CS 261
Fall 2016

Mike Lam, Professor

Signals

Processes and shells

- job – shell-level abstraction for a group of processes
  - All spawned as a result of the same command
- Foreground vs. background jobs
  - A single foreground job (interactive I/O)
  - Zero or more background jobs
  - Use '&' to start something in the background
    - Ex: "`./my_prog &`"
  - Use **CTRL-Z** to send foreground job to background
  - Use **CTRL-C** to interrupt the foreground job
  - `fg`: promote background job to foreground

Extra demos

- `pmap` and `strace`
  - See effects of ASLR and syscalls

Signals

- Signal: abstraction for exceptional control flow
  - A standard, clean way to handle exceptions
  - Low-level details do not matter
- Signals are sent and received
  - Kernel sends a signal when it detects an exception
  - Processes can also send each other signals
  - The destination process may ignore the signal, terminate, or catch the signal w/ a signal handler
- `man signal` for complete guide ("`kill -l`" for short list)
  - We'll just learn the basics today

Important signals

- `SIGINT` (#2) – interrupt from keyboard (CTRL-C)
- `SIGABRT` (#6) – abort() function was called
- `SIGBUS` (#7) – I/O bus error
- `SIGFPE` (#8) – floating-point exception
- `SIGKILL` (#9) – kill process
- `SIGSEGV` (#11) – segmentation fault
- `SIGALRM` (#14) – interval timer; set with `alarm`( )
- `SIGTERM` (#15) – terminate process (softer than SIGKILL)
- `SIGCHILD` (#17) – a child process has terminated
- `SIGUSR1` / `SIGUSR2` – custom signals

Handling signals in C

- `#include <signal.h>`
- `signal()` / `sigaction()`: install signal handler
  - Parameters:
    - `signum` – signal number
    - `handler` – new action
      - `SIG_IGN` – ignore
      - `SIG_DFL` – restore default
      - otherwise: the address of a signal handler function (i.e., a function pointer)
  - Signal handlers must take an int (the signal number) and return void
  - `sigaction` is more portable but also more complex
- `raise()` / `kill()`: send a signal
  - Former sends to current process, latter sends to a specific pid

Signal example (SIGALRM)

```
void handler (int sig)
{
    write(1, "Signal!\n", 9);
}

int main ()
{
    signal(SIGALRM, handler);       // install signal handler

    alarm(1);                       // set alarm

    while (1) { }                   // infinite loop

    return 0;
}
```

Safe signal handlers

- Most important
  - Keep it simple
  - Only use async-signal-safe functions
    - See `man signal` for a list
- Less important
  - Save/restore "errno" global variable
  - Declare global variables as "volatile"
  - Declare global flags using atomic type

Signal example (SIGINT)

```
#define BUFSIZE 1024

void handler (int sig)
{
    write(1, "Signal!\n", 9);
}

int main ()
{
    char buf[BUFSIZE];
    int i = 0;

    // install signal handler
    signal(SIGINT, handler);

    // read / print loop
    while (fgets(buf, BUFSIZE, stdin) != 0) {
        printf("Line %d: %s", i++, buf);
    }

    return 0;
}
```

Signal example (SIGSEGV)

```
void handler (int sig)
{
    write(1, "OK\n", 4);
    exit(0);
}

int main ()
{
    int *p = 0;

    signal(SIGSEGV, handler);       // install segfault handler

    int v = *p;                     // null pointer dereference

    printf("Here!\n");              // won't get here
    return v;
}
```

Signal example (raising signals)

```
void handler (int sig)
{
    write(1, "Hello!\n", 8);
}

int main ()
{
    signal(SIGUSR1, handler);
    raise(SIGUSR1);
    raise(SIGSEGV);
    return 0;
}
```

Signals in debuggers

- By default, signals are caught by gdb
  - Most cause execution to be paused for debugging
    - E.g., SIGINT (CTRL-C)
  - Some are also passed through to the user program
    - Not SIGINT, but SIGSEGV and others
- GDB allows you to change this behavior
  - `info signal` – show current behavior
  - `handle <signal> <option>` – change behavior
    - `stop/nostop`: pause the program?
    - `print/noprint`: notify the user w/ a message?
    - `pass/nopass`: pass signal through to program?

Parallel computation w/ processes

- Spawn multiple processes
  - Use a shell script or multiple `fork()` calls
  - Processes run concurrently
    - If CPU is single-core, they multitask on that core
    - If CPU is multi-core, they execute in parallel
- Communicate via signals, files, or sockets
  - No shared memory space
  - Use message-passing to coordinate computation
    - More about this in CS 361 (and potentially CS 470)
  - Next week we'll see a different approach
    - Shared memory: multiple threads share a single address space
    - Potentially faster but more dangerous