Signals

Goals of this Lecture

• Help you learn about:
  • Sending signals
  • Handling signals

  … and thereby …

  • How the OS exposes the occurrence of some exceptions to application processes
  • How application processes can control their behavior in response to those exceptions
Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. Handling Signals
5. Alarms
6. Children and signals
7. Conclusion

Definition of Signal

**Signal**: A notification of an event
- Exception occurs (interrupt, trap, fault, or abort)
- Context switches to OS
- OS sends signal to application process
- When application process regains CPU, default action for that signal executes
  - Can install a **signal handler** to change action
  - (Optionally) Application process resumes where it left off
Examples of Signals

User types Ctrl-c
• Interrupt occurs
• Context switches to OS
• OS sends 2/SIGINT signal to application process
• Default action for 2/SIGINT signal is “terminate”

Process makes illegal memory reference
• Fault occurs
• Context switches to OS
• OS sends 11/SIGSEGV signal to application process
• Default action for 11/SIGSEGV signal is “terminate”

Unix Process Control

Diagram showing states of processes:
- Running Foreground Process
- Running Background Process
- Stopped Background Process
- Non-Existing Process

Commands:
- Ctrl-c
- Ctrl-z
- fg
- kill -2 pid
- command
- command &
- bg
- kill -20 pid
Process Control Implementation

Exactly what happens when you:

• Type Ctrl-c?
  - Keystroke generates interrupt
  - OS handles interrupt
  - OS sends a 2/SIGINT signal

• Type Ctrl-z?
  - Keystroke generates interrupt
  - OS handles interrupt
  - OS sends a 20/SIGTSTP signal

Recall “Exceptions and Processes” lecture

Process Control Implementation (cont.)

Exactly what happens when you:

• Issue a “kill –sig pid” command?
  - kill command executes trap
  - OS handles trap
  - OS sends a sig signal to the process whose id is pid

• Issue a “fg” or “bg” command?
  - fg or bg command executes trap
  - OS handles trap
  - OS sends a 18/SIGCONT signal (and does some other things too!)

Recall “Exceptions and Processes” lecture
Sending Signals via Keystrokes

Three signals can be sent from keyboard:

- **Ctrl-c → 2/SIGINT signal**
  - Default action is “terminate”
- **Ctrl-z → 20/SIGTSTP signal**
  - Default action is “stop until next 18/SIGCONT”
- **Ctrl-\ → 3/SIGQUIT signal**
  - Default action is “terminate”

Sending Signals via Commands

**kill Command**

`kill -signal pid`

- Send a signal of type `signal` to the process with id `pid`
- No signal type name or number specified => sends 15/SIGTERM signal
- Default action for 15/SIGTERM is “terminate”

**Examples**

- `kill -2 1234`
- `kill -SIGINT 1234`
  - Same as pressing Ctrl-c if process 1234 is running in foreground
Sending Signals via Function Calls

**raise()**

```c
int raise(int iSig);
```

- Commands OS to send a signal of type `iSig` to current process
- Returns 0 to indicate success, non-0 to indicate failure

**Example**

```c
int iRet = raise(SIGINT); /* Process commits suicide. */
assert(iRet != 0);        /* Shouldn't get here. */
```

Sending Signals via Function Calls

**kill()**

```c
int kill(pid_t iPid, int iSig);
```

- Sends a `iSig` signal to the process whose id is `iPid`
- Equivalent to `raise(iSig)` when `iPid` is the id of current process
- Editorial: Better function name would be `sendsig()`

**Example**

```c
pid_t iPid = getpid();         /* Process gets its id.*/
int iRet = kill(iPid, SIGINT); /* Process sends itself a
assert(iRet != 0);                SIGINT signal (commits suicide) */
```
Handling Signals

Each signal type has a default action
• For most signal types, default action is “terminate”

A program can *install* a signal handler to change action of (almost) any signal type

Uncatchable Signals

Special cases: A program *cannot* install a signal handler for signals of type:

• 9/SIGKILL
  • Default action is “terminate”

• 19/SIGSTOP
  • Default action is “stop until next 18/SIGCONT”
Installing a Signal Handler

**signal()**

```c
sighandler_t signal(int iSig,
        sighandler_t pfHandler);
```

- Installs function `pfHandler` as the handler for signals of type `iSig`
- `pfHandler` is a function pointer:
  ```c
typedef void (*sighandler_t)(int);
```
- Returns the old handler on success, `SIG_ERR` on error
- After call, `(*pfHandler)` is invoked whenever process receives a signal of type `iSig`

---

Installing a Handler Example 1

**Program testsignal.c:**

```c
#define _GNU_SOURCE /* Use modern handling style */
#include <stdio.h>
#include <assert.h>
#include <signal.h>

static void myHandler(int iSig) {
    printf("In myHandler with argument \d\n", iSig);
}
```

...
Installing a Handler Example 1 (cont.)

Program testsignal.c (cont.):

```c
... int main(void) {
    void (*pfRet)(int);
    pfRet = signal(SIGINT, myHandler);
    assert(pfRet != SIG_ERR);
    printf("Entering an infinite loop\n");
    for (;;) ;
    return 0;
}
```

Installing a Handler Example 2

Program testsignalall.c:

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <assert.h>
#include <signal.h>

static void myHandler(int iSig) {
    printf("In myHandler with argument %d\n", iSig);
}
...```

Installing a Handler Example 2 (cont.)

Program testsignalall.c (cont.):

...  
int main(void) {  
    void (*pfRet)(int);  
    pfRet = signal(SIGHUP, myHandler); /* 1 */  
    pfRet = signal(SIGINT, myHandler); /* 2 */  
    pfRet = signal(SIGQUIT, myHandler); /* 3 */  
    pfRet = signal(SIGILL, myHandler); /* 4 */  
    pfRet = signal(SIGTRAP, myHandler); /* 5 */  
    pfRet = signal(SIGABRT, myHandler); /* 6 */  
    pfRet = signal(SIGBUS, myHandler); /* 7 */  
    pfRet = signal(SIGFPE, myHandler); /* 8 */  
    pfRet = signal(SIGKILL, myHandler); /* 9 */  
    pfRet = signal(SIGKILL, myHandler); /* 9 */  
    pfRet = signal(SIGKILL, myHandler); /* 9 */  

    /* Etc., for every signal. */  
    printf("Entering an infinite loop\n");  
    for (;;)  
        ;  
    return 0;  
}  

This call fails
Installing a Handler Example 3

Program generates lots of temporary data
• Stores the data in a temporary file
• Must delete the file before exiting

```c
...  
int main(void) {
    FILE *psFile;
    psFile = fopen("temp.txt", "w");
    ...
    fclose(psFile);
    remove("temp.txt");
    return 0;
}
```

Example 3 Problem

What if user types Ctrl-c?
• OS sends a 2/SIGINT signal to the process
  • Default action is terminate

Problem: The temporary file is not deleted
• Process terminates before `remove("temp.txt")` is executed

Challenge: Ctrl-c could happen at any time
• Which line of code will be interrupted?

Solution: Install a signal handler
• Define a “clean up” function to delete the file
• Install the function as a signal handler for 2/SIGINT
Example 3 Solution

```c
... static FILE *psFile; /* Must be global. */
static void cleanup(int iSig) {
    fclose(psFile);
    remove("temp.txt");
    exit(0);
}
int main(void) {
    void (*pfRet)(int);
    psFile = fopen("temp.txt", "w");
    pfRet = signal(SIGINT, cleanup);
    ...
    cleanup(0);  /* or raise(SIGINT); */
    return 0;  /* Never get here. */
}
```

SIG_IGN

Predefined value: **SIG_IGN**

Can use as argument to `signal()` to **ignore** signals

```c
int main(void) {
    void (*pfRet)(int);
    pfRet = signal(SIGINT, SIG_IGN);
    assert(pfRet != SIG_ERR);
    ...
}
```

Subsequently, process will ignore 2/SIGINT signals
**SIG_DFL**

Predefined value: **SIG_DFL**

Can use as argument to `signal()` to restore default action

```c
int main(void) {
    void (*pfRet)(int);
    ...
    pfRet = signal(SIGINT, somehandler);
    assert(pfRet != SIG_ERR);
    ...
    pfRet = signal(SIGINT, SIG_DFL);
    assert(pfRet != SIG_ERR);
    ...
}
```

Subsequently, process will handle 2/SIGINT signals using default action for 2/SIGINT signals (“terminate”)

---

**Alarms**

`alarm()`

```c
unsigned int alarm(unsigned int uiSec);
```

- Sends 14/SIGALRM signal after `uiSec` seconds
- Cancels pending alarm if `uiSec` is 0
- Uses **real time**, alias **wall-clock time**
  - Time spent executing other processes counts
  - Time spent waiting for user input counts
- Return value is irrelevant for our purposes

**Used to implement time-outs**
Alarm Example

Program testalarmtimeout.c:

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig)
{
    printf("Sorry. You took too long.\n");
    exit(EXIT_FAILURE);
}
```

Alarm Example (cont.)

Program testalarmtimeout.c (cont.):

```c
int main(void) {
    int i;
    sigset_t sSet;

    /* Make sure SIGALRM signals are not blocked. */
    sigemptyset(&sSet);
    sigaddset(&sSet, SIGALRM);
    sigprocmask(SIG_UNBLOCK, &sSet, NULL);

    ...  
    Safe, but shouldn't be necessary
```
Alarm Example (cont.)

Program testalarmtimeout.c (cont.):

```c
...  
signal(SIGALRM, myHandler);
printf("Enter a number: ");
alarm(5);
scanf("%d", &i);
alarm(0);
printf("You entered the number %d.\n", i);
return 0;
}
```

Handling Child Process Exit

Parent process runs its own code after fork

Foreground process:
- Parent calls `wait()`
- Child process exits, shell handles next command

Background process:
- Shell handles next command
- Child process exits
- When does zombie get harvested?
Solution: Signal Child Exit

Once child exits
- OS sends SIGCHLD to parent process
- Parent's signal handler calls `wait()`

What happens if multiple children exit?
- Call `wait()` too few times → zombies
- Call `wait()` too many times → parent blocks

Solution: `waitpid()` with WNOHANG
- Harvest if exist, return immediately otherwise
- Safe to call within `do..while()` loop

Predefined Signals

List of the predefined signals:

```
$ kill -l
1) SIGHUP   2) SIGINT   3) SIGQUIT   4) SIGILL
5) SIGTRAP  6) SIGABRT  7) SIGBUS    8) SIGFPE
9) SIGKILL 10) SIGUSR1  11) SIGSEGV   12) SIGUR2
13) SIGPIPE 14) SIGALRM 15) SIGTERM   17) SIGCHLD
18) SIGCONT 19) SIGSTOP  20) SIGTSTP   21) SIGTIN
22) SIGTTOU 23) SIGURG   24) SIGXCPU   25) SIGXFSZ
26) SIGVTALRM27) SIGPROF 28) SIGWINCH  29) SIGIO
30) SIGPWR  31) SIGSYS   34) SIGRTMIN+135) SIGRTMIN+5
36) SIGRTMIN+2 37) SIGRTMIN+3 38) SIGRTMIN+4 39) SIGRTMIN+5
40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8 43) SIGRTMIN+9
44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8 57) SIGRTMAX-7 58) SIGRTMAX-6 59) SIGRTMAX-5
60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2 63) SIGRTMAX-1
64) SIGRTMAX
```

See Bryant & O'Hallaron book for default actions, triggering exceptions
Application program can define signals with unused values
Summary

Signals
- A **signal** is an asynchronous event
- Sending signals
  - `raise()` or `kill()` **sends** a signal
- Catching signals
  - `signal()` **installs** a signal handler
  - Most signals are **catchable**
- Beware of **race conditions**
  - `sigprocmask()` **blocks** signals in any **critical section** of code
  - Signals of type x automatically are blocked while handler for type x signals is running

Summary (cont.)

Alarms
- Call `alarm()` to deliver 14/SIGALRM signals in **real/wall-clock time**
- Alarms can be used to implement **time-outs**

Interval Timers
- Call `setitimer()` to deliver 27/SIGPROF signals in **virtual/CPU time**
- Interval timers are used by **execution profilers**
Summary (cont.)

For more information:

Bryant & O’Hallaron, *Computer Systems: A Programmer’s Perspective*, Chapter 8