Signals

Benjamin Brewster

Slides adapted from Jon Herlocker, OSU

Inter-Process Communication (IPC)

- This lecture is about communications between processes that don’t involve:
  - files
  - RAM

Event Notification

- When a user process wants to contact the kernel, it uses a system call
- But how does the kernel initiate contact with a user process?
  - There are certain events that occur for which the kernel needs to notify a user process
- The answer:
  - Signals!

Signals

- Have many similarities to "exceptions" in Java and C++
  - Although they are much less flexible than exceptions
- There are a fixed set of signals
  - You cannot create your own signals, though the *meaning* of some signals is up to you

Uses for Signals

- Notifications from the Kernel
  - Process has done something wrong (bus error, segmentation fault, etc)
  - A timer has expired (an alarm)
  - A child has completed executing
  - An event associated with the terminal has occurred (control-C, process trying to read from the background, etc)
  - The process on the other end of a communication link has gone away

Uses for Signals

- User process to user process notifications
  - Suspend and resume execution of process
  - Terminate process
  - 2 user-defined signals

Notification of Wrongdoing

- Many signals result from the process doing something it is not supposed to
  - Trying to read or write to an invalid memory address (bus error = SIGBUS; segmentation fault = SIGSEGV)
  - Floating-point error - SIGFPE
  - Trying to execute an illegal machine instruction - SIGILL
  - Executing an illegal system call - SIGSYS
- Why is a signal needed?
  - Gives the process a chance to:
    - clean up before being terminated
    - ignore the event if appropriate

Timers!

- If you want to wait a specified period of time…
  - You *can* do a busy wait which will consume the cpu continuously while accomplishing nothing (BAD)
  - Or you can tell the kernel that you want to be notified when after a certain amount of time passes
- To set a timer in UNIX
  - Call the alarm() or ualarm() functions
  - After the time you specify has passed, the kernel will send your process a SIGALARM signal
- This is how sleep() works:
  - sleep calls alarm()
  - sleep calls pause() - which puts process into waiting state
  - when SIGALARM is received - sleep returns

Child has completed

- Normally, the wait() and waitpid() calls will suspend the process until a child has completed executing
- Using the signal allows a parent process to do other work instead of going to sleep and be notified when a child completes
- Then when the SIGCHILD is received, the process can call wait() or waitpid()

Terminating Processes

- Several signals (SIGTERM, SIGKILL) provide a way to indicate that other processes should terminate
- Sent from user process to another user process
- SIGTERM can be caught by the receiver, and potentially ignored
- SIGKILL cannot be caught - guaranteed to end a process

kill

- The first parameter is the signal to send
- The second parameter is the pid of the process being signaled
- The given pid affects who the signal is sent to:
  - If pid > 0, then the signal will be sent to the process with a process id of pid
  - If pid == 0, then the signal is sent to all processes in the same process group as the sender
  - more trickiness for pid < 0
- kill *–signal pid*
  - `kill -TERM 1234`
  - `kill -15 1234`
  - `kill - KILL 1234`
  - `kill -9 1234`

User-defined Signals

- UNIX has reserved two signals for user-defined use:
  - SIGUSR1
  - SIGUSR2
- They have no special meaning to the kernel, but you can use them to signal between processes
- The author of both the sending process and receiving process must interpret the meaning of SIGUSR1 and SIGUSR2

Abnormal Termination

- Some signals result in an "abnormal termination"
- This occurs if the process has done something it shouldn't - i.e. corrupt or incorrect code
- When an abnormal termination occurs, a memory core dump is created
  - The file "core" contains the contents of all variables, hardware registers, and the kernel process info from the kernel at the time the termination occurred
  - The "core" file can often be used after the fact to identify what went wrong

Receiving Signals

- A process can decide how it wants to handle signals that are received
  - A process can define a different reaction to each kind of signal
- There are three options to handling each type of signal:
  1. Ignore it - continue executing as if it had never occurred
  2. Take the default action. This is different for each type of signal
  3. Specify a function that should be called when a type of signal arrives

Signal Handling API

- Functions for managing "signal sets":
  - segemptyset(), sigfillset(), sigaddset(), sigdelset()
- Set a signal handler function:
  - sigaction()
  - A signal handler function that you write can run whenever a specified signal occurs

Signal Sets

- A *signal set* is simply a list of signal types
- A signal set is defined using the special type sigset_t defined in <signal.h>

Signal Sets

- To define a signal set
  - `sigset_t my_signal_set;`
- To initialize or reset the signal set to have no signal types:
  - `sigemptyset(&my_signal_set);`
- To initialize or reset the signal set to have all types of signals:
  - `sigfillset(&my_signal_set)`
- To add a single signal type to the set:
  - `sigaddset(&my_signal_set, SIGINT)`
- To remove a single signal type from the set:
  - `sigdelset(&my_signal_set, SIGQUIT)`

`sigaction()`

- `sigaction` registers a signal handling function that you've created for a specified set of signals

sigaction()

- `int sigaction(int signo, struct sigaction *newact, struct sigaction *origact)`
- First parameter is the signal type of interest
- The second parameter is a pointer to a special sigaction structure
  - Describes the action to be taken upon receipt
- The third parameter is a pointer to another sigaction structure
  - The sigaction() function will use this pointer to write out what the setting were before the change was requested

The sigaction structure

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

Pointers to functions in C

- They look complicated, but they are really simple
- In the sigaction structure, it is defined as
  - `void (*sa_handler)(int);`
- The '*' in front of the variable name, and the parentheses indicate that this is a pointer to a function
- The "void" indicates that the function `sa_handler` does not return anything
- The "int" indicates that the pointer should be to a function that has one parameter - an integer
  - This is the types of the args in the argument list for `sa_handler`

Using pointers to functions

```
int myFunction (int inputArg);

int main()
{
   int (*fp)(int) = myFunction;
   printf("%d\n", fp(10));
}

int myFunction (int inputArg)
{
   return inputArg + 1;
}
```

The sigaction structure

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

- The first attribute should be set to one of three values:
  - SIG_DFL - take the default action
  - SIG_IGN - ignore the signal
  - A pointer to a function that should be called when this signal is received

The sigaction structure

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

- The sa_mask attribute includes what signals should be blocked while the signal handler is executing
  - *Blocked* means that the signals are put on hold until your signal handler is done executing
  - You need to pass this a signal set

The sigaction structure

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

- The third attribute of the sigaction provides additional instructions (flags)
  - SA_RESTHAND - resets the signal handler to SIG_DFL (default action) after the first signal has been received and handled
  - SA_SIGINFO - tells the kernel to call the function specified in the fourth attribute (sa_sigaction), instead of the first attribute (sa_handler). More detailed information is passed to this function
  - Make sure that you set sa_flags to 0 if you aren't planning to set any flags

The sigaction structure

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

- sa_sigaction specifies an alternative signal handler function to be called. This attribute will only be used if the SA_SIGINFO flag is set in sa_flags.
- The siginfo_t structure contains information such as which process sent you the signal.
- Most of the time you will use sa_handler and not sa_sigaction

Catching SIGINT

```
#include <signal.h>

void catchint(int signo)
{
   printf("Caught an interrupt: %d\n", signo);
}

main()
{
   struct sigaction act;
   act.sa_handler = catchint;
   act.sa_flags = 0;
   sigfillset(&(act.sa_mask));

   sigaction(SIGINT, &act, NULL);

   while (1)
   {
      printf("sleeping for ten seconds...\n");
      sleep(10000);
   }
   exit(0);
}
```

Ignoring SIGINT

```
#include <signal.h>

main()
{
    struct sigaction act;
    act.sa_handler = SIG_IGN;

    sigaction(SIGINT, &act, NULL);

    while (1)
    {
        printf("sleeping for ten seconds...\n");
        sleep(10000);
    }
    exit(0);
}
```

- What is SIGINT?
- Why would we ignore it?

Temporarily changing a signal handler

- Sometimes you only want to change a signal handler temporarily
- Afterwards you need to change it back to what it used to be
- So how do you save the original signal handler and restore it when you are done?
  - Use the third parameter to sigaction()

```
#include <signal.h>
void catchint(int signo)
{
   printf("Caught an interrupt: %d\n", signo);
}
int main()
{
   struct sigaction restOfTheTime_act, initial_act;
   int counter = 0;
   initial_act.sa_handler = catchint;
   initial_act.sa_flags = 0;
   sigfillset(&(initial_act.sa_mask));
   sigaction(SIGINT, &initial_act, &restOfTheTime_act);
   while (1)
   {
      counter++;
      if (counter == 10)
      {
        printf("resetting the signal action\n");
        sigaction(SIGINT, &restOfTheTime_act, NULL);
      }
      printf("sleeping for ten seconds...\n");
      sleep(10000);
   }
   exit(0);
}
```

Blocking signals

- It is also possible to *block* signals from occurring during your program execution
- Blocking a signal simply means that it is delayed until you unblock the signal
- This could be useful if you have code where it is extremely critical that you don't get interrupted
- Blocking is done using:
  - sigprocmask()
- Not used very frequently

Waiting for a signal

- Sometimes a process has nothing to do, so you consider calling sleep()
- However, you want the process to be able to respond to signals, which it can’t do in sleep()
- To handle this:
  - pause()

Waiting for a signal

- If a signal is ignored, then the `pause()` continues to be in effect
- If a signal causes a termination, `pause()` does nothing (because the process goes away)
- If a signal is caught, the appropriate signal handler function will be called. After the signal handler function is done, `pause()` returns -1 and sets errno to EINTR
  - You could issue another `pause()`, for example

Waiting for a signal

- Why does `pause()` return an error when a signal is caught?
- Because it suspends a process/thread permanently
- A signal coming in signifies some “abnormal” behavior
  - Even though this is the point of `pause()`!

Signals and system calls

- Signals can arrive any time
  - including in the middle of a system call!
- System calls are savvy about signals and prevent data loss and corruption from occuring
  - They also prevent partial actions from happening

Signals and system calls

- How should signals in "long running" system calls be handled
  - For example: `write()`, `read()`, and `pause()`
- Normally, system calls will return an error: EINTR
- You can tell system calls to automatically restart by setting SA_RESTART in the sa_flags variable

Sending signals to yourself

- You can send yourself a specified signal immediately with `raise()`:
  - `int raise(int sig)`
- The `alarm()` function sends your process a SIGALARM signal at a later time
  - `unsigned int alarm(unsigned int seconds);`
  - Note that `alarm()` will return immediately, unlike `sleep()`
  - You can only have one alarm active at any time