Operating System Concepts

Lecture 6: Signals

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

Other system calls for process control

- OS must include calls to enable special control of a process:
  - Priority manipulation:
    - the `nice(incr)` system call adjusts the process priority by adding `incr` to its nice value
      - lower nice values have higher scheduling priority
      - a process could be **"nice"** and reduce its share of the CPU by adjusting its nice value
  - Debugging support:
    - the `ptrace( )` system call allows a process to be put under control of another process by having its system calls intercepted; very useful for breakpoint debugging
    - the other process can check the arguments of the system call made by the process being traced, set breakpoints, examine registers, etc.
  - Alarms and time:
    - the `sleep( )` system call puts a process on a timer queue waiting for some number of seconds, supporting alarm functionality

Process termination in UNIX systems

- the `kill` **system call** sends a signal to a process or process group based on the specified PID
- the `kill` **command** sends a `SIGTERM` signal by default
- the `killall` **command** sends an arbitrary signal to processes based on process name

Process monitoring in UNIX systems

- `ps` displays information about a selection of the active processes
  - `ps -el` lists complete information about all processes that are currently active in the system
  - `ps -u [username]` lists all processes created by a specific user
- `top` provides a dynamic real-time view of a running system (repetitive update on active processes)
- `pstree` displays a tree of processes

Shell

- Acts as a process control system
  - allowing programmers to create and manage a set of processes to do some tasks
  - Windows, Linux, MacOS have their own shells
- When you log in to a machine running UNIX, you create a shell process
- Every command launched in the shell is a child process of the shell process (an implicit `fork( )` and `execve( )` pair)
- The separation of `fork( )` and `execve( )` enables features like input/output redirection, pipes, etc.
  - the shell runs code after the call to `fork( )` and before the call to `execve( )`

Today's class

- Signals
  - Generation
  - Disposition
  - Blocking
- Interprocess communication (IPC)

Signals are software interrupts

- Signals provide a way of handling **asynchronous events** and sometimes **synchronous events** (e.g. divide by zero, segmentation fault)
  - they are small messages sent to a process, hence can be viewed as a form of IPC
  - often sent by the kernel, but can be sent from other processes too
- Mac OS X 10.6.8 and Linux 3.2.0 each support 31 different signals
- Every signal has a name that begins with SIG
  - names are defined by positive integer constants in `<signal.h>`
    - e.g. 9 is `SIGKILL`: kill; 11 is `SIGSEGV`: segmentation fault (e.g. dereferencing a null pointer), 8 is `SIGFPE`: erroneous arithmetic operation (e.g. divide by zero)
  - use `kill -l` to get a list of signals (this is architecture dependent)
- every signal has a default action associated with it; this action is performed once the signal is delivered unless a custom handler was installed
  - default actions: (a) terminate, (b) terminate with a **core dump**, (c) ignore, and (d) stop

Terminology

- We say that a signal is
  - posted (or generated or sent) if the event that causes it has occurred
  - delivered (or caught) if the action associated with it is taken
    - this action is referred to as **signal disposition**
  - pending if it was posted but not yet delivered
    - intermediate state between generated and delivered
    - signals will be pending if the target process blocks them
  - blocked if the target process does not want it delivered
    - the target process asked (using **signal mask**) the kernel to block that signal
- A signal can be process-directed or thread-directed (such as `SIGSEGV` and `SIGFPE`)
  - a process-directed signal may be delivered to any one of the threads that does not currently have the signal blocked

Signal generation

- Using a keystroke combination in shell
  - Ctrl-C causes `SIGINT` (interrupt) to be generated and sent to the foreground process
  - Ctrl-Z causes `SIGTSTP` (stop) to be generated and sent to the foreground process (note `SIGSTOP` ≠ `SIGTSTP`)
  - Ctrl-\ causes `SIGQUIT` (quit) to be generated
- OS kernel wants to notify a process that its execution has led to a hardware exception (e.g., divide by zero, floating-point overflow, segmentation fault)
- The `kill` command or the `kill(pid_t pid, int sig)` system call is used
- A software condition occurs, e.g.
  - `SIGURG` (generated when out-of-band data arrives over a network connection)
  - `SIGPIPE` (generated when a process writes to a pipe that has no reader)
  - `SIGALRM` (generated when a timer set by the `alarm` function expires)

POSIX.1 reliable-signal

- A process can send a signal to another process or a group of processes
  - using the `kill( )` system call
  - **only if it has permission:** the real or effective user ID of the receiver is the same as that of the sender
- A process can send a signal with accompanying **data** to another process (just like IPC)
  - using the `sigqueue( )` function
- A process can send itself a signal using the `raise( )` function
  - similar to `kill(getpid(), sig)`
- Waiting for a signal
  - the `pause( )` system call puts a process to sleep until any signal is caught; this signal can be generated by the `alarm( )` function
    - it returns only when a signal was caught and the signal-catching function returned

POSIX.1 reliable-signal

- Signal dispositions
    1. `SIG_DFL`: let the default action happens (in most cases terminate process or terminate process with a core dump; in some cases ignore)
    2. `SIG_IGN`: ignore the signal (except `SIGKILL` and `SIGSTOP` which can never be ignored as they are surefire ways of terminating/stopping a process)
    3. Address of a user-defined signal handler (a function that takes a single integer argument and returns void); kernel catches it by invoking this function
        - `SIGKILL` and `SIGSTOP` cannot be caught
        - a signal handler can return or call exit; if it returns, the normal sequence of instructions are executed
- System calls can be interrupted by a signal
    - some of them are automatically restarted, e.g., `ioctl`, `read`, `readv`, `write`, `writev`, `wait`, and `waitpid`

POSIX signal environment

- Examining or modifying the action associated with a particular signal (except for `SIGKILL` and `SIGSTOP`)
  - `sigaction( )` supersedes `signal( )` from earlier releases of the UNIX System
- Blocking signals in a signal set from delivery to a process
  - `sigprocmask( )`
- Manipulating the signal set (a bit vector)
  - `sigemptyset( ), sigfillset( ) , sigaddset( ) , sigdelset( ), sigismember( )`
  - when a signal is caught and the handler is entered, the current signal is automatically added to the signal mask of the process; this is to prevent subsequent occurrences of that signal from interrupting the signal handler
- Returning the set of signals that are blocked from delivery and currently pending for the calling process
  - `sigpending( )`

Example of the sigaction( ) system call

```c
struct sigaction {
               void     (*sa_handler)(int);
               void     (*sa_sigaction)(int, siginfo_t *, void *);
               sigset_t   sa_mask;
               int        sa_flags;
               void     (*sa_restorer)(void);
};
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <signal.h>

void signal_callback_handler(int signum) {
   printf("Caught the signal!\n");

   // uncomment the next line to break the loop when signal is received
   // exit(1);
}

int main() {
   struct sigaction sa;
   sa.sa_flags = 0;
   sigemptyset(&sa.sa_mask);
   sa.sa_handler = signal_callback_handler;
   sigaction(SIGINT, &sa, NULL);  // we are not interested in the old disposition

   // sigaction(SIGTSTP, &sa, NULL);
   while (1) {}
}
```

Process creation and signals

- When a process calls `fork`
  - child inherits parent's signal dispositions and signal mask
  - child starts off with a copy of parent's memory image, so signal-handlers are accessible
- When a process calls `exec`
  - the disposition of any signal being caught (not ignored) changes to its default action
  - the status of all other signals is left alone
  - the signal mask is preserved across the `exec` system call

Process state changes and signals

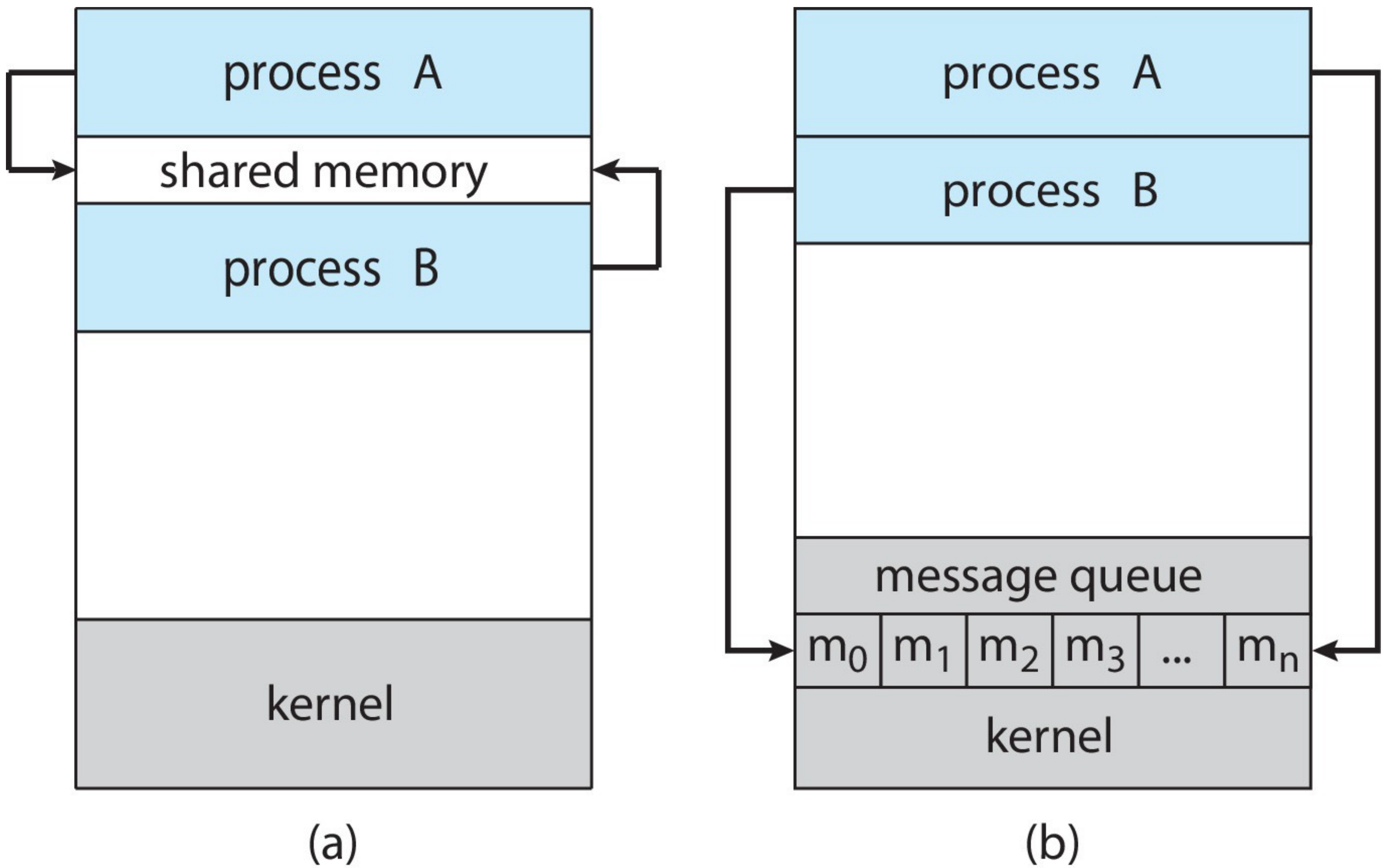
- When a process changes state (terminates, stops, or resumes), the kernel sends `SIGCHLD` to its parent
  - by default, this signal is ignored unless the parent installs a hander for it
  - the handler can call `waitpid(-1, &status, options)` with appropriate options (i.e. flags bitwise-or'ed together) to understand which process has changed state and record its new state
    - see the man page of `waitpid` for the list of options
  - **note:** if multiple signals come in at the same time, the signal handler is invoked only once, so `waitpid` should be called in a loop inside the handler as multiple processed might have changed state

Interprocess Communication

Cooperating processes

- Any two processes are either independent or cooperating
- Cooperating processes work with each other to accomplish a single task
  - improve performance
    - overlapping activities or performing work in parallel
  - improve program structure
    - each cooperating process is smaller than a single monolithic program
- They may need to share information
  - OS makes it possible!
  - shared memory vs. message passing approaches

Models of interprocess communication



Homework

- See examples posted on Canvas