CMSC 216
Introduction to Computer Systems
Lecture 17
Process Control

Jan Plane & Alan Sussman
{jplane,als}@cs.umd.edu

Administrivia

- Project 4 posted, with public tests, and submit server open
- Project 3 secret tests posted soon

Sections 8.2-8.5, Bryant and O'Hallaron

PROCESS CONTROL

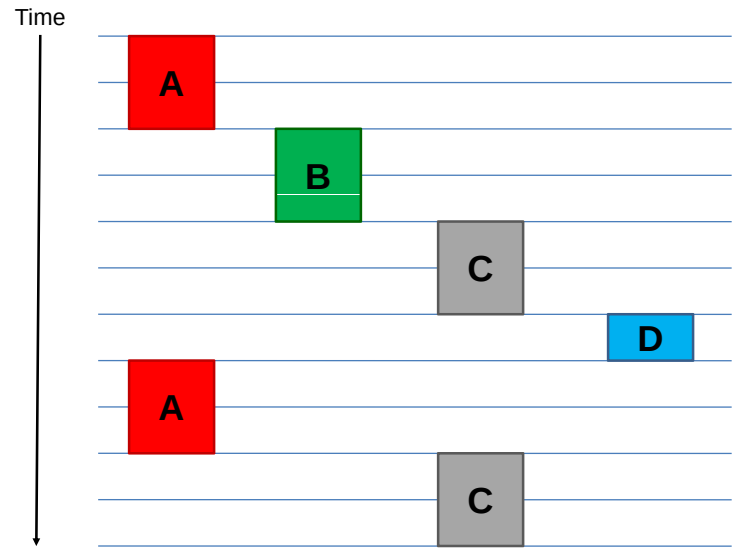
Processes

- 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



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:

```
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:

```
fprintf(stderr, fmt, ...);
fprintf(stderr, ": %s\n", strerror(errno));
exit(exit_code);
```

- **`void errx(int exit_code, const char *fmt, ...);`**
  - equivalent to:

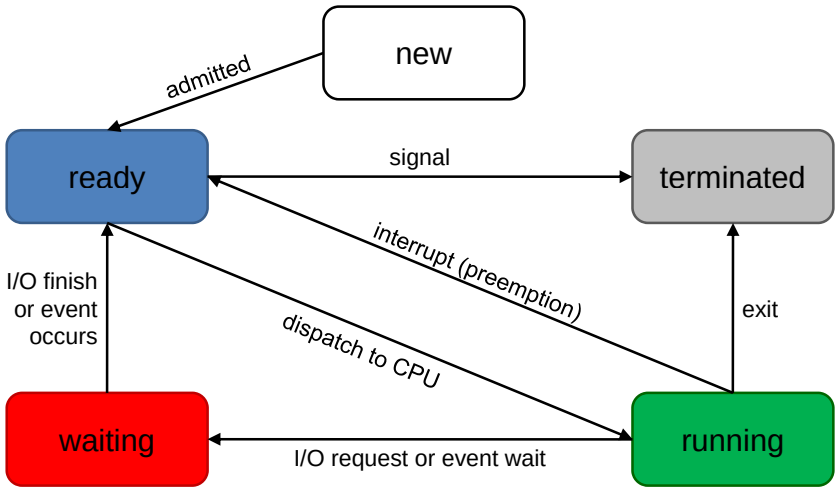
```
fprintf(stderr, fmt, ...);
exit(exit_code);
```

- 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



Process IDs

- Every active process has a unique process ID, which can be obtained via **`getpid()`**
- The process ID of the current process' parent (the process which created the current process) can be obtained via **`getppid()`**
- Function prototypes:

```
#include <unistd.h>
#include <sys/types.h>

pid_t getpid(void);
pid_t getppid(void);
```

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:

```
#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