Today

- Process Management

Review

- Bringing scheduling all together

Processes and Their Threads

virtual address space

Each process has a virtual address space (VAS): a private name space for the virtual memory it uses.

The VAS is both a "**sandbox**" and a "**lockbox**": it limits what the process can see/do, and protects its data from others.

+

main thread

Each process has a **main thread** bound to the VAS, with a stack.

If we say a process does something, we really mean its thread does it.

The kernel can suspend/ restart a thread wherever and whenever it wants.

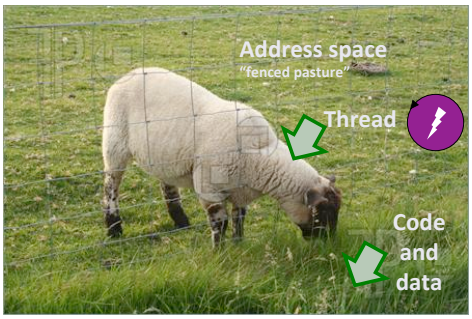
+...

other threads (optional)

On real systems, a process can have multiple threads.

We presume that they can all make system calls and **block** independently.

Sheep Analogy



All Together Now

- A process is a running program
- A running program (a process) has at least one thread ("main")
  - It may (optionally) create other threads.
- Threads execute the program ("perform the script").
- Threads execute on the "stage" of the process virtual memory, with access to a private instance of the program's code and data.
- A thread can access any virtual memory in its process but is contained by the "fence" of the process virtual address space.
- Threads run on cores: a thread's core executes instructions for it.
- Sometimes threads idle to wait for a free core or for some event. Sometimes cores idle to wait for a ready thread to run.
- The OS kernel shares/multiplexes the computer's memory and cores among the virtual memories and threads.

Process management

- OS offers system call APIs for managing processes.
  - Create processes (children)
  - Control processes
  - Monitor process execution
  - "Join": wait for a process to exit and return a result
  - "Kill": send a signal to a process
  - Establish interprocess communication (IPC: later)
  - Launch a program within a process
- We study the Unix process abstraction as an example.
  - Illustrative and widely used for 40+ years!

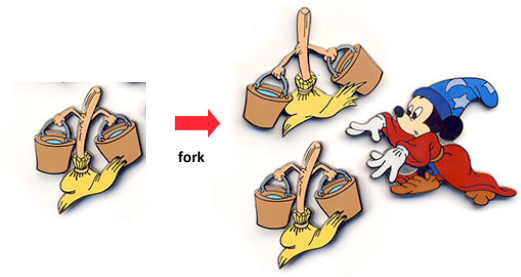
Process Management

- The *process manager* must provide for:
  - Process creation
  - Process termination
  - Process synchronization
  - Inter-process communication
  - Process scheduling

Program Perspective

- Programs use system calls to create and manage processes.
  - The specific system calls used depend upon the type of the system.

The essence of Unix process "fork"



Oh Ghost of Walt, please don't sue me.

Sorcerer's Apprentice Atari Game

Review: fork

```
int pid;
int status = 0;

if (pid = fork()) {
      /* parent */
      ….
} else {
      /* child */
      ….
      exit(status);
}
```

The **fork** syscall returns *twice*:

1. It returns a zero in the context of the new child process.
2. It returns the new child process ID (pid) in the context of the parent.

Exit and wait

- exit(int rv)
  - Causes the program to exit with the main method returning the specified return value (rv).
    - e.g. exit(-1);
  - Reaching the end of the main method results in an implicit exit(0).
- wait(int *status)
  - Causes a process to wait until any one of its child processes has completed.
  - The waitpid system call can be used to wait for a specific child process to complete.
  - status is loaded with the return value from the child's call to exit. Use NULL to discard status.

Fork Problem

```
int main() {
    int x = 27;
    int pid = fork();
    if (pid != 0) {
        printf("Parent's x before wait is %d\n",x);
        x = x + 5;
        wait(NULL);
        printf("Parent's x after wait is %d\n",x);
    } else {
        printf("Child's x before sleep is %d\n",x);
        sleep(5);
        x = x + 10;
        printf("Child's x after sleep is %d\n",x);
    }
}
```


Another Fork Program

```
int main() {
    int pid= fork();
    int i;

    if (pid != 0 ) {
        for(i=0; i<10; i++) {
            printf("Parent process %d running.\n", getpid());
            sleep(1);
        }
        wait(NULL);
    }
    else {
        for(i=0; i < 10; i++) {
            printf("Child process %d running.\n", getpid());
            sleep(1);
        }
    }
}
```

getpid syscall: Get processID of current process.

A simple program: sixforks

```
…
int main(int argc, char* argv) {
  fork();
  fork();
  fork();
  fork();
  fork();
  fork();
  printf("Process %d exiting.\n", getpid());
}
```

How many processes are created by these six forks?

A simple program: sixforks

```
…
int main(int argc, char* argv) {
  fork();
  fork();
  fork();
  fork();
  fork();
  fork();
  printf("Process %d exiting.\n",
       getpid());
}
```

```
$ ./sixforks
Process 15191 exiting.
Process 15200 exiting.
Process 15195 exiting.
Process 15194 exiting.
Process 15197 exiting.
Process 15202 exiting.
Process 15193 exiting.
Process 15198 exiting.
Process 15215 exiting.
Process 15217 exiting.
Process 15218 exiting.
Process 15203 exiting.
$ Process 15212 exiting.
Process 15196 exiting.
Process 15222 exiting.
Process 15213 exiting.
Process 15221 exiting.
Process 15224 exiting.
Process 15206 exiting.
Process 15216 exiting.
Process 15205 exiting.
Process 15207 exiting.
Process 15201 exiting.
Process 15214 exiting.
Process 15225 exiting.
Process 15199 exiting.
Process 15226 exiting.
Process 15208 exiting.
Process 15229 exiting.
Process 15220 exiting.
Process 15209 exiting.
Process 15232 exiting.
Process 15219 exiting.
Process 15233 exiting.
Process 15223 exiting.
Process 15210 exiting.
Process 15234 exiting.
Process 15228 exiting.
Process 15192 exiting.
Process 15230 exiting.
Process 15211 exiting.
Process 15227 exiting.
Process 15239 exiting.
Process 15231 exiting.
Process 15242 exiting.
Process 15243 exiting.
Process 15240 exiting.
Process 15236 exiting.
Process 15241 exiting.
Process 15244 exiting.
Process 15247 exiting.
Process 15235 exiting.
Process 15245 exiting.
Process 15250 exiting.
Process 15248 exiting.
Process 15249 exiting.
Process 15204 exiting.
Process 15238 exiting.
Process 15251 exiting.
Process 15237 exiting.
Process 15252 exiting.
Process 15253 exiting.
Process 15246 exiting.
Process 15254 exiting.
```


Project 2

- System Calls
  - Interrupts
- Due in 2 weeks

Exceptions: trap, fault, interrupt

	intentional happens every time	**unintentional** contributing factors
synchronous caused by an instruction	**trap: system call** open, close, read, write, fork, exec, exit, wait, kill, etc.	**fault** invalid or protected address or opcode, page fault, overflow, etc.
asynchronous caused by some other event	"software interrupt" software requests an interrupt to be delivered at a later time	**interrupt** caused by an external event: I/O op completed, clock tick, power fail, etc.

Looking Ahead

- Process Communication
- Storage
- Midterm next Wednesday
  - Post a midterm prep document
  - More on the types of questions