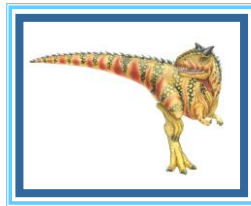


Operating-System Structures



Operating-System Structures

- Operating System Services
- User Operating System Interface
- System Calls
- Types of System Calls
- System Programs
- Operating System Design and Implementation
- Operating System Structure
- Virtual Machines
- Operating System Debugging
- Operating System Generation
- System Boot



Objectives

- To describe the services an operating system provides to users, processes, and other systems
- To discuss the various ways of structuring an operating system
- To explain how operating systems are installed and customized and how they boot



Operating System Services

- Operating systems provide an environment for execution of programs and services to programs and users
- One set of operating-system services provides functions that are helpful to the user:
 - **User interface** - Almost all operating systems have a user interface (UI).
 - Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**, **Batch**
 - **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
 - **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
 - **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file information, permission management.



Operating System Services (Cont.)

- **Communications** – Processes may exchange information, on the same computer or between computers over a network
 - ▶ Communications may be via shared memory or through message passing (packets moved by the OS)
- **Error detection** – OS needs to be constantly aware of possible errors
 - ▶ May occur in the CPU and memory hardware, in I/O devices, in user program
 - ▶ For each type of error, OS should take the appropriate action to ensure correct and consistent computing
 - ▶ Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system



Operating System Services (Cont.)

- Ensuring the efficient operation of the system itself via **resource sharing**
 - **Resource allocation**
When multiple users or multiple jobs running concurrently, resources must be allocated to each of them
 - ▶ CPU cycles (resource #1)
 - ▶ Main memory
 - ▶ File storage
 - ▶ Printers, scanners, camera, etc.
 - **Accounting**
Keep track of which users use how much and what kinds of computer resources
 - ▶ CPU time user is notified if his processes overuse CPU
 - ▶ Disks usage user gets notification if his files fill network disks



Operating System Services (Cont.)

■ Protection and security

Owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other

● Protection

all access to system resources is controlled – OS must provide ownership, permissions, and authentication control engines

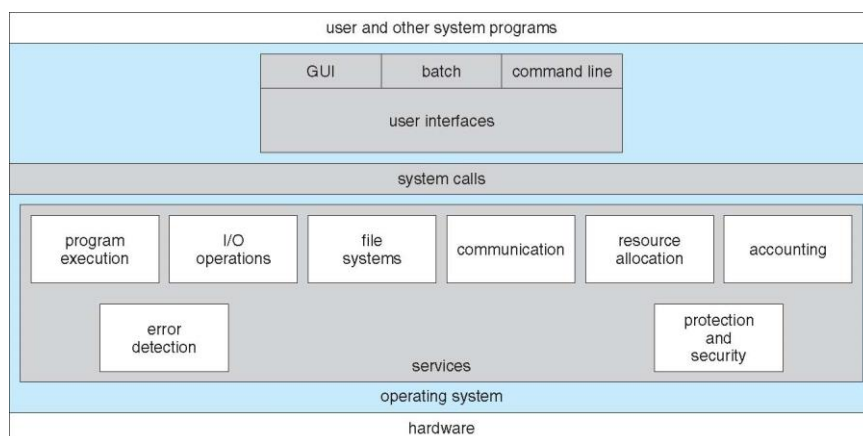
● Security

Protect system from outsiders (and the users themselves) requires

- ▶ User authentication
- ▶ Defend I/O devices from invalid access attempts
- ▶ If a system is to be protected and secure, precautions must be instituted throughout it.
- ▶ A chain is only as strong as its weakest link.



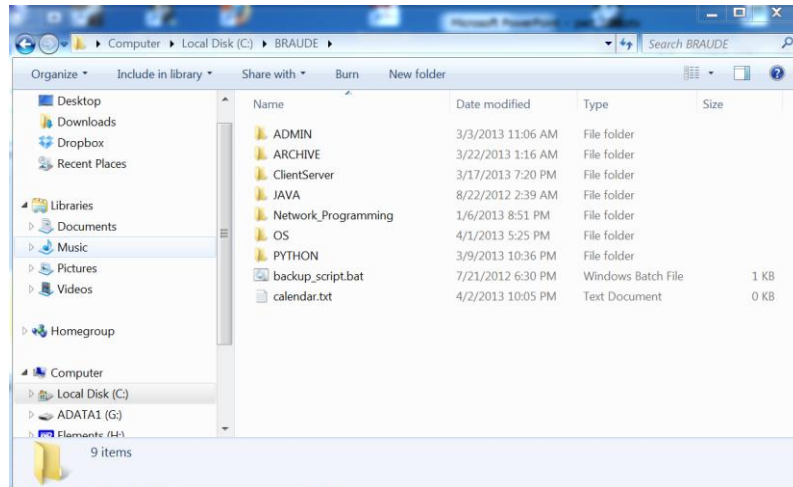
A View of Operating System Services





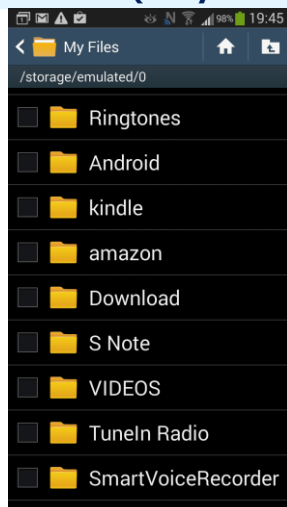
A View of Operating System Services

GUI – Graphical User Interface to File System Windows 7



A View of Operating System Services

GUI – Graphical User Interface to File System Android (Jelly Bean 4.2)





A View of Operating System Services

Batch File Interface with the Operating System (windows)

```
@ECHO OFF
:: Windows NT 4 / 2000 only
IF NOT "%OS%"=="Windows_NT" 1 GOTO Syntax

:: Keep variables local
SETLOCAL ENABLEEXTENSIONS

:: Parameter check
ECHO.%1 | FIND "?" >NUL
IF NOT ERRORLEVEL 1 GOTO Syntax
:: Extract drive letter
SET Drive=%1
IF DEFINED Drive SET Drive=%Drive:~0,1%
CALL :Drive %Drive%

:: FAT
SET FS=FAT
SET FS=FAT
:: Test "last accessed" time, if 00:00 for every file we may presume FAT
FOR /F "TOKENS=2,3* DELIMS= " %%A IN ('DIR/A/TA/P/-P/W/-W %Drive% 2^>NUL ^| FIND ":" ^| FIND "-"") DO IF NOT
"%%A"=="00:00" SET FS=
DIR %Drive% >NUL 2>&1
IF ERRORLEVEL 1 GOTO NotReady
IF NOT "%FS%"==" " GOTO Display

:: NTFS
SET FS=NTFS
SET FS=NTFS
:: NTFS check needs a temporary file name
SET TEMPFIL=
FOR %%A IN (0 1 2 3 4 5 6 7 8 9) DO FOR %%B IN (0 1 2 3 4 5 6 7 8 9) DO CALL :TempFile %%A%%B %1
IF "%TEMPFIL%"==" " GOTO NoTemp
:: Test alternate data streams, a feature unique for NTFS
(ECHO %~nx0 > %TEMPFIL%:NTFSTEST) >NUL 2>&1
IF NOT EXIST %TEMPFIL% SET FS=unknown
IF EXIST %TEMPFIL% DEL %TEMPFIL%
```



UNIX Batch File Example (BASH)

```
hname=`hostname`
echo "Welcome on $hname."
echo -e "Kernel Details: " `uname -smr`
echo -e " " `bash --version` "
echo -ne "Uptime: "; uptime
echo -ne "Server time : "; date
lastlog | grep "root" | awk '{print "Last login from : "$3
print "Last Login Date & Time: ",$4,$5,$6,$7,$8,$9;}'

DISK_SIZE_BYTES=`df -PT -B 1 | awk '{if ($7 == "/") print $3}'`

TEMP_FILE=/tmp/largest_files.tmp

printf "\n%-18s %-10s %-15s %-25s %s\n\n" "[SIZE (BYTES)]" "[% OF DISK]" \
"[OWNER]" "[LAST MODIFIED ON]" "[FILE]"

SUM=0

for FILE in `find $SEARCH_DIR -type f -exec du {} \; | sort -rn | head -$FILES_COUNT | awk '{print $2}'`
do
FILE_SIZE_BYTES=`stat --print %s $FILE`
OWNER=`stat --printf %U $FILE`
LAST_MODIFIED_ON=`stat --printf %y $FILE | cut -c 1-19`

FILE_SIZE_BYTES1=`expr $FILE_SIZE_BYTES \* 100`
PERCENTAGE=`expr $FILE_SIZE_BYTES1 / $DISK_SIZE_BYTES` %
printf "\n%-18s %-12s %-14s %-25s %s" \
"$FILE_SIZE_BYTES" "$PERCENTAGE" "$OWNER" "$LAST_MODIFIED_ON" "$FILE"
SUM=$((SUM+FILE_SIZE_BYTES))
done
```



A View of Operating System Services

CLI – Command Line Interface (Windows)

C:\Windows\system32\cmd.exe

```
C:\BRAUDE>find.py proj*.py

C:\BRAUDE\PYTHON\Courses\MSU_COURSE\proj02.py
C:\BRAUDE\PYTHON\Courses\MSU_COURSE\proj06.py
C:\BRAUDE\PYTHON\Courses\MSU_COURSE\proj09Skel.py
C:\BRAUDE\PYTHON\Projects\PROJ1\proj1.py
C:\BRAUDE\PYTHON\Projects\PROJ2\proj2_advanced_solution.py
C:\BRAUDE\PYTHON\Projects\PROJ2\proj2_prep.py
C:\BRAUDE\PYTHON\Projects\PROJ2\proj2_solution.py

C:\BRAUDE>
```

Exercise: Write find.py in Python (20 lines!) [\[link to solution\]](#)



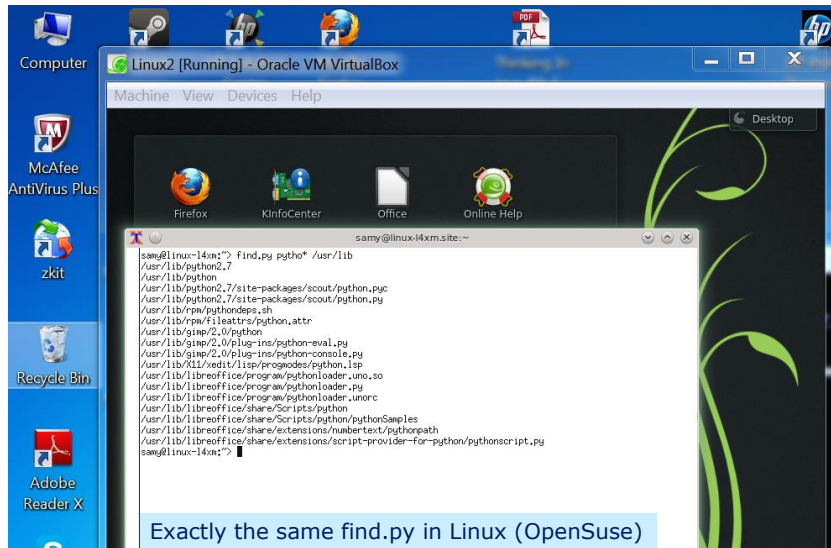
A View of Operating System Services

CLI – Command Line Interface (Windows)

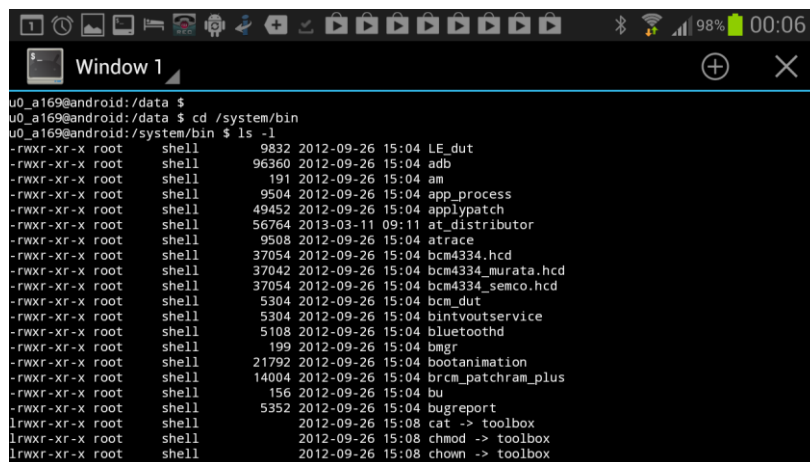
```
samy@linux-14xm:~$ ls -l /usr/bin | wc -l
2441
samy@linux-14xm:~$ ls -l /usr/bin/p*
-rwxr-xr-x 1 root root 38876 Feb 14 23:01 /usr/bin/pacat
-rwxr-xr-x 1 root root 579 Mar 1 12:00 /usr/bin/packagekit-bugreport.sh
-rwxr-xr-x 1 root root 3541 Feb 27 10:03 /usr/bin/package-manager
-rwxr-xr-x 1 root root 9972 Feb 27 10:03 /usr/bin/package-manager-su
-rwxr-xr-x 1 root root 13924 Feb 14 23:01 /usr/bin/pactl
-rwxr-xr-x 1 root root 55320 Feb 14 23:01 /usr/bin/pactl
-rwxr-xr-x 1 root root 2293 Feb 14 23:00 /usr/bin/padsp
-rwxr-xr-x 1 root root 22112 Jan 27 02:41 /usr/bin/paintopnm
-rwxr-xr-x 1 root root 13948 Jan 27 02:41 /usr/bin/pamddnoise
-rwxr-xr-x 1 root root 13900 Jan 27 02:41 /usr/bin/pamarith
-rwxr-xr-x 1 root root 9824 Jan 27 02:41 /usr/bin/pambackground
-rwxr-xr-x 1 root root 9768 Jan 27 02:41 /usr/bin/pambayer
-rwxr-xr-x 1 root root 9796 Jan 27 02:41 /usr/bin/pamchannel
-rwxr-xr-x 1 root root 19028 Jan 27 02:41 /usr/bin/pamcomp
-rwxr-xr-x 1 root root 13920 Jan 27 02:41 /usr/bin/pamcut
-rwxr-xr-x 1 root root 5688 Jan 27 02:41 /usr/bin/pamdeinterlace
-rwxr-xr-x 1 root root 9792 Jan 27 02:41 /usr/bin/pamdepth
-rwxr-xr-x 1 root root 9796 Jan 27 02:41 /usr/bin/pamdice
-rwxr-xr-x 1 root root 19012 Jan 27 02:41 /usr/bin/pamdiathermy
-rwxr-xr-x 1 root root 9776 Jan 27 02:41 /usr/bin/pamedge
-rwxr-xr-x 1 root root 5656 Jan 27 02:41 /usr/bin/pamendian
-rwxr-xr-x 1 root root 9800 Jan 27 02:41 /usr/bin/pamenlarge
-rwxr-xr-x 1 root root 9808 Jan 27 02:41 /usr/bin/pamexec
-rwxr-xr-x 1 root root 9788 Jan 27 02:41 /usr/bin/pamfile
```



Linux (guest OS) on top of Windows 7 (host OS)



Android Command Line Console (Galaxy Note 2)



Can be installed on Android device and used to practice shell



User Operating System Interface - CLI

- **Command Line Interface** (CLI) or **command interpreter** allows direct command entry
 - Sometimes implemented in **kernel** (shell internal commands)
 - Sometimes by **systems programs** (/usr/bin)
 - Sometimes multiple flavors implemented – **shells**
 - Primarily fetches a command from user and passes it to the OS kernel – immediately or through intermediate agents
 - ▶ Sometimes commands built-in, sometimes just names of programs
 - If the latter, adding new features doesn't require shell modification

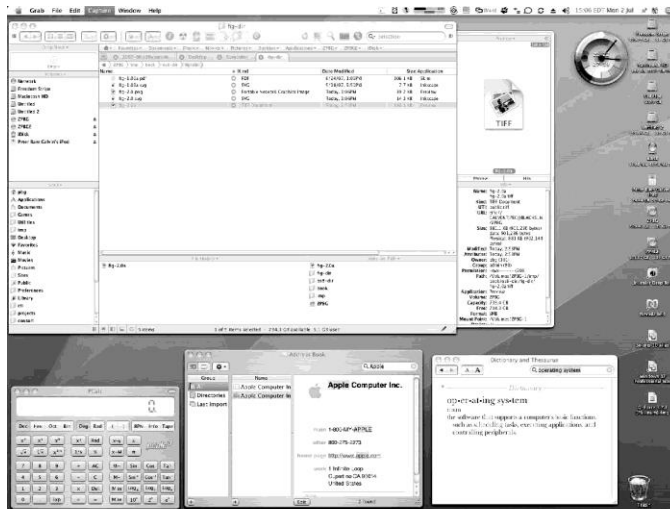


User Operating System Interface - GUI

- User-friendly **desktop** metaphor interface
 - Usually mouse, keyboard, and monitor
 - **Icons** represent files, programs, actions, etc
 - Various mouse buttons over objects in the interface cause various actions (provide information, options, execute function, open directory (known as a **folder**))
 - Invented at Xerox PARC (later copied by Steve Jobs to Macintosh, and Bill Gates to Windows)
- Many systems now include both CLI and GUI interfaces
 - Microsoft Windows is GUI with CLI “command” shell
 - Apple Mac OS X as “Aqua” GUI interface with UNIX kernel underneath and shells available
 - Solaris is CLI with optional GUI interfaces (Java Desktop, KDE)



The Mac OS X GUI



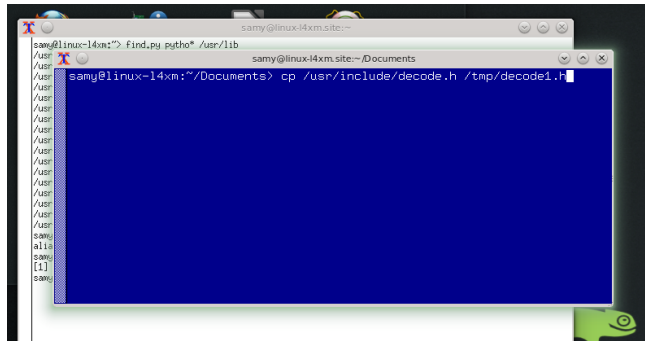
System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- 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)
 - Java API for the Java virtual machine (JVM)
- Why use APIs rather than system calls?

(Note that the system-call names used throughout this text are generic)



Example of Linux cp system program



[Click here to see how the cp system program Was implemented](#)



Example: Simple unsafe version of cp

```
#include <stdio.h>
#include <fcntl.h>
#include <syscall.h>
#define PERMS 0666 /* rwx for owner, group, others */
#define BUFSIZE 4096

void error(char *, ...) ;

/* copyfile.c: copy f1 to f2 */
main(int argc, char* argv[])
{
    int f1, f2, n;
    char buf[BUFSIZE];

    if (argc != 3)
        error("Usage: copyfile file1 file2");
    if ((f1 = open(argv[1], O_RDONLY, 0)) == -1)
        error("copyfile: can't open %s", argv[1]);
    if ((f2 = creat(argv[2], PERMS)) == -1)
        error("copyfile: can't create %s, mode %03o", argv[2], PERMS);
    while ((n = read(f1, buf, BUFSIZE)) > 0)
        if (write(f2, buf, n) != n)
            error("copyfile: write error on file %s", argv[2]) ;
    return 0;
}
```



Example of System Calls 2 (Linux)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <dirent.h>
```

```
int open(const char *pathname, int flags) ;
int open(const char *pathname, int flags, mode_t mode) ;
int close(int fd) ;
int creat(const char *pathname, mode_t mode) ;
ssize_t read(int fd, void* buf, size_t noct) ;
ssize_t write(int fd, const void* buf, size_t noct) ;
off_t lseek(int fd, off_t offset, int ref);
int stat(const char* path, struct stat* buf);
DIR* opendir(const char* pathname);
struct dirent* readdir(DIR* dp);
int closedir(DIR* dp);
```

- fd = file descriptor, noct = number of chars, dp = directory pointer, mode=permissions
- fopen() is not a system call! It is a wrapper on open() (or CreateFile())
- * There are hundreds of system calls in Linux!



Shorthands (Linux)

- fd = file descriptor
- noct = number of chars
- dp = directory pointer
- mode = permission bit string
- fopen() is not a system call!!!
 - It is a wrapper (API) on open() in Linux
 - or CreateFile() in Windows
- There are hundreds of system calls in Linux!
- Note:

```
/*linux/types.h"
typedef __kernel_size_t      size_t;
typedef __kernel_ssize_t     ssize_t;
```

```
/*asm/posix_types.h"
typedef unsigned int __kernel_size_t;
typedef int __kernel_ssize_t;
```

ssize_t is used for functions whose return value could either be a valid size, or a negative value to [indicate an error](#)



System Calls for Process Management

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

/* Process Management */

pid_t fork();
int execl(const char * path, const char * arg0, ..., NULL);
int execv(const char * path, char * argv[]);
int execlp(const char * filename, const char * arg0, ..., NULL);
int execvp(const char * filename, char * argv[]);
pid_t wait(int* pstatus);
pid_t waitpid(pid_t pid, int* pstatus, int opt);
pid_t getpid();
pid_t getppid();
void exit(int* status);
```



Process System Calls

```
pid_t fork();
    Creating a child process

int execl(const char * path, const char * arg0, ..., NULL);
int execlp(const char * filename, const char * arg0, ..., NULL);
    Variadic argument list
    The p stands for the shell PATH (so in the second form
    you do not have to supply a full path to the program)

int execv(const char * path, char * argv[]);
int execvp(const char * filename, char * argv[]);
    Arguments are passed by a single vector argv[]
    p stands for PATH

void exit(int* status);
    Terminate process
```



Examples of System Calls (Linux)

- In the `execlp` and `execvp` calls the executable file specified by the "filename" parameter is searched in the directories listed in the PATH environment variable
- All command line system programs are executed by the `exec*` system calls
- For example, "ls -l" is run via:

```
char *argv[] = {"ls", "-l", "/usr/src", (char*)NULL};  
execv("/usr/bin/ls", argv) ...
```



Python API to System Calls

- Python provides a higher layer API for running system programs via the `os` and `subprocess` module:

Windows:

```
os.system('c:/Windows/System32/mspaint.exe d:\\os.jpg')
```

```
argv = ['mspain.exe', 'd:/os.jpg']  
os.execv('c:/windows/system32/mspaint.exe', argv)
```

Linux:

```
os.system('/usr/bin/nano ~/proj5/apples.c')
```

```
argv = ['ls', '-l', '/usr/src']  
os.execv('/usr/bin/ls', argv)  
os.execl('/usr/bin/ls', 'ls', '-l', '/usr/src')  
# can also be done like this:  
os.execl('/usr/bin/ls', *argv)
```



System Call Numbers (Linux)

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

int main(void) {
    long ID1, ID2;

    /* direct system call */
    /* SYS_getpid (defined as syscall no. 20) */

    ID1 = syscall(20);
    printf ("syscall(SYS_getpid)=%ld\n", ID1);

    /* "libc" wraps syscall(20) as: getpid() */

    ID2 = getpid();
    printf ("getpid()=%ld\n", ID2);
    return(0);
}

/* Internally, syscall() is invoked by software interrupt 0x80 to transfer control to
the kernel. System call table is defined in Linux kernel source file
arch/i386/kernel/entry.S */
```



Linux System Calls List

- Full lists of system calls can be viewed in the following links:
- http://docs.cs.up.ac.za/programming/asm/derick_tut/syscalls.html
- <http://man7.org/linux/man-pages/man2/syscalls.2.html>
- <http://asm.sourceforge.net/syscall.html>
- Microsoft Windows does not have an officially declared boundary between API and system calls (try to Google for Windows system calls, and see what this means)
- The official Microsoft Windows term is Win32 API – a set of elementary interface functions to the operating system core (which is not documented and its source code is never released)



MS Windows CreateFile System Call

```
#include <windows.h>
#include <tchar.h>
#include <stdio.h>
#include <strsafe.h>

/* prototype */
HANDLE WINAPI CreateFile(
    _In_      LPCTSTR lpFileName,
    _In_      DWORD dwDesiredAccess,
    _In_      DWORD dwShareMode,
    _In_opt_  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    _In_      DWORD dwCreationDisposition,
    _In_      DWORD dwFlagsAndAttributes,
    _In_opt_  HANDLE hTemplateFile
);

Example:
    hFile = CreateFile(argv[1],           // name of the write
                       GENERIC_WRITE,    // open for writing
                       0,                 // do not share
                       NULL,              // default security
                       CREATE_NEW,        // create new file only
                       FILE_ATTRIBUTE_NORMAL, // normal file
                       NULL);             // no attr. template
```



Microsoft CreateProcess

- Windows combines **fork** and **exec** to a single system call:
CreateProcess (10 arguments)
- There is no Win32 API to fork a process
- No parent child relationship !
- There is no getppid API in Windows
- *Note* – special privileges are required to create a new process in Windows



MS Windows CreateProcess System Call

```
#include <windows.h>
#include <stdio.h>

int main( VOID )                                // Parent process
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi; /* like a PCB in Linux */
    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);
    ZeroMemory( &pi, sizeof(pi) );
    // Start the child process.
    if( !CreateProcess( NULL, // No module name (use command line).
        "C:\\WINDOWS\\system32\\mspaint.exe", // Command line: // Run mspaint.exe (child process)
        NULL, // Process handle not inheritable.
        NULL, // Thread handle not inheritable.
        FALSE, // Set handle inheritance to FALSE.
        0, // No creation flags.
        NULL, // Use parent's environment block.
        NULL, // Use parent's starting directory.
        &si, // Pointer to STARTUPINFO structure.
        &pi ) // Pointer to PROCESS_INFORMATION structure.
    )
    {
        printf( "CreateProcess failed (%d).\n", GetLastError() );
        return -1;
    }

    // Wait until child process exits -----> Software Interrupt ...
    WaitForSingleObject( pi.hProcess, INFINITE );
    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```



MS Windows CreateProcess System Call

The screenshot shows the Visual Studio 2012 IDE. The code from the previous slide is open in the editor. The Output window shows the following text:

```
Microsoft (R) C/C++ Optimizing Compiler Version 17.00.50727.1 for x86
Copyright (C) Microsoft Corporation. All rights reserved.

Microsoft (R) Incremental Linker Version 11.00.50727.1
Copyright (C) Microsoft Corporation. All rights reserved.

/out:wins.exe
C:\BRAUDE\OS\CODE\test>ls
wins.c
C:\BRAUDE\OS\CODE\test>gcc wins.c
C:\BRAUDE\OS\CODE\test>ls
wins.c wins.exe
C:\BRAUDE\OS\CODE\test>wins.exe
```

The program successfully compiles and runs, opening the Paint application.

The program easily compiles with cl



Platform Independent API (Python)

The following code works on Windows, Linux, Macintosh, Android

File management:

```
import os
f = open("c:/braude/os/db.csv")           # Windows file
f = open("/home/users/samyz/braude/os/db.csv") # Unix file: same API!
line = f.next()                           # Works the same in:
print line                                 # windows, unix,
                                           # macintosh, android

f.close()
os.listdir("c:/braude/os/db.csv")
os.listdir("/home/users/samyz/braude")
os.mkdir("c:/FSGEN/dir_52/dir_89")
os.rmdir("c:/FSGEN/dir_52")
os.rmdir("/usr/home/samyz/FSGEN/dir_52")
```



Platform Independent API (Python)

Process management:

The following code works on Windows, Linux, and Macintosh

```
import subprocess
# Starting the MS Windows paint program
p1 = subprocess.call("c:/windows/system32/mspaint.exe")
# Starting the Unix X calculator
p2 = subprocess.call("/usr/bin/X11/xcalc")
# Starting the MS Windows paint program in the background
p1 = subprocess.Popen("c:/windows/system32/mspaint.exe")
p1.pid
p1.communicate(input="Hello")
p3.send_signal(10)
p1.terminate()
```



Python/Linux API

Linux Process management:

```
# scan the list of all Linux system programs
from subprocess import Popen, PIPE
p = Popen(['ls', '-l', '/usr/bin'], stdout=PIPE)
total = 0
for line in p.stdout:
    fields = line.strip().split()
    if not len(fields) == 9:
        continue
    perms, nlinks, user, group, size, month, day, time, file = fields
    size = int(size)
    #print "System program file=%s, size=%d" % (file, size)
    print perms
    total += int(size)

print "System programs total size (mega bytes) = %.2f MB", float(total)/(2**20)
p.terminate()
```

```
samy@linux-l4xm:~$ ls -l /usr/bin/p*
-rwxr-xr-x 1 root root 38876 Feb 14 23:01 /usr/bin/pacat
-rwxr-xr-x 1 root root 579 Mar 1 12:00 /usr/bin/packagekit-bugreport.sh
-rwxr-xr-x 1 root root 3541 Feb 27 10:03 /usr/bin/package-manager
```

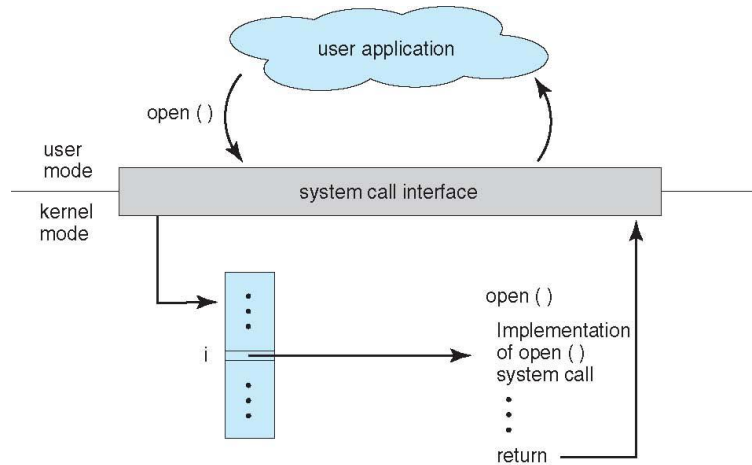


System Call Implementation

- Typically, a number associated with each system call
 - System-call interface maintains a table indexed according to these numbers
- The system call interface invokes intended system call in OS kernel and returns status of the system call and any return values
- The caller need know nothing about how the system call is implemented
 - Just needs to obey API and understand what OS will do as a result call
 - Most details of OS interface hidden from programmer by API
 - ▶ Managed by run-time support library (set of functions built into libraries included with compiler)

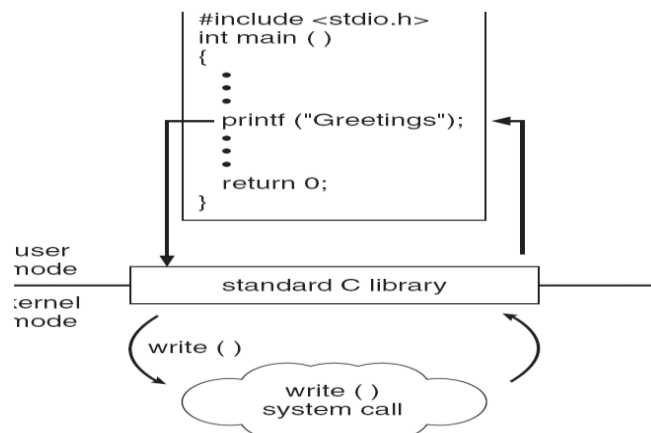


API – System Call – OS Relationship



Standard C Library Example

- Below is a C program invoking printf() standard library function
- printf()** is not a system call! It is an API to the **write()** system call



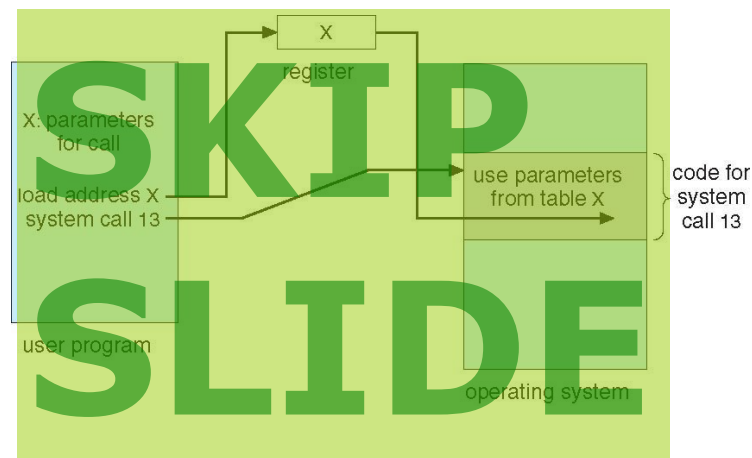


System Call Parameter Passing

- 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*
 - ▶ In some cases, may be more parameters than 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



Parameter Passing via Table





Types of System Calls

- **Process control**
 - end, abort
 - load, execute
 - create process, terminate process
 - get process attributes, set process attributes
 - wait for time
 - wait event, signal event
 - allocate and free memory
- **File management**
 - create file, delete file
 - open, close file
 - read, write, reposition
 - get and set file attributes



Types of System Calls (Cont.)

- **Device management**
 - request device, release device (e.g., diskonkey, camera, earphones)
 - read, write, reposition
 - get device attributes, set device attributes
 - logically attach or detach devices
- **Information maintenance**
 - get **time** or **date**, set **time** or **date**
 - get system data, set system data
 - get and set process, file, or device attributes
- **Communications**
 - create, open, close communication connections points (ports)
 - send, receive messages thru connection points (sockets)



Examples of Windows and Unix System Calls

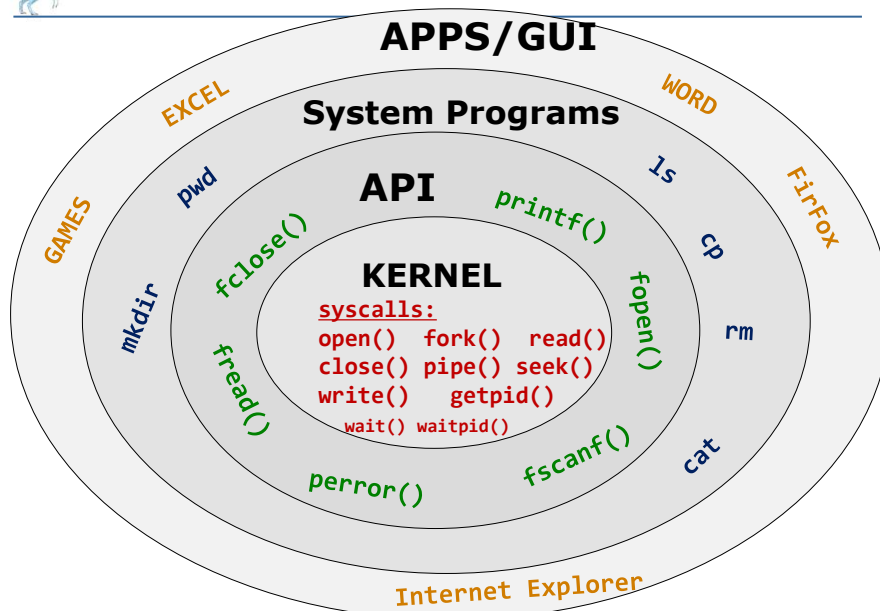
	Windows	Unix
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Manipulation	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Manipulation	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information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

ioctl - device-specific input/output operations

shmget - shared memory get



Famous Onion Diagram





Process Creation

- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- A process is identified and managed via a **process identifier (pid)**
- **Resource sharing**
 - Parent and children share all resources
 - Children share subset of parent's resources
 - Parent and child share no resources
- **Execution**
 - Parent and children execute concurrently
 - Parent can wait until children terminate

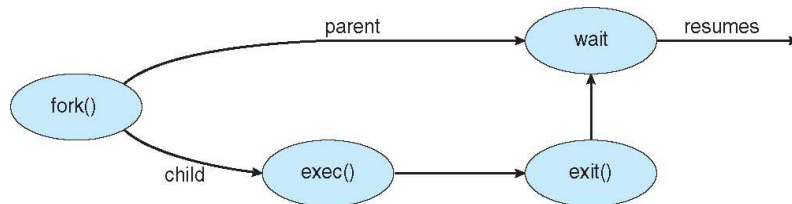


Process Creation (Cont.)

- Address space
 - Child duplicate of parent
 - Child has a program loaded into it
- 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



Process Creation



C Program Forking Separate Process

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

int main()
{
    pid_t pid;
    pid = fork();           /* fork a child process */
    if (pid < 0) {          /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    } else if (pid == 0) { /* child process ok! */
        execlp("/usr/bin/wc", "wc", "-l", "oliver_twist.txt", NULL);
    } else {               /* parent process */
        wait(NULL);        /* parent will wait for the child */
        printf ("Child Completed\n");
    }
    return 0;
}
```



fork() and exec()

Stage 1: Call fork()

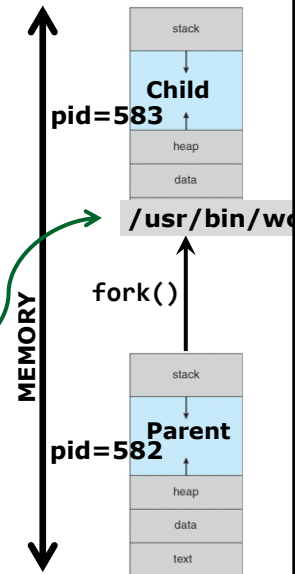
Child process is an identical copy of the parent process
This includes the PCB fields (open files, priority, ...)
The two PCB's are identical except for the pid field!
The child process gets its identity from a new pid.
Unix fork() resembles genetic cloning.
At this stage, both parent and child share the same data,

Heap, stack, open files etc ..., and therefore can have perfect cooperation.

However, in the two previous examples we decided to Replace the child with a completely new process and lose all shared data (nothing is left after execlp).

Stage 2: Call execlp() with program "/usr/bin/wc"

Child process text section is replaced with the machine code of the program "/usr/bin/wc".
The data, heap, and stack (and PCB field) are reset and No shared data is left anymore.
The only link is the parent has the child pid and can wait for him to complete and collect its exit code



Exercise: what will be printed here? (C++)

```
#include <unistd.h>
#include <iostream>
using namespace std;

int main()
{
    cout << "0. I am process " << getpid() << endl;
    fork();
    cout << "1. I am process " << getpid() << endl;
    fork();
    cout << "2. I am process " << getpid() << endl;
}

// How many processes are involved here ???
```



Exercise: what will be printed here? (C++)

```
#include <unistd.h>
#include <iostream>
using namespace std;

int main()
{
    cout << "0. I am process " << getpid() << endl;
    fork();
    cout << "1. I am process " << getpid() << endl;
    fork();
    cout << "2. I am process " << getpid() << endl;
    fork();
    cout << "3. I am process " << getpid() << endl;
}

// How many processes are involved here ???
```



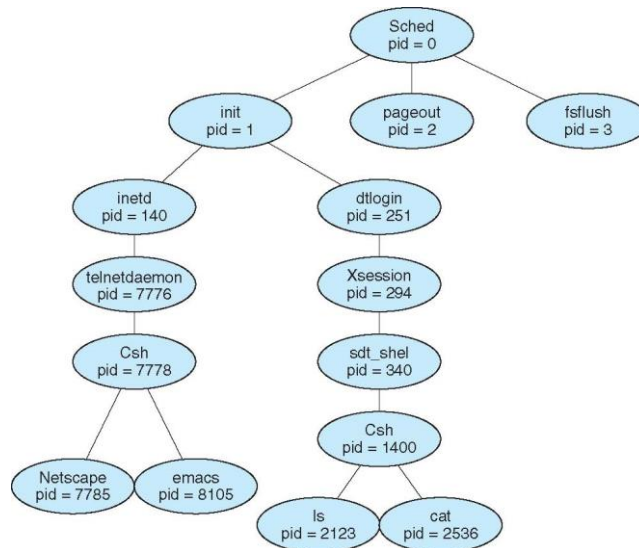
Exercise: who is my parent ?

```
#include <iostream>
using namespace std;
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    cout << "My process id is: " << getpid() << endl ;
    cout << "My parent process id is: " << getppid() << endl ;
    cout << "I am running from a Linux command line" << endl ;
    cout << "Please find the name of my parent?" << endl ;
    return 0;
}
```



A Tree of Processes on Solaris (SUN)



Windows 7 – Process Table

Image Name	PID	User Name	CPU	Memory (Private Working Set)	Description
AcroRd32.exe *32	6828	shedy	00	6,196 K	Adobe Reader
cmd.exe	6896	shedy	00	624 K	Windows Command Processor
zkl.exe *32	6904	shedy	00	18,700 K	TK DLL
SnippingTool.exe	6992	shedy	01	2,752 K	
System Idle Process	0	SYSTEM	98	24 K	Percentage of time the processor ...
System	4	SYSTEM	00	100 K	NT Kernel & System
smss.exe	384	SYSTEM	00	92 K	Windows Session Manager
nvSCPAPISvc.exe *32	428	SYSTEM	00	484 K	Stereo Vision Control Panel API S...
csrss.exe	672	SYSTEM	00	1,088 K	Client Server Runtime Process
wininit.exe	748	SYSTEM	00	236 K	Windows Start-Up Application
csrss.exe	772	SYSTEM	00	1,524 K	Client Server Runtime Process
winlogon.exe	812	SYSTEM	00	288 K	Windows Logon Application
services.exe	856	SYSTEM	00	3,832 K	Services and Controller app
lsass.exe	872	SYSTEM	00	3,476 K	Local Security Authority Process
lsm.exe	880	SYSTEM	00	760 K	Local Session Manager Service
svchost.exe	996	SYSTEM	00	1,796 K	Host Process for Windows Services
svchost.exe	1088	SYSTEM	00	109,872 K	Host Process for Windows Services
svchost.exe	1152	SYSTEM	00	13,136 K	Host Process for Windows Services
spoolsv.exe	1484	SYSTEM	00	1,484 K	Spooler SubSystem App
sqlwriter.exe	1496	SYSTEM	00	684 K	SQL Server VSS Writer - 64 Bit
armssvc.exe *32	1648	SYSTEM	00	176 K	Adobe Acrobat Update Service
ksfilter.exe	1712	SYSTEM	00	3,172 K	K9 Web Protection Filter
svchost.exe *32	1748	SYSTEM	00	1,208 K	Host Process for Windows Services
mfevtps.exe	1812	SYSTEM	00	1,920 K	McAfee Process Validation Service
c2c_service.exe *32	1940	SYSTEM	00	736 K	Skype C2C Service
WLIDSVC.EXE	1976	SYSTEM	00	1,512 K	Microsoft® Windows Live ID Serv...
svchost.exe	2092	SYSTEM	00	1,300 K	Host Process for Windows Services
mcsfield.exe	2148	SYSTEM	00	103,696 K	McAfee On-Access Scanner Service
mefire.exe	2180	SYSTEM	00	1,612 K	McAfee Core Firewall Service
WLIDSVC.EXE	2476	SYSTEM	00	172 K	Microsoft® Windows Live ID Serv...
SearchIndexer.exe	2904	SYSTEM	00	11,612 K	Microsoft Windows Search Indexer
McAfeeHost.exe	3168	SYSTEM	00	17,784 K	McAfee Service Host



Process Termination

- Process last statement asks the operating system to delete it (**exit**)
 - Output data from child to parent (via **wait**)
 - Process' resources are de-allocated by operating system
- Parent may terminate execution of children processes (**abort**)
 - Child has exceeded allocated resources
 - Task assigned to child is no longer required
 - If parent is exiting
 - ▶ Some operating systems do not allow child to continue if its parent terminates
 - All children terminated - **cascading termination**

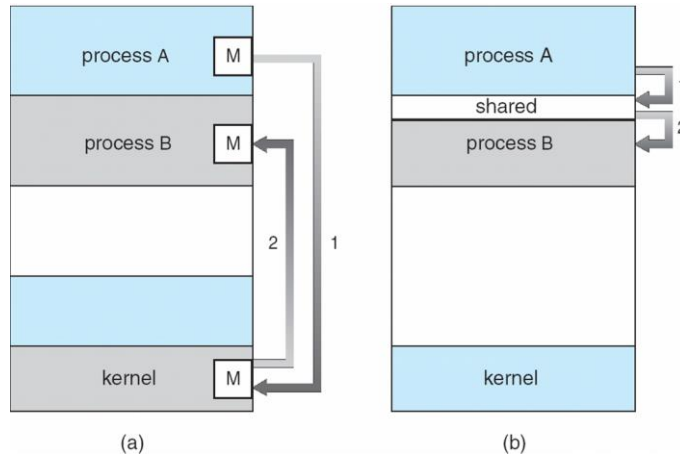


Inter-process Communication

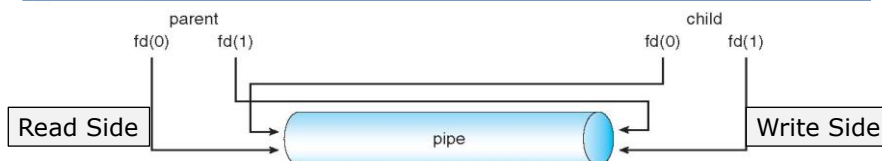
- Processes within a system may be **independent** or **cooperating**
- **Cooperating** process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
 - **Information sharing**
 - **Computation speedup**
 - **Modularity**
 - **Convenience**
- Cooperating processes need **inter-process communication (IPC)**
- Two models of IPC
 - **Shared memory**
 - **Message passing**



Communications Models



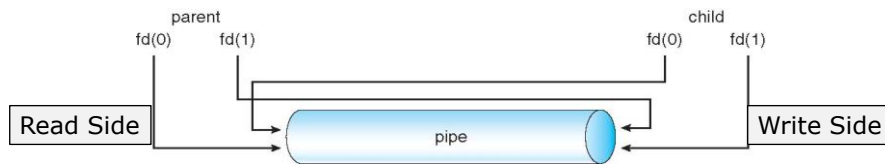
Unix pipe() system call



- **pipe()** is a system call which enables **IPC**
IPC = inter-process communication
- It opens a **pipe object**, which is a **buffer** in main memory that is treated as a "**virtual file**" (buffer is owned by the Operating System!)
- The pipe can be used by the creating process, as well as all its child processes - for **reading** and **writing**.
- One process can write to this "virtual file" (pipe) and another process can read from it
- **Blocking:** If a process tries to read before something is written to the pipe, the process is suspended until something is written
- The pipe system call assigns **two available positions** in the process's open file table and allocates them for the read and write ends of the pipe



Reasons to use buffering inside the kernel



- Sharing
 - Shared memory between two processes
- Caching
- Solve speed differences
 - Slow/Fast Writer
 - Slow/Fast Reader
- Solve user/kernel permission problems
 - Only the OS controls who can share pipe memory



Example: pipe() system call

```
// Simple Example
// Create a pipe with two ends for read and write
// If successful, pip[2] will be assigned two
// file descriptors

#include <unistd.h>
int fd[2];
int result;
result = pipe(fd);
if (result == -1) {
    perror("pipe failed!");
    exit(1);
}

write(fd[1], "Hi Mom!", 7);

// read from the pipe
char instr[20];
read(fd[0], instr, 7);
```



Unix fork() and pipe() system calls

```
// Simple Example: no closing of pipe
#include <unistd.h>

int main() {
    int pid, fd[2];
    char instring[20];

    pipe(fd);

    pid = fork();
    if (pid == 0) {
        // child: sends message to parent
        // send 7 characters in the string, including end-of-string
        write(fd[1], "Hi Mom!", 7);
    } else {
        // parent: receives message from child
        // read from the pipe
        read(fd[0], instring, 7);
    }
}
```



Python fork() and pipe(): High Level API

```
import os

rd, wd = os.pipe()

os.write(wd, 'Hello')

print os.read(rd, 5) # 'Hello' is printed.

os.close(rd)
os.close(wd)
```




Python fork() and pipe(): High level API

```
import os

rd, wd = os.pipe()
pid = os.fork() # creating a child process ! Unix only

if pid:
    # parent process
    os.write(wd, 'Hello') # Send 'Hello' to the child
    os.close(wd)
    os.waitpid(pid, 0)
else:
    # child process
    print os.read(rd, 4) # 'Hello' is printed by the child
```

Note:

- Use the subprocess module to make something that works In both platforms
- What will happen if the child tries to read more bytes?



System Programs

- System programs provide a convenient environment for program development and execution. They can be divided into:
 - File manipulation
 - Status information
 - File modification
 - Programming language support
 - Program loading and execution
 - Communications
 - Application programs
- Most users' view of the operation system is defined by system programs, not the actual system calls



File Manipulations in Python (1)

```
import os

# Get current directory
os.getcwd()
os.getcwdu() #unicode version (hebrew)

# Change current directory
os.chdir("c:/workspace")

# list directory
os.listdir("c:/workspace")

# Create a new directory
os.mkdir("c:/workspace/newdir")

# Recursively create directory at any depth
os.makedirs("c:/workspace/dir1/dir2/dir3")
```



File Manipulations in Python (2)

```
import os

# Rename file name
os.rename(file1, file2)

#remove file
os.remove ("c:/workspace/file.txt")

#remove empty directory
os.rmdir ("c:/workspace/newdir")
```



os.stat(file)

```
os.stat("c:/workspace/server.py")  
⇒ nt.stat_result(st_mode=33206, st_ino=0L, st_dev=0,  
  st_nlink=0, st_uid=0, st_gid=0, st_size=489L,  
  st_atime=1367180347L, st_mtime=1367180347L,  
  st_ctime=1367171683L)
```

st_mode	- protection bits,
st_ino	- inode number,
st_dev	- device,
st_nlink	- number of hard links,
st_uid	- user id of owner,
st_gid	- group id of owner,
st_size	- size of file, in bytes,
st_atime	- time of most recent access,
st_mtime	- time of most recent content modification,
st_ctime	- platform dependent time of most recent metadata change on Unix, or the time of creation on Windows



System Programs

- Provide a convenient environment for program development and execution
 - Some of them are simply user interfaces to system calls
 - Others are considerably more complex
- **File management**
Create, delete, copy, rename, print, dump, list, and generally manipulate files and directories
- **Status information**
 - Some ask the system for info - date, time, amount of available memory, disk space, number of users
 - Others provide detailed performance, logging, and debugging information
 - Typically, these programs format and print the output to the terminal or other output devices
 - Some systems implement a registry - used to store and retrieve configuration information



System Programs (Cont.)

■ File modification

- Text editors to create and modify files
- Special commands to search contents of files or perform transformations of the text

■ Programming-language support

Compilers, assemblers, debuggers and interpreters sometimes provided

■ Program loading and execution

Absolute loaders, re-locatable loaders, linkage editors, and overlay-loaders, debugging systems for higher-level and machine language

■ Communications

Provide the mechanism for creating virtual connections among processes, users, and computer systems

- Allow users to send messages to one another's screens, browse web pages, send electronic-mail messages, log in remotely, transfer files from one machine to another



Operating System Design and Implementation

■ NOT COVERED IN THIS COURSE