

Recitation this Week

Topic: System call in Linux kernel

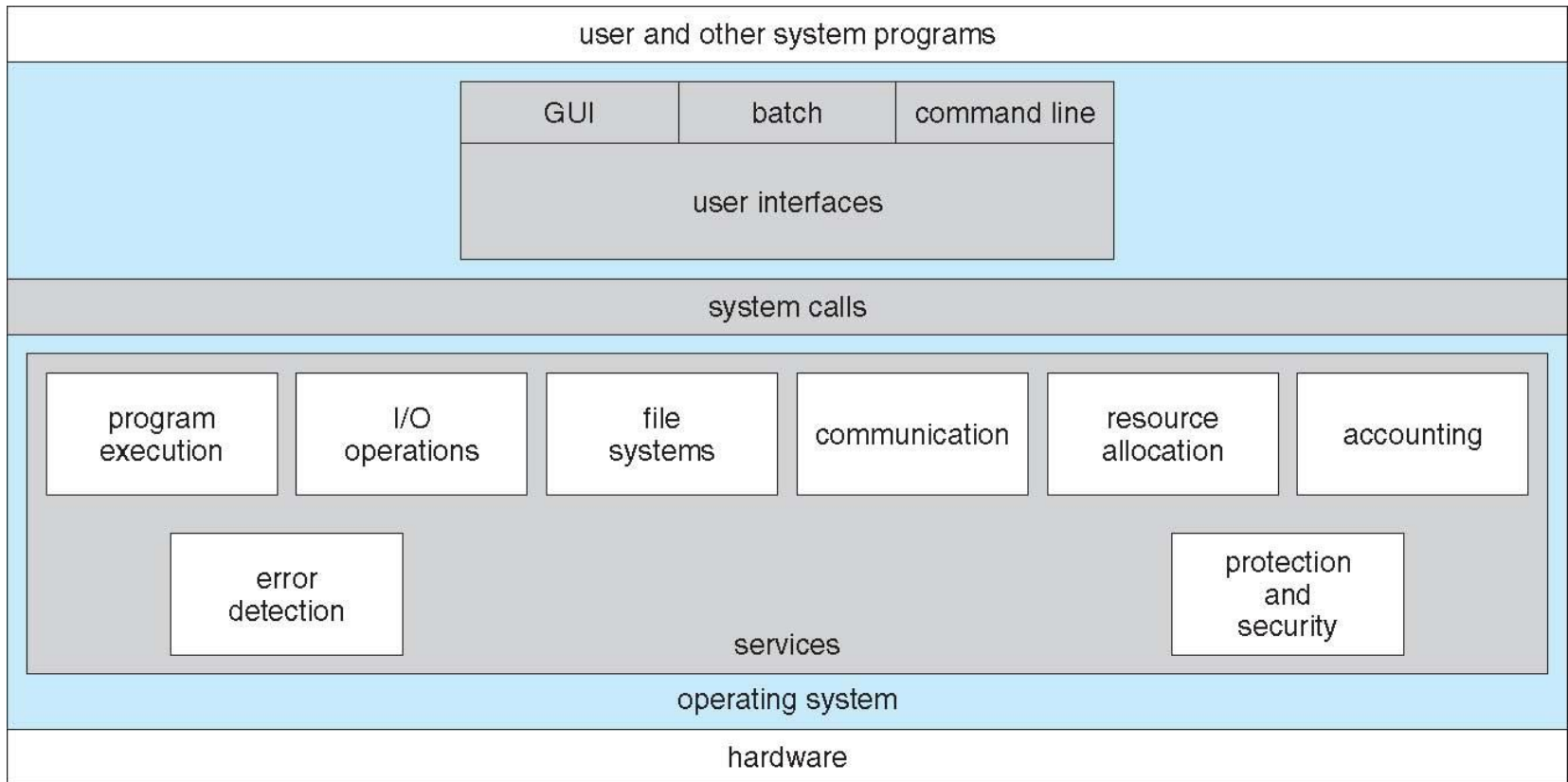
Review: how a system call works

Demo: how to add a new system call to Linux kernel

Gilman 1265, Thursday 12:10-1:00pm

System Call

A View of Operating System Services



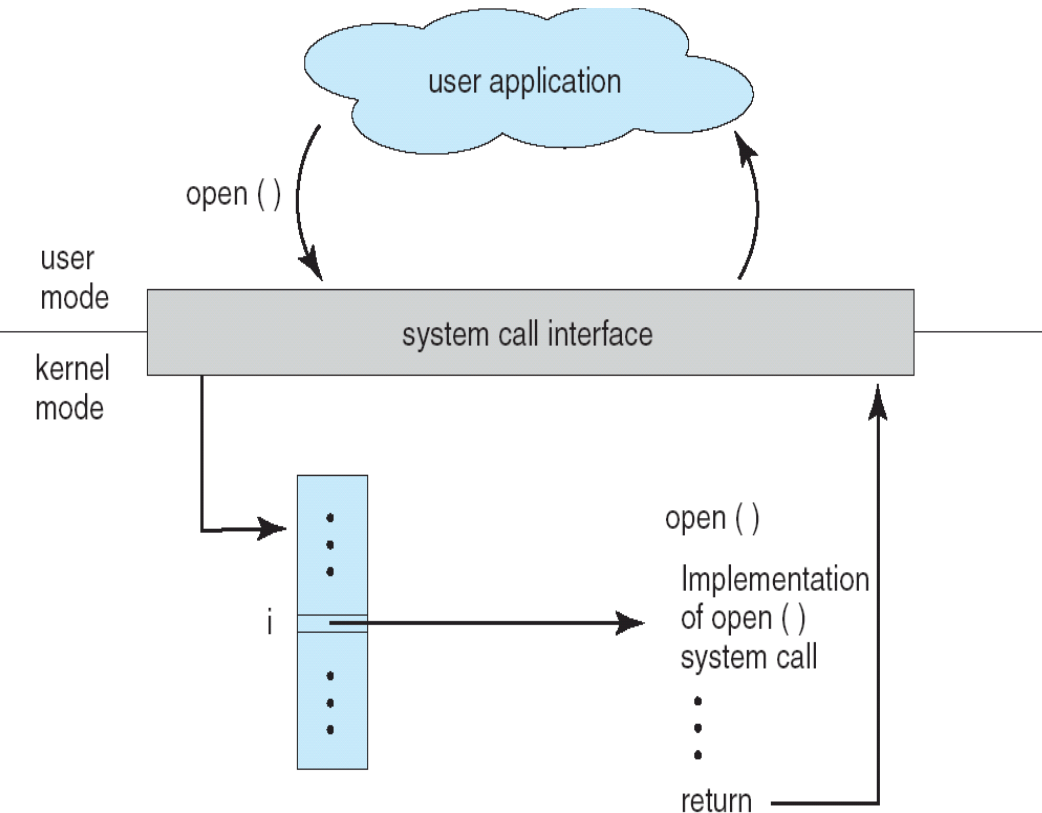
System calls:

Programming interface to the services provided by the OS

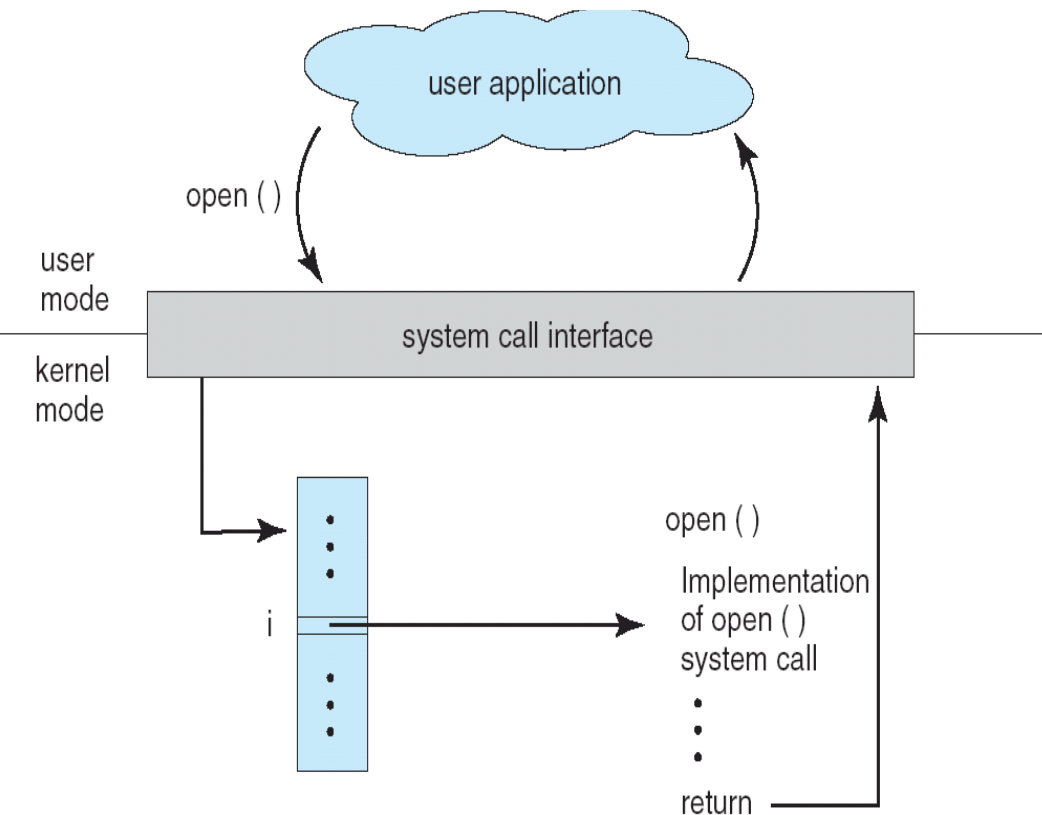
System Call Implementation

🖥️ In kernel

- 🖥️ Associates a number with each system call
- 🖥️ Maintains a table indexed according to these numbers; each entry of the table points to the system service code

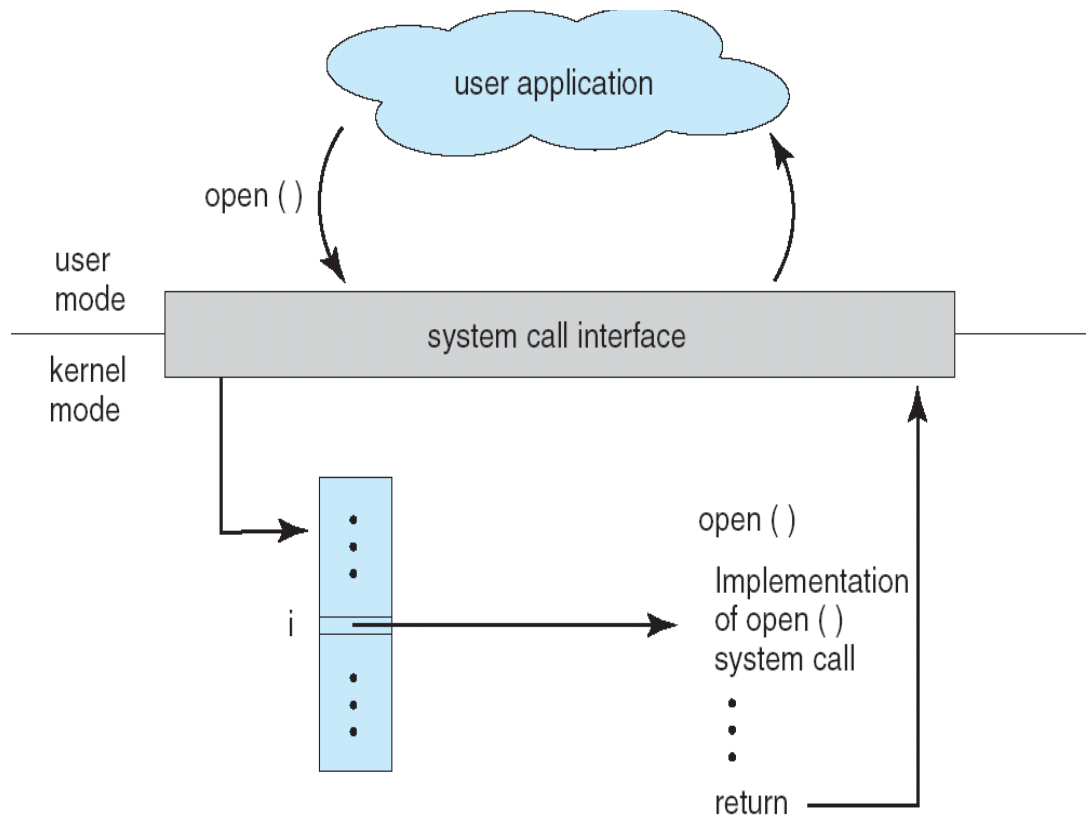



System Call Implementation



🖥️ In User mode: System-call interface is provided by the run-time support system (i.e., part of the libraries of a compiler)

System Call Implementation

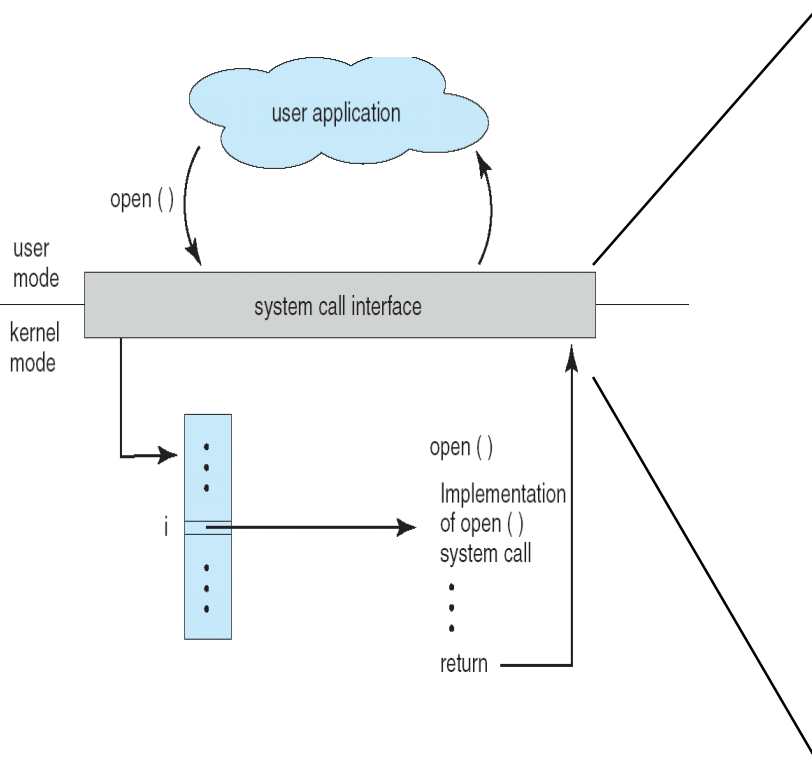


 The system call interface includes stub functions that can invoke the implementation code of intended system calls in OS kernel and return status of the system calls and any return values

System Call Implementation

Example: In system call interface (library), stub function for open:

```
open() {  
    ... load arguments...  
    eax ← _NR_open; //NR_open is the  
    internal ID of the system call  
  
    INT 80h; //interrupt is triggered  
    ...  
}
```



Types and Examples of System Calls

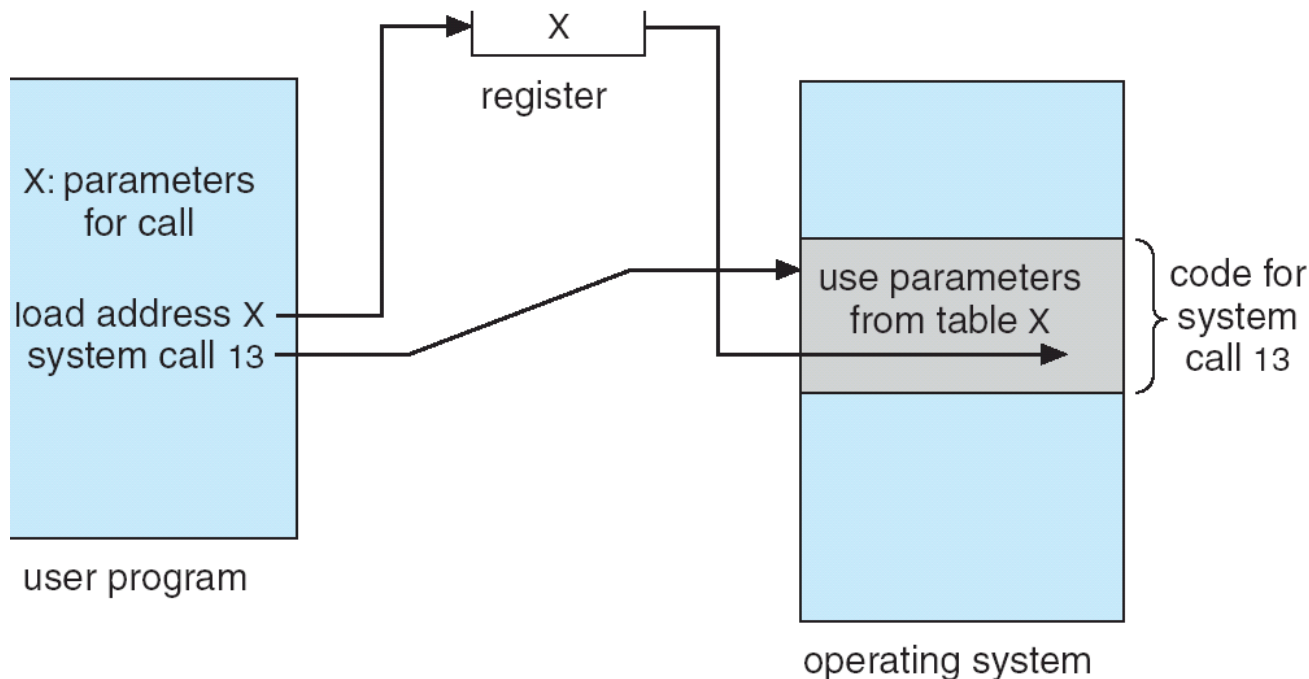
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()
Protection	SetFileSecurity() InitializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

System Call Parameter Passing

- ❏ Often, more information is required than simply identity of desired system call
 - ❏ Exact type and amount of information vary according to OS and call
- ❏ Three general methods used to pass parameters to the OS
 - ❏ Simplest: pass the parameters in *registers*

System Call Parameter Passing

- Three general methods used to pass parameters to the OS
 - Simplest: pass the parameters in *registers*
 - Parameters stored in a *block*, or table, in memory, and address of block passed as a parameter in a register
 - This approach taken by Linux and Solaris



System Call Parameter Passing

- ❏ Three general methods used to pass parameters to the OS
 - ❏ Simplest: pass the parameters in *registers*
 - ❏ Parameters stored in a *block*, or table, in memory, and address of block passed as a parameter in a register
 - ❏ This approach taken by Linux and Solaris
 - ❏ Parameters placed, or *pushed*, onto the *stack* by the program and *popped* off the stack by the operating system
 - ❏ Block and stack methods do not limit the number or length of parameters being passed

System Calls vs API

- ❏ System services are mostly accessed by programs via a high-level [Application Program Interface \(API\)](#) rather than direct system call
- ❏ Three most common APIs are Win32 API for Windows, POSIX API for POSIX-based systems (including virtually all versions of UNIX, Linux, and Mac OS X), and Java API for the Java virtual machine (JVM)
- ❏ Why use APIs rather than system calls? [Portability!](#) [Transparency!](#)

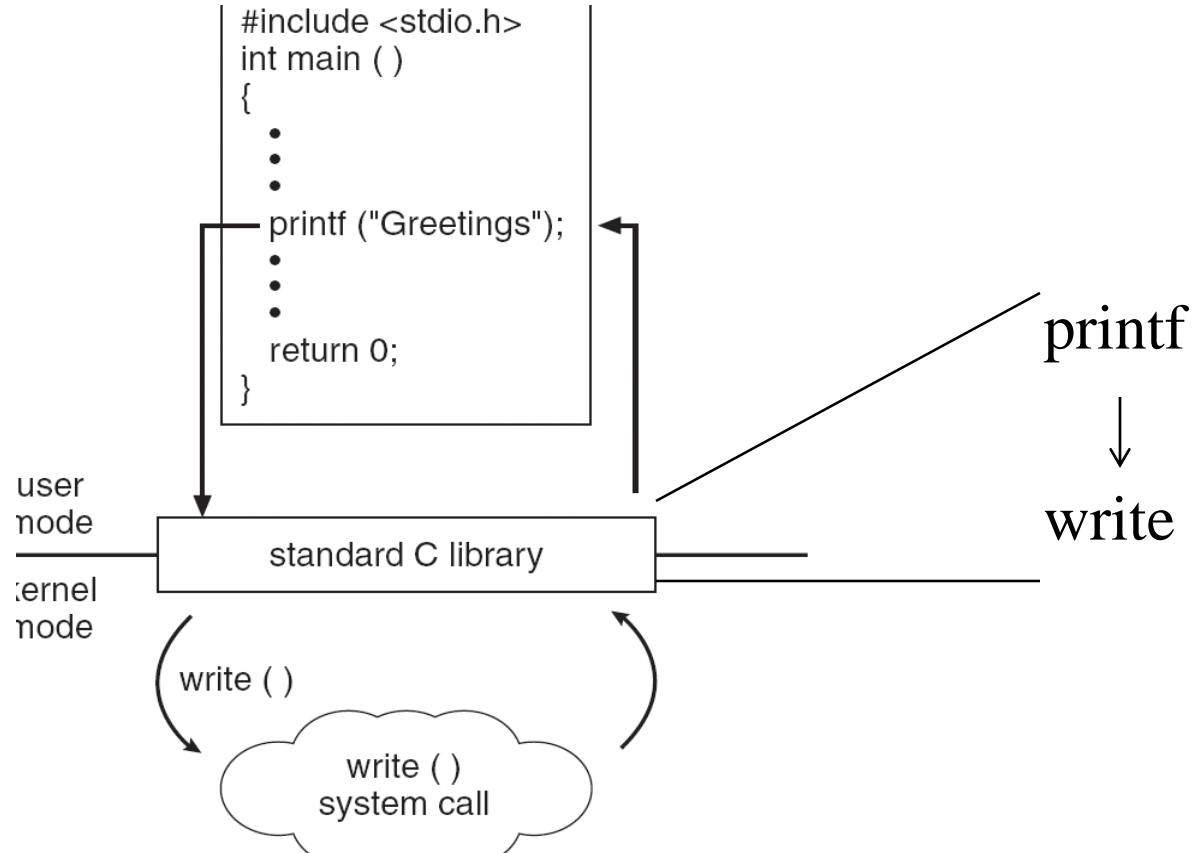


System Calls vs API

- ❏ The caller needs know nothing about how the system call is implemented
 - ❏ Just needs to obey API and understand what OS will do as a result
 - ❏ More details of OS interface hidden from programmer by API (provided by programming language libraries)

Standard C Library Example

 C program invoking printf() library call, which calls write() system call



System Programs

- ❏ System programs provide a convenient environment for program development and execution. They fall into these categories:
 - ❏ File manipulation: `ls`, `mkdir`, `rm`, `cd`, ...
 - ❏ Status information of the system: `date`, `uname`, `df`, ...
 - ❏ File modification: `cat`, `chmod`, ..
 - ❏ Programming language support: `cc`, `gcc`, ...
 - ❏ Program loading and execution: `ld`, `link`, ...
 - ❏ Communications: `talk`, `message`, ...
- ❏ Implementation: Some are simply user interfaces to system calls; others are considerably more complex
- ❏ Most users' view of the operation system is defined by system programs, not the actual system calls

Example

System program: mkdir

Usage: mkdir [OPTION] ... DIREcTORY ...

Create the DIREcTORY(ies), if they do not already exist.

Mandatory arguments to long options are mandatory for short options too.

-m, --mode=MODE set file mode (as in chmod), not a=rwx – umask

-p, --parents no error if existing, make parent directories as needed

-v, --verbose print a message for each created directory

-Z, --context=CTX set the SELinux security context of each created
directory to CTX

--help display this help and exit

--version output version information and exit

(run “mkdir –help” can get the above)

Example

SYNOPSIS

```
#include <sys/stat.h>
#include <sys/types.h>
int mkdir(const char *pathname, mode_t mode);
```

DESCRIPTION

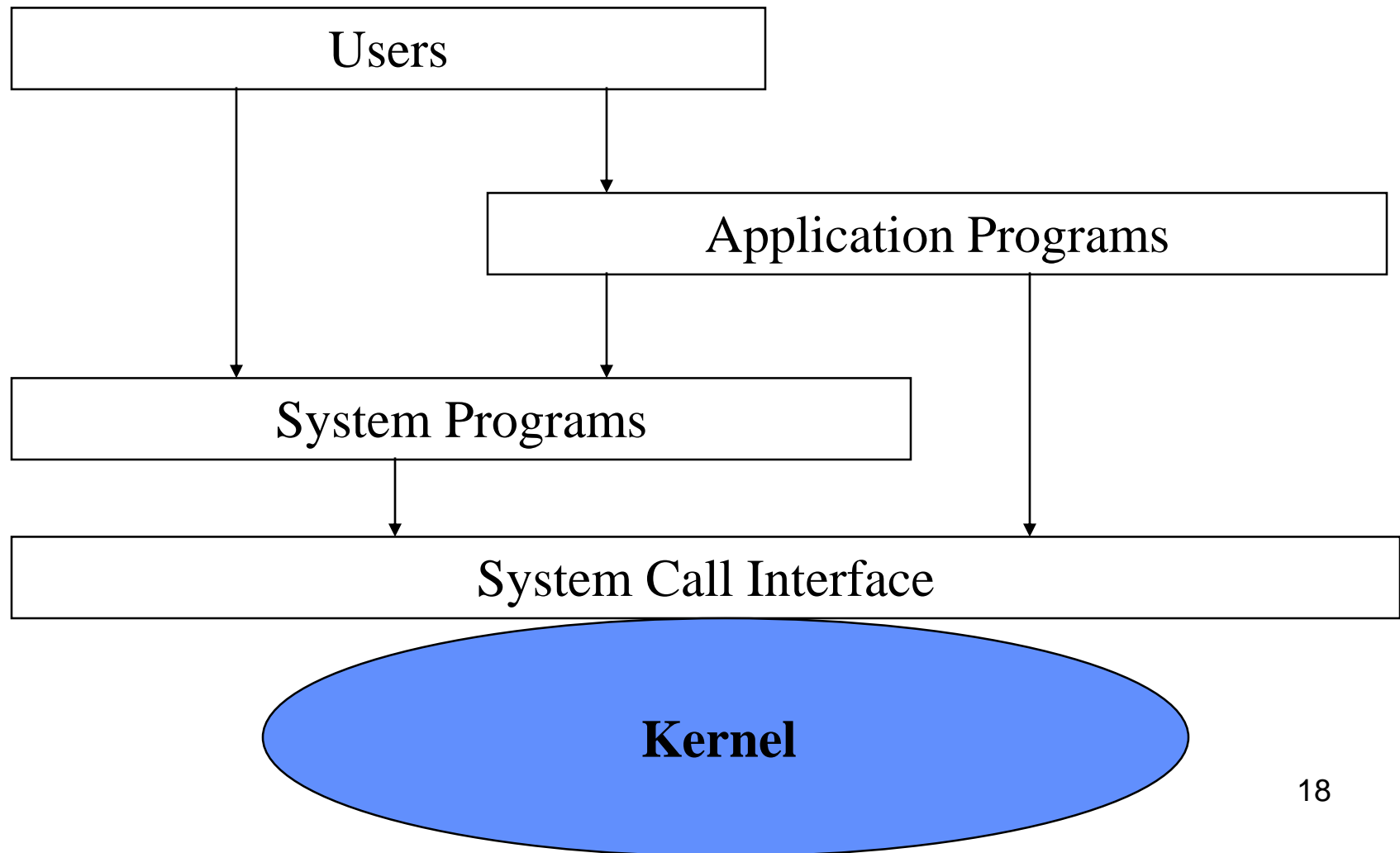
mkdir() attempts to create a directory named pathname.

The argument mode specifies the permissions to use. It is modified by the processes umask in the usual way: the permission of the created directory are (mode & ~umask & 0777). Other mode bits of the created directory depend on the operating system. For Linux, see below.

... ..

(run “man 2 mkdir” can get the above)

Relation between Users, Apps, System Calls, and System Programs



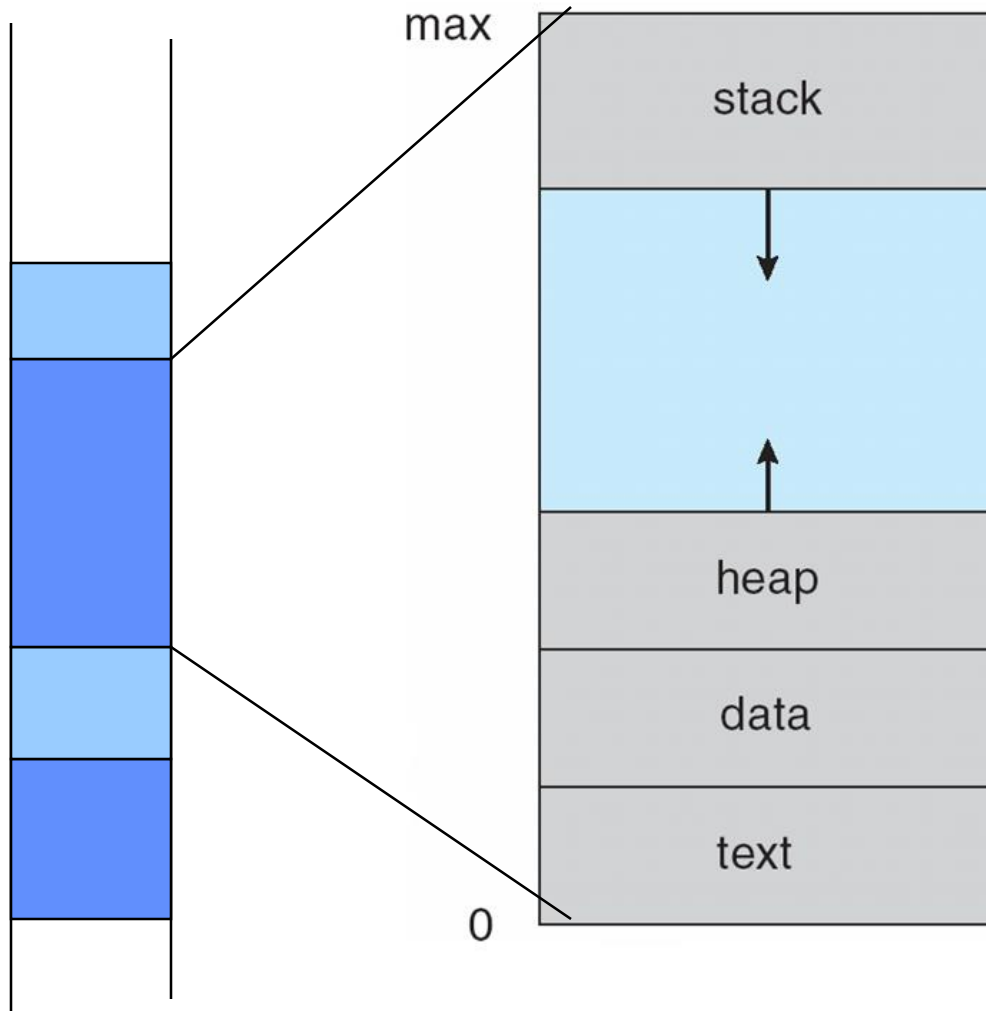
Summary

- System call vs procedure call
- System call vs programming language API
- Implementation details of system calls
- System programs vs Application programs

Process Management (I)

August 28, 2017

Process Concept



- ❏ Process – a program in execution (def. in books)
- ❏ Process – a data structure for the OS to manage a program.
- ❏ Traditional Process execution progresses in sequential fashion
- ❏ The user memory space of a process includes:
 - ❏ text section (code)
 - ❏ stack
 - ❏ data section
 - ❏ heap

Process Control Block (PCB)






Information associated with each process

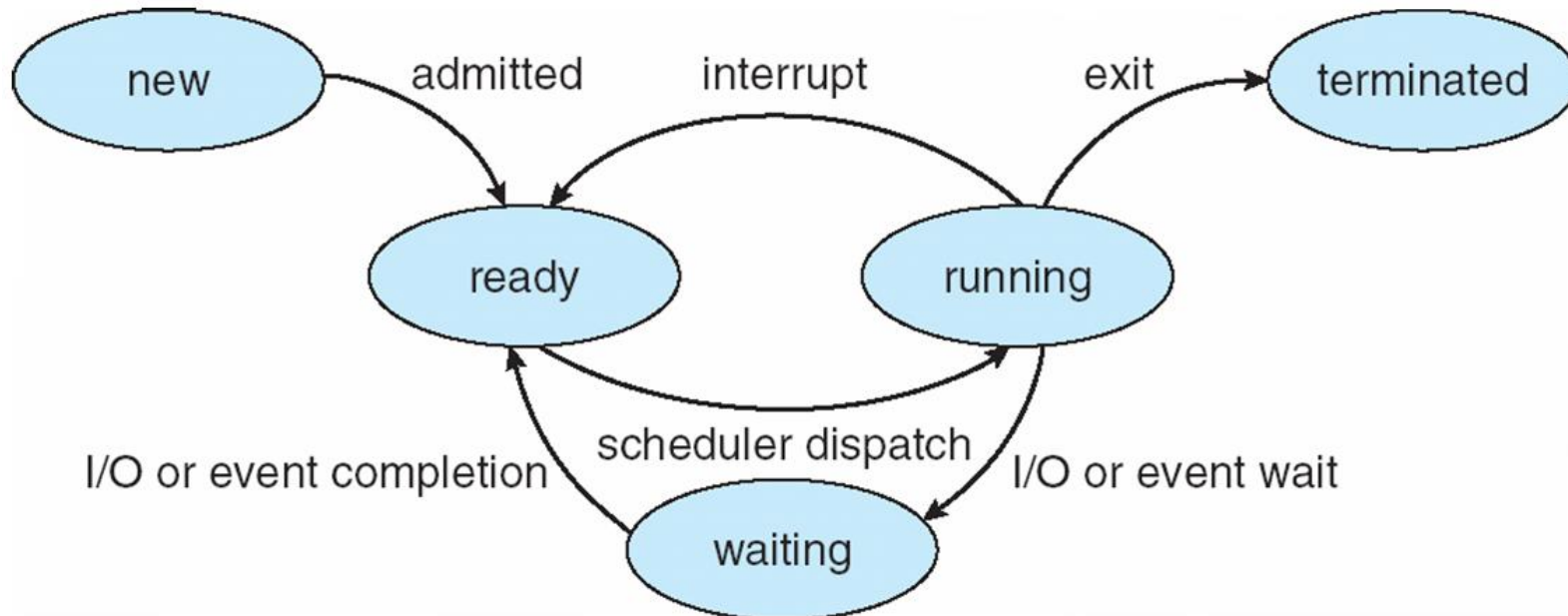
process state
process number
program counter
registers
memory limits
list of open files
...

- ❏ Process state: new/ready/running/waiting
- ❏ Process number: uniquely identifying a process
- ❏ Program counter: **used to** store the address of the instruction to be executed next
- ❏ Registers: **used to** store values of CPU registers
- ❏ CPU scheduling information
- ❏ Memory-management information
- ❏ ...

PCBs are stored in kernel space (not in user space)

Process State

-  **new:** The process is being created
-  **running:** Instructions are being executed
-  **waiting:** The process is waiting for some event to occur
-  **ready:** The process is waiting to be assigned to a processor
-  **terminated:** The process has finished execution



Process Creation: Unix Example

fork system call

- ❏ is called by a process (denoted as parent process) to create a child process
- ❏ makes a new (child) process which is a duplicate of the parent process except that :
 - ❏ new process has a different process ID (PID)
 - ❏ fork system call returns different values to child and parent
 - ❏ returns 0 to child process
 - ❏ returns PID of the child process to the parent process

C Program Forking Separate Process

....

```
int main() {  
    pid_t pid;  
    int var;  
    (1) var =0;  
    (2) pid = fork(); /* fork another process */  
    (3) if (pid < 0) { /* error occurred */  
    (4)         fprintf(stderr, "Fork Failed");  
    (5)         exit(-1);  
    (6) }else if (pid == 0) { /* child process */  
    (7)         var=1;  
    (8) }else{ /* parent process */  
    (9)         var=2;  
    (10)        wait (NULL);  
    (11)        printf ("In parent process, var=%d\n", var);  
    (12)        exit(0);  
    (15)}  
    (14)printf("In child process, var=%d\n", var);  
}
```

Change Child Process's Code

```
#include <unistd.h>
```

```
int execlp (const char *path, const char *arg0, ..., NULL);
```

path: path of the program file

arg0: name of the program file




arg 1, arg2 , ..., NULL: arguments ending with NULL

Change Child Process's Code/Data

... ..

```
int main() {  
    pid_t pid;  
    (1) pid = fork(); /* fork another process */  
    (2) if (pid < 0) { /* error occurred */  
    (3)         fprintf(stderr, "Fork Failed");  
    (4)         exit(-1);  
    (5) } else if (pid == 0) { /* child process */  
    (6)         execvp("/bin/ls", "ls", NULL);  
    (7) } else { /* parent process */  
            /* parent will wait for the child to complete */  
    (8)         wait (NULL);  
    (9)         printf ("Child Complete");  
    (10)        exit(0);  
    }  
}
```

Process Termination

-  Process executes last statement and asks the operating system to delete it (**exit**), and then the process' resources are de-allocated.
-  A parent process can make **wait** system call to wait for its child process to terminate before the parent process can continue.
-  Parent may terminate execution of children processes (**abort**) by using kill system call.

An Example

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
int main() {
    pid_t pid;
    int status;
    int i;

    pid=fork();
    if(pid == 0) {
        pid=getpid();
        printf("This is child process
        #%d\n", pid);
        exit(0);
    } else if (pid>0) {
```

```
        pid=fork();
        if(pid == 0) {
            pid=getpid();
            printf("This is child
            process #%d\n", pid);
            while(1) {printf("?");};
        }

        sleep(1);
        printf("process %d
        terminates.\n", wait(NULL));

        kill(pid,SIGKILL);
    }
}
```