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*

\[
\text{<startup>}
\quad \text{inst}_1
\quad \text{inst}_2
\quad \text{inst}_3
\quad \ldots
\quad \text{inst}_n
\quad \text{<shutdown>}
\]
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 Exceptional Control Flow

• 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

```
80483b7:   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”
Application programs

Operating system

Hardware

Beautiful interface

Ugly interface
**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

- Process A
- Process B

User code
OS
User code
OS
User code
OS' process abstraction: Basic Syscalls for Managing Processes

- **fork** spawns new process
  - Called once, returns twice
- **exit** terminates own process
  - Puts it into “zombie” status until its parent reaps
- **wait** and **waitpid** wait for and reap terminated children
- **execve** runs new program in existing process
  - Called once, never returns
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");
}
```

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

```c
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();
    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

```c
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`

```
linux> ./forks7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640

[1] 6639

linux> ps
  PID TTY          TIME CMD
6585 ttty9    00:00:00 tcsh
6639 ttty9    00:00:03 forks
6640 ttty9    00:00:00 forks <defunct>
6641 ttty9    00:00:00 ps

linux> kill 6639
[1] Terminated

linux> ps
  PID TTY          TIME CMD
6585 ttty9    00:00:00 tcsh
6640 ttty9    00:00:00 ps
```
**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
Multitasking

• OS runs many processes “concurrently”
  – Process: a running program

• Context switches from one process to another
  – Suspend process when it needs to do I/O or timer expires
  – Resume process when I/O available or given scheduling opportunity

• Appears as if all processes executing simultaneously
What really happens

What we think it happens!
Unix/Linux Process Hierarchy

init [1]

Login shell

Child

Daemon e.g. httpd

Grandchild

Child

Grandchild
Shell Programs

- A shell is a user-level program that runs programs on behalf of the user.
  - sh        Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - csh       BSD Unix C shell (tcsh: enhanced csh)
  - bash      “Bourne-Again” Shell

```
int main() {
    char cmdline[MAXLINE];

    while (1) {
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

Execution is a sequence of read/evaluate steps:
- Reads a command line from the user.
- Parses the command line and runs programs on behalf of the user.
void eval(char *cmdline) {
    char *argv[MAXARGS]; /* argv for execve() */
    int bg; /* should the job run in bg or fg? */
    pid_t pid; /* process id */

    bg = parseline(cmdline, argv);
    if (!builtin_command(argv)) {
        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);
            }
        }
        if (!bg) { /* parent waits for fg job to terminate */
            int status;
            if (waitpid(pid, &status, 0) < 0)
                unix_error("waitfg: waitpid error");
        }
        else { /* otherwise, don’t wait for bg job */
            printf("%d %s", pid, cmdline);
        }
    } else
}
What Is a “Background Job”?

• Some programs run “for a long time”
  – Example: “delete this file in two hours”

  ```
  unix> sleep 7200; rm /tmp/junk  # shell stuck for 2 hours
  ```

• A “background” job is a process we don't want to wait for

  ```
  unix> (sleep 7200 ; rm /tmp/junk) &
  [1] 907
  unix> # ready for next command
  ```
Problem with Simple Shell Example

• Does not reap background jobs
  – Those jobs will become zombies when they terminate
    • Will never be reaped because shell (typically) will not terminate
  – Will run out of the maxproc quota imposed by OS
    • so that OS does not run out of memory with too many zombies

Solution: Signal

- Have OS interrupt the shell process to alert it when a background job completes

```bash
unix> ulimit -u
202752
```

The maximum number of processes available to a single user.
Signals

- A *signal* is a small message that notifies a process of an event
  - sent from the kernel to a process (an “upcall”)
  - The only information in a signal is its small integer ID (1-30)

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt (e.g., ctl-c from keyboard)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate &amp; Dump</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>

The list is much bigger than this of course.
Receiving Signals

• When kernel is about to schedule a runnable process, check for pending signals.
  • If none exists
    – Pass control to next instruction in the logical flow for process
  • Else
    – For each pending signal $k$, trigger corresponding action for process.

• Each signal type has a default action, e.g.
  – Terminate process: terminates and (optionally) dump core
  – Stop process (until restarted by a SIGCONT signal)
  – Ignore the signal

• User process can “catch” a signal by executing a user-defined function.
Unix systems mechanisms used for sending signals rely on the notion of a process group.
Process Groups

- Every process belongs to exactly one process group

```
<table>
<thead>
<tr>
<th>Process Group</th>
<th>PID</th>
<th>PGID</th>
</tr>
</thead>
<tbody>
<tr>
<td>Foreground job</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Child</td>
<td>21</td>
<td>20</td>
</tr>
<tr>
<td>Child</td>
<td>22</td>
<td>20</td>
</tr>
<tr>
<td>Background job #1</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Background job #2</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>
```

getpgrp()
Return process group of current process

setpgid()
Change process group of a process
Sending Signals with `/bin/kill` Program

- `/bin/kill` program sends arbitrary signal to a process or process group

- Examples
  
  `/bin/kill -9 24818`
  
  Send SIGKILL to process 24818

  `/bin/kill -9 -24817`
  
  Send SIGKILL to every process in process group 24817
Sending Signals from the Keyboard

• Typing `ctrl-c` (`ctrl-z`) sends a **SIGINT (SIGTSTP)** to every job in the foreground process group.
  – **SIGINT** - default action is to terminate each process
  – **SIGTSTP** - default action is to stop (suspend) each process

What is a job?
A Job

- Unix shells use the abstraction of a job to represent the processes that are created as a result of evaluating a single command line.

- At any point in time, there is at most one foreground job and zero or more background jobs.
Installing Signal Handlers

• User program can alter default signal action using signal syscall
  – handler_t *signal(int signum, handler_t *handler)
  – Returns: ptr to previous handler if OK, SIG_ERR on error

• Different values for handler:
  – SIG_IGN: ignore
  – SIG_DFL: revert to the default action
  – The address of a signal handler function
    • Called when process receives signal of type signum
    • When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig) {
    printf("You think hitting ctrl-c will stop the bomb?\n");
    sleep(2);
    printf("Well...");
    sleep(1);
    printf("OK\n");
    exit(0);
}

main() {
    signal(SIGINT, handler); /* installs ctrl-c handler */
    while(1) {
    }
}
Conclusions

• How user-level program interacts with OS
  – 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

• Multitasking is the first step of concurrency!