3. System Calls

fork, copy-on-write, exec, wait, signal, limited direct execution, instructions/system calls/commands, kernel mode, user mode, mode switch/transition, preemptive multitasking, trap table (interrupt vector table), sync/async interrupts, software/exception/hardware interrupt, timer interrupt, Process ID (PID), (call) stack, kernel stack, privileged operation/instruction, cooperative vs preemptive (timer interrupt)

3.1 System Calls

Using `fork()` and `exec()` system calls to create a new process and run a program in it. The `wait()` system call is used to wait for the child process to complete.

3.1.1 fork()

> Creates a new process

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

int main(int argc, char *argv[]) {
  printf("hello (pid:%d)\n", (int) getpid());
  int rc = fork();

  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child (new process)
    printf("child (pid:%d)\n", (int) getpid());
  } else {
    // parent goes down this path (main)
    printf("parent of %d (pid:%d)\n",
    rc, (int) getpid());
  }
  return 0;
 }
```

```
prompt> ./p1
hello (pid:29146)
parent of 29147 (pid:29146)
child (pid:29147)
prompt>
```

Note that the parent receives the child's process ID from the `fork()` system call. The child receives `0` from the `fork()` system call, hence printing "child...". One can then use this to write code that is executed by the child process.

`fork()` uses **copy-on-write** to avoid allocating memory unnecessarily. Copy-on-write means that the new process can just keep using the memory of the parent process as long as both processes just issues reads. As soon as one of them issues a write, then the two processes need their own private copy.

3.1.2 wait()

> Waits for the child process to complete and execute remaining code of the parent process

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

int main(int argc, char *argv[]) {
  printf("hello (pid:%d)\n", (int) getpid());
  int rc = fork();

  if (rc < 0) { // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("child (pid:%d)\n", (int) getpid());
  } else { // parent goes down this path
    int rc_wait = wait(NULL);
    printf("parent of %d (rc_wait:%d) (pid:%d)\n",
    rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

```
prompt> ./p2 hello (pid:29266) child (pid:29267) parent of 29267 (rc_wait:29267)
(pid:29266) prompt>
```

That way we can always ensure that the parent process waits for the child process to complete.

3.1.3 exec()

> Replaces the current process with a new program

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

int main(int argc, char *argv[]) {
```

```
    printf("hello (pid:%d)\n", (int) getpid());
    int rc = fork();

    if (rc < 0) { // fork failed; exit
      fprintf(stderr, "fork failed\n");
      exit(1);
    } else if (rc == 0) { // child (new process)
      printf("child (pid:%d)\n", (int) getpid());
      char *myargs[3];
      myargs[0] = strdup("wc"); // program: "wc"
      myargs[1] = strdup("p3.c"); // arg: input file
      myargs[2] = NULL; // mark end of array
      execvp(myargs[0], myargs); // runs word count
      printf("this shouldn't print out");
    } else { // parent goes down this path
      int rc_wait = wait(NULL);
      printf("parent of %d (rc_wait:%d) (pid:%d)\n",
      rc, rc_wait, (int) getpid());
    }
    return 0;
  }
```

```
prompt> ./p3 hello (pid:29383) child (pid:29384) 29 107 1030 p3.c parent of
29384 (rc_wait:29384) (pid:29383) prompt>
```

Here, it **loads** code (and static data) from the given executable (`wc`) and **overwrites** the current code segment (and current static data) with it; essentially re-initializing the process with the new program.

3.1.4 Why fork-exec?

```
#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[]) {
int rc = fork();
 if (rc < 0) {
 // fork failed
 fprintf(stderr, "fork failed\n");
 exit(1);
 } else if (rc == 0) {
 // child: redirect standard output to a file
 close(STDOUT_FILENO);
 open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC,
 S_IRWXU);
 // now exec "wc"...
 char *myargs[3];
 myargs[0] = strdup("wc"); // program: wc
```

```
    myargs[1] = strdup("p4.c"); // arg: file to count
    myargs[2] = NULL; // mark end of array
    execvp(myargs[0], myargs); // runs word count
    } else {
    // parent goes down this path (main)
    int rc_wait = wait(NULL);
    }
    return 0;
```

The `fork()/exec()` combination is a powerful way to create and manipulate processes

3.1.5 Process Control and Users

The `kill()` system call is used to send **signals** to a process (e.g., pause, die, etc.)

users can generally only control their own processes

--

3.2 Process Execution

time-sharing created performance issues. The second is *control*: how can we ensure that one process doesn't interfere with another?

3.2.1 Limited Direct Execution

To make a program run as fast as possible, **limited direct execution** is used. This means that the CPU runs the program's instructions directly, without any intervention from the operating system.

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

Figure 6.1: **Direct Execution Protocol (Without Limits)**

This approach (which is not *limited*) raises a few problems:

- **security**: a user program could issue a privileged operation

- **control**: how can we ensure that one process doesn’t interfere with another?

Trap instructions are what's really "inside" a system call.

We introduce the concept of **kernel mode** and **user mode**:

- **user mode**: code that runs in user mode is restricted; e.g. the process cannot issue I/O requests: doing so would raise an exception and the OS would kill it.
- **kernel mode**: code that runs in kernel mode can do anything it wants.

But how can a process on user mode perform privileged operations?

- by using **system calls**

1. The program must execute a special **trap** instruction to switch to kernel mode.
2. When finished, the OS calls a **return-from-trap** instruction to switch back to user mode.

The hardware needs to be a bit careful; hence pushes the program counter, flags and other registers onto the **kernel stack**; then the "return-from-trap" instruction" will pop these values off the stack and restore the user process.

How does the calling process know which code to run inside the OS?

-> The kernel does so by using a **trap table** (also known as an interrupt vector table).

- When booting up, the kernel tells the hardware what code to run when certain exceptional events occur.
- That way the hardware knows the locations of these **trap handlers** with special instructions.

OS @ boot (kernel mode)	Hardware	
initialize trap table		
	remember address of... syscall handler	

OS @ run (kernel mode)	Hardware	Program (user mode)
Create entry for process list		
Allocate memory for program		
Load program into memory		
Setup user stack with argv		
Fill kernel stack with reg/PC		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to main	
		Run main()
		...
		Call system call
		trap into OS
	save regs (to kernel stack)	
	move to kernel mode	
	jump to trap handler	
Handle trap		
Do work of syscall		
return-from-trap		
	restore regs (from kernel stack)	
	move to user mode	
	jump to PC after trap	
		...
		return from main
		trap (via `exit()`)
Free memory of process		
Remove from process list		

Figure 6.2: **Limited Direct Execution Protocol**

To specify exact system call, a **system call number** is used, such that the OS can verify if it's valid or not (a form of *protection*).

3.2.2 Switching Between Processes

When an application is, e.g., divides by zero, or tries to access undefined memory, it will **generate an trap** to the OS; then the OS will have control and will likely kill the offending process.

That is the **passive approach** to switching between processes.

However, what if the process are not cooperating? -> By using **timer interrupts**.

> The timer device is programmed to raise an interrupt every ms and a pre-configured interrupt handler in the OS is ran. Thus the OS has regained control.

In addition to creating trap table entries for system calls, the OS also **starts a timer** which is a privileged operation. Then it is safe for the OS to take control eventually (periodically) and switch to another process.

Context Switch

If the decision is made to switch to another process (made by the **scheduler**), then the OS executes low-level code to perform a **context switch**.

1. The OS saves the current process's registers and memory mappings.
2. The OS loads the new process's registers and memory mappings.
3. The OS runs the new process.

To save the context of the currently-running program, the OS will execute assembly code to save the general purpose registers, PC, and the kernel stack pointer and then restore said registers, PC and stack pointer for the new process.

OS @ boot (kernel mode)	Hardware
initialize trap table	
	remember addresses of...
	syscall handler
	timer handler
start interrupt timer	
	start timer
	interrupt CPU in X ms

OS @ run (kernel mode)	Hardware	Program (user mode)
		Process A
		...
	timer interrupt	
	save regs(A) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
Handle the trap		
Call `switch()` routine		
save regs(A) → proc_t(A)		
restore regs(B) ← proc_t(B)		
switch to k-stack(B)		
return-from-trap (into B)		
	restore regs(B) ← k-stack(B)	
	move to user mode	
	jump to B's PC	
		Process B
		...

Figure 6.3: **Limited Direct Execution Protocol (Timer Interrupt)**

3.3 Review questions and problems

1. What is the purpose of system calls?

- Abstracts the hardware and provides a way for user programs to interact with the OS.
- Ensures that user programs cannot do anything they want (security).
- Ensures that user programs cannot interfere with each other (control).

2. Briefly describe the difference between asynchronous and asynchronous interrupts.

- Sync: For instance when the user program divides by zero, and exception is raised and the OS takes control.
- Async: For example when an I/O operation is completed, an interrupt is raised and the OS takes control.

3. Codes:

Normal fork

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

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int)getpid());
  int x = 100;
  int rc = fork();
  printf("rc is %d\n", rc);
  if (rc < 0) {
    // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc) {
    // printf("rc is %d\n", rc);

    printf("ddBefore x is %d child\n", x);
    x = 200;
    // child (new process)
    printf("hello, I am child (pid:%d)\n", (int)getpid());
    printf("after x is %d child\n", x);

  } else {
    wait(NULL);
    printf("Before x is %d parent\n", x);
    x = 300;
    // parent goes down this path (original process)
    printf("hello, I am parent of %d (pid:%d)\n", rc, (int)getpid());
    printf("after x is %d parent\n", x);
  }
  printf("hell i m unkown %d\n", (int)getpid());
  return 0;
}

// Answer: They will have their own cpoy of the variables. The child will
have a
// copy of the parent's variables. They will not share the same memory
space.
```

Run other process

```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) {
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
        // child (new process)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (word count)
        myargs[1] = strdup("p3.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
        printf("this shouldn't print out");
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
        printf("hello, I am parent of %d (wc:%d) (pid:%d)\n",
        rc, wc, (int) getpid());
    }
    return 0;
}
```

Redirect output to a file

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

int
main(int argc, char *argv[])
{
    int rc = fork();
    if (rc < 0) {
```

```
        // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    } else if (rc == 0) {
  // child: redirect standard output to a file
  close(STDOUT_FILENO);
  open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

  // now exec "wc"...
        char *myargs[3];
        myargs[0] = strdup("wc");   // program: "wc" (wor count)
        myargs[1] = strdup("p4.c"); // argument: file to count
        myargs[2] = NULL;           // marks end of array
        execvp(myargs[0], myargs);  // runs word count
    } else {
        // parent goes down this path (original process)
        int wc = wait(NULL);
  assert(wc >= 0);
    }
    return 0;
}
```

Summary

ASIDE: KEY PROCESS API TERMS

- Each process has a name; in most systems, that name is a number known as a **process ID** (**PID**).
- The **fork()** system call is used in UNIX systems to create a new process. The creator is called the **parent**; the newly created process is called the **child**. As sometimes occurs in real life [J16], the child process is a nearly identical copy of the parent.
- The **wait()** system call allows a parent to wait for its child to complete execution.
- The **exec()** family of system calls allows a child to break free from its similarity to its parent and execute an entirely new program.
- A UNIX **shell** commonly uses `fork()`, `wait()`, and `exec()` to launch user commands; the separation of fork and exec enables features like **input/output redirection**, **pipes**, and other cool features, all without changing anything about the programs being run.
- Process control is available in the form of **signals**, which can cause jobs to stop, continue, or even terminate.
- Which processes can be controlled by a particular person is encapsulated in the notion of a **user**; the operating system allows multiple users onto the system, and ensures users can only control their own processes.
- A **superuser** can control all processes (and indeed do many other things); this role should be assumed infrequently and with caution for security reasons.