Operating Systems & Concurrent Programming

Workbook

Cezary Sobaniec

v1.0
2020-06-08

Poznań University of Technology
Institute of Computing Science

Contents

1 Getting started

1.1 Prelude

1. Linux vs other Uni* operating systems.
2. Your own Linux: bare metal, virtual (e.g. using VirtualBox) or Cygwin.
3. Our resources: servers polluks.cs.put.poznan.pl and sirius.cs.put.poznan.pl, domain controller.
4. Access protocols and tools:
   - using Linux: `ssh` and `scp`,
   - using Windows: PuTTY and WinSCP.

1.2 Logging in to the system

1. Log in to the system. Run *Terminal emulator*.
2. Change your initial password using `passwd`.
3. Run `ls` (ang. *list*) command using various forms:

```
# ls
# ls -l
# ls -a
```

```
# ls -la
# ls -al
# ls --all
# ls -l /usr
```

General format of running commands:

```
# COMMAND -SWITCHES ARGUMENTS
```

4. Log in to the system using text mode. Press **Ctrl–Alt–F1** or **Ctrl–Alt–F2**, and then **Alt–F7**.
5. Check current directory with `pwd` (ang. *print working directory*) command. Change current directory using `cd` (ang. *change directory*):

```
# cd /
# cd /usr
# cd local
# cd bin
# cd ..
# cd man
# cd .
# cd
```

6. Explore directory structure of the system. Check the following directories: `/bin`, `/etc`, `/dev`, `/home`, `/lib`, `/var`, `/usr`, `/sbin`, `/tmp`.

1.3 System manual

1. Load system manual for command `ls`:

```
# man ls
```

and try to navigate through the documentation using:

q quit,

Enter, **j**, **k** single line scrolling,

Spacja, **Ctrl–f**, **Ctrl–b**, **Ctrl–d**, **Ctrl–u** full/half page scrolling,

g, **G** first/last line,

/ search the document,

n, **N** next/previous occurrence of the searched pattern.

2. Switch the default language:

```
# export LANG=pl_PL.UTF-8
# man ls
```

pl stands for polish language, PL stands for Poland, UTF-8 is an encoding standard.

3. Try other commands, e.g. `rm`:

```
# rm abcd
rm: nie można usunąć 'abcd': Nie ma takiego pliku ani katalogu
# export LANG=de_DE.UTF-8
# rm abcd
rm: das Entfernen von 'asdf' ist nicht möglich: Datei oder Verzeichnis nicht gefunde
# export LANG=fr_FR.UTF-8
# export LANG=ru_RU.UTF-8
# export LANG=zh_TW.UTF-8
```

4. Sections of the standard system manual: 1 – commands, 2 – system calls, 3 – library functions, 4 – special files, 5 – configuration files, 6 – games, 7 – miscellany section, 8 – administration and privileged commands.

```
# man sleep
# man 3 sleep
# whatis sleep
```

References to man pages: e.g. sleep(3) denotes a man page for sleep in section 3. Introductory pages: intro(n).

5. Read about standard directory structure: hier(7).

6. Find a command for creating new directories. Use `apropos` for that purpose:

```
# apropos directory
# apropos "make.*director"
```

Characters "`.*`" comes from *regular expression* notations and denote any sequence of characters, including an empty sequence.

7. Check the `whereis` command:

```
# whereis ls
```

8. Test an alternative manual system called `info`:

```
# info gawk
```

Use the following commands:

q	quit,
Tab	go to the next link,
Enter	go to the page pointed by highlighted link,
l	back to the previous (last) page,
n, **p**	go to the next/previous page in a sequence,
u	go up in page hierarchy,
t	go to the top page,

`/` search the documentation,
`i` go to index.

Test another viewer of the `info` manual:

```
# pinfo gawk
```

9. Prepare a manual page for printing:

```
# man -t ls > out.ps
# evince out.ps
```

1.4 The shell

1. Check the shell you are working with by pressing `Ctrl-x, Ctrl-v`.
2. Recall recently issued command using ↑ i ↓, and rerun them. Display the list using `history` command.
3. Test interactive reverse search of the history of commands by pressing `Ctrl-r` and typing fragments of the command.
4. Use the file completion mechanism for commands, file and directory names:

```
# mk Tab Tab d Tab
# ls p Tab
```

   Find all commands with names starting with `pre`.

5. Stop long-running commands:

```
# sleep 10
Ctrl-c
```

6. Try scrolling of the terminal using `Shift-PgUp`, `Shift-PgDn`, `Shift-`↑ i `Shift-`↓ (also in text mode). Try `Scroll Lock`!
7. Try using line editing shortcuts:

`Ctrl-a`, `Ctrl-e` go to start/end of line,
`Ctrl-f`, `Ctrl-b` go left/right by char by char,
`Ctrl-k` delete the text from cursor to the end of line,
`Ctrl-w` delete the word to the left from the cursor position.

8. Refresh the screen using `Ctrl-l` and using the `clear` command.
9. Check what `Ctrl-d` does.
10. Use *glob names* for accessing files, glob(7). Character * represents any sequence

of characters (including an empty seq), `?` represents any single character, `[]` represents a single character from the explicit set of characters:

```
# touch a.txt b.txt c.txt
# touch a.dat b.dat ab.dat
# ls *.txt
# ls a*
# ls *.*
# ls *
# ls [ab].txt
# ls ?.dat
# ls *.???
```

Warning: do not confuse glob names with regular expressions! These are two, distinct mechanisms.

11. Print all programs from `/usr/bin` directory with a single-character name[1]. Print all programs with names consisting of 3 characters. Print all programs with a single digit in names.

[1] Run `shopt -s globasciiranges` in case of inconsistent results.

2 File System

2.1 Directories and files

1. Create example sub-directories in your home directory:

```
# mkdir x1
# mkdir x2 x3
# mkdir -p x1/x4/x5
```

   Display the directory structure using `ls` and `tree` commands:

```
# ls -lR
# tree
# tree x2
```

   Remove an empty directory using `rmdir`:

```
# rmdir x1
# rmdir -p x2
```

2. Create example (empty) files using `touch`:

```
# touch a.txt
# touch b.csv c.dat
```

3. Try copying files using `cp`:

```
# cp a.txt b.txt
# cp -i a.txt b.txt
# cp a.txt b.txt x1/
```

```
# cp a.txt x1/d.txt
# cp -v *.txt x2/
# cp -t x2/ *.txt
```

Pay attention to the context of `cp` invocation, e.g.:

```
# cp a.txt ab
```

What if a) `ab` does not exist, b) `ab` is a regular file, a) `ab` is a directory.

4. Try removing files using `rm`:

```
# rm a.txt b.txt
# rm *.txt
# rm -i *.txt
```

This command is used also for removing whole directory structures:

```
# rm -r x1/
```

Check the meaning of `-f` switch of `rm` command.

5. Try copying of whole directory structures:

```
# cp -r x1/ x5
```

What if `x5` exists and is a directory? Try the following variant:

```
# cp -r -T x1 x5
```

6. Rename a file:

```
# mv a.txt b.txt
```

Move a file:

```
# mv a.txt x2/
```

Rename and move:

```
# mv a.txt x2/b.txt
```

Check the functions of `-v`, `-i`, `-t`, `-T` switches. Check the behaviour of `mv` commend in case of directories:

```
# mv x1/ x2/
# mv a.txt ..
# mv ../a.txt .
```

(*) 7. How to remove a file named `-v`?

```
# ls > -v
# rm -v
rm: missing operand
Try 'rm --help' for more information.
```

(*) 8. Try another, more general command `mmv(1)` for renaming files, e.g.:

```
# mmv "*.txt" "#u1.TXT"
```

2.2 Access rights

1. Analyse access rights of the following files:

```
-rwxr-xr-x  1 conrad   users         164 Feb 21 17:19 test2
-rw-rw----  1 kamil    students    24250 May 27  2002 data.txt
-r-----r--  1 marcus   users       28014 Feb 21 17:43 example.jpg
-rwxrwxrwx  1 conrad   students     4563 Mar  8 04:43 help.html
-r-xr--r--  1 voytek   users        4611 Mar  8 04:42 solitaire
```

   a) Which files can be modified by user conrad who belongs to groups users and students?
   b) Which files can be read by user marcus who belongs to the students group?
   c) Who can execute the program `solitaire`?

2. Modify access rights using `chmod` command:

```
# ls -l a.txt
# chmod u+x a.txt
# ls -l a.txt
# chmod go-rwx a.txt
# chmod u=rw,g=r,o= a.txt
# chmod u=rwx,go-wx a.txt
```

3. Which access rights are necessary for:

   a) listing the contents of a directory,
   b) entering a directory,
   c) creating and deleting files inside a directory.

4. Modify access rights for a whole directory structure:

```
# chmod -R g+w x1/
# chmod -R -c g+w x1/
# chmod -R -c g+x x1/
# chmod -R -c g+X x1/
```

5. Try numeric representations of access rights, e.g.:

```
# chmod 764 a.txt
```

6. Check the access rights of newly created file depending on the *umask*:

```
# umask
# umask 077
# touch g.txt
# ls -l g.txt
# rm g.txt
# umask 007
# touch g.txt
```

```
# ls -l g.txt
```

7. Check the access rights of the following files: `/etc/passwd`, `/etc/shadow`, `/usr/bin/passwd`. Why a regular user can change his/her password?

   Set SUID and SGID access rights to your files:

   ```
   # chmod u+s a.txt
   # chmod g+s a.txt
   ```

8. Check access rights of `/tmp` directory. Try to create and delete files inside this directory.
9. Read about `chown` and `chgrp` commands for changing ownership of files.

2.3 Searching for files

Command `locate`

1. Find all files`locate` containing the word "print" in their names:

   ```
   # locate print
   ```

   Try to be more specific:

   ```
   # locate "/usr/*print*"
   ```

   Find all files which are named exactly "print".
2. Find all subdirectories of `/usr` with name `bin`.
3. Find all programs with two letter names, and the letters should be "a" to "e".

Command `find`

1. Search `/usr/share` directory structure and find all files with extension `TXT`. Compare the results with results returned by `locate`:

   ```
   # find /usr/share -name "*.TXT"
   ```

2. Find all sub-directories of `/usr/share`:

   ```
   # find /usr/share -type d
   ```

   Find all sub-directories of `/usr` with name `bin`.
3. Find all files within `/usr/share` that do not include dots in their names:

   ```
   # find /usr/share ! -name "*.*"
   ```

4. Find all files under `/usr` with no content:

   ```
   # find /usr -size 0
   ```

and files with size between 100 and 200 MiB:

```
# find /usr -size +100M -size -200M
```

Find the biggest file inside `/usr`.

5. Using time expressions (`-mtime`, `-atime`, `-ctime`, and also `-mmin`, `-amin`, `-cmin`) find:
   a) files under `/usr` have been modified over the past week,
   b) files that have not been used for over a month,
   c) files from your home directory that have changed its status today.
6. Execute a command for every file found:

```
# find /usr -size 0 -exec ls -l {} \;
# find /usr -size 0 -ls
```

2.4 Links

1. Create a hard link to a file:

```
# ln file.txt link.txt
```

   a) modify the contents of the original file and the link,
   b) modify access rights,
   c) check i-node number of the files using `ls -i`,
   d) observe the link counter while creating and removing links,
   e) create a hard link to a directory.
2. Create a symbolic link to a file:

```
# ln -s file.txt link.txt
```

   and repeat above tests.

3 Editors

3.1 Legendary vi

1. Run `vi`, and then switch from *command mode* to *inserting mode* using: **i**, **a**, **A** or **R**. Switch back to command mode using **Esc**.
2. Try navigating commands: **j**, **k**, **h**, **l**, **0**, **$**, **G**, **w**, **e**, **b**.
3. Delete characters: **x**, **X** and **dd**. Undo by pressing **u** and redo by pressing **Ctrl-r**.
4. Search the document using **/** i **?**. Go to the next occurrence of the characters using n and N. Use *regular expressions* and find in the document:
   - lines starting with „#”,
   - empty lines,
   - non-empty lines outside comments,
   - numbers,
   - hexadecimal numbers (as in C programming language),
   - sequences of spaces longer than 1.
5. Copy and paste: **yy**, **p**, **P**.
6. Parametrize commands with location markers: **d0**, **dG**, **d$**, **c$**, **y$**. Try numerical parametrization of any command: **10G**, **5i**, **d5d**, **d2w**, **d3k**, **d10h**, **d/x**.

7. File operations: saving **:w**, including external file **:r**. Exiting: **:q**, **:q!**, with saving: **:x**, **ZZ**.
8. Configuration **:set number**, **:set autoindent**. Permanent configuration: files `~/.exrc` and `~/.vimrc`.
9. Vim improvements: : block highlighting **V**, **v**, **Ctrl-v** + one of the standard commands: **d**, **y**, **c**. Syntax highlighting **:syntax on**.

3.2 Other tools

1. Start `mcedit` editor which is a part of Midnight Commander (command `mc`). Edit, save by pressing **F2**, and exit by pressing **F10**.
2. Start `joe` editor. Try different commands starting with **Ctrl-k**, e.g. **Ctrl-k, h**. Exit using **Ctrl-k, q**.
3. Start `pico` or `nano` editor. Exit using **Ctrl-x**.

4
Processes

4.1 List of processes

1. List your own processes using `ps` command. Compare the results with results returned by `ps -x` and `ps -ax`. Check `-l` and `-u` switches. Run `tty` command in different terminal emulators and in different text-based consoles. Compare the values with column TTY of `ps` command output.
2. Observe process inheritance by running `ps -l` and by checking columns: PID (*process identifier*) and PPID (*parent process identifier*). Find the top-level process in the inheritance tree. Try `pstree` command.
3. Observe the list of processes using interactive `top` command. Try the following commands:

   P sort according to processor usage,
   M sort according to memory allocation,
   u list only processes of a given user,
   1 display information about every core (including *hyper-threading*),
   q quit.

   Try also other variants of `top`: `atop` and `htop`.
4. Check the *load* of the system by running `uptime`. Run `xload` program and start in several consoles the following infinite loop:

```
# while true; do : ; done
```

5. By means of `pgrep` command find identifiers of all your shells, and all processes of `root` user.

4.2 Signals

Signals are used for communicating processes. They are short messages sent to processes for informing about a given situation. Usually signals are either ignored by process, or cause them to stop running. Signals have numerical values (starting with 1), and do not allow to convey any additional information within the message.

1. Check the list of available signals by reading signal(7) manual page.
2. Run the `sleep` command:

   ```
   # sleep 100
   ```

   Check its PID, and send different signals to it using the `kill` command. Try the following signals: HUP(1), INT(2), TERM(15), QUIT(3), KILL(9).

   ```
   # kill -2 12345
   # kill -INT 12345
   ```

   Try to send the same signals to `vi`.
3. Try using commands: `killall` and `pkill` .

4.3 Process priorities

1. Decrease priority of a given process by *increasing* its *nice* value:

   ```
   # renice +5 12345
   ```

   The *nice* value can be observed using `ps -l` command. Nice ranges from -20 (the highest priority) to 19 (the lowest priority) with default value of 0. Regular users can only increase nice value of their processes[1].
2. Run an infinite loop in the terminal and observe the CPU load using `top`:

   ```
   # while true; do : ; done
   ```

   a) How does the nice value of the interpreter influence CPU load?
   b) Run $n + 1$ infinite loops in separate consoles, where n is the number of cores of the CPU. Increase the *nice* value for one of the interpreters and watch the CPU usage of the processes. Please pay attention to the load of the system.

[1] That's why the value is called "nice": you are nice to the other users of the system by giving them more processing power

3. Check how the *nice* value of the interpreter influences priorities of child processes.
4. Run a new process with modified *nice* value:

```
# nice -10 sleep 100
```

4.4 Handling of many processes in the interpreter

1. Suspend editing session of `vi` editor by pressing **Ctrl–z** and resume it by running command `fg`. Suspend it once again and run another instance of the editor. List current sessions using `jobs` command. Resume the first session using `fg`:

```
# fg %1
```

   Suspend the editor and resume the other instance. This way you can switch between two interactive applications.

2. Run `sleep` command, suspend it, and resume to work in background:

```
# sleep 100
^Z
[1]+  Stopped                 sleep 100
# bg
[1]+ sleep 100 &
```

3. Run a new instance of `sleep` in background:

```
# sleep 200 &
# ps -x
```

   Change the running mode of the `sleep` process to foreground mode using `fg`.

4. Suspend the `sleep` command by means of sending SIGSTOP signal. Next, resume it by sending SIGCONT.
5. Start the `screen` command for multiplexing the console:

```
# screen
```

   Start `vi` editor. Next, create a new "window" by pressing **Ctrl–a c**, and run `mc` inside. Switch between virtual windows by pressing **Ctrl–a n** (next) or **Ctrl–a p** (previous). All available windows can be listed by pressing **Ctrl–a "** or **Ctrl–a w**. You can also switch between windows using their identifiers: **Ctrl–a 0** ... **Ctrl–a 9**. Disconnect from the session by pressing **Ctrl–a DD**. Reconnect to the session:

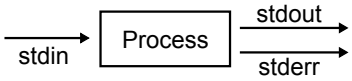
```
# screen -d -R
```

   Leave a running process (e.g. `vi`) on the server this way, and then try to reconnect to it.

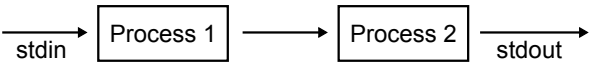
5
Pipes

Processes are isolated in modern operating systems. Their address spaces are disjoint. In order to communicate processes you have to use dedicated system mechanisms. The simplest form of communication was introduced in the previous chapter, i.e. signals. This chapter discusses pipes, which are used for passing streams of data from one process to another.

Every process has its own standard streams: standard input (*stdin*), standard output (*stdout*), and standard error (*stderr* – a stream for diagnostic messages):



The streams are by default connected to keyboard, and the terminal. However, we can redirect the streams and read/write data from files and/or from other processes. The redirections are achieved by using <, >, and | special characters in interpreters. Programs that are supposed to read from standard input, do some processing, and write to standard output, are called *filters*. We can connect the output of a one process with input of another one:



This way we create a *pipe*. This chapter presents a few filter programs, and possible combinations of them using pipes.

5.1 Filter cat

1. Run `cat` without any arguments and type a few lines of input:

```
# cat
Ala ma
Ala ma
kota
kota
^D
```

2. Redirect standard output to a file:

```
# cat > a.txt
Ala ma
kota
^D
```

3. Display contents of a file:

```
# cat a.txt
# cat < a.txt
# cat - a.txt
```

   Filters invoked with an argument do not read from the standard input, but from the given file instead. You can explicitly request reading from standard input using the name "`-`".

4. What will happen after running the following command?

```
# cat < a.txt > b.txt
```

5. What are the effects of the following command[1]?

```
# cat a.txt b.txt > c.txt
```

6. Try appending to a file:

```
# cat a.txt >> b.txt
```

7. Try providing contents of a new file inline:

```
# cat << EOF
> Ala ma
> kota
> EOF
Ala ma
kota
```

   The text "EOF" serves as an indicator of the end of data. It can be any text.

8. Display the contents of a file with numbered lines:

[1] The results justify the name of the `cat` program which comes from *concatenating*.

```
# cat -n a.txt
```

9. Check other variants of `cat`: `dog` and `tac`.

5.2 Filters head and tail

The `head` filter display initial lines from the stream, while the `tail` filter displays the last lines.

1. Display initial lines of a file:

   ```
   # head a.txt
   # head < a.txt
   # cat a.txt | head
   ```

2. Display lines from 3 to 5 from file `/etc/passwd` by combining `head` and `tail`. Display the second to the last line from the file.

   Hint: the results are contained in the first 5 lines of the file.

3. Write to `head.lines` first lines from all `.txt` files from the current directory.
4. Display all but the first and last line from `a.txt`.
5. Try monitoring the contents of `a.txt` by running in one terminal:

   ```
   # tail -f a.txt
   ```

   and by appending something to the file in the other window:

   ```
   # echo "Ala ma kota" >> a.txt
   # ps x >> a.txt
   ```

5.3 Filter grep

The `grep` filter selects from the input stream lines containing a given pattern. The pattern is represented by *regular expressions* (RE), which are similar to glob patterns used by the interpreter but use different syntax. In short:

`abc` regular characters represent just the text (e.g. “abc” in this case),

`^a` lines starting with “a”,

`a$` lines ending with “a”,

`.` any single character,

`[a-z]` single character from the set, in this case a lowercase letter,

`a*` asterisks is an *operator*, denoting a repetition of the pattern on the left, it can be repeated 0 or more times, therefore an empty string also matches,

`.*` any sequence of characters including an empty sequence.

While passing REs to applications remember to use quotes, because REs use many characters which are special ones for the interpreter.

1. Display list of sub-directories of the current directory. Display list of symbolic links from `/usr/bin`.
2. Display information about regular files from the current directory that have the write permission for group.
3. Display lines from `a.txt` *not containing* any numbers.
4. Display all lines from `a.txt` excluding comments, e.g. lines starting with "#".
5. Display names of text files from the current directory that contain (do not contain) polish letters "ą", "ć" or "ę".
6. Search your home directory for files containing word "abc".
7. Search your home directory for files with `.txt` extension that contain word "abc".
8. Copy all files with `.txt` extension from you account that contain word "abc" to `/tmp` directory.
9. Read the documentation of other variants of `grep`: `egrep` and `fgrep`.

Exercises 7 and 8 require some additional knowledge. Every process, while exiting, returns to the system the so called *exit status*, which is a 1 byte value ranging from 0 to 255. Value 0 represents successful completion. All other values denote some errors. Usually processes return 1 in case of errors:

```
# touch a.txt
# rm a.txt
# echo $?
0
# rm a.txt
rm: cannot remove 'a.txt': No such file or directory
# echo $?
1
```

In this example we have referred to a special variable `$?` representing the exit status of the last command. Exit status is returned by all processes, including `grep`: it returns 0 if matching lines were found, and 1 otherwise:

```
# echo "X Y" > a.txt
# grep X a.txt
X Y
# echo $?
0
# grep Z a.txt
```

```
# echo $?
1
```

We can use the exit status as a condition for `find` command. For example:

```
# find . -name "*.txt" -exec grep abc {} \; -print
```

In this example `find` will search for `.txt` files, and it will run `grep` for every file found. Whenever `grep` returns 0, the whole `-exec` condition is accepted, and the processing continues. As a result, in this example `find` will print names of `.txt` files that contain "abc". Modify the command to make the results more clear by withholding printing of lines containing "abc".

5.4 Filter wc

The `wc` filter (*word count*) counts all lines, words and characters in the input stream.

1. Count all processes in the system.
2. Count symbolic links in `/usr/bin`.
3. Count all regular files in `/etc` and its subdirectories. Do it in two ways: starting from `ls` and `find`.
4. Count the length of the third line in `/etc/passwd`.
5. Count empty lines in `/etc/profile`.
6. Count total number of lines of your `.txt` files in:
   a) the current directory,
   b) in all subdirectories of your home directory.

5.5 Filter tr

The `tr` filter (*translate*) substitutes characters from the first argument (string) with respective characters from the second argument.

1. Substitute all lowercase letters to capital letters:

   ```
   # ls -l | tr a-z A-Z
   ```

2. Encrypt the data using ROT13 algorithm:

   ```
   # cat a.txt | tr a-z n-za-m
   ```

   What happens when you filter the data through `tr` twice?

3. Display the contents of `a.txt` in a single line:

```
# cat a.txt | tr  '\n'  ' '
```

4. Remove repetitive spaces from input data:

   ```
   # ls -l | tr -s ' '
   ```

   Next, additionally substitute spaces with tabs:

   ```
   # ls -l | tr -s ' ' '\t'
   ```

5. Extract all words from `a.txt` by putting them in separate lines. Additionally remove all punctuation marks:

   ```
   # cat a.txt | tr -d '.,:;!?'
   ```

6. Convert a DOS/Windows text file (using CR LF characters for representating new lines) to Unix text file (using LF characters). Example DOS file can be created using `vi` editor by setting additional option before writing:

   ```
   :set fileformat=dos
   ```

5.6 Filter cut

The `cut` filter selects columns from input data. By default the columns are separated by tabs.

1. Display access right to files in the current directory:

   ```
   # ls -l | cut -d ' ' -f 1
   ```

2. Display names and descriptions of users from `/etc/passwd` (fields 1 and 5).
3. Display sizes of files from the current directory.
4. Display sizes of files from the current directory preserving alignment to the right, e.g.:

   ```
   # ls -l | cut ...
     1021
     2534
      326
    83017
   154203
     2061
   ```

5.7 Filter sort

The `sort` filter just sorts the input data. It does not change the input file, because `sort` is a filter, therefore the results are passed to the standard output.

1. How to permanently change the contents of `a.txt` so that it is sorted? Consider using a temporary file.
2. Try sorting a file containing numbers, e.g.:

   ```
   50
   9
   2000
   100
   ```

3. Display list of files in the current directory sorted by file sizes. Next, display only names of the files in the same order.
4. Display list of user names from `/etc/passwd` sorted by user identifiers (field 3) in descending order.

5.8 Filter uniq

The `uniq` filter is used to eliminate repeated lines from input data.

1. Pass the following data to `uniq` (interactively or from a file):

   ```
   a
   b
   b
   a
   ```

2. Display list of users having at least 1 process in the system.
3. Display list of active users in the system along with corresponding number of processes they own, i.e.:

   ```
   # ps -axu | ...
   postfix 7
   root 80
   voytek 20
   ```

4. Display list of users having at least 2 processes in the system.
5. Display list of at most 3 users having the largest number of processes in the system.
6. Display 3 most frequently used words from `a.txt`.
7. Display in alphabetical order list of 3 largest files from `/usr/bin`.
8. Display names of users being the owners of 10 processes allocating the largest amounts of RAM in the system.
9. Starting with the `history` command, display a statistics of most frequently used commands.

6 The Shell

6.1 Environment variables

1. Set, get, use, and remove an environment variable:

```
# x=abc
# echo $x
abc
# x=out.txt
# cat $x
# unset x
```

2. Test different methods of referencing variables:

```
# x=5
# echo Ala ma $x kotów
# echo "Ala ma $x kotów"
# echo 'Ala ma $x kotów'
# echo "Ala ma $xx3 kotów"
# echo "Ala ma ${x}x3 kotów"
```

   Quotes are necessary when variables contain special shell characters. Compare:

```
# echo "*"
# echo *
```

3. Check availability of environment variables in child processes:

```
# x=10
```

```
# echo $x
10
# sh
sh# echo $x

sh# exit
# export x
# sh
sh# echo $x
10
sh# exit
```

4. List all exported environment variables using `env`. Read and modify the most popular ones:
   a) HOME
      ```
      # HOME=/tmp
      # cd
      # pwd
      ```
   b) PS1
      ```
      # PS1="C:\> "
      C:\> ls
      ```
   c) TERM – terminal configuration. Set it to various standard values, like `vt100`, `ansi`, `dumb`, and try to use full-screen programs like `vi`, `mc`.
   d) PATH
      ```
      # PATH=/tmp
      # ls
      bash: ls: No such file or directory
      # PATH=/bin:/usr/bin
      ```
      Check whether the current directory `.` is included in the list of directories enumerated in PATH.
   e) EDITOR – the default editor.
   f) PAGER – the default pager (e.g. `more` or `less`).
5. Modifications of environment variables are not permanent. In order to set them automatically modify appropriate configuration files for your interpreter. Test this mechanism for Bash interpreter by appending to `.bashrc` the following line:

   ```
   echo "This is .bashrc"
   ```

   and to `.bash_profile`:

   ```
   echo "This is .bash_profile"
   ```

   Then log into the system in text mode and observe the messages. Set an exported variable inside one of these files:

```
export X="Ala ma kota"
```

You can load dynamically settings from a file by running:

```
# . .bashrc
# source .bashrc
```

6. Try capturing the output of a command:

```
  sh# x=‘hostname -f‘
bash# x=$(hostname -f)
bash# x=$(grep -l abc $(find . -type f -name "*.txt"))
```

The first example uses reverse apostrophe, usually available on keyboards on the left of the `1` key.

6.2 Various shell mechanisms

6.2.1 Grouping of commands

1. Run a sequence of commands:

```
# echo "Start"; sleep 5; echo "End"
```

2. Test conditional execution of commands:

```
# grep -q xyz a.txt && echo "Found"
# grep -q xyz a.txt || echo "Not found"
# grep -q xyz a.txt && echo "Found" || echo "Not found"
```

3. Try grouping of commands:

```
# (grep -q xyz a.txt || grep -q xyz b.txt) && echo Jest
# { grep -q xyz a.txt || grep -q xyz b.txt; } && echo Jest
```

Commands in () brackets are run in a new shell. Compare:

```
# (cd /usr; pwd); pwd
# { cd /usr; pwd; }; pwd
```

Run `xload` command with 5 seconds delay without blocking the current shell.

6.2.2 Redirection of streams

1. Every process has its own standard streams as it was mentioned in Chapter 5. Both output streams can be redirected independently: standard output (1) using >, and standard error (2) using 2>. Run the following commands and watch the output on the terminal and in the output files:

```
# find /etc -name mtab
```

```
# find /etc -name mtab > out.txt
# find /etc -name mtab 2> out.txt
# find /etc -name mtab 1> out.txt
# find /etc -name mtab > out1.txt 2> out2.txt
# find /etc -name mtab > out.txt 2> out.txt
# find /etc -name mtab > out.txt 2>&1
# find /etc -name mtab >& out.txt
# (find /etc -name mtab 2>&1) > out.txt
```

In order to ignore a stream use redirection to a special file `/dev/null` which is a kind of infinite data absorber:

```
# find /etc -name mtab 2> /dev/null
```

2. Try stream redirections in case of pipes:

```
# find /etc -name "a*.conf" | head -n 2
# find /etc -name "a*.conf" | head -n 2 > out
# find /etc -name "a*.conf" 2> /dev/null | head -n 2
# find /etc -name "a*.conf" 2>&1 | head -n 2
# find /etc -name "a*.conf" |& head -n 2
# find /etc -name "a*.conf" 2>&1 > /dev/null | head -n 2
```

6.2.3 Calculations

1. Try `expr` command:

```
# expr 2 + 2
4
# expr 3 \* 5
15
# expr \( 2 + 3 \) \* 5
25
```

2. Bash interpreter has introduced a few useful improvements in the context of calculations:

```
# echo $((2+2))
# x=3
# echo $(($x*5))
# x=$(( ($x+1)*3 ))
# ((x=$x+1))
```

(*) 3. Check out the dedicated `bc` interpreter for calculations:

```
# bc
2+5
7
2.3/(0.779+0.123)
2
```

```
scale=3
2.3/(0.779+0.123)
2.549
x=24
y=36
sqrt(x^2+y^2)
43.26
scale=50
sqrt(2)
1.41421356237309504880168872420969807856967187537694
quit
# man bc
```

6.2.4 Aliases

Aliases are useful shortcuts for running commands with default switches and arguments.

1. Define an alias:

   ```
   # alias l="ls -lF"
   # l
   # l /usr
   ```

2. You can check the type of a command by running:

   ```
   # type l
   l is aliased to `ls -lF'
   ```

3. Remove an alias using `unalias`.

6.3 Basics of scripting

1. Create a file `test.sh` with the following contents:

   ```
   echo "Hello world"
   ```

   and run it using shell explicitly:

   ```
   # sh test.sh
   Hello world
   ```

   Try to run it as a regular program:

   ```
   # ./test.sh
   ```

   Add the execute permission and try again.

2. Add to the script a special comment in *the first* line (comments generally start with a "#" character):

   ```
   #!/bin/sh
   ```

   When the first character of the comment is "!", then the rest of the first line is treated as a path to the interpreter for the script.

3. Add to the script lines displaying its arguments:

   ```
   echo "Arg 1 = $1"
   echo "Arg 2 = $2"
   echo "Arg 3 = $3"
   ```

   and run the script with different arguments:

   ```
   # ./test.sh a b c
   # ./test.sh "a b" c
   # ./test.sh a\ b c
   # ./test.sh *
   # ./test.sh "*"
   # ./test.sh \*
   ```

   What does the `$0` variable contain?

4. Use the `shift` command:

   ```
   echo "$1 $2 $3"
   shift
   echo "$1 $2 $3"
   ```

5. In Bash you can use simple notation to access positional variables above `$9`:

   ```
   #!/bin/bash
   echo "Arg 10 = ${10}"
   ```

   Please note the modified comment in the first line!

6.4 Conditional statement

1. Check the `test` program:

   ```
   # test ab = ab
   # echo $?
   0
   # test ab = cd
   # echo $?
   1
   ```

   Check numerical operators: `-eq` (=), `-ne` (≠), `-gt` (>), `-ge` (≥), `-lt` (<), `-le` (≤), e.g.:

```
# test 5 -gt 3
```

Check file system operators: `-f` (regular file), `-d` (directory), `-L` (symbolic link), `-r` (read access), `-w` (write access), `-x` (execute permission), e.g.:

```
# test -d dir1
```

Try negating the condition:

```
# test ! -f a.txt
```

and defining compound conditions (`-a` denotes AND, `-o` denotes OR):

```
# test -f a.txt -a -w a.txt
```

2. Try the following example script testing availability of an argument:

```
if test -n $1
then
  echo "Arg 1 = $1"
else
  echo "Argument missing."
fi
```

Notes:

a) Pay attention to line breaks!
b) What is the problem with this implementation? Try to fix it.
c) In general you can use *any* command as a testing program, not just `test`.

3. Substitute the `test` program with `[` (available as `/usr/bin/[`):

```
if [ -n $1 ]
...
```

It does the same but requires an additional argument: `]`. The resulting syntax looks like we are just surrounding the condition with square brackets.

4. Write a simple script appending a text passed as the second argument to the file identified by the first argument. The script should inform about errors by printing adequate messages:

a) wrong number of arguments,
b) missing file identified by the first argument,
c) missing write permissions to the file.

In case of errors the script should return exit status 1 (using `exit` command).

(*) 5. Bash can handle extended notation of conditions using double square brackets. The condition besides the standard tests can use regular expressions, e.g.:

```
if [[ "$1" =~ ^[0-9]{2}-[0-9]{3}$ ]]
```

6.5 Loops

6.5.1 The `for` loop

1. Try the following example of loop:

   ```
   for x in a b c
   do
     echo $x
   done
   ```

   Control variable `x` gets value from the list following the `in` statement.

2. Try iterating over a list of files matching a given glob pattern:

   ```
   for f in *.txt
   ```

3. Try numerical iterations:

   ```
   for x in `seq 1 10`
   ```

   Bash introduces another useful statement for generating lists of numerical values:

   ```
   for x in {1..10}
   for x in {120..100..2}
   ```

   Propose an iteration from 0 to 20, and then back to 0.

4. Write a script that will iterate over the contents of a directory pointed by an argument, and display only subdirectories from that directory.

6.5.2 The `while` loop

1. General syntax for the `while` loop is the following:

   ```
   while condition
   do
     commands
   done
   ```

   where the condition uses the same syntax as the `if` statement.

2. Write a script that displays all arguments passed to it, regardless of their number.

3. Write a script that appends a text given by the first argument to all files with extensions given by the following arguments.

Both loops can be interrupted by issuing the `break` command. The next iteration can be started from within the loop by issuing `continue`.

6.6 Functions

Functions are a convenient tool for structuring the code. Basically functions are just scripts inside scripts.

1. Define a function and call it:

```
function foo() {
  echo "Hello"
}

foo
```

2. Try passing arguments to a function:

```
function foo() {
  echo "Hello $1"
}

foo Voytek
```

3. Returning from a function:

```
function foo() {
  echo "Start"
  return
  echo "End"
}
```

The return statement can be supplemented with a number representing the exit status.

6.7 Input processing

1. Check the `read` command for reading from standard input:

```
# read x
Ala ma kota
# echo $x
Ala ma kota
# read x y z
Ala ma kota
# echo $x
Ala
```

Arguments of the command are names of variables that will be initiated with subsequent words from the input. The last variable will hold all remaining text.

2. Try processing of input data containing lists of natural numbers, and print the sums of the numbers. The structure of the main loop is the following:

```
while read LINE
do
  ... # calculate and print the sum
done
```

3. Check out the `dialog` program for implementing full-screen text-based user interface, e.g.:

```
# dialog  --msgbox "Ala ma kota" 10 40
```

There are quite a few other standard dialog windows for interacting the user:

```
# dialog --help
```

7 Unix System Call Interface

7.1 Using C programming language compiler

1. Create a text file `hello.c` with the following code:

```
#include <stdio.h>

void main()
{
  printf("Hello world!\n");
}
```

2. Compile the program using C compiler:

```
# cc hello.c
```

   By default the compiler creates `a.out` file with the binary code in case of successful compilation. You can specify another name using `-o` switch:

```
# cc -o hello hello.c
```

3. Run the program:

```
# ./hello
Hello world!
```

4. Turn on printing of all warnings during compilation:

```
# cc -Wall -o hello hello.c
hello.c:3:6: warning: return type of 'main' is not 'int' [-Wmain]
```

```
void main()
     ^~~~
```

Fix the problem in `hello.c` and compile it once again.

Write the code in such a way, that it does not generate any warnings during compilation.

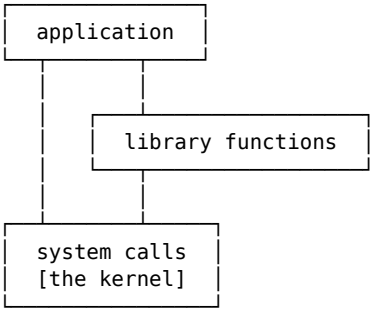
7.2 System manual

1. Check out the manual page `printf(3)`:

   ```
   # man 3 printf
   ```

   Please note the SYNOPSIS section. It contains a list of all header files necessary to successful compilation of a program using this function.

2. The remaining exercises in this workbook consist in developing small programs using *system calls* of the operating system. System calls are documented in Section 2 of the system manual. We will be using also so called *library functions*, which are wrapping system calls to make them more useful and easy to use.



   Let us consider the `write` system call and the `printf` library function. Both functions can be used for printing messages on the screen. However, the `printf` function is more convenient, because it can format the message (e.g. numbers), and handles buffering. Internally the `printf` function invokes `write` for actual passing the message text to the kernel.

7.3 Passing arguments

Arguments passed to programs at the command line are visible inside the programs as arguments of the `main()` function. The first argument is of `int` type, and represents the

number of arguments, including the name of the program itself. The second argument is an array (type `char* []`), and it holds strings representing successive arguments.

```
int main(int argc, char* argv[])
{
  int i;
  for(i=0; i<argc; i++)
  {
    printf("argument %d: %s\n", i, argv[i]);
  }
}
```

For simple conversion of integer numbers one can use `atoi()` function.

7.4 Error handling

Every system call returns an integer value representing the processing status. In general, value `-1` denotes an error. Detailed type of error can be determined by reading the value of global variable `errno` of type `int`. Descriptive message can be printed using library function `perror()`, which reads the `errno` variable and prints appropriate message on standard error stream. It is a good practice to check return value of all system calls.

1. The following example is a typical piece of code for testing return values of system calls. This example issues the `open()` system call (discussed in the next chapter), which is used for opening files.

   ```
   fd = open("example.file", O_RDONLY);
   if (fd == -1)
   {
     printf("errno: %d\n", errno);
     perror("Opening a file");
     exit(1);
   }
   ```

   Try to run the program with an argument representing the name of the file to be opened. Try opening: existing file, non-existing file, a file with no read permission (e.g. `/etc/shadow`).

2. Change the default language of messages before opening a file:

   ```
   setlocale(LC_ALL, "pl_PL.UTF-8");
   ```

8 Processing Files

8.1 Basic functions

1. Files can be opened using the `open()` system call. The function should be called with 2 or 3 arguments. The first argument is the path of the file being opened. The second argument is the mode of opening the file. The third (optional) argument represents access rights of the file being created. The second arguments can be specified by combining (bitwise sum) different, predefined flags:

   `O_RDONLY` opening for reading,
   `O_WRONLY` opening for writing,
   `O_RDWR` opening for reading and writing,
   `O_CREAT` creating a file when it is not existing.

   The following is an example of file creation:

   ```
   fd = open("example.file", O_WRONLY | O_CREAT, 0644);
   ```

   Initial access rights are represented by an octal number[1], therefore 0644 is equivalent to `rw-r--r--`.

   The `open()` system call returns a so-called *file descriptor*. It is a natural number which is an index in the table of opened files maintained by the system for every process. The table is inaccessible directly, but can be manipulated by a few sys-

[1] In C programming language a number starting with 0 denotes an octal number.

tem calls. By default, the first 3 positions in the table are occupied by standard streams, as in the following example:

0	`stdin`	standard input
1	`stdout`	standard output
2	`stderr`	standard error
3	`example.file`	regular file opened by the process
4		
...	...	

In a typical situation the return value of the first invocation of `open()` is 3, because this the index of the first free position in the table of opened files.

1. All files opened by a process are closed at the end of its execution. However, it is a good practice to close all files that are not used any more, because the capacity of the table of opened files is limited:

   ```
   close(fd);
   ```

   The only argument of `close()` function is the file descriptor. This function is a rare case when we can ignore the return value.

2. Writing to a file:

   ```
   write(fd, "Hello", 5);
   ```

   The second argument of `write()` system call is an address in memory where the data to be written start. In the example above it is an address of static string in memory. The third argument specifies the number of bytes to be written. The function returns the number of bytes that were successfully written. It can be less than the third argument. As usual: `-1` denotes an error.

   Write a simple program that creates a file with a simple contents.

3. Reading from a file:

   ```
   char buf[20];
   ...
   n = read(fd, buf, 20);
   ```

   The `read()` system call has analogous arguments as `write()`. The address in memory has to be modifiable. The third argument denotes the requested number of bytes to be read. The actual bytes that will be read can be smaller. A special case is 0, which means that there is nothing more to be read, which is usually described as *end of file* (EOF).

   Write a simple program that reads a string from a file and prints it on the screen.

4. Write a program for displaying the contents of an arbitrary large file (like `cat`). The processing has to be done in a loop, by reading and printing small portions of the file.

```
while((n=read(fd, buf, 20)) > 0)
{
  write(1, buf, n);
}
```

This example writes on the screen using `write()` system call. The file descriptor is set to predefined value 1, which represents the standard output.

What is the optimal size of the chunk of data to be processed at a time. Consider extreme values, like 1 B and 1 GiB.

5. Write a program appending a string given by the second argument to a file identified by the first argument. You can use an additional flag `O_APPEND` for opening files in the appending mode:

   ```
   fd = open("example.file", O_WRONLY | O_APPEND);
   ```

6. Remove an existing file:

   ```
   fd = unlink("example.file");
   ```

   As you can see, removing a file does not require prior opening.

7. Open an existing file, and remove all its contents by setting its size to 0 by means of `ftruncate()`.

   ```
   fd = open("przyklad.txt", O_WRONLY);
   ftruncate(fd, 0);
   ```

   Remember to check the return values of system calls!

8. The current position during processing a file is indicated by a *file pointer*. By default we start at the beginning of the file (unless you use the `O_APPEND` flag). Every read or write operation moves file pointer by the number of bytes read or written. It is possible to change the current position of the file pointer by means of `lseek()` system call:

   ```
   int lseek(int fd, int offset, int whence);
   ```

   The `offset` argument denotes the new position, while `whence` denotes the reference: `SEEK_SET` – the beginning of the file, `SEEK_CUR` – the current position, `SEEK_END` – the end of the file. Successful call returns the new position counted from the beginning of the file. The following example invocation:

   ```
   lseek(fd, -128, SEEK_CUR);
   ```

   moves the file pointer back by 128 bytes.

   Write a program that will check and print the number of bytes of a given file.

8.2 Implementation of simple Unix tools

Discussed so far system calls are sufficient to implement some basic Unix tools. However, please consider performance issues.

1. Implement a program checking whether two files given by arguments are identical like the standard `cmp` command.
2. Implement a program similar to the standard `tee` tool which copies data from standard input to standard output and to a file. The program should handle the standard `-a` switch. Example usage of the tool:

   ```
   # ps ax | ./mytee -a wynik.log
   ```

3. Implement a program calculating the number of characters, words and lines in a file like the standard `wc` command.
4. Implement a program displaying lines of a text file given by the second argument starting with the text given by the first argument. Example usage of the tool:

   ```
   # ./mygrep abc a.txt
   ```

   should produce results identical to:

   ```
   # grep ^abc a.txt
   ```

5. Implement a program displaying the contents of a text file in reverse order, like the `tac` command.
6. Implement a program joining 2 sorted text files given by arguments into a single sorted output. For example, for the following input files:

   File a.txt:
   ```
   abc
   envelope
   zero
   ```

   File b.txt:
   ```
   b
   cigarette
   rover
   ```

   the output should be:

   ```
   abc
   b
   cigarette
   envelope
   rover
   zero
   ```

9 Processes

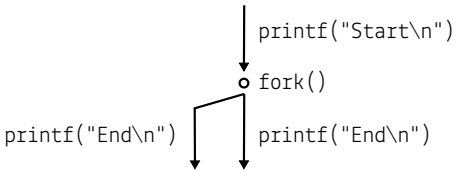
9.1 The fork function

The `fork()` function is used for creating new processes. New processes in Unix operating systems are clones of the processes invoking `fork()`. The new process starts the execution with the first instruction following `fork()` invocation. Both processes run concurrently with no additional coordination.

1. Run the following piece of code and observe the results:

```
printf("Start\n");
fork();
printf("End\n");
```

The execution can be illustrated as follows:



The first message is printed in one process, and the second message is printed by two processes (the parent and the child).

What happens when we remove the new line characters ("\n") from printf() messages?

2. Add sleep() invocation right after fork(), and run the program in background. Check the list of processes using ps -l, and watch the parent process identifiers (PPID).

```
# ./a.out &
# ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 R  1011  2567  2566  0  75   0 -  1233 -      pts/2    00:00:00 bash
0 S  1011  2589  2567  0  75   0 -   337 schedu pts/2    00:00:00 a.out
0 S  1011  2590  2589  0  75   0 -   337 schedu pts/2    00:00:00 a.out
1 R  1011  2591  2567  0  75   0 -   905 -      pts/2    00:00:00 ps
```

3. Check the return value of fork(). Print identifiers of the processes (PID) and its parents (PPID), after getting them by means of getpid() and getppid() system calls:

```
int x;
x = fork();
printf("Ret val of fork=%d  PID=%d  PPID=%d\n", x, getpid(), getppid());
```

4. How many processes will be running after executing the following code:

```
fork();
fork();
```

And what about this example:

```
if (fork()==0) {
  fork();
}
```

9.2 The exec function

The exec() function is used to load new code from a file, and *substitute* the currently running code with the new one. The process is reinitialized but retains its PID, table of opened files, and environment variables. The new code starts its execution with the very first instruction. There is no return from the function in case of successful invocation, because the original code is overwritten. There is a whole family of exec() functions with different lists of arguments:

execl() arguments for the program are passed as arguments of the function,

execv() arguments for the program are passed as an array,

`execlp()` like `execl()`, but additionally the program is searched in the directories listed in the PATH environment variable,

`execvp()` like `execv()`+ path lookup,

`execle()` like `execl()`, but with additional argument representing the new list of environment variables for the process,

`execvpe()` like `execvp()` + new environment.

1. Run the following example, observe the results, and compare it with invocation of `fork()`:

   ```
   printf("Start\n");
   execl("/bin/ls", "ls", "-l", NULL);
   printf("End\n");
   ```

   The first argument of `execl()` is the path pointing to a file with the binary code to load. Next arguments of the function represent arguments for the program: its name, argument 1, 2, 3, ... etc. The list must be ended with an empty pointer NULL. Alternatively you can use `execlp()`:

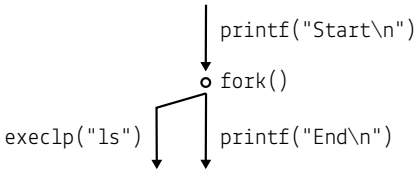
   ```
   execlp("ls", "ls", "-l", NULL);
   ```

   Please note the first two arguments!

2. Whenever you want to execute an external program and continue processing, you have to start a new process:

   ```
   printf("Start\n");
   if (fork()==0)
   {
     /* child process */
     execlp("ls", "ls", "-l", NULL);
     exit(1);
   }
   printf("End\n");
   ```

   Run the program several times and observe the order of messages. The execution can be illustrated as follows:

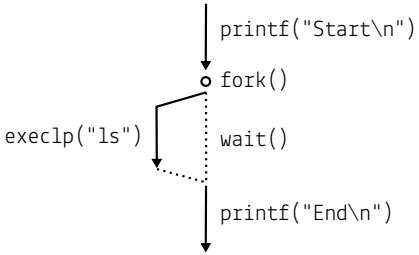
   

   What is the reasoning for `exit(1)`?

3. Add to the parent process invocation of `wait()` system call which waits for completion of a child process. This introduces a simple form of synchronization of the processes:

```
if (fork()==0)
{
  /* child process */
  ...
}
else
{
  /* parent process */
  wait(NULL);
  printf("End\n");
}
```

The execution can be illustrated as follows:



4. The `wait()` function has one more use: it can read the exit status of a child process:

```
if (fork()==0)
{
  /* child process */
  exit(5);
}
else
{
  /* parent process */
  int stat;
  wait(&stat);
  printf("status=%d\n", stat>>8);
}
```

The exit code of a normally terminated process is available in the second byte of the integer value. For this reason the variable was shifted 8 bits to the right (the `>>` operator).

5. When a process ends it returns its exit status code, which has to be read by the parent process (by means of `wait()`). If the parent process has already ended, then the `init` process (PID=1) takes care of it. However, when the parent process is running and it does not invoke `wait()`, the child process that has ended becomes a *zombie* process. You can observe such process by adding a `sleep()` invocation to the parent process and running `ps` command.
6. Write a program using `execvp()` function that will execute a program with all given arguments and measure its execution time. Example usage:

   ```
   # ./run find /usr -type f -size +100k -mtime -30
   ...
   Exec time: 0.537s
   ```

   For getting precise time you can use `gettimeofday()` system call.

9.3 Redirections of standard streams

The distinction between process creation and running a new code makes it possible to do additional processing between these actions. As an example we can consider manipulation of table of open files, which is preserved during invocation of both `fork()` and `exec()`.

1. Write the output of `ps ax` to a file:

   ```
   int fd;
   close(1);
   fd = open("out.txt", O_WRONLY | O_CREAT, 0644);
   execlp("ps", "ps", "ax", NULL);
   ```

   The `open()` system call opens a file by allocating the first free position in the table of opened files. In this case it will use descriptor 1, because it was just freed by `close(1)`. The `ps` command writes its output to descriptor 1, and does not realize that it was now substituted with a descriptor of an newly opened file. As a result, the whole output is written to the `out.txt` file.
2. The table of opened files can be manipulated by functions `dup()`, `dup2()`, or `dup3()`, which duplicate a given descriptor:

   ```
   int fd;
   fd = open("out.txt", O_WRONLY | O_CREAT, 0644);
   dup2(fd, 1);
   ```

3. Write a program that will start processing equivalent to the following shell invocation:

   ```
   # grep xy < dane.txt >> wynik1.txt 2> wynik2.txt
   ```

10 Programming Pipes

We have introduced unidirectional pipes in Chapter 5. In this chapter we will try to achieve similar results in a programmatic way. There are two types of pipes: unnamed and named pipes. The shell provides easy access to unnamed pipes using the "|" character. Read more on pipes on pipe(7) manual page.

10.1 Unnamed pipes

1. An unnamed pipe can be created by means of `pipe()` system call. It takes an array of 2 integer values as an input, and fills the array with two descriptors representing the pipe. The descriptor at index 0 is supposed to be read from, and the descriptor at index 1 is supposed to be written to. Try to create a pipe, and access it in a single process:

   ```
   int fd[2];

   pipe(fd);
   write(fd[1], "Hello", 6);
   read(fd[0], buf, 6);
   printf("%s\n", buf);
   ```

2. When you create a new process, the table of open files remains untouched. Therefore, if you create a pipe, and then start a new process, both processes will

have access to the pipe. Modify the previous example, by moving the writing invocation to the child process:

```
pipe(fd);
if (fork()==0)
{
  write(...);
  exit(0);
}
read(...);
```

Add a `sleep()` invocation before `write()`, and observe the 2 processes and their synchronization. Unlike files, some I/O operations on pipes may be blocking:

a) a process reading from an empty pipe will be suspended till the data is available,

b) a process writing to a full pipe will be suspended till the buffer is freed.

The default size of the pipe buffer may vary depending on the system. Usually it has at least 4 KiB, but newer Linux systems use a buffer as large as 16 memory pages, which gives us 64 KiB.

3. Write a program that will start processing equivalent to shell invocation: `ls | tr a-z A-Z`:

```
pipe(fd);
if (fork()==0)
{
  dup2(fd[1], 1);
  execlp("ls", "ls", NULL);
}
else {
  dup2(fd[0], 0);
  execlp("tr", "tr", "a-z", "A-Z", NULL);
}
```

Complete the code so that the program will end.

4. Modify the previous example so that the output will be written to `out.txt` file, like in shell invocation:

```
# ls | tr a-z A-Z > out.txt
```

5. Write a program executing:

```
# ls -l /tmp | sort -n -k 5,5 | tail -5
```

10.2 Named pipes

The only way to communicate 2 processes by means of unnamed pipe is by creating the pipe in a common ancestor, e.g. one process is parent of the other or both have a common parent. In order to communicate any two unrelated processes one has to use *named pipes*, which work in the same way but are initiated differently. Named pipes are objects visible in the file system as special files. One can use `mkfifo()` library function to create such a special file. There is also a command `mkfifo` which does the same. The named pipe has to be created only once, and can be reused several times.

1. Try communicating two processes using named pipes directly in the shell:

   ```
   # mkfifo my.fifo
   # ls -l my.fifo
   ...
   # echo "Hello world" > my.fifo
   ```

   Then start a new console, and run:

   ```
   # cat < my.fifo
   ```

   Try the same operations in reverse order.

2. Write 2 programs communicating through a named pipe. The first program should write something to the pipe:

   ```
   mkfifo("my.fifo", 0600);
   fd = open("my.fifo", O_WRONLY);
   write(fd, "Hello", 6);
   ```

   and the second should read from the pipe. Supplement the code with additional messages printed on the screen to see when and how the processes synchronize with each other.

3. Write programs that will execute `ps ax | tr a-z A-Z` using a named pipe.

11 Inter Process Communication

Pipes introduced in the previous chapter are used for communicating processes. They are convenient because the mechanism controls the flow of the data: writing process is suspended when the buffer is full, and the reading process is suspended when the buffer is empty. The downside of pipes is the overhead introduced by necessary copying of data from one process to the system (kernel), and then from the system to the other process. It requires issuing of system calls, and switching from application level to system level, and then back to application level. The fastest way of passing data between processes is to use shared memory, because the data copied into shared memory block is immediately available for reading by other processes using this block. Unfortunately, processes – for the sake of security – are isolated, their address spaces are disjoint. We need a special, system mechanism to define blocks of shared memory, accessible by several processes at the same time.

11.1 Memory-mapped I/O

11.1.1 Introduction

Before introducing shared memory we have to present an alternative method of processing files. Instead of invoking system calls discussed in Chapter 8, like `open()`, `read()`, `write()`, we can *map* the contents of a file into memory, and access it directly as regular data structures. The operating system will load into RAM missing parts of the file contents from the disk drive when necessary, and only when the data parts

are accessed. Modified parts of the the memory will be at some point written back to the disk to achieve persistency. For many applications this method of processing files is not only more convenient but also more efficient, because the data does not have to be copied twice: from the drive to system buffers, and then from system buffers to application memory.

There are two modes of mapping files: *private* and *shared*. Private mapping does not affect mapping of other processes of the same file (or region of the file). Updates to the mapping are not visible to other processes mapping the same file, and are not carried through to the underlying file. Shared mapping, as the name stands, is shared between processes mapping the same file. Updates to the mapping are visible to other processes mapping the same file, and are carried through to the underlying file.

Besides mapping an existing file, it is also possible to map *anonymously*. It is equivalent to mapping `/dev/zero` file, which is special file generating zeros. Anonymously mapped region of memory is therefore filled with zeros. Consequently, we have the following 4 possible ways of mapping files:

Mode	Mapping of a file	Anonymous mapping
`MAP_PRIVATE`	memory block initiated with the contents of a file	memory allocation
`MAP_SHARED`	shared access to a file	shared memory

Remarks:

- Updates of privately mapped memory pages creates copies of these pages, which is known as *copy-on-write*.
- The `exec()` system call cancels any mapping.
- New process created by `fork()` system call inherits the mapping of the parent process, however, privately mapped memory remains unique for that process.
- A file memory-mapped in shared mode becomes a persistent shared memory block stored on a disk.
- Memory blocks mapped anonymously in private mode are just blocks of dynamically allocated memory. The standard `malloc()` function uses this method for allocating RAM when the block has a size of at least 128 KiB.
- Memory mapping of a process with identifier PID can be observed in `/proc/PID/maps`.

11.1.2 Memory mapping in practice

1. Memory mapping is created using `mmap(2)` system call:

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
```

In case of error the function returns MAP_FAILED. Example usage:

```
struct stat sb;
int fd = open("a.txt", O_RDWR);
fstat(fd, &sb);
void *ptr = mmap(NULL, sb.st_size, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
if (ptr == MAP_FAILED) perror("mmap");
```

2. A new file can be initialized using ftruncate() and fallocate() system calls. The ftruncate() function truncates the file to the specified size (possibly 0), while fallocate() sets the new size of the file:

```
fd = open("dane.txt", O_CREAT | O_RDWR);
ftruncate(fd, 0);
fallocate(fd, 0, 0, 4096);
```

3. The memory mapping can be cancelled using munmap():

```
int length = 4096;
void *ptr = mmap(NULL, length, PROT_READ, MAP_SHARED, fd, 0);
...
munmap(ptr, length);
```

4. The data modified in memory can be synchronized with the contents of the file on disk using msync():

```
msync(ptr, length, MS_SYNC);
```

This function will block the process till the modifications are written to the disk. The function can also be issued in asynchronous mode:

```
msync(ptr, length, MS_ASYNC);
fdatasync(fd);   // or fsync(fd)
```

which synchronises the data with system buffers. Actual write is forced in this case by means of fdatasync().

Whenever a file has been modified, one can force its refreshing in memory by issuing:

```
msync(ptr, length, MS_INVALIDATE);
```

5. Other useful functions related to memory mapping:

mprotect()	change the memory protection mode (PROT_NONE, PROT_READ, PROT_WRITE),
mlock()	lock the memory by preventing it from being paged to the swap area,
mlockall()	lock the whole memory of the process,

`madvise()` an advice concerning expected usage pattern of the memory block.

6. Write a program displaying the contents of a file using memory mapping. Compare performance of the "classical" implementation with this one.
7. Consider once again exercises from Section 8.2. Implementation of which of them can be simplified by means of memory mapped I/O?

11.2 Introduction to POSIX IPC

Inter process communication (IPC) *sensu largo* covers any mechanism provided by the operating system for exchanging information between isolated processes, like for example pipes. However, IPC *sensu stricto* denotes a specific set of such mechanisms introduced originally in System V, which was one of the first commercial versions of the Unix operating system. System V IPC[1] covers:

Shared memory: a mechanism for creating memory blocks outside processes that can be attached to address spaces of several processes for shared access.

Semaphores: a synchronization mechanism for coordination of process execution.

Message queues: a mechanism for exchanging messages between processes.

System V IPC was adopted by majority of other versions of Unix systems, and is also available in Linux system. However, the aforementioned set of communication mechanisms was standardized later on, and is known as POSIX IPC. Nowadays, these two sets of system APIs coexist, because none of them dominates the other. They have their strong and weak points. In this workbook we will present only the POSIX IPC standard. This standard has the following characteristics:

- The POSIX IPC objects are identified as files in the file system, and one can manipulate the objects using many standard Unix commands, like `rm`, `mv`.
- Access rights for IPC objects are defined in the same way as for regular files (`chmod`). It is also possible to use *Access Control Lists* (ACL).
- Compilation requires linking with librt (*realtime*) or pthread library, e.g.:

  ```
  # gcc -lrt -o mytest mytest.c
  ```

  The `-l` switch expects the name of the library without the initial "lib".

[1] See svipc(7) manual page for more information.

11.3 POSIX shared memory

General introduction to POSIX shared memory API is available on shm_overview(7) manual page.

1. Create a shared memory block using `shm_open()` system call which acts like standard `open()` function for files:

   ```
   int fd = shm_open("/mymem", O_CREAT | O_RDWR, 0600);
   ```

   The name of the shared memory block should be in the form: `/myname`. Creation of a new memory block results in creation of a file in `/dev/shm` directory, which is special filesystem stored in RAM (ramdisk). The arguments and flags used by the `shm_open()` system call are identical to those used by `open()`.

2. Set the size of the block of memory using `ftruncate()`:

   ```
   ftruncate(fd, 4096);
   ```

   You can dynamically change the size of the memory block without loosing its contents.

3. Actual access to the memory block is possible after mapping the special file from `/dev/shm` directory to memory using `mmap()` system call discussed in Section 11.1. You have to use the shared mode.

4. The shared memory block can be removed using `shm_unlink()`:

   ```
   shm_unlink("/mymem");
   ```

5. Write two programs exchanging data through a block of shared memory. The first program should be writing in an infinite loop to the same location in the memory block alternately two strings:

   ```
   XXXXXXXXXX
   0000000000
   ```

   Initially you can start with a 1 second delay between writes, but later on the program should ran as quickly as possible. The second program should be reading the string from the memory block, and checking whether it is "XXXXXXXXXX" or "OOOOOOOOOO". Whenever the program reads something else, it should print the string on the screen. In case of correct communication, the second program should not write anything on the screen, even while running with full speed.

11.4 POSIX semphores

Semaphores are used to synchronize processes. Usually semaphore's state is represented by a natural number with values dependent on the type of the semaphore. In general we distinguish two types of semaphores:

Binary semaphores can be in one of two states: locked (0) and unlocked (1). The meaning is analogous to railway semaphores: unlocked semaphores allows access, while locked semaphores blocks. Binary semaphores are usually used to implement a *critical secions* (or *critical regions*), i.e. parts of code that can be run concurrently by at most 1 process, so that competing processes will *mutually exclude* themselves.

Counting semaphores can get any value from 0 up. The semaphore value may represent the number of available resources. The value can be freely increased or decreased on condition that the final value never drops below 0.

The are two basic operations on semaphores:

V *verhogen* (increase), other names: up, release. This operation increases the value of the semaphore by 1 or some other value. The V operation can be performed at any time.

P *proberen* (decrease), other names: down, acquire. This operation decreases the value of the semaphore by 1 or some other value. The P operation can be performed only when the resulting value can remain non-negative. Otherwise, the operation blocks till the condition can be satisfied.

POSIX semaphores are counting ones. They come in two forms: named and unnamed semaphores. The only difference between them is the way they are created/initialized. You can read more about POSIX standard for handling sempahors on sem_overview(7) manual page.

11.4.1 Unnamed semaphores

Unnamed semaphores, as the name suggests, do not have names. As a result they can be accessed only by processes that have access to common shared memory. The semaphore is represented by a structure `sem_t` which has to be initialized before use:

`sem_init()` The second argument of the function indicates whether the semaphore is to be shared between threads of a single process (0) or between processes (1). In the latter case, the `sem_t` structure has to be located in shared memory. The third argument denotes the initial value of the semaphore.

`sem_destroy()` Destroys a semaphore previously initialized with `sem_init()`.

After successful initialization, the semaphore can be handled with the following functions:

`sem_post()` incrementation of the semaphore's value by 1. This is the V operation that can be performed at any time.

`sem_wait()` decrementation of the semaphore's value by 1. This is the P operation, that can be blocking.

`sem_trywait()` non-blocking version of `sem_wait()`: the value is decreased when possible, otherwise an error is returned.

`sem_getvalue()` reading of the current value of the semaphore.

Compilation of programs using POSIX semaphores requires linking with pthread (POSIX threads) library:

```
# gcc -lpthread sem-test.c
```

Synchronization of related processes requires allocation of shared memory:

```
sem_t *s;
s = mmap(NULL, sizeof(sem_t), PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0);
if (s == MAP_FAILED) perror("mmap");
sem_init(s, 1, 1);
```

11.4.2 Named semaphores

Synchronization of unrelated processes requires the use of named semaphores, that are identified by names in the form of /mysem. The names are structured alike shared memory block because named semaphores are stored as small regions of shared memory, and visible as files in `/dev/shm` directory.

1. Create a named semaphore:

   ```
   sem_t *s = sem_open("/mysem", O_CREAT, 0600, 1);
   if (s==SEM_FAILED) perror("sem_open");
   ```

   The last 2 arguments are optional and are used when a new semaphore is to be created, namely: the initial access rights and initial value of the semaphore. New semaphore is represented by a file `sem.mysem` in `/dev/shm` directory.

2. Close and destroy a semaphore:

   ```
   sem_close(s);
   sem_unlink("/mysem");
   ```

   The semaphore will be removed only after closing it by all processes accessing it.

11.4.3 Synchronization problems

1. Write a program implementing a function that should be run by at most 1 process at a time. The function should be called 3 times by the process and its child process. Observe how the processes compete for entry into the critical section. Use unnamed semaphores.
2. Write a general program for manipulation of named semaphores, and verify behaviour of the synchronisation object while performing V and P operations. Example invocations can look like this:

   ```
   # ./sem /mysem          # create the semaphore
   # ./sem /mysem 1        # increment by 1
   # ./sem /mysem -2       # decrement by 2
   ```

   The program can display the current value of the semaphore before and after the operation.
3. Supplement the solution of the exercise 5 from section 11.3 with operations on semaphores so that the memory will be accesses in critical section resulting in correct data transfers.
4. Solve the problem of *readers and writers*. The reading room has capacity of n readers. Readers come to the reading room, allocate a single place, and occupy it for some time, then leave. After some time they come again and the procedure repeats. The reading room is also used by writers. However, a writer can only work when the reading room is empty, i.e. there must be no other reader nor writer. The writer occupy the room for some time, then leaves, and comes back after a while. Provide implementation of 2 programs implementing a reader and a writer, so that it is possible to dynamically start new processes while complying with the restrictions.

   Pay attention to the properties of concurrent processing: *safety* and *liveness*. Consider also whether you program is *deadlock free*.
5. Read about the *dining philosophers problem*, propose a solution using semaphores, and implement the proposal.

12 Multithreaded Processing

Threads are lightweight processes, e.g. autonomous, independently scheduled sequences of processor instructions. There are different approaches to implementation of this concept along with different programming interfaces allowing for creation and management of threads. In this chapter we will experiment with POSIX threads – a standardized interface, commonly available in Unix systems. POSIX threads are just functions that can be started concurrently to the `main()` function of C programs. We can pass a single pointer to such function, and the function can return a single pointer. This way it is possible to freely parametrize threads, and check their return values. Compilation of programs using POSIX threads requires linking with pthread library:

```
# cc -lpthread -o thtest thtest.c
```

More introductory information about POSIX threads can be found on pthreads(7) manual page.

12.1 Thread management

1. Create a new thread running along the main thread:

```
void* worker(void* arg) {
  ...
}

pthread_t th;
```

```
pthread_create(th, NULL, worker, NULL);
```

The second argument of `pthread_create()` function is for passing initial attributes of the thread. The third argument is the address of the function implementing the thread. The fourth argument is the pointer to be passed to the thread function. Newly created thread starts immediately and executes concurrently with the other threads of the process.

2. List currently working processes and identify threads:

```
# ps x
# ps -L x
# ps -T x
```

Pay special attention to columns: PID, LWP (*light weight process*) and SPID.

3. Test passing arguments to threads, for example start 2 threads and pass different integer values to them:

```
int x = 1;
pthread_create(&th, NULL, worker, &x);
```

4. Test reception of data from threads using `pthread_join()` function, which is similar to `wait()` system call, i.e. it waits for termination of the thread and then gets a pointer from the thread:

```
void* ptr;
pthread_join(th, &ptr);
```

Please note, that the address returned by the thread must survive after the thread has ended. It means that it cannot be an automatic variable, that is allocated on the stack. It should be newly allocated memory (from the heap), or some global memory of the process. The thread can just return the pointer or can use the `pthread_exit()` library function:

```
void* worker(void* arg) {
  char* ret = malloc(1024);
  strcpy(ret, "Ala ma kota");
  pthread_exit(ret);
}
```

5. Other useful management functions:
   - A thread can be cancelled by another thread by means of `pthread_cancel()` function.
   - A thread can be detached using the `pthread_detach()` function. A detached thread automatically releases all its resources when it terminates. However, you cannot wait for a detached thread using `pthread_join()`.

12.2 Thread synchronization

There are 3 different mechanisms offered by POSIX threads for synchronization purposes:

Mutexes are simple binary semaphores for implementing critical sections,

Counting semaphores already introduced in Section 11.4,

Conditional variables used for implementing *monitors* (see Section 12.2.2).

The counting semaphores used for synchronizing threads are exactly the same semaphores as used for processes. When synchronizing threads of a single process we can use unnamed semaphores stored in internal memory of the process, and we do not need to map memory, which make the semaphores lightweight.

12.2.1 Mutexes

Mutexes are simple, binary semaphores which are either locked or unlocked. The following functions are used for managing mutexes:

`pthread_mutex_init()`
: initializes a mutex using specified set of attributes;

`pthread_mutex_lock()`
: locks the mutex, which may result in blocking of the calling thread when the mutex has already been locked;

`pthread_mutex_trylock()`
: locks the mutex when it was unlocked, or returns `EBUSY` when it was already locked;

`pthread_mutex_unlock()`
: unlocks the mutex, which under normal circumstances should be done by the thread that has acquired it;

`pthread_mutex_destroy()`
: destroys the previously initialized mutex.

All the above functions take as an argument `pthread_mutex_t` structure representing the mutex.

12.2.2 Conditional variables

Conditional variables allow for controlled falling asleep and waking up of processes. This is the primary tool for constructing *monitors*. Conditional variables are represented by `pthread_cond_t` structure, and are initialized using `pthread_cond_init()` func-

tion. A thread can be awaken using `pthread_cond_signal()` function or by means of `pthread_cond_broadcast()`. The `signal()` function activates a single waiting thread, while the `broadcast()` function activates all waiting threads associated with a given conditional variable. Threads can fall asleep using the `pthread_cond_wait()` function. In order to call the `pthread_cond_wait()` function one has to first acquire a mutex, which is then atomically released while falling asleep. Before resuming the thread the mutex will be automatically acquired again. There is also another function for time-restricted waiting: `pthread_cond_timedwait()`. The following example demonstrates a typical application of conditional variables:

1. The waiting thread:

```
pthread_cond_t  c;
pthread_mutex_t  m;
...
pthread_mutex_lock(&m);
pthread_cond_wait(&c, &m);
pthread_mutex_unlock(&m);
```

2. The awakening thread:

```
pthread_mutex_lock(&m);
pthread_cond_signal(&c);
pthread_mutex_unlock(&m);
```

Although this is not required, it is a good practice to awake threads from within a critical section of the mutex guarding the critical section. Thanks to that we can avoid *race conditions*.

After waking up, the thread should check whether the current conditions allow processing to continue, because of *spurious wakeup*, i.e. wakeups not caused by explicit invocations of `signal()` or `broadcast()`. Therefore, waiting for a given condition should be implemented like this:

```
pthread_mutex_lock(&m);
while (!condition) {
  pthread_cond_wait(&c, &m);
}
...
pthread_mutex_unlock(&m);
```

12.2.3 The producer-consumer problem

Let us consider two processes: a producer and a consumer, that exchange data chunks using a shared buffer of N slots. Whenever the producer creates a new chunk of data it tries to put it into the buffer. When the buffer is full, the producer suspends its

processing. The consumer waits for new data chunks in the buffer, and consumes them. It suspends its processing when the buffer is empty. A buffer of size $N = 1$ causes alternating activation of the producer and the consumer, because it is not possible to produce and consume at the same time, which is suboptimal in case of multiprocessors. A buffer of size $N = 2$ makes it possible to engage both processes at the same time: in the first phase the producer writes to position 1, and the consumer reads from position 2, and in the second phase, the producer writes to position 2, and the consumer reads from position 1. However, this assumes that the speed of production and consumption is equal and constant. Whenever there are some fluctuations of the processing time, it may be advantageous to use a buffer of size $N > 2$, which is then organized as a *circular buffer* (or *circular queue*). The bigger the buffer, the larger fluctuations may be compensated by it. Similar problems are considered in logistics: we have factories (producers), warehouses (buffers), and shops (consumers). The problem boils down to proper synchronization of the producer and the consumer.

The solution

We will use two counting semaphores: S_p for the producer, and S_c for the consumer. The buffer will be represented by an array with indexes ranging from 0 to $N - 1$. Assuming the buffer is empty at the beginning, the initial values of semaphores are: $S_p = N$, $S_c = 0$. The following is a pseudocode of the producer and the consumer:

```
Producer:                    Consumer:
  i:=0                         i:=0
  while true:                  while true:
    P(Sp)                        P(Sc)
    produce(i)                   consume(i)
    i:=(i+1) mod N               i:=(i+1) mod N
    V(Sc)                        V(Sp)
```

1. What is the order of data chunks being consumed?
2. Does this solution guarantee safety, i.e. the producer never overwrites unread slots, and the consumer never reads from empty slots?
3. Does this solution guarantee liveness, i.e. the producer and the consumer finally will proceed to the next slot?
4. Is it safe to start concurrently two (or more) producers and/or two (or more) consumers?

12.2.4 Other synchronization problems

1. Try to solve the producer-consumer problem discussed in Section 12.2.3 using conditional variables. How many conditional variables need to be used? Try

to avoid unnecessary awakening of threads, especially in a generalized solution with arbitrary number of producers and consumers.

2. Implement a function `barrier(n)` that should synchronize n threads. The function should suspend all calling threads except the n-th one. The n-th invocation should also resume processing of all threads that have called the function previously. Barriers are used to synchronize groups of threads that should work in rounds, where results of the previous round are necessary to start the next round.
3. Consider a special variant of counting semaphore that is bounded on both sides, i.e. its value should always be in range $[0, k]$. The semaphore should provide the following operations:

   $P(s, i)$ decrease the semaphore's value by i unless the final value drops below zero,

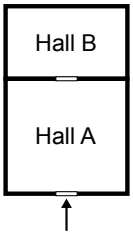
   $V(s, i)$ increase the semaphore's value by i unless the final value exceeds k.

   Both operations can be blocking, when the change cannot be applied immediately. Additionally, the functions can return an error when $i > k$. You can also consider a non-blocking implementation of operation $V(s, i)$, which just truncates the value of the semaphore at k. Solve the producer-consumer problem using such semaphores.

A
Final Assignments

A.1 The museum

Image a small museum with just 2 halls: A and B. While visiting the museum you enter the hall A, and you can reach hall B by passing through hall A. The situation is depicted in the following figure:



People visiting the museum enter hall A, watch the exhibition, and decide whether they want to continue by proceeding to hall B. Some people leave the museum immediately, others enter hall B, watch the second exhibition, and than enter once again hall A, and finally leave. For safety reasons there must be no more than N_A people in hall A at the same time, and no more than N_B people in hall B ($N_B < N_A$). Write a program(s) implementing synchronization algorithm that will allow:

1. visiting the museum at the same time by as many people as possible,
2. leaving the museum by people visiting hall B in the shortest possible time.

A.2 The producer-consumer problem revisited

Write two programs implementing the producer and the consumer of the problem discussed in Section 12.2.3, but in a generalized version that makes it possible to run several producers and several consumers at the same time. A buffer of N slots should allow for running up to N concurrent processes, producing and consuming different items. The processes should run in loops, with different and variable speeds. The ordering of items being consumed should reflect production end times.

Index