OPERATING SYSTEMS TEACHING NOTES

1 TABLE OF CONTENTS

2 SETUP

2.1 COURSE DETAILS

2.1.1 LECTURE 1

1. Course site
    a. Graded work
    b. Final grade calculation
    c. Attendance requirements
    d. Practical assignments grading
2. Book
3. Course structure
    a. Start with practical aspects
        i. UNIX Command line usage
        ii. UNIX Shell programming
        iii. UNIX Processes
        iv. UNIX Threads
    b. Finish with theoretical aspects
        i. Scheduling
        ii. Memory management
        iii. File systems
        iv. Boot
        v. etc

2.1.2 SEMINAR 1

1. Attendance requirements (see course website)

2.1.3 LAB 1

1. Attendance requirements (see course website)
2. Practical assignments grading (see course website)

2.2 LINUX SETUP

1. You already have a **Linux** computer. Great! Make sure you use exclusively the command line for this class. The practical tests and exams are done exclusively in the command line, so you need to be proficient in using it. You still need to install a few programs: emacs, valgrind, gcc, gdb, zip, unzip, and man pages
2. You have a **MacOS** computer. Also great! Make sure of the same thing above and install the same programs as above.
3. You have a **Windows** computer
    a. We do not recommend installing Linux on the same computer as Windows because you risk losing your files, especially if you are totally new at this.
    b. A safe choice is using virtualization (ie VirtualBox or vmWare). Follow the instructions at the link below, paying special attention to everything written there.
        http://www.cs.ubbcluj.ro/~rares/course/os/res/env-setup/windows/index.html
    c. If you have Windows 10, you can also install the Ubuntu Linux module from Windows Store
4. You can also get a free small Linux VM (more than sufficient for our needs) in the AWS cloud
    https://aws.amazon.com/free/

2.2.1 LECTURE 1

1. Show them the VirtualBox installation used for teaching

a. Explain what VirtualBox does
b. Show the machine booting
c. Login to console
d. Explain why working directly in console is not very productive

2. Go briefly through the installation instructions with the students, insisting on them reading everything carefully
3. Show the AWS option a little bit

2.2.2 LAB 1

1. Go briefly through the installation instructions with the students
2. The students should setup their Linux, by themselves, at home, and if necessary, as for help in the second lab

2.3 CONNECTING TO A LINUX COMPUTER

Even if you have a Linux computer already, you will need to connect to a school server to do your tests and exams. Below are the details of how to connect. The first time you connect, you will get a warning. Click or type "Yes", depending on the situation.

1. From Windows, use Putty to connect using the SSH protocol
2. From Linux or MacOS, use the command line to run commands like those below
   a. `ssh -p 8937 username@www.scs.ubbcluj.ro  # from home`
   b. `ssh username@linux.scs.ubbcluj.ro  # from campus`
   c. `ssh username@172.30.0.9  # from campus`

2.3.1 LECTURE 1

1. Show a Putty connection to the local VM

2.3.2 LAB 1

1. Have everybody connect to linux.scs.ubbcluj.ro using their passwords, and use the connections to work through what follows
2. Open two consoles at the same time, one for manuals and editors and another for executing commands

3 GETTING STARTED

3.1 COMMAND LINE

1. Why?
   a. It is the most powerful and flexible way to get things done in an operating system, provided that you know what you are doing
   b. Hundreds of programs already written that can do things for you already and can work and interact with each other (practically a Lego set of programs). Using them properly yields solutions fast.
   c. No real production program is run from Eclipse, IntelliJ, Code Blocks, Borland, ..., but from the command line
   d. Many times, the command line is all you have available to interact with an OS (especially in the cloud)
   e. Very cool, although ugly: cryptic one-liner for what would require lots of lines of C code. What are the top ten most frequent names of our students?

```
grep -E /home/scs /etc/passwd | awk -F: '{print $5}' | grep -E -v "^ex_|\."
| sed -E "s/ /\n/g" | grep -E -i "[a-z]{3,}" | tr "A-Z" "a-z" | sort | uniq
-c | sort -n -r | head -10
      83 andrei
      71 alexandru
      53 mihai
      47 maria
      41 daniel
      34 bogdan
```

```
32 vlad
31 andreea
31 alexandra
31 adrian
```

2. The command line is a program that allows us to run other programs
3. Students probably ran programs before by clicking on icons, or clicking "Run" in a development environment like Borland. The command line is just another way of running programs, and it is the primary way of running commands in production environments.
4. Can be found in any OS:
    a. Linux: Sh, Bash, Ksh, ...
    b. Windows: Cmd, Powershell, ...
    c. MacOS: Terminal (actually Bash)
5. We will use Linux, and the command line is case sensitive: `mkdir` is not the same thing as `MkDir`, nor is `hello.c` the same as `Hello.C`

3.1.1 LECTURE 1

1. Go through the "why" ideas above with examples
2. Show the complex command, explaining it superficially
3. Show the SH commands available on the system `/bin/ls /bin | /usr/bin/grep -E sh | grep -E '^[a-z]{0,2}sh$' | sort | uniq` and explain a little about them
4. Show case sensitivity effects

3.2 TRICKS, SHORTCUTS, AND PITFALLS OF THE CONSOLE

1. The mouse is really neat for copy/paste. Everything you select in the console is already copied to the clipboard. Right-click (Windows) or middle-click (Linux) will paste.
2. There are several keyboard shortcuts, which although identical to those used in Windows, do totally different things
    a. `Tab` - Autocomplete
    b. `Up/Down arrow` - navigate through the history of commands
    c. `Ctrl-C` - Stop the currently running program; really useful when you have an infinite loop
    d. `Ctrl-S` - Lock the console. You will be tempted to save your work with this, and then get confused.
    e. `Ctrl-Q` - Unlock console. This is the antidote for `Ctrl-S`. Note that everything you typed while the console was locked will show up after you press this.
    f. `Ctrl-Z` - Suspend the execution of the current program. Do not use this to "undo" anything ...
    g. `Ctrl-A` - Jump to the beginning of the line
    h. `Ctrl-E` - Jump to the end of the line
    i. `Ctrl-F` - Move forward on character
    j. `Ctrl-B` - Move backward on character
    k. `Ctrl-D` - End of file when providing program input. When pressed on the first position of the command line, ends the connection (similar to running `exit`).
    l. `Ctrl-R` - Search through the history of commands
    m. `Ctrl-K` - Cut the text from the current position to the end of the line
    n. `Ctrl-Y` - Paste what was cut with `Ctrl-K`

3.2.1 LAB 1

1. Run command `sort` and stop it with `Ctrl-C`
2. You can try the same with `while true; do date; sleep 1; done`, but you will need to keep `Ctrl-C` pressed a lot in order to have it happen for the while and not for the date or sleep commands (which will not stop the loop if killed)
3. Show how tab completion and history navigation works in the command line
4. Lock the console, type frantically, unlock the console
5. Give students 5 minutes to play with the rest of the keyboard shortcuts

3.3 COMMANDS AND PATHS

1. A command is a program, any program
2. We will only use commands that work in the console, meaning the interface is exclusively text
3. To run a command, type its name followed by whatever arguments necessary, everything separated by space. For instance:
    a. To list the content of the current directory run `ls`
    b. To see the content of file `/etc/passwd`, use `cat /etc/passwd`. Here, `/etc/passwd` is an argument
    c. To see the content of the current directory, run `ls -l`. Here, `-l` is an argument too
    d. To see the content of the current directory, with all the details, and including the hidden files, run `ls -l -a` or `ls -l --all`. Here, `-a` and `--all` have the same effect, one being the short form, and the other being the long form.
    e. To create a directory name `abc`, run `mkdir abc`
    f. To display the content of the current directory, with all the details but without the annoying colors, run `ls -l --color=never`. Here, `--color=never` is an argument with value. Sometimes, the equal sign is not necessary, but always consult the manual (command `man`) or the `--help` option (eg `ls --help`)
    g. To do the same thing above, for the directory `/etc`, run `ls -l --color=never /etc`
4. Paths
    a. UNIX file system has a single root, unlike Windows which has a root for every drive mounted (ie C:, D:, etc)
    b. The UNIX file system root is `/`, and all drives are mounted as directories, somewhere in the file system
    c. The UNIX file separator is /, unlike Windows, where the separator is \
    d. Every user has a home directory, which is the current directory when you connect over SSH. Run command `pwd` to find the path to your current directory.

3.3.1 LECTURE 1

1. Go through the ideas above, showing them in the command line

3.3.2 SEMINAR 1

1. Go through the ideas above explaining the command structure
    a. Space is separator
    b. First word is the command
    c. Next words are arguments
        i. Values: ls /etc
        ii. Options
            1. Short form: `ls -l`
            2. Long form: `ls --all`
            3. Short form with value: `cut -d :  -f 1,2,3 /etc/passwd`
            4. Long form with value: `cut --delimiter=: --fields=1,2,3 /etc/passwd`
            5. Combined short forms: `ps -e -f` is equivalent with `ps -ef`

3.3.3 LAB 1

1. The students execute the commands above and explain what is going on

3.4 LINUX MANUALS

1. If you need to learn about a command or C function or many other things, you can use the built-in manual pages
2. To read about command `ls`, run command `man ls`, to read about C function `pthread_create` run `man pthread_create`
3. The manual will open in the pager, you need to know how to navigate and exit the pager
    a. Page Down: `SPACE` (you can also use the `PgDn` key, but in some rare and weird cases suspends the command)
    b. Page Up: `b` (you can also use the `PgUp` key, but with the same risk as above)

    c. Search: `/`
    d. Exit: `q`
4. Structure
    a. Synopsis
    b. For C functions, list of headers to be included
    c. Arguments
    d. Return value
    e. See also
5. Finding the manual page you need
    a. `apropos ls`
    b. `whatis ls`
6. Manual sections
    a. `man open`
    b. `man 2 open`
7. Bash built-in commands manual pages
    a. Some built-in Bash commands do not have their own manual pages, but rather they appear in the bash manual page
    b. This is not applicable if the proper manual pages are installed
    c. However, if they are not installed, run `man bash` and scroll a lot to find details about `cd`, `read`, `shift`, `fg`, `bg`, `jobs`, ...

3.4.1 LECTURE 1

1. Show the manual for command `ls`, demonstrate navigation and search
2. Demonstrate finding the manual page you need, and explain the results displayed by `apropos` and `whatis`
3. Demonstrate manual sections

3.4.2 LAB 1

1. Have the students exercise what was demonstrated during the lecture, or if they did not have a lecture yet, guide them through doing those things
2. Give the students 5-10 minutes to learn about a few of the commands required in the graduation exam. See 3.c at the link below

https://www.cs.ubbcluj.ro/wp-content/uploads/tematica-licenta-informatica-engleza-2021.pdf

3.5 CONNECTING COMMANDS WITH PIPE

1. A pipe | takes the output of the command before it and passes it as input to the command after it
2. If you want to display the first 5 lines, in alphabetical order, of a file you need to first sort it, and then take the first 5 lines. That means redirecting the output of the command `sort`, to the input of the command `head`:
    ```
    sort a.txt |  head -n 5
    ```
3. If you now look again at the very long command in section 3, you will notice it makes heavy use of connecting commands through pipe. This is a very popular practice in command line usage.

3.5.1 LECTURE 1

1. Go over the ideas above
2. Explain the subcommands in the long command in section 3
3. Give a few more examples using commands like `sort`, `uniq`, `grep`, `find`, `ps`, `who`, `last`, `wc`, ...

3.6 USING A COMMAND LINE EDITOR

1. Editor choices: vim, emacs, nano, joe, micro
2. Students should choose one and learn how to do the following things

a. Start the editor
b. Exit the editor
c. Exit the editor without saving
d. Open a file
e. Save
f. Save to a file
g. Go up/down/left/right/home/end
h. Go page up/page down
i. Go to a certain line
j. Copy/Cut/Paste a few characters
k. Copy/Cut/Paste a whole line
l. Copy/Cut/Paste a few lines
m. Search
n. Search/replace
o. Undo

3.6.1 LECTURE 1

1. Emphasize the need for students to choose and learn a command line editor
2. Suggest VIM as the safest (but weirdest) choice

3.6.2 LAB 1

1. Configure your editor. You can do this for Vi/Vim. Emacs, Nano, Joe and Micro, but the example below is for Vim. If you want to configure other editors, search their documentation for the config file name and location, and the values you need to write to achieve the same things as below.
    a. Create/edit file `~/.vimrc` and write in it the lines below, to enable syntax coloring, tab size of 4 and tab insertion as spaces

```
syntax on
set tabstop=4
set expandtab
```

    b. Explain `~` as alias for the home directory
    c. Explain that files having their name starting with `.` are hidden (`ls -a` was mentioned above)

3.7 BASICS OF C PROGRAMMING IN THE LINUX COMMAND LINE

1. Development style
    a. Text editor
    b. Command line compiling
    c. Command line program execution
    d. Debugging
        i. Print to console
        ii. gdb (not needed so much)
    e. Detecting memory problems: valgrind
2. C language
    a. String vs byte array: strings end with 0, while buffers must be accompanied by their length somehow
    b. There are no references in C, only pointers (memory addresses)
        i. `&n` is the address of variable `n`
        ii. `*p` is the content of pointer `p`.
    c. Command line arguments: `int main(int argc, char** argv)`
        i. `argv[0]` - name of the command
        ii. `argv[1]` is the first argument, argv[2] is the second argument, and so on
        iii. `argc` - length of array `argv`
    d. Memory
        i. Allocation - `malloc`

ii. Deallocation - `free`

3. Text files
    a. All files are binary, but some of them contain only bytes between 0 and 127, which can be displayed as text, using the `ASCII` encoding.
        i. Discuss briefly other encodings (eg extended `ASCII` - visible in window borders of the Linux installer)

3.7.1 LECTURE 1

1. Implement a program that reads names from the command line and prints out a greeting

```
#include <stdio.h>
int main(int argc, char** argv) {
  char name[64];
  while(scanf("%s", name) == 1) {
    printf("Hello %s!\n", name);
  }
  return 0;
}
```

2. Improve the program, by remembering names already greeted and print a different greeting for them

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

struct node {
  char* name;
  struct node* next;
};

struct node* add(struct node* head, char* name) {
  struct node* n;
  struct node* p;

  n = (struct node*)malloc(sizeof(struct node));
  n->name = (char*)malloc(strlen(name)+1);
  strcpy(n->name, name);
  n->next = NULL;

  if(head == NULL) {
    return n;
  }

  p = head;
  while(p->next != NULL) {
    p = p->next;
  }
  p->next = n;

  return head;
}

void clear(struct node* head) {
  if(head == NULL) {
    return;
  }
  clear(head->next);
  free(head->name);
  free(head);
}
```

```
int known(struct node* head, char* name) {
  struct node* p;

  if(head == NULL) {
    return 0;
  }
  p = head;
  while(p != NULL && strcmp(p->name, name) != 0) {
    p = p->next;
  }
  if(p == NULL) {
    return 0;
  }
  return 1;
}

int main(int argc, char** argv) {
  char name[64];
  struct node* head = NULL;

  while(scanf("%s", name) == 1) {
    if(known(head, name)) {
      printf("Hello again, %s!\n", name);
    }
    else {
      head = add(head, name);
      printf("Hello %s!\n", name);
    }
  }

  clear(head);
  return 0;
}
```

3. Execute the program with `valgrind` to ensure there are no memory problems.
4. Create some memory problems by not allocating, or by not deallocating and run with `valgrind`. Explain the output and how to deal with it.
5. Improve the program to switch between the two functioning modes based on a command line argument

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

struct node {
  char* name;
  struct node* next;
};

struct node* add(struct node* head, char* name) {
  struct node* n;
  struct node* p;

  n = (struct node*)malloc(sizeof(struct node));
  n->name = (char*)malloc(strlen(name)+1);
  strcpy(n->name, name);
  n->next = NULL;

  if(head == NULL) {
    return n;
  }

  p = head;
  while(p->next != NULL) {
    p = p->next;
  }
  p->next = n;

  return head;
}

void clear(struct node* head) {
  if(head == NULL) {
    return;
  }
  clear(head->next);
  free(head->name);
  free(head);
}
```

```c
int known(struct node* head, char* name) {
  struct node* p;

  if(head == NULL) {
    return 0;
  }
  p = head;
  while(p != NULL && strcmp(p->name, name) != 0) {
    p = p->next;
  }
  if(p == NULL) {
    return 0;
  }
  return 1;
}


int main(int argc, char** argv) {
  char name[64];
  int recognize = 0;
  struct node* head = NULL;

  if(argc > 1 && strcmp(argv[1],"recognize") == 0) {
    recognize = 1;
  }
  while(scanf("%s", name) == 1) {
    if(recognize && known(head, name)) {
      printf("Hello again, %s!\n", name);
    }
    else {
      head = add(head, name);
      printf("Hello %s!\n", name);
    }
  }

  clear(head);
  return 0;
}
```

6. Remind everybody to learn `vi` or another command line editor of their choice

3.7.2 LAB 1,2

7. Hello World in C
   a. Write a "hello world" program in C. Guide the students in using a command line editor.

```c
#include <stdio.h>
int main(int argc, char** argv) {
  printf("Hello world!\n");
  return 0;
}
```

   b. Discuss the main function declaration and establish the form below as the standard main function declaration:
      i. Returns `int`
      ii. Takes arguments with which we can access the command line arguments
      iii. Returns `0` if all goes well (the truth value of the return/exit value will be discussed later on)

```c
#include <stdio.h>
int main(int argc, char** argv) {
  return 0;
}
```

   c. Compile it with `gcc -Wall -g -o hello hello.c` and explain why we want each argument and insist that they be used all the time
   d. Run the program as `/home/scs/an1/gr211/abir1234/hello` (the students will need to first run `pwd` to get their absolute path)
   e. Run it again as `./hello` and explain `.` and `..`
8. Dealing with compiler errors
   a. Have the students introduce errors in the code (eg delete `#` before `include`, the `)` after `argv`, and the `"` after `\n`), then recompile, and fix the code based on the compilation errors
   b. Change the `printf` line to `printf("Hello world! %f\n", sqrt(2))`, then recompile and deal with the missing header, then recompile and run
9. Implement a program that involves text and binary file I/O and memory allocation

a. Consider a text file storing a matrix as follows: the numbers of rows and columns are on the first line separated by space, and on the subsequent lines, are the elements of the matrix separated by space. Here is file `m.txt`

```
7 3
1 2 3
4 5 6
7 8 9
10 11 12
13 14 15
16 17 18
19 20 21
```

b. Read a matrix of integers from a text file and display to it in the console.

i. Explain double pointers

ii. Explain two-dimensional matrix allocation and deallocation

iii. Compile `gcc -Wall -g -o matrix-from-text matrix-from-text.c`

iv. Run `./matrix-from-text m.txt`

v. Run with `valgrin` as `valgrind ./matrix-from-text m.txt`

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

int main(int argc, char** argv) {
  int** m;
  int rows, cols, i, j;
  FILE* f;

  f = fopen(argv[1], "r");
  fscanf(f, "%d %d", &rows, &cols);

  m = (int**)malloc(rows*sizeof(int*));
  for(i=0; i<rows; i++) {
    m[i] = (int*)malloc(cols*sizeof(int));
    for(j=0; j<cols; j++) {
      fscanf(f, "%d", &m[i][j]);
    }
  }
  fclose(f);
```

```
  for(i=0; i<rows; i++) {
    for(j=0; j<cols; j++) {
      printf("%3d ", m[i][j]);
    }
    printf("\n");
  }

  for(i=0; i<rows; i++) {
    free(m[i]);
  }
  free(m);

  return 0;
}
```

c. Store the matrix in a binary file.

i. Compile `gcc -Wall -g -o matrix-to-binary matrix-to-binary.c`

ii. Run `./matrix-to-binary m.txt m.bin`

iii. Run with `valgrind` as `valgrind ./matrix-to-binary m.txt m.bin`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char** argv) {
  int** m;
  int rows, cols, i, j, fd;
  FILE* f;

  f = fopen(argv[1], "r");
  fscanf(f, "%d %d", &rows, &cols);

  m = (int**)malloc(rows*sizeof(int*));
  for(i=0; i<rows; i++) {
    m[i] = (int*)malloc(cols*sizeof(int));
    for(j=0; j<cols; j++) {
      fscanf(f, "%d", &m[i][j]);
    }
  }
  fclose(f);
```

```
  fd = open(argv[2], O_CREAT | O_WRONLY, 00600);
  write(fd, &rows, sizeof(int));
  write(fd, &cols, sizeof(int));
  for(i=0; i<rows; i++) {
    for(j=0; j<cols; j++) {
      write(fd, &m[i][j], sizeof(int));
    }
  }
  close(fd);

  for(i=0; i<rows; i++) {
    free(m[i]);
  }
  free(m);

  return 0;
}
```

d. Verify the binary file generated by the program by displaying it in the console using command `xxd m.bin`

```
00000000: 0700 0000 0300 0000 0100 0000 0200 0000  ................
00000010: 0300 0000 0400 0000 0500 0000 0600 0000  ................
00000020: 0700 0000 0800 0000 0900 0000 0a00 0000  ................
00000030: 0b00 0000 0c00 0000 0d00 0000 0e00 0000  ................
00000040: 0f00 0000 1000 0000 1100 0000 1200 0000  ................
00000050: 1300 0000 1400 0000 1500 0000            ............
```

e. Load the matrix form the binary file and display it in the console.

i. Compile `gcc -Wall -g -o matrix-from-binary matrix-from-binary.c`

ii. Run `./matrix-from-binary m.bin`

iii. Run with `valgrin` as `valgrind ./matrix-from-binary m.bin`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char** argv) {
  int** m;
  int rows, cols, i, j, fd;

  fd = open(argv[1], O_RDONLY);
  read(fd, &rows, sizeof(int));
  read(fd, &cols, sizeof(int));

  m = (int**)malloc(rows*sizeof(int*));
  for(i=0; i<rows; i++) {
    m[i] = (int*)malloc(cols*sizeof(int));
    for(j=0; j<cols; j++) {
      read(fd, &m[i][j], sizeof(int));
    }
  }
  close(fd);
```

```
  for(i=0; i<rows; i++) {
    for(j=0; j<cols; j++) {
      printf("%3d ", m[i][j]);
    }
    printf("\n");
  }

  for(i=0; i<rows; i++) {
    free(m[i]);
  }
  free(m);

  return 0;
}
```

4 COMMAND LINE UTILITIES

4.1 REGULAR EXPRESSIONS (POSIX ERE = EXTENDED REGULAR EXPRESSIONS)

- Very weird and ugly, but compact and flexible language for matching text.
- Why use them?
  - How can you find all the lines of a text file that contains phone numbers?
  - How can you remove all the extra spaces at the end of each line of a text file?
  - How can you remove any duplicated spaces from a text file?
  - If you ask a user to input an email in a text field, how do you verify that they input something that at least looks like an email?
- Regular expression rules
  - Every character that appears in a regular expression can have two meanings, normal or special, depending on the escape character `\` appearing in front of it.
  - Depending on the program used to process the regular expressions, a character's special meaning is achieved with or without escaping it. Below are the meanings as required by the programs we will use for this class.

`.`	Matches any single character
`\`	Escape, changes the meaning of the character following it, between normal and special
`[abc]`	Matches any single character that appears in the list (in this case `a` or `b` or `c`)
`[a-z]`	Matches any single character that belongs to the range (in this case any lower-case letter)
`[^0-9]`	Matches any single character that does not belong to the range (in this case anything that is not a digit)
`^`	Beginning of line
`$`	End of line
`\<`	Beginning of word
`\>`	End of word
`( )`	Group several characters into an expression
`*`	Previous expression zero or more times
`+`	Previous expression one or more times
`?`	Previous expression zero or one times
`{m,n}`	Previous expression at least `m` and at most `n` times
`|`	Logical OR between parts of the regular expression

4.1.1 SEMINAR 1

1. Go through the aspects above, giving the following examples for the rules

a. `.*` - any sequence of characters
b. `[a-zA-Z02468]` - any letter, regardless of its case, and any even digit
c. `[ ,]` - space or comma
d. `^[^0-9]+$` - non-empty lines containing any characters except digits
e. `([Nn][Oo] )+` - any refusal, no matter how insistent (eg No no no no no)

4.1.2 LAB 3, 4

By the time this lab is taught, the students would have had a seminar on regular expressions, grep, sed and awk, although the awk material may not have been fully covered in the seminar. Remind the regular expression rules to the students by either writing them on the board or asking them to use their seminar notes. Then move to the problems in the next sections.

4.2 GREP

1. Program that searches through files using regular expressions
2. Why is it called `grep` and not something more intuitive? It was very intuitive to those who implemented `grep`, but not so much to us. Google "why is grep called grep" and enjoy. While at it, go learn about `ed` (the original UNIX editor). It is available in your Linux installation. Have Google handy if you play with it ...
3. Option arguments that we will use a lot
    a. `-E` - use extended regular expressions (POSIX ERE)
    b. `-v` - display lines that do not match the given regular expression
    c. `-i` - ignore upper/lower case when matching (match case-insensitively)
    d. `-q` - Do not display the matching lines, just exit with 0 if found, or 1 if not found. We will use this only later , when we get to Shell Programming.
4. File /etc/passwd contains all the users in the system, providing their usernames, full names, home directories, etc
    a. We will use this file a lot to exercise text matching
    b. It is structured as follows, using `:` as field separator
        `username:password:user-id:group-id:user-info:home-directory:shell`
    c. The password field will always be x, the actual encrypted password residing in file /etc/shadow which cannot be read by regular users
    d. The user-info field, usually contains the full name of the user

4.2.1 SEMINAR 1

1. Let's search for things in file /etc/passwd
    a. Display all lines containing "dan". The solution is below
        i. `grep -E "dan" /etc/passwd`
    b. Display the line of username "dan". The username is the first field on the line, it is not empty, and it ends at the first `:`. We will rely on these aspects to ensure that we only search the usernames, and not anything else.
        i. `grep -E -i "^dan:" /etc/passwd`
    c. Display the lines of all users who do not have digits in their username.
        i. `grep -E "^[^0-9]+:" /etc/passwd`
    d. Display the lines of all users who have at least two vowels in their username. This is a little tricky, because the vowels do not need to be consecutive, so we need to allow for any characters between the vowels (including none), but we cannot allow : to appear between vowels, or else we would be searching outside the username.
        i. `grep -E -i "^[^:]*[aeiou][^:]*[aeiou][^:]*:" /etc/passwd`
        ii. `grep -E -i "^[^:]*([aeiou][^:]*){2,}:" /etc/passwd`
    e. There will be lots of users displayed for the problem above, so let's search for usernames with at least 5 vowels in their username. The first solution above will be really long for this case, but the second will be very easy to adapt, by changing 2 into 5.
        i. `grep -E -i "^[^:]*([aeiou][^:]*){5,}:" /etc/passwd`
    f. Display the lines of all the users not having bash as their shell. The shell is the last value on the line, so we will use that when searching.
        i. `grep -E -v "/bash$" /etc/passwd`

g. Display the lines of all users named Ion. We will have to search in the user-info field (the fifth field) of each line, ignore the upper/lower case of the letters, and ensure that we do not display anybody containing the sequence "ion" in their names (eg Simion, Simionescu, or Ionescu).
    i. `grep -E -i "^([^:]*:){4}[^:]*\<ion\>" /etc/passwd`

2. Let's consider a random text file a.txt, and search for things in it
    a. Display all the non-empty lines
        i. `grep -E "." a.txt`
    b. Display all the empty lines
        i. `grep -E "^$" a.txt`
    c. Display all lines containing an odd number of characters
        i. `grep -E "^(..)*.$" a.txt`
    d. Display all lines containing an ocean name
        i. `grep -E -i "\<atlantic\>|\<pacific\>|\<indian\>|\<arctic\>|\<antarctic\>" a.txt`
    e. Display all lines containing an email address
        i. What does an email address look like? It has the following structure.
            1. username - let's assume it can contain any character, except for @, *, !, and ?
            2. @ - separator between the username and the hostname
            3. hostname
                a. Sequence of at least two elements separated by .
                b. Let's assume an element can contain any letter, digit, dash, or underscore
        ii. `grep -E -i "\<[^@*\!?]+@[a-z0-9_-]+(\.[a-z0-9_-]+)+\>" a.txt`

4.2.2 LAB 3, 4

Ask the students to login to linux.scs.ubbcluj.ro so that they have more complex input data available.

1. Display the lines in /etc/passwd that belong to users having three parent initials in their name, even if the initials do not have a dot after them. You will notice that the regular expression accepts things that are not really parent initials, but there is not much else that we can do ...
    a. `grep -E " [A-Z]\.?[A-Z]\.?[A-Z]\.? " /etc/passwd`
    b. `grep -E " ([A-Z]\.?){3,} " /etc/passwd`
2. Display the lines in /etc/passwd that belong to users having names of 12 characters or longer (this year there is one with a 13 character name)
    a. `grep -E -i "^([^:]*:){4}[^:]*[a-z]{12,}" /etc/passwd`

4.3 SED

1. Program for searching processing text by performing search/replace, transliterations, line deletion, etc
2. By default, it does not modify the file, but displays the result of processing the input file
3. We will use only the features presented below
    a. Search/replace
        i. Command structure: `sed -E "s/regex/replacement/flags" a.txt`
        ii. `s` - is the search/replace command
        iii. `/` is the separator, and can be any other character. The first character after the command `s` is considered to be the separator
        iv. The flags at the end can be `g`, `i`, or both
            1. `g` - Perform the replacement everywhere on the line. Without it, only the first appearance will be replaced
            2. `i` - Perform a case-insensitive search
        v. The replacement can contain reference to the expressions grouped in the regex as `\1`, `\2`, etc, the number being the order in which the groups appear in the regex
    b. Transliterate
        i. Command structure: `sed -E "y/characters/replacement/" a.txt`
        ii. `y` - is the transliteration command

iii. `/` is the separator, and can be any other character. The first character after the command `y` is considered to be the separator
iv. The `characters` and the `replacement` must have the same length

c. Delete lines matching a regular expression
i. Command structure: `sed -E "/regex/d" a.txt`
ii. `/` is the separator
iii. `d` - is the line deletion command

4.3.1 SEMINAR 1

1. Let's manipulate the content of /etc/passwd
   a. Display all lines, replacing all vowels with spaces
      i. `sed -E "s/[aeiou]/ /gi" /etc/passwd`
   b. Display all lines, converting all vowels to upper case
      i. `sed -E "y/aeiou/AEIOU/" /etc/passwd`
   c. Display all lines, deleting those containing numbers of five or more digits:
      i. `sed -E "/[0-9]{5,}/d" /etc/passwd`
   d. Display all lines, swapping all pairs of letters
      i. `sed -E "s/([a-z])([a-z])/\2\1/gi" /etc/passwd`
   e. Display all lines, duplicating all vowels
      i. `sed -E "s/([aeiou])/\1\1/gi" /etc/passwd`

4.3.2 LAB 3, 4

Ask the students to login to linux.scs.ubbcluj.ro so that they have more complex input data available.

1. Convert the content of /etc/passwd using a sort of Leet/Calculator spelling (eg Bogdan -> B09d4n)
   a. `sed -E "y/elaoszbg/31405289/" /etc/passwd`
2. Convert the content of /etc/passwd surrounding with parentheses and sequence of 3 or more vowels
   a. `sed -E "s/([aeiou]{3,})/(\1)/gi" /etc/passwd`

4.4 AWK

1. Given a separator character (by default it is space), treats the input text as a table, with each line being a row, and the fields of each row the tokens of the line, as determined by the separator.
2. Processes the input based on a program written in a simple C-like language
3. A program is a sequence of instruction blocks, prefixed by an optional selector
4. Each block in the program is applied to every line of input matching its selector. If the block does not have a selector, it is applied to every line of input
5. A selector is any valid conditional expression, or one of the following two special selectors
   a. BEGIN - the block associated with this selector is executed before any input has been processed
   b. END - the block associated with this selector is executed after all input has been processed
6. Special variables
   a. NR - number of the current line of input
   b. NF - the number of fields on the current line
   c. $0 - the entire input line
   d. $1, $2, ... - the fields of the current line
7. The AWK program can be written in a file, or provided directly on the command line between apostrophes

4.4.1 SEMINAR 1

1. Manipulate the content of /etc/passwd, using AWK with the program provided on the command line
   a. Display all the usernames, but only the usernames, and nothing else. We will use argument `-F` to tell AWK that the input file is separated by `:`, and then we will print the first field of each line, by not providing any selector for the block.
      i. `awk -F: '{print $1}' /etc/passwd`

b. Print the full name (the user info field) of the users on odd lines
   i. `awk -F: 'NR % 2 == 1 {print $5}' /etc/passwd`
c. Print the home directory of users having their usernames start with a vowel
   i. `awk -F: '/^[aeiouAEIOU]/ {print $6}' /etc/passwd`
d. Print the full name of users having even user ids
   i. `awk -F: '$3 % 2 == 0 {print $5}' /etc/passwd`
e. Display the username of all users having their last field end with "nologin"
   i. `awk -F: '$NF ~ /nologin$/ {print $1}' /etc/passwd`
f. Display the full names of all users having their username longer than 10 characters
   i. `awk -F: 'length($1) > 10 {print $5}' /etc/passwd`

2. Keep using /etc/passwd as input file, but provide AWK programs in a file. The command will look like
   a. `awk -F: -f prog.awk /etc/passwd`
   b. Provide the content of file `prog.awk` so that the command above will print all user on even line having a group id less than 20

```
NR % 2 == 0 && $4 < 20 {
  print $5
}
```

   c. Display the sum of all user ids

```
BEGIN {
  sum=0
}

{
  sum += $3
}

END {
  print sum
}
```

   d. Display the product of the differences between the user id and the group id

```
BEGIN {
  prod=1
}

{
  prod *= $3-$4
}

END {
  print prod
}
```

4.4.2 LAB 3, 4

If the students have not yet done AWK in the seminar, give them the AWK basics or postpone section for a week. Ask the students to login to linux.scs.ubbcluj.ro so that they have more complex input data available.

1. Display the full names (but only the full names) of the students belonging to group 211
   a. `awk -F: '$6 ~ /\/gr211\// {print $5}' /etc/passwd`
2. Count the numbers of male and female users in /etc/passwd, accepting as true the following incorrect assumptions:
   a. All users have their last name as the first name in the user-info filed (5th field)
   b. All women have one of their first or middle names ending in the letter "a"
   c. `awk -F: -f prog.awk /etc/passwd`

```
BEGIN {
    m=0
    w=0
}

# The space at the beginning of the regular
# expressions is for not matching the last name
$5 ~ / [a-zA-Z]*[b-z]\>/ {
    m++
}

$5 ~ / [a-zA-Z]*a\>/ {
    w++
}

END {
    print "Men:", m
    print "Women:", w
}
```

4.5 OTHER USEFUL COMMANDS

4.5.1 LAB 3, 4

Ask the students to login to linux.scs.ubbcluj.ro so that they have more complex input data available. The last problem is rather large and relies on an AWK for loop which was not taught in the seminar.

1. Display only the last name of each user in /etc/passwd, considering the last name to be the first word in the 5th field, and accepting it only if it starts with a capital letter
   a. `awk -F: '$5 ~ /^[A-Z]/ {print $5}' /etc/passwd | cut -d' ' -f1`
      or, with `awk` instead of `cut`
   b. `awk -F: '$5 ~ /^[A-Z]/ {print $5}' /etc/passwd | awk '{print $1}'`
2. Extent the solution above to only show the top 10 most frequent last names, ordered descending by their popularity
   a. `... | sort | uniq -c | sort -n -r | head -n 10`
3. Display all the directories under /etc that contain files with the extension .sh. Each directory should be displayed only once. Hide the permission denied errors given by find.
   a. `find /etc -name "*.sh" 2>/dev/null | sed -E "s/\/[^\/]*$//" | sort|uniq`
      or simpler by using a different sed separator
   b. `find /etc -name "*.sh" 2>/dev/null | sed -E "s,/[^/]*$,," | sort|uniq`
4. Display in the pager, the number of processes of each username, sorting their usernames descending by their process count.
   a. `ps -ef| awk '{print $1}'|sort|uniq -c|sort -n -r | less`
5. Display the processes that involve editing a C file
   a. `ps -ef| grep -E "\.c\>"`
6. Display in the pager, the usernames with the most logins in the system.
   a. `last | cut -d' ' -f1|sort|uniq -c | sort -n -r|less`
7. Display in the pager the top of usernames by their time spent logged on in the system. The solution will be built gradually following the steps below
   a. Display all the usernames and their time spent in the system, ignoring the other fields displayed by command `last`.
      i. `last | awk '{print $1, $10}'`
   b. Making the time spent in the system field uniform across the output, by adding a 0+ to all entries missing a day element. That is, (03:35) should become (0+03:35).
      i. `... |sed -E "s/\(([0-9][0-9]:)/(0+\1/"`
   c. Calculate the time spent in the system in minutes for each entry
      i. `... | sed -E "s/[():+]/ /g"|awk '{print $1, ($2*24*60+$3*60+$4)}'`
   d. Calculate the total time spent in the system by each user
      i. `... | awk -f prog.awk`

```
{
    arr[$1] += $2
}

END {
    for(u in arr) {
        print u, arr[u]
    }
}
```

or, directly on the command line

ii. `... | awk '{arr[$1] += $2} END {for(u in arr) print u, arr[u]}'`

e. Sort the output descending by the time spent in the system and pipe it to the pager

i. `... | awk 'v{print $2, $1}' | sort -n -r | less`

or, simpler

ii. `... | sort -k2nr | less`

5 UNIX SHELL PROGRAMMING

5.1 STANDARD INPUT/OUTPUT/ERROR AND I/O REDIRECTIONS

1. `0` = standard input - where you read from when you use "scanf" or "gets" in C, "cin" in C++, or "input" in Python.
2. `1` = standard output - where you write when you use "printf" or "puts" in C, "cout" in C++, or "print" in Python.
3. `2` = standard error - similar to the standard output, but conventionally used to display errors, in order to avoid mixing results with errors.
4. What's the deal with `0`, `1`, and `2`?
    a. When you open a file in a program (written in any language), you get back some kind of variable that allows you to operate on the file (`FILE*` from `fopen`, `int` from `open`, etc). This variable contains an integer, representing the roughly the order number of the file opened by the program.
    b. Whenever you start a program, it will have three files already open: `0`, `1`, and `2`. Yeah, the program treats the command line like a file: it writes to it and it reads from it. Very natural, isn't it?
5. Many of the commands we will use act as filters: read the standard input, process it, and then print the results to the standard output. The errors will be printed to the standard error.
6. I/O redirections
    a. What if I want the output of a command to be stored in a file?
        i. `ls -l --color=never /etc > output.txt`
    b. What if I want to add the output of another command to the same file?
        i. `ps -ef >> output.txt`
    c. What if I want the standard output of a command to be sent to the standard input of another command?
        i. `ls| sort`
    d. What if I want the standard input to be taken from a file?
        i. `sort < a.txt`
    e. Redirect the errors of a command to a file
        i. `rm some-file-that-does-not-exist.c 2> output.err`
    f. Redirect both the standard and error output in the same file
        i. `rm some-file-that-does-not-exist.c > output.all 2>&1` - read as, redirect standard output to output.all, and the error output to the same place where the standard output goes
7. /dev/null
    a. The file that contains nothing, and everything that you write to it, disappears. The Windows equivalent is NUL
    b. Used mainly to hide program output (either standard or error)

5.1.1 LECTURE 2

1. Present the items 1-5 above, then do the example below
2. Write a C program that outputs * instead of every read character, except for new line. Implement two versions, one using C library functions and another using system calls. Here is file `filter.c`

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

void with_lib() {
  char s[64];
  char* p;
  int i;

  while(1) {
    p = fgets(s, 64, stdin);
    if(p == NULL) {
      break;
    }
    for(i=0; i<strlen(s); i++) {
      if(s[i] != '\n') {
        s[i] = '*';
      }
    }
    fputs(s, stdout);
  }
}
```

```c
void with_syscall() {
  char s[64];
  int i, k;

  while(1) {
    k = read(0, s, 64);
    if(k <= 0) {
      break;
    }
    for(i=0; i<k; i++) {
      if(s[i] != '\n') {
        s[i] = '*';
      }
    }
    write(1, s, k);
  }
}

int main(int argc, char** argv) {
  if(argc > 1 && strcmp("lib", argv[1]) == 0) {
    with_lib();
  }
  else if(argc > 1 && strcmp("sys", argv[1]) == 0) {
    with_syscall();
  }
  else {
    fputs("Unsupported mode", stderr);
  }

  return 0;
}
```

3. Present items 6-7 above explaining the redirections mechanisms as you go along
4. Use the program above to test the I/O redirections
    a. `cat /etc/passwd | ./filter sys`
    b. `./filter sys < /etc/passwd`

5.2 COMMAND TRUTH VALUES

1. The truth value of a command execution is determined by its exit code. The rule is the opposite of the C convention, with `0` being `true`, and anything else being `false`. Basically, there is only one way a command can be executed successfully, but many ways in which it can fail. The exit code is not the output of the command.
2. There are two standard commands `true` and `false`, that simply return `0` or `1`.
3. Command `test` offers a lot of options for comparing integers, strings and verifying file and directory attributes

5.2.1 LECTURE 2

1. Commands can be chained using logical operators `&&` and `||`. Lazy logical evaluation can be used to nice effects. The negation operator `!` reverses the truth value of a command.
    a. `true || echo This should not be displayed`
    b. `false || echo This should definitely be displayed`
    c. `true && echo This should also be displayed`
    d. `false && echo Should never be displayed as well`
    e. `grep -E -q "=" /etc/passwd || echo There are no equal signs in file /etc/passwd`
    f. `test -f /etc/abc || echo File /etc/abc does not exist`
    g. `test 1 -eq 2 || echo Not equal`
    h. `test "asdf" == "qwer" || echo Not equal`
    i. `! test -z "abc" || echo Empty string`
2. Test command conditional operators
    a. String: `==`, `!=`, `-n`, `-z`
    b. Integers: `-lt`, `-le`, `-eq`, `-ne`, `-ge`, `-gt`
    c. File system: `-f`, `-d`, `-r`, `-w`, `-x`

5.3 SHELL VARIABLES AND EMBEDDED COMMANDS

1. Present variable definition and reference.

2. Present embedded command usage

5.3.1 LECTURE 2

1. Defined as `A="Tom"` or `B=5`
2. Embedded commands
    a. Delimited by ` (back-quote)
    b. Are replaced by the output of the command
    c. Store a command output in a variable: `N=`grep -E "/gr211/" /etc/passwd | wc -l`
3. Referred as `$A` or `${A}`
    a. `echo $A is a human`
    b. `echo $Acat is a feline or an application server` - doesn't work
    c. `echo ${A}cat is a feline or an application server`
4. When used in strings delimited by ", variables and embedded commands will be replaced by their value. Strings delimited by ' do not allow any substitutions in their content.
    a. `echo "$A$A is a GPS navigator"`
    b. `echo "There are `grep "/gr211/" /etc/passwd | wc -l` students in group 211"`

5.4 SHELL SCRIPTS

1. What is a script?
2. File execution permissions, and permissions in general
3. Special variables
4. Examples

5.4.1 LECTURE 2

1. Any text file with execution permissions can be a script, if it contains commands interpretable by the current shell
2. Comments start with `#`
3. Hello World example
    a. Create file `a.sh` with the content below

```
echo Hello World
```

    b. Give the script execution permissions using `chmod 700 a.sh`
    c. Execute the script using `./a.sh`
4. Permissions
    a. Run `ls -l` and see the first 10 characters on each line
        i. The first character tells the file type: `-` is a regular file, d is a directory
        ii. Characters 2-4 show the permissions for the owner of the file (field 3 displayed by `ls -l`)
        iii. Characters 5-7 show the permissions for the group of the file (field 4 displayed by `ls -l`)
        iv. Characters 8-10 show the permissions for everybody else
    b. Each permission triplet describes the read, write and execution permissions
    c. Can be described as a number, by considering each of the 3 positions to be a binary digit.
        i. 7 = 111 = `rwx`
        ii. 6 = 110 = `rw-`
        iii. 5 = 101 = `r-w`
    d. Command `chmod` is used to assign permissions to files
        i. `chmod 700 a.sh` gives the owner of the file full permissions, and nothing to the group or the others
5. Hello World example with shell specification
    a. Create file `hello.sh` with the content below

```
#!/bin/bash

echo Hello World
```

    b. Give the script execution permissions using `chmod 700 hello.sh`
    c. Execute the script using `./hello.sh`
6. Special variables

a. `$0` - The name of the command
b. `$1` - `$9` - Command line arguments
c. `$*` or `$@` - All the arguments together
d. `$#` - Number of command line arguments
e. `$?` - Exit code of the previous command

7. Special variables example, special-vars.sh

```
#!/bin/bash

echo Command: $0
echo First four args: $1 $2 $3 $4
echo All args: $@
echo Arg count: $#

true
echo Command true exited with code $?

false
echo Command false exited with code $?
```

a. `chmod 700 special-vars.sh`
b. `./special-vars.sh a b c d e f g h i j k l m`

8. Accessing arguments using `shift`. Script `using-shift.sh`

```
#!/bin/bash

echo Command: $0
echo First four args: $1 $2 $3 $4
echo All args: $@
echo Arg count: $#

shift
echo Some args: $1 $2 $3 $4
echo All args: $@
echo Arg count: $#

shift 3
echo Some args: $1 $2 $3 $4
echo All args: $@
echo Arg count: $#
```

a. `chmod 700 using-shift.sh`
b. `./using-shift.sh a b c d e f g h i j k l m`

5.5 UNIX SHELL FOR LOOP

5.5.1 LECTURE 2

1. Similar to the Python `foreach`
2. The variable cycles through a list of space separated values
3. Basic example `using-for.sh`, showing `do` on the same line or on the next line. Semicolon is the command separator.

```
#!/bin/bash

for A in a b c d; do
    echo Here is $A
done

for A in a b c d
do
    echo Here is $A
done
```

a. `chmod 700 using-for.sh`
b. `./using-for.sh`

4. Iterating over the command line arguments, `for-args.sh`, showing the short and not very intuitive second possibility

```
#!/bin/bash

for A in $@; do
    echo Arg A: $A
done

for A; do
    echo Arg B: $A
done
```

a. `chmod 700 for-args.sh`
b. `./for-args.sh a b c d e f g h i j k l m`

5.5.2 SEMINAR 2

By the time of the seminar, the students would have already had the lecture on Shell Programming. So we will do more complex examples.

1. The list of values through which for iterates can be specified either explicitly as above, or through filename wildcards, or embedded commands
    a. Count all the lines of code in the C files in the directory given as command line argument, excluding lines that are empty or contain only spaces

```
#!/bin/bash

S=0
for F in $1/*.c; do
    N=`grep -E "[^ ]" $F | wc -l`
    S=`expr $S + $N`
done
echo $S
```

    b. Filenames that contain spaces will cause problems here
2. Filename wildcards
    a. Similar but much simpler than regular expressions
    b. Rules:
        i. `*` - Matches any sequence of characters, including a void sequence, but not the first dot in a filename
        ii. `?` - Matches any single character, but not the first dot in a filename
        iii. `[abc]` - List of optional characters, support ranges like the regular expressions
        iv. `[!abc]`- Negated list of optional characters (similar to `[^abc]` from regular expressions)
    c. Example: list all the file starting with a letter and having an extension of exactly two characters
        i. `ls [a-zA-Z]*.??`
3. Count all the lines of code in the C files in the directory given as command line argument and its subdirectories, excluding lines that are empty or contain only spaces

```
#!/bin/bash

S=0
for F in `find $1 -type f -name "*.c"`; do
    N=`grep -E "[^ ]" $F | wc -l`
    S=`expr $S + $N`
done
echo $S
```

    a. Filenames that contain spaces will cause problems here as well
    b. Solving this problem with `find ... | while read F`, will avoid the space in file name problems, but incrementing S will not work because while is executed in a sub-shell. Solutions to overcoming this are outside the scope of this course.

5.6 UNIX SHELL IF/ELIF/ELSE/FI STATEMENT

5.6.1 LECTURE 2

1. Every command is a condition
2. Commands can be grouped with parentheses and logical operators
    a. Check whether a file does not exist or if it exists whether it is not readable
    b. `! test -f a.txt || ( test -f a.txt && ! test -r a.txt )`

3. Present the basic IF syntax, using script `basic-if.sh` which checks each argument and announces whether it is a file, or a directory, or a number, otherwise it states that it does not know what it is. Just like `do`, `then` can be either on the same line or on the next line. Do not introduce the [...] syntax.

```
#!/bin/bash

for A in $@; do
    if test -f $A; then
        echo $A is a file
    elif test -d $A
    then
        echo $A is a dir
    elif echo $A | grep -E -q "^[0-9]+$"; then
        echo $A is a natural number
    else
        echo We do not know what $A is
    fi
done
```

   a. `chmod 700 basic-if.sh`
   b. `./basic-if.sh /etc /etc/passwd . 1234 a2b rr`

5.6.2 SEMINAR 2

By the time of the seminar, the students would have already had the lecture on Shell Programming. So we will introduce the [...] syntax of the conditions and do more complex examples.

1. To make the condition look a bit more natural, there is a second syntax, in which `[` is an alias of command `test` and `]` marks the end of the command `test`. Pay attention to leaving spaces around these square brackets or there will be syntax errors.
2. The basic IF example from the lecture, can be re-written as follows

```
#!/bin/bash

for A in $@; do
    if [ -f $A ]; then
        echo $A is a file
    elif [ -d $A ]
    then
        echo $A is a dir
    elif echo $A | grep -E -q "^[0-9]+$"; then
        echo $A is a number
    else
        echo We do not know what $A is
    fi
done
```

5.7 UNIX SHELL WHILE STATEMENT

5.7.1 LECTURE 2

1. Read user input until the input is stop
2. The user input is read with command read which stores the input in the variable given as argument
3. Script basic-while.sh

```
#!/bin/bash

while true; do
    read X
    if test "$X" == "stop"; then
        break
    fi
done
```

   a. `chmod 700 basic-while.sh`
   b. `./basic-while.sh`
4. Find all the files in the directory given as first command line argument, larger in size than the second command line argument. Script `large-files.sh`

```
#!/bin/bash

D=$1
S=$2

find $D -type f | while read F; do
    N=`ls -l $F | awk '{print $5}'`
    if test $N -gt $S; then
        echo $F
    fi
done
```

a. `chmod 700 large-files.sh`
b. `./large-files.sh`
c. This example also makes it clear why the AWK program must be provided between apostrophes, not quotes

5.7.2 SEMINAR 2

By the time of the seminar, the students would have already had the lecture on Shell Programming. So we will do more complex examples.

1. The while loop also accepts the [...] condition syntax
2. Read the console input until the user provides a filename the exists and can be read

```
#!/bin/bash

F=""
while [ -z "$F" ] || [ ! -f "$F" ] || [ ! -r "$F" ]; do
    read -p "Provide an existing and readable file path:" F
done
```

or

```
#!/bin/bash

F=""
while test -z "$F" || ! test -f "$F" || ! test -r "$F"; do
    read -p "Provide an existing and readable file path:" F
done
```

5.8 UNIX SHELL PROGRAMMING EXAMPLES

5.8.1 LECTURE 3

1. Find all the students in group 211
    a. `grep -E "/an1/gr211/" /etc/passwd`
2. Display the most frequent names of the users (first, middle, etc. but not last) in the system. This is similar to a problem solved in lab 3/4 but still a little different as there are more non-last names to a user. Present the thought process for solving the problem, and each command in the pipe chain.

```
awk -F: '{print $5}' /etc/passwd | \
cut -d ' ' -f2- | \
sed -E "s/[ -]/\n/g" | \
grep -E -v "\.|^.$" | \
tr '[A-Z]' '[a-z]' | \
sort | \
uniq -c | \
sort -n -r | \
less
```

a. The data we need is in the 5th field of `/etc/passwd`
b. Of the 5th field we need all the words except for the first. We can do this with `cut`, `sed`, or `awk`.
c. Now we put each word on a line, by replacing space with newline. As some names are linked by dash, we also replace the dash with new line
d. We eliminate initials (as much as we can) by eliminating all lines that contain either dot or a single character
e. To avoid upper/lower case issues with `sort` and `uniq`, we convert everything to lower case. We can do it with `sed`'s `y` command, but we would need to type the whole alphabet, so we use command `tr` which also does transliteration and supports a shorter input.
f. Finally we sort the names and `uniq`-count them and sort them descending, displaying them in a pager.

3. Stop student processes older than the number of seconds given as command line argument.

a.
```
#!/bin/bash

for X in `ps -ef |grep -E -v "^root " | tail -n +2 | awk '{print $1 ":" $2}'`; do
    U=`echo $X|cut -d: -f1`
    P=`echo $X|cut -d: -f2`

    echo $U $P
    if grep -E "^$U:" /etc/passwd | cut -d: -f6 | grep -E -q "/scs/"; then
        A=`ps -o etime $P | tail -n 1 | awk -F: '{print ($1*60+$2)}'`
        if [ $A -ge $1 ]; then
            echo "Should kill $U $P $A"
        fi
    fi
done
```

   i. Explain the [...] condition syntax for those who didn't have the seminar yet
   ii. To speed up the script, we skip all processes belonging to `root`.
   iii. As `ps` displays a header, we skip that as well

b.
```
#!/bin/bash

ps -ef | \
grep -E -v "^root " | \
tail -n +2 | \
awk '{print $1, $2}' | \
while read U P; do
    echo $U $P
    if grep -E "^$U:" /etc/passwd | cut -d: -f6 | grep -E -q "/scs/"; then
        A=`ps -o etime $P | tail -n 1 | awk -F: '{print ($1*60+$2)}'`
        if [ $A -ge $1 ]; then
            echo "Should kill $U $P $A"
        fi
    fi
done
```

   i. Same solution as above, but with while and with command splitting over multiple lines to make it more readable

4. Present the main sources of Shell script syntax errors
   a. Missing spaces around the condition square brackets
   b. Missing quotes in conditions around blank variables

5.8.2 SEMINAR 2

1. Write a script that monitors the state of a directory and prints a notification when something changed

```
#!/bin/bash

D=$1
if [ -z "$D" ]; then
  echo "ERROR: No directory provided for monitoring" >&2
  exit 1
fi

if [ ! -d "$D" ]; then
  echo "ERROR: Directory $D does not exist" >&2
  exit 1
fi

STATE=""
while true; do
    S=""
    for P in `find $D`; do
        if [ -f $P ]; then
          LS=`ls -l $P | sha1sum`
          CONTENT=`sha1sum $P`
        elif [ -d $P ]; then
          LS=`ls -l -d $P | sha1sum`
          CONTENT=`ls -l $P | sha1sum`
        fi
        S="$S\n$LS $CONTENT"
    done
    if [ -n "$STATE" ] && [ "$S" != "$STATE" ]; then
      echo "Directory state changed"
    fi
    STATE=$S
    sleep 1
done
```

a. We use sha1sum to get a checksum that is statistically impossible to be identical for different contents
b. We checksum the details of the file (`ls -l`) as well as its content
c. For directories, we use the `-d` flag of ls to list the directory details and not its content, and we use the output of `ls -l` for the directory content
d. We only handle regular files as directories when building the state. We should also address pipes, links, etc but it is outside the scope of this course.

5.8.3 LAB 4, 5

1. Implement and test the script presented in the seminar (see section above)
2. Re-write the same script using conditions without square brackets (ie `[ -f $P ]` becomes `test -f $P`)
3. If there is any time left, solve some of the practice the students find difficult. For instance, here is a solution for problem 10
    a. Display the session count and full names of all the users who logged into the system this month, sorting the output by the session count in descending order. Use the `-t` option of command `last` to get this month's sessions, and the command `date` to generate the required timestamp in the expected format.

```
#!/bin/bash

D=`date +%Y%m`
T="${D}01000000"
last -t $T | \
sed -E "s/ .*//" | \
sort | \
uniq -c | \
sort -n -r | \
while read L U; do \
    N=`grep -E "^$U:" /etc/passwd | cut -d: -f5`
    echo $L $N
done | \
less
```

    b. We could also build the timestamp in a more compact way
        i. `last -t `date +%Y%m01000000``

6 UNIX PROCESSES

6.1 PROCESSES AND CONCURRENT EXECUTION

1. A process is a program in execution. Running a program multiple times will result in multiple processes
2. Processes that run simultaneously are said to be concurrent
3. A quick check shows that a system usually has significantly more processes than CPUs and cores. SO how can all those processes be active at the same time since there are not enough processing units?
    a. The system gives each process a quanta of time on the CPU and switches between processes after each quanta
    b. These time quants are imperceptibly small, so all processes seem to make progress simultaneously.
4. Problem: processes may yield wrong results when working on the same resources as other processes, even though they would be otherwise correct. Such a situation is called race condition.

6.1.1 LECTURE 3

1. Go over the aspects above, emphasizing that the way the students have programed until now was ignoring the reality of concurrency, and that this reality is here to stay.
2. Show and explain the race condition created by running the `run.sh` script below

```
#!/bin/bash

F=$1
N=0
while [ $N -lt 200 ]; do
    K=`cat $F`
    K=`expr $K + 1`
    echo $K > $F
    N=`expr $N + 1`
done
```

```
#!/bin/bash

echo 0 > a.txt

# The script on the left is inc.sh

./inc.sh a.txt &
./inc.sh a.txt &
./inc.sh a.txt &
```

3. Show and explain the race condition created by running the `inc.c` program below simultaneously on the same file

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <string.h>

int main(int argc, char** argv) {
    int f, k, i;

    f = open(argv[1], O_RDWR);
    if(argc > 2 && strcmp(argv[2], "reset") == 0) {
        k = 0;
        write(f, &k, sizeof(int));
        close(f);
        return 0;
    }
    for(i=0; i<255*255; i++) {
        lseek(f, 0, SEEK_SET);
        read(f, &k, sizeof(int));
        k++;
        lseek(f, 0, SEEK_SET);
        write(f, &k, sizeof(int));
    }
    close(f);
    return 0;
}
/* Script for simultaneous execution
#!/bin/bash

./inc b.dat reset
./inc b.dat &
./inc b.dat &
./inc b.dat &
*/
```

4. Explain how very often processes get interrupted in the middle of what students usually consider a single (atomic) instruction, such as `n++`

6.1.2 SEMINAR 3

Spend a few minutes discussing the concepts above, so that the students understand how processes can get interrupted in the middle of an apparently atomic instruction, and how while stopped, other processes may do things that can corrupt the final result. They are not used to think of this and need to adopt it. The "my program works fine by itself" approach is no longer valid.

6.2 CREATING PROCESSES IN UNIX

6.2.1 LECTURE 4

1. The UNIX way of creating a process is to duplicate the current process, by making a copy of it using the system call `fork`
2. This leads to another unfamiliar aspect: source code that looks simple and appears to start at the beginning and finish at the end, will actually execute along split paths. Take for instance the code below. It will print "after" twice. That's because after calling `fork`, there will be two processes: the original one (called parent), and the copy of it (called child). Both processes continue from the instruction after `fork`, and thus will both print "after".

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

int main(int argc, char** argv) {
    printf("before\n");
    fork();
    printf("after\n");
    return 0;
}
```

3. Calling fork in a loop can be dangerous if done wrongly, because the number of processes will grow exponentially. The code below will create 7 child processes:

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

int main(int argc, char** argv) {
    int i;
    for(i=0; i<3; i++) {
        fork();
    }
    return 0;
}
```

6.3 DISTINGUISHING BETWEEN THE PARENT AND THE CHILD PROCESS

6.3.1 LECTURE 4

1. What is the point of creating copies of the same process? It would make sense if each copy could execute something else, but as of now, that does not seem to be the case.
2. Actually, there is a way to distinguish between the parent and child processes, by the return value of `fork`: it returns `0` in the child, and the PID of the child in the parent. Consider the code below, ignoring the wait system call which will be explained a little bit later. PID is the current processes ID and PPID is the parent's ID. The output will be the text on the right.

```
#include<stdio.h>
#include<unistd.h>
#include<sys/types.h>
#include<sys/wait.h>

int main(int argc, char** argv) {
    int r;
    r = fork();
    printf("PID=%d PPID=%d R=%d\n", getpid(), getppid(), r);
    wait(0);
    return 0;
}
```

```
PID=1612 PPID=1436 R=1613
PID=1613 PPID=1612 R=0
```

3. This allows us to execute different code in the child, than in the parent. The exit call in the child ensures that the child is not going to continue with the parent code after finishing the if-statement. The parent never enters the if-statement

```
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
#include<sys/types.h>
#include<sys/wait.h>

int main(int argc, char** argv) {
    int pid;
    pid = fork();
    if(pid == 0) {
        printf("Child-only code\n");
        exit(0);
    }
    printf("Parent-only code\n");
    wait(0);
    return 0;
}
```

4. This allows us to write concurrent programs, like a server that serves multiple requests simultaneously. A schematic sequential server code would look like the code below. If Google were implemented like this, any search you make would have to wait until all the other searches done by other people in the world before you, are completed. Definitely not a good user experience ...

```
while(1) {
    req = get_request();
    res = process_request(req);
    send_response(res);
}
```

5. We can now write this code as follows. Every request is served by a separate process, and the main server (the parent process) can get a new request immediately and spawn another child process to serve it concurrently.

```
while(1) {
    req = get_request();
    pid = fork();
    if(pid == 0) {
        res = process_request(req);
        send_response(res);
        exit(0);
    }
}
```

6.4 ZOMBIE PROCESSES

6.4.1 LECTURE 4

1. It is natural that a parent process be interested in the execution status (exit code) of a child process it created). To get that state, the parent process uses either the `wait` or `waitpid` system calls.
2. What if the child process ends before the parent calls wait? The system keeps the child process in a zombie state: it doesn't execute anything, but it appears in the list of processes.
3. A zombie process stops when the parent calls `wait` or `waitpid`.
4. Zombie processes are problematic because if they are not "waited" they will grow in number, occupying PIDs and bringing the system to a point where no other processes can be created because of lack of PIDs.
5. Let's go zombie watching. Run the program on the left in one console and the command on the right in another console

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char** argv) {
    int pid;
    pid = fork();
    if(pid == 0) {
        sleep(10); exit(0);
    }
    sleep(15); wait(0); sleep(5);
    return 0;
}
```

```
while true; do clear; ps -f -u rares; sleep 1; done
```

6. If you replace "rares" with the username you are using to run the program, you will see for the first 5 seconds both the parent and child running. For the next 5 seconds, the child will still be there, although it has finished, but it will have a "defunct" label next to it. That is the zombie process. After the parent calls `wait`, the zombie will disappear, and the parent process will run for 5 more seconds.
7. To avoid zombie processes, always call `wait` for each child process you create.
8. The system call `wait` waits for any child process to end. If you want to wait for a specific one, use `waitpid`.
9. The argument for `wait` is a pointer to `int`, where it will return the child process exit code. But since we do not care for that value here, we just pass a zero (`NULL`).

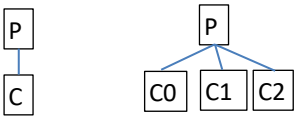
6.4.2 SEMINAR 3

1. The correct way to create a child process must involve calls to fork, exit and wait.
2. To create just on child process, you need to write something like the code on the left. To create multiple processes, you need to write something like the code on the right

```
pid = fork();
if(pid == 0) {
    // do some stuff
    exit(0);
}
// do some other stuff
wait(0);
```

```
for(i=0; i<3; i++) {
    pid = fork();
    if(pid == 0) {
        // do some stuff
        exit(0);
    }
}
// do some other stuff
for(i=0; i<3; i++) {
    wait(0);
}
```

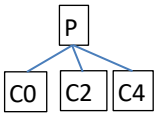
3. Problems
   a. Draw the process hierarchy of the two examples above



   b. Draw the process hierarchy of the code below

```
for(i=0; i<6; i++) {
    if(i % 2 == 0) {
        pid = fork();
        if(pid == 0) {
            exit(0);
        }
    }
}
```

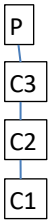
   c.



   d. Draw the process hierarchy when calling `f(3)`, `f` being the function defined below.

```
void f(int n) {
    if(n > 0) {
        if(fork() == 0) {
            f(n-1);
        }
        wait(0);
    }
    exit(0);
}
```

   e.

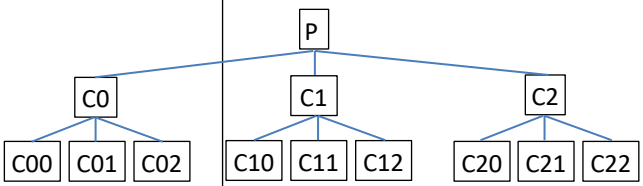


6.4.3 LAB 6

1. Write a program that creates the hierarchy of processes in the diagram below.

```
int main(int argc, char** argv) {
    int i, j;

    for(i=0; i<3; i++) {
        if(fork() == 0) {
            printf("%d %d\n", getppid(), getpid());
            for(j=0; j<3; j++) {
                if(fork() == 0) {
                    printf("%d %d\n", getppid(), getpid());
                    exit(0);
                }
            }
            for(j=0; j<3; j++) {
                wait(0);
            }
            exit(0);
        }
    }
    for(i=0; i<3; i++) {
        wait(0);
    }
    return 0;
}
```



6.5 UNIX SIGNALS

6.5.1 LECTURE 5

1. You may have noticed that there is no `wait` called in the concurrent server code above. That is because we don't have any good place to call it, without knowing about signals. We could try the solution below, but it will make the entire server sequential, so there is not point to it. Calling wait after the infinite loop will never be reached, so it is pointless.

```
while(1) {
    req = get_request();
    pid = fork();
    if(pid == 0) {
        res = process_request(req);
        send_response(res);
        exit(0);
    }
    wait(0); // all is sequential now :-(
}
```

2. Signals are mechanisms that interrupt a processes execution and determine it to run a handler (usually a default one) associated with the signal number.
3. Ctrl-C actually sends signal number 2, also known as SIGINT to the process, which causes the default handler to get executed, and stop the process.
4. When you close a Putty session rudely (from the window X button), all processes in the session receive signal number 1 (SIGHUP - hangup) whose default behavior is also to stop the process.
5. A process can assign a custom function to be executed when a certain signal is received. Let's write a program that refuses to stop when it receives Ctrl-C

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

void f(int sgn) {
    printf("I refuse to stop!\n");
}

int main(int argc, char** argv) {
    signal(SIGINT, f);
    while(1);
    return 0;
}
```

6. The `signal` system call does not signal anything, it only registers a function to be called when the signal is received. To send a signal to a process use the system call `kill` (don't confuse it with the command having the same name)
7. So how do we use signals to solve the concurrent server zombie problem? Whenever a child process stops, the parent receives signal SIGCHLD. We can register a function to SIGCHLD that simply calls wait, and thus the child process, stops being a zombie right away. This is shown in the solution on the right. An even simpler method is to ignore the SIGCHLD signal which causes the system to immediately kill the child process and not turn it into a zombie. This is shown in the solution on the left.

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

void f(int sgn) {
    wait(0);
}

int main(int argc, char** argv) {
    signal(SIGCHLD, f);
    while(1) {
        req = get_request();
        pid = fork();
        if(pid == 0) {
            res = process_request(req);
            send_response(res);
            exit(0);
        }
    }
    return 0;
}
```

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

int main(int argc, char** argv) {
    signal(SIGCHLD, SIG_IGN);
    while(1) {
        req = get_request();
        pid = fork();
        if(pid == 0) {
            res = process_request(req);
            send_response(res);
            exit(0);
        }
    }
    return 0;
}
```

6.5.2 SEMINAR 3

If the students have not seen this in the lecture yet, go with them over signals briefly, just enough to solve the problem below.

1. Implement a program that upon receiving SIGINT (Ctrl-C) asks the user if he/she is sure the program should stop, and if the answer is yes, stops, otherwise it continues.

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

void f(int sgn) {
    char s[32];
    printf("Are you sure you want me to stop [y/N]? ");
    scanf("%s", s);
    if(strcmp(s, "y") == 0) {
        exit(0);
    }
}

int main(int argc, char** argv) {
    signal(SIGINT, f);
    while(1);
    return 0;
}
```

6.5.3 LAB 6

1. Write a program that creates three child processes that just loop forever. When the parent process gets SIGUSR1 it sends SIGUSR2 to the child processes. When the parent process gets SIGUSR2, it sends SIGKILL to the child processes. The child processes should report the signals received. Note in the solution below that the child process does not override the SIGKILL handler, as it will not have any effect.

```c
int children[3];

void hp(int sgn) {
  int i;
  if(sgn == SIGUSR1) {
    printf("%d: sending SIGUSR2\n",getpid());
    for(i=0; i<3; i++) {
      kill(children[i], SIGUSR2);
    }
  }
  else if(sgn == SIGUSR2) {
    printf("%d: sending SIGKILL\n",getpid());
    for(i=0; i<3; i++) {
      kill(children[i], SIGKILL);
    }
  }
}
```

```c
void hc(int sgn) {
  if(sgn == SIGUSR2) {
    printf("%d: Received SIGUSR2\n",getpid());
  }
}
int main(int argc, char** argv) {
  int i;
  signal(SIGUSR1, hp); signal(SIGUSR2, hp);
  for(i=0; i<3; i++) {
    children[i] = fork();
    if(children[i] == 0) {
      signal(SIGUSR2, hc);
      while(1);
      exit(0);
    }
  }
  wait(0); wait(0); wait(0);
  return 0;
}
```

6.6 RUNNING OTHER PROGRAMS USING THE EXEC SYSTEM CALLS

6.6.1 LECTURE 5

1. To run another program from an existing process, the UNIX system offers the exec system calls. There are six of them, but we will only present four of them, and use probably just one or two. The table below shows the four variants differing by whether the program arguments are an array or specified directly as function arguments, and whether the program is given with an absolute path or whether it should be searched for in the `PATH`.

		Search PATH for the program	
		Yes	**No**
Arguments passed as	**Array**	`char* a[] = {"grep", "-E", "/an1/gr911/", "/etc/passwd", NULL}` `execvp("grep", a);`	`char* a[] = {"/bin/grep", "-E", "/an1/gr911/", "/etc/passwd", NULL}` `execv("/bin/grep", a);`
	List	`execlp("grep", "grep", "-E", "/an1/gr911/", "/etc/passwd", NULL);`	`execl("/bin/grep", "/bin/grep", "-E", "/an1/gr911/", "/etc/passwd", NULL);`

2. What exactly is the `PATH`? A UNIX Shell variable (you can however find it in Windows under same name) that contains paths where the Shell should look for the program you run, unless you specify and absolute or relative path like `./myprog`
3. Unexpected behavior: the exec system calls re-use the current process to run the other program. Essentially they wipe out the current process code, and replace it with the code of the new program. If the exec call fails (usually due to the program not being found) the calling process continues to execute.

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

int main(int argc, char** argv) {
    execlp("grep", "grep", "-E", "/an1/gr211/", "/etc/passwd", NULL);
    printf("If grep is in the PATH, then execlp succeeds, and this will never be printed.\n");
    return 0;
}
```

4. As the first argument of a program (argument 0) is always the command name, we need to pass that argument explicitly, hence the duplication of the command name.
5. The last NULL argument is required in order to mark the end of the arguments.

6. If the exec system calls overwrite the current process, how can I run another program and still keep my process? Simple, just create a child process and run exec in it.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char** argv) {
    int pid = fork();
    if(pid == 0) {
        execlp("grep", "grep", "-E", "/an1/gr211/", "/etc/passwd", NULL);
        exit(1);
    }
    // do some other work in the parent
    wait(0);
    return 0;
}
```

7. Although the exec call overwrites the child process code, we still want to keep the `exit` call there, just in case `execlp` fails, thus preventing the child process from executing parent code.

6.6.2 SEMINAR 3

If the students have not seen this in the lecture yet, go with them over exec briefly, just enough to solve the problem below.

1. Write a C program that measures the duration of another programs execution given as command line argument along with its own arguments (i.e. `./measure grep -E "/an1/gr911/" /etc/passwd`). In solving this problem, we will rely on the fact that although `argc` gives us the length of `argv`, `argv` also has an extra element with the value `NULL` (i.e. `argv[argc] == NULL`). In other words, `argc` is redundant, yet comfortable to use. The ending `NULL` of `argv`, allows as to pass it directly to `execvp`.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>

int main(int argc, char** argv) {
    struct timeval t0, t1;
    double duration;
    int status;

    if(argc < 2) {
        fprintf(stderr, "No command provided.\n");
        exit(1);
    }

    gettimeofday(&t0, NULL);
    if(fork() == 0) {
        execvp(argv[1], argv+1);
        exit(1);
    }
    wait(&status);
    gettimeofday(&t1, NULL);

    duration = ((t1.tv_sec - t0.tv_sec)*1000.0 + (t1.tv_usec - t0.tv_usec)/1000.0)/1000.0;
    printf("Duration: %lf seconds\n", duration);

    return WEXITSTATUS(status);
}
```

6.6.3 LAB 6

1. Write a program that gets programs as command line arguments, and executes them using fork and exec. In case exec fails or the program it executes exits with an error code, report the relevant details standard error. Notice in the solution below, the usage of `strerror`, `errno`, `wait` and `WEXITSTATUS` in the solution above. Read the UNIX manual to understand what they do.

```
int main(int argc, char** argv) {
  int i, status;
  for(i=1; i<argc; i++) {
    if(fork() == 0) {
      if(execlp(argv[i], argv[i], NULL) == -1) {
        fprintf(stderr, "Failed to execute program \"%s\": %s\n", argv[i], strerror(errno));
        exit(0);
      }
    }
    wait(&status);
    if(WEXITSTATUS(status) != 0) {
      fprintf(stderr, "Program \"%s\" failed with exit code %d\n", argv[i], WEXITSTATUS(status));
    }
  }
  return 0;
}
```

6.7 UNIX PIPE COMMUNICATION

6.7.1 LECTURE 5

1. Suppose we want to speed-up a calculation by parallelizing it using processes.
    a. Present parallel addition of a vector's elements
    b. The program below is a very simple implementation of the idea above but the result will by 6 instead of 10. Why? The child process stores a[2] in its own memory, and the parent process use the a[2] from its own memory, which does not have the child's calculation.

```
int main(int argc, char**argv) {
    int a[4] = {1, 2, 3, 4};
    int pid;

    pid = fork();
    if(pid == 0) {
        a[2] += a[3];
        exit(0);
    }
    a[0] += a[1];
    wait(0);
    a[0] += a[2];
    printf("%d\n", a[0]);
    return 0;
}
```

2. To fix this we need a way of communicating between processes. The easiest way to do this is using pipes.
    a. A pipe is a buffer in memory that is opened like a file, twice, once for reading and once for writing. The opening is done automatically when we create it and we get back an array with the two `int` descriptors that result. The descriptor on position 0 is for reading, and the one on position 1 is for writing.
    b. Special `read` behaviors
        i. Removes data from the pipe
        ii. If the pipe is empty, it waits for some data to arrive or for no writer to be connected to the pipe
        iii. It may not return all the data requested to be read, but it returns the length of what it actually read
    c. Special `write` behaviors
        i. Adds data to the pipe
        ii. If the pipe is full, it waits for some space to me made or for no reader to be connected to the pipe
        iii. It may not write all the data was given, but it returns the length of what it actually wrote
    d. Behaviors b.ii and c.ii above can lead to your processes getting stuck on pipe operations. To avoid such situations, make sure you close each pipe end (the `int` descriptors) as soon as it is not needed anymore.
    e. It is inherited from parent to child, this being the only mechanism of sharing it. Hence, in order for two processes to communicate through a pipe, one must inherit it from the other, or both must inherit it from a common ancestor.
    f. Here is the re-implementation of the program above, with inter-process communication using pipes.

```
 1 int main(int argc, char**argv) {
 2     int a[4] = {1, 2, 3, 4};
 3     int pid, p[2];
 4
 5     pipe(p);
 6     pid = fork();
 7     if(pid == 0) {
 8         close(p[0]);
 9         a[2] += a[3];
10         write(p[1], &a[2], sizeof(int));
11         close(p[1]);
12         exit(0);
13     }
14     close(p[1]);
15     a[0] += a[1];
16     read(p[0], &a[2], sizeof(int));
17     close(p[0]);
18     wait(0);
19     a[0] += a[2];
20     printf("%d\n", a[0]);
21     return 0;
22 }
```

    g. What would happen if we move line 14 between lines 5 and 6? Child inherits a closed pipe end, namely `p[1]`.
    h. Advice: use a separate pipe for each direction of communication.
    i. Attention: do not confuse a pipe with its ends; there are always two ends to a pipe.
3. Eeny, meeny, miny, moe (children's counting): the parent process and to child processes send a number around, each decrementing it; all processes stop when the number is less or equal to zero.

```
int main(int argc, char**argv) {
  int p2a[2], a2b[2], b2p[2], n;

  pipe(p2a); pipe(a2b); pipe(b2p);

  if(fork() == 0) { // A
    close(p2a[1]); close(a2b[0]);
    close(b2p[0]); close(b2p[1]);
    while(1) {
      if(read(p2a[0], &n, sizeof(int)) <= 0) {
        break;
      }
      if(n <= 0) {
        break;
      }
      printf("A: %d -> %d\n", n, n-1);
      n--;
      write(a2b[1], &n, sizeof(int));
    }
    close(p2a[0]); close(a2b[1]);
    exit(0);
  }

  if(fork() == 0) { // B
    close(p2a[0]); close(p2a[1]);
    close(a2b[1]); close(b2p[0]);
    while(1) {
      if(read(a2b[0], &n, sizeof(int)) <= 0) {
        break;
      }
      if(n <= 0) {
        break;
      }
      printf("B: %d -> %d\n", n, n-1);
      n--;
      write(b2p[1], &n, sizeof(int));
    }
    close(a2b[0]); close(b2p[1]);
    exit(0);
  }

  close(p2a[0]); close(b2p[1]);
  close(a2b[0]); close(a2b[1]);
  n = 20;
  write(p2a[1], &n, sizeof(int));
  while(1) {
    if(read(b2p[0], &n, sizeof(int)) <= 0) {
      break;
    }
    if(n <= 0) {
      break;
    }
    printf("P: %d -> %d\n", n, n-1);
    n--;
    write(p2a[1], &n, sizeof(int));
  }
  close(p2a[1]); close(b2p[0]);
  wait(0); wait(0);
  return 0;
}
```

a. Notice how many `close` function calls we have. We always close all unnecessary pipe ends as soon as possible.
b. Try moving in child process A, the first four `close` calls, near the last two. When you run the program, it will be stuck. Why?

6.7.2 SEMINAR 4

1. Review the following with the students
    a. Special behavior of `read` and `write` when working with pipes
    b. Bi-directional communication: use one pipe per direction

6.7.3 LAB 7

1. Review the following with the students
    a. Special behavior of `read` and `write` when working with pipes
    b. Bi-directional communication: use one pipe per direction

6.8 UNIX FIFO COMMUNICATION

6.8.1 LECTURE 6

1. What if we want to communicate between processes that do not have a common ancestor written by us? Since pipes are only transmitted through inheritance, we need another mechanism.
2. FIFOs are very similar in functionality to pipes and can be used to communicate between any processes.
3. The differences of using FIFO versus pipes are:
    a. FIFOs are files on disk, and consequently they have a unique system-wide ID (the file path) that can be used by processes to address them)
    b. Pipe creation also opens them, but FIFOs need to be created and opened explicitly
    c. Pipes are destroyed when all their ends closed, FIFOs need to be deleted explicitly
    d. FIFOs may live beyond the processes that use them, and if not properly handled, may also keep data during executions, thus potentially creating problems
4. How can we create FIFOs
    a. Using the command `mkfifo`
        i. Example: `mkfifo myfifo`

   b. Using the C instruction `int mkfifo(const char *pathname, mode_t mode)`
      i. Example: `mkfifo("myfifo", 0600)`
5. FIFO access control
   a. Having an unexpected process read or write from you FIFO, will result and data being removed or added to the FIFO in ways your implementation does not expect
   b. For the beginning, it is a good policy to create your FIFOs somewhere in your home directory, and give permissions only to yourself (i.e. 600)
6. FIFO removal can be done either with the `rm` command on the `int unlink(const char *path)` C instruction
7. Eeny, meeny, miny, moe with two processes using FIFO.
   a. Create FIFOs beforehand, using the command `mkfifo a2b b2a`

```
// Program A
int main(int argc, char**argv) {
  int a2b, b2a, n;

  a2b = open("a2b", O_WRONLY);
  b2a = open("b2a", O_RDONLY);

  n = 20;
  write(a2b, &n, sizeof(int));
  while(1) {
    if(read(b2a, &n, sizeof(int)) <= 0) {
      break;
    }
    if(n <= 0) {
      break;
    }
    printf("A: %d -> %d\n", n, n-1);
    n--;
    write(a2b, &n, sizeof(int));
  }
  close(a2b); close(b2a);
  return 0;
}
```

```
// Program B
int main(int argc, char**argv) {
  int a2b, b2a, n;

  a2b = open("a2b", O_RDONLY);
  b2a = open("b2a", O_WRONLY);



  while(1) {
    if(read(a2b, &n, sizeof(int)) <= 0) {
      break;
    }
    if(n <= 0) {
      break;
    }
    printf("B: %d -> %d\n", n, n-1);
    n--;
    write(b2a, &n, sizeof(int));
  }
  close(a2b); close(b2a);
  return 0;
}
```

   b. Remove the FIFOs at the end using command `rm a2b b2a`
8. Special `open` behavior for FIFO: waits for the FIFO to be open for the complementary operation before returning
   a. This is a very useful synchronization mechanism, as open ensures that when a FIFO is open, there is another process ready to deal with this processes operations (i.e. read what this process writes, or write so that this process can read)
   b. Can lead to deadlock, if the FIFOs are not open in the same order. To try it, swap the `open` lines in program B and run the programs again.

6.8.2 SEMINAR 4

1. Ways to communicate through pipe/FIFO
   a. Send byte buffers, prefixed by their length

```
// WRITER
int main(int argc, char**argv) {
  int f, n;
  char* s = "Hello!";

  f = open("w2r", O_WRONLY);
  n = strlen(s)+1;
  write(f, &n, sizeof(int));
  write(f, s, n);
  close(f);


  return 0;
}
```

```
// READER
int main(int argc, char**argv) {
  int f, n;
  char* s;

  f = open("w2r", O_RDONLY);
  read(f, &n, sizeof(int));
  s = (char*)malloc(n);
  read(f, s, n);
  free(s);
  close(f);

  return 0;
}
```

      i. If sending strings in this way, it is a good idea to send the ending zero
   b. Send structures as byte buffers.

```
// header.h

typedef struct {
  int a;
  char b[10];
  float c;
} abc;
```

```
// WRITER
#include "header.h"

int main() {
  int f;
  abc x;

  x.a = 5;
  strcpy(x.b, "qwerty");
  x.c = 1.0f;

  f = open("w2r", O_WRONLY);
  write(f, &x, sizeof(abc));
  close(f);

  return 0;
}
```

```
// READER
#include "header.h"

int main() {
  int f;
  abc x;

  f = open("w2r", O_RDONLY);
  read(f, &x, sizeof(abc));
  close(f);

  return 0;
}
```

i. The length prefix is not required, as we know the length of the message.

ii. If field `b` inside `abc` would be declared as `char*`, and allocated dynamically, this solution would send only the address `b`, not its content, which in the reader does not mean anything.

2. Retrying read/write operations on pipe/FIFO
   a. Both read and write will return the number of bytes they processed (i.e. read or written). If that number is less than what we asked, it may mean the data has not arrived yet, or there is no more space in the pipe/FIFO. This is very common when the size of the data is larger.
   b. A simple way to do this is to write a function that tries to read/write a given number of times.

```
int stubborn_read(int fd, void* buf, int count, int trials) {
  int k, total = 0, n = 0;

  while(total < count && n < trials && (k=read(fd, buf+total, count-total)) > 0) {
    total += k;
    n++;
  }
  return k < 0 ? k : total;
}
```

```
int stubborn_write(int fd, void* buf, int count, int trials) {
  int k, total = 0, n = 0;

  while(total < count && n < trials && (k=write(fd, buf+total, count-total)) > 0) {
    total += k;
    n++;
  }
  return k < 0 ? k : total;
}
```

6.8.3 LAB 7

1. Review the special behavior of `open` when working with FIFO
2. Implement a program that continuously reads two integers from the console, sends them to another process, gets back from that process their sum and product, and display them to the console. The program stops when the sum and the product are equal.
   a. Implementation with pipes

```
int main(int argc, char** argv) {
  int p2c[2], c2p[2], a, b, p, s;

  pipe(p2c); pipe(c2p);
  if(fork() == 0) {
    close(p2c[1]); close(c2p[0]);
    while(1) {
      if(read(p2c[0],&a,sizeof(int)) <= 0){
        break;
      }
      if(read(p2c[0],&b,sizeof(int)) <= 0){
        break;
      }
      s = a + b;
      p = a * b;
      write(c2p[1], &s, sizeof(int));
      write(c2p[1], &p, sizeof(int));
      if(p == s) {
        break;
      }
    }
    close(p2c[0]); close(c2p[1]);
    exit(0);
  }
}
```

```
  close(p2c[0]); close(c2p[1]);
  while(1) {
    printf("a=");scanf("%d", &a);
    printf("b=");scanf("%d", &b);
    write(p2c[1], &a, sizeof(int));
    write(p2c[1], &b, sizeof(int));
    if(read(c2p[0],&s,sizeof(int)) <= 0){
      break;
    }
    if(read(c2p[0],&p,sizeof(int)) <= 0){
      break;
    }
    printf("a+b=%d\na*b=%d\n", s, p);
    if(p == s) {
      break;
    }
  }

  close(p2c[1]); close(c2p[0]);
  wait(0);
  return 0;
}
```

b. Implementation with FIFO (create the FIFOs before running the programs, using `mkfifo a2b b2a`)

```
// Program A
int main(int argc, char** argv) {
  int a2b, b2a, a, b, p, s;

  a2b = open("a2b", O_WRONLY);
  b2a = open("b2a", O_RDONLY);
  while(1) {
    printf("a=");scanf("%d", &a);
    printf("b=");scanf("%d", &b);
    write(a2b, &a, sizeof(int));
    write(a2b, &b, sizeof(int));
    if(read(b2a, &s, sizeof(int)) <= 0) {
      break;
    }
    if(read(b2a, &p, sizeof(int)) <= 0) {
      break;
    }
    printf("a+b=%d\na*b=%d\n", s, p);
    if(p == s) {
      break;
    }
  }
  close(a2b); close(b2a);
  return 0;
}
```

```
// Program B
int main(int argc, char** argv) {
  int a2b, b2a, a, b, p, s;

  a2b = open("a2b", O_RDONLY);
  b2a = open("b2a", O_WRONLY);
  while(1) {
    if(read(a2b, &a, sizeof(int)) <= 0) {
      break;
    }
    if(read(a2b, &b, sizeof(int)) <= 0) {
      break;
    }
    s = a + b;
    p = a * b;
    write(b2a, &s, sizeof(int));
    write(b2a, &p, sizeof(int));
    if(p == s) {
      break;
    }
  }
  close(a2b); close(b2a);
  return 0;
}
```

6.9 POPEN

6.9.1 LECTURE 6

1. If you need to run a program or any general Shell command from C code, and get back its standard output, or send data to its standard input, you need `popen`.
2. Write a program that generates the "99 bottles of beer" song, and displays it in a pager like `less`.

```
int main(int argc, char** argv) {
    int i;
    FILE* fp;

    fp = popen("less", "w");
    for(i=99; i>0; i--) {
        fprintf(fp, "%d bottles of beer on the wall,\n", i);
        fprintf(fp, "    %d bottles of beer.\n", i);
        fprintf(fp, "If one of those bottles should happen to fall,\n");
        fprintf(fp, "    %d bottles of beer on the wall.\n\n", i-1);
    }
    pclose(fp);
    return 0;
}
```

3. Write a C program that displays the user with the most processes in the system, and how many processes he or she has. Use a Shell command to find the data.

```
int main(int argc, char** argv) {
    int n;
    char u[64];
    FILE* fp;

    fp = popen("ps -ef | awk '{print $1}' | sort | uniq -c | sort -nr | head -n 1", "r");
    fscanf(fp, "%d %s", &n, u);
    pclose(fp);
    printf("%s: %d\n", u, n);
    return 0;
}
```

6.10 UNIX PROCESS FILE DESCRIPTOR MANIPULATION WITH DUP() AND DUP2()

6.10.1 LECTURE 6

1. The goal of this section is to see behind the scenes of input/output redirections, and eventually be able to write in C the equivalent of the command `ps -ef | grep -E "^root" | awk '{print $2}'`
2. The process file descriptor table is an array holding handles for accessing the files open by the process.
3. Consider the program below.

```
int main(int argc, char**argv) {
    int fd, myfifo, pa[2], pb[2];

    fd = open("a.txt", O_RDWR);
    myfifo = open("myfifo", O_RDONLY);
    pipe(pa);
    pipe(pb);

    return 0;
}
```

The file descriptor table will look as follows

Index	Value
0	Handle for reading from the console (standard input)
1	Handle for writing to the console (standard output)
2	Handle for writing to the console (standard error)
3	Handle for reading/writing to file a.txt. Variable `fd` has the value of the index, i.e. 3
4	Handle for reading from FIFO myfifo. Variable `myfifo` has the value of the index, i.e. 4
5	Handle for reading from the first PIPE created. Variable `pa[0]` has the value of the index, i.e. 5
6	Handle for writing to the first PIPE created. Variable `pa[1]` has the value of the index, i.e. 6
7	Handle for reading from the second PIPE created. Variable `pb[0]` has the value of the index, i.e. 7
8	Handle for writing to the second PIPE created. Variable `pb[1]` has the value of the index, i.e. 8

4. The file descriptor table can be manipulated using the system calls `dup()` and `dup2()`.
    a. `int dup(int oldfd)` makes a copy of the handle at index `oldfd` to a new entry, and returns the index of the entry just created. For example, adding line `int x = dup(myfifo)` to the program above, just before the `return`, will result in the following line being added to the file descriptor table, and variable `x` will have the value 9.

9	Handle for reading from FIFO myfifo.

    b. `int dup2(int oldfd, int newfd)` makes a copy of the handle at index `oldfd` to index `newfd`, overwriting the existing value (it also closes it silently before overwriting it). For example, adding line `dup2(pa[0], 0)` to the program above, just before the `return`, will result in line 0 containing the handle for reading from the first PIPE created. Consequently, any reading from the standard input will use the first PIPE created.

0	Handle for reading from the first PIPE created.

5. Let's implement in C a program that does the equivalent of running the command `ps -ef | grep -E "^root" | awk '{print $2}'`, by running the three programs and transferring the data between them using pipes.

```
int main(int argc, char** argv) {
  int p2g[2], g2a[2];

  pipe(p2g); pipe(g2a);
  if(fork() == 0) {
    close(p2g[0]);close(g2a[0]);close(g2a[1]);
    dup2(p2g[1], 1);
    execlp("ps", "ps", "-ef", NULL);
    exit(1);
  }
  if(fork() == 0) {
    close(p2g[1]); close(g2a[0]);
    dup2(p2g[0], 0); dup2(g2a[1], 1);
    execlp("grep", "grep", "-E", "^root", NULL);
    exit(1);
  }
  if(fork() == 0) {
    close(p2g[0]);close(p2g[1]);close(g2a[1]);
    dup2(g2a[0], 0);
    execlp("awk", "awk", "{print $2}", NULL);
    exit(1);
  }

  close(p2g[0]); close(p2g[1]);
  close(g2a[0]); close(g2a[1]);

  wait(0); wait(0); wait(0);

  return 0;
}
```

6. How do you undo a `dup2` call: use `dup` to make a copy of the entry that will be overwritten, and when you want to reset it, use again `dup2` with the copy.

```
int x = dup(1);
dup2(p[1], 1);
....
dup2(x, 1);
```

6.10.2 SEMINAR 4

1. Implement a simple `popen`/`pclose` API using `fork`, `exec`, and `dup2`.

```
FILE* mypopen(char* cmd, char* type) {
    int p[2], caller_idx, child_idx;
    pipe(p);

    caller_idx = 0;
    if(type[0] == 'w') {
        caller_idx = 1;
    }
    child_idx = (caller_idx+1) % 2;

    if(fork() == 0) {
        close(p[caller_idx]);
        dup2(p[child_idx], child_idx);
        if(execlp("bash", "bash", "-c", cmd, NULL) < 0) {
            close(p[child_idx]);
            exit(1);
        }
    }

    close(p[child_idx]);
    return fdopen(p[caller_idx], type);
}
```

```
void mypclose(FILE* fd) {
    fclose(fd);
    wait(0);
}


// Usage example
FILE* f=mypopen("who", "r");
// read from f
mypclose(f);


FILE* f=mypopen("less","w");
// write to f
mypclose(f);
```

   a. Variable `caller_idx` is the pipe end to be returned to the caller. As the child process needs to do the opposite operation on the pipe (i.e. if the caller wants to read, the child process must write, and vice versa), the child process will close it. Variable `child_idx` is the other pipe end, which will be closed by the caller but used by the child process.

6.11 UNIX IPC SHARED MEMORY

6.11.1 LECTURE 7

1. UNIX IPC is a set of mechanisms (other than pipe and FIFO) for communicating among processes (IPC = Inter-Process Communication)
   a. **Semaphore**: synchronization mechanism
   b. **Message queues**: message-based communication
   c. **Shared memory**: communication through a common memory region
2. We will only look into shared memory
3. IPC identification

a. Just like FIFOs, IPCs allow any two processes to communicate through them, but unlike FIFOs, IPCs are not files on disk
b. Identified by a number that is unique in the system
c. This number is chosen by the developer (explicitly or using specialized functions) and provided to all processes that need to communicate through that specific IPC

4. IPC cleanup
   a. IPCs are persistent structures in the operating system
   b. The operating system imposes limits for the size and number of IPCs
   c. If not cleanup up properly, the system will refuse to allow the creation of new IPCs
   d. You can see all existing IPCs using the command `ipcs`
   e. You can delete an IPC using command `ipcrm`, as long as you have permissions on that IPC

```
> ipcs

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x58df7f12 0          rares      644        100        0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

> ipcrm -M 0x58df7f12
```

5. Shared memory API
   a. Create or get a handle to an existing shared memory segment: `shmget`
   b. Attach - map the shared memory segment to a pointer in your process: `shmat`
   c. Detach - unmap the shared memory segment from you pointer: `shmdt`
   d. Control (configure, delete, etc.) the shared memory segment: `shmctl`
6. Implement two programs that communicate through a shared memory segment containing four integers: `a`, `b`, `s`, and `p`. The first program set `a` and `b` to some values, and the second sets `s` to the `a+b` and `p` to `a*b`;

```
// header.h

struct absp {
    int a;
    int b;
    int s;
    int p;
};
```

```
//Program A
#include "header.h"
int main() {
  int shmid, k = 0;
  struct absp *x;

  shmid = shmget(1234, sizeof(struct absp),
                 IPC_CREAT | 0600);
  x = shmat(shmid, 0, 0);

  while(1) {
    x->a = k++ % 100;
    x->b = k++ % 100;
    if(x->p == x->s) {
      break;
    }
  }

  shmdt(x);
  shmctl(shmid, IPC_RMID, NULL);
  return 0;
}
```

```
// Program B
#include "header.h"
int main() {
  int shmid;
  struct absp *x;

  shmid = shmget(1234, 0, 0);

  x = shmat(shmid, 0, 0);

  while(1) {
    x->s = x->a + x->b;
    x->p = x->a * x->b;
    if(x->p == x->s) {
      break;
    }
  }

  shmdt(x);
  return 0;
}
```

   a. The programs above will not function in any way close to what we want, because we have no mechanisms to ensure they work on the shared memory in turns. Instead, they work ignoring the presence of the other.
   b. If the two programs are started manually one by one, it is likely that program A deletes the shared memory before B is done with it, resulting in errors. This can happen if A finds `p` and `s` being equal simply because their value happens to be zero, and then breaks from the loop and deletes the shared memory.
   c. So, shared memory without synchronization mechanisms is guaranteed to cause trouble.

7 POSIX THREADS (PTHREADS)

7.1 PTHREAD CREATION AND SCHEDULING

1. Threads are another mechanism for implementing concurrent programs that allow faster creation, reduced memory usage, and much simpler communication. POSIX threads (Pthreads) are the native threads implementation in Linux, but every modern operating system provides thread creation libraries. Below is a very basic thread creation example, using the Pthreads library.

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

void* f(void* a) {
    printf("f\n");
    return NULL;
}
```

```
int main(int argc, char** argv) {
    pthread_t t;

    pthread_create(&t, NULL, f, NULL);
    printf("main\n");
    pthread_join(t, NULL);
    return 0;
}
```

   a. A thread always executes a given function, in our case `f()`.
   b. The Pthread library requires the thread function to have a specific signature. It should have a single `void*` argument and it should return `void*`.
   c. The call to `pthread_create` creates and starts a thread that executes the given function pointer `f`, and populates the thread handler `t`.
   d. The second argument to `pthread_create` is a pointer to `pthread_attr_t` which controls various aspects of the thread creation and execution. In our case, by setting it to `NULL`, we use the default settings.
   e. The fourth argument to `pthread_create` is the argument to be passed to the function `f()`. In our case, `f()` doesn't use its argument, so we pass `NULL`.
   f. The call to `pthread_join` waits for the thread identified by the handle `t` to finish.
   g. The second argument to `pthread_join` will contain the value returned by function `f()`. Passing it as `NULL` means we do not need this value, so we ignore it.
   h. To compile a program that uses Pthreads, you need to either pass the `-pthread` argument to `gcc`, or to tell it to use the Pthreads library, which you can do passing the `-lpthread` argument.

      `gcc -Wall -g -o a a.c -pthread` or `gcc -Wall -g -o a a.c -lpthread`

   i. The output of this program depends on how the operating system schedules the execution of these threads, consequently we may see the two `printf`-ed values displayed in <u>any</u> order. This is essential to understanding how concurrency works.

2. It is difficult to see the non-deterministic aspect of thread scheduling using the program above, because the thread execution is extremely short, and it takes a lot of trials to see the lines printed in a different order. Running the program on a busier computer would make this a little easier to see, but we can make this easily obvious using thread functions that do a lot more.

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

int n = 1;

void* fa(void* a) {
    int i;
    for(i=0; i<n; i++) {
        printf("fa\n");
    }
    return NULL;
}

void* fb(void* a) {
    int i;
    for(i=0; i<n; i++) {
        printf("fb\n");
    }
    return NULL;
}
```

```
int main(int argc, char** argv) {
    int i;
    pthread_t ta, tb;

    if(argc > 1) {
        sscanf(argv[1], "%d", &n);
    }

    pthread_create(&ta, NULL, fa, NULL);
    pthread_create(&tb, NULL, fb, NULL);

    for(i=0; i<n; i++) {
        printf("main\n");
    }

    pthread_join(ta, NULL);
    pthread_join(tb, NULL);

    return 0;
}
```

   a. We now have two thread functions that is each run in a separate thread
   b. The command line argument specifies how many iterations each thread will do. Notice that each thread has access to the global variables.
   c. To show the thread scheduling without having long printouts, we pipe the output of the program through `uniq -c` without sorting the out first, thus displaying how many iterations each thread did before the scheduler switched to another. Below are two executions of the same program, taken a few moments apart.

```
./tb 10000|uniq -c
  201 main
    1 fa
   51 main
    1 fa
    1 main
    1 fa
   27 main
    1 fa
   30 main
    1 fa
 9690 main
10000 fb
 9995 fa
```

```
./tb 10000|uniq -c
 189 fa
   1 main
  50 fa
   2 main
   1 fa
   1 main
   1 fa
   1 main
  31 fa
   1 main
  31 fa
   1 main
  34 fa
   1 main
  35 fa
   1 main
 168 fa
   1 main
  11 fa
   1 main
```

```
   1 fa
  27 main
   1 fa
   1 main
  35 fa
   1 main
  34 fa
   1 main
   1 fa
   1 main
   1 fa
  34 main
   1 fa
   1 main
   1 fa
  33 main
   1 fa
  36 main
   1 fa
9855 main
5153 fb
   1 fa
```

```
   1 fb
  15 fa
   1 fb
   1 main
   1 fa
  34 fb
   1 main
  36 fb
   1 fa
  36 fb
   1 fa
   1 fa
   1 fb
   1 fa
  35 fb
   1 fa
  34 fb
   1 fa
  37 fb
   1 fa
 115 fb
   1 fa
```

```
   1 fb
  28 fa
   1 fb
   1 fa
   2 fa
   1 fb
   1 fa
   1 fa
   1 fb
  31 fa
   1 fa
   1 fb
   1 fa
   1 fb
   1 fa
   1 fa
   1 fb
   1 fa
  34 fb
   1 fa
```

```
   1 fb
   2 fa
   1 fb
  31 fa
   1 fb
  32 fa
   1 fb
  35 fa
   1 fb
   1 fa
  35 fa
   1 fb
  37 fa
   1 fb
9107 fa
4466 fb
```

7.2 PTHREAD ARGUMENT PASSING

1. The last program can be written using a single thread function that gets value to display in the console as argument.

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

int n = 1;

void* f(void* a) {
    int i;
    for(i=0; i<n; i++) {
        printf("%s\n", (char*)a);
    }
    return NULL;
}

int main(int argc, char** argv) {
    int i;
    pthread_t ta, tb;
    if(argc > 1) {
        sscanf(argv[1], "%d", &n);
    }

    pthread_create(&ta, NULL, f, "fa");
    pthread_create(&tb, NULL, f, "fb");

    for(i=0; i<n; i++) {
        printf("main\n");
    }

    pthread_join(ta, NULL);
    pthread_join(tb, NULL);

    return 0;
}
```

2. If we want to pass `n` as an argument to the thread too (instead of having it a global variable), we need to declare a `struct` since the thread function can only get one argument.

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

struct arg_t {
    char* name;
    int count;
};

void* f(void* a) {
    int i;
    struct arg_t* x = (struct arg_t*)a;
    for(i=0; i<x->count; i++) {
        printf("%s\n", x->name);
    }
    return NULL;
}

int main(int argc, char** argv) {
    int i, n = 1;
    pthread_t ta, tb;
    struct arg_t aa, ab;
    if(argc > 1) {
        sscanf(argv[1], "%d", &n);
    }

    aa.name = "fa"; ab.name = "fb";
    aa.count = n; ab.count = n;

    pthread_create(&ta, NULL, f, &aa);
    pthread_create(&tb, NULL, f, &ab);

    for(i=0; i<n; i++) {
        printf("main\n");
    }

    pthread_join(ta, NULL);
    pthread_join(tb, NULL);

    return 0;
}
```

3. Let's create 10 threads, pass to each of them the order number in which it was created, and have it displayed. The program will include a serios bug that causes to a race condition and yields puzzling outputs.

```c
#include <stdio.h>
#include <pthread.h>

void* f(void* a) {
    printf("%d\n", *((int*)a));
    return NULL;
}

int main(int argc, char** argv) {
    int i;
    pthread_t t[10];

    for(i=0; i<10; i++) {
        pthread_create(&t[i], NULL, f, &i);
    }

    for(i=0; i<10; i++) {
        pthread_join(t[i], NULL);
    }

    return 0;
}
```

a. Multiple executions of this program yielded the outputs below

A	B	C	D	E
1	1	1	1	0
2	5	5	2	1
6	2	3	6	1
6	6	4	6	1
3	6	2	3	1
4	6	7	4	1
7	7	6	7	1
8	8	8	8	1
9	9	4	8	1
4	4	5	8	1

b. Analyzing the outputs above, we notice that:
   i. executions A, B, C, and D do not display 0
   ii. execution D does not display 9
   iii. execution E displays one 0 and then nine 1s
   iv. all executions display duplicate numbers

c. The root cause of these behaviors is the passing of `&i` to all threads. While we intend to pass to each thread the order number at which it was created, in reality, we pass to all threads the same value: *the address of* `i`. The main function changes the value of `i` in two `for` cycles (one for thread creation and one for joining). Depending how the threads get the CPU, they will print whatever they find at the memory location `&i`.

d. To avoid this situation, we can either have an array of `int`s and send each thread the address of another element (first solution below), or dynamically allocate an `int` before creating each thread (second solution below).

```c
#include <stdio.h>
#include <pthread.h>


void* f(void* a) {
  printf("%d\n", *(int*)a);
  return NULL;
}


int main(int argc, char** argv) {
  int i, a[10];
  pthread_t t[10];


  for(i=0; i<10; i++) {
    a[i] = i;
    pthread_create(&t[i], NULL, f, &a[i]);
  }


  for(i=0; i<10; i++) {
    pthread_join(t[i], NULL);
  }

  return 0;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void* f(void* a) {
  printf("%d\n", *(int*)a);
  free(a);
  return NULL;
}

int main(int argc, char** argv) {
  int i;
  int* p;
  pthread_t t[10];

  for(i=0; i<10; i++) {
    p = (int*)malloc(sizeof(int));
    *p = i;
    pthread_create(&t[i], NULL, f, p);
  }

  for(i=0; i<10; i++) {
    pthread_join(t[i], NULL);
  }

  return 0;
}
```

e. Notice where we free the allocated memory in the second solution. What would happen if we freed it right after the call to `pthread_create`?

f. The outputs of these implementations will still be non-deterministic due to the order of the `printfs`, but they will contain all the expected values.

A B C D E

0 5 1 6 3 4 2 7 9 8	0 5 2 1 8 6 3 7 9 4	1 4 5 2 3 6 0 8 7 9	1 4 0 2 5 3 6 7 8 9	0 6 1 4 2 7 3 8 9 5

8 POSIX SYNCHRONIZATION MECHANISMS

1. Let's write a program that creates 10 threads, each of which increments a global variable as many times given as a command line argument.

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

int count = 0;

void* f(void* a) {
    int i;
    for(i=0; i<*(int*)a; i++) {
        count++;
    }
    return NULL;
}

int main(int argc, char** argv) {
    int i, n = 10;
    pthread_t t[10];

    if(argc > 1) {
        sscanf(argv[1], "%d", &n);
    }

    for(i=0; i<10; i++) {
        pthread_create(&t[i], NULL, f, &n);
    }

    for(i=0; i<10; i++) {
        pthread_join(t[i], NULL);
    }

    printf("%d\n", count);

    return 0;
}
```

2. We mentioned before that race conditions do not always manifest, and that the best way to make them more likely to appear is to increase the concurrency. We will run this program with various arguments and repeating those executions to see whether the output gets corrupted or not.

```
for n in 10 100 1000 10000 100000 1000000
do
    k=0
    while [ $k -lt 10 ]; do
        ./th $n
        k=`expr $k + 1`
    done
done
```

Correct result:	10	100	1000	10000	100000	1000000
	100	1000	10000	86337	610108	5068606
	100	1000	10000	74379	641742	5342264
	100	1000	9446	85734	590373	5777329
	100	1000	10000	82615	674556	5122867
	100	1000	9526	81122	601525	5477098
	100	1000	9754	79517	1000000	5288377
	100	1000	10000	79977	671355	5405598
	100	1000	10000	78419	599169	5340372
	100	1000	9115	81924	559812	5411218
	100	951	10000	77140	557539	2303521

3. You can see that as the concurrency increases, the values are more and more corrupted. Even though with lower concurrency things appear to be correct, it is just a matter of time until the data will be corrupted there as well.
4. This situation is called race condition, with `count` being the critical resource and the line `count++;` being the critical section.
5. Race conditions are solved using synchronization mechanisms.

8.1 MUTEXES

1. A mutex is a synchronization mechanism that has two main operations: `lock` and `unlock`.
2. Only one thread can complete the `lock` operation at one time, any other threads attempting to `lock` the mutes, will have to wait until the "winning" thread calls `unlock`.
3. The race condition in the code above can be avoided using a mutex as shown below

```c
#include <stdio.h>
#include <pthread.h>

int count = 0;
pthread_mutex_t m;

void* f(void* a) {
    int i;
    for(i=0; i<*(int*)a; i++) {
        pthread_mutex_lock(&m);
        count++;
        pthread_mutex_unlock(&m);
    }
    return NULL;
}

int main(int argc, char** argv) {
    int i, n = 10;
    pthread_t t[10];
    if(argc > 1) {
        sscanf(argv[1], "%d", &n);
    }

    pthread_mutex_init(&m, NULL);
    for(i=0; i<10; i++) {
        pthread_create(&t[i], NULL, f, &n);
    }

    for(i=0; i<10; i++) {
        pthread_join(t[i], NULL);
    }
    pthread_mutex_destroy(&m);

    printf("%d\n", count);

    return 0;
}
```

a. We wrap the critical section between mutex `lock`/`unlock` calls.
b. The `lock`/`unlock` operations could be moved outside the `for` loop. However, that would drastically reduce the concurrency, as the entire code of the thread would be executed non concurrently.
c. The second argument to pthread_mutex_init is a pointer to pthread_mutexattr_t which controls various aspects of the mutex creation and execution. In our case, by setting it to NULL, we use the default settings.

8.2 READ-WRITE LOCKS

8.3 CONDITIONAL VARIABLES

8.4 SEMAPHORES

8.5 BARRIERS