Operating System

System Calls

System Call

- A **system call** is the programmatic way in which a computer program requests a service from the kernel of the operating system on which it is executed.
- Example: fork, exec etc.

Services Provided by System Calls

- Process creation and management
- Main memory management
- File Access, Directory and File system management
- Device handling(I/O)
- Protection
- Networking etc.

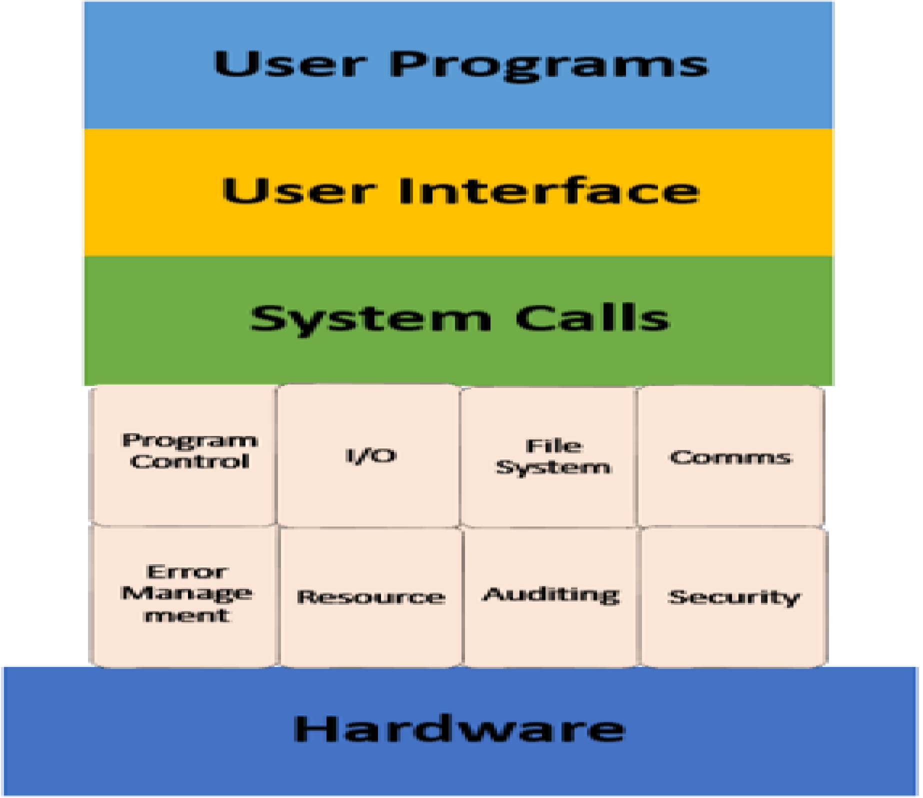
Types of System Calls

- **Process control:** end, abort, create, terminate, allocate and free memory.
- **File management:** create, open, close, delete, read file etc.
- Device management
- Information maintenance
- Communication

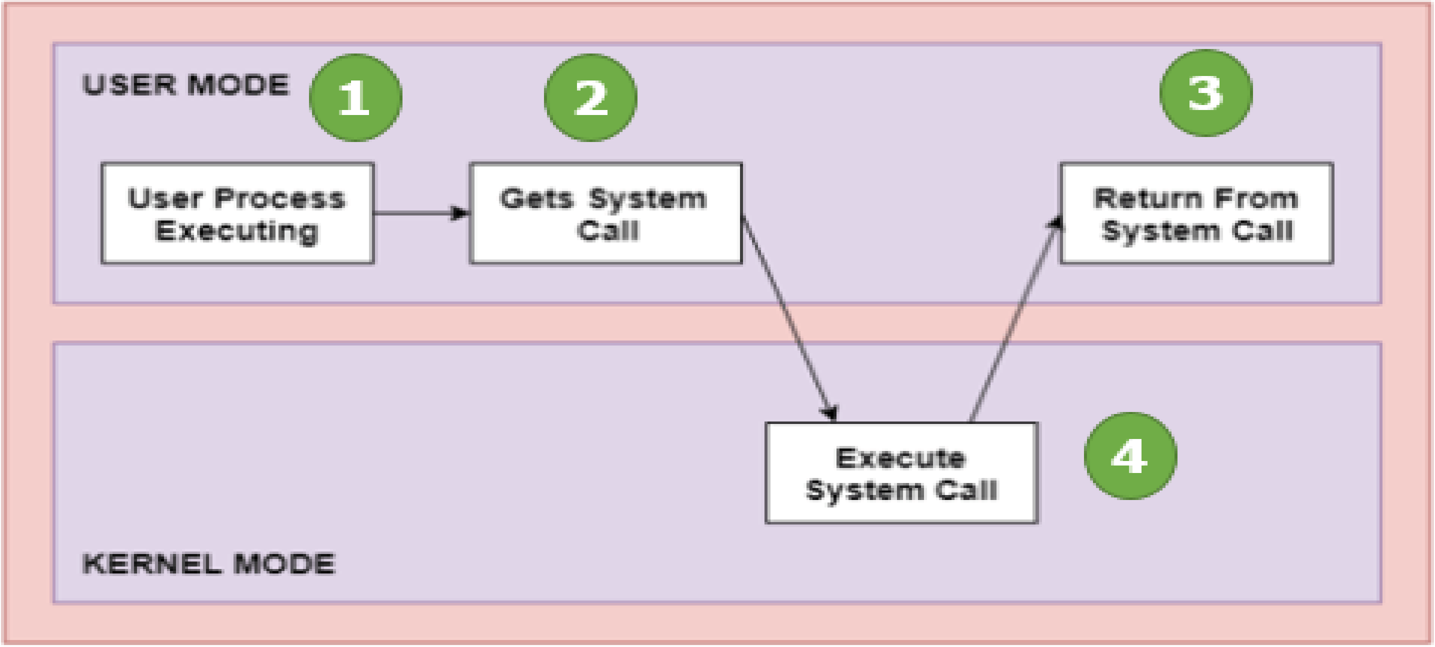
Examples:

	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() InitlializeSecurityDescriptor() SetSecurityDescriptorGroup()	chmod() umask() chown()

How does it work(1)



How does it work(2)



Operating System Services

- One set of operating-system services provides functions that are helpful to the user:
  - 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(2)

- Another set of OS functions exists for 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
    - Many types of resources - Some (such as CPU cycles, main memory, and file storage) may have special allocation code, others (such as I/O devices) may have general request and release code
  - **Accounting -** To keep track of which users use how much and what kinds of computer resources

Operating System Services(3)

- **Protection and security -** The 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** involves ensuring that all access to system resources is controlled
  - **Security** of the system from outsiders requires user authentication, extends to defending external 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.

What API does the OS provide to user programs?

- API = Application Programming Interface
  = functions available to write user programs
- API provided by OS is a set of “system calls”
  – System call is a function call into OS code that runs at a higher privilege level of the CPU
  – Sensitive operations (e.g., access to hardware) are allowed only at a higher privilege level
  – Some “blocking” system calls cause the process to be blocked and descheduled (e.g., read from disk)

So, should we rewrite programs for each OS?

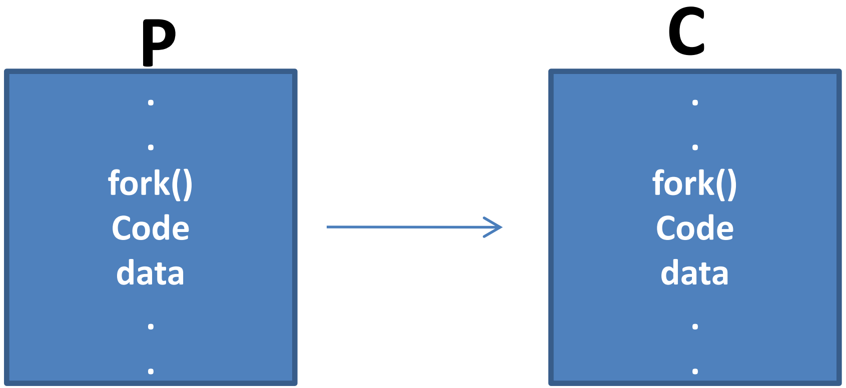
- POSIX API: a standard set of system calls that an OS must implement
  – Programs written to the POSIX API can run on any POSIX compliant OS
  – Most modern OSes are POSIX compliant
  – Ensures program portability
- Program language libraries hide the details of invoking system calls
  – The printf function in the C library calls the write system call to write to screen
  – User programs usually do not need to worry about invoking system calls

Process related system calls (in Unix)

- fork() creates a new child process
– All processes are created by forking from a parent
– The init process is ancestor of all processes
- exec() makes a process execute a given executable
- exit() terminates a process
- wait() causes a parent to block until child terminates
- Many variants exist of the above system calls with different arguments

What happens during a fork?

- A new process is created by making a copy of parent's memory image
- The new process is added to the OS process list and scheduled
- Parent and child start execution just after fork (with different return values)
- Parent and child execute and modify the memory data independently



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

int main(int argc, char *argv[]) {
 printf("hello world (pid:%d)\n", (int) getpid());
 int rc = fork();
 if (rc < 0) { //fork fails
   fprintf(stderr, "fork failed\n");
 } else if (rc == 0) { //new process (children)
   printf("hello, I am child (pid:%d)\n", (int) getpid());
 } else {  // parent(the main process)
   printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
 }
 return 0;
}
```

```
hello world (pid:29146)
hello, I am parent of 29147 (pid:29146)
hello, I am child (pid:29147)
```

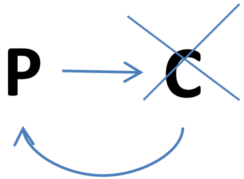
Waiting for children to die…

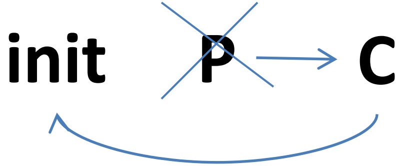
- Process termination scenarios

– By calling exit() (exit is called automatically when end of main is reached)

– OS terminates a misbehaving process

- Terminated process exists as a zombie
- When a parent calls wait(), zombie child is cleaned up or “reaped”
- wait() blocks in parent until child terminates (non-blocking ways to invoke wait exist)
- What if parent terminates before child? init process adopts orphans and reaps them





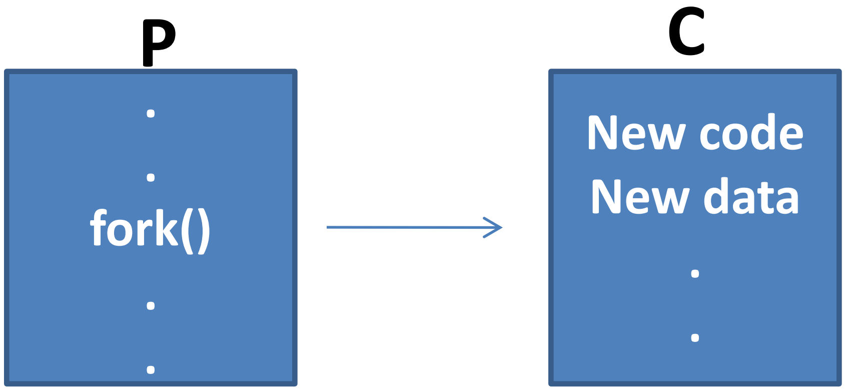
```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>
int main(int argc, char *argv[]) {
 printf("hello world (pid:%d)\n", (int) getpid());
 int rc = fork();
 if (rc < 0) {
   fprintf(stderr, "fork failed\n");
   exit(1);
 } else if (rc == 0) {
   printf("hello, I am child (pid:%d)\n", (int) getpid());
 } else {
   int wc = wait(NULL);
   printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc, (int) getpid());
 }
 return 0;
}
```

Some facts…

- 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 waits until children terminate

What happens during exec?

- After fork, parent and child are running same code
  – Not too useful!
- A process can run exec() to load another executable to its memory image
  – So, a child can run a different program from parent
- Variants of exec(), e.g., to pass command line arguments to new executable



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

int main(int argc, char *argv[]) {
 printf("hello world (pid:%d)\n", (int) getpid());
 int rc = fork();
 if (rc < 0) {
   fprintf(stderr, "fork failed\n");
   exit(1);
 } else if (rc == 0) {
   printf("hello, I am child (pid:%d)\n", (int) getpid());
   char *myargs[3];
   myargs[0] = strdup("wc");
   myargs[1] = strdup("p3.c");
   myargs[2] = NULL;
   execvp(myargs[0], myargs);
   printf("this shouldn't print out");
 } else {
   int wc = wait(NULL);
   printf("hello, I am parent of %d (wc:%d) (pid:%d)\n", rc, wc, (int) getpid());
 }
 return 0;
}
```

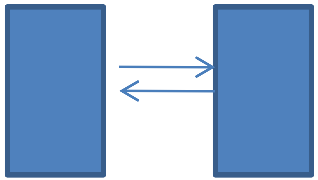
Case study: How does a shell work?

- In a basic OS, the init process is created after initialization of hardware
- The init process spawns a shell like bash
- Shell reads user command, forks a child, execs the command executable, waits for it to finish, and reads next command
- Common commands like ls are all executables that are simply exec'ed by the shell

prompt>ls

a.txt b.txt c.txt

init

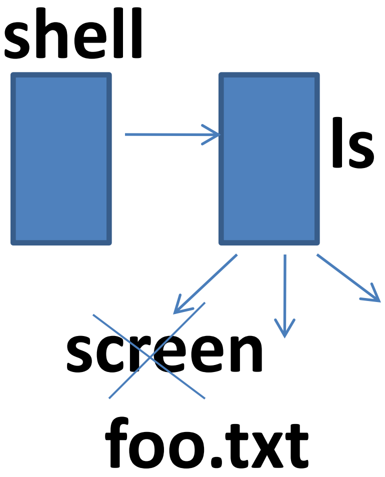


fork()

exec(ls)

More funky things about the shell

- Shell can manipulate the child in strange ways
- Suppose you want to redirect output from a command to a file
- prompt>ls > foo.txt
- Shell spawns a child, rewires its standard output to a file, then calls exec on the child



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

int main(int argc, char *argv[]) {
 printf("hello world (pid:%d)\n", (int) getpid());
 int rc = fork();
 if (rc < 0) {
   fprintf(stderr, "fork failed\n");
   exit(1);
 } else if (rc == 0) {
   close(STDOUT_FILENO); //close standard output
   open("./p4.output", O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU); // open a file for showing output
   char *myargs[3];
   myargs[0] = strdup("wc");
   myargs[1] = strdup("p4.c");
   myargs[2] = NULL;
   execvp(myargs[0], myargs);
 } else {
   int wc = wait(NULL);
 }
 return 0;
}
```

Thank You!!!!!