15-213/18-243
Intro to Computer Systems

by btan
with reference to Spring 10's slides

News

- Cachelab due tomorrow 11:59pm
- Proclab out tomorrow
- Today’s recitation will be on Process Control and Signal Handling

Processes

- What is a program?
  - Written according to a specification that tells users what it is supposed to do
  - A bunch of data and instructions stored in an executable binary file
  - Stateless since binary file is static

Processes

- What is a process?
  - A running **instance** of a program in execution
  - One of the most profound ideas in CS

- A fundamental abstraction provided by the OS
  - Single thread of execution (linear control flow) ….
  - … until you create more threads (later in the course)
  - **Stateful:**
    - Full set of **private** address space and registers
    - Other state like open file descriptors and etc.

Processes

- Four basic process control functions
  - fork()
  - exec*() and other variants such as execve()
    - But they all fundamentally do the same thing
  - exit()
  - wait()

Standard on all UNIX-based systems

Don’t be confused:
Fork(), **E**xit(), **W**ait() are all wrappers provided by CSAPP

Processes

- fork()
  - Creates or spawns a child process
  - OS creates an exact duplicate of parent’s state:
    - Virtual address space (memory), including heap and stack
    - Registers, except for the return value (%eax/%rax)
    - File descriptors **but files are shared**
  - **Result →** Equal but **separate** state
  - Returns 0 for child process but child’s PID for parent

Processes

- exec*()
  - Replaces the current process’s state and context
  - Provides a way to load and run **another** program
    - Replaces the current running memory image with that of new program
    - Set up stack with arguments and environment variables
    - Start execution at the entry point
  - The newly loaded program’s perspective: as if the previous program has not been run before
  - It is actually a family of functions
    - man 3 exec

Processes

- exit()
  - Terminates the current process
  - OS frees resources such as heap memory and open file descriptors and so on…
  - Reduce to a zombie state =]
    - Must wait to be **reaped** by the parent process (or the **init** process if the parent died)
    - Reaper can inspect the exit status

Processes

- wait()
  - Waits for a child process to change state
  - If a child terminated, the parent “reaps” the child, freeing all resources and getting the exit status
  - Child fully “gone” ☹
  - For details: man 2 wait

Processes (Concurrency)

```
pid_t child_pid = fork();

if (child_pid == 0){
   /* only child comes here */

   printf(“Child!\n”);

   exit(0);
}
else{
   printf(“Parent!\n”);
}
```

- What are the possible output (assuming fork succeeds) ?
  - Child!, Parent!
  - Parent!, Child!
- How to get the child to always print first?

Processes (Concurrency)

```
int status;
pid_t child_pid = fork();

if (child_pid == 0){
   /* only child comes here */

   printf(“Child!\n”);

   exit(0);
}
else{
   waitpid(child_pid, &status, 0);

   printf(“Parent!\n”);
}
```

- Waits til the child has terminated. Parent can inspect exit status of child using ‘status’
  - WEXITSTATUS(status)

- Output always: Child!, Parent!

Processes (Concurrency)

```
int status;
pid_t child_pid = fork();
char* argv[] = {“ls”, “-l”, NULL};
char* env[] = {…, NULL};

if (child_pid == 0){
   /* only child comes here */

   execve(“/bin/ls”, argv, env);

   /* will child reach here? */
}
else{
   waitpid(child_pid, &status, 0);

   … parent continue execution…
}
```

- An example of something useful.
- Why is the first arg “ls”?
- Will child reach here?

Processes

- Four basic States
  - Running
    - Executing instructions on the CPU
    - Number bounded by number of CPU cores
  - Runnable
    - Waiting to be running
  - Blocked
    - Waiting for an event, maybe input from STDIN
    - Not runnable
  - Zombie =]
    - Terminated, not yet reaped

Signals

- Primitive form of interprocess communication
- Notify a process of an event
- Asynchronous with normal execution
- Come in several types
  - man 7 signal
- Sent in various ways
  - Ctrl+C, Ctrl+Z
  - kill()
  - kill utility

Signals

- Handling signals
  - Ignore
  - Catch and run signal handler
  - Terminate, and optionally dump core
- Blocking signals
  - sigprocmask()
- Waiting for signals
  - sigsuspend()
- Can’t modify behavior of SIGKILL and SIGSTOP
- **Non-queuing**

Signals

- Signal handlers
  - Can be installed to run when a signal is received
  - The form is   void handler(int signum){ …. }
  - **Separate** flow of control in the same process
  - Resumes normal flow of control upon returning
  - Can be called **anytime** when the appropriate signal is fired

Signals (Concurrency)

```
….install sigchld handler…

pid_t child_pid = fork();

if (child_pid == 0){
   /* child comes here */

   execve(……);
}
else{

   add_job(child_pid);

}
```

What could happen here?

```
void sigchld_handler(int signum)
{
    int status;

    pid_t child_pid =
      waitpid(-1, &status, WNOHANG);

    if (WIFEXITED(status))
       remove_job(child_pid);
}
```

How to solve this issue?
Block off SIGCHLD signal at the appropriate places. You’d have to think of it yourself.

ProcLab

- A series of puzzles on process control and signal handling
- Correct use of system functions
- Test your understanding of the concepts
- Should not need to write a lot of code
- 5 Style points – Yes, we will **read** your code
- Details in the handout

Q & A

- Thank you