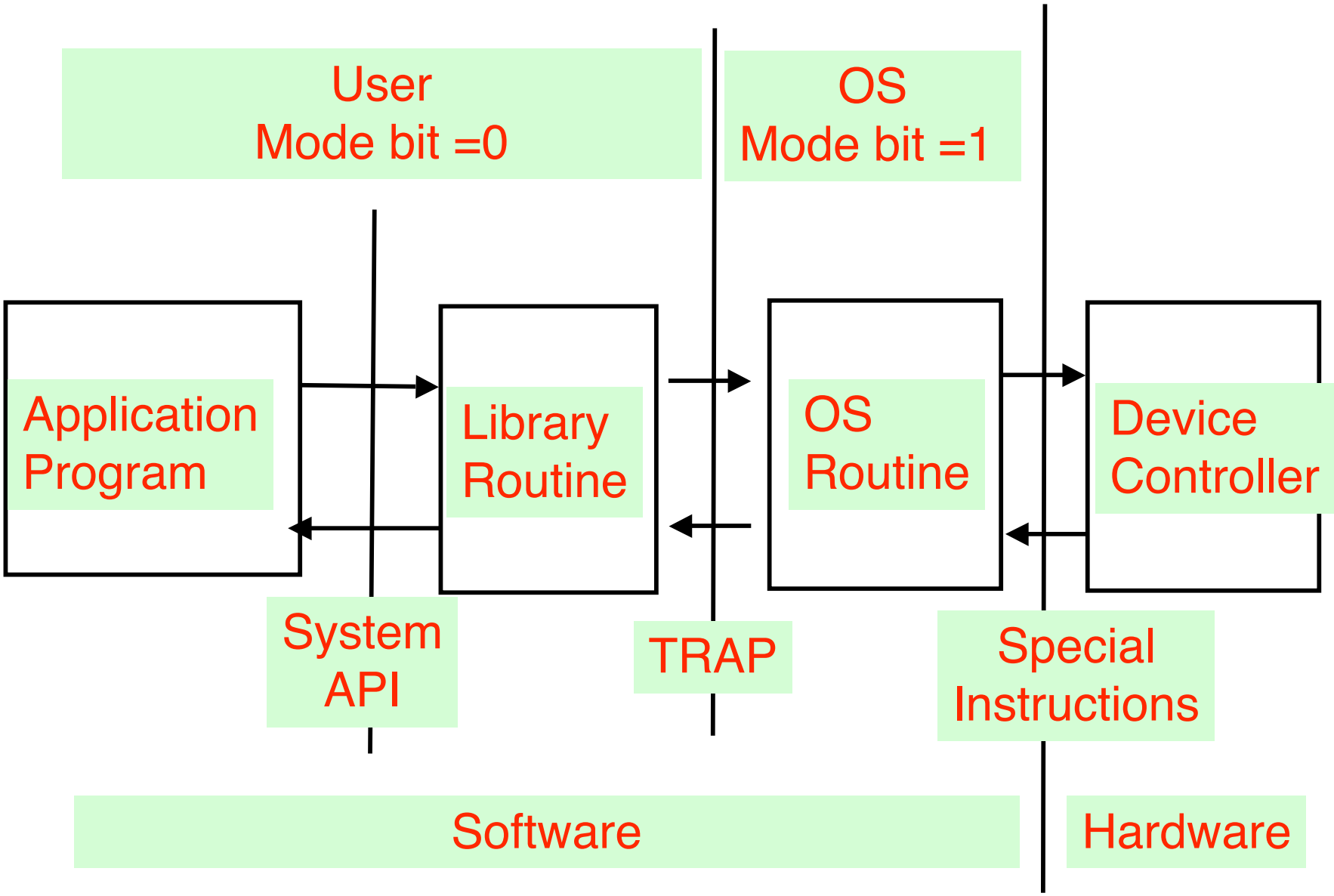
- When disk controller completes previous transfer, it generates an interrupt.
- Interrupt handler changes the state of a process that was waiting for just-completed transfer from wait-state to ready-state.
- It also examines queue of I/O requests to obtain next request.
- I/O is initiated on next request.
- CPU scheduler called.
- Control returned to user mode.

System Calls

- Provide "direct access" to operating system services (e.g., file system, I/O routines, memory allocate & free routines) by user programs.
- System calls are special, and in fact, are treated like interrupts.
- Programs that make system calls were traditionally called "system programs" and were traditionally implemented in assembly language
  - Win32 API in Windows
- Each system call had a particular number. Instruction set has a special instruction for making system calls:

SVC	**(IBM 360/370)**
trap	**(PDP 11)**
tw	**(PowerPC) - trap word**
tcc	**(Sparc)**
break	**(MIPS)**

System Calls



Steps in System Call

- User program pushes parameters to read on stack
- User program executes CALL instruction to invoke library routine **read** in assembly language
- Read routine sets up the register for system call number
- Read routine executes TRAP instruction to invoke OS
- Hardware sets the mode-bit to 1, saves the state of the executing read routine, and transfers control to a fixed location in kernel
- Kernel code, using a table look-up based on system call number, transfers control to correct system call handler

Steps in System Call (cont)

- OS routine copies parameters from user stack, sets up device driver registers, and executes the system call using privileged instructions
- OS routine can finish the job and return, or decide to suspend the current user process to avoid waiting
- Upon return from OS, hardware resets the mode-bit
- Control transfers to the read library routine and all registers are restored
- Library routine terminates, transferring control back to original user program
- User program increments stack pointer to clear the parameters

CSE 380
Computer Operating Systems

Instructor: Insup Lee

University of Pennsylvania
Fall 2003

Lecture Note 2.2: Unix Processes, Threads

Creating processes in UNIX

- To see how processes can be used in application and how they are implemented, we study how processes are created and manipulated in UNIX.
- Important source of information on UNIX is “man.”
- UNIX supports multiprogramming, so there will be many processes in existence at any given time.
  - Processes are created in UNIX with the fork() system call.
  - When a process P creates a process Q, Q is called the child of P and P is called the parent of Q.

Process Hierarchies

- Parent creates a child process, child processes can create its own process
- Forms a hierarchy
  - UNIX calls this a *process group*
- Signals can be sent all processes of a group
- Windows has no concept of process hierarchy
  - all processes are created equal

Initialization

At the root of the family tree of processes in a UNIX system is the special process init:

- created as part of the bootstrapping procedure
- process-id = 1
- among other things, init spawns a child to listen to each terminal, so that a user may log on.
- do "man init" to learn more about it

UNIX Process Control

UNIX provides a number of system calls for process control including:

- fork - used to create a new process
- exec - to change the program a process is executing
- exit - used by a process to terminate itself normally
- abort - used by a process to terminate itself abnormally
- kill - used by one process to kill or signal another
- wait - to wait for termination of a child process
- sleep - suspend execution for a specified time interval
- getpid - get process id
- getppid - get parent process id

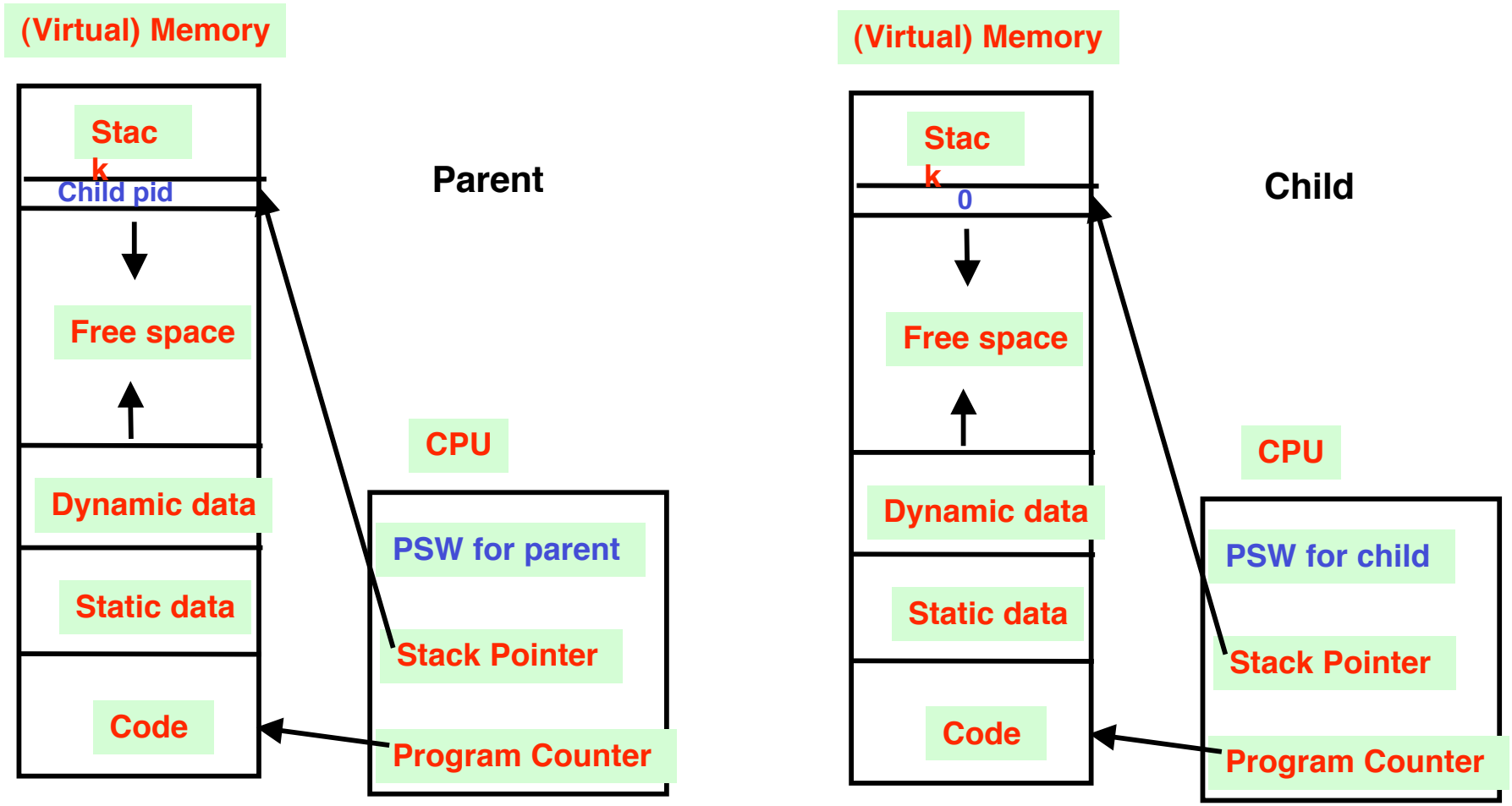
The Fork System Call

- The **fork()** system call creates a "clone" of the calling process.

- Identical in every respect except
  - the parent process is returned a non-zero value (namely, the process id of the child)
  - the child process is returned zero.

- The process id returned to the parent can be used by parent in a **wait** or **kill** system call.

Snapshots after fork()



Example using fork

```
 1. #include <unistd.h>
 2. main(){
 3.   pid_t pid;
 4.   printf("Just one process so far\n");
 5.   pid = fork();
 6.   if (pid == 0) /* code for child */
 7.      printf("I'm the child\n");
 8.   else if (pid > 0) /* code for parent */
 9.      printf("The parent, child pid =%d\n",
10.             pid);
11.   else /* error handling */
12.      printf("Fork returned error code\n");
13. }
```

Sample Question

```
main(){
   int x=0;
   fork();
   x++;
   printf("The value of x is %d\n", x);
}
```

What will be the output?

Spawning Applications

fork() is typically used in conjunction with exec (or variants)

```
pid_t pid;
if ( ( pid = fork() ) == 0 ) {
    /* child code: replace executable image */
    execv( "/usr/games/tetris", "-easy" )
} else {
    /* parent code: wait for child to terminate */
    wait( &status )
}
```

exec System Call

A family of routines, **execl**, **execv**, ..., all eventually make a call to **execve**.

execve(program_name, arg1, arg2, ..., environment)

- text and data segments of current process replaced with those of **program_name**
- stack reinitialized with parameters
- open file table of current process remains intact
- the last argument can pass environment settings
- as in example, **program_name** is actually path name of executable file containing program

Note: unlike subroutine call, there is no return after this call. That is, the program calling exec is gone forever!

Parent-Child Synchronization

- **exit(status)** - executed by a child process when it wants to terminate. Makes **status** (an integer) available to parent.

- **wait(&status)** - suspends execution of process until *some* child process terminates
  - **status** indicates reason for termination
  - return value is process-id of terminated child

- **waitpid (pid, &status, options)**
  - pid can specify a specific child
  - Options can be to wait or to check and proceed

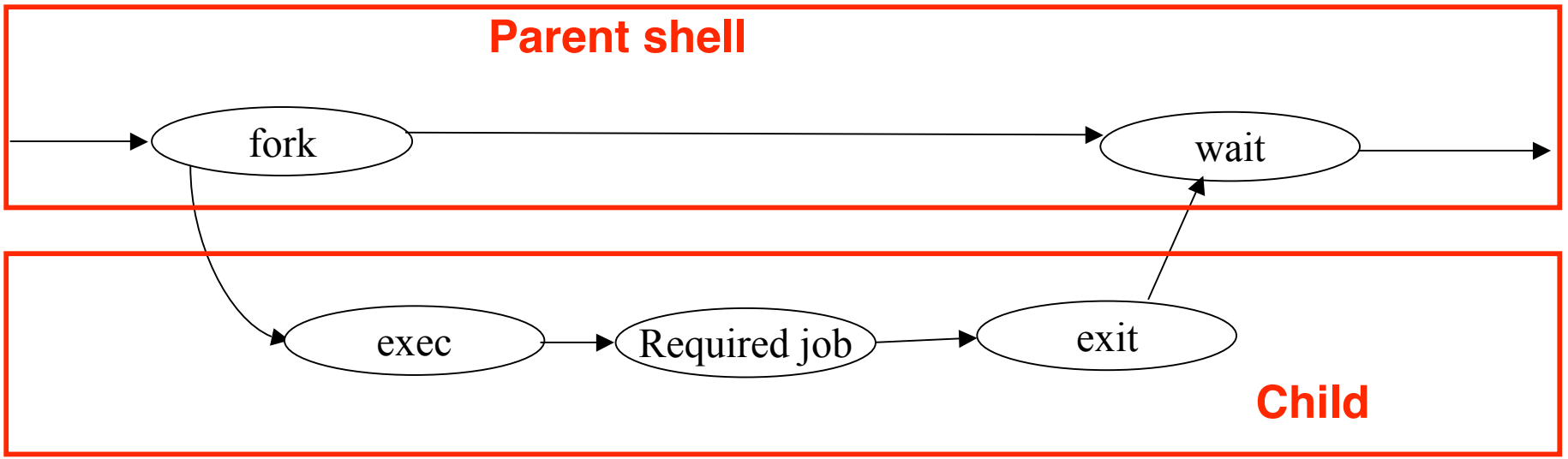
Process Termination

- Besides being able to terminate itself with **exit**, a process can be killed by another process using **kill**:
  - **kill(pid, sig)** - sends signal **sig** to process with process-id **pid**. One signal is **SIGKILL** (terminate the target process immediately).

- When a process terminates, all the resources it owns are reclaimed by the system:
  - "process control block" reclaimed
  - its memory is deallocated
  - all open files closed and Open File Table reclaimed.

- Note: a process can kill another process only if:
  - it belongs to the same user
  - super user

How shell executes a command



- when you type a command, the shell forks a clone of itself
- the child process makes an exec call, which causes it to stop executing the shell and start executing your command
- the parent process, still running the shell, waits for the child to terminate

CSE 380
Computer Operating Systems

Instructor: Insup Lee

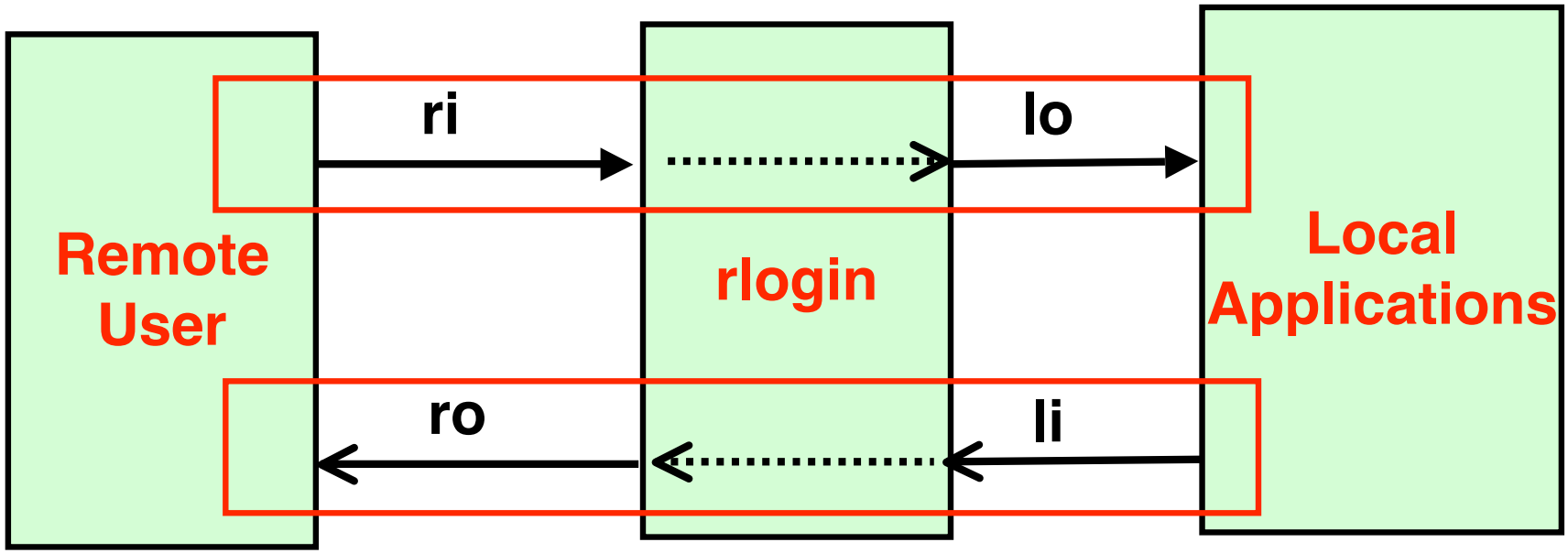
University of Pennsylvania, Fall 2003

Lecture Note 2.3: Threads

Introduction to Threads

- Multitasking OS can do more than one thing concurrently by running more than a single process
- A process can do several things concurrently by running more than a single thread
- Each thread is a different stream of control that can execute its instructions independently.
- Ex: A program (e.g. Browser) may consist of the following threads:
  - GUI thread
  - I/O thread
  - Computation thread

When are threads useful?



Challenges in single-threaded soln

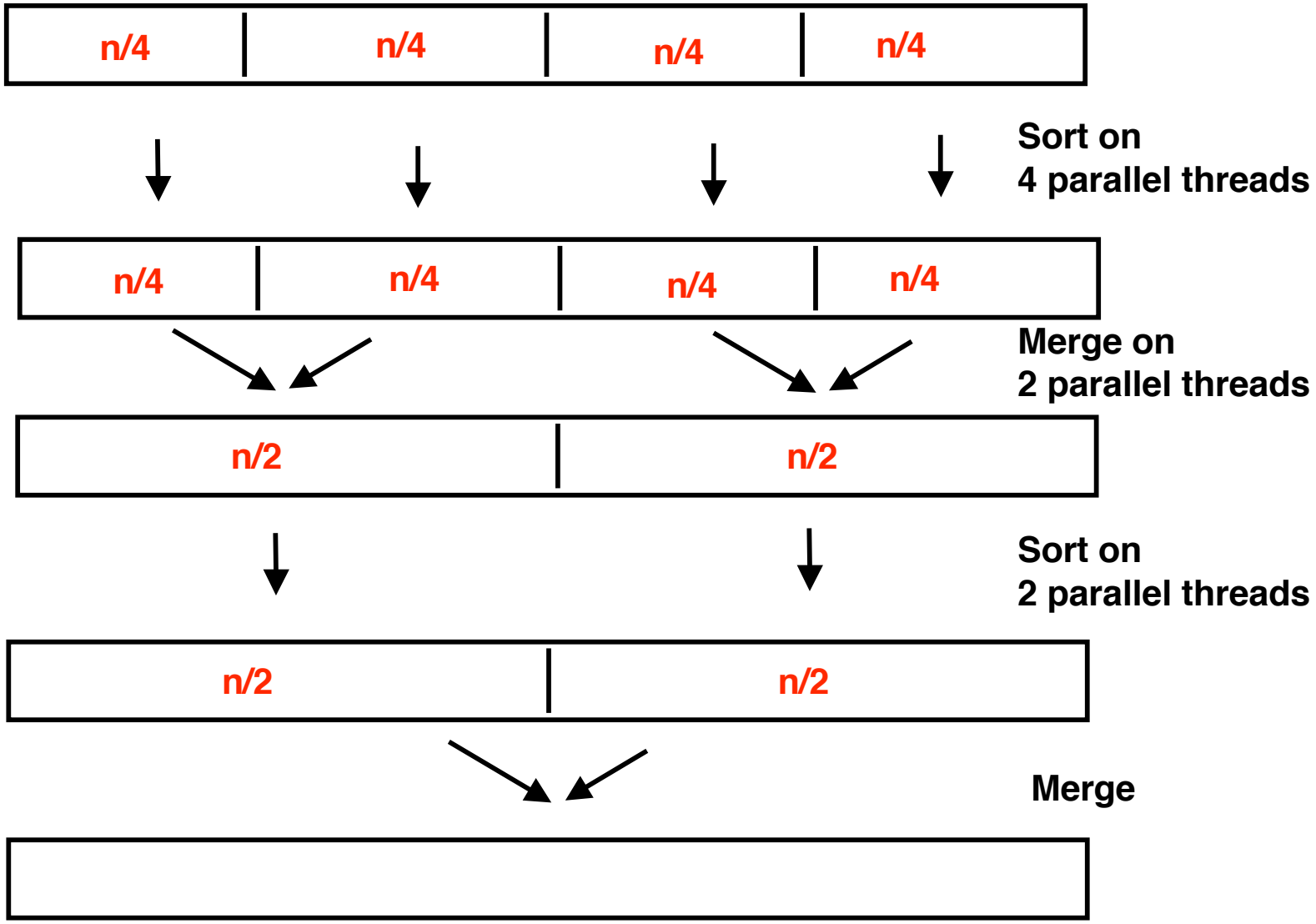
- There are basically 4 activities to be scheduled
  - read(li), read(ri), write(lo), write(ro)
- **read** and **write** are blocking calls
- So before issuing any of these calls, the program needs to check readyness of devices, and interleave these four operations
  - System calls such as FD_SET and select
- Bottomline: single-threaded code can be quite tricky and complex

Solution with Threads

```
incoming(int ri, lo){
 int d=0;
 char b[MAX];
 int s;
 while (!d) {
  s=read(ri,b,MAX);
  if (s<=0) d=1;
  if (write(lo,b,s)<=0)
  d=1;
  }
}
```

```
outgoing(int li, ro){
 int d=0;
 char b[MAX];
 int s;
 while (!d) {
  s=read(li,b,MAX);
  if (s<=0) d=1;
  if (write(ro,b,s)<=0)
  d=1;
  }
}
```

Parallel Algorithms: Eg. mergesort



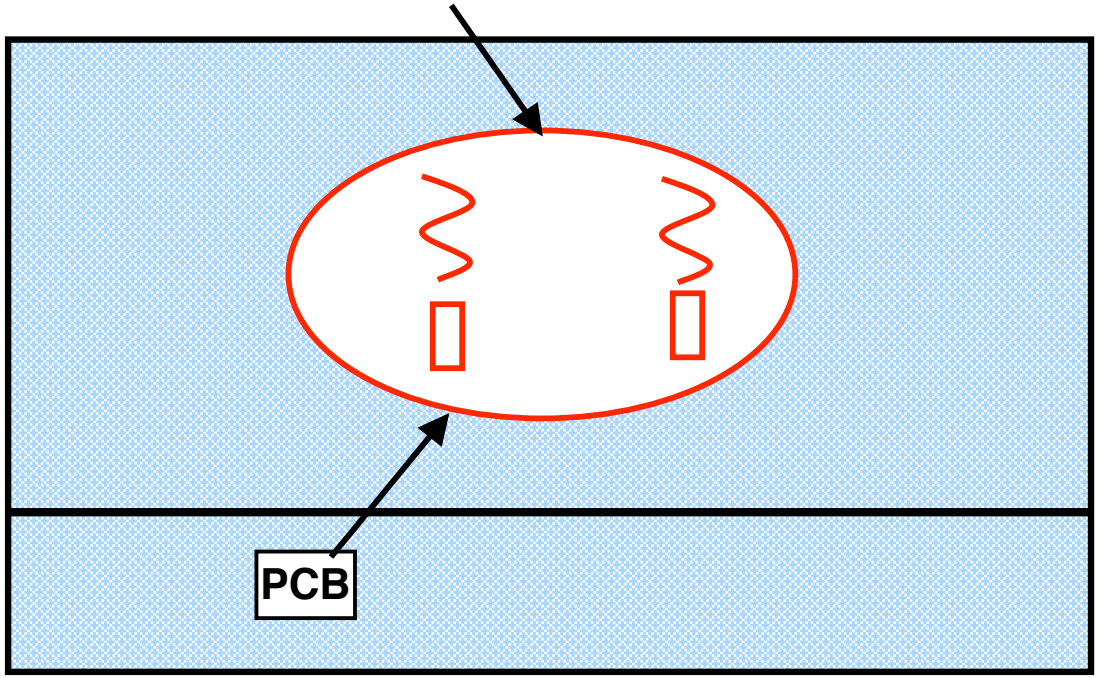
Is there a speed-up ?

Benefits of Threads: Summary

1. Superior programming model of parallel sequential activities with a shared store
2. Easier to create and destroy threads than processes.
3. Better CPU utilization (e.g. dispatcher thread continues to process requests while worker threads wait for I/O to finish)
4. Guidelines for allocation in multi-processor systems

Threads Model

Two threads residing in the same user process



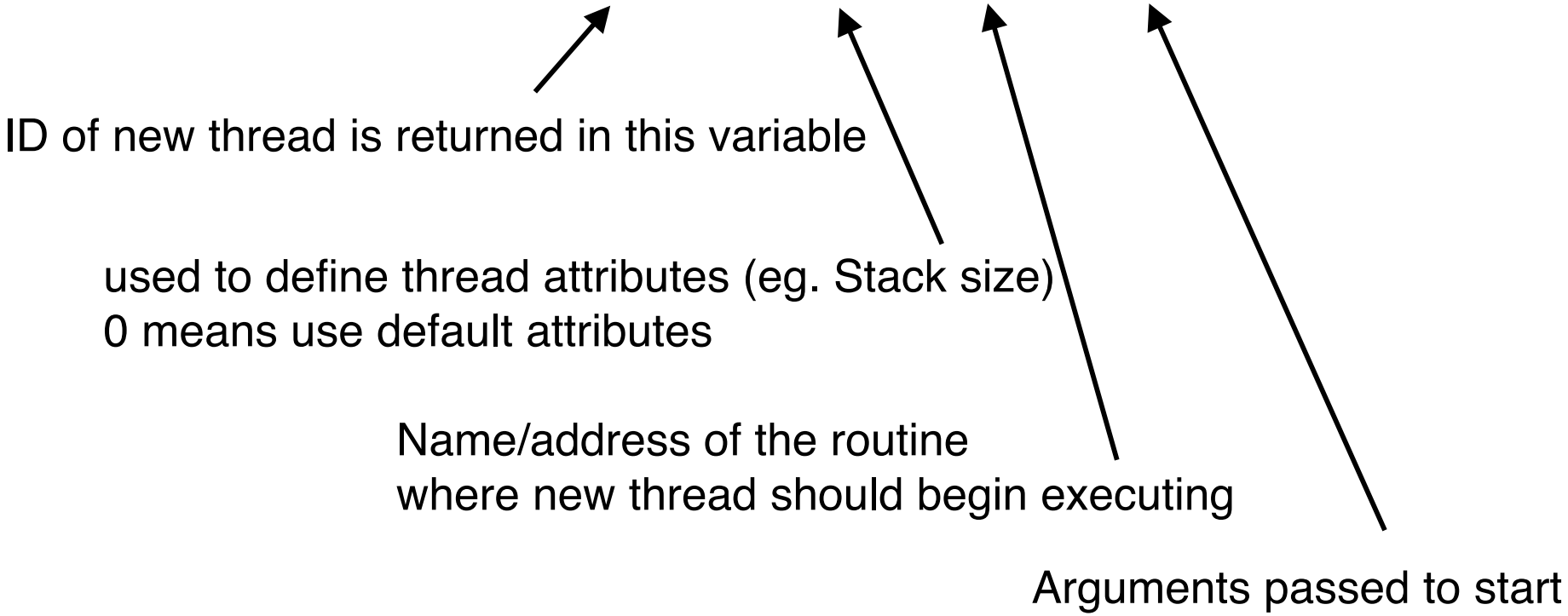
Each thread has its own stack, why?

Processes and Threads

- A UNIX Process is
  - a running program with
  - a bundle of resources (file descriptor table, address space)
- A thread has its own
  - stack
  - program counter (PC)
  - All the other resources are shared by **all** threads of that process. These include:
    - open files
    - virtual address space
    - child processes

Thread Creation

- POSIX standard API for multi-threaded programming
- A thread can be created by **pthread_create** call
- pthread_create (&thread, 0, start, args)



Sample Code

```
typedef struct { int i, o } pair;
rlogind ( int ri, ro, li, lo) {
  pthread_t in_th, out_th;
  pair in={ri,lo}, out={li,ro};
  pthread_create(&in_th,0, incoming, &in);
  pthread_create(&out_th,0, outgoing, &out);
}
Note: 2 arguments are packed in a structure
```

Problem: If main thread terminates, memory for in and out structures may disappear, and spawned threads may access incorrect memory locations

If the process containing the main thread terminates, then all threads are automatically terminated, leaving their jobs unfinished.

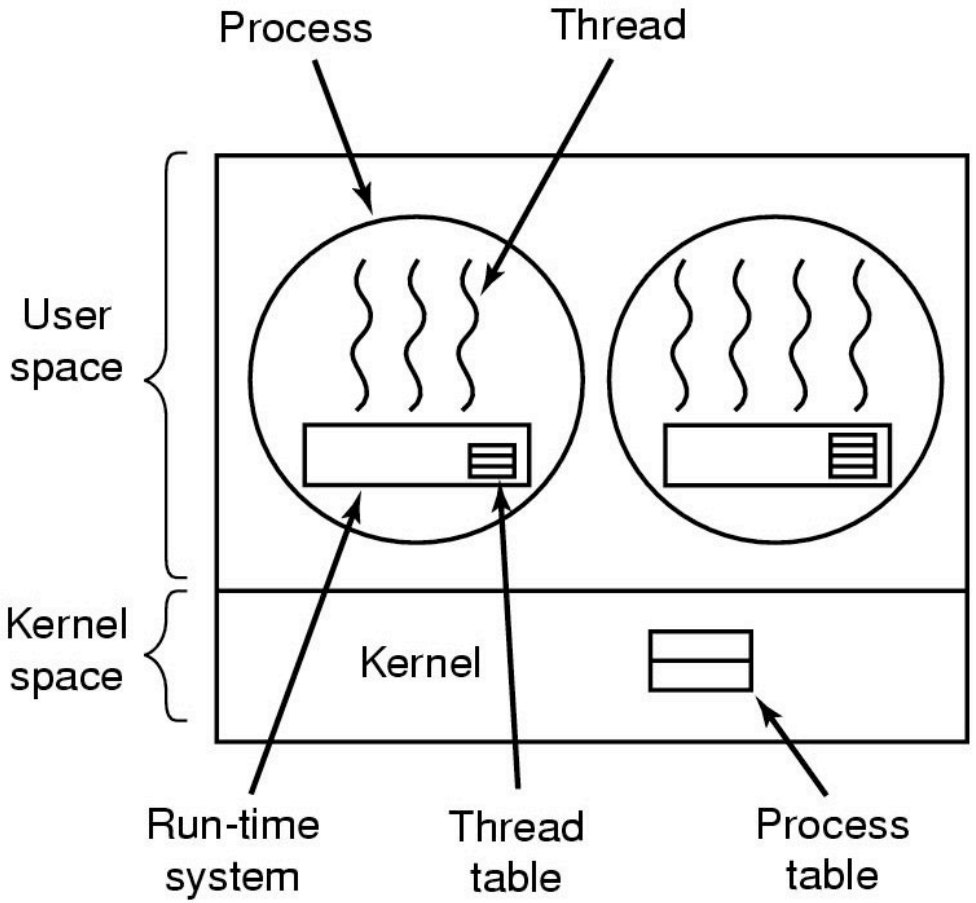
Ensuring main thread waits…

```
typedef struct { int i, o } pair;
rlogind ( int ri, ro, li, lo) {
  pthread_t in_th, out_th;
  pair in={ri,lo}, out={li,ro};
  pthread_create(&in_th,0, incoming, &in);
  pthread_create(&out_th,0, outgoing, &out);
  pthread_join(in_th,0);
  pthread_join(out_th,0);
}
```

Thread Termination

- A thread can terminate
  1. by executing **pthread_exit**, or
  2. By returning from the initial routine (the one specified at the time of creation)
- Termination of a thread unblocks any other thread that's waiting using **pthread_join**
- Termination of a process terminates all its threads

Implementing Threads in User Space

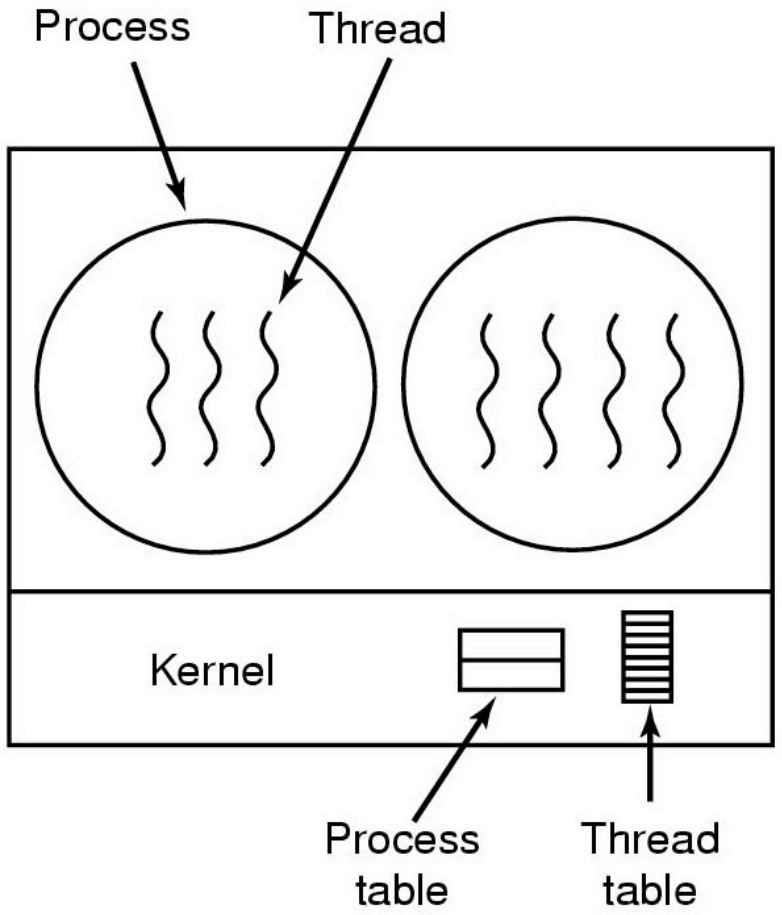


A user-level threads package

User-Level Threads

- The run-time support system for threads is entirely in user space.
- The threads run on top of a run-time system, which is a collection of procedures that manage threads.
- As far as the OS is concerned, it is a single (threaded) process.
- Threads can be implemented on an OS that does not support threads.
- Each process can have its own customized scheduling algorithm.

Implementing Threads in the Kernel



A threads package managed by the kernel

Kernel-supported Threads

- No run-time system is needed.
- For each process, the kernel has a table with one entry per thread, for thread's registers, state, priority, and other information.
- All calls that might block a thread are implemented as system calls, at considerably greater cost than a call to a run-time system procedure.
- When a thread blocks, the kernel can run either another thread from the same process, or a thread from a different process.

User-level vs. Kernel-supported Threads

- If OS does not support threads, a library package in user space can do threads management
- What are the trade-offs for user-level vs kernel-level threads?
- Assume:
  - Process A has one thread and Process B has 100 threads.
  - Scheduler allocates the time slices equally
- User-level Thread:
  - A thread in process A runs 100 times as fast as a thread in process B.
  - One blocking system call blocks all threads in process B.
- Kernel-supported Threads:
  - Process B receives 100 times the CPU time than process A.
  - Switching among the thread is more time-consuming because the kernel must do the switch.
  - Process B could have 100 system calls in operation concurrently.