CPSC 457 Tutorial
Week 3

System calls

- System calls provide an interface to the services made available by an operating system
- Some of the low-level tasks are written in assembly language and the others are generally in C/C++
- Mostly, developers are not aware of the details of system calls, they use APIs(Application Programming Interface)

APIs

- A set of functions that are available to an application programmer with:
  - Function names
  - Parameter types
  - Return values
- Architectural differences in computers makes it harder to directly use system calls
- APIs also provide portability
- The most common APIs are:
  - Windows API for Windows
  - POSIX API for UNIX, Linux, Mac OS
  - Java API for Java Virtual Machine

APIs (cont'd)

- API calls are accessible via libraries, the one in C programming language is called **libc**

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

APIs (cont'd)

- Functions in APIs are actually wrappers for the real system calls

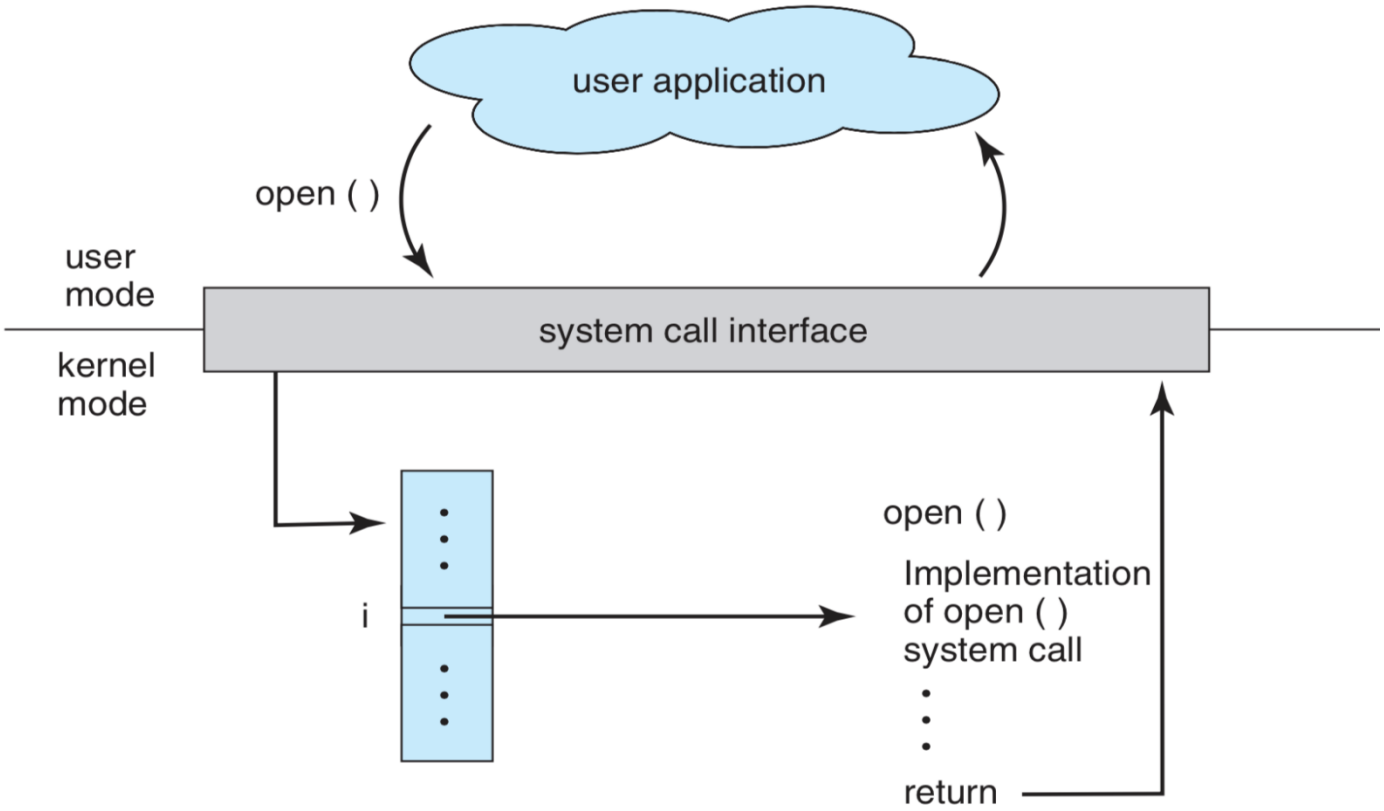


Figure 1: Handling **open()** function call

Taken from Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2012. Operating System Concepts (9th ed.)

System call examples

Type	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()

Trap

- Trap is a software generated interrupt caused either by an exception or a system call
- A trap usually results in switching to the kernel mode and OS performs some action before switching back to user mode
- Actions before trap:
  - Save the registers by pushing them onto the stack
  - Store any arguments that are to be passed(usually on registers)
- On trap execution:
  - Execute Trap after switching the mode

Blocking vs Non-blocking system calls

- In blocking system calls, user cannot do anything until the system call returns
- Input/output operations may result in blocking system calls
- Non-blocking system calls return almost immediately or the result is sent to the caller later via message/signal

Processes

- A process is a program in execution
- Process is more than the program code, it also includes:
  - **Program counter:** current activity in the program sequence
  - The content of the CPU's registers
  - **Program stack:** contains program data(e.g. function parameters, return address, local variables)
  - **Data section:** contains global variables
  - **Heap:** Memory that is allocated dynamically on run time

Processes (cont'd)

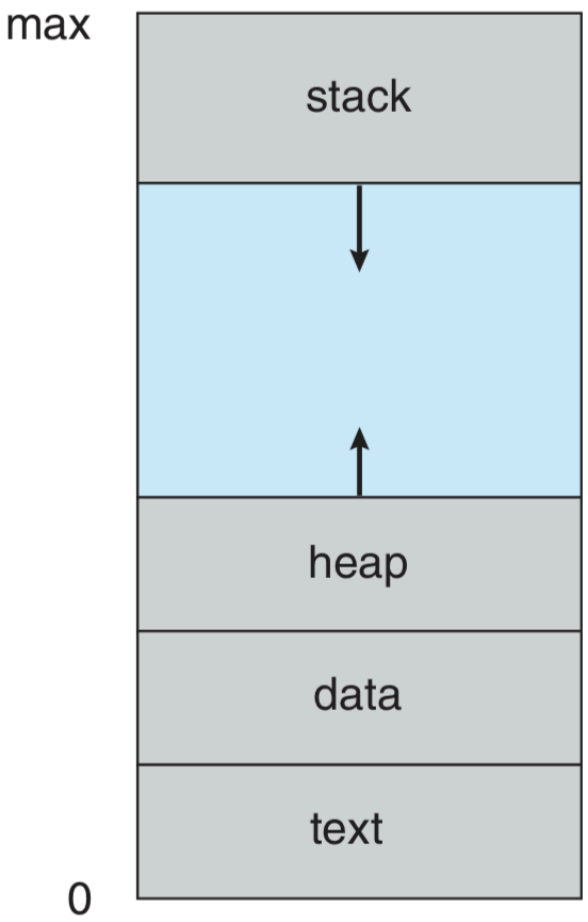
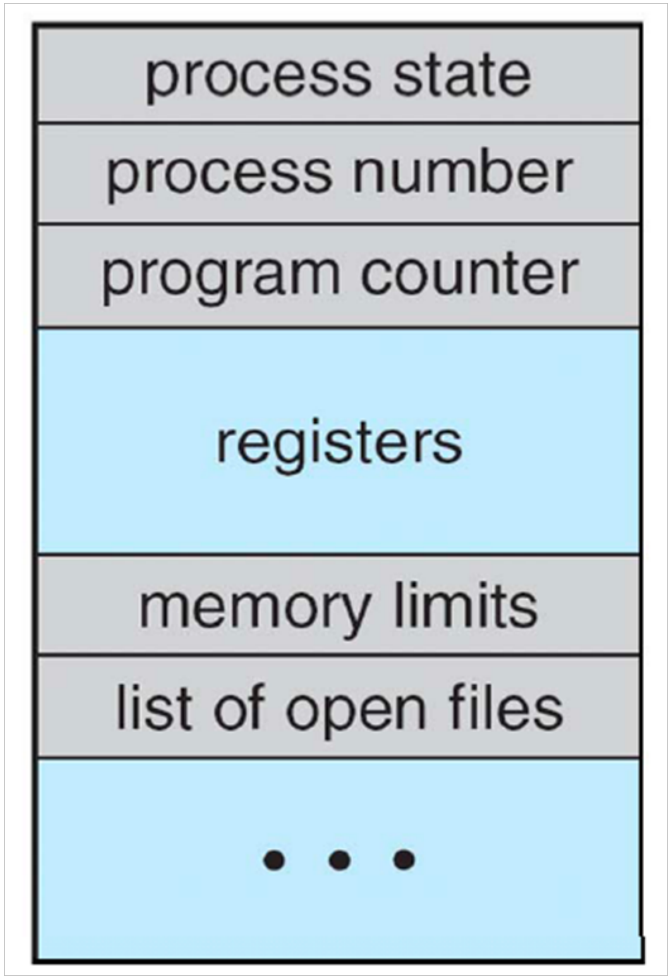


Figure 2: Process in memory

Taken from Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. 2012. Operating System Concepts (9th ed.)

Process control block

- PCB contains information about the process
- Also called **task control block**
- **Process state:** running, ready, waiting, …
- **Program counter:** current instruction location
- Register contents
- Memory management information
- I/O status information



fork()

- **fork()** duplicates the current process
- The only way to differentiate the child and parent process is looking to the return value of the function
  - Returns **0** in the child process
  - Returns **child process pid** in the parent
- Both parent and child continue execution after fork function call
- **fork()** is the only way to create a process in Unix-like operating systems

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

int main() {
    pid_t childPID;

    childPID = fork();
    if(childPID >= 0) { //Fork is successful
        if(childPID == 0) {
            //Child process block
        } else {
            //Parent process block
        }
    } else{
        //Fork failed
        return 1;
    }

    return 0;
}
```

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

int main() {
    printf("A\n");//Only first process exists

    fork();
    printf("B\n");//First process+first child

    fork();//Called by both the parent and child
    printf("C\n");//4 processes prints it

    return 0;
}
```

First process output: ABC
First child output: BC
Second child output: C
Third child output: C

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

int main() {
    for(int i=0; i<4; i++){
        fork();
    }
    printf("X\n");
    return 0;
}
```

- First process runs the loop 4 times → creates 4 children
- 1st child runs the loop 3 times
- 2nd child runs the loop 2 times
- 3rd child runs the loop once
- 1st child's 1st child runs the loop 2 times
- 1st child's 2nd child runs the loop once
- 2nd child's 1st child runs the loop once
- 1st child's 1st child's 1st child runs the loop once
- # of times the loop is run + 1(original process) = 16 (# of Xs in the output)

fork bomb

- Duplicates the process infinitely many times, crashes the system due to resource starvation

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

int main() {
    while(1){
        fork();
    }
    return 0;
}
```

Fork bomb in shell: `:(){ :|: & };:`

Creates a function with name `:`, without any parameters. In the definition, it sends its output to the same function again, which is run at the background. After function definition, it is called.(`;` is command separator)

exec function family

- After **fork()** system call, one of the two processes typically uses **exec()** to replace the current process image with a new one
- Possible functions are `execl`, `execv`, `execle`, `execve`, `execlp`, `execvp`
  - `man 3 exec`

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>

int main() {
    char *ls_args[] = {"ls", "-l", NULL};
    pid_t child_pid;

    child_pid = fork();
    if(child_pid == 0) {
        /* This is done by the child process. */
        execvp(ls_args[0], ls_args);

        /* If execv returns, it must have failed. */
        printf("Unknown command\n");
        exit(1);
    }
    else {
        /* This is run by the parent.  Wait for the child to
terminate. */
        wait(NULL);
        return 0;
    }
}
```

Processes in Linux

- View running processes
  - `ps` command (with multiple options)
    - Gives a snapshot of running processes
  - `top` command
    - Continuous statistics until the user types `q`
- Kill a running process
  - Get the process ID using `ps`
    - `kill pid` / `kill -9 pid`
  - `pkill [name]`
    - `name` is the full or partial process name

What is Shell Programming?

- Commonly known as Shell Scripts
  - A script is a collection of commands that are stored in a file
  - Mostly used for automate things
- Example:
  - A script displaying "Hello World" to the standard output

```
#!/bin/bash

# My first script

echo "Hello world"
```

A special line telling the OS to use /bin/bash to interpret this script.

Comment

Display "Hello world"

Why Shell Scripts?

- Too many commands to remember
- Automate things:
  - Scheduled task (e.g., backup at 3am everyday)
  - Certain sequence of commands are often being executed
- To run a Shell script:
  - Create a .sh file
  - Type your codes into the file
  - Assign execute permission to the file
  - Run the command in the file using
    - `$./file-name.sh`
    - `$./file-name.sh arg1 arg2 agr3`

File permissions

- Every file in linux has 3 attributes
  - Owner permission
  - Group permission
  - World permission
- To view the permissions of files and directories, use the following command
  - `ls -l`

Change permission

0	No permission	---
1	Execute permission	--x
2	Write permission	-w-
3	Execute and write permission	-wx
4	Read permission	r--
5	Read and execute permission	r-x
6	Read and write permission	rw-
7	All permissions	rwx

```
chmod 777 hello.sh
```

Variables

- By convention, environment variables and shell variables introduced by the OS are in capital letters
- It is better to use lower case letters for your own variables to prevent confusion
- Put no space between variable name and equality sign, and between equality sign and value

```
#!/usr/bin/env bash
name=”Emanuel Onu”
age=25
echo $name
echo $age
```

Special variables

```
#!/usr/bin/env bash
echo $0 #file name of the current script
echo $# #number of arguments given to the script
echo $n #the argument passed to the script in the
        #n^th location, e.g $1
```

Arrays

```
#!/usr/bin/env bash
name[0]="Victoria"
name[1]="Paul"
name[2]="Jennifer"
name[3]="Peter"

echo ${name[1]}
echo "${name[0]} ${name[1]}"
echo ${name[*]}
```

```
#!/bin/bash

numbers=(1 2 3 4 5)

echo ${numbers[@]}
echo ${numbers[*]}
```

Arithmetic Operations

```
#!/usr/bin/env bash

num1=3
num2=5

expr $num1 + $num2
expr $num1 - $num2

let total=$num1*$num2
total2=$(($num1/$num2))
echo $total
echo $total2
```

Decision Making

- Syntax:

```
if condition; then
  body
fi

if condition; then
  body
else
  body
fi
```

```
#!/bin/bash

name=“Emma“
if [ “$name” = “Emma” ];then
  echo “Emanuel”
elif [ “$name” = “Peter” ];then
  echo ”Peterson”
else
  echo “Unknown name”
fi
```

Numeric/String Comparisons

Numeric comparisons

`$a -lt $b` → a<b
`$a -gt $b` → a>b
`$a -le $b` → a<=b
`$a -ge $b` → a>=b
`$a -eq $b` → a is equal to b
`$a -ne $b` → a is not equal to b

String comparisons

`"$a" = "$b"` → a is same as b
`"$a" == "$b"` → a is same as b
`"$a" != "$b"` → a and b are different
`-z "$a"` → a is empty

Loops

- For loop syntax:

```
for arg in [list];
do
  body
done
```

```
#!/bin/bash

numbers=(1 2 3 4 5)
for k in ${numbers[@]};do
  echo “Step $k”
done
```

- While loop syntax:

```
while [condition];
do
  body
done
```

```
#!/bin/bash

count=4
while [ $count –gt 0 ]; do
  echo “Count is $count”
  let count-=1
done
```

Functions

- Syntax

```
function f_name {
  body
}

function f_name(){
  body
}
```

```
function print_o {
  echo ”Hello world”
}

function addition(){
  echo “$(($1+$2))”
}

print_o
addition 3 5
```

Shell Scripts Tutorials

- Writing Shell Scripts http://linuxcommand.org/lc3_writing_shell_scripts.php
- Advanced Bash-Scripting Guide http://tldp.org/LDP/abs/html/
- Linux Shell Scripting Tutorial v1.05r3 A Beginner's handbook http://www.freeos.com/guides/lsst/
- The Beginner's Guide to Shell Scripting: The Basics https://www.howtogeek.com/67469/the-beginners-guide-to-shell-scripting-the-basics/

Some UNIX utilities

- **head**
  - Reads some few lines from the given text writes them to standard output (10 lines by default)
    - `head [file_name]`
    - `head -n5 [file_name]`
- **sort**
  - Sorts lines of text files (alphabetically, in reverse order, by number, etc.)
    - `sort [file_name]`
    - `sort -r [file_name]`

Some UNIX utilities

- **find**
  - It is used to find files and directories, perform operations on them
    - `find [path] –name [file_name]`
    - `find [path] –name [file_name] --delete`
    - `find –type d –name [dir_name]`
- **awk**
  - Data manipulation and report generation
    - `awk ‘{print} ’[file_name]`
    - `awk ‘[pattern] print’ [file_name]`
    - `awk ‘print $1,$3’ [file_name]`
    - `awk -v FS=, '{print $1, $3}' [file_name]`

system() vs popen()

- `system(const char *command)` passes the command to the command processor
- It is an easy way to process commands when control over input/output file streams is not necessary
- `popen(const char *command, const char *mode)` also executes shell commands
- In addition, it creates a pipe between the calling program and executed program

```
#include <iostream>
#include <cstdlib> //Needed for system()

using namespace std;

int main() {
    cout<<"Hi!"<<endl;

    //The commands are platform specific!
    system("ps aux | grep firefox");
    system("ls -l");

    cout<<"Bye!"<<endl;
    return 0;
}
```

```
#include <iostream>
#include <cstdio>
#define MAX_LEN 100

using namespace std;
int main(){
    FILE *fp;
    char buffer[MAX_LEN];

    //Gives the first 10 lines of the source code
    fp = popen("cat popen.cpp | head -n 10", "r");

    if(!fp) {
        //Handle error
    }
    while(fgets(buffer, MAX_LEN, fp)) {
        printf("%s", buffer);
    }

    pclose(fp);
    return 0;
}
```