BasicOS

Introduction

Ludovic Apvrille ludovic.apvrille@telecom-paris.fr
Eurecom, office 470

https://perso.telecom-paris.fr/apvrille/BasicOS/

Outline

Introduction to OS

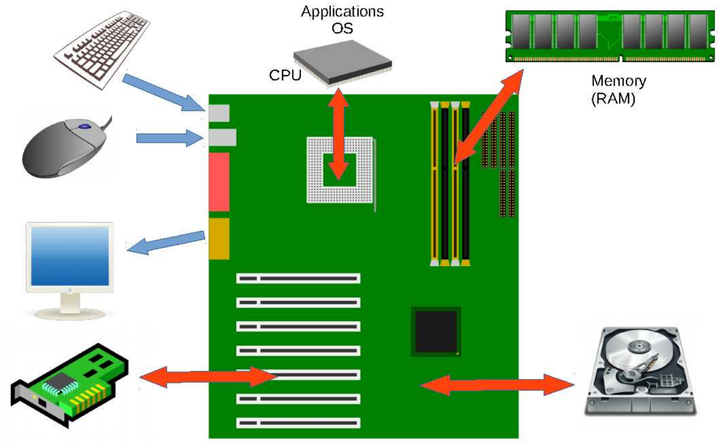
The Basics of C

Advanced Concepts in C

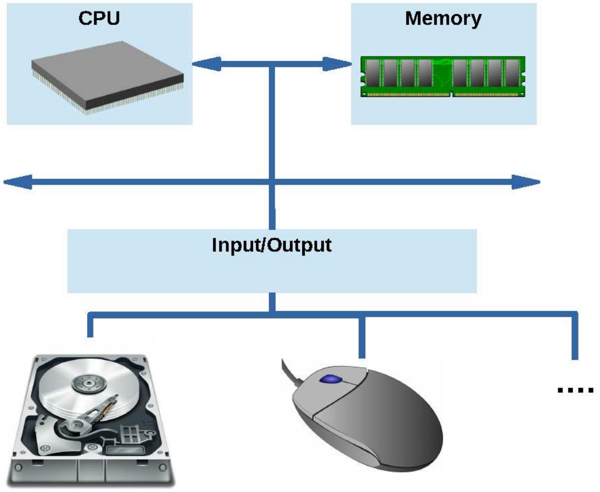
Protection and System Calls

What is a Computer System?

In other words: what are the main components of a PC?



Computer System: Simplified View



What is an Operating System?

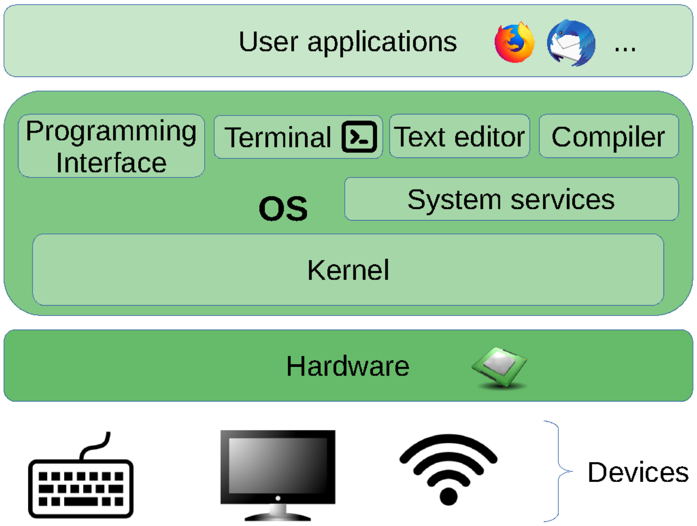
Definition

The most fundamental program of a computer system

Objectives

- **Make computers convenient to use** i.e. simplify programmers' tasks
  - Abstract hardware concerns
    - e.g., simplify memory allocations
- **Use hardware in an efficient manner**
- **Security**
  - Protect systems from wrong and malicious utilizations

Layers of a Computer System



User applications ...

Programming Interface

Terminal

Text editor

Compiler

OS

System services

Kernel

Hardware

Devices

Main Services

- Program execution
- Resource allocation and release
- I/O operations
- Files handling
- Communication
  - Between programs running on the same computer
  - Between programs running on different computers
- Error detection or handling
  - Hardware failure, illegal memory access, illegal instruction, exception (divide by zero)
- Accounting
- Security

While ensuring . . .

- Ease of use
- Efficiency
- System protection

Operating Systems: a Chronology

1950 → 1960: transistors

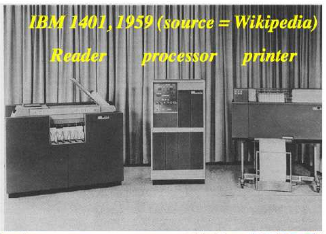
- First OS written in assembly language

1970 → 1980: integrated circuits

- From millions of code of assembly language → C
- CPU and memory partitioning
- Genenis of UNIX

1980 → now: large scale integrated circuits

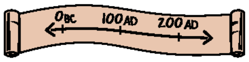
- Graphical user interfaces, networking
- GNU/Linux, Windows, macOS, Solaris, Android, . . .



Apple II, 1977-1988

(source = Wikipedia)

UNIX: History



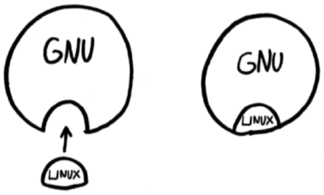
Idea originated in 1965

- Research lab of AT&T (Bell Labs)
- Idea of Ken Thompson: develop what no computer company was ready to provide i.e. a multi-user and multiprocessing OS
- Multics created in cooperation with MIT and General Electric
- Less complex version of Multics: UNIX, operational at Bell Labs in 1971
  - Fully written in assembly language

Diffusion in acamedia and companies

- Code is modified by graduate students to make UNIX more robust
- Rewritten in C

GNU/Linux (Free Software)



GNU/Linux (a.k.a. Linux) = GNU Operating System + the Linux kernel

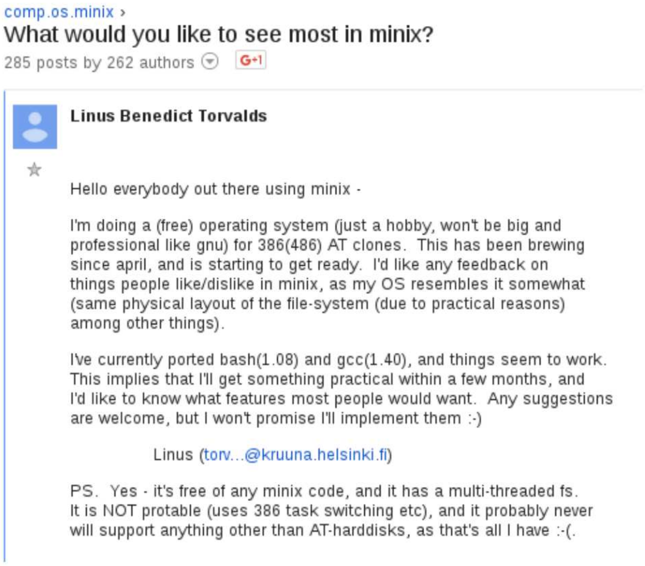
The GNU Operating System

- *GNU's Not Unix!*
- Applications, libraries, and developer tools
- Started in 1984

The Linux Kernel

- Created in 1991 by Linus Torvalds
- *See next slide*

First Post by Linus Torvald



comp.os.minix ›

What would you like to see most in minix?

285 posts by 262 authors

Linus Benedict Torvalds

Hello everybody out there using minix -

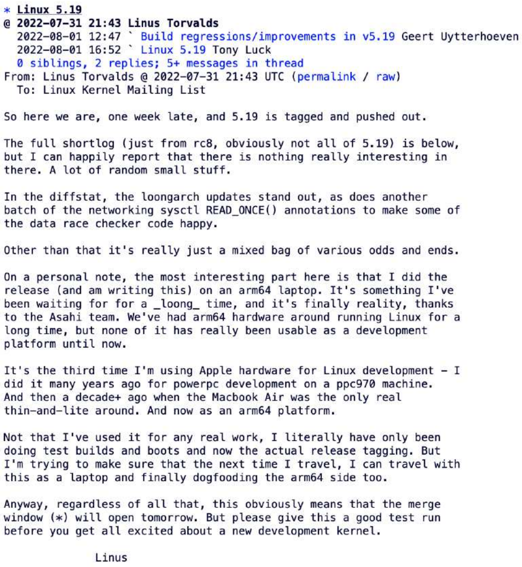
I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torv...@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT protable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.

And a Recent Post by Linus Torvald. . .



```
* Linux 5.19
@ 2022-07-31 21:43 Linus Torvalds
  2022-08-01 12:47 ` Build regressions/improvements in v5.19 Geert Uytterhoeven
  2022-08-01 16:52 ` Linux 5.19 Tony Luck
  0 siblings, 2 replies; 5+ messages in thread
From: Linus Torvalds @ 2022-07-31 21:43 UTC (permalink / raw)
  To: Linux Kernel Mailing List

So here we are, one week late, and 5.19 is tagged and pushed out.

The full shortlog (just from rc8, obviously not all of 5.19) is below,
but I can happily report that there is nothing really interesting in
there. A lot of random small stuff.

In the diffstat, the loongarch updates stand out, as does another
batch of the networking sysctl READ_ONCE() annotations to make some of
the data race checker code happy.

Other than that it's really just a mixed bag of various odds and ends.

On a personal note, the most interesting part here is that I did the
release (and am writing this) on an arm64 laptop. It's something I've
been waiting for for a _loong_ time, and it's finally reality, thanks
to the Asahi team. We've had arm64 hardware around running Linux for a
long time, but none of it has really been usable as a development
platform until now.

It's the third time I'm using Apple hardware for Linux development - I
did it many years ago for powerpc development on a ppc970 machine.
And then a decade+ ago when the Macbook Air was the only real
thin-and-lite around. And now as an arm64 platform.

Not that I've used it for any real work, I literally have only been
doing test builds and boots and now the actual release tagging. But
I'm trying to make sure that the next time I travel, I can travel with
this as a laptop and finally dogfooding the arm64 side too.

Anyway, regardless of all that, this obviously means that the merge
window (*) will open tomorrow. But please give this a good test run
before you get all excited about a new development kernel.

              Linus
```

Outline

Introduction to OS

The Basics of C

Advanced Concepts in C

Protection and System Calls

C

C in a nutshell

- Developed by Dennis Ritchie, early 70s, for UNIX
- Low-level language
  - Direct manipulation of memory addresses
  - Incorporate assembly language

Why programming in C (and not in python, . . .)?

Partially covered

- Basic control structure (*for*, *if*, etc.)
- Macros
- Compilation, multi-file project

Covered

- Library functions and system calls
- Pointers and memory allocations
- Characters and strings

Helloworld in C

```
#include <stdio.h>

int main() {
    printf("Hello World!\n");
    return 0;
}
```

- *printf* is a function of the C library (a.k.a. "libC")
- Use "man" to have an information on a function:

```
$ man -s 3 printf
PRINTF(3)                                          Linux Programmer s

NAME
       printf, fprintf, ...

SYNOPSIS
       #include <stdio.h>
       ....
```

Compilation and Execution

- Compilation transforms a C program into machine language
- Files is compiled for the host Operating System

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

- Execution creates a process in the OS

```
$ ./hello
Hello World!
```

Enhanced Helloworld in C

- Taking as argument a first name

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    if (argc < 2) {
        printf("Usage: ./hello <First Name>\n");
        return 1;
    }

    printf("Hello %s!\n", argv[1]);
    return 0;
}
```

Enhanced Helloworld in C (Cont.)

- Taking as argument "Last_First" names, and printing "Hello First Last!"

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

void usage() {
    printf("Usage: ./HelloFirstLastName <Lastname_Firstname>. Maximum size
              of input: 49 characters\n");
}

int main(int argc, char *argv[]) {
    if (argc < 2) {
        usage();
        return 1;
    }

    if (strlen(argv[1]) >= 50) {
        usage();
        return 1;
    }
```

Enhanced Helloworld in C (Cont.)

```
    int index = 0;
    char *total = argv[1];
    int max = strlen(total);
    int found = 0;

    while(index < max) {
      if (total[index] == '_') {
            found = 1; break;
      }
      index ++;
    }

    if (found == 0) {
      usage();
      return -1;
    }
}
```

Enhanced Helloworld in C (Cont.)

```
    char firstName[50], lastName[50];

    memcpy(lastName, total, index);
    lastName[index] = '\0';
    memcpy(firstName, &total[index+1], max-index);
    firstName[max-index] = '\0';

    printf("Hello %s %s!\n", firstName, lastName);
    return 0;
}
```

Outline

Introduction to OS

The Basics of C

Advanced Concepts in C

Protection and System Calls

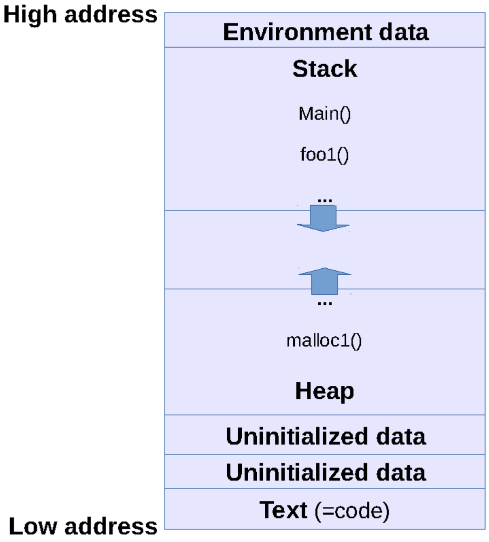
Process Data

- Data of processes are stored in memory

Various data of a process

- **Program code** = text section (static)
- **Current Activity**
  - Program counter = Processor's register
  - Next instruction to execute
- **Stack**: function calls are stored in a LIFO manner
  - Function parameters
  - Return address
  - Local variables
- **Heap**: Data section

Memory Layout of a C Program



High address

Environment data

Stack

Main()

foo1()

...

...

malloc1()

Heap

Uninitialized data

Uninitialized data

Text (=code)

Low address

Memory Allocation in C Programs

Allocation in the stack

Function calls, function parameters, local variables

```
myLovelyFunction(int y) {
    char tab[50];
    ...
}
```

Allocation in the heap

Memory allocations with *malloc()*, disallocation with *free()*

```
char * myLovelyFunction(int size) {
    char * name = (char *) ( malloc (sizeof(char) * size));
    ...
    return name;
}
...
free(name);
```

Memory Allocation in C Programs: Example

```c
int a;

int funnyAllocation(char *buf, int b) {
  a = 5;
  b = b + 1;
  strcpy(buf, "hello");
  return 7;
}

int main( int argc, char*argv[] ) {
  int b = 3;

  char *buf = (char *) ( malloc(sizeof(char) * 20) );

  int returned = funnyAllocation(buf, b);

  printf("The returned value is: %d\n", returned);
  printf("The value of b is: %d\n", b);
  printf("The content of buf is: %s\n", buf);
  free(buf);
}
```

Memory Allocation in C Programs (Cont.)

```
$ gcc -Wall -o procmem procmem.c

$ ./procmem
The returned value is: 7
The value of b is: 3
The content of buf is: hello
```

Values (*) and Addresses (&)

```c
void updateValue(int *p) {
    *p = 10;
}

int main() {
    int x = 5;

    printf("Before. Value of x: %d\n", x);
    printf("Before. Adress of x: %p\n", &x);

    updateValue(&x);

    printf("After. Value of x: %d\n", x);
    printf("After. Address of x: %p\n", &x);

    return 0;
}
```

Values (*) and Addresses (&) (Cont.)

```
$ ./pointers
Before. Value of x: 5
Before. Adresse of x: 0x7ff7be3193c8
After. Value of x: 10
After. Address of x: 0x7ff7be3193c8
```

Memory Allocation Error

```
void updateValue(int *p) { *p = 10; }

int main() {
    int *x = (int *)1200000;

    printf("Before. Value of x: %d\n", *x);
    printf("Before. Adresse of x: %p\n", x);

    updateValue(x);

    printf("After. Value of x: %d\n", *x);
    printf("After. Address of x: %p\n", x);
}
```

```
$ ./pointers
Segmentation fault: 11
```

How to solve this problem?

Structures

```
#include <stdio.h>
#include <math.h>

typedef struct {
    double x;
    double y;
} Point;

// Function to compute the distance between two points
double distance(Point a, Point b) {
    double dx = a.x - b.x;
    double dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
}

int main() {
    Point p1; p1.x = 0.0; p1.y = 0.0;
    Point p2 = {3.0, 4.0};
    printf("The distance between p1 and p2 is: %f\n", distance(p1, p2));
    return 0;
}
```

Structures and pointers

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

typedef struct {
    double x;
    double y;
} Point;

// Function to compute the distance between two points
double distance(Point *a, Point *b) {
    double dx = a->x - b->x;
    double dy = a->y - b->y;
    return sqrt(dx * dx + dy * dy);
}

int main() {
    Point *p1 = (Point*) malloc(sizeof(Point));
    Point *p2 = (Point*) malloc(sizeof(Point));
    ...
}
```

Structures and pointers (Cont.)

```
    if(p1 == NULL || p2 == NULL) {
        printf("Memory not allocated.\n");
        return 1;
    }
    p1->x = 0.0; p1->y = 0.0; p2->x = 3.0; p2->y = 4.0;

    printf("The distance between p1 and p2 is: %f\n", distance(p1, p2));
    free(p1); free(p2);
    return 0;
}

$ ./distance
The distance between p1 and p2 is: 5.000000
```

Outline

Introduction to OS

The Basics of C

Advanced Concepts in C

Protection and System Calls

Hardware Protection

Protection of what?

- Devices
  - Prevent illegal use of devices
- Memory
  - Prevent a process from accessing the memory of the OS and of another processes
- CPU
  - Prevent illegal instructions
  - Prevent a process from jeopardizing processing resources

→ Dual Mode

One hardware protection is called **Dual Mode**

Dual Mode of Processors

User mode

Privileged assembly instructions cannot be executed

- → If so, the system raises an interrupt

Monitor mode

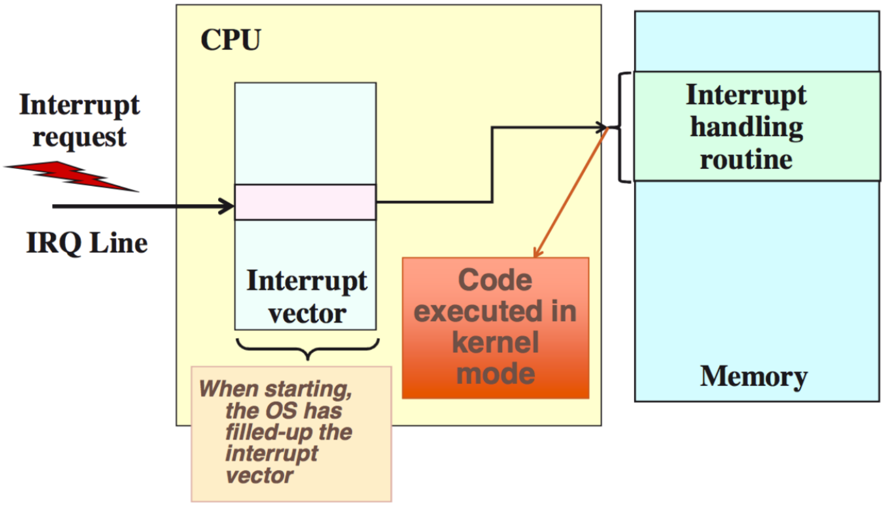
= Supervisor mode, system mode, privileged mode, kernel mode, etc.

- In this mode, privileged assembly instructions can be executed
- **Not** related at all to the *administrator* or *root* of a machine

Mode switching

- Monitor mode → user mode: a specific assembly instruction
- User mode → monitor mode: interrupt (a.k.a. "trap")

Interrupts



When starting, the OS has filled-up the interrupt vector

Protection: Use of Dual Mode

1. Hardware starts in monitor mode
2. OS boots in monitor mode
3. OS starts user processes in user mode
   - So, user processes cannot execute privileged instructions
4. When an interrupt occurs:
   - Hardware switches to monitor mode
   - Routine pointed to by interrupt vector is called
     - Vector was setup by the OS at boot time

⇒

The Operating System is in monitor mode whenever it gains control, i.e., when its code is executed in the CPU

Hardware Protection

Goals

Prevent instructions that shall not be executed

- Divide by zero, privileged instruction in user mode, access to a bad memory access

Mechanisms

- Hardware detects illegal instructions and accordingly generates interrupts
- The control is transferred to the OS
  - Faulty program is aborted
  - Error message (popup window, message in console or terminal)
  - Program's memory may be dumped for debug purpose
    - Under Unix, it is dumped to a file named *core*
- If faulty element = OS: blue screen, kernel panic, . . .

Manual pages

Help on functions provided by the OS

- Section 1: shell functions
- Section 2 : system calls (see next slides)
- Section 3: functions of LibC (library for C programs)

Examples

```
$ man ls
LS(1)                                                        User Commands
          ls — list directory contents
...
$ man sleep
SLEEP(1)                                                     User Commands
          sleep — delay for a specified amount of time
...
$ man —s3 sleep
SLEEP(3)                                             Linux Programmer s Manual
          sleep — sleep for a specified number of seconds
```

System Calls (a.k.a. ”Syscalls”)

Definition

- Interface between user processes and the Operating System
- Executed in monitor mode → ability to execute privileged instructions

- Windows: systems calls are included in the Win32/Win64 API
- Solaris

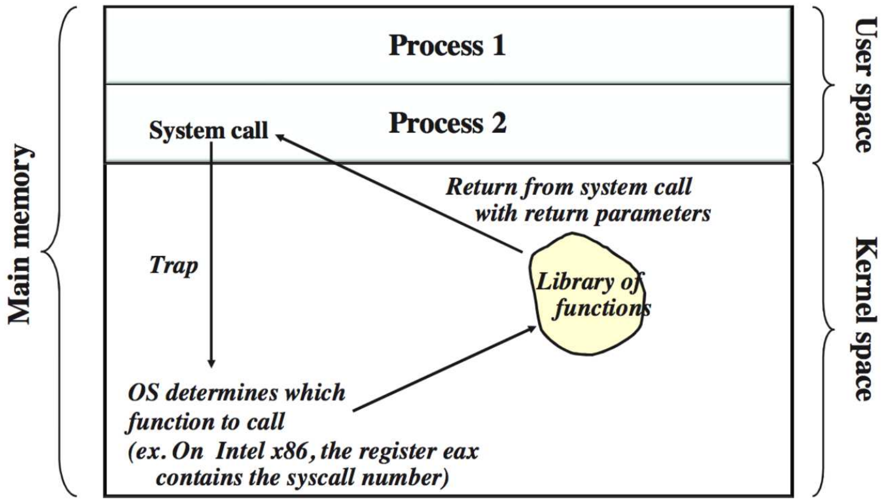
```
$ man —s2 read
System Calls
NAME
            read, readv, pread — read from file
SYNOPSIS
#include <unistd.h>
ssize_t read(int fildes, void *buf, size_t nbyte);
...
```

- macOS (Similar result in GNU/Linux)

```
$ man —s2 read
READ(2)   BSD System Calls Manual   READ(2)

NAME
       pread, read, readv —— read input
...
```

System Calls: Implementation



Main memory

Process 1

Process 2

System call

User space

Kernel space

Return from system call with return parameters

Trap

Library of functions

OS determines which function to call (ex. On Intel x86, the register eax contains the syscall number)

Categories of System Calls

- Process control
  - Create, allocate and free memory, exit, . . .
- File manipulation
  - Create, open, close, read, write, attributes management, . . .
- Device manipulation
  - Request, read, write, attributes management, . . .
- Getting and setting system related information
  - Time management, process management, . . .
- Communications
  - Send or receive messages, create communication links, . . .

System Calls: an Example

Objective: *Making a program that takes as input a text and a path to a file and that writes the text to the specified file*

```c
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>
#include <string.h>


int main(int argc, char*argv[]) {

  int out_fd;
  int written;

  if (argc < 3) {
    printf("usage: writeToFile <file> <text>\n");
    exit(1); // in bash,  a non zero code means an error
  }

  char *file = argv[1];

  if ( (out_fd = open(file, O_WRONLY | O_SYNC | O_CREAT)) < 0) {
    printf("Could not open the file %s\n", file);
    exit(1);
  }

  ...
```

System Calls: an Example (Cont.)

```c
  char * toBeWritten = argv[2];

  written = write(out_fd, toBeWritten, strlen(toBeWritten) );

  if (written < strlen(toBeWritten)) {
    printf("Write in file %s failed\n", file);
    exit(1);
  }

  if (close(out_fd) < 0) {
    printf("Could not close the file %s\n", file);
  }

  printf("Text %s successfully written to %s\n", toBeWritten, file);
  exit(0);
}
```

System Calls: an Example (Cont.)

- Compilation, execution (in GNU/Linux)

```
$ gcc −Wall −o writeToFile writeToFile.c

$ ./writeToFile /tmp/test helloworld
Text helloworld successfully written to /tmp/test

$ cat /tmp/test
helloworld
```

- Another way to do (i.e., without our program):

```
$ echo hellothere > /tmp/test

$ cat /tmp/test
hellothere
```