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
Agenda

Unix Process Control

Signals
Sending Signals
Handling Signals
Alarms

Appendix

Race Conditions and Critical Sections
Blocking Signals
Interval Timers
**What is a 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

```c
void handler(int iSig) {
...
}
```
Unix Process Control

- **Running Foreground Process**
  - ↓ `command`
  - ↑ `Ctrl-c`
  - ↓ `Ctrl-z`
  - ↑ `fg`

- **Stopped Background Process**
  - ↓ `Ctrl-z`
  - ↑ `fg`

- **Non-Existing Process**
  - ↓ `command`
  - ↑ `Ctrl-c`
  - ↓ `Ctrl-z`
  - ↑ `fg`

- **Running Background Process**
  - ↓ `command &`
  - ↑ `kill -2 pid`
  - ↓ `kill -2 pid`
  - ↑ `fg`

- **Kill Process**
  - ↓ `kill -20 pid`
  - ↑ `bg`

[Demo 1]
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
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
Agenda

Unix Process Control

Signals

Sending Signals
Handling Signals
Alarms

Appendix

Race Conditions and Critical Sections
Blocking Signals
Interval Timers
**Signals**

**Signal:** A notification of an exception

**Typical signal sequence:**
- Process P is executing
- Exception occurs (interrupt, trap, fault, or abort)
- OS gains control of CPU
- OS wishes to inform process P that something significant happened
- OS **sends** a signal to process P
  - OS sets a bit in **pending bit vector** of process P
  - Indicates that OS is sending a signal of type X to process P
  - A signal of type X is **pending** for process P
Signals

Typical signal sequence (cont.):

- Sometime later…
- OS is ready to give CPU back to process P
- OS checks *pending* for process P, sees that signal of type X is pending
- OS forces process P to *receive* signal of type X
  - OS clears bit in process P’s *pending*

- Process P executes action for signal of type X
  - Normally process P executes *default action* for that signal
  - If *signal handler* was installed for signal of type X, then process P executes signal handler
  - Action might terminate process P; otherwise…
- Process P resumes where it left off
Examples of Signals

User types Ctrl-c
- Interrupt occurs
- OS gains control of CPU
- OS sends 2/SIGINT signal to process
- Process receives 2/SIGINT signal
- Default action for 2/SIGINT signal is “terminate”

Process makes illegal memory reference
- Segmentation fault occurs
- OS gains control of CPU
- OS sends 11/SIGSEGV signal to process
- Process receives 11/SIGSEGV signal
- Default action for 11/SIGSEGV signal is “terminate”
Signals as Callbacks

Weak analogy:
- **Trap** (and fault and abort) is similar to **function call**
  - App process requests service of OS
- **Signal** is similar to **function callback**
  - OS informs app process that something happened
Agenda

Unix Process Control

Signals

**Sending Signals**

Handling Signals

Alarms

Appendix

Race Conditions and Critical Sections

Blocking Signals

Interval Timers
Sending Signals via Keystrokes

User can send three signals 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”
User can send any signal by executing command:

```
kill command
  • kill -sig pid
  • Send a signal of type sig to process pid
  • No -sig option specified => sends 15/SIGTERM signal
    • Default action for 15/SIGTERM is “terminate”
  • You must own process pid (or have admin privileges)
  • Commentary: Better command name would be sendsig

Examples
  • kill -2 1234
  • kill -SIGINT 1234
  • Same as pressing Ctrl-c if process 1234 is running in foreground

[Demo 2]"
**Sending Signals via Function Calls**

Program can send any signal by calling function:

**raise() function**
- `int raise(int iSig);`
- Commands OS to send a signal of type `iSig` to calling process
- Returns 0 to indicate success, non-0 to indicate failure

**Example**
- `iRet = raise(SIGINT);`
  - Send a 2/SIGINT signal to calling process
Sending Signals via Function Calls

**kill() function**

- `int kill(pid_t iPid, int iSig);`
- Sends a `iSig` signal to the process `iPid`
- Equivalent to `raise(iSig)` when `iPid` is the id of current process
- You must own process `pid` (or have admin privileges)
- Commentary: Better function name would be `sendsig()`

**Example**

- `iRet = kill(1234, SIGINT);`
  - Send a 2/SIGINT signal to process 1234
Agenda

Unix Process Control

Signals

Sending Signals

Handling Signals

Alarms

Appendix

Race Conditions and Critical Sections

Blocking Signals

Interval Timers
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()` function

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

- Install function `pfHandler` as the handler for signals of type `iSig`
- `pfHandler` is a function pointer:
  ```
  typedef void (*sighandler_t)(int);
  ```
- Return the old handler on success, `SIG_ERR` on error
- After call, `(*pfHandler)` is invoked whenever process receives a signal of type `iSig`
Signal Handling Example 1

Program testsignal.c:

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

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

int main(void)
{  signal(SIGINT, myHandler);
   printf("Entering an infinite loop\n");
   for (;;)
   {
      ;
   return 0; /* Never get here. */
}
```

[Demo 3] Error handling code omitted in this and all subsequent programs in this lecture
# Signal Handling Example 2

**Program testsignalall.c:**

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

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

int main(void)
{  int i;
    /* Install myHandler as the handler for all kinds of signals. */
    for (i = 1; i < 65; i++)
        signal(i, myHandler);
    printf("Entering an infinite loop\n");
    for (;;)
        ;
    return 0; /* Never get here. */
}
```

[Demo 4]
Signal Handling 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 for 2/SIGINT 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)
{
  …
  psFile = fopen("temp.txt", "w");
  signal(SIGINT, cleanup);

  …
  cleanup(0); /* or raise(SIGINT); */
  return 0; /* Never get here. */
}
```
Predefined value: **SIG_DFL**

Use as argument to `signal()` to **restore default action**

```c
int main(void)
{
    ...
    signal(SIGINT, somehandler);
    ...
    signal(SIGINT, SIG_DFL);
    ...
}
```

Subsequently, process will handle 2/SIGINT signals using default action for 2/SIGINT signals ("terminate")
**SIG_IGN**

Predefined value: **SIG_IGN**

Use as argument to `signal()` to **ignore signals**

```c
int main(void)
{
    signal(SIGINT, SIG_IGN);
    ...
}
```

Subsequently, process will ignore 2/SIGINT signals
SIG_IGN Example

Program testsignalignore.c:

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

int main(void)
{
    signal(SIGINT, SIG_IGN);
    printf("Entering an infinite loop\n");
    for (;;)
        ;
    return 0; /* Never get here. */
}
```

[Demo 5]
Agenda

Unix Process Control
Signals
Sending Signals
Handling Signals

Alarms

Appendix

Race Conditions and Critical Sections
Blocking Signals
Interval Timers
Alarms

**alarm() function**

- `unsigned int alarm(unsigned int uiSec);`
- Sends `14/SIGALRM` signal after `uiSec` seconds
- Cancels pending alarm if `uiSec` is 0
- Uses *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**
Program testalarm.c:

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

static void myHandler(int iSig)
{  printf("In myHandler with argument %d\n", iSig);
    alarm(2); /* Set another alarm */
}

int main(void)
{  signal(SIGALRM, myHandler);
    alarm(2); /* Set an alarm. */
    printf("Entering an infinite loop\n");
    for (;;)
    
        ;
    return 0; /* Never get here. */
}
```

[Demo 6]
Program testalarmtimeout.c:

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

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

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

[Demo 7]
Summary

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) SIGUSR2
13) SIGPIPE 14) SIGALRM 15) SIGTERM 17) SIGCHLD
18) SIGCONT 19) SIGSTOP 20) SIGTSTP 21) SIGTTIN
22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO
30) SIGPWR 31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1
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
Signals

- A **signal** is an asynchronous event
- Sending signals
  - From the keyboard
  - By calling function: `raise()` or `kill()`
  - By executing command: `kill`
- Catching signals
  - `signal()` installs a signal handler
  - Most signals are catchable

Alarms

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

Unix Process Control
Signals
Sending Signals
Handling Signals
Alarms

Appendix

Race Conditions and Critical Sections
Blocking Signals
Interval Timers
Race Conditions and Critical Sections

Race condition
• A flaw in a program whereby the correctness of the program is critically dependent on the sequence or timing of events beyond the program’s control

Critical section
• A part of a program that must execute atomically (i.e. entirely without interruption, or not at all)
Race condition example:

```c
int iBalance = 2000;
...
static void addBonus(int iSig)
{
  iBalance += 50;
}
int main(void)
{
  signal(SIGINT, addBonus);
  ...
  iBalance += 100;
  ...
```
Race condition example in assembly language

```c
int iBalance = 2000;
...
void addBonus(int iSig)
{
    iBalance += 50;
}
int main(void)
{
    signal(SIGINT, addBonus);
    ...
    iBalance += 100;
    ...
}
```

Let's say the compiler generates that assembly language code:

```assembly
movl iBalance, %ecx
addl $50, %ecx
movl %ecx, iBalance
movl iBalance, %eax
addl $100, %eax
movl %eax, iBalance
```
(1) main() begins to execute

```c
int iBalance = 2000;
...
void addBonus(int iSig)
{
    iBalance += 50;
}
int main(void)
{
    signal(SIGINT, addBonus);
    ...
    iBalance += 100;
    ...
}
```

- Before main() runs, the balance is 2000.
- When `addBonus` runs, it increments the balance by 50.
- When main() runs, it increments the balance by 100.

The assembly code shows the effects of these operations:

- `movl iBalance, %ecx`
- `addl $50, %ecx`
- `movl %ecx, iBalance` (Setting the new balance as 2050)
- `movl iBalance, %eax`
- `addl $100, %eax`
- `movl %eax, iBalance` (Setting the new balance as 2150)
Race Condition Example (cont.)

(2) SIGINT signal arrives; control transfers to addBonus()

```c
int iBalance = 2000;
...

void addBonus(int iSig)
{
    iBalance += 50;
}

int main(void)
{
    signal(SIGINT, addBonus);
    ...
    iBalance += 100;
    ...
}
```

```
movl iBalance, %ecx
addl $50, %ecx
movl %ecx, iBalance

movl iBalance, %eax
addl $100, %eax
movl %eax, iBalance
```
Race Condition Example (cont.)

(3) addBonus() terminates; control returns to main()

```c
int iBalance = 2000;
...

void addBonus(int iSig)
{
    iBalance += 50;
}

int main(void)
{
    signal(SIGINT, addBonus);
    ...
    iBalance += 100;
    ...

    movl iBalance, %ecx
    addl $50, %ecx
    movl %ecx, iBalance

    movl iBalance, %eax
    addl $100, %eax
    movl %eax, iBalance

    Lost $50 !!!
```
Critical Sections

Solution: Must make sure that **critical sections** of code are not interrupted

```c
int iBalance = 2000;
...
void addBonus(int iSig)
{
    iBalance += 50;
}
int main(void)
{
    signal(SIGINT, addBonus);
    ...
    iBalance += 100;
    ...
}
```
Agenda

Unix Process Control

Signals

Sending Signals

Handling Signals

Alarms

Appendix

Race Conditions and Critical Sections

Blocking Signals

Interval Timers
Blocking Signals

Blocking signals

• A process can **block** a signal type to prohibit signals of that type from being received (until unblocked at a later time)
• Differs from **ignoring** a signal

Each process has a **blocked** bit vector in the kernel

• OS uses **blocked** to decide which signals to force the process to receive
• User program can modify **blocked** with **sigprocmask()**
Function for Blocking Signals

**sigprocmask() function**

- `int sigprocmask(int iHow,
  const sigset_t *psSet,
  sigset_t *psOldSet);`

  - `psSet`: Pointer to a signal set
  - `psOldSet`: (Irrelevant for our purposes)
  - `iHow`: How to modify the **blocked** bit vector
    - `SIG_BLOCK`: Add signals in `psSet` to **blocked**
    - `SIG_UNBLOCK`: Remove signals in `psSet` from **blocked**
    - `SIG_SETMASK`: Install `psSet` as **blocked**

- Returns 0 iff successful

**Functions for constructing signal sets**

- `sigemptyset()`, `sigaddset()`, ...
int main(void)
{
    sigset_t sSet;
    signal(SIGINT, addBonus);
    ...
    sigemptyset(&sSet);
    sigaddset(&sSet, SIGINT);
    sigprocmask(SIG_BLOCK, &sSet, NULL);
    iBalance += 100;
    sigprocmask(SIG_UNBLOCK, &sSet, NULL);
    ...
}

Block SIGINT signals
Critical section
 Unblock SIGINT signals
Blocking Signals in Handlers

How to block signals when handler is executing?

- While executing a handler for a signal of type X, all signals of type X are blocked automatically
- When/if signal handler returns, block is removed

```c
void addBonus(int iSig)
{
    iBalance += 50;
}
```

SIGINT signals automatically blocked in SIGINT handler
Agenda

Unix Process Control
Signals
Sending Signals
Handling Signals
Alarms

Appendix
Race Conditions and Critical Sections
Blocking Signals
Interval Timers
Setitimer() function

```c
int setitimer(int iWhich,
              const struct itimerval *psValue,
              struct itimerval *psOldValue);
```

- Send 27/SIGPROF signal continually
- `psValue` specifies timing
- `psOldValue` is irrelevant for our purposes
- Use CPU time
  - Time spent executing other processes does not count
  - Time spent waiting for user input does not count
- Return 0 if successful, -1 otherwise

Used by execution profilers
Program testitimer.c:

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/time.h>

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

int main(void)
{  struct itimerval sTimer;
    signal(SIGPROF, myHandler);
    sTimer.it_value.tv_sec = 1;     /* Send first signal in 1 second */
    sTimer.it_value.tv_usec = 0;    /* and 0 microseconds. */
    sTimer.it_interval.tv_sec = 1;  /* Send subsequent signals in 1 sec */
    sTimer.it_interval.tv_usec = 0; /* and 0 microsecond intervals. */
    setitimer(ITIMER_PROF, &sTimer, NULL);
    printf("Entering an infinite loop\n");
    for (;;)
      ;
    return 0; /* Never get here. */
}
```
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

Interval Timers
- Call `setitimer()` to deliver 27/SIGPROF signals in CPU time
- Interval timers are used by execution profilers
For more information:

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

We have covered:

Programming in the large
• The C programming language
• Testing
• Building
• Debugging
• Program & programming style
• Data structures
• Modularity
• Performance
Course Summary

We have covered (cont.):

Under the hood
- Number systems
- Language levels tour
  - Assembly language
  - Machine language
  - Assemblers and linkers
- Service levels tour
  - Exceptions and processes
  - Storage management
  - Dynamic memory management
  - Process management
  - I/O management
  - Signals
The Rest of the Course

Lecture on Wednesday
  • Program Verification

Assignment 7
  • Due on Dean’s Date (5/12) at 5PM
  • Cannot submit late (University regulations)
  • Cannot use late pass

Office hours and exam prep sessions
  • Will be announced on Piazza

Final exam
  • When: Tuesday 5/19, 1:30 PM
  • Where: Friend Center 101
  • Closed book, closed notes, no electronic devices