Languages for Informatics
8 – Signals and Error Handling

Department of Computer Science
University of Pisa
Largo B. Pontecorvo 3
56127 Pisa
Topics

- Linux programming environment (2h)
- Introduction to C programming (12h)
- Basic system programming in Linux (10h)
  1. Signals and Error Handling
  2. Low-Level System Calls in C
  3. Multi-Tasking in C
  4. Multi-Threading in C
  5. Machine-To-Machine Communication in C
Overview

1. Introduction

2. Signals
   - Signal Handling
   - Race Conditions and Critical Sections
   - Blocking Signals
   - Alarms
   - Interval Timers

3. Error Handling
Introduction

Signals
- Signal Handling
- Race Conditions and Critical Sections
- Blocking Signals
- Alarms
- Interval Timers

Error Handling
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
What happens exactly?

- **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?**
  - `kill` command executes **trap**
  - OS handles trap by sending a **signal** to the process whose id is **pid**

- **Issue a `fg` or `bg` command?**
  - `fg` or `bg` command executes **trap**
  - OS handles trap by sending a **18/SIGCONT signal.**
Signals

Q1 How does the OS communicate to an application process?

A1 through Signals.
1 Introduction

2 Signals
   - Signal Handling
   - Race Conditions and Critical Sections
   - Blocking Signals
   - Alarms
   - Interval Timers

3 Error Handling
Definition

- A **signal** is an **asynchronous** notification of an event that is sent to a process.
  - may occur any time by the kernel (e.g. segment violation, floating point error, illegal instruction) or by the terminal driver (due to user interaction)
  - Event gains attention of the OS
  - OS stops the app. process immediately, sending it a signal
  - **Signal handler** executes to completion
  - Application process resumes where it left off

```
mov ldr add str ldr mov
```

```
void handler(intSig) {
    ...
}
```
Examples of Signals

- **User types Ctrl-c**
  - Interrupt occurs
  - OS stops the application process immediately, sending it a \texttt{2/SIGINT} signal
  - Process receives \texttt{2/SIGINT} signal to process
  - Default action for \texttt{2/SIGINT} signal is to terminate the process.

- **Process makes illegal memory reference**
  - Segmentation fault occurs
  - OS stops application process immediately, sending it a \texttt{11/SIGSEGV} signal
  - Process receives \texttt{11/SIGSEGV} signal to process
  - Default action for \texttt{11/SIGSEGV} signal is to terminate the process.
Keystroke based Signals

Three signals can be sent from the keyboard.

- **Ctrl-C** 2/SIGINT signal
  - Default handler **interrupts** process
- **Ctrl-Z** 20/SIGSTP signal
  - Default handler **suspends** process
- **Ctrl-\** 3/SIGQUIT signal
  - Default handler **exits** process
### Sending Signals via Commands

- **kill function**
  - **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)
    - When no signal type name or number specified, default is **15/SIGTERM** (exit)

- **Examples**
  - `$ kill -2 1234`
  - `$ kill -SIGINT 1234`
  - Same as pressing **Ctrl-c** if process 1234 is running in foreground
## Sending Signals via POSIX Function Call

**Syntax**

```c
#include <signal.h>
int kill(pid_t pid, int iSig);
```

- **Process Identification Data Type** `pid_t` is `signed int`
- `getpid()` returns the pid of the calling process
- `iSig` is the signal to be sent.
- **Return**: 0 on SUCCESS; -1 otherwise.

**Example (Demo):**

```c
int main () {
    pid_t process_id;
    process_id = getpid();

    printf("PID: \%d\n", process_id);
    kill(process_id, SIGKILL);   /* suicide */
...
1 Introduction

2 Signals
   - Signal Handling
   - Race Conditions and Critical Sections
   - Blocking Signals
   - Alarms
   - Interval Timers

3 Error Handling
Signal Handling

- Each signal type has a default handler
  - Most default handlers exit the process
- A program can install a customized 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
Installing a Signal Handler `signal()`

- Whenever process receives a signal of type `iSig`, a signal handler function is invoked
- Syntax:

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

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

    ```c
    typedef void (*sighandler_t)(int);
    ```

  - Returns the old handler on success, `SIG_ERR` on error
### Handler Example 1

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>  // sleep
#include <stdlib.h>
#include <string.h>  // strsignal

volatile int flag = 1;  // don’t cache variable

void myHandler(int iSig) {
    printf("In myHandler with argument %d/%s \n", iSig, strsignal(iSig));
    flag = 0;
}
```

### Note

For a real-world program, **printing from a signal handler is not very safe.** A signal handler should do as little as it can, preferably only setting a flag here or there.
int main() {
    /* *************************************************************
     * void (*ret)(int); // fct-pointer int->void
     * int ret = signal(SIGINT, myHandler); // handle Ctrl-C
     *
     * Something went wrong */
     * exit(EXIT_FAILURE);
     */
    int i = 0;
    while (flag) {
        printf("%d \n", i);
        i++;
        sleep(1);
    }
    return 0;
}
### Signals types (Excerpt)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Action</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Term</td>
<td>Interrupt from keyboard <code>Ctrl-C</code></td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>3</td>
<td>Core</td>
<td>Quit from keyboard <code>Ctrl-\</code></td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Core</td>
<td>Illegal Instruction</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Core</td>
<td>Floating point exception</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Term</td>
<td>Kill signal <code>kill -9</code></td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>Core</td>
<td>Invalid memory reference</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>15</td>
<td>Term</td>
<td>Termination signal</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>14</td>
<td>Term</td>
<td>Timer signal from alarm</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>18</td>
<td>Cont</td>
<td>Continue if stopped</td>
</tr>
<tr>
<td>SIGTSTP</td>
<td>20</td>
<td>Stop</td>
<td>Quit from keyboard <code>Ctrl-Z</code></td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>30</td>
<td>Term</td>
<td>User-defined signal</td>
</tr>
</tbody>
</table>

For details, type `kill -l` and check out `man 7 signal`. 
Suppose program generates lots of data
- Data is buffered
- Data written to file when buffer full or `fclose()` occurs

```c
int main(void) {
    FILE *psFile;
    psFile = fopen("temp.txt", "w");
    ...
    fclose(psFile); /* only here all data flushed to file */
    return 0;
}
```
What if user types `Ctrl-c`?
- OS sends a `2/SIGINT0` signal to the process
- Default handler of `2/SIGINT` exits the process

Problem: Data not flushed to file `temp.txt`.
- Process dies before `fclose()` 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 set flag and close file
- Install the function as a signal handler for `2/SIGINT`
FILE *psFile;

static void cleanup(int iSig) {
    fclose(psFile);
    exit(1);
}

int main(void) {
    void (*pfRet)(int);
    pfRet = signal(SIGINT, cleanup);

    char *file = "temp.txt";
    psFile = fopen(file, "w");
    /* do something */
    fclose(psFile);    // dump to file, if possible
    return 0;
}
**Ignoring Signals**

- **Ignoring**: The signal is discarded by the kernel without any action being taken.
- Pre-defined signal handler `SIG_IGN` tells the kernel to ignore signals
- **Note**: `SIGKILL` and `SIGSTOP` cannot be caught or ignored
- Code snippet

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

- Subsequently, process will ignore `2/SIGINT` signals
Pre-defined signal handler **SIG_DFL**

- tells the kernel that there is no user signal handler i.e., to take default action

Code snippet (Demo)

```c
void myHandler(int iSig) {
    printf("!?!X! -- I got signal %d \n", iSig);
    (void) signal(SIGINT, SIG_DFL); // default next time.
}

int main() {
    void (*ret)(int);
    ret = signal(SIGINT, myHandler); // handle ctrl-c
    ...
}
```

- Hence, process will handle **2/SIGINT** signals using the default handler.
1 Introduction

2 Signals
   - Signal Handling
   - Race Conditions and Critical Sections
   - Blocking Signals
   - Alarms
   - Interval Timers

3 Error Handling
Race Conditions and Critical Sections

Race condition

**Def.**: 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

**Def.**: A part of a program that must execute atomically (i.e. entirely without interruption, or not at all)
Race Condition Example (1)

Consider a handler for hypothetical "update monthly salary" signal:

```c
int iBalance = 200;

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

int main() {
    void (*ret)(int); // fct-pointer int->void
    ret = signal(SIGINT, addBonus);
    ...
    iBalance += 100;
    ...
} 
```
Race Condition Example (2)

Race condition example in assembly language

```assembly
int iBalance = 200;

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

int main() {
    void (*ret) (int); // fct-pointer int->void
    ret = signal(SIGINT, addBonus);

    iBalance += 100;
    ...
}
```

Let the compiler generates that assembly language code
**Race Condition Example (3)**

```c
int iBalance = 200;

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

int main() {
    void (*ret) (int); // fct-pointer int->void
    ret = signal(SIGINT, addBonus);
    ...

    iBalance += 100;
    ...
}
```

1. **main()** starts to execute

   ```c
   movl iBalance, %ecx
   addl 100, %ecx
   movl %ecx, iBalance
   ```

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

   ```c
   movl iBalance, %ecx
   addl 100, %ecx
   movl %ecx, iBalance
   ```

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

   ```c
   movl iBalance, %ecx
   addl 100, %ecx
   movl %ecx, iBalance
   ```

   ```c
   movl iBalance, %ecx
   addl 50, %ecx
   movl %ecx, iBalance
   ```
Race Condition Example (4)

2 SIGINT interrupt; addBonus() executed

```c
int iBalance = 200;

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

int main() {
    void (*ret)(int); // fct-pointer int->void
    ret = signal(SIGINT, addBonus);

    iBalance += 100;

    movl iBalance, %ecx
    addl 100, %ecx
    movl %ecx, iBalance
}
```
addBonus() terminates; main() to be continued

```c
int iBalance = 200;

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

int main() {
    void (*ret) (int); // fct-pointer int->void
    ret = signal(SIGINT, addBonus);
    ...
    iBalance += 100;
    ...
}
```

50 € lost!!
Critical Sections

- The **critical sections** are the transactions.
- This piece of the code must not be interrupted.

```c
int iBalance = 200;

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

int main() {
    void (*ret) (int); // fct-pointer int->void
    ret = signal(SIGINT, addBonus);

    iBalance += 100;
    ...
}
```
1 Introduction

2 Signals
   - Signal Handling
   - Race Conditions and Critical Sections
   - Blocking Signals
   - Alarms
   - Interval Timers

3 Error Handling
**Blocked Signal**

- **Blocking** a signal means telling the operating system to **hold it until** it is unblocked at a later time.
- For each process, the kernel also maintains the set of blocked signals in a "blocked bit" vector (in the kernel memory space). The corresponding bit in the pending signal vector is not cleared until the signal is unblocked and received.
- Blocking certain signals **forces an execution order.**
  - Differs from **ignoring** a signal.
  - **briefly** blocking signals prevent them from interrupting sensitive operations by another signal (race condition).
Blocking/unblocking a signal (1)

Procedure:

- System call `sigemptyset(const sigset_t *psSet)` simply initializes the signalmask to empty, i.e. all signals will be received.

- System calls `sigaddset(sigset_t *psSet, int signum)` and `sigdelset` subroutines respectively add and delete the individual signal specified by the Signal Number `signum` parameter.

- Having added here all the signals you want to have blocked, then block them all with `sigprocmask`
System call `sigprocmask()` can examine and modify the set of blocked signals. `SIGKILL` and `SIGSTOP` cannot be blocked by `sigprocmask()`.

Prototype

```c
int sigprocmask(int how, const sigset_t *psSet, sigset_t *oldset);
```

The 1. argument can be:

- `SIG_BLOCK`: adds the signal types specified by the second argument set to the list of already-blocked signal types.
- `SIG_UNBLOCK`: removes the signal types specified by the second argument set to the list of already-blocked signal types.
- `SIG_SETMASK`: set to list of blocked signal types to the ones specified by the second argument.

`psSet` is a pointer of type `sigset_t *`, which points to a “bit vector”

`psOldSet` is irrelevant for our purposes.
Blocking Signals Example

```c
int main () {

    sigset_t sSet;         // Declaration of signal set
...

    void (*ret) (int);     // fct-pointer int->void
    ret = signal(SIGINT, addBonus);
...
    sigemptyset(&sSet);     // init mask
    sigaddset(&sSet, SIGINT);   // add Ctrl-C
    sigprocmask(SIG_BLOCK, &sSet, NULL);  // block SIGINT

    iBalance += 100;        // Critical Section

    sigprocmask(SIG_UNBLOCK, &sSet, NULL); // unblock SIGINT
...
```
The `signal()` function (usually) resets the signal action back to SIG_DFL (default) for almost all signals. This means that the `signal()` handler must reinstall itself as its first action. This opens up a window of vulnerability.

The effects of `signal()` in a multi-threaded process are unspecified.

The exact behaviour of `signal()` varies among systems.

The system call `sigaction()` overcomes above drawbacks. Signal that caused the handler to be triggered will by default already be blocked inside the handler.
**sigaction()**: A more robust approach to signal()

The system call `sigaction()` installs an appropriate handler

- **Prototype**
  ```c
  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)
- **Automatically blocks** signals of type `iSig`
- **Returns 0 iff successful**
int main () {

    int iRet;     // instead of a pointer

    struct sigaction sAction;
    sAction.sa_flags = 0;   // to deploy the handler
    sAction.sa_handler = &addBonus;
    sigemptyset(&sAction.sa_mask);
    iRet = sigaction(SIGINT, &sAction, NULL);
    if (iRet != 0) {
        /* Something went wrong */
        exit(EXIT_FAILURE);
    }

    iBalance += 100;
    ...
}
1 Introduction

2 Signals
   - Signal Handling
   - Race Conditions and Critical Sections
   - Blocking Signals
   - Alarms
   - Interval Timers

3 Error Handling
The `alarm()` function sets a real-time timer.

Prototype:

```c
unsigned int alarm(unsigned int uiSec);
```

Sends `14/SIGALRM` signal after `uiSec` seconds

If there is a previous `alarm()` request with time remaining, the return value is this value.

Cancels pending alarm with `alarm(0)`.

**Note**

If you simply want your process to wait for a given number of seconds, you should use the `sleep()` function.
# Alarms

**Example – Timeout**

```c
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h> // alarm

static void myHandler(int iSig)
{
    printf("\nSorry. You took too long.\n") ;
    exit(EXIT_FAILURE) ;
}
```
```c
int main(void)
{
    int timeout = 5;
    signal(SIGALRM, myHandler);
    printf("Press a key within %d s: ", timeout);
    alarm(sec);  // set alarm

    getc(stdin);

    printf("Good job! \n");
    return 0;
}
```
1 Introduction

2 Signals
   • Signal Handling
   • Race Conditions and Critical Sections
   • Blocking Signals
   • Alarms
   • Interval Timers

3 Error Handling
The function `setitimer()` sets value of an interval timer.

Syntax

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

The system provides each process with three interval timers, each decrementing in a distinct time domain.

When any timer expires, a signal is sent to the process, and the timer (potentially) restarts.

The time `ITIMER_PROF` in which profiles the time spent by the app in user and kernel space, sends `SIGPROF`

- `psValue` specifies timing
- `psOldValue` is irrelevant for our purposes
Interval Timers (cont’d)

- **Uses 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
Example
Execution Profiler

```c
#include <signal.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include "heavyload.h" // a HUGE for-loop

volatile int count = 0;

static void myHandler(int iSig)
{
    count++;
}
```
Example

Execution Profiler (Demo)

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

    pfRet = signal(SIGPROF, myHandler);
    if (pfRet != 0) { /* Something went wrong */
        exit(EXIT_FAILURE);
    }
    sTimer.it_value.tv_sec = 0;    /* A */
    sTimer.it_value.tv_usec = 10;   /* B */
    sTimer.it_interval.tv_sec = 0; /* C */
    sTimer.it_interval.tv_usec = 10; /* D */
    iRet = setitimer(ITIMER_PROF, &sTimer, NULL); /* start profiling */
    ... /* check for errors */
    heavyload();       // a HUGE for-loop
    printf("Time elapsed: %lf sec. \n", (double) count/1000);
    return 0;
}
```
### List of 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  
16) SIGSTKFLT  
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  
...
1 Introduction

2 Signals
   - Signal Handling
   - Race Conditions and Critical Sections
   - Blocking Signals
   - Alarms
   - Interval Timers

3 Error Handling
Basic error codes are defined in the standard library `<errno.h>`. The global `errno` variable is used by many functions to return error values. When opening a file with `fopen()` fails, the function only returns `NULL` without cause of the failure. The value of `errno` is defined only after a function call for which it is explicitly stated to be set. A string description of the numeric error code can be returned by `char* strerror( int errno)` from the standard library `<string.h>`. 
Example

errno, strerror

#include <stdio.h>
#include <errno.h>
#include <string.h>

int main () {
    FILE *f = fopen("nonexistent.txt", "r");
    if (f == NULL) {
        int r = errno;
        printf("Open file failed errno value %d\n", errno);
        printf("String error '%s' \n", strerror(r));
    }
    return 0;
}
Example

**errno, strerror**

- When the file does not exist,

**Output**

```
Open file failed errno value 2
String error 'No such file or directory'
```

- When the file does not have sufficient access rights,

**Output**

```
Open file failed errno value 13
String error 'Permission denied'
```
perror

While `strerror()` returns a pointer to the error message string, the function

```c
void perror(const char *s);
```

can be used to print a customized string along with the human-readable error message.

The following two implementations are equivalent:

```c
printf("fopen() : %s\n", strerror(errno));
perror("fopen() ");
```
## Other relevant `errno` codes

### Common `errno` codes,

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Macro</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>ENOMEM</td>
<td>Cannot allocate memory</td>
</tr>
<tr>
<td>17</td>
<td>EEXIST</td>
<td>File exists</td>
</tr>
<tr>
<td>20</td>
<td>ENOTDIR</td>
<td>Not a directory</td>
</tr>
<tr>
<td>21</td>
<td>EISDIR</td>
<td>Is a directory</td>
</tr>
<tr>
<td>27</td>
<td>EFBIG</td>
<td>File too large</td>
</tr>
<tr>
<td>28</td>
<td>ENOSPC</td>
<td>No space left on device</td>
</tr>
<tr>
<td>30</td>
<td>EROFS</td>
<td>Read-only file system</td>
</tr>
</tbody>
</table>
**errno with signals**

How to deal with errno and signal handler in Linux?

- The `errno` code becomes likely corrupted.
- A properly written signal handler saves and restores the value of `errno`.

**Sample code:**

```c
void signal_handler(int signo)
{
    int temp_errno = errno;
    /* code here may change the errno */
    errno = temp_errno;
}
```
In UNIX, the ___________ system call is used to send a signal.

1. sig
2. send
3. kill
4. sigsend