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

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

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

- **Non-Existing Process**
  - ↑ `kill -2 pid`
  - ↓ `command &` (Ctrl-z)

- **Running Background Process**
  - ↑ `kill -2 pid`
  - ↓ `kill -20 pid`
  - ↑ bg

- `fg`
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
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
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
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** $\Rightarrow$ **2/SIGINT** signal
  - Default action is “terminate”
- **Ctrl-z** $\Rightarrow$ **20/SIGTSTP** signal
  - Default action is “stop until next 18/SIGCONT”
- **Ctrl-\** $\Rightarrow$ **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
```
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:
    ```c
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`
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
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:

```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.
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. */
}
```

Will fail:
```c
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

```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

...  
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
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. */
}
```
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("Sorry. 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;
}
```
List of the predefined signals:

<table>
<thead>
<tr>
<th>Number</th>
<th>Signal</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIGHUP</td>
</tr>
<tr>
<td>2</td>
<td>SIGINT</td>
</tr>
<tr>
<td>3</td>
<td>SIGQUIT</td>
</tr>
<tr>
<td>4</td>
<td>SIGILL</td>
</tr>
<tr>
<td>5</td>
<td>SIGTRAP</td>
</tr>
<tr>
<td>6</td>
<td>SIGABRT</td>
</tr>
<tr>
<td>7</td>
<td>SIGBUS</td>
</tr>
<tr>
<td>8</td>
<td>SIGFPE</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
</tr>
<tr>
<td>10</td>
<td>SIGUSR1</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
</tr>
<tr>
<td>12</td>
<td>SIGUSR2</td>
</tr>
<tr>
<td>13</td>
<td>SIGPIPE</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
</tr>
<tr>
<td>15</td>
<td>SIGTERM</td>
</tr>
<tr>
<td>16</td>
<td>SIGCHLD</td>
</tr>
<tr>
<td>17</td>
<td>SIGTSTP</td>
</tr>
<tr>
<td>18</td>
<td>SIGTTIN</td>
</tr>
<tr>
<td>19</td>
<td>SIGURG</td>
</tr>
<tr>
<td>20</td>
<td>SYXCPU</td>
</tr>
<tr>
<td>21</td>
<td>SIGXFSZ</td>
</tr>
<tr>
<td>22</td>
<td>SIGUSTO</td>
</tr>
<tr>
<td>23</td>
<td>SIGURG</td>
</tr>
<tr>
<td>24</td>
<td>SIGXCPU</td>
</tr>
<tr>
<td>25</td>
<td>SIGXFSZ</td>
</tr>
<tr>
<td>26</td>
<td>SIGVTALRM</td>
</tr>
<tr>
<td>27</td>
<td>SIGPROF</td>
</tr>
<tr>
<td>28</td>
<td>SIGWINCH</td>
</tr>
<tr>
<td>29</td>
<td>SIGIO</td>
</tr>
<tr>
<td>30</td>
<td>SIGPWR</td>
</tr>
<tr>
<td>31</td>
<td>SIGSYS</td>
</tr>
<tr>
<td>32</td>
<td>SIGRTMIN</td>
</tr>
<tr>
<td>33</td>
<td>SIGRTMIN+1</td>
</tr>
<tr>
<td>34</td>
<td>SIGRTMIN+2</td>
</tr>
<tr>
<td>35</td>
<td>SIGRTMIN+3</td>
</tr>
<tr>
<td>36</td>
<td>SIGRTMIN+4</td>
</tr>
<tr>
<td>37</td>
<td>SIGRTMIN+5</td>
</tr>
<tr>
<td>38</td>
<td>SIGRTMIN+6</td>
</tr>
<tr>
<td>39</td>
<td>SIGRTMIN+7</td>
</tr>
<tr>
<td>40</td>
<td>SIGRTMIN+8</td>
</tr>
<tr>
<td>41</td>
<td>SIGRTMIN+9</td>
</tr>
<tr>
<td>42</td>
<td>SIGRTMIN+10</td>
</tr>
<tr>
<td>43</td>
<td>SIGRTMIN+11</td>
</tr>
<tr>
<td>44</td>
<td>SIGRTMIN+12</td>
</tr>
<tr>
<td>45</td>
<td>SIGRTMIN+13</td>
</tr>
<tr>
<td>46</td>
<td>SIGRTMIN+14</td>
</tr>
<tr>
<td>47</td>
<td>SIGRTMIN+15</td>
</tr>
<tr>
<td>48</td>
<td>SIGRTMAX-12</td>
</tr>
<tr>
<td>49</td>
<td>SIGRTMAX-13</td>
</tr>
<tr>
<td>50</td>
<td>SIGRTMAX-14</td>
</tr>
<tr>
<td>51</td>
<td>SIGRTMAX-15</td>
</tr>
<tr>
<td>52</td>
<td>SIGRTMAX-16</td>
</tr>
<tr>
<td>53</td>
<td>SIGRTMAX-17</td>
</tr>
<tr>
<td>54</td>
<td>SIGRTMAX-18</td>
</tr>
<tr>
<td>55</td>
<td>SIGRTMAX-19</td>
</tr>
<tr>
<td>56</td>
<td>SIGRTMAX-20</td>
</tr>
<tr>
<td>57</td>
<td>SIGRTMAX-21</td>
</tr>
<tr>
<td>58</td>
<td>SIGRTMAX-22</td>
</tr>
<tr>
<td>59</td>
<td>SIGRTMAX-23</td>
</tr>
<tr>
<td>60</td>
<td>SIGRTMAX-24</td>
</tr>
<tr>
<td>61</td>
<td>SIGRTMAX-25</td>
</tr>
<tr>
<td>62</td>
<td>SIGRTMAX-26</td>
</tr>
<tr>
<td>63</td>
<td>SIGRTMAX-27</td>
</tr>
<tr>
<td>64</td>
<td>SIGRTMAX-28</td>
</tr>
</tbody>
</table>

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: \texttt{raise()} or \texttt{kill()}
  • By executing command: \texttt{kill}
• Catching signals
  • \texttt{signal()} installs a signal handler
  • Most signals are catchable

Alarms
• Call \texttt{alarm()} to send 14/SIGALRM signals in wall-clock time
• Alarms can be used to implement time-outs
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
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;