CMSC 216
Introduction to Computer Systems
Lecture 15
Process Control
Administrivia

• Read Sections 8.2-8.5, on process control
Sections 8.2-8.5, Bryant and O'Hallaron

**PROCESS CONTROL**
Processes

• Definition: "an instance of a program in execution"

• A process provides a context for the executing program: current memory state, open files, register contents, PC, environment variables

• When you type in the name of a program in the shell, the shell creates a new process in which that program will run
Multitasking

• Even though it appears to us that many programs run at the same time, in reality, only one process is able to use a CPU at a time
• The CPU is divided between all the currently running processes by the OS, which forces processes to be suspended
• To each process, it appears as if the process has sole control of the CPU, and logical flow of one process isn't affected by others (except for IPC)
Multitasking example

Time

A
B
C
D
A
C

CMSC 216 - Wood, Sussman, Herman, Plane
User and kernel modes

• Some instructions are restricted (or *privileged*), so that only the OS can execute them
  – e.g., halting the CPU, performing I/O, creating processes

• When a process needs to do one of these things, the process must enter kernel mode

• Normally, processes are in user mode
User and kernel modes, cont.

- When in user mode, trying to perform a privileged instruction or access kernel-reserved memory results in a fatal protection fault.
- Should a process need to do one of these things, it needs to use the system call interface to access these instructions/areas of memory.
Implementation of multitasking

• To move processes in and out of the CPU, the kernel needs to maintain the context (execution state) of each process

• Data structures used to maintain context:
  – page table, stores processes' address spaces
  – process table, stores information about each process
  – file table, lists which files are opened by each process
Context switching

• When a process is to be moved out of the CPU, the context is saved and the context of the incoming process is restored
• This is how the kernel schedules processes in the CPU
• Processes can be switched out whenever the kernel deems it necessary or appropriate
  – when a process is waiting for I/O
  – when a process has been using the CPU for a long time
System calls

• "The system call is the fundamental interface between an application and the Linux kernel."
  -- the `syscalls` manual page

• System calls include functions to do things like
  – open/close/create/delete files
  – read from/write to files
  – create processes
  – read the system clock
Error handling

• When a system call fails, it sets the global integer 
  `errno` to a specific number to indicate what went 
  wrong, and often returns -1

• When you call any system call, you need to check to 
  see if it failed

• For example, if the `fork()` function fails, it returns -1:
  ```c
  if ( ((pid = fork()) < 0) ) {
    perror("fork error");
    exit(EX_OSERR);
  }
  ```

• The book defines extra functions (e.g. `Fork()`) that 
  call the corresponding existing function but do error 
  handling, as above, to save on code writing
The `err.h` functions

- The header file `<err.h>` provides access to prototypes of functions that combine error reporting and exits:
  - `void err(int exit_code, const char *fmt, ...);`
    - equivalent to:
      ```c
      fprintf(stderr, fmt, ...);
      fprintf(stderr, ": %s\n", strerror(errno));
      exit(exit_code);
      ```
  - `void errx(int exit_code, const char *fmt, ...);`
    - equivalent to:
      ```c
      fprintf(stderr, fmt, ...);
      exit(exit_code);
      ```
- Non-exiting functions `warn()` and `warnx()` also exist
- Information on these can be found via "`man errx`"
- Can be used to reduce error-checking code while maintaining flexibility in handling
Process life cycle

• Every process goes through several states (the exact type/number vary by OS)

• Typical states:
  – new: process has just been created
  – running: process is executing on the CPU
  – ready: process is waiting to be run
  – waiting: process is waiting for an event (e.g., I/O, signal)
  – terminated: process is finished executing
Process state transitions

- New
  - Admitted
  - Signal
  - Exit
- Ready
  - Signal
  - Interrupt (preemption)
  - Dispatch to CPU
- Waiting
  - I/O finish or event occurs
  - I/O request or event wait
- Terminated
- Running
Process IDs

• Every active process has a unique process ID, which can be obtained via \texttt{getpid()}

• The process ID of the current process' parent (the process which created the current process) can be obtained via \texttt{getppid()}

• Function prototypes:

  \begin{verbatim}
  #include <unistd.h>
  #include <sys/types.h>

  pid_t getpid(void);
  pid_t getppid(void);
  \end{verbatim}
Creating new processes

• In UNIX, a new process is created by an existing process, making a parent-child relationship between the two processes

• The system call to do this is `fork()`; creates a new copy of the parent process
  – all variables (the whole address space) are copied
  – point of execution (PC) is copied
  – file table information is copied
The `fork()` function

- Prototype:
  
  ```c
  #include <unistd.h>
  #include <sys/types.h>
  pid_t fork(void);
  ```

- Returns *twice*, in BOTH parent and child
  - -1: error occurred
    - generally due to process table being full or resource limit reached
  - 0: returned to child process
  - > 0: returns pid of child process to parent
fork() example

/* #include statements omitted */

int main() {
    int var = 313;
    pid_t child_pid;
    if ((child_pid = fork()) < 0)
        err(EX_OSERR, "fork error");
    if (child_pid) { /* parent code */
        printf("Parent pid = %d; my child has pid = %d\n", 
               getpid(), child_pid);
        var++;
        printf("Var in parent = %d\n", var);
    }
    else { /* child code */
        printf("Child pid = %d; my parent has pid = %d\n", 
                getpid(), getppid());
        var--;
        printf("Var in child = %d\n", var);
    }
    return 0;
}
Execution order after a `fork()`

- The previous example's output on one machine was:
  ```
  Child pid = 18532; my parent has pid = 18531
  Var in child = 312
  Parent pid = 18531; my child has pid = 18532
  Var in parent = 314
  ```

- On Grace, it was:
  ```
  Parent pid = 726; my child has pid = 727
  Var in parent = 314
  Child pid = 727; my parent has pid = 726
  Var in child = 312
  ```

- It could even be:
  ```
  Parent pid = 23892; my child has pid = 23894
  Child pid = 23894; my parent has pid = 23892
  Var in child = 312
  Var in parent = 314
  ```

- Print order **within** a process is (usually) determinate
- Print order **between** processes is not
fork() semantics

• Some things inherited by a child from its parent process:
  – process credentials: user and group ID (UIDs and GIDs in UNIX terminology)
  – environment
  – a copy of the parent's memory contents, including program code, runtime stack, and heap
  – open file descriptors – FILE *’s from fopen(). The current file position is also shared between the parent and child, which can cause file consistency issues.
  – signal handling settings (a UNIX way of handling events external to the process, from the operating system or another program)
fork() semantics, con't.

- Some things inherited by a child from its parent process, con't:
  - current working directory (set with `cd`, viewed with `pwd`
  - root directory
  - resource limits (that can be set and viewed with the tcsh `limit` command, or `ulimit` in bash)
  - the controlling terminal (the program that controls `stdin`, `stdout`, and `stderr` for the process, which is usually a shell), so the child reads input from and prints output to the same devices that the parent does
  - "nice" value (to determine process priority for scheduling by OS)
fork() semantics, con't.

• Some things that are unique to a child process:
  – its process ID
  – it has a different parent process ID (the parent, not the parent’s parent)
  – it has its own copy of file descriptors and directory streams.
  – its process times are unique to it
  – its resource utilizations are initially set to 0
  – its pending signals are initialized to the empty set
The dangers of `fork()`

- The process table in the kernel can hold only a finite number of processes; what happens if you fill it up?
  ```c
  #include <unistd.h>
  int main() {
      while (1) fork();
  }
  ```
- That is a fork **bomb**, and it is a Very Bad Thing
- Fork bombs can be unintentional; a loop that doesn't terminate correctly can easily cause one
  - many students write them accidentally
- Often, it can require sysadmin intervention (e.g., reboot, killing all user processes)
- Lesson: Be careful when using `fork()` in a loop