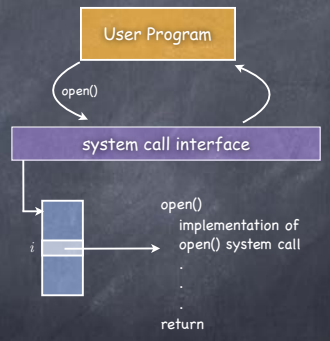
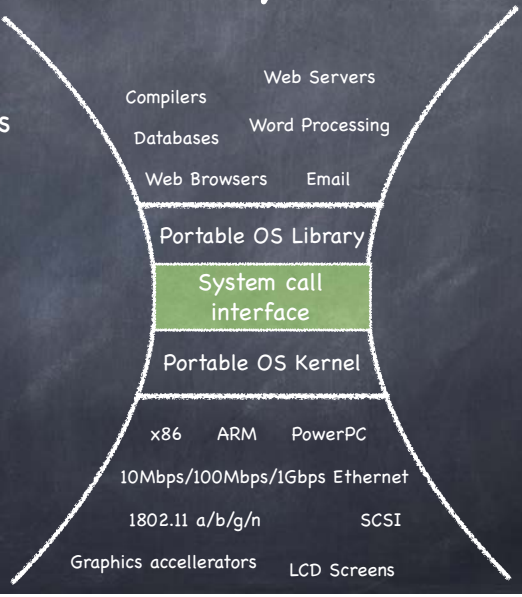
System calls

- 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 interface 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	Signals/Upcalls
Hardware-defined	Kernel-defined
Interrupt vector for handlers (kernel)	Handlers (user)
Interrupt stack (kernel)	Signal stack or process stack (user)
Interrupt masking (kernel)	Signal masking (user)
Processor state (kernel)	Processor State (user)

Booting an OS Kernel

- Basic Input/Output System
  - In ROM; includes the first instructions fetched and executed

(1) BIOS copies Bootloader, checking its cryptographic hash to make sure it has not been tampered with

Booting an OS Kernel

(2) Bootloader copies OS Kernel, checking its cryptographic hash

Booting an OS Kernel

(2) Bootloader copies OS Kernel, checking its cryptographic hash

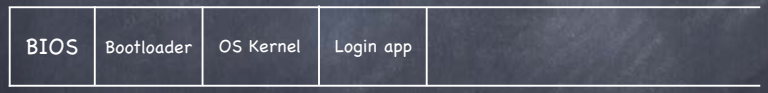
Booting an OS Kernel

③ Kernel initializes its data structures (devices, interrupt vector table, etc)

Booting an OS Kernel

④ Kernel: Copies first process from disk

Booting an OS Kernel



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

And the dance begins!

Shall we dance?

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

...via system calls!

CreateProcess (Windows)

fork + exec (UNIX)

Starting a new process: the recipe

1. Allocate & initialize PCB
2. Create and initialize a new address space
3. Load program intro address space
4. Allocate user-level and kernel-level stacks.
5. Initialize H/W 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()

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

In action

```
#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!

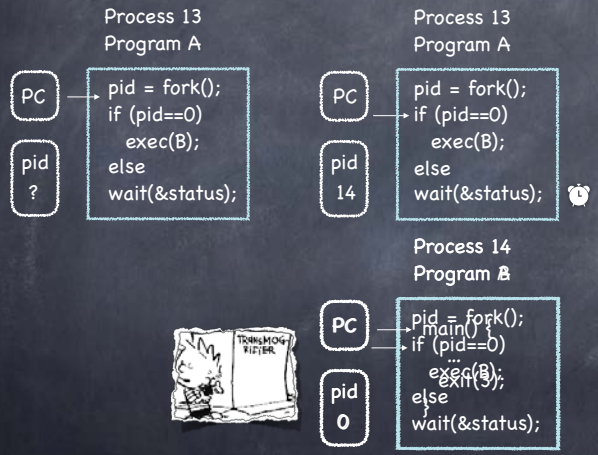
wait() and exit()

- 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...

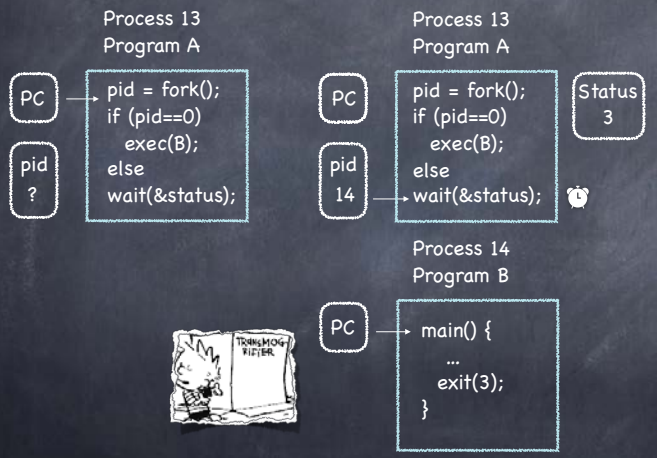
Creating and managing processes

Syscall	Description
fork()	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
exec (proc, args)	Run the application prog in the current context with the specified args
wait (&status)	Pause until some child process has exited
exit (status)	Tell kernel current process is complete and its data structures (stack, heap, code) should be garbage collected. May keep PCB.
kill (pid, type)	Send a signal of a specified type to a process (a bit of an overdramatic misnomer...)

In action



In action



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

ID	Name	Default Action	Corresponding Event
2	SIGINT	Terminate	Interrupt (e.g., CTRL-C from keyboard)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
20	SIGSTP	Stop until SIGCONT	Stop signal from terminal (e.g., CTRL-Z from keyboard)

Signal Example

```
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);
}
```

Handler Example

```
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"
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

}

CPU Scheduling