Review of Process metadata/communication

* Process components:
  + PID
  + PPID
  + Program Counter: addr of next instruction
  + Process state (running, waiting to run, terminated)
  + Address space (stack/heap)
  + Registers
  + File descriptors
* Recall scheduling model:
  + Only a few processes get to run at a time
  + OS is responsible for scheduling (determines how long each process can run)
    - Once time expires, OS will perform context switch (take process off CPU, put new one on)
  + Operating System (OS) : Software that lies above hardware and is responsible for abstracting its behavior to end users
    - Ex: writing files ( User doesn’t have to create inode and track disk blocks, OS does this automatically)
  + Communication with signals
    - Signal: A notification of sorts that some exceptional event has occurred (and has to be dealt with)
      * Ex. SIGSEV
    - Signal handler: allows user to specify a function to run when a given signal is received by a process

Process Creation

* fork() is the primary way to do this (a system call)
  + All fork() does is create an exact copy of the process that calls fork()
  + Child starts execution right after fork (practically speaking)
  + Copies don’t share:
    - PID (PPID as well)
    - Address space
  + Child won’t call fork (why?)
    - Wouldn’t make sense (infinite recursion)
    - Child starts execution (roughly) when fork is about to return
      * Allows fork to specify specific return values for parent and child
  + When fork returns, how do I know if I’m the child or I’m the parent?
    - Child returns 0; parent returns the PID of the child
* How can I made child run some other program other than parent process?
  + Exec functions: given path to executable and a list of arguments, run that program in the current process
  + What does exec return? NOTHING!
    - Intuition: If I’m running a new program, I do not want exec to return, I just want to run the specified program
  + Pseudocode for new processes:

int PID = fork(..);

if (PID == 0) {

execvp(\*desired program and args\*);

}

* + How do I know which processes are currently running?
    - Ps: utility that shows running processes
    - Top: live look at process resource consumption (memory, CPU time, % of CPU time, etc)
  + Aside: if cloning/forking is the only way to create a new process, where does the first process come from?
    - The scheduler (why? Need to schedule processes of course)
    - Init: master/original process
      * Bootstrapped by the OS (along with scheduler) on startup
  + Parent responsible for the child (once child is done executing, parent needs to know
    - Wait() -> must be done by the parent so that the appropriate clean up can occur
      * If not, then the child is a Zombie
        + What’s so bad about a zombie?

The process is DONE but it’s data persists and takes up memory (address space)

* + - * Orphan : when the parent terminates before the child while child is still running
        + Child becomes an orphan

Reparented -> init is the child’s parent

\*\*\* Still a problem… because init never stops running and doesn’t wait on children

Solution: “reaper” process, looks for terminated orphans and uses kill on them to terminate them

Thread

* “lightweight” process: a thread of execution that exists in the context of a process and shares a heap with other threads (in reality, each thread has a private stack).
* Created in context of a process using pthread\_create(\*\*) (specify a function pointer and a struct of arguments to run the function with).
* Why use these?
  + Shared address space: allows multiple execution context to modify the same data
    - Can partition working data among threads to speed up execution
  + Context switching with processes is VERY expensive (think of what needs to happen)
    - Threads get context switched, but bc they share address spaces, it is not nearly as expensive
  + Gives us powerful sense of parallelism
* What are some issues?
  + Sharing memory: consistency issues with data
    - Ex: t1 writes x = 5 and expects to see that value, but before it sees it, t2 comes along and writes x = 7
      * In real life, imagine if deposit to your account gets wiped out by an expense you made because of competing threads modifying the same data
    - Solution to these problems: Mutual Exclusion