Processes (Part III)

Godmar Back

Virginia Tech

May 31, 2020

VIRGINIA TECH

Process Management

- OS provide APIs (system calls) to manage processes
- Process creation
  - includes way to set up new process's environment
- Process termination
  - Normal termination (`exit()`, return from main())
  - Abnormal termination (due to misbehavior: "crash", due to outside intervention: "kill")
  - In either case, OS cleans up (reclaims all memory, closes all low-level file descriptors)
- Process interaction; examples include
  - Waiting for a process to finish
  - Stopping/continuing a process
- Change a process's scheduling and other attributes
- Reporting and profiling facilities
- OS provides facilities to be used by or in coordination with control programs (shell, GUI, Task Manager)
  - Examples include Ctrl-C, Ctrl-Z

VIRGINIA TECH

Process Management (Windows)

- OS provide APIs (system calls) to manage processes
- Example: CreateProcessA in Windows

```
BOOL CreateProcessA(
  LPCSTR                lpApplicationName,
  LPSTR                 lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCSTR                lpCurrentDirectory,
  LPSTARTUPINFOA        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

- Creates ("spawns") a new process, and instruct it to run a new program with arguments and attributes

VIRGINIA TECH

Process Management (Unix)

- Unix separates process creation from loading a new program
- The `fork()` system call creates a new process, but does not load a new program
- The newly created process is called a child process (the creating process is referred to as parent)
  - Corollary: Unix processes form a tree-like hierarchy
  - Child processes may inherit parts of their environment from their parents, but are otherwise distinct entities
- The child process then may change/set up the environment and, when ready, load a new program that replaces the current program but retains certain aspects of the environment (`exec()`)
- The parent has the option of waiting (via `wait()`) for the child process to terminate, which is also called "joining" the child process
  - Parent can also learn how the child process terminated, e.g. the code that the child passed to `exit()`

VIRGINIA TECH

Comparison of `fork()` and `exec()`

- `fork()`
  - Keeps program and process, but also creates a new process
  - New process is a clone of the parent; child state is a (now separate) copy of parent's state, including everything: heap, stack, file descriptors
  - Called once, returns twice (once in parent, once in child)
- `exec()`
  - Keeps process, but discards old program and loads a new program
  - Reinitializes process state (clears heap + stack, starts at new program's `main()`); except it retains file descriptors
  - If successful, is called once but does not return
  - includes multiple variants (`execvp()`, etc.)

VIRGINIA TECH

fork/exec/exit/wait

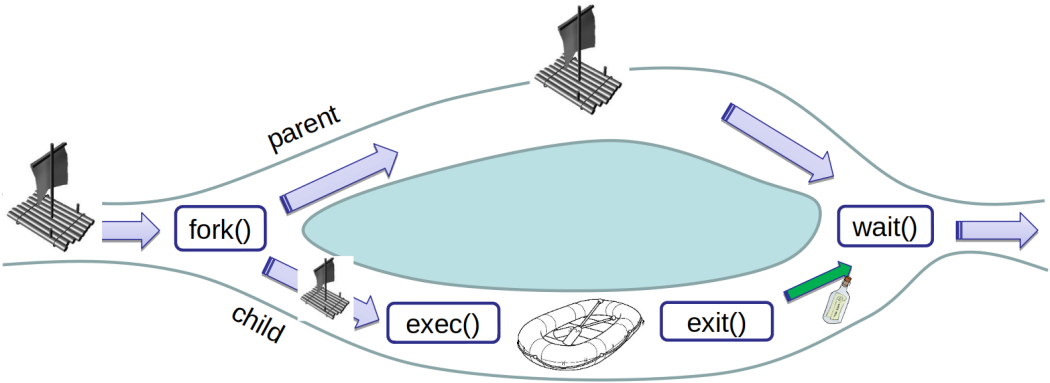

Figure 1: Parent/child control flows in typical scenario where a child process is forked with the intent of executing a program

VIRGINIA TECH

Some Unix Jargon for various scenarios

Zombies

Processes that have exited, but whose parent is still alive **and** has not (yet) waited for them. They will exist until either their parent waits for them ("reaps them") or their parent exits.

Orphans & Daemons

Processes that are alive, but whose parent exited without waiting for them. They are reassigned to the `init` process (pid 1). Usually unintended. If intended, the orphan may also be called a daemon.

Run-Aways

Processes that are alive, have not exited, are always READY/RUNNING and thus, if scheduled, use up 100% of a CPU without performing useful work.