Development in C

Operating Systems UE
2022W

David Lung, Florian Mihola, Andreas Brandstätter,
Axel Brunnbauer, Peter Puschner

Technische Universität Wien
Computer Engineering
Cyber-Physical Systems

2022-10-11

Content

Part I (Oct 11)

- Create executables
  - Compile and link
  - Makefiles
- Program conventions
  - Argument parsing
  - Man pages
  - Error handling
  - Signals

Part II (Oct 13)

- Memory management
  - Memory Areas in C
  - Dynamic Memory Management
- Error detection
  - Avoid errors
  - Static and dynamic program analysis
  - Debugging with gdb

Part I

Program Creation and Conventions

Program Creation

Single source file

hello.c

```c
#include <stdio.h>

void say_hello(const char *name)
{
    printf("Hello %s!\n", name)
}

int main(int argc, char *argv[])
{
    say_hello("Michael");
    return 0;
}
```

```
$ gcc -o hello hello.c
```

Program Creation

Multiple source files

hello.h

```
void say_hello(const char *name);
```

main.c

```
#include "hello.h"

int main(int argc, char *argv[])
{
    say_hello("Michael");
    return 0;
}
```

hello.c

```
#include "hello.h"
#include <stdio.h>

void say_hello(const char *name)
{
    printf("Hello %s!\n", name)
}
```

First compile each source file, which creates object files (***.o**):

```
$ gcc -c -o main.o main.c      # slow
$ gcc -c -o hello.o hello.c    # slow
```

Then link the object files together, which creates an executable:

```
$ gcc -o hello main.o hello.o  # fast
```

Program Creation

Multiple source files

hello.h

```
void say_hello(const char *name);
```

main.c

```
#include "hello.h"

int main(int argc, char *argv[])
{
    say_hello("Michael");
    return 0;
}
```

hello.c

```
#include "hello.h"
#include <stdio.h>

void say_hello(const char *name)
{
    printf("Hello %s!\n", name)
}
```

After editing **hello.c**: no need to recompile **main.c**

```
$ gcc -c -o hello.o hello.c     # slow
$ gcc -o hello main.o hello.o   # fast
```

For large projects compilation of all source files takes a lot of time. Recompiling only files which have changed is much faster.

Makefiles

- Purpose of Makefiles:
  - Automate this process
  - Compile only files which have changed
- Structure of Makefiles: list of rules:

```
result: ingredients
    recipe
```

```
target: dependencies
    commands
```

- Example:

```
hello: main.o hello.o
    gcc -o hello main.o hello.o     # linking

main.o: main.c hello.h
    gcc -c -o main.o main.c         # compiling

hello.o: hello.c hello.h
    gcc -c -o hello.o hello.c       # compiling
```

- Execution:

```
$ make hello
```

Makefiles

make tool

`make`

- build automation tool
- executes commands specified via rules in a Makefile to update a target, e.g., a program (make(3))
- by default make searches for a file with the exact name **Makefile** in the current directory

- Builds the specified target (e.g., `$ make hello`)
- If no target was specified, the first target in the Makefile is built
- Checks if any of the dependencies (and possible dependencies of these dependencies) is more recent than the target
  - If not, then the target is up to date and there is nothing to do
  - Otherwise, all required commands to update the target are executed

Makefiles

Files:	main.c
	hello.c
	hello.h

How make works:

```
hello:  main.o hello.o
    gcc -o hello main.o hello.o

main.o:  main.c hello.h
    gcc -c -o main.o main.c

hello.o:  hello.c hello.h
    gcc -c -o hello.o hello.c
```

Start with a clean environment, no object files or executables have been built yet.

```
$ make
```

Makefiles

Files:	main.c
	hello.c
	hello.h

How make works:

```
hello:  main.o hello.o
    gcc -o hello main.o hello.o

main.o:  main.c hello.h
    gcc -c -o main.o main.c

hello.o:  hello.c hello.h
    gcc -c -o hello.o hello.c
```

- First target: **hello**
  - The file does not exist, so it must be built
  - The dependencies are **main.o** and **hello.o**
  - Neither of these files exists, so they must be built

Makefiles

Files:	main.c	**main.o**
	hello.c	**hello.o**
	hello.h	

How make works:

```
hello:  main.o hello.o
    gcc -o hello main.o hello.o

main.o:  main.c hello.h
    gcc -c -o main.o main.c

hello.o:  hello.c hello.h
    gcc -c -o hello.o hello.c
```

- First target: **hello**
- Search for a rule to build **main.o**
  - The dependencies are **main.c** and **hello.h**
  - There is no rule to build these dependencies, but they exist as files
  - Execute the commands to build **main.o**
- Do the same to build **hello.o**

Makefiles

Files:	main.c	main.o
	hello.c	hello.o
	hello.h	**hello**

How make works:

```
hello:  main.o hello.o
    gcc -o hello main.o hello.o

main.o:  main.c hello.h
    gcc -c -o main.o main.c

hello.o:  hello.c hello.h
    gcc -c -o hello.o hello.c
```

- First target: **hello**
- Search for a rule to build **main.o**
- Do the same to build **hello.o**
- Now all dependencies of **hello** are up to date
  - Execute the commands to build **hello**

Makefiles

Files:	main.c	8:10	main.o	9:20
	hello.c	**11:30**	hello.o	9:21
	hello.h	7:40	hello	9:22

How make works:

```
hello:  main.o hello.o
    gcc -o hello main.o hello.o

main.o:  main.c hello.h
    gcc -c -o main.o main.c

hello.o:  hello.c hello.h
    gcc -c -o hello.o hello.c
```

- Each file has a modification time stamp. Currently, the program executable **hello** is newer than any of its dependencies.
- We edit **hello.c**; all other files are left unchanged. Now the file **hello.c** is newer than the executable.

Makefiles

How make works:

Files:	main.c	8:10	main.o	9:20
	hello.c	11:30	hello.o	9:21
	hello.h	7:40	hello	9:22

```
hello:  main.o hello.o
    gcc -o hello main.o hello.o

main.o:  main.c hello.h
    gcc -c -o main.o main.c

hello.o:  hello.c hello.h
    gcc -c -o hello.o hello.c
```

- First target: **hello**
  - The dependencies are **main.o** and **hello.o**
  - **hello** is newer than **main.o** and **hello.o**; however we must check their dependencies as well
  - **main.o** is newer than **main.c** and **hello.h**; therefore it is up to date
  - **hello.o** is newer than **hello.h**, but it is older than **hello.c**; therefore we must rebuild **hello.o**

Makefiles

Files:	main.c	8:10	main.o	9:20
	hello.c	11:30	**hello.o**	**11:40**
	hello.h	7:40	hello	9:22

How make works:

```
hello:  main.o hello.o
    gcc -o hello main.o hello.o

main.o:  main.c hello.h
    gcc -c -o main.o main.c

hello.o:  hello.c hello.h
    gcc -c -o hello.o hello.c
```

- First target: **hello**
- Run the commands to build **hello.o**

Makefiles

How make works:

Files:	main.c	8:10	main.o	9:20
	hello.c	11:30	hello.o	11:40
	hello.h	7:40	**hello**	**11:41**

```
hello:  main.o hello.o
    gcc -o hello main.o hello.o

main.o:  main.c hello.h
    gcc -c -o main.o main.c

hello.o:  hello.c hello.h
    gcc -c -o hello.o hello.c
```

- First target: **hello**
- Run the commands to build **hello.o**
- Now **hello.o** is newer than **hello**; therefore **hello** must be rebuilt as well
  - Execute the commands to build **hello**

Makefiles

Makefile variables

- Automatic variables
  - $@: target
  - $<: first dependency
  - $^: all dependencies
- Custom variables

```
CC = gcc

hello: main.o hello.o
    $(CC) -o $@ $^

main.o: main.c hello.h
    $(CC) -c -o $@ $<

hello.o: hello.c hello.h
    $(CC) -c -o $@ $<
```

Makefiles

Makefile variables

- Automatic variables
  - $@: target
  - $<: first dependency
  - $^: all dependencies
- Custom variables
- Pattern rules (add dependency list to include headers!)

```
CC = gcc

hello: main.o hello.o
    $(CC) -o $@ $^

%.o:  %.c
    $(CC) -c -o $@ $<

main.o:  main.c hello.h
hello.o:  hello.c hello.h
```

Makefiles

Makefile conventions

- Standard rules:
  - **all**: points to the target(s) to be built by default
  - **clean**: remove all files produced during the build process
- PHONY targets: no file is created (rule is executed without checking for a corresponding file)
- Variables for object files, compiler flags and linker flags

```
CC = gcc

.PHONY: all clean
all:  hello

hello: main.o hello.o
    $(CC) -o $@ $^

%.o: %.c
    $(CC) -c -o $@ $<

clean:
    rm -rf *.o hello
```

Makefiles

Complete Example

```
CC = gcc
DEFS = -D_BSD_SOURCE -D_SVID_SOURCE -D_POSIX_C_SOURCE=200809L
CFLAGS = -Wall -g -std=c99 -pedantic $(DEFS)

OBJECTS = main.o hello.o

.PHONY: all clean
all: hello

hello: $(OBJECTS)
    $(CC) $(LDFLAGS) -o $@ $^

%.o: %.c
    $(CC) $(CFLAGS) -c -o $@ $<

main.o: main.c hello.h
hello.o: hello.c hello.h

clean:
    rm -rf *.o hello
```

Shell Commands

```
gcc -c -o hello.o hello.c
make clean
ls -l -a /usr/bin
cd ~/Documents
```

. . . contain:

- program name or path to the executable
- options that control the behaviour of the command
- positional arguments / operands that are objects on which the command is executed upon (e.g., files, directories)

Options

- Options come before other arguments
- Option short form: `-` followed by **one** character
  - Example: `-c`
  - Combine several options: `-l -a` $\rightarrow$ `-la`
- Option long form: `--` followed by a string
  - Example: `--version`
- Order of options usually irrelevant
- Options usually occur at most once

Option-Arguments

- Options can require an argument
  - Example short form:

    ```
    gcc -o hello main.o hello.o
    make -f ~/Documents/Makefile all
    ```

  - Example long form:

    ```
    make --file=~/Documents/Makefile all
    ```

- How to differ between option-arguments and positional arguments? ⇒ end of the options list, indicated by:
  - A string that is not an option (does not start with `-` or `--`)

    ```
    ls ~/Documents
       ↑
    gcc -o main.o main.c
               ↑
    ```

  - A string which is not an argument of an option (i.e., a string following an option which takes not argument)

    ```
    gcc -c main.c hello.c
        ↑
    ```

  - The string `--`

    ```
    rm -- -f        → deletes file '-f'
       ↑
    ```

Pass Arguments to `main`

- Function prototype for `main`:

```
int main (int argc, char **argv);
```

or:

```
int main (int argc, char *argv[]);
```

(both are equivalent)

argc Number of elements in `argv`
argv Array of command line arguments (`argv[0]...argv[argc-1]`)

- `argv[0]` is usually the program name or path to the executable (but not necessarily)
- `argv[argc]` is always `NULL` according to the standard
- Value of `argc`: number of command line arguments + 1

Pass Arguments to `main`

Examples

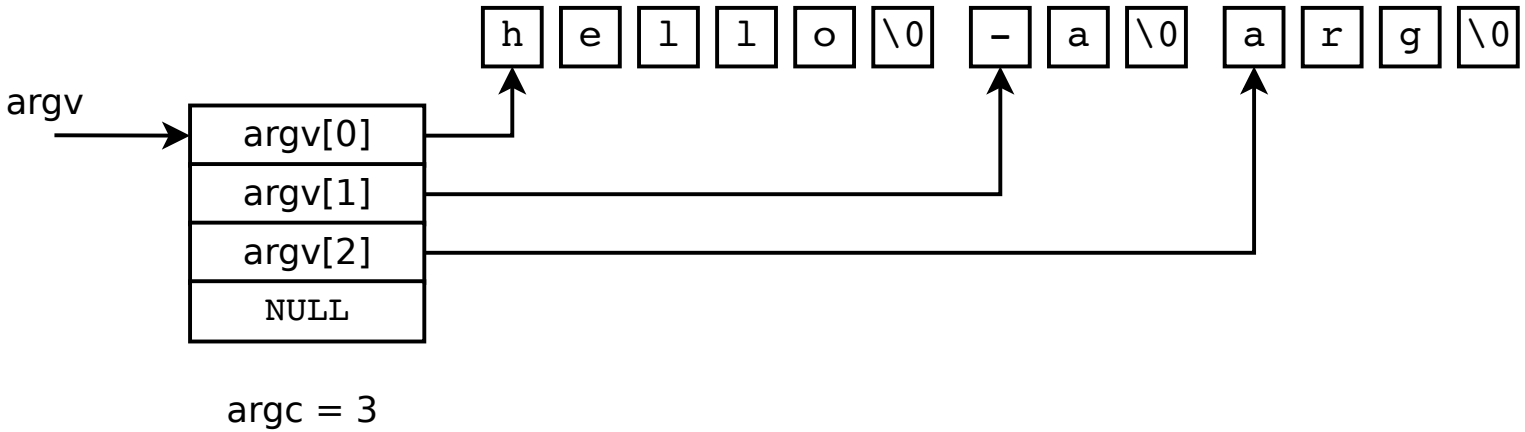
Value of `argc` and `argv` ...

- `make`
  - argc: 1
  - argv[0]: `make`
- `gcc -c main.c hello.c`
  - argc: 4
  - argv[0]: `gcc`
  - argv[1]: `-c`
  - argv[2]: `main.c`
  - argv[3]: `hello.c`
- `hello "-f -o hello.txt"`
  - argc: 2
  - argv[0]: `hello`
  - argv[1]: `-f -o hello.txt`

Pass Arguments to `main`

`char **`

- `hello -a arg`



Handle Options with `getopt`

- Use function `getopt()` for handling options of a C program[1]

```
int getopt(int argc, char *const argv[],
           const char *optstring);
```

`optstring` Specification of valid options as string

- Specification of legitimate option characters
  - Example: `"a:o"`
    - `a:` Option that requires an argument
    - `o` Option without an argument
  - Order is usually irrelevant (e.g., `"oa:"`)

[1] getopt_long() supports also long option names

`getopt` - Usage

- Call `getopt()` repeatedly
  - Returns option characters consecutively
  - Invalid option: outputs an error message and returns `'?'`
  - No more option character: returns -1
  - `optarg` contains option-argument (`char *`)
  - `optind` is the index of the next element in `argv` (`int`)

- Task of the programmer:
  - Count occurence of an option
  - Handle invalid options
  - Save option-arguments
  - Check for correct number of positional arguments

getopt - Usage

Example: Program with options a (with argument) and o (without argument)

```
char *a_arg = NULL;
int opt_o = 0;
int c;
while ( (c = getopt(argc, argv, "a:o")) != -1 ){
  switch ( c ) {
    case 'a': a_arg = optarg;
      break;
    case 'o': opt_o++;
      break;
    case '?': /* invalid option */
      break;
  }
}
if ( a_arg == NULL )
  /* option 'a' did not occur */

if ( opt_o > 1 )
  /* option 'o' occurs more than once */
if ( opt_o != 1 )
  /* option 'o' does not occur exactly once */
```

getopt - Positional Arguments

Example: Program which requires 2 positional arguments

optind: index of next argument to be parsed by getopt()

- When getopt() is done, optind is the index of the first positional argument

```
hello -a optarg arg1 arg2
                ↑
```

- After option handling with getopt(): argc = 5, optind = 3

```
while ( (c = getopt(argc, argv, "a:o")) != -1 ) {
  switch ( c ) {
    ...
  }
}
if ( (argc - optind) != 2 )
  /* number of positional arguments is not 2 */

char *input  = argv[optind];
char *output = argv[optind+1];
```

Usage Message

Usage Message

= documents correct calling interface, e.g.,

```
Usage:  myprog [-n] file...
```

(Syntax also used in man pages, so-called Synopsis.)

- Optional arguments marked with `[]`
  ```
  hello -a optarg [-o] arg1
  ```
- Alternative options denoted as `[x|y]`
  ```
  hello [-a optarg | -o] arg1
  ```
- Alternative required arguments denoted as `{x|y}`
  ```
  hello {-a optarg | -o} arg1
  ```
- Conditionally valid options
  ```
  hello [-a optarg [-o]] arg1
  ```
- One or more occurence of a positional argument
  ```
  hello -a optarg file...
  ```

Usage Message

Example

```
char *myprog;

void usage(void) {
  fprintf(stderr,"Usage: %s [-a file] file\n", myprog);
  exit(EXIT_FAILURE);
}

int main (int argc, char *argv[]) {
  myprog = argv[0];
  int c;
  while ( (c = getopt(argc, argv, "a:")) != -1 ) {
    switch ( c ) {
      ...
    }
  }
  if ( ... ) /* user did not correctly call program */
    usage();

  ...
}
```

Man Pages

Man Pages

= collection of help and documentation pages (manual).

- Divided into several chapters, here important:
  1. Command line programs
  2. System calls (C functions)
  3. Library calls (C functions)
  7. Miscellaneous

Man Pages

- Example: "For more information see make(1)"
  - Meaning: "Further information can be found in the manual page of `make` in chapter 1."
  - Read man page in Linux:
    `man make` or `man 1 make`
- Different man pages with same name, e.g.:
  - getopt(1): the shell command
  - getopt(3): the C function
- Searching the man pages
  - `man -k keyword`
  - `apropos keyword`

Error Handling

- A function can indicate an error through its return value
  - Many functions return -1 or `NULL` on error and set the global variable `errno` (errno(3))
  - `strerror()`: returns a human-readable description of the error code (strerror(3))
- **Check the return value if subsequent code depends on the successful execution of a function!**

```
FILE *file;
if ( (file = fopen("input.txt", "r")) == NULL )
  fprintf(stderr, "fopen failed: %s\n", strerror(errno);
```

- If a function indicates an error:
  - In general: recovery strategy
  - In this course: write an error message to `stderr`, followed by proper termination (freeing all resources and exiting with `EXIT_FAILURE`)

Error Handling

- Meaningful error messages:
  - Which program? (`argv[0]`)
  - What problem? (e.g., "fopen failed")
  - Cause? (`strerror(errno)`)
- Termination of the program
  1. Free all resources (memory, files, ...)
  2. Call `exit()` with correct exit code as argument (exit(3))
     - ✓ `EXIT_SUCCESS` on successful termination
     - ✗ `EXIT_FAILURE` on error

```
FILE *file;
if ( (file = fopen("input.txt", "r")) == NULL ) {
  fprintf(stderr, "[%s] ERROR: fopen failed: %s\n",
          prog_name, strerror(errno));
  exit(EXIT_FAILURE);
}
```

```
[./prog] ERROR: fopen failed: No such file or directory
```

Signals

Signal

= notification to the process about an event [?]

- Event generates signal
- Defined via numbers with symbolic name: `SIGxxxx` (see signal(7))

Signals

Sources

- Hardware exception, e.g.,
  - Invalid memory access (SIGSEGV)
  - Division by 0 (SIGFPE)
- Software event, e.g.,
  - Termination (SIGTERM)
  - Child process terminated (SIGCHLD)
  - User-defined, e.g., for synchronization of processes (SIGUSR1, SIGUSR2)
- User sends the signal via terminal, e.g.,
  - Interrupt, e.g., via keyboard (SIGINT, <Ctrl>-C)
  - Quit from keyboard (SIGQUIT, <Ctrl>-\)

- Commands to create signals, e.g.,
  - kill(1), kill(2), raise(3), abort(3)
  - Examples:
    `kill 1521` (SIGTERM)
    `kill -9 1521` (SIGKILL)

Signals

Blocking System and Library Calls

- Signals interrupt operating system routines

```
while ((cnt = read(fd, buf, BUF_SIZE)) == -1) {
    if (errno != EINTR) {
        /* read failed with another error than EINTR */
    }
}
```

Handle Signals

- Default actions: ignore, terminate, core dump, suspend, resume
- Handling can be configured for most of the signals (exceptions: SIGKILL, SIGSTOP)
  - Default action / ignore / individual signal handler
  - Configuration via sigaction(2)
- Delivery can be blocked via signal mask of the process

Signal Configuration

```
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

- Configures handling of the signal `signum`
- If `act` $\neq$ NULL, the given configuration will be used (set configuration)
- If `oldact` $\neq$ NULL, the former configuration will be saved into `oldact` (this can also be used to read the current configuration by setting `act` $=$ NULL)

Signal Configuration

```
#include <signal.h>

struct sigaction {
    void     (*sa_handler)(int);
    sigset_t   sa_mask;
    int        sa_flags;
    ...
};
```

sa_handler	Address of signal handler, or `SIG_DFL` (default handler), or `SIG_IGN` (ignore)
sa_mask	Signals that should be blocked while the handler for this signal is executed
sa_flags	Additional options (see man page)

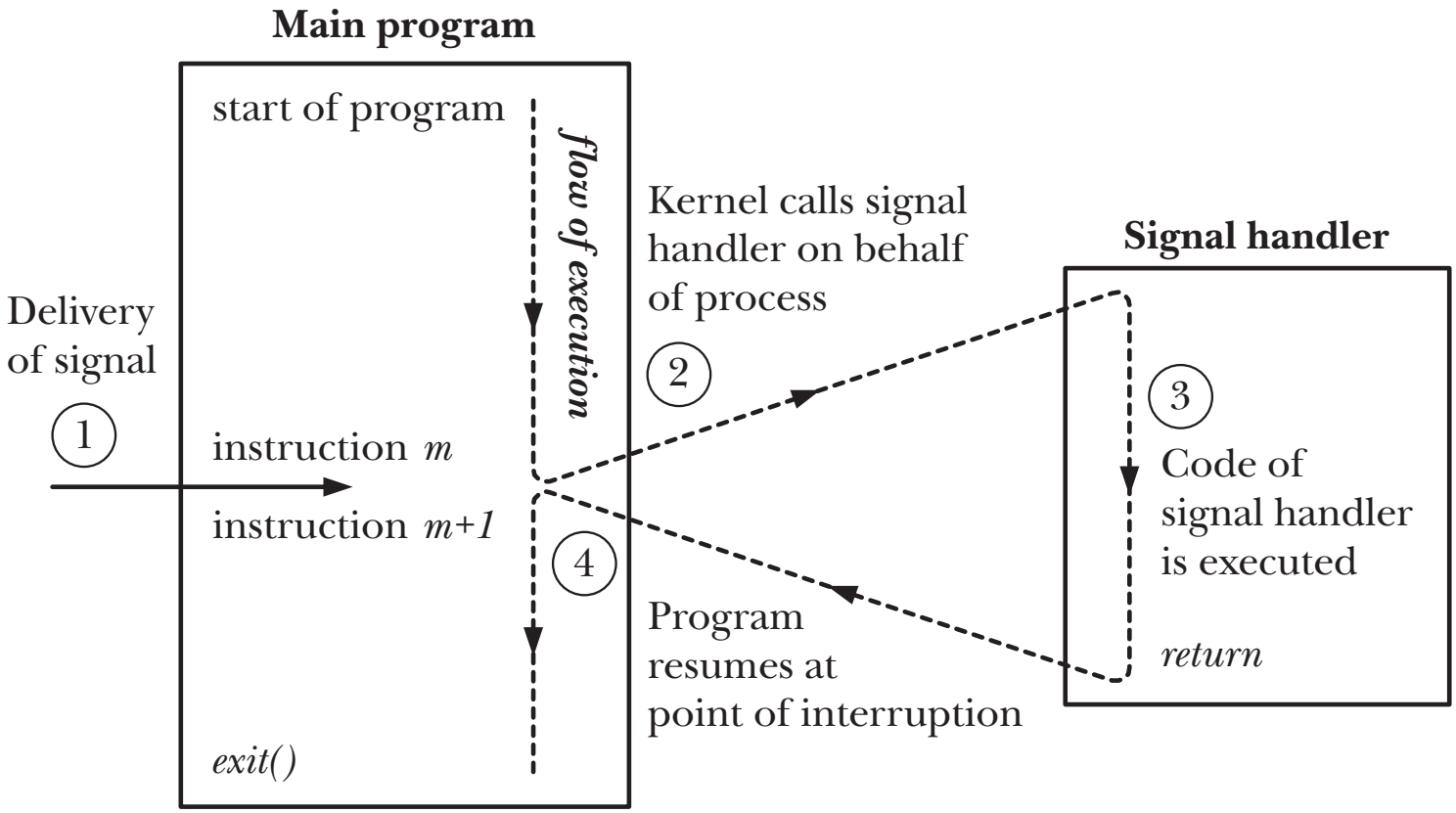
Signal Configuration

Example

```
void handle_signal(int signal)
{
    ...
}

int main(int argc, char *argv[])
{
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa)); // initialize sa to 0

    sa.sa_handler = handle_signal;

    sigaction(SIGINT, &sa, NULL);

    ...
}
```

Signal Handler

Procedure



Source: Michael Kerrisk - The Linux Programming Interface, p. 399

Signal Handler

- C function

```
void handle_signal(int signal);
```

- Reaction to asynchronous events
  - Keep signal handler as simple as possible
  - Be careful with the usage of static variables (consistency)
  - Handler can be interrupted by another signal (block via signal mask)
  - In the signal handler use async-signal-safe functions only.

async-signal-safe functions

= functions that are allowed to be called within a signal handler.
Beside others, do not use: `malloc()`, `printf()`, `exit()`, etc. in a signal handler → comply with the list in signal(7).

Signal Handler

Example 1

- Terminate via `_exit(2)`

```
void handle_signal(int signal)
{
    _exit(1);
}

int main(int argc, char **argv)
{
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa)); // initialize sa to 0

    sa.sa_handler = handle_signal;
    sigaction(SIGINT, &sa, NULL);

    ...
}
```

Signal Handler

Example 2

- Set global quit flag (must be of special data type: `sig_atomic_t` and must be declared `volatile`, otherwise compiler might optimize it away)

```
volatile sig_atomic_t quit = 0;

void handle_signal(int signal)
{
    quit = 1;
}

int main(int argc, char **argv)
{
    struct sigaction sa;
    memset(&sa, 0, sizeof(sa)); // initialize sa to 0
    sa.sa_handler = handle_signal;
    sigaction(SIGINT, &sa, NULL);

    while (!quit) {
        ...
    }
    return 0;
}
```

Material

- GNU Make Manual
  `http://www.gnu.org/software/make/manual/html_node/`
- Utility Conventions for Argument Syntax
  `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap12.html`
- The GNU C Library Reference Manual, Ch. 24 (signals)
  `http://www.gnu.org/software/libc/manual/html_node/`