System calls

- Programming interface to the services provided by the OS
- Application can think of OS as providing a library of services
- Much care spent in keeping the interface secure
  - E.g., parameters are copied to kernel space before they are checked
- Mostly accessed through an API (Application Programming Interface)
  - Win32, POSIX, Java API

The Skinny

- Syscall interface allows separation of concern
  - Innovation
- Narrow
  - Simple
  - Powerful
  - Highly portable
  - Robust

Asynchronous notifications in user space

- Interrupts inform kernel of asynchronous events — what about processes?
  - Signals (UNIX); Asynchronous events (Windows)
- Why?
  - Pre-empting user level threads
  - Asynchronous I/O
  - Suspending/resuming a process (e.g., for debugging)
  - Adapting to changing HW resources provided by OS (e.g., memory)
- Upon receipt
  - Ignore
  - Terminate process
  - Catch through handler

"Everything must change, so that everything can stay the same" — "The Leopard", by T. di Lampedusa
“Everything must change, so that everything can stay the same”

The Leopard, by T. di Lampedusa

Interrupts/Exceptions
- Hardware-defined
- Interrupt vector for handlers (kernel)
- Interrupt stack (kernel)
- Interrupt masking (kernel)
- Processor state (kernel)

Signals/Upcalls
- Kernel-defined
- Handlers (user)
- Signal stack or process stack (user)
- Signal masking (user)
- Processor State (user)

Booting an OS Kernel

1. BIOS
- Basic Input/Output System
- In ROM; includes the first instructions fetched and executed
- BIOS copies Bootloader, checking its cryptographic hash to make sure it has not been tampered with

2. Bootloader copies OS Kernel, checking its cryptographic hash

Booting an OS Kernel

BIOS  Bootloader  OS Kernel

2. Bootloader copies OS Kernel, checking its cryptographic hash
Booting an OS Kernel

1. Kernel initializes its data structures (devices, interrupt vector table, etc)

2. BIOS

3. Bootloader

4. OS Kernel

5. Login app

Booting an OS Kernel

1. Kernel initializes its data structures (devices, interrupt vector table, etc)

2. BIOS

3. Bootloader

4. OS Kernel

5. Login app

Shall we dance?

- All processes are progeny of that first process
- Created with a little help from its friend...

And the dance begins!

<table>
<thead>
<tr>
<th>BIOS</th>
<th>Bootloader</th>
<th>OS Kernel</th>
<th>Login app</th>
</tr>
</thead>
</table>

4. Kernel: Copies first process from disk
   Changes PC and sets mode bit to 1

CreateProcess (Windows)

...via system calls!

forc + exec (UNIX)
Starting a new process: the recipe

1. Allocate & initialize PCB
2. Create and initialize a new address space
3. Load program into address space
4. Allocate user-level and kernel-level stacks.
5. Initialize HW context to begin execution at start
6. Copy arguments (if any) to the base of the user-level stack
7. Inform scheduler that a new process is ready
8. Transfer control to user mode

Which API?

Windows: CreateProcess System Call (simplified)
if (CreateProcess(
    NULL, // No module name (use command line)
    argv[1], // Command line
    NULL, // Process handle not inheritable
    NULL, // Thread handle not inheritable
    FALSE, // Set handle inheritance to FALSE
    0, // No creation flags
    NULL, // Use parent's environment block
    NULL, // Use parent's starting directory
    &si, // Pointer to STARTUPINFO structure
    &pi )); // Ptr to PROCESS_INFORMATION structure

Which API?

Unix: fork() and exec() 

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

int main() {
    int child_pid = fork();
    if (child_pid == 0) { // child process
        printf("I am process #\%d\n", getpid());
        return 0;
    } else { // parent process
        printf("I am the parent of process #\%d\n", child_pid);
        return 0;
    }
}
```

Possible outputs?
Which API?

Unix: `fork()` and `exec()`

- **fork()**
  - int pid = fork()
  - Creates a complete copy (child) of the invoking process (parent)
  - Returns twice (!), to both the parent and the child process, setting pid to different values
    - for the child: pid := 0;
    - for the parent: pid := child's process id

- **exec()**
  - Loads executable in memory & starts executing it
  - code, stack, heap are overwritten
  - the process is now running a different program!

wait() and exit()

- **wait()** causes parent to wait until child terminates
  - parent gets return value from child
  - if no children alive, wait() returns immediately
- **exit()** is called after program terminates
  - closes open files
  - deallocates memory
  - deallocates most OS structures
  - checks if parent is alive. If so...

Creating and managing processes

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork()</td>
<td>Create a child process as a clone of the current process. Return to both parent and child. Return child's pid to parent process; return 0 to child</td>
</tr>
<tr>
<td><code>exec</code> (proc, args)</td>
<td>Run the application <code>prog</code> in the current context with the specified <code>args</code></td>
</tr>
<tr>
<td><code>wait</code> (&amp;status)</td>
<td>Pause until some child process has exited</td>
</tr>
<tr>
<td><code>exit</code> (status)</td>
<td>Tell kernel current process is complete and its data structures (stack, heap, code) should be garbage collected. May keep PCB.</td>
</tr>
<tr>
<td><code>kill</code> (pid, type)</td>
<td>Send a signal of a specified type to a process (a bit of an overdramatic misnomer...)</td>
</tr>
</tbody>
</table>

In action

```c
int pid = fork();
if (pid==0)
    exec(B);
else
    wait(&status);
```

Process 13
- Program A
- PC
- pid
- pid
- exit(3)

Process 14
- Program B
- PC
- pid
- pid
- wait(&status)
In action

Process 13
Program A

pid = fork();
if (pid==0)
exec(B);
else
wait(&status);

Process 14
Program B

main() {
...
exit(3);
}

What is a shell?

Job control system

- Runs programs on behalf of the user
- Allows programmer to create/manage set of programs
  - sh Original Unix shell (Bourne, 1977)
  - csh BSD Unix C shell (tcsh enhances it)
  - bash “Bourne again” shell

- Every command typed in the shell starts a child process of the shell
- Runs at user-level. Uses syscalls: fork, exec, etc.

The Unix shell (simplified)

while(! EOF)
read input
handle regular expressions
int pid = fork() // create child
if (pid == 0) { // child here
  exec(“program”, argc, argv0,...);
} else { // parent here
...
}

More on signals

<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., CTRL-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>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>
<tr>
<td>20</td>
<td>SIGSTP</td>
<td>Stop until</td>
<td>Stop signal from terminal (e.g., CTRL-Z from keyboard)</td>
</tr>
</tbody>
</table>
```c
int main() {
    pid_t pid[N];
    int i, child_status;

    for (i = 0; i < N; i++) { // N forks
        if ((pid[i] = fork()) == 0) {
            while(1); // child infinite loop
        }
    }

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        // parent continues executing
        printf("Killing proc. %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status)) { // parent checks for each child's exit
            printf("Child %d terminated w/exit status %d\n", wpid, WEXITSTATUS(child_status));
        } else {
            printf("Child %d terminated abnormally\n", wpid);
        }
    }
    exit(0);
}
```

```c
void int_handler(int sig) {
    printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

int main() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler); // register handler for SIGINT
    for (i = 0; i < N; i++) { // N forks
        if ((pid[i] = fork()) == 0) {
            while(1); // child infinite loop
        }
    }

    /* Parent terminates the child processes */
    for (i = 0; i < N; i++) {
        // parent continues executing
        printf("Killing proc. %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    /* Parent reaps terminated children */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status)) { // parent checks for each child's exit
            printf("Child %d terminated w/exit status %d\n", wpid, WEXITSTATUS(child_status));
        } else {
            printf("Child %d terminated abnormally\n", wpid);
        }
    }
    exit(0);
}
```

### Kernel Operation (conceptual, simplified)

- Initialize devices
- Initialize "first process"

```c
while (TRUE) {
    □ while device interrupts pending
        - handle device interrupts
    □ while system calls pending
        - handle system calls
    □ if run queue is non-empty
        - select a runnable process and switch to it
    □ otherwise
        - wait for device interrupt
}
```

### Threads

An abstraction for concurrency

- CPU Scheduling

Kernel Operation An abstraction for concurrency

Rethinking the process abstraction

- The Process, as we know it, serves two key purposes in the OS:
  - It defines the granularity at which the OS offers isolation:
    - each process defines an address space that identifies what can be touched by the program
  - It defines the granularity at which the OS offers scheduling and can express concurrency:
    - each process defines a stream of instructions executed sequentially

Thread: a new abstraction for concurrency

- A single-execution stream of instructions that represents a separately schedulable task:
  - OS can run, suspend, resume a thread at any time
  - bound to a process (lives in an address space)
  - Finite Progress Axiom: execution proceeds at some unspecified, non-zero speed
- Virtualizes the processor:
  - programs run on machine with an infinite number of processors (hint: not true)
- Allows to specify tasks that should be run concurrently...
  - ...and lets us code each task sequentially