Processes

What is a Process

- Process: An execution stream and its associated state
- Execution Stream
  - Set of instructions
  - "Thread of control"
- Process State
  - Hardware state
    - Privilege level, segments, page tables
  - OS State
    - Priority, I/O Buffers, Heap, memory map
  - Resource state
    - I/O requests
- An abstraction to make it easier to program both OS and applications
  - Encapsulate state into manageable unit

Programs and Processes

- A process is not a program
- Program: Static code and static data

Program

```
int foo() {
    return 0;
}

int main() {
    foo();
    return 0;
}
```

Process

```
int foo() {
    return 0;
}

int main() {
    foo();
    return 0;
}
```

Heap
Stack
Registers

- OS can host multiple processes of same program
  - E.g. many users can run 'ls' at the same time
- One program can invoke multiple processes
  - E.g. make runs many processes to compile code
- No one-to-one mapping between programs and processes

System Classification

- Uniprogramming: Only one process at a time
  - Examples: Original systems and older PC Oses
    - DOS
  - Advantages: Easier for OS designer
  - Disadvantages: Terrible utilization, poor usability
- Multiprogramming: Multiple processes at a time
  - Examples: Every modern OS you use
  - Note: Multiprogramming is different from multiprocessing
    - Multiprocessing: Systems with multiple processors
  - Advantages: Better utilization and usability
  - Disadvantages: Complex OS design

PIDs

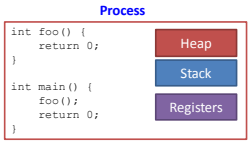
- Each system must track running processes
  - Needs some way to identify them
- Process IDs
  - OS level handle to uniquely identify each process
  - Unsigned integer
- Used every time the system needs to operate on a process
  - Just like file descriptors

Managing processes (Unix)

- Finding processes
  - 'ps'
- Monitoring Processes
  - 'top'
- Stopping processes
  - 'kill <pid>'
- Background/Foreground
  - ctrl-z followed by 'bg'/'fg'
  - ./hello &

Process State

- Q: Is this all a process is?
- A: No, additional state inside the OS
  - What is it's name?
  - Who is the owner?
  - What permissions does it have?
  - How long has it been running?



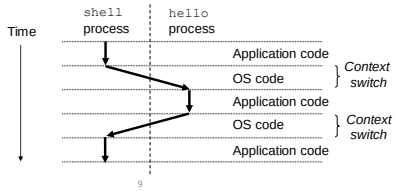
- OS tracks the state of each process in the system
  - Stored in a process list, not a table (why?)
  - Process ID (PID) used to look up process entry from list
  - OS state of a process referred to as the **Process Control Block (PCB)**

/proc

- File system based export of process information
  - Contains information on every process on the system
  - Organized by PID
  - **/proc/self** exports information for the running process
- Information available
  - Resources in use
  - Resources allowed to use
  - Memory map
  - Entire memory image
  - Cmdline
  - Etc…

Context Switching

- OS provides the illusion of a dedicated machine per process
- Process
  - OS's abstraction of a running program
- Context switch
  - Saving context of one process, restoring that of another one
  - Distorted notion of time



9

Context Switch implementation

- Machine dependent code (Assembly!)
  - Different for MIPS, ARM, x86, etc.
  - Save process state
    - Registers, PCB, etc
- Tricky: OS must save state without changing state
- Requires special hardware support
  - Save process state on each trap/interrupt
  - Very nasty

Dispatch Mechanism

- OS maintains list of all processes (PCBs)
- Each process has a mode
  - Running: Executing on the CPU
  - Ready: Waiting to execute on CPU
  - Blocked: Waiting for I/O or synchronization with another thread
- Dispatch Loop

```
while (1) {
    run process for a while;
    stop process and save its state;
    load state of another process;
}
```

How does dispatcher gain control?

- Must change from user to system mode
  - Problem: Only one CPU, and CPU can only do one thing at a time
  - A user process running means the dispatcher isn't
- Two ways OS gains control
- Traps: Events caused by process execution
  - System calls, page faults, Exceptions (segfault, etc)
- Hardware interrupts: Events external to process
  - Typing at keyboard, network packet arrivals
  - Control switch to OS via Interrupt Service Routine (ISR)
- How does OS guarantee it will regain control?

Process Creation

- Two ways to create a process
  - Build one from scratch
  - Clone an existing one
- Option 1: From scratch (Windows – CreateProcess(…))
  - Load specified code and data into memory
  - Create empty call stack
  - Create and initialize PCB (make it look like a context switch)
  - Add process to ready list
- Option 2: Cloning (UNIX – fork())
  - Stop current process and save its state
  - Copy code, data, stack and PCB
  - Add new Process to ready list
  - Do we really need to copy everything?

Creating a process (Windows and UNIX)

Windows

```
BOOL WINAPI CreateProcess( _In_opt_    LPCTSTR lpApplicationName,
                           _Inout_opt_ LPTSTR lpCommandLine,
                           _In_opt_    LPSECURITY_ATTRIBUTES lpProcessAttributes,
                           _In_opt_    LPSECURITY_ATTRIBUTES lpThreadAttributes,
                           _In_        BOOL bInheritHandles,
                           _In_        DWORD dwCreationFlags,
                           _In_opt_    LPVOID lpEnvironment,
                           _In_opt_    LPCTSTR lpCurrentDirectory,
                           _In_        LPSTARTUPINFO lpStartupInfo,
                           _Out_       LPPROCESS_INFORMATION lpProcessInformation );
```

UNIX

```
int fork();
```

Creating processes in UNIX

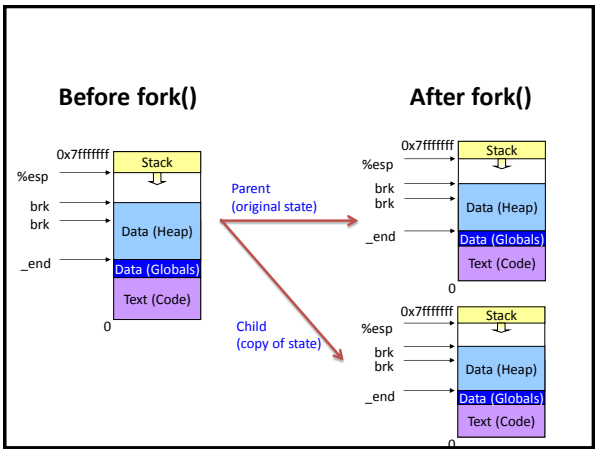
- Combination of fork() and exec(…)
  - fork(): Clone current process
  - exec(…): copy new program on top of current process

```
int main() {
    int pid;
    char * cmd = "/bin/sh";

    pid = fork();

    if (pid == 0) {
        // child process
        exec(cmd);
        // exec does not return, WE NEVER GET HERE
    } else {
        // parent process - wait for child to finish
        wait(pid);
    }
}
```

- Advantage: Flexible, clean, simple



Parent-child relationship

- Parent/child processes are always connected
  - Parent creates processes
  - Parent must also "reap" children
- Reaping
  - Waiting for a process to die
  - What is the point?
    - 'int main()' returns a value
    - Where does that value go?

waiting

- wait(int * ret_val);
  - Takes a pointer to an int variable
  - Sets that value to whatever the child's main() returns
- waitpid(int pid, int * ret_val, int options)
  - Waits for a specific process to exit