What is a Process?

- A process is a program during execution.
  - Program = static file (image)
  - Process = executing program = program + execution state.
- A process is the basic unit of execution in an operating system
  - Each process has a number, its process identifier (pid).
- Different processes may run different instances of the same program
  - E.g., my $javac$ and your $javac$ process both run the Java compiler
- At a minimum, process execution requires following resources:
  - Memory to contain the program code and data
  - A set of CPU registers to support execution

---

Process in Memory

- Program to process.
  - What you wrote
```java
void X {int b} {
  if(b == 1) {
    ...
  }
  int main() {
    int a = 2;
    X(a);
  }
}
```
  - What is in memory
```
main: a = 2
X, b = 2
```
  - Stack
```
void X {int b} {
  if(b == 1) {
    ...
  }
  int main() {
    int a = 2;
    X(a);
  }
}
```
  - Heap
```
void X {int b} {
  if(b == 1) {
    ...
  }
  int main() {
    int a = 2;
    X(a);
  }
```
  - Code

---

Processes and Process Management

- A program consists of code and data
- On running a program, the loader:
  - reads and interprets the executable file
  - sets up the process’ memory to contain the code & data from executable
  - pushes “argc”, “argv” on the stack
  - sets the CPU registers properly & calls “_start()”
- Program starts running at _start()
  `_start(args) {`  
  `_initio();`  
  `main(args);`  
  `exit(0);`  
  
  we say “process” is now running, and no longer think of “program”
- When main() returns, OS calls “exit()” which destroys the process and returns all resources

---

Keeping track of a process

- A process has code.
  - OS must track program counter (code location).
- A process has a stack.
  - OS must track stack pointer.
- OS stores state of processes’ computation in a process control block (PCB).
  - E.g., each process has an identifier (process identifier, or PID)
- Data (program instructions, stack & heap) resides in memory, metadata is in PCB (which is a kernel data structure in memory)
**Context Switching**

- The OS periodically switches execution from one process to another.
- Called a **context switch**, because the OS saves one execution context and loads another.

**What causes context switches?**

- Waiting for I/O (disk, network, etc.)
  - Might as well use the CPU for something useful
  - Called a blocked state
- Timer interrupt (preemptive multitasking)
  - Even if a process is busy, we need to be fair to other programs
- Voluntary yielding (cooperative multitasking)
- A few others
  - Synchronization, IPC, etc.

**Process Life Cycle**

- Processes are always either **executing**, waiting to execute or blocked waiting for an event to occur.
- A preemptive scheduler will force a transition from running to ready. A non-preemptive scheduler waits.

**Process Contexts**

**Example: Multiprogramming**

- OS has PCBs for active processes.
- OS puts PCB on an appropriate queue.
  - Ready to run queue.
  - Blocked for IO queue (Queue per device).
  - Zombie queue.
- Stopping a process and starting another is called a context switch.
  - 100-10,000 per second, so must be fast.

**When a process is waiting for I/O what is its scheduling state?**

1. Ready
2. Running
3. Blocked
4. Zombie
5. Exited
Why Use Processes?

Consider a Web server
- get network message (URL) from client
- fetch URL data from disk
- compose response
- send response

How well does this web server perform?
- With many incoming requests?
- That access data all over the disk?

Why Use Processes?

Consider a Web server
- get network message (URL) from client
- create child process, send it URL
  
  Child
  - fetch URL data from disk
  - compose response
  - send response

- If server has configuration file open for writing
  - Prevent child from overwriting configuration
- How does server know child serviced request?
  - Need return code from child process

Where do new processes come from?

- Parent/child model
- An existing program has to spawn a new one
  - Most OSes have a special 'init' program that launches system services, login daemons, etc.
  - When you log in (via a terminal or ssh), the login program spawns your shell

Approach 1: Windows CreateProcess

- In Windows, when you create a new process, you specify a new program
  - And can optionally allow the child to inherit some resources (e.g., an open file handle)

Approach 2: Unix fork/exec()

- In Unix, a parent makes a copy of itself using fork()
  - Child inherits everything, runs same program
  - Only difference is the return value from fork()
  - A separate exec() system call loads a new program

  Major design trade-off:
  - How easy to inherit
  - Vs. Security (accidentally inheriting something the parent didn’t intend)

  Note that security is a newer concern, and Windows is a newer design...

The Convenience of separating Fork/Exec

- Life with CreateProcess(filename);
  - But I want to close a file in the child.
  - CreateProcess(filename, list of files);
  - And I want to change the child’s environment.
  - CreateProcess(filename, CLOSE_PID, new_envp);
  - Etc. (and a very ugly etc.)

  fork() = split this process into 2 (new PID)
  - Returns 0 in child
  - Returns pid of child in parent

  exec() = overlay this process with new program (PID does not change)
### The Convenience of Separating Fork/Exec

- Decoupling fork and exec lets you do anything to the child’s process environment without adding it to the CreateProcess API.
  
  ```c
  int pid = fork();             // create a child
  if(0 == pid) {               // child continues here
    // Do anything (unmap memory, close net connections...)
    exec("program", argc, argv0, argv1, ...);
  }
  
  fork() creates a child process that inherits:
  - identical copy of all parent’s variables & memory
  - identical copy of all parent’s CPU registers (except one)
  
  Parent and child execute at the same point after fork() returns:
  - by convention, for the child, fork() returns 0
  - by convention, for the parent, fork() returns the process identifier of the child
  - fork() return code a convenience, could always use getpid()

```

### Program Loading: exec()

- The exec() call allows a process to “load” a different program and start execution at main (actually start).
- It allows a process to specify the number of arguments (argc) and the string argument array (argv).
- If the call is successful
  - it is the same process
  - but it runs a different program !!
- Exec does not return!
  
  ```c
  if(0 == pid) {         // child continues here
    exec("calc", argc, argv0, argv1, ...);
    exit(exec_status);
  } else {              // parent continues here
    printf("Who’s your daddy?");
    child_status = wait(pid);
  }
  
```

### General Purpose Process Creation

In the parent process:

```c
main() {
  ... int pid = fork();         // create a child
  if(0 == pid) {               // child continues here
    exec_status = exec("calc", argc, argv0, argv1, ...);
    exit(exec_status);
  } else {                     // parent continues here
    printf("Who’s your daddy?");
    child_status = wait(pid);
  }
}
```

---

### A shell forks and then execs a calculator

```c
int pid = fork();
if(pid == 0) {
  close("./.history");
  exec("/bin/calc");
} else {
  wait(pid);
}
```

### A shell forks and then execs a calculator

```c
int pid = 128;
open files = ".history"
lst_cpu = 0

int sh = fork();
if(sh == 0) {
  close("./.history");
  exec("/bin/calc");
  exec_in(ln);
  do_init();
  close(ln);
  while(1) { q = 7;
    a = 2;
  }
} else {
  wait(pid);
}
```

---

### At what cost, fork()?

- Simple implementation of fork():
  - allocate memory for the child process
  - copy parent’s memory and CPU registers to child’s
  - Expensive
- In 99% of the time, we call exec() after calling fork():
  - the memory copying during fork() operation is useless
  - the child process will likely close the open files & connections
  - overhead is therefore high
- vfork():
  - a system call that creates a process “without” creating an identical memory image
  - child process should call exec() almost immediately
  - Unfortunate example of implementation influence on interface
  - Current Linux & BSD 4.4 have it for backwards compatibility
  - Copy-on-write to implement fork avoids need for vfork
Orderly Termination: exit()

- After the program finishes execution, it calls exit().
- This system call:
  - Takes the "result" of the program as an argument.
  - Closes all open files, connections, etc.
  - Deallocates memory.
  - Deallocates most of the OS structures supporting the process.
  - Checks if parent is alive:
    - If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the zombie/defunct state.
    - If not, it deallocates all data structures, the process is dead.
  - Cleans up all waiting zombies.
- Process termination is the ultimate garbage collection (resource reclamation).

The wait() System Call

- A child program returns a value to the parent, so the parent must arrange to receive that value.
- The wait() system call serves this purpose:
  - It puts the parent to sleep waiting for a child's result.
  - When a child calls exit(), the OS unblocks the parent and returns the value passed by exit() as a result of the wait call (along with the pid of the child).
  - If there are no children alive, wait() returns immediately.
  - Also, if there are zombies waiting for their parents, wait() returns one of the values immediately (and deallocates the zombie).

Process Control

OS must include calls to enable special control of a process:

- Priority manipulation:
  - nice(), which specifies base process priority (initial priority).
  - In UNIX, process priority decays as the process consumes CPU.
- Debugging support:
  - ptrace(), allows a process to be put under control of another process.
  - The other process can set breakpoints, examine registers, etc.
- Alarms and time:
  - Sleep puts a process on a timer queue waiting for some number of seconds, supporting an alarm functionality.

Tying it All Together: The Unix Shell

```c
while(! EOF) {
    read input
    handle regular expressions
    int pid = fork(); // create a child
    if(pid == 0) {
        // child continues here
        exec(“program”, argc, argv0, argv1, …);
    }
    else { // parent continues here
        …
    }
}
```

- Translates <CTRL-C> to the kill() system call with SIGKILL.
- Translates <CTRL-Z> to the kill() system call with SIGSTOP.
- Allows input-output redirections, pipes, and a lot of other stuff that we will see later.

Summary

- Understand what a process is.
- The high-level idea of context switching and process states.
- How a process is created.
- Pros and cons of different creation APIs:
  - Intuition of copy-on-write fork and vfork.

-