



# 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

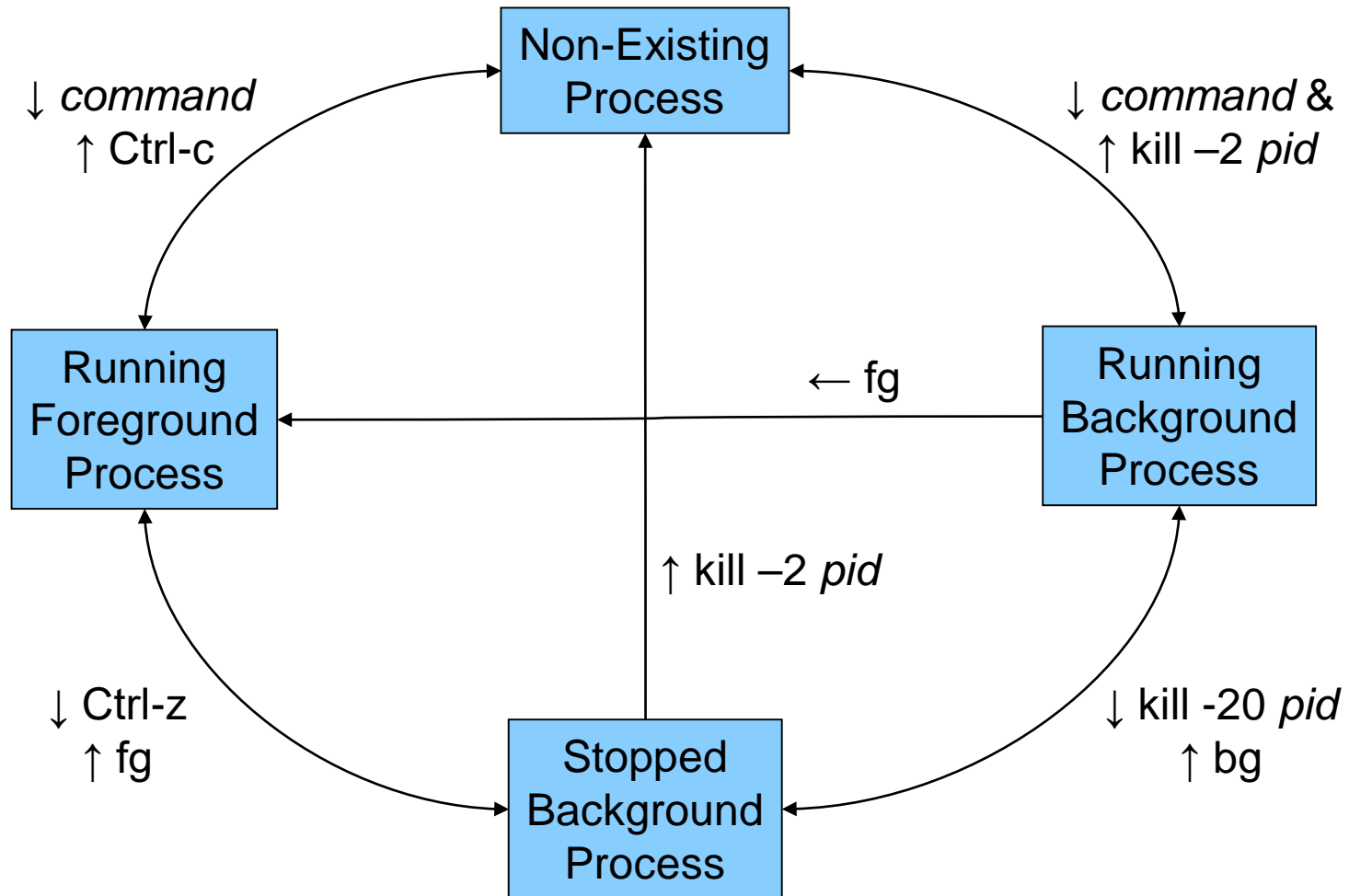
Signals

Sending Signals

Handling Signals

Alarms

# Unix Process Control



# 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

# Agenda



Unix Process Control

**Signals**

Sending Signals

Handling Signals

Alarms

# 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”

# Agenda



Unix Process Control

Signals

**Sending Signals**

Handling Signals

Alarms

# 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”

# Sending Signals via Commands



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  $\Rightarrow$  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

# 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



# 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

# 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`

# SIG\_DFL



Predefined value: **SIG\_DFL**

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

```
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

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

Subsequently, process will ignore 2/SIGINT signals

# 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”

# Signal Handling Example 1



Program testsignal.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. */
}
```

Error handling code omitted  
in this and all subsequent  
programs in this lecture

# Signal Handling Example 2



## Program testsignalall.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. */
}
```

Will fail:  
`signal(9, myHandler)`  
`signal(19, myHandler)`

# Signal Handling 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 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

```
...
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. */
}
```

# Agenda



Unix Process Control

Signals

Sending Signals

Handling Signals

**Alarms**

# Alarms



## `alarm()` function

- `unsigned int alarm(unsigned int uiSec);`
- Send 14/SIGALRM signal after `uiSec` seconds
- Cancel pending alarm if `uiSec` is 0
- Use **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:

```
#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. */
}
```

# Alarm Example 2



## Program testalarmtimeout.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;
}
```

# Summary



List of the predefined signals:

```
$ kill -l
```

1) SIGHUP	2) <b>SIGINT</b>	3) <b>SIGQUIT</b>	4) SIGILL
5) SIGTRAP	6) SIGABRT	7) SIGBUS	8) SIGFPE
9) <b>SIGKILL</b>	10) SIGUSR1	11) <b>SIGSEGV</b>	12) SIGUSR2
13) SIGPIPE	14) <b>SIGALRM</b>	15) <b>SIGTERM</b>	17) SIGCHLD
18) <b>SIGCONT</b>	19) <b>SIGSTOP</b>	20) <b>SIGTSTP</b>	21) SIGTTIN
22) SIGTTOU	23) SIGURG	24) SIGXCPU	25) SIGXFSZ
26) SIGVTALRM	27) <b>SIGPROF</b>	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

# Summary



## Signals

- 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



# Summary (cont.)



For more information:

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

# Wrapping Up the Course



## Assignment 7

- Due on Dean's Date at 5 PM
- No extensions past 11:59 PM without permission of the Dean

## Office hours and exam prep sessions

- Will be announced on Piazza

## Final exam

- Monday, 5/20, 7:30 – 10:30 PM, McCosh 10
- Covers everything, emphasizing material since the midterm
- Mixture of short-answer questions and writing snippets of code
- Closed book and notes, no electronic anything
- Relevant reference material will be provided

Old exams and study guide will be posted on schedule page

# 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 end.



```
return EXIT_SUCCESS;
```