Operating systems. Lecture 4

Michał Goliński

2018-10-23

Introduction

Recall

- Filesystems
- Virtual File System
- Processing text in `bash`
  - `cat`, `od`, `base64`
  - `head`, `tail`, `split`, `csplit`
  - `sort`, `uniq`
  - `cut`, `paste`, `join`
  - `tr`, `grep`, `sed`

Questions?

Plan for today

- Virtual memory
- Revisiting regular expressions
- `bash`
  - variables
  - expansion
  - loops
  - conditionals

Virtual memory

Introduction

Arguably the most important resource in a computer is the memory. Running processes have to fit in the available memory (as the CPU can only operate on data in memory).

Problems to solve

- Memory contains data and code mixed together. For this reason moving process's data in memory for defragmenting would be very difficult.
- All program have a so-called *entry point* (the memory address of the first instruction). Different programs might use the same entry point.
- One process should not be able to read memory of another process (security).
- Many programs claim they need more memory they actually use. It might be a good idea to be able to overcommit.
- Multiple programs use the same library – it would be beneficial for them to share just one copy of this library in memory. At the same time the library might need to have

Paging

Almost all current general purpose operating systems use a mechanism called paging. Some special operating systems (in particular some embedded devices) might not use virtual memory but most often they do not have oven an operating system at all.

Paging cont.

Paging uses an intermediate layer between a running process and physical memory. A process uses *logical addresses* which live in *pages*. Pages are loaded into physical memory *frames* as needed. Pages under Linux have 4kB of space.

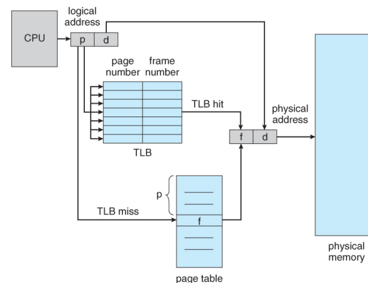

Figure 1: Translating logical to physical addresses

Allocating memory

A process can ask the OS for more memory – this process is called *allocation*. Kernel allocates a memory area for a process by assigning a new contiguous area aligned on pages boundaries or returns an error. In most cases the precise address is not relevant for the process, only the amount obtained.

MMU

The translation between logical and physical addresses has to be very fast. For this reason all modern CPUs have a special hardware device called *memory management unit* and *translation lookaside buffer*. Both of them ensure that very little time is needed for the translation.

Swap

Unused pages can be written to disk to save precious memory space. Under Windows the pages end up in a so-called swap file (growing as needed). Under Linux one can usually has a dedicated swap partition or a file of fixed size.

Page fault

If the processor determines that an address requested by a process belongs to a page that is currently not loaded into memory, we have a *page fault*. The process is paused and the operating system is asked to load the page. A free frame is either found or, if memory is full, another frame (usually belonging to another process) is *swapped* to disk. The requested page is put in the empty frame.

Each process has its own *page table* which contains the mapping of pages to frames for this particular process. Corresponding page table is loaded when a process is being run.

Eviction

When there is little free memory left the system has to decide which pages (of the running or maybe paused processes) can be swapped to disk. It should take a page that will not be needed for some time. Usually the *least recently used* (LRU) is being chosen.

Thrashing

If pages are moved often between swap and main memory we speak of *thrashing*. This degrades performance and responsiveness considerably and is a sign that the system needs more physical memory.

Security

Process address space is strictly checked by the operating system. Pages can be marked read-only or non-executable. If a process tries to access an address that is outside the range of pages it was given, or writes to a read-only page, or tries to execute a page that was not marked as executable, a *segmentation fault* is generated. By default the process is killed.

Reusing the same frame

Multiple processes may use the same memory area. This is used when a program requires *shared libraries* (DLLs in Windows). Only one copy of a library needs to be loaded into memory and it will be mapped into address spaces of multiple processes that need it. Each

such process would have private pages for private variables used by the library and would share the code.

Copy on write

Often a page can be reused but a process still needs to be able to write data there. This is done by marking a page as copy-on-write. When a write occurs, a copy of the original page is created and the process writes into the copy.

Kernel address space

In each process the memory addresses ranging from 0xFFFF800000000000 to 0xFFFFFFFFFFFFFFFF are reserved for the kernel data and they are the same in every process. Accessing this memory is possibly only for the kernel, not for user programs.

User address space

In each process addresses ranging from 0x0000000000000000 to 0x00007FFFFFFFFFFF are available for the process's data, code etc.

Address space of a process

The current address space of a process (valid addresses in virtual memory) can be looked up in the file `/proc/PID/maps`, where `PID` is the process number.

Stack

All running processes have a private memory space called the stack. Stack is used for temporary variables when a function is being called, e.g., all the function arguments and automatic variables in C live on the stack. If A process *overflows* its stack, i.e., by a too deep recursion, it will get the segmentation fault signal. Although stack can be very large if requsted, usually it is not that big and large amounts of data should not be put on the stack. By convention the stack is located close to the upper limit of the address space and grows downwards.

Heap

The data that does not belong to the currently executed function lives in the *heap*. The only way to access the heap is through pointers (directly or indirectly). New space on the heap is obtained in C by calling the `malloc` function or the `new` operator. Heap is the right place for large amounts of data.

Regular expressions revisited

Sample regular expressions

- URL
- e-mail address
- IP4 address
- any line that does not contain “foo”
- palindrome

Useful `grep` invocations

```
$ grep -R regex #search in files recursively
$ grep -l regex #print names of matching files
$ grep -C2 regex #show surrounding lines
```

Sample `sed` invocations

```
$ sed -r 's/ .*//'
$ sed -r 's/.{16}//'
$ sed -ri 's/ ([aiouwz]) / \1~/' *.tex
```

Structural programming in `bash`

Shell variables

Variables in bash can contain any text. A variable name has to start with a letter. Names are case-sensitive. They are created with the following syntax:

```
$ a=1
$ A=”some text”
```

Environment variable

A shell variable can be promoted to an environment variable that is passed to commands. This is done by the `export` command. Once exported the variable does not need to be reexported if its value changes.

```
$ a=1
$ env | grep ^a
```

```
$ export a
$ env | grep ^a
$ a=2
$ env | grep ^a
```

The `PATH` variable

This variable contains the list of directories that are searched for executables (commands). The list is separated by colons.

The `PS1` environment variable

This variable allows one to customize the prompt.

The `IFS` environment variable

This variable contains the characters that separate arguments (default: space, tab, newline).

The `LANG` environment variable

This variable is used by many programs (C library in faact) to determine `locale`, i.e., the language of the system.

```
$ echo $LANG
$ man man
$ LANG=en_US.UTF-8 man man
```

Variable substitution

The shell *substitutes* the variable value when it encounters the character `$` and the variable name. Non-existing variables are expanded to empty sequences. If we want to tell the shell where the variable name ends we can use braces.

```
$ echo $var iable
$ echo $variable
$ echo ${var}iable
```

Suppressing substitution

Sometimes we need a raw `$`. To suppress the variable expansion and leave the string as-is, use the single quotes:

```
$ echo $variable
$ echo '$variable'
```

Double quotes do not suppress expansion but skip cutting arguments at spaces.

Command substitution

A very important and useful construction does something similar with commands. When the shell encounters text in backticks or in `$(` and `)` it runs it as command and replaces the text with the command's output.

```
$ echo $(seq -w 1 100)
```

Calculations

The old way of doing calculations employs the `let` command. More modern (but works only in bash, not in older shells) employs double parentheses. Then one calculates almost like in C.

```
$ a=10
$ a=$(( 2*a ))
$ (( ++a ))
```

Observe that one does not use the `$` character. Every variable is understood to contain a number.

Brace expansion

Sometimes we need a list of elements of strictly prescribed list. Brace expansion can be helpful:

```
$ echo {1,3,5,7}
$ echo Ep{01,07,13,87}.mkv
$ echo {1..9}{0..9}{0..9}
```

Conditional expressions

Status code

Every command when run leaves a small integer that can be checked – this is the so-called *status-code*. In C this is the value returned from the `main` function. By convention when

this value is 0 it means that the command completed successfully. Non-zero values mean some kind of error.

The status code is used instead of boolean values in conditional expressions.

Test command (`[`)

Often a boolean test is needed (check if file exists, compare numbers). The old way of testing these conditions is by the `test` program. This program can also be run as `[`. Details can be found by reading `man test`.

```
$ test -e file.txt
$ [ -e file.txt ]
$ [ 11 < 2 ]
$ [ 11 -lt 2 ]
```

Bash conditionals (`[[`)

The new way of testing conditionals is by the `[[ ]]` builtin. It works similarly to `[` but has sane meaning for < and > and is generally considered better.

Conditional instruction

The `if` construction looks like this

```
if command
then
  commands to run when true
else
  commands to run when false
fi
```

Note the `fi`. True and false are chosen based on the status code of the `command`.

`case`

There is also an equivalent of the switch instruction:

```
case expression in
    pattern1 )
        statements ;;
    pattern2 )
        statements ;;
    ...
esac
```

Loops

The `for` loop

The `for` loop iterates over a list of values, with each iteration the given variable's value is set to consecutive values from the list. We can have many commands inside the loop

```
$ for i in 1 2 3
> do
> echo $i
> done
```

me_irl

```
$ for i in $(curl -o- -L 'https://uo.amu.edu.pl/kursy' | grep 'a
↪  href.*kursy' | sed -r ”s/.*'(.*)'.*/\1/g”)
> do
> echo https://uo.amu.edu.pl$i
> curl -o- https://uo.amu.edu.pl$i 2>/dev/null | grep ”Kurs nie
↪  został uruchomiony.”
> done
```

The `while` loop

While loop looks works similarly to the while loop in C.

Examples

Countdown

```
a=10
while [[ $a -ge 0 ]]
do
  echo $a
  (( --a ))
  sleep 0.2
done
```

Primality test

```
read n
k=2
while [[ $k -lt $n ]]
do
  if (( n % k ))
  then
    (( ++k ))
  else
    echo $n is not prime, $k is a non-trivial divisor.
    break
  fi
done
if [[ $k -eq $n ]]
then
  echo $n is prime.
fi
```