

# Processes and Threads

Manish Shrivastava

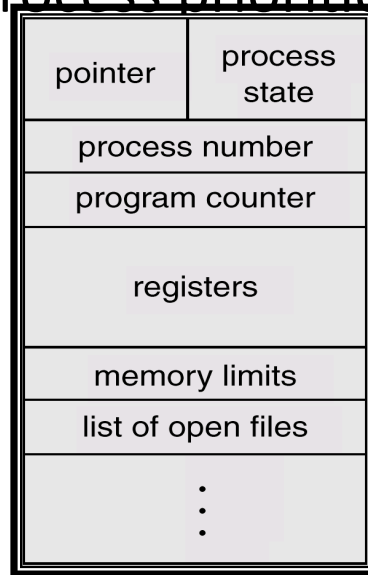
# Process Control Block (PCB)

pointer	process state
process number	
program counter	
registers	
memory limits	
list of open files	
⋮	

# Process Control Block (PCB)

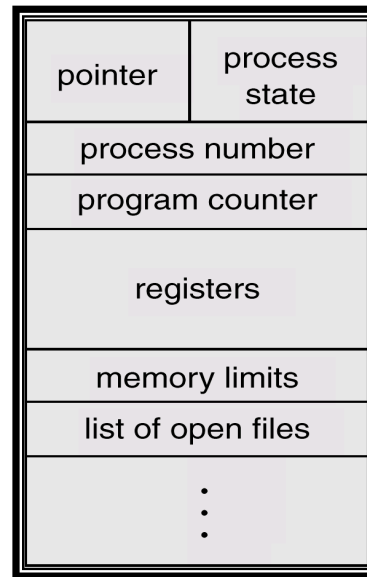
Information associated with each process.

- **Process state:** new, ready, running,...
- **Program Counter (PC):** address of the next instruction to execute
- **CPU registers:** data registers, stacks, condition-code information, etc.
- **CPU scheduling information:** process priorities, pointers to scheduling queues, etc.



# Process Control Block (PCB)

- **Memory-management information:** locations including value of base and limit registers, page tables and other virtual memory information.
- **Accounting information:** the amount of CPU and real time used, time limits, account numbers, job or process numbers etc.
- **I/O status information:** List of I/O devices allocated to this process, a list of open files, and so on

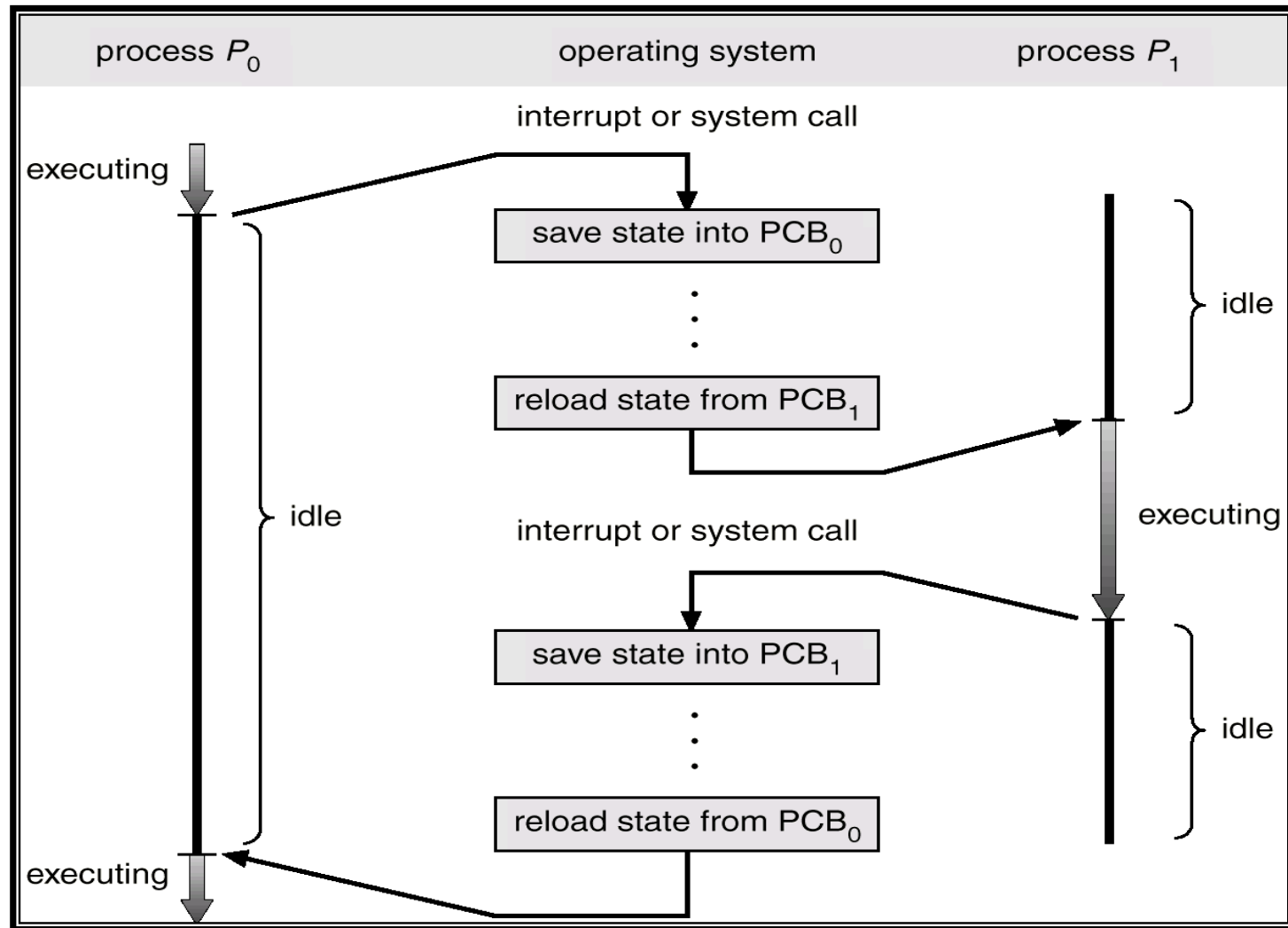


# Process Control Block (PCB)

- The process control block in the Linux operating system is represented by the C structure `task_struct`, which is found in the `<linux/sched.h>` include file in the kernel source-code directory.

```
long state; /* state of the process */
struct sched_entity se; /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

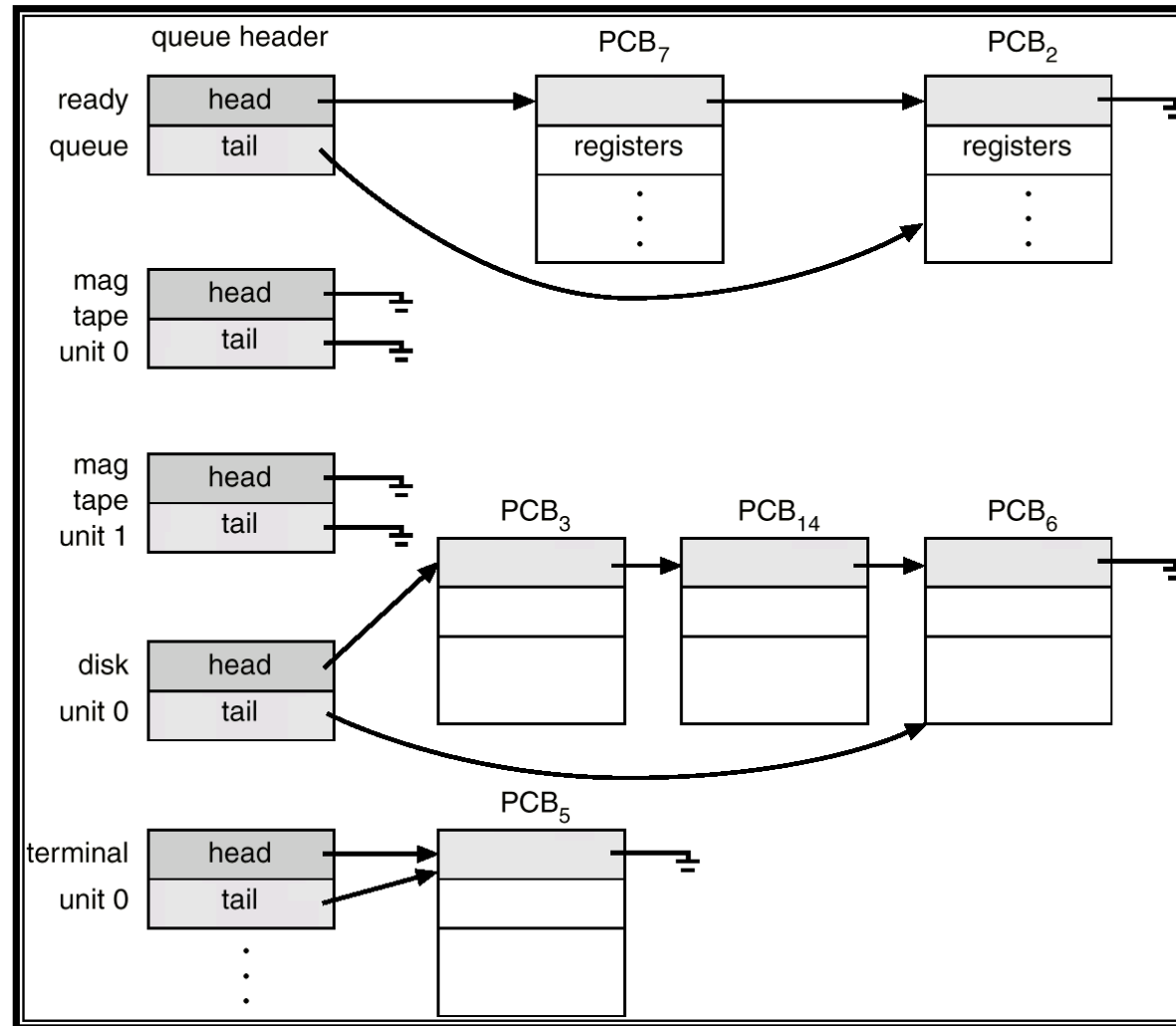
# CPU Switch From Process to Process



# Process Scheduling Queues

- Scheduling is to decide which process to execute and when
- The objective of multi-program
  - To have some process running at all times.
- **Timesharing:** Switch the CPU frequently that users can interact can interact with the program while it is running.
- If there are many processes, scheduling is used to decide which process to execute and when.
- Scheduling queues:
  - Job queue – set of all processes in the system.
  - Ready queue – set of all processes residing in main memory, ready and waiting to execute.
  - Device queues – set of processes waiting for an I/O device.
    - Each device has its own queue.
- Process migrates between the various queues during its life time.

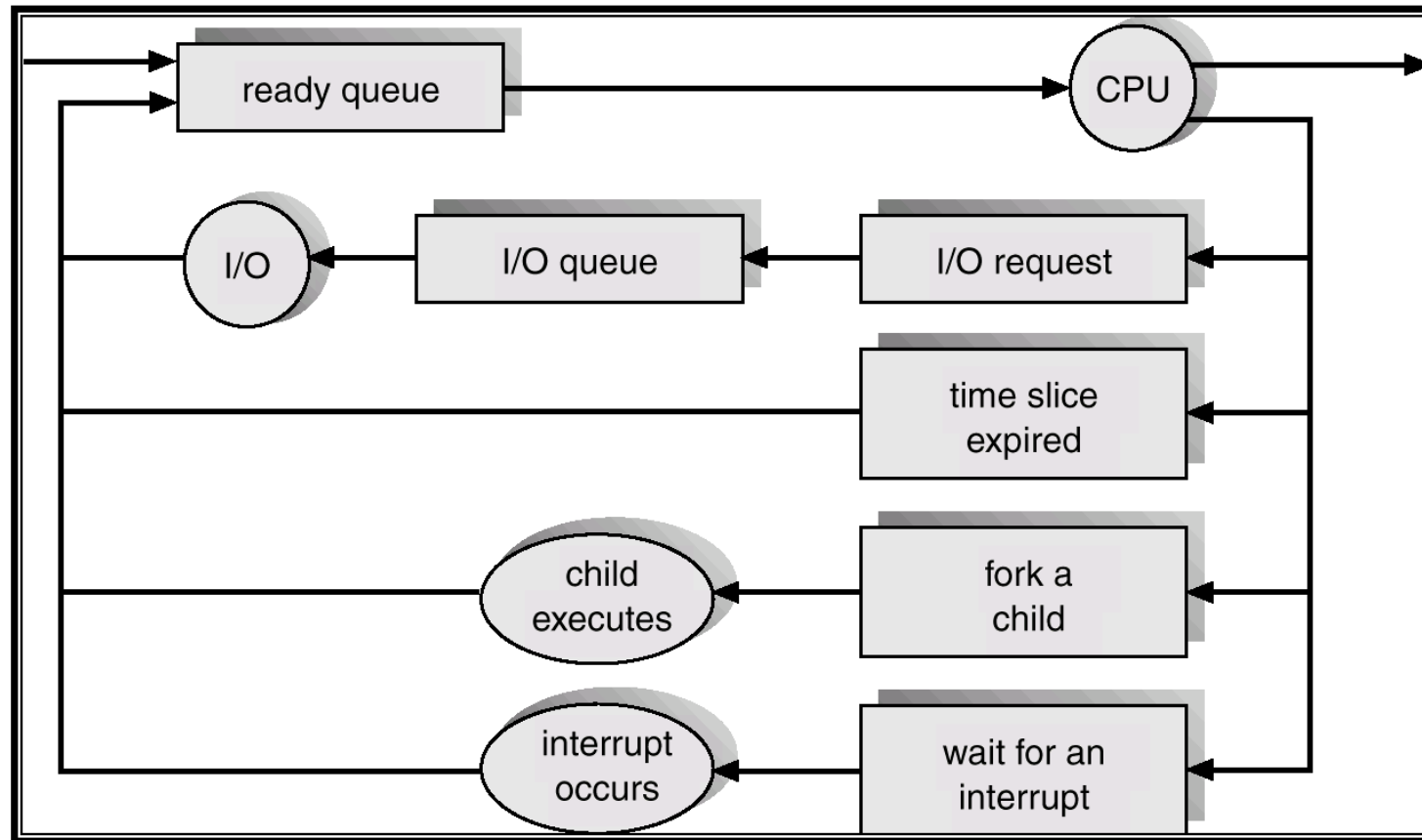
# Ready Queue And Various I/O Device Queues





# Representation of Process Scheduling

- Queuing Diagram




# Representation of Process Scheduling

- A new process is initially put in the ready queue
- Once a process is allocated CPU, the following events may occur
  - A process could issue an I/O request
  - A process could create a new process
  - The process could be removed forcibly from CPU, as a result of an interrupt.
- When process terminates, it is removed from all queues. PCB and its other resources are de-allocated.

# Process Schedulers

- A process migrates between the various scheduling queues throughout its lifetime.
- The OS must select a process from the different process queues in some fashion. The selection process is carried out by a scheduler.
- In a batch system the processes are spooled to mass-storage device.
- **Long-term (LT) scheduler (or job scheduler)** – selects which processes should be brought into the ready queue.
- The LT scheduler controls degree of multiprogramming (i.e., the number of processes active in the system)
- DoM is stable: average rate of process creation == average departure rate of processes.

# Process Schedulers

- The LT scheduler should make a careful selection. 
- Most processes are either I/O bound or CPU bound.
  - **I/O bound process** spends more time doing I/O than it spends doing computation.
  - **CPU bound process** spends most of the time doing computation.
- The LT scheduler should select a good mix of I/O-bound and CPU-bound processes.
- Example:
  - If all the processes are I/O bound, the ready queue will be empty
  - If all the processes are CPU bound, the I/O queue will be empty, the devices will go unutilized and the system will be imbalanced.

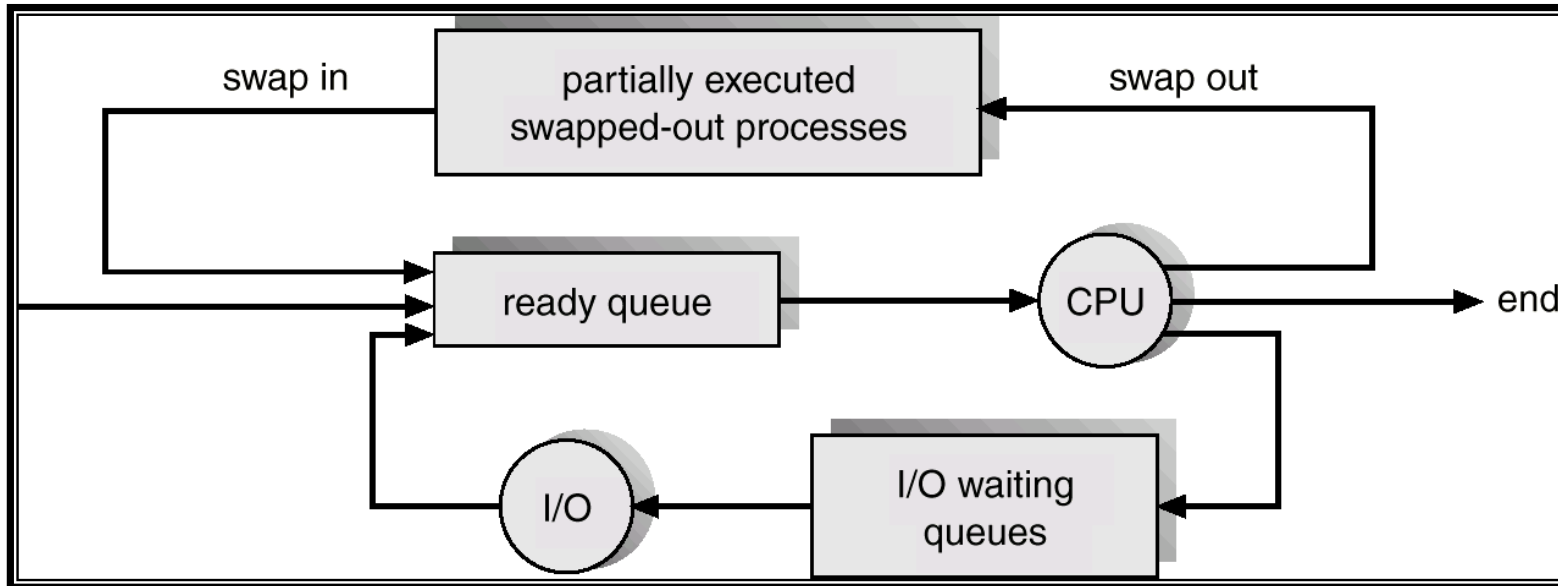
# Process Schedulers

- **Short-term (ST) scheduler**
  - selects which process should be executed next and allocates CPU.
  - It is executed at least once every 100 ms.
  - If 10 ms is used for selection, then 9 % of CPU is used (or wasted)
- The long-term scheduler executes less frequently.

# Process Schedulers

- Some OSs introduced a **Medium-Term** scheduler using swapping.
  - Advantageous to remove the processes from the memory and reduce the multiprogramming.
- **Swapping:** removal of process from main memory to disk to improve the performance. At some later time, the process can be reintroduced into main memory and its execution can be continued when it left off.
- Swapping improves the process mix (I/O and CPU), when main memory is unavailable.

# Addition of Medium Term Scheduling



# Context Switch

- Context switch is a task of switching the CPU to another process by saving the state of old process and loading the saved state for the new process
- Context of old process is saved in PCB and loads the saved context of old process.
- Context-switch time is **overhead**; the system does no useful work while switching.
- New structures threads were incorporated.
- Time dependent on hardware support.
  - 1 to 1000 microseconds



# Operations on Processes

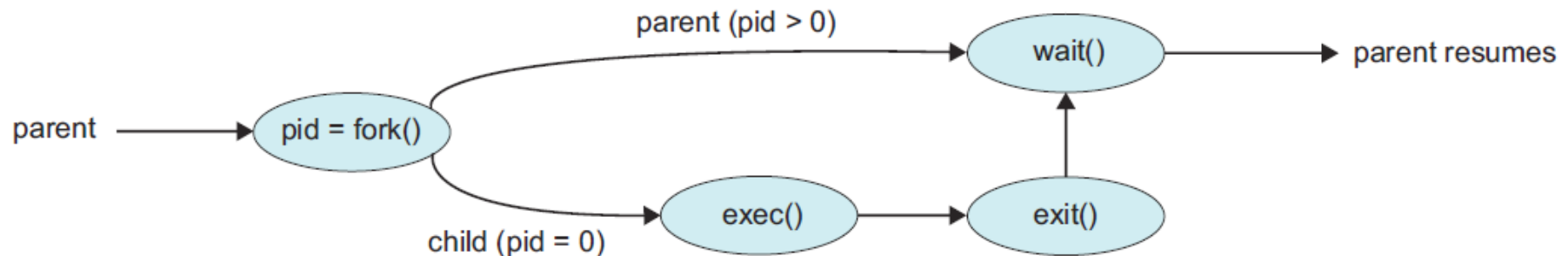
- Process Creation
- Process Termination

# Process Creation


- A system call is used to create process.
  - Assigns unique id
  - Space
  - PCB is initialized.
- The creating process is called parent process.
- Parent process creates children processes, which, in turn create other processes, forming a tree of processes.
  - In UNIX **pagedaemon**, **swapper**, and **init** children are root process. Users are children of **init** process.
- A process needs certain resources to accomplish its task.
  - CPU time, memory, files, I/O devices.

# Process Creation

- When a process creates a new process,
  - Resource sharing possibilities.
    - Parent and children share all resources.
    - Children share subset of parent's resources.
    - Parent and child share no resources.
  - Execution possibilities
    - Parent and children execute concurrently.
    - Parent waits until children terminate.
  - Address space
    - Child duplicate of parent.
    - Child has a program loaded into it.



# Process Creation

- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program.
  - The new process is a copy of the original process.
  - The exec system call is used after a fork by one of the two processes to replace the process memory space with a new program.
- WINDOWS NT supports both models:
  - /Parent address space can be duplicated or
  -  parent can specify the name of a program for the OS to load into the address space of the new process.

# UNIX: fork() system call

- fork() is used to create processes. It takes no arguments and returns a process ID.
- fork() creates a new process which becomes the child process of the caller.
- After a new process is created, both processes will execute the next instruction following the fork() system call.
- The checking the return value, we have to distinguish the parent from the child.
- fork()
  - If returns a negative value, the creation is unsuccessful.
  - Returns 0 to the newly created child process.
  - Returns positive value to the parent.
- Process ID is of type pid\_t defined in sys/types.h
- getpid() can be used to retrieve the process ID.
- The new process consists of a copy of address space of the original process.

# UNIX: fork() system call

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#define MAX_COUNT 200
#define BUF_SIZE 100
void main(void)
{
    pid_t pid;
    int i;
    char buf[BUF_SIZE];
    fork();
    pid=getpid();
    for(i=1; i<=MAX_COUNT;i++)
    {
        sprintf(buf,"This line is from pid %d, value=%d\n",pid,i);
        write(1,buf,strlen(buf));
    }
}
```

# UNIX: fork() system call

- If the fork() is executed successfully, Unix will
  - Make two identical copies of address spaces; one for the parent and one for the child.
  - Both processes start their execution at the next statement after the fork().

## Parent

```
main()
{
    fork();
    pid=...;
}
```

## Child

```
main()
{
    fork();
    pid=...
}
```

# UNIX: fork() system call

```
#include <stdio.h>
#include <sys/types.h>
#define MAX_COUNT 200
void ChildProcess(void);
void ParentProcess(void);
#define BUF_SIZE 100
void main(void)
{
    pid_t pid;
    pid=fork();
    if (pid==0)
        ChildProcess();
    else
        ParentProcess();
}
```

```
Void ChildProcess(void)
{
    int i;
    for(i=1;i<=MAX_COUNT;i++)
    {
        printf(buf,"This line is from child, value=%d\n",i);
        Printf(" *** Child Process is done ***\n");
    }
}
```

```
Void ParentProcess(void)
{
    int i;
    for(i=1;i<=MAX_COUNT;i++)
    {
        printf(buf,"This line is from parent, value=%d\n",i);
        printf(" *** Parent Process is done ***\n");
    }
}
```



# UNIX: fork() system call

## Parent

```
void main(void)
{
    pid=fork();
    if (pid==0)
        ChildProcess();
    else
        ParentProcess();
}
}
Void ChildProcess(void)
{
}

Void ParentProcess(void)
{
}
```

PID=3456

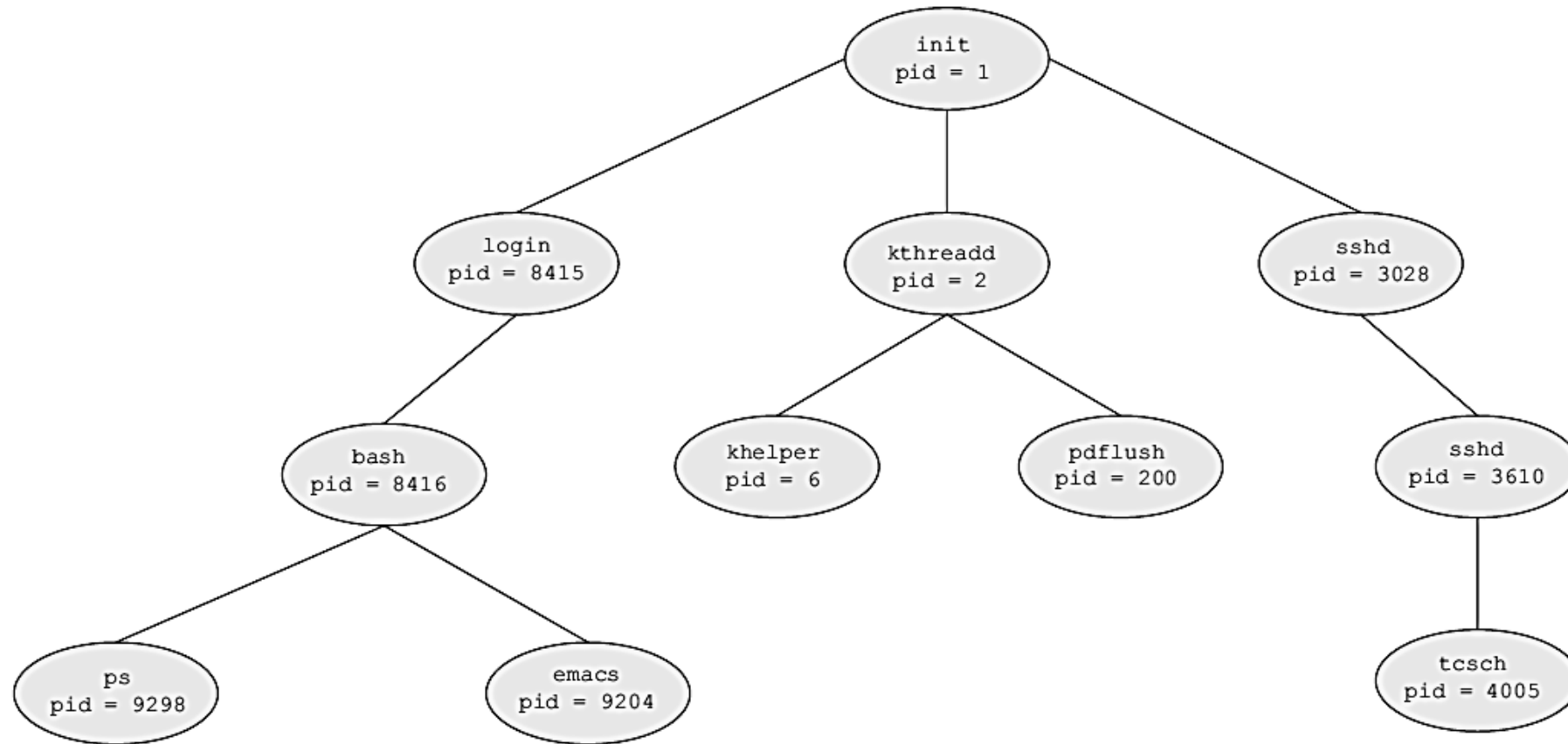
## Child

```
void main(void)
{
    pid=fork();
    if (pid==0)
        ChildProcess();
    else
        ParentProcess();
}
}
Void ChildProcess(void)
{
}

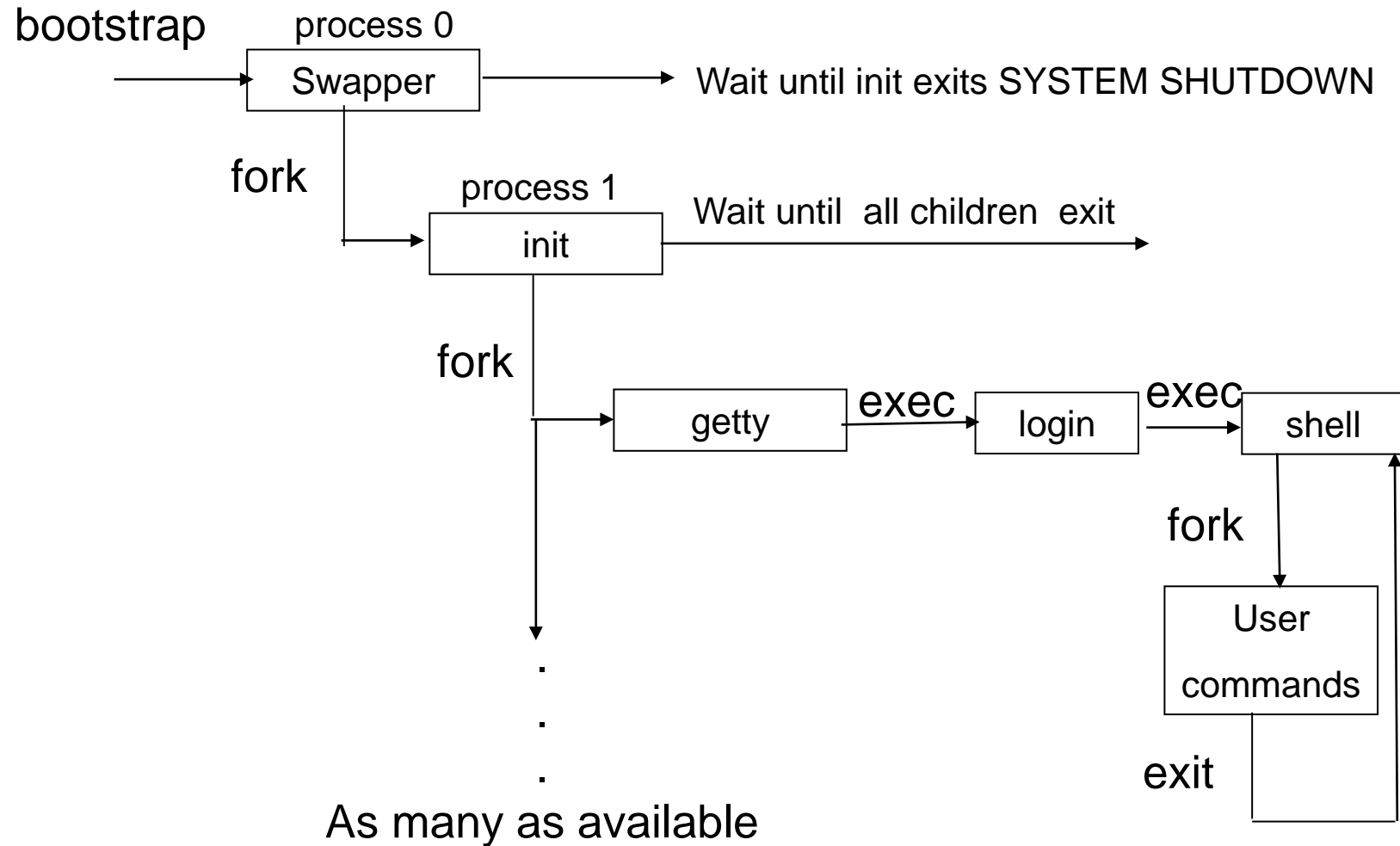
Void ParentProcess(void)
{
}
```

PID=0

# Processes Tree on a UNIX System



# UNIX system initialization



# Process Termination

- Process executes last statement and asks the operating system to decide it (**exit**).
  - Output data from child to parent (via **wait**).
  - Process' resources are deallocated by operating system.
- Parent may terminate the execution of children processes (**abort**).
  - Child has exceeded allocated resources.
  - Task assigned to child is no longer required.
  - Parent is exiting.
  - Many operating system does not allow child to continue if its parent terminates leading to phenomenon of **Cascading termination**.

# Process Termination

- A process that has terminated, but whose parent has not yet called `wait()`, is known as a **zombie** process.
  - When a process terminates, its resources ~~are deallocated~~ by the operating system. However, its entry in the process table must remain there until the parent calls `wait()`, ~~because the process table contains the process's exit status.~~

```
pid_t pid;  
int status;  
  
pid = wait(&status);
```

- If a parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as **orphans**.
- UNIX address this scenario by assigning the *init* process as the new parent to orphan processes.

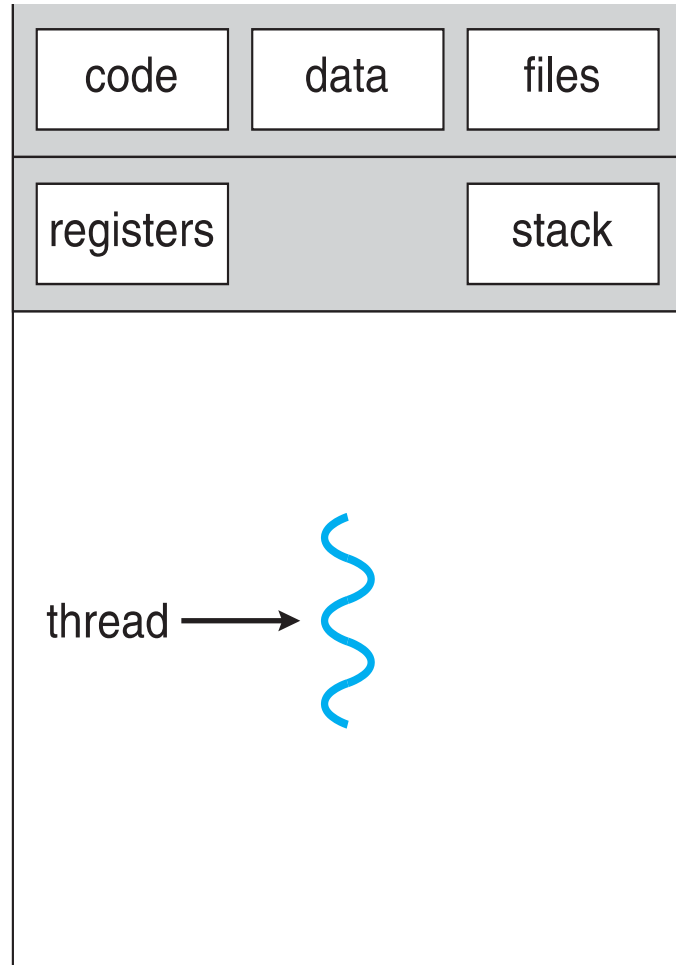
# Cooperating Processes

- The processes can be independent or cooperating processes.
- *Independent* process **cannot** affect or be affected by the execution of another process.
- *Cooperating* process **can** affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up: Break into several subtasks and run in parallel
  - Modularity: Constructing the system in modular fashion.
  - Convenience: User will have many tasks to work in parallel
    - Editing, compiling, printing

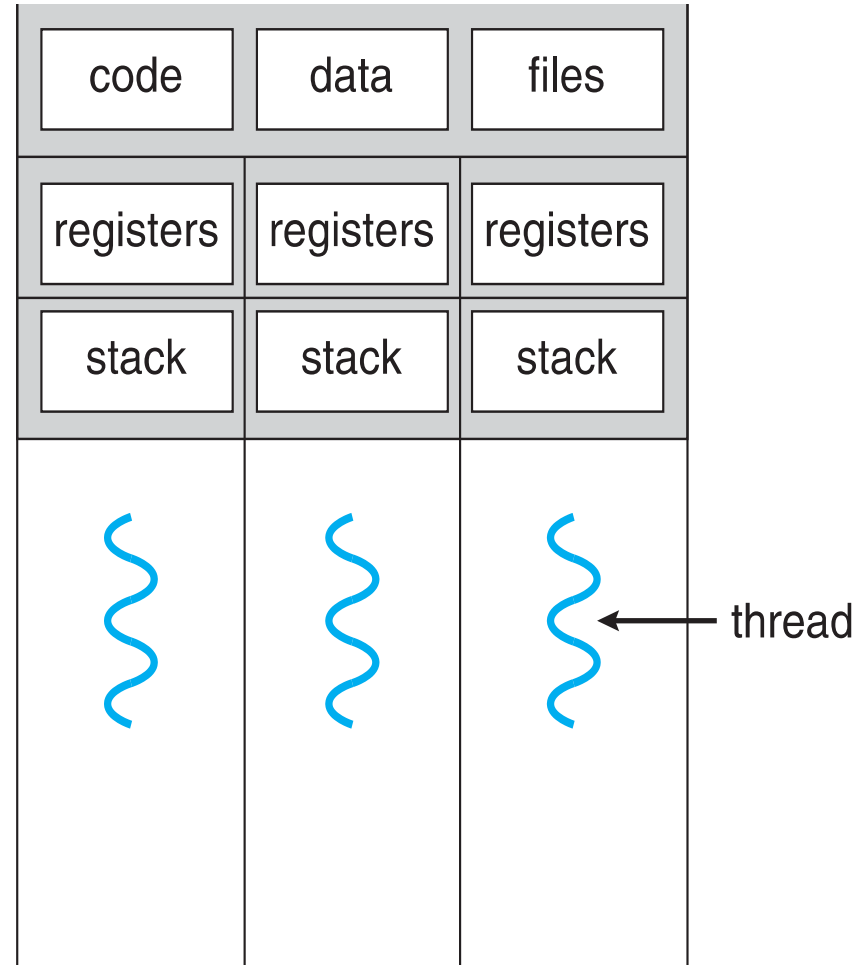
# Definition

- Process: group resources together
- Thread: entity scheduled for execution in a process
- **“Single sequential stream of instructions within a process”**
- “Lightweight process”

# Thread of Execution



single-threaded process



multithreaded process



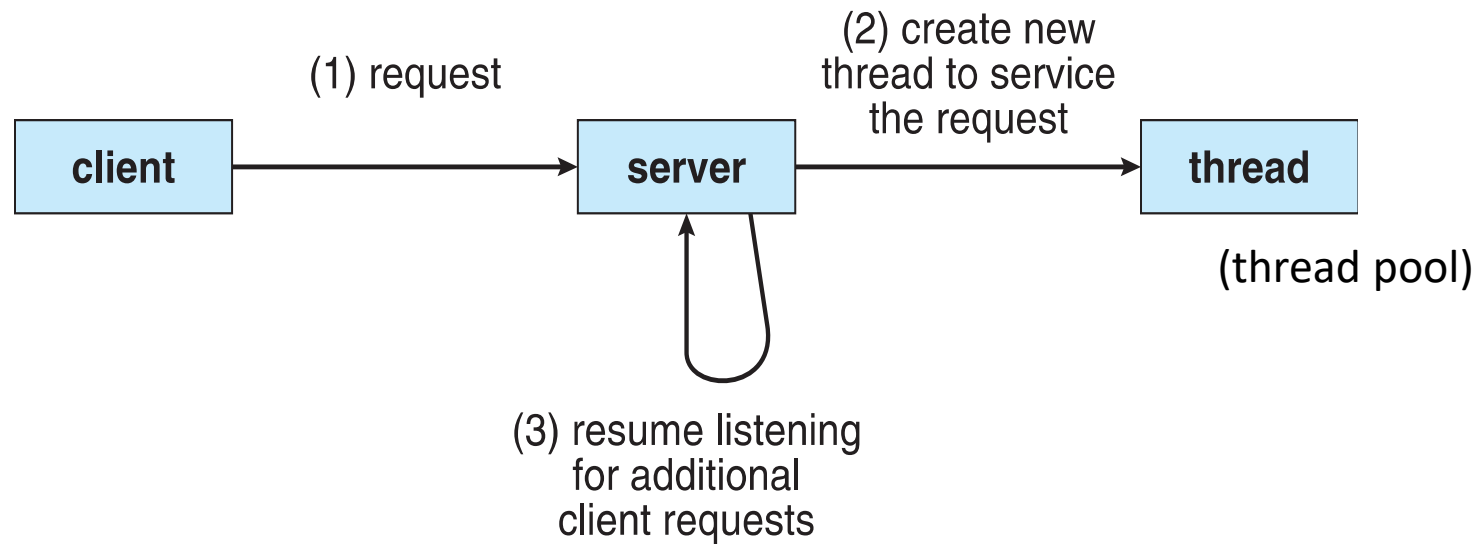
# Thread vs. Process

- Threads have their own:
  - Thread ID (TID) (compare to PID)
  - Program counter (PC)
  - Register set
  - Stack
- Threads commonly share:
  - Code section (text)
  - Data section
  - Resources (files, signals, etc.)

# Why Threads?

- Enable **multi-tasking** within an app
  - Update display
  - Fetch data
  - Spell checking
  - Answer a network request
- Reduced **cost** (“lightweight” process)
  - Processes are heavy to create
  - IPC for threads cheaper/easier than processes
- Can “simplify” code & increase efficiency
- Kernels are generally multithreaded (different threads provide different OS services)

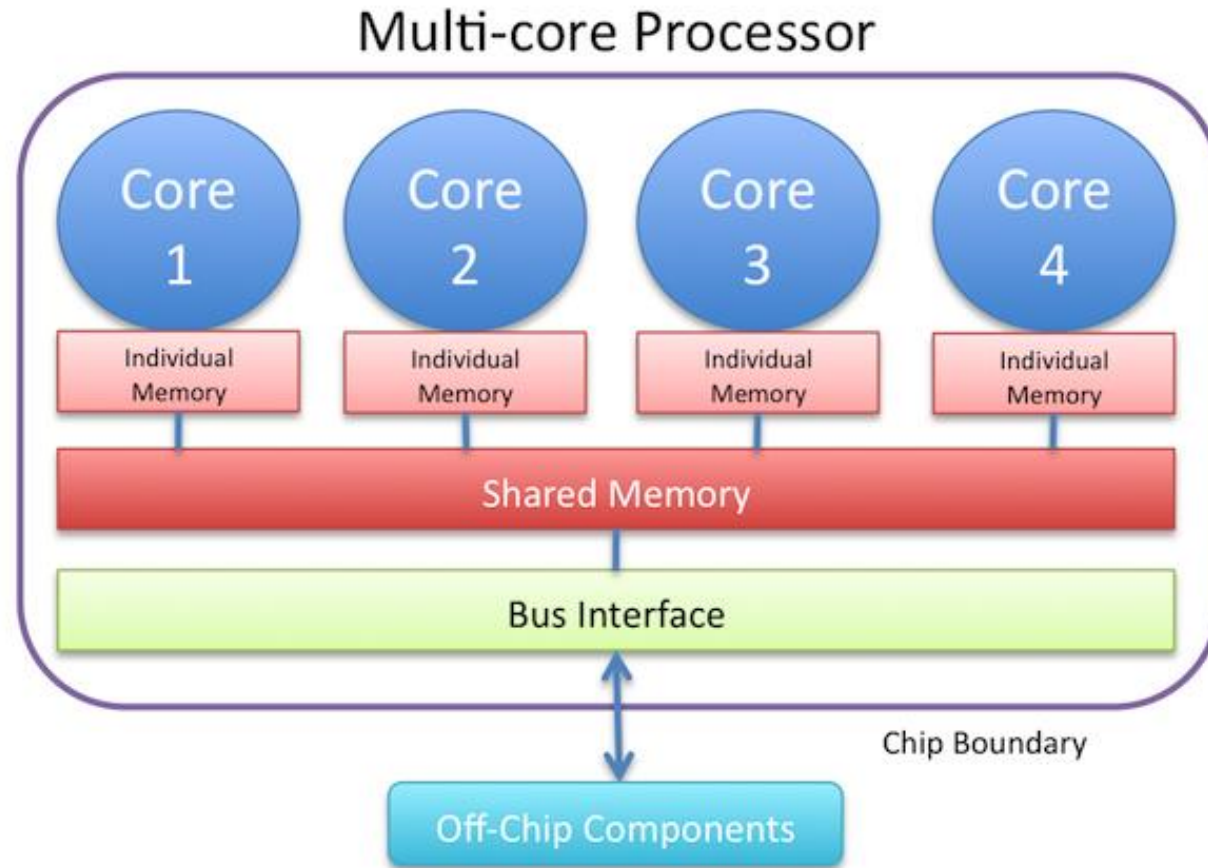
# Multi-Threaded Server



# 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
- **Scalability** – process can take advantage of multiprocessor architectures

# Multicore Systems



# Multicore Programming

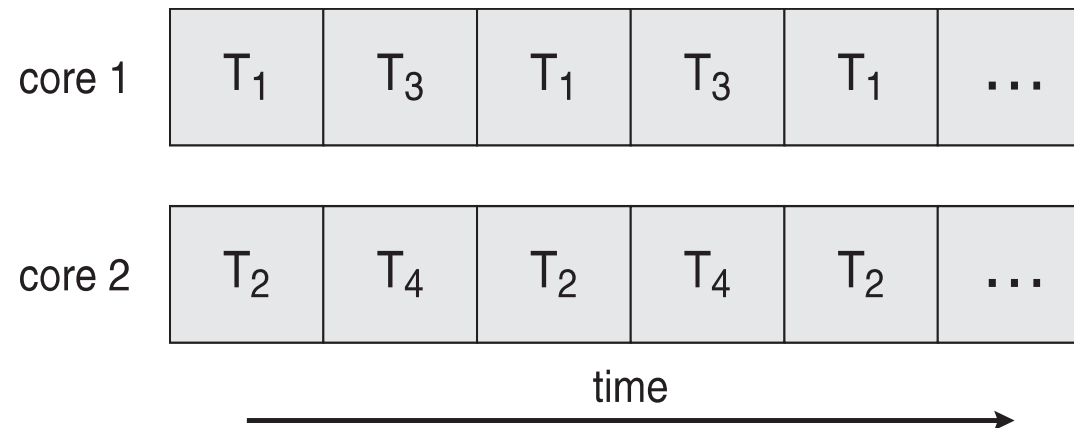
- **Multicore** systems putting pressure on programmers; challenges include:
  - **Dividing activities** (which tasks to parallelize)
  - **Balance** (if/how to parallelize tasks)
  - **Data splitting** (how to divide data)
  - **Data dependency** (thread synchronization)
  - **Testing and debugging** (how to test different execution paths)
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
  - Single processor/core, scheduler providing concurrency

# Concurrency vs. Parallelism

## □ Concurrent execution on single-core system



## □ Parallelism on a multi-core system



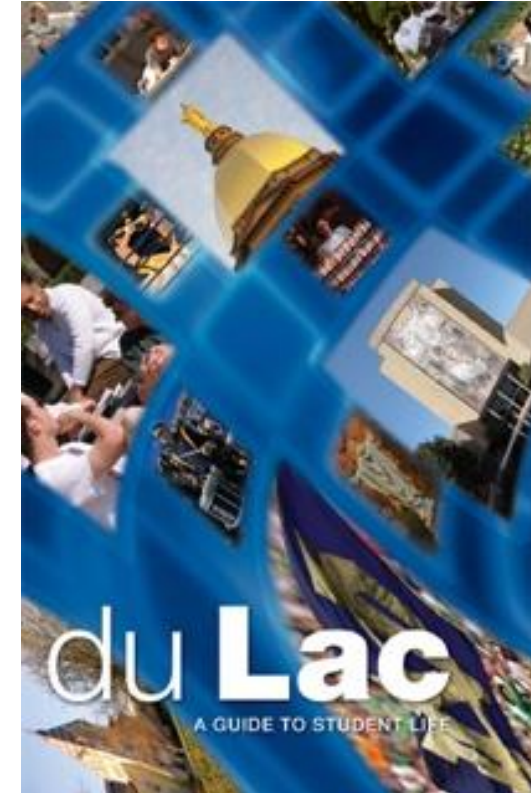
# Multicore Programming

- 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 (“hyperthreading”)
  - CPUs have cores as well as ***hardware threads***
  - Consider Oracle SPARC T4 with 8 cores and 8 hardware threads per core

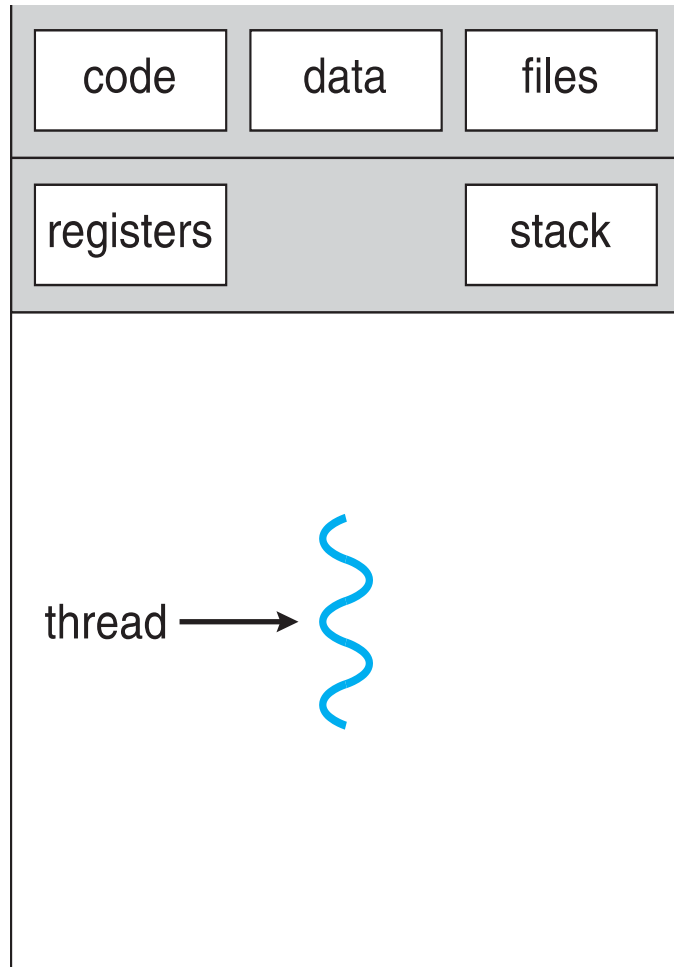


# Data vs. Task Parallelism

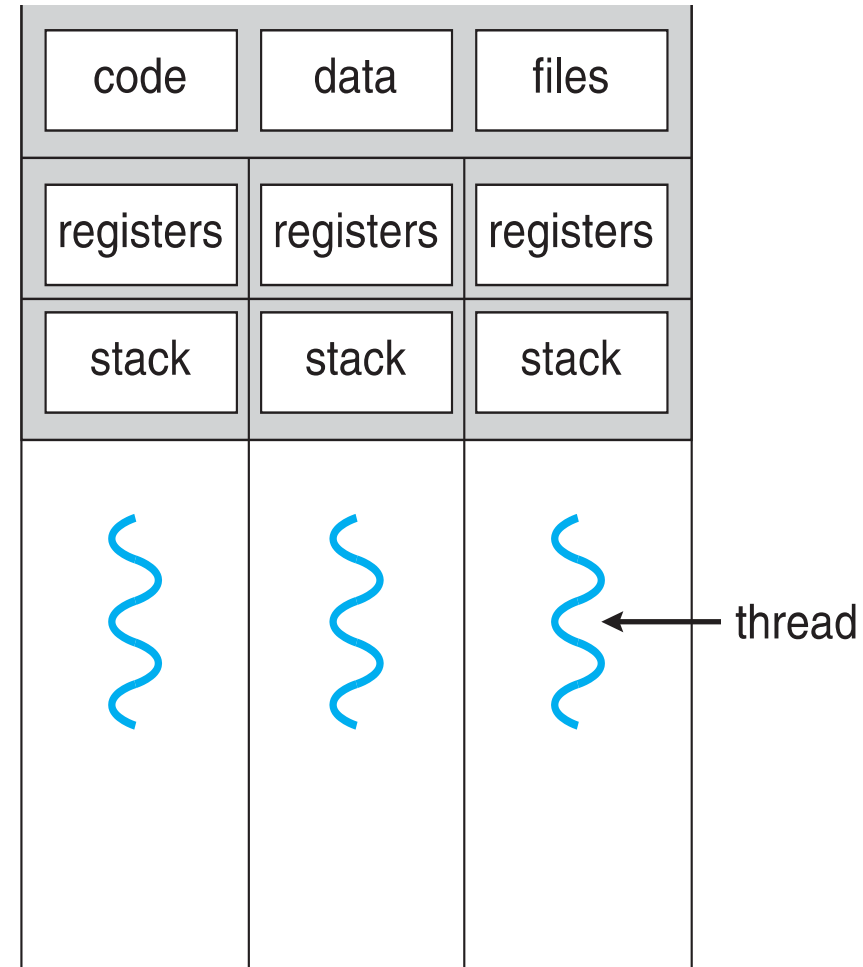
- Count number of times each character in alphabet occurs
- Data Parallelism
  - Thread 1 does page 1-100
  - Thread 2 does page 100-200
- Task Parallelism
  - Thread 1 does letters A-F, all pages
  - Thread 2 does letters G-L, all pages



# Single and Multithreaded Processes



single-threaded process



multithreaded process

# User Threads and Kernel Threads

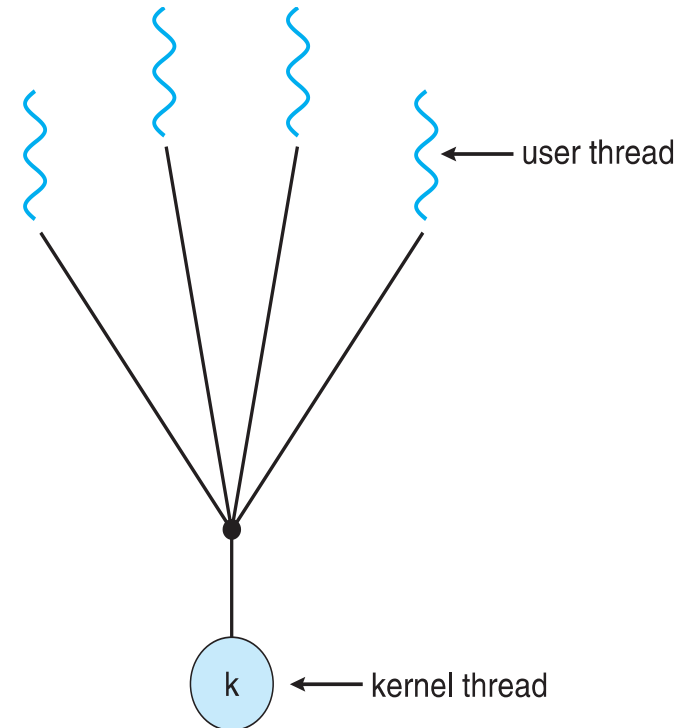
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Win32 threads
  - Java threads
- **Kernel threads** - Supported by the Kernel, “**schedulable entity**”
- Examples – virtually all general purpose operating systems, including:
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

# Multithreading Models

- Many-to-One
- One-to-One
- Many-to-Many

# Many-to-One

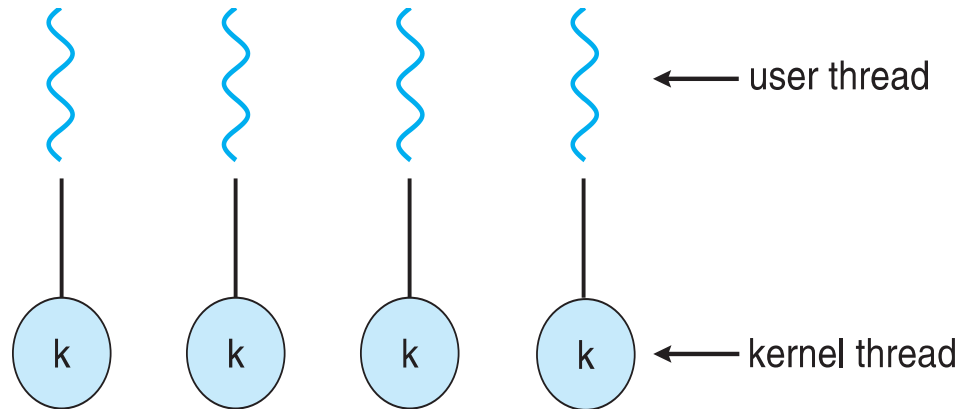
- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable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



# One-to-One

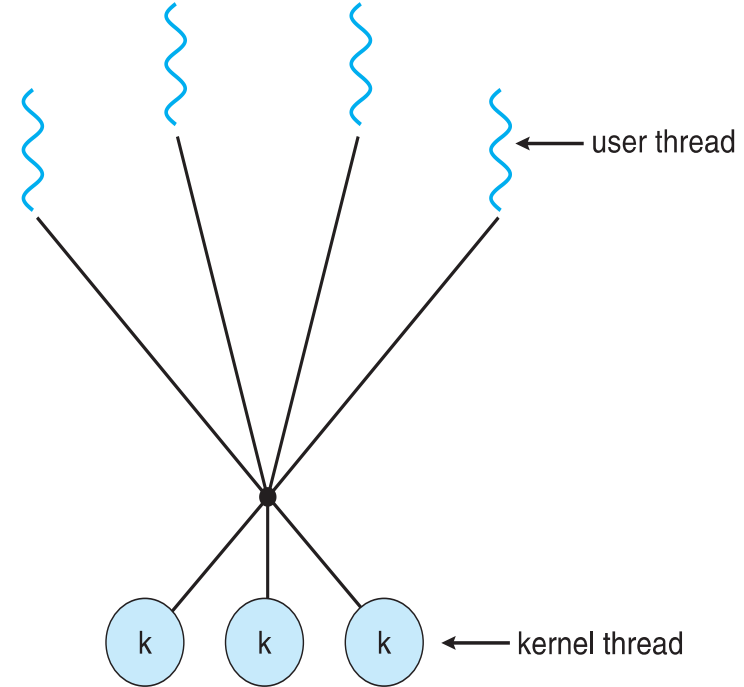
- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead

- Examples
  - Windows NT/XP/2000
  - Linux
  - Solaris 9 and later



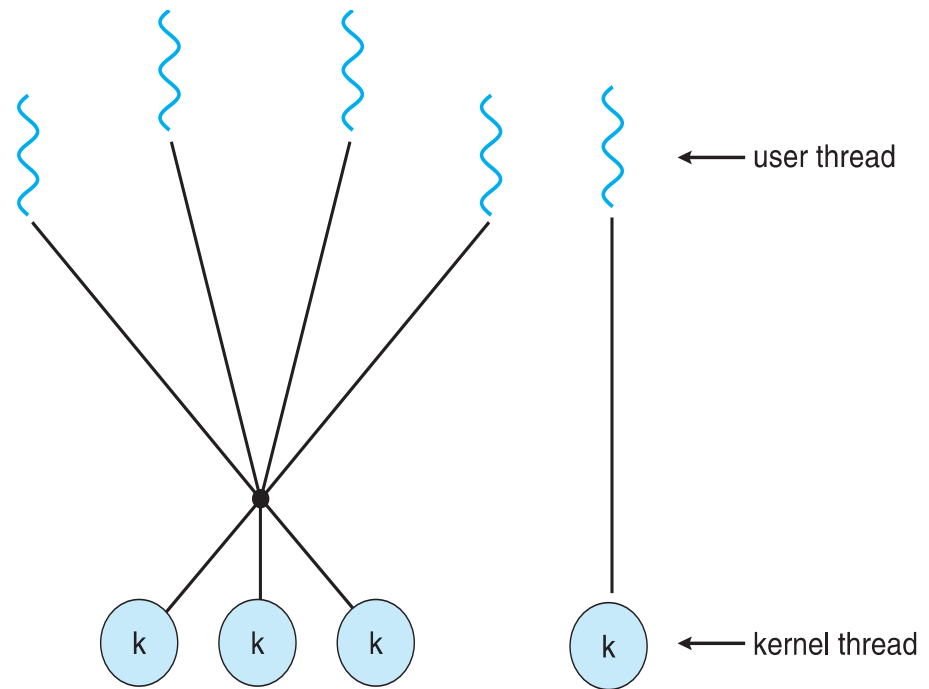
# Many-to-Many Model

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



# Two-level Model

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





# 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
- ***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)

# 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;
    }
}
```

# 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);
}
```

**Figure 4.9** Multithreaded C program using the Pthreads API.

# 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);
```

**Figure 4.10** Pthread code for joining ten threads.

# Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Examples:
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
  - Microsoft Threading Building Blocks (TBB)

# Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e., tasks could be scheduled to run periodically

- Windows API:

```
ThreadPool.QueueUserWorkItem(new WaitCallback(ThreadProc));
```

```
...
```

```
static void ThreadProc(Object stateinfo) {
```

```
...
```

```
}
```

# 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

```
#pragma omp parallel
```

Create as many threads as there are cores

```
#pragma omp parallel for  
for(i=0; i<N; i++) {  
    c[i] = a[i] + b[i];  
}
```

Run for loop in parallel

```
#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;  
}
```



# 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 “`^{} - ^{ printf("I am a block"); }`”
  - Blocks placed in dispatch queue
    - Assigned to available thread in thread pool when removed from queue
  - Two types of dispatch queues:
    - serial – blocks removed in FIFO order, queue is per process, called **main queue**
      - Programmers can create additional serial queues within program
    - concurrent – removed in FIFO order but several may be removed at a time
- ```
dispatch_queue_t queue = dispatch_get_global_queue  
(DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);  
  
dispatch_async(queue, ^{ printf("I am a block."); });
```

# Threading Issues:

## Semantics of `fork()` and `exec()`

- Does **`fork()`** duplicate only the calling thread or all threads?
  - When is duplicating all threads a really bad idea?
  - Some OSes have two versions of `fork`
  - POSIX: only the calling thread
- **`Exec()`** usually works as normal – replace the running process including all threads

# Threading Issues:

## Signal Handling

- **Signals** are used in UNIX systems to notify a process that a particular event has occurred.
- A **signal handler** is used to process signals
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled by one of two signal handlers:
    1. default
    2. user-defined
- Every signal has **default handler** that kernel runs when handling signal
  - **User-defined signal handler** can override default
  - For single-threaded, signal delivered to process

# Threading Issues:

## Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

# Thread Cancellation (Cont.)

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state
  - `pthread_setcancelstate()` -> **enable/disable**
  - `Pthread_setcanceltype()` :

| Mode         | State    | Type         |
|--------------|----------|--------------|
| Off          | Disabled | –            |
| Deferred     | Enabled  | Deferred     |
| Asynchronous | Enabled  | Asynchronous |

- If thread has cancellation disabled, cancellation remains pending until thread enables it
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - `pthread_testcancel()`
- Asynchronous: terminate immediately

# Thread-Local Storage

- **Thread-local storage (TLS)** allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)
- Different from local variables
  - Local variables visible only during single function invocation
  - TLS visible across function invocations
- Similar to **static** data
  - TLS is unique to each thread

# Linux Threads

n Linux refers to them as **tasks** rather than **threads**

n Thread creation is done through **clone()** system call

n **clone()** allows a child task to share the address space of the parent task (process)

l 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.   |

**THANK YOU**