Signals

Context of Remaining Lectures

Second half of COS 217 takes 2 tours:

1. "Language levels" tour
   C → assembly language → machine language
   Illustrated by assembly language asgt, buffer overrun asgt
2. "Service levels" tour
   C → C standard library → operating system (OS)
   Illustrated by heap manager asgt, shell asgt

The 2 remaining lectures flesh out the "service levels" tour

Goals of Remaining Lectures

Two fundamental questions:

Q1: How does the **OS** communicate to an **application process**?
Q2: How does an **application process** communicate to the **OS**?

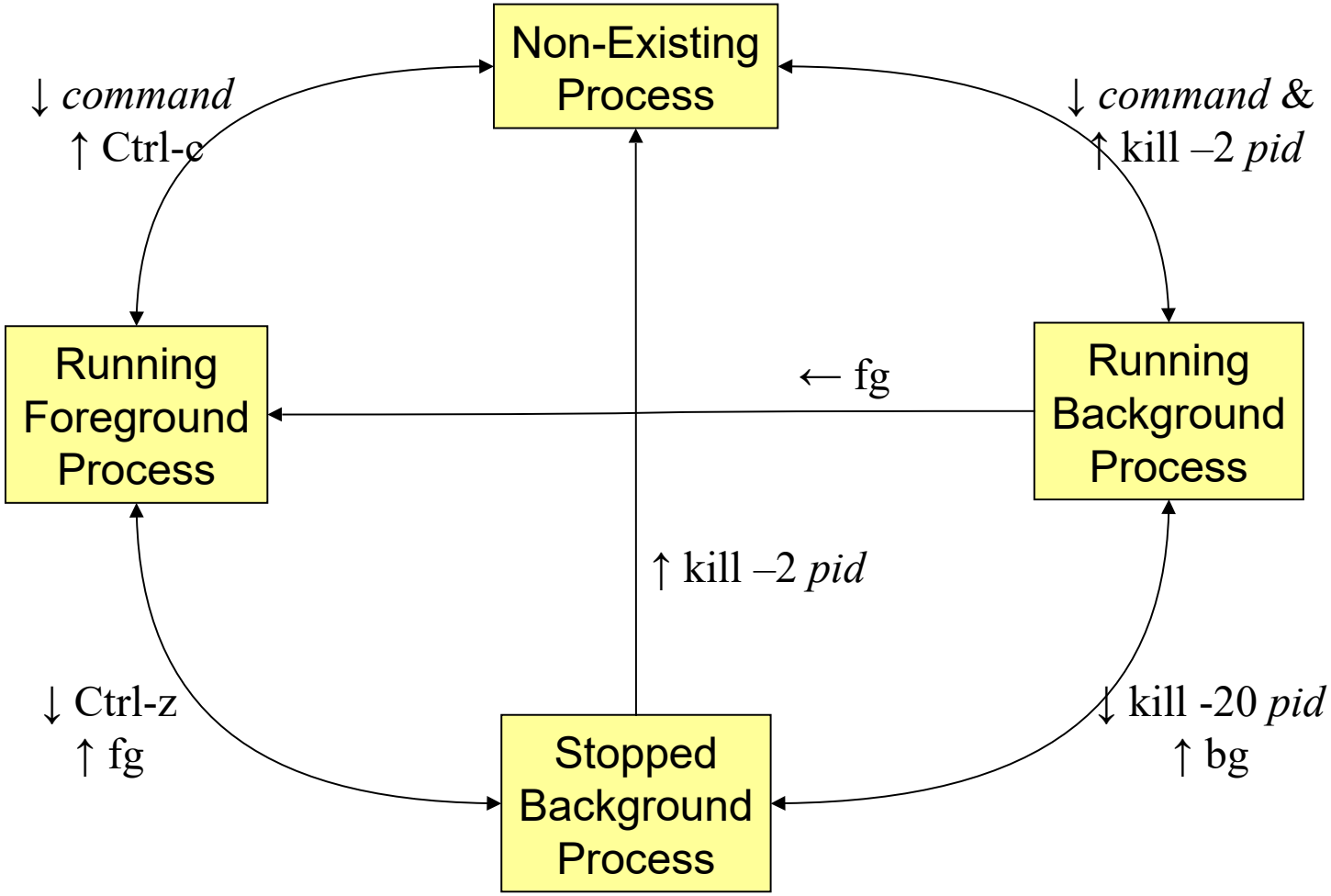
This lecture: Q1

Next lecture: Q2

Outline

1. **UNIX Process Control**
2. Signals
3. C90 Signal Handling
4. C90 Signal Blocking
5. POSIX Signal Handling/Blocking
6. Conclusion
7. (optional) Alarms and Interval Timers

UNIX Process Control



UNIX Process Control

[Demo of UNIX process control using infloop.c]

Process Control Implementation

Exactly what happens when you:

- Type Ctrl-c?
  - Keyboard sends hardware interrupt
  - Hardware interrupt is handled by OS
  - OS sends a 2/SIGINT **signal**
- Type Ctrl-z?
  - Keyboard sends hardware interrupt
  - Hardware interrupt is handled by OS
  - OS sends a 20/SIGTSTP **signal**
- Issue a "kill –*sig pid*" command?
  - OS sends a *sig* **signal** to the process whose id is *pid*
- Issue a "fg" or "bg" command?
  - OS sends a 18/SIGCONT **signal** (and does some other things too!)

Outline

1. UNIX Process Control
2. **Signals**
3. C90 Signal Handling
4. C90 Signal Blocking
5. POSIX Signal Handling/Blocking
6. Conclusion
7. (optional) Alarms and Interval Timers

Signals

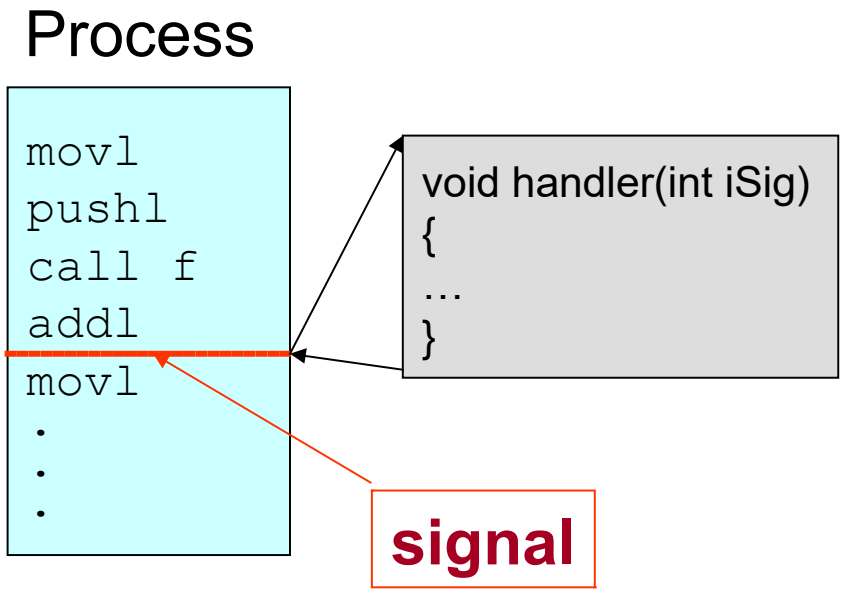
Q1: How does the **OS** communicate to an **application process**?

A1: **Signals**

Definition of Signal

Signal: A notification of an event

- Event gains attention of the OS
- OS stops the application process immediately, sending it a signal
- **Signal handler** executes to completion
- Application process resumes where it left off



Examples of Signals

User types Ctrl-c

- Event gains attention of OS
- OS stops the application process immediately, sending it a 2/SIGINT signal
- Signal handler for 2/SIGINT signal executes to completion
  - Default signal handler for 2/SIGINT signal exits process

Process makes illegal memory reference

- Event gains attention of OS
- OS stops application process immediately, sending it a 11/SIGSEGV signal
- Signal handler for 11/SIGSEGV signal executes to completion
  - Default signal handler for 11/SIGSEGV signal prints "segmentation fault" and exits process

Sending Signals via Keystrokes

Three signals can be sent from keyboard:

- Ctrl-c → 2/SIGINT signal
  - Default handler exits process
- Ctrl-z → 20/SIGTSTP signal
  - Default handler suspends process
- Ctrl-\ → 3/SIGQUIT signal
  - Default handler exits process

Sending Signals via Commands

kill Command

`kill -signal pid`

- Send a signal of type **`signal`** to the process with id **`pid`**
- Can specify either signal type name (-SIGINT) or number (-2)

◦ No signal type name or number specified => sends 15/SIGTERM signal

- Default 15/SIGTERM handler exits process

◦ Editorial comment: 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 Call

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

Note: C90 function

Sending Signals via Function Call

`kill()`

```
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 comment: Better function name would be `sendsig()`

Example

```
pid_t iPid = getpid(); /* Process gets its id.*/
kill(iPid, SIGINT);  /* Process sends itself a
                        SIGINT signal (commits
                        suicide?) */
```

Note: POSIX (not C90) function

Signal Handling

Each signal type has a default handler

- Most default handlers exit the process

A program can install its own handler for signals of any type

Exceptions: A program *cannot* install its own handler for signals of type:

- 9/SIGKILL
  - Default handler exits the process
  - Catchable termination signal is 15/SIGTERM
- 19/SIGSTOP
  - Default handler suspends the process
  - Can resume the process with signal 18/SIGCONT
  - Catchable suspension signal is 20/SIGTSTP

Outline

1. UNIX Process Control
2. Signals
3. **C90 Signal Handling**
4. C90 Signal Blocking
5. POSIX Signal Handling/Blocking
6. Conclusion
7. (optional) Alarms and Interval Timers

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:

```
#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.):

```
…
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 1 (cont.)

[Demo of testsignal.c]

Installing a Handler Example 2

Program testsignalall.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 */
   …
```

Installing a Handler Example 2 (cont.)

Program testsignalall.c (cont.):

```
    …
    /* Etc., for every signal. */

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

Installing a Handler Example 2 (cont.)

[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

```
…
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 handler of 2/SIGINT exits the process

Problem: The temporary file is not deleted

- Process dies before `remove("tmp.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


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

Predefined Signal Handler: SIG_IGN

Pre-defined signal handler: SIG_IGN

Can install to ignore signals

```
int main(void) {
    void (*pfRet)(int);
    pfRet = signal(SIGINT, SIG_IGN);
    …
}
```

Subsequently, process will ignore 2/SIGINT signals

Predefined Signal Handler: SIG_DFL

Pre-defined signal handler: SIG_DFL

Can install to restore default signal handler

```
int main(void) {
    void (*pfRet)(int);
    pfRet = signal(SIGINT, somehandler);
    …
    pfRet = signal(SIGINT, SIG_DFL);
    …
}
```

Subsequently, process will handle 2/SIGINT signals using the default handler for 2/SIGINT signals

Outline

1. UNIX Process Control
2. Signals
3. C90 Signal Handling
4. **C90 Signal Blocking**
5. POSIX Signal Handling/Blocking
6. Conclusion
7. (optional) Alarms and Interval Timers

Race Conditions in Signal Handlers

A **race condition** is a flaw in a program whereby the correctness of the program is critically dependent on the sequence or timing of other events.

Race conditions can occur in signal handlers…

Race Condition Example


```
void addSalaryToSavings(int iSig) {
     int iTemp;
     iTemp = iSavingsBalance;
     iTemp += iMonthlySalary;
     iSavingsBalance = iTemp;
}
```

Handler for hypothetical
"update monthly salary" signal

Race Condition Example (cont.)

(1) Signal arrives; handler begins executing

```
void addSalaryToSavings(int iSig) {
    int iTemp;
    iTemp = iSavingsBalance;
    iTemp += iMonthlySalary;
    iSavingsBalance = iTemp;
}
```

2000

Race Condition Example (cont.)

(2) Another signal arrives; first instance of handler is interrupted; second instance of handler begins executing

```
void addSaliryToSavings(int iSig) {
    int iTemp;
    iTemp = iSavingsBalance;
    iTemp += iMonthlySalary;
    iSavingsBalance = iTemp;
}
```

2000

```
void addSaliryToSavings(int iSig) {
    int iTemp;
    iTemp = iSavingsBalance;
    iTemp += iMonthlySalary;
    iSavingsBalance = iTemp;
}
```

2000

Race Condition Example (cont.)

(3) Second instance executes to completion

```
void addSalaryToSavings(int iSig) {
    int iTemp;
    iTemp = iSavingsBalance;
    iTemp += iMonthlySalary;
    iSavingsBalance = iTemp;
}
```

2000

```
void addSalaryToSavings(int iSig) {
    int iTemp;
    iTemp = iSavingsBalance;
    iTemp += iMonthlySalary;
    iSavingsBalance = iTemp;
}
```

2000

2050

2050

Race Condition Example (cont.)

(4) Control returns to first instance, which executes to completion

```
void addSalaryToSavings(int iSig) {
    int iTemp;
    iTemp = iSavingsBalance;        2000
    iTemp += iMonthlySalary;        2050
    iSavingsBalance = iTemp;        2050
}
```

Lost 50 !!!

Blocking Signals in Handlers

Blocking signals

- To **block** a signal is to **queue** it for delivery at a later time

Why block signals when handler is executing?

- Avoid race conditions when another signal of type x occurs while the handler for type x is executing

How to block signals when handler is executing?

- **Automatic** during execution of signal handler!!!
- Previous sequence **cannot happen!!!**
- While executing a handler for a signal of type x, all signals of type x are blocked
- When/if signal handler returns, block is removed

Race Conditions in General

Race conditions can occur elsewhere too

```
int iFlag = 0;

void myHandler(int iSig) {
    iFlag = 1;
}

int main(void) {
    if (iFlag == 0) {
        /* Do something */
    }
}
```

Problem: **`myflag`** might become **`1`** just after the comparison!

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

Blocking Signals in General

How to block signals in general?

- Not possible in C90
- Possible using POSIX functions…

Outline

1. UNIX Process Control
2. Signals
3. C90 Signal Handling
4. C90 Signal Blocking
5. **POSIX Signal Handling/Blocking**
6. Conclusion
7. (optional) Alarms and Interval Timers

POSIX Signal Handling

C90 standard

- Defines **`signal()`** and **`raise()`** functions
  - Work across all systems (UNIX, LINUX, Windows), but…
  - Work **differently** across some systems!!!
    - On some systems, signals are blocked during execution of handler for that type of signal -- but not so on other (older) systems
    - On some (older) systems, handler installation for signals of type x is cancelled after first signal of type x is received; must reinstall the handler -- but not so on other systems
- Does not provide mechanism to block signals in general

POSIX Signal Handling

POSIX standard

- Defines **`kill()`**, **`sigprocmask()`**, and **`sigaction()`** functions
  - Work the same across all POSIX-compliant UNIX systems (Linux, Solaris, etc.), but…
  - Do not work on non-UNIX systems (e.g. Windows)
- Provides mechanism to block signals in general

Blocking Signals in General

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()`**

`sigprocmask()`

```
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 iff successful

Functions for constructing signal sets

- **`sigemptyset()`**, **`sigaddset()`**, …

Note: No parallel function in C90

Blocking Signals Example


```
sigset_t sSet;
int main(void) {
    int iRet;
    sigemptyset(&sSet);
    sigaddset(&sSet, SIGINT);
    iRet = sigprocmask(SIG_BLOCK, &sSet, NULL);
    assert(iRet == 0);
    if (iFlag == 0) {
        /* Do something */
    }
    iRet = sigprocmask(SIG_UNBLOCK, &sSet, NULL);
    assert(iRet == 0);
    …
}
```

Blocking Signals in Handlers

Signals of type x automatically are blocked when executing handler for signals of type x

Additional signal types to be blocked can be defined at time of handler installation…

Installing a Signal Handler

`sigaction()`

```
int sigaction(int iSig,
              const struct sigaction *psAction,
              struct sigaction *psOldAction);
```

- **`iSig`**: The type of signal to be affected
- **`psAction`**: Pointer to a structure containing instructions on how to handle signals of type **`iSig`**, including signal handler name and which signal types should be blocked
- **`psOldAction`**: (Irrelevant for our purposes)
- Installs an appropriate handler
- Automatically blocks signals of type **`iSig`**
- Returns 0 iff successful

Note: More powerful than C90 **`signal()`**

Installing a Handler Example

Program testsigaction.c:

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

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

Installing a Handler Example (cont.)

Program testsigaction.c (cont.):

```
…
int main(void) {
    int iRet;
    struct sigaction sAction;
    sAction.sa_flags = 0;
    sAction.sa_handler = myHandler;
    sigemptyset(&sAction.sa_mask);
    iRet = sigaction(SIGINT, &sAction, NULL);
    assert(iRet == 0);

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

Installing a Handler Example (cont.)

[Demo of testsigaction.c]

Outline

1. UNIX Process Control
2. Signals
3. C90 Signal Handling
4. C90 Signal Blocking
5. POSIX Signal Handling/Blocking
6. **Conclusion**
7. (optional) Alarms and Interval Timers

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) 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
```

Applications can define their own signals

- An application can define signals with unused values

Summary

Signals

- A **signal** is an asynchronous event mechanism
- C90 **`raise()`** or POSIX **`kill()`** sends a signal
- C90 **`signal()`** or POSIX **`sigaction()`** installs a signal handler
  - Most predefined signals are “catchable”
- Beware of race conditions
- Signals of type x automatically are blocked while handler for type x signals is running
- POSIX **`sigprocmask()`** blocks signals in any critical section of code

Summary

Q: How does the OS communicate to application programs?

A: Signals

For more information:

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

Outline

1. UNIX Process Control
2. Signals
3. C90 Signal Handling
4. C90 Signal Blocking
5. POSIX Signal Handling/Blocking
6. Conclusion
7. **(optional) Alarms and Interval Timers**

Alarms

`alarm()`

```
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 meaningless

Used to implement time-outs

Alarm Example 1

Program testalarm.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.):

```
…
int main(void)
{
    void (*pfRet)(int);
    sigset_t sSet;
    int iRet;

    /* Make sure that SIGALRM is not blocked. */
    sigemptyset(&sSet);
    sigaddset(&sSet, SIGALRM);
    iRet = sigprocmask(SIG_UNBLOCK, &sSet, NULL);
    assert(iRet == 0);

    pfRet = signal(SIGALRM, myHandler);
    assert(pfRet != SIG_ERR);
    …
```

Alarm Example 1 (cont.)

Program testalarm.c (cont.):

```
    …

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

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

    return 0;
}
```

Alarm Example 1 (cont.)

[Demo of testalarm.c]

Alarm Example 2

Program testalarmtimeout.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("\nSorry.  You took too long.\n");
   exit(EXIT_FAILURE);
}
```

Alarm Example 2 (cont.)

Program testalarmtimeout.c (cont.):

```
int main(void) {
    int i;
    void (*pfRet)(int);
    sigset_t sSet;
    int iRet;

    /* Make sure that SIGALRM is not blocked. */
    sigemptyset(&sSet);
    sigaddset(&sSet, SIGALRM);
    iRet = sigprocmask(SIG_UNBLOCK, &sSet, NULL);
    assert(iRet == 0);
    …
```

Alarm Example 2 (cont.)

Program testalarmtimeout.c (cont.):

```
    …
    pfRet = signal(SIGALRM, myHandler);
    assert(pfRet != SIG_ERR);

    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]

Interval Timers

`setitimer()`

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

- Sends 27/SIGPROF signal continually
- Timing is specified by **`psValue`**
- **`psOldValue`** is irrelevant for our purposes
- Uses **virtual time**, alias **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:

```
#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);
}
…
```

Interval Timer Example (cont.)

Program testitimer.c (cont.):

```
…
int main(void)
{
    int iRet;
    void (*pfRet)(int);
    struct itimerval sTimer;

    pfRet = signal(SIGPROF, myHandler);
    assert(pfRet != SIG_ERR);
    …
```

Interval Timer Example (cont.)

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;

    iRet = setitimer(ITIMER_PROF, &sTimer, NULL);
    assert(iRet != -1);

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

Interval Timer Example (cont.)

[Demo of testitimer.c]

Summary

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