Announcements

• Next Tuesday (July 7) office hour is postponed:
  • Friday, July 10?
Announcements

• *No* lab period tonight

• There is a lab this week (see Piazza) and it is due Sunday 10pm
Agenda

• The course half way point
• Unix processes
• *fork, wait and exec* system calls
First Half of CSC209

- Shell as user interface
- C language syntax and semantics
Second Half of CSC209

- Mechanisms and abstractions provided by *Unix* operating systems
  - Processes
  - Files
  - Inter-process communication: signals and pipes
  - Network programming with sockets
  - Parallelism and concurrency
- Advanced shell usage
Week 8 lab exercise
Agree or Disagree
Agrée or Disagree

You can only really do three things with the C language programming language:

1) Perform arithmetic
2) Evaluate logical expressions
3) Access memory
Unix Processes

Kerrisk ch. 6
“A process is an instance of an executing program.”

Kerrisk p113
Unix Processes

• Every system has *many* processes currently active:
  • Special operating system processes
  • User process
  • Your personal shell, compiler and test programs...
ps aux
Unix Processes

• Each process is given the illusion of isolation:
  • Exclusive control of the CPU
  • Virtual memory address space
  • Currently open files (including notions of stdout, stdin and stderr)
  • Current working directory
  • Other system resources integral to its execution
Unix Processes

- The OS kernel is an arbitrator that divides physically limited resources out among processes:
  - Memory
  - CPU time
  - Disk
  - Network
  - Access to peripherals, etc.
Unix Processes

• The OS kernel grants each process a slice of the CPU (a short period of time during which the process may run), and then preemptively stops that process and switches to let another one run for a time

• Your processes generally do not even notice that this is happening (the illusion of exclusivity)
System Calls

- The kernel lets processes use system calls in order to make requests:
  - File I/O
  - Certain kinds of memory management (\texttt{mmap}, but not necessarily \texttt{malloc})
  - Process management
  - Communications (networking and IPC)
getpid, getppid - get process identification

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

From the manpage: “`getpid()` returns the process ID of the calling process.

`getppid()` returns the process ID of the parent of the calling process.”
getpid.c
What’s the difference between a system library function like `strncpy` and a system call function like `getpid`?
Process State

The **scheduler** decides which of the ready processes to run.

A process is *ready* if it could use the CPU immediately.

A process is *blocked* if it waiting for an event (I/O, signal).

Only one process can be running on a uniprocessor.
sleep.c
fork system call

Kerrisk ch. 24
The `fork` system call creates a copy of the currently running process, diverging from the point of the system call itself:

- The newly created *child* process receives a return value of 0 from the call

- The original *parent* process receives a the PID of the newly created child
fork

pid=123:

A();
B();
C();

pid = fork();
// pid == ???
D();
E();
F();
fork

pid = 123:

A();
B();
C();
pid = fork();
// pid == ???
D();
E();
F();
fork

pid=123:

A();
B();
C();

pid = fork();
// pid == ???
D();
E();
F();
fork

pid = 123:

A();
B();
C();

pid = fork();
// pid == ???
D();
E();
F();
fork

pid=123:

A();
B();
C();

pid = fork();
// pid == ???
D();
E();
F();
fork

parent (pid=123):

A();
B();
C();

pid = fork();
// pid == ???
D();
E();
F();

child (pid=456):

A();
B();
C();

pid = fork();
// pid == ???
D();
E();
F();
fork

parent (pid=123):
A();
B();
C();
pid = fork();
// pid == 456
D();
E();
F();

child (pid=456):
A();
B();
C();
pid = fork();
// pid == ???
D();
E();
F();
fork

**parent (pid=123):**

```c
A();
B();
C();
pid = fork();
// pid == 456
D();
E();
F();
```

**child (pid=456):**

```c
A();
B();
C();
pid = fork();
// pid == 0
D();
E();
F();
```
fork

parent \((pid=123)\):

A();
B();
C();

\[
\text{pid} = \text{fork}();
\]

// \(\text{pid} == 456\)

D();
E();
F();

child \((pid=456)\):

A();
B();
C();

\[
\text{pid} = \text{fork}();
\]

// \(\text{pid} == 0\)

D();
E();
F();
fork

parent (pid=123):
A();
B();
C();
pid = fork();
// pid == 456
D();
E();
F();

child (pid=456):
A();
B();
C();
pid = fork();
// pid == 0
D();
E();
F();
fork

parent (pid=123):
A();
B();
C();
pid = fork();
// pid == 456
D();
E();
F();

child (pid=456):
A();
B();
C();
pid = fork();
// pid == 0
D();
E();
F();
fork

parent \((\text{pid}=123)\):  

\[
\begin{align*}
A() ; \\
B() ; \\
C() ; \\
pid = \text{fork}() ; \\
// \text{ pid } = 456 \\
D() ; \\
E() ; \\
F() ;
\end{align*}
\]

child \((\text{pid}=456)\):  

\[
\begin{align*}
A() ; \\
B() ; \\
C() ; \\
pid = \text{fork}() ; \\
// \text{ pid } = 0 \\
D() ; \\
E() ; \\
F() ;
\end{align*}
\]
fork

parent (pid=123):
A();
B();
C();
pid = fork();
  // pid == 456
D();
E();
F();

child (pid=456):
A();
B();
C();
pid = fork();
  // pid == 0
D();
E();
F();
fork

parent (\textit{pid}=123):
\begin{verbatim}
A();
B();
C();
pid = fork();
// \textit{pid} == 456
D();
E();
F();
\end{verbatim}

child (\textit{pid}=456):
\begin{verbatim}
A();
B();
C();
pid = fork();
// \textit{pid} == 0
D();
E();
F();
\end{verbatim}
fork

**parent** *(pid=123)*:

```
A();
B();
C();
pid = fork();
// pid == 456
D();
E();
F();
```

**child** *(pid=456)*:

```
A();
B();
C();
pid = fork();
// pid == 0
D();
E();
F();
```
fork

**parent (pid=123):**

```c
A();
B();
C();
pid = fork();
// pid == 456
D();
E();
F();
```

**child (pid=456):**

```c
A();
B();
C();
pid = fork();
// pid == 0
D();
E();
F();
```
fork

parent (pid=123):
A();
B();
C();
pid = fork();
// pid == 456
D();
E();
F();

child (pid=456):
A();
B();
C();
pid = fork();
// pid == 0
D();
E();
F();
fork.c
Fork can and will fail (i.e. return -1) if your user account has created too many processes, or if a system wide limit has been reached.
fork — what’s initially the same between parent and child?

- Properties of parent *inherited* by child:
  - UID, GID
  - Controlling terminal
  - Current working directory and notion of root directory
  - Signal mask, environment, resource limits
  - Shared memory (SHM) segments
fork — what’s changes between parent and child?

- Differences between parent and child
  - PID, PPID
  - Return value from `fork()`
  - Pending *alarms* cleared for child
  - Pending *signals* are cleared for child
Process Termination

Kerrisk ch. 25
Process Termination

• A process terminates when either it explicitly calls `exit(int status)` or implicitly when it returns from `main` with a status code.

• Status code of 0 indicates success (or, the absence of failure)

• Anything else indicates failure

• The Bash shell stores the exit status code of the last process run in a special variable named `$?`. 
exitstatus.c
Process Termination

- Every normal process is the child process of some parent process

- A terminating process sends its parent a SIGCHLD signal and waits for its parent to accept its exit code
What happens if the parent *exits* before the child?
Orphaned Processes

• Any process whose parent terminates before it does will become orphaned, and its parent process becomes PID 1 (the init process, which is the first process in the entire system)
forkorphan.c
How does a parent process \textit{wait} for its child to exit before itself terminating?
wait - wait for child process to change state

\[ \textit{pid_t \ wait(int *status)}; \]

- A process that calls \texttt{wait()} can:
  - \textit{block} (if all of its children are still running)
  - return immediately with the termination status of a child (if a child has terminated and is waiting for its termination status to be fetched)
  - return immediately with an error (if it doesn’t have any child processes.)
wait.c
Zombies

• A *zombie* process:
  • a process that is “waiting” for its parent to accept its exit status code
  • a parent accepts a child’s status code by executing `wait()`
  • shows up as Z in `ps -a`
• A terminating process may be a (multiple) parent; the kernel ensures all of its children are orphaned and adopted by *init*
zombie.c
wait and waitpid

- **wait()** can
  - block
  - return with termination status
  - return with error
- If there is more than one child, **wait()** returns on termination of *any* of its children
- **waitpid()** can be used to wait for a *specific* child PID
  - Also has an option to block or not to block
pid_t waitpid(pid_t pid,
  int *status,
  int options);

if pid == -1:
  Wait for any child (otherwise wait only for that specific child pid)
if option == WNOHANG:
  Return immediately if there is no child to wait for (i.e. do not block)
if option == 0:
  Do wait (block) until there is a child to deal with

wait(&status) is equivalent to waitpid(-1, &status, 0)
waitpid.c
waitmany.c
and
kill(1)
fork is the only way to create new processes

... how do we ever run existing programs then?
exec

Kerrisk ch. 27
**exec** - replace the currently running process

- A family of system calls with several different variations

- Replaces the program that the process is currently running with another

- On success, `exec` will *never* return (because success means another program is now running in your place), and on failure will return `-1`
.

```
/example (pid=123):
...
exec*("/bin/ls");
// Never run...
```
```bash
./example (pid=123):
...
xexec*("/bin/ls");
// Never run...
```

```bash
/bin/ls (pid=123):
...
```

```bash
... code for /bin/ls ...
```
Properties of `exec`

- New process *inherits* from calling process:
  - *PID* and *PPID*
  - Real UID, GID
  - Controlling terminal
  - CWD, root directory, resource limits
  - Pending signals
  - Pending alarms
Variations of exec

```c
int execve(const char *filename, char *const argv[], char *const envp[]);
int execv(const char *path, char *const argv[]);
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
            char *const envp[]);
```
Variations: `execv`

```c
int execv(const char *path,
           char *const argv[]);
```

Exec the binary executable located at `path`, passing in the given `argv` array (which must be NULL terminated)
execv.c
Variations: exec*p*

```c
int execvp(const char *file, char *const argv[]);
int execlp(const char *file, const char *arg, ...);
int execvpe(const char *file, char *const argv[],
              char *const envp[]);
```

Use the **PATH** environment variable to search for executables with the name specified in `file`
execvp.c
Variations: **exec**1*

```c
int exec1(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ..., char * const envp[]);
```

Uses C *variadic* functions (which allow a variable number of parameters) to let you specify the contents of `argv` (must have signal the end with an explicit `NULL`).
Variations: exec*e

\[ \text{int execve(const char *filename, char *const argv[], char *const envp[]);} \]

\[ \text{int execvpe(const char *file, char *const argv[], char *const envp[]);} \]

\[ \text{int execl(const char *path, const char *arg, ..., char * const envp[]);} \]

Specify the environment (envp array) to exec
the program in.
forkexec.c
How a shell works

- **PID 123**
  - Process running shell
  - Parent
    - PID 123
      - Process running shell

  - fork()

- Child
  - Process running shell
  - PID 456

  - exec()

  - Child
    - Process running program
      - PID 456

  - exit()

- Parent
  - Process running shell
  - PID 123

  - waitpid()

  - signal

- Child
  - Process terminated
    - PID 456

- Parent
  - Process running shell
  - PID 123
fork is the only way to create new processes

exec is the only way to run existing programs
Midterms