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. Race Conditions and Critical Sections
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. Conclusion

Unix Process Control

- Non-Existing Process
  - command and Ctrl-c
  - command & kill -2 pid
  - fg

- Running Foreground Process
  - command
  - Ctrl-z
  - fg

- Running Background Process
  - kill -2 pid

- Stopped Background Process
  - kill -20 pid
  - bg
Unix Process Control

[Demo of Unix process control using infloop.c]

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

Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. Handling Signals
5. Race Conditions and Critical Sections
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. 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
  - Sets a bit in a vector indicating that a signal of type X occurred
- 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"
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”
- Editorial: 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

Sending Signals via Function Calls

**raise()**

```
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**

```
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) */
```

Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. **Handling Signals**
5. Race Conditions and Critical Sections
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. Conclusion
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()
sighandler_t signal(int iSig,
  sighandler_t pfHandler);

• Installs function pfHandler as the handler for signals of type iSig
• pfHandler is a function pointer:
  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;
}
```

[Demo of testsignal.c]
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);
}
```

```
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 */

    ...  

    This call fails
}
```
Program testsignalall.c (cont.):

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

[Demo of testsignalall.c]
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 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) {
    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")

---

**Outline**

1. Unix Process Control
2. Signals
3. Sending Signals
4. Handling Signals
5. **Race Conditions and Critical Sections**
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. Conclusion
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

Race condition example:

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

To save slide space, we ignore error handling here and subsequently
Race Condition Example (cont.)

Race condition example in assembly language

```c
int iBalance = 2000;
...
void addBonus(int iSig) {
    iBalance += 50;
}
int main(void) {
    signal(SIGINT, addBonus);
    ...
    iBalance += 100;
    ...
    movl iBalance, %eax
    addl $100, %eax
    movl %eax, iBalance
    movl iBalance, %ecx
    addl $50, %ecx
    movl %ecx, iBalance
    movl iBalance, %eax
    addl $100, %eax
    movl %eax, iBalance
}
```

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

Race Condition Example (cont.)

(1) main() begins to execute

```c
int iBalance = 2000;
...
void addBonus(int iSig) {
    iBalance += 50;
}
int main(void) {
    signal(SIGINT, addBonus);
    ...
    iBalance += 100;
    ...
    movl iBalance, %eax
    addl $100, %eax
    movl %eax, iBalance
    movl iBalance, %ecx
    addl $50, %ecx
    movl %ecx, iBalance
    movl iBalance, %eax
    addl $100, %eax
    movl %eax, iBalance
```

2000
2100
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, %eax
    addl $100, %eax
    movl %eax, iBalance
}
```

Lost $50 !!!

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, %eax
    addl $100, %eax
    movl %eax, iBalance
    movl iBalance, %ecx
    addl $50, %ecx
    movl %ecx, 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;
    ...
```

Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. Handling Signals
5. Race Conditions and Critical Sections
6. **Blocking Signals**
7. Alarms
8. (If time) Interval Timers
9. Conclusion
Blocking Signals

Blocking signals
• To block a signal is to queue it for delivery at a later time
• Differs from ignoring a signal

Each process has a signal mask in the kernel
• OS uses the mask to decide which signals to deliver
• User program can modify mask with `sigprocmask()`

Function for Blocking Signals

`sigprocmask()`
```c
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 signal mask
  • `SIG_BLOCK`: Add `psSet` to the current mask
  • `SIG_UNBLOCK`: Remove `psSet` from the current mask
  • `SIG_SETMASK`: Install `psSet` as the signal mask
• Returns 0 if successful

Functions for constructing signal sets
• `sigemptyset()`, `sigaddset()`, ...
Blocking Signals Example

```c
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);
    ...
}
```

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;
}
```
Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. Handling Signals
5. Race Conditions and Critical Sections
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. Conclusion

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 1

Program testalarm.c:

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

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

Alarm Example 1 (cont.)

Program testalarm.c (cont.):

```c
... int main(void) {
    sigset_t sSet;
    /* Make sure SIGALRM signals are not blocked. */
    sigemptyset(&sSet);
    sigaddset(&sSet, SIGALRM);
    sigprocmask(SIG_UNBLOCK, &sSet, NULL);
    signal(SIGALRM, myHandler);
    ...

    Safe, but shouldn't be necessary; compensates for a Linux bug
```
Alarm Example 1 (cont.)

Program testalarm.c (cont.):

```c
...
/* Set an alarm. */
alarm(2);

printf("Entering an infinite loop\n");
for (;;) ;

return 0;
```

[Demo of testalarm.c]
Alarm Example 2

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 2 (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);

    ...  
```
Alarm Example 2 (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;
}
```

Alarm Example 2 (cont.)

[Demo of testalarmtimeout.c]
Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. Handling Signals
5. Race Conditions and Critical Sections
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. Conclusion

Interval Timers

\texttt{setitimer()}

\begin{verbatim}
int setitimer(int iWhich,
    const struct itimerval *psValue,
    struct itimerval *psOldValue);
\end{verbatim}

- Sends \texttt{27/SIGPROF} signal continually
- \texttt{psValue} specifies timing
- \texttt{psOldValue} is irrelevant for our purposes
- Uses \texttt{virtual time}, alias \texttt{CPU time}
  - Time spent executing other processes does not count
  - Time spent waiting for user input does not count
- Returns 0 iff successful

Used by execution profilers
Interval Timer Example

Program testitimer.c:

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

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

...
Program testitimer.c (cont.):

    /* Send first signal in 1 second, 0 microseconds. */
    sTimer.it_value.tv_sec = 1;
    sTimer.it_value.tv_usec = 0;

    /* Send subsequent signals in 1 second,
       0 microseconds intervals. */
    sTimer.it_interval.tv_sec = 1;
    sTimer.it_interval.tv_usec = 0;

    setitimer(ITIMER_PROF, &sTimer, NULL);

    printf("Entering an infinite loop\n");
    for (;;)
        ;
    return 0;

[Demo of testitimer.c]
Outline

1. Unix Process Control
2. Signals
3. Sending Signals
4. Handling Signals
5. Race Conditions and Critical Sections
6. Blocking Signals
7. Alarms
8. (If time) Interval Timers
9. Conclusion

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) SIGUSR2
13) SIGPIPE  14) SIGALRM  15) SIGTERM  17) SIGCHLD
18) SIGCONT  19) SIGSTOP  20) SIGTSTP  21) SIGTTIN
22) SIGTTOU  23) SIGURG  24) SIGKILL  25) SIGXFSZ
26) SIGVTALRM  27) SIGPROF  28) SIGWINCH  29) SIGIO
30) SIGFPE  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
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

On a Personal Note…

(Premature) congratulations!
The course was good for me
  Thank you!
I hope the course was good for you
  I hope you learned a lot
  I hope you had fun
I hope you do well on the final assignment/exam
I wish you all the best