How user-level programs interact with the OS?

- **Invoke syscalls**
  - Send/receive data
  - Create/delete files
  - Execute/kill other programs

- **Generate exceptions** (to be handled by OS)
  - Touch illegal memory
  - Divide by zero

- **Get interrupted** by OS
  - OS preempts a program to execute other programs
  - OS does “upcalls” to user-programs via signals
Control Flow

• A CPU core reads and executes a sequence of instructions, one at a time
  – This sequence is the CPU’s control flow

Physical control flow

<startup>

inst_1
inst_2
inst_3
...
inst_n
<shutdown>
Altering the Control Flow

- Up to now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return

- “exceptional control flow” -- react to changes in system state
  - data arrives from a disk or a network adapter
  - instruction divides by zero
  - user hits Ctrl-C at the keyboard
  - System timer expires
Types of Exceptions

- **Asynchronous** *(interrupts)*: caused by events external to CPU
  - hitting Ctrl-C at the keyboard
  - arrival of a packet from a network
  - arrival of data from a disk

- **Synchronous**: caused by executing an instruction
  - Intentional *(Traps)*: e.g. INT 0x80 (syscall) INT 0x3 (breakpoint traps)
  - Unintentional but recoverable *(Faults)*: e.g. page faults, protection faults
  - unintentional and unrecoverable *(Aborts)*: e.g. memory error
Handling exceptions: Interrupt Vectors

- Each type of exception corresponds to a number $k$, used to index into exception table (a.k.a. interrupt vector).
- Handler $k$ is called each time exception $k$ occurs.
- Only OS, not user-level programs, can set up interrupt vector.
OS Handles exceptions

- **fault**: return to `l_current`
- **trap, interrupt**: return to `l_next`
- **Abort**: machine reboot or process killed
Trap Example: Syscall

- **User-level program calls**: `open(filename, options)`
- **Function** `open` **executes system call instruction** `int`

```
0804d070 <__libc_open>:
    ... 804d082: cd 80          int $0x80
    804d084: 5b                  pop %ebx
    ...  
```

- **OS** must find or create file, get it ready for reading or writing
- **Returns** integer file descriptor
Fault Example: Page Fault

- User-level program writes to a memory address
- That address is currently not yet loaded from disk to memory

```
80483b7:   c7 05 10 9d 04 08 0d  movl   $0xd,0x8049d10
```

- OS loads required page into physical memory
- Returns to faulting instruction
- Successful on second try
Fault Example: Invalid Memory Reference

- User-level program de-references an invalid pointer

```
c7 05 60 ff ff ff ff movl $0xd,0xffffffff
```

- **OS** detects invalid address
- Sends **SIGSEGV** signal to user process
- User process exits with “segmentation fault”
OS abstraction: processes

- A **process** is an instance of a running program.
  - Not the same as “program” or “processor”

- With process, OS provides a running program with two abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
  - Private virtual address space
    - Each program seems to have exclusive use of main memory

- How are these Illusions maintained?
  - Context switches between multiple processes
  - Virtual memory
OS interleaves execution of different processes: context switch
Basic UNIX syscalls for managing processes

- **fork**
  - Create a new process

- **exit**

- **wait**
  - Synchronize among processes

- **execve**

Syscalls are documented in man pages section 2: `man -s 2 fork`

Standard C library provides wrapper functions for many syscalls
Example from C

```c
#include <unistd.h>
#include <sys/syscall.h>
#include <errno.h>

...

int rc;
rc = syscall(SYS_chmod, "/etc/passwd", 0444);
if (rc == -1)
    fprintf(stderr, "chmod failed, errno = %d\n", errno);
```
fork: Creating New Processes

- `int fork(void)`
  - creates a new process (child process) that is identical to the calling process (parent process)
- Fork is called once but returns twice

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent: child pid is %d\n", pid);
}
```
Understanding fork

Process n

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

Child Process m

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

pid = m

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

pid = 0

```c
pid_t pid = fork();
if (pid == 0) {
    printf("hello from child\n");
} else {
    printf("hello from parent\n");
}
```

text:

- hello from parent
- hello from child
Fork Example #1

- Parent and child both run same code
  - Distinguish parent from child by return value from `fork`
- Start with same state, but each has private copy of memory
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

```c
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```
Fork Example #2

• Both parent and child can continue forking

```c
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```
Fork Example #3

• Both parent and child can continue forking

```c
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

Fork Example #4

- Both parent and child can continue forking

```c
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```
Fork Example #5

- Both parent and child can continue forking

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```
**exit**: Ending a process

- `void exit(int status)`
  - exits a process
    - Normally return with status 0
- `atexit(fname)` registers functions to be executed upon exit

```c
void cleanup(void) {
    printf("cleaning up\n");
}

void fork6() {
    atexit(cleanup);
    fork();
    fork();
    exit(0);
}
```
Zombies!

• Idea
  – When process terminates, still consumes system resources (i.e. an entry in process table)
    • Why? So that parents can learn of children’s exit status
  – Called a “zombie”

• Reaping
  – Performed by parent on terminated child
  – Parent is given exit status information
  – OS discards process

• What if parent doesn’t reap?
  – If parent has terminated, then child will be reaped by init process (the great-great-...-grandparent of all user-level processes)
Zombie Example

void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n", getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n", getpid());
        while (1)
            ; /* Infinite loop */
    }
}

- **ps** shows child process as "defunct"
- Killing parent allows child to be reaped by **init**
wait: Synchronizing with Children

- int wait(int *child_status)
  - Blocks until some child exits, return value is the pid of terminated child
  - If multiple children completed, will take in arbitrary order (use waitpid to wait for a specific child)

```c
void fork8() {
    if (fork() == 0) {
        printf("HC: hello from child\n");
    }
    else {
        printf("HP: hello from parent\n");
        wait(NULL);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    exit(0);
}
```

This is how child process is reaped by parent process.
**execve**

- `int execve(char *fname, char *argv[], char *envp[])`
  - Executes program named by `fname`
  - Does not return on success

```c
if ((pid = fork()) == 0) { /* Child runs user job */
    if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}
```
**execve**: Load a new program image

- **execve** causes OS to overwrite code, data, and stack of process
  - keeps pid, open files and signal context
Conclusions

• How user-level program interacts with OS
  – Hardware mechanism: exception control flow
  – Generated externally (interrupts) or internally (traps and faults)

• OS abstraction: Processes
  – Each process appears to have total control of CPU + private memory space
  – UNIX syscalls: fork, wait, exec