Process Creation

- Processes get created (and destroyed) all the time in a typical computer
- Some by explicit user command
- Some by invocation from other running processes
- Some at the behest of the operating system
- How do we create a new process?

Creating a Process Descriptor

- The process descriptor is the OS' basic per-process data structure
- So a new process needs a new descriptor
- What does the OS do with the descriptor?
- Typically puts it into a *process table*
  - The data structure the OS uses to organize all currently active processes

What Else Does a New Process Need?

- A virtual address space
- To hold all of the segments it will need
- So the OS needs to create one
  - And allocate memory for code, data and stack
- OS then loads program code and data into new segments
- Initializes a stack segment
- Sets up initial registers (PC, PS, SP)

Choices for Process Creation

1. Start with a "blank" process
   - No initial state or resources
   - Have some way of filling in the vital stuff
     - Code
     - Program counter, etc.
   - This is the basic Windows approach
2. Use the calling process as a template
   - Give new process the same stuff as the old one
   - Including code, PC, etc.
   - This is the basic Unix/Linux approach

Starting With a Blank Process

- Basically, create a brand new process
- The system call that creates it obviously needs to provide some information
  - Everything needed to set up the process properly
  - At the minimum, what code is to be run
  - Generally a lot more than that
- Other than bootstrapping, the new process is created by command of an existing process

Windows Process Creation

- The `CreateProcess()` system call
- A very flexible way to create a new process
  - Many parameters with many possible values
- Generally, the system call includes the name of the program to run
  - In one of a couple of parameter locations
- Different parameters fill out other critical information for the new process
  - Environment information, priorities, etc.

Process Forking

- The way Unix/Linux creates processes
- Essentially clones the existing process
- On assumption that the new process is a lot like the old one
  - Most likely to be true for some kinds of parallel programming
  - Not so likely for more typical user computing

Why Did Unix Use Forking?

- Avoids costs of copying a lot of code
  - *If* it's the same code as the parents' . . .
- Historical reasons
  - Parallel processing literature used a cloning fork
  - Fork allowed parallelism before threads invented
- Practical reasons
  - Easy to manage shared resources
    - Like stdin, stdout, stderr
  - Easy to set up process pipe-lines (e.g. ls | more)
  - Share exclusive-access resources (e.g. tape drives)

What Happens After a Fork?

- There are now two processes
  - With different IDs
  - But otherwise mostly exactly the same
- How do I profitably use that?
- Program executes a fork
- Now there are two programs
  - With the same code and program counter
- Write code to figure out which is which
  - Usually, parent goes "one way" and child goes "the other"

Forking and the Data Segments

- Forked child shares the parent's code
- But not its stack
  - It has its own stack, initialized to match the parent's
  - Just as if a second process running the same program had reached the same point in its run
- Child should have its own data segment, though
  - Forked processes do not share their data segments

Forking and Copy on Write

- If the parent had a big data area, setting up a separate copy for the child is expensive
  - And fork was supposed to be cheap
- If neither parent nor child write the parent's data area, though, no copy necessary
- So set it up as copy on write
- If one of them writes it, then make a copy and let the process write the copy
  - The other process keeps the original

Sample Use of Fork

```
if (fork() ) {
    /* I'm the parent!   */
     execute parent code
} else {
    /* I'm the child! */
     execute the child code
}
```

- Parent and child code could be very different
- In fact, often you want the child to be a totally different program
  - And maybe not share the parent's resources

But Fork Isn't What I Usually Want!

- Indeed, you usually don't want another copy of the same process
- You want a process to do something entirely different
- Handled with exec
  - A Unix system call to "remake" a process
  - Changes the code associated with a process
  - Resets much of the rest of its state, too
    - Like open files

The `exec` Call

- A Linux/Unix system call to handle the common case
- Replaces a process' existing program with a different one
  - New code
  - Different set of other resources
  - Different PC and stack
- Essentially, called after you do a fork

Using exec

```
if (fork() ) {
    /* I'm the parent!   */
     continue with what I was doing before
} else {
    /* I'm the child! */
     exec("new program", <program arguments>;
}
```

- The parent goes on to whatever is next
- The child replaces its code with "new program"

How Does the OS Handle Exec?

- Must get rid of the child's old code
  - And its stack and data areas
  - Latter is easy if you are using copy-on-write
- Must load a brand new set of code for that process
- Must initialize child's stack, PC, and other relevant control structure
  - To start a fresh program run for the child process

New Processes and Threads

- All processes have at least one thread
  - In some older OSes, never more than one
    - In which case, the thread is not explicitly represented
  - In newer OSes, processes typically start with one thread
- As process executes, it can create new threads
- New thread stacks allocated as needed

A Thread Implementation Choice

- Threads can be implemented in one of two ways

1. The kernel implements them
2. User code implements them

- These alternatives have fundamental differences

User Threads

- The kernel doesn't know about multiple threads per process
- The process itself knows
- So the process must schedule its threads
- Since the kernel doesn't know the process has multiple threads,
  - The process can't run threads on more than one core
- Switching threads doesn't require OS involvement, though
  - Which can be cheaper

Typical Use of User Threads

- A server process that expects to have multiple simultaneous clients
- Server process can spawn a new user thread for each client
- And can then use its own scheduling methods to determine which thread to run when
- OS need not get involved in running threads
  - No context switch costs to change from one client to another

Kernel Threads

- The OS is aware that processes can contain more than one thread
- Creating threads is an OS operation
- Scheduling of threads handled by OS
  - Which can schedule several process threads on different cores simultaneously
- Saves the program complexity of handling threads
- But somewhat more heavyweight

Typical Use of Kernel Threads

- A program that can do significant parallel processing on its data
- Each parallel operation is run as a kernel thread
  - All sharing the same data space and code
  - But each with its own stack
- If multiple cores available, OS can achieve true parallelism for the program

Process Termination

- Most processes terminate
  - All do, of course, when the machine goes down
  - But most do some work and then exit before that
  - Others are killed by the OS or another process
- When a process terminates, the OS needs to clean it up
  - Essentially, getting rid of all of its resources
  - In a way that allows simple reclamation