Operating systems. Lecture 5

Michał Goliński

2018-10-30

Introduction

Recall

- Virtual memory
- Structural programming in `bash`
  - conditionals
  - `if`
  - `for`
  - `while`

Questions?

Plan for today

- Lifecycle of a process
- `bash`
  - functions
  - scripts

How does the CPU work?

Assembly

The only language the CPU understands is the machine language. It is in binary, so not very human friendly. The assembly language is a low level language that has an almost one-to-one correspondence to the machine language. Different architectures use different assemblies.

One can find out the code generated from C code by using, e.g., the following two commands:

```
gcc -O1 -g -c <program.c>
objdump -S -M intel <program.o>
```

Registers

The CPU has a very small amount of very fast memory: registers. Operating on RAM can take many CPU cycles, while operating on registers is usually 1-2 cycles. Some of the registers are used in general computation (e.g., most calculations leave their result in the RAX register) and some have very specific meaning (e.g., the instruction couter IC).

Life of a process

Process state graph

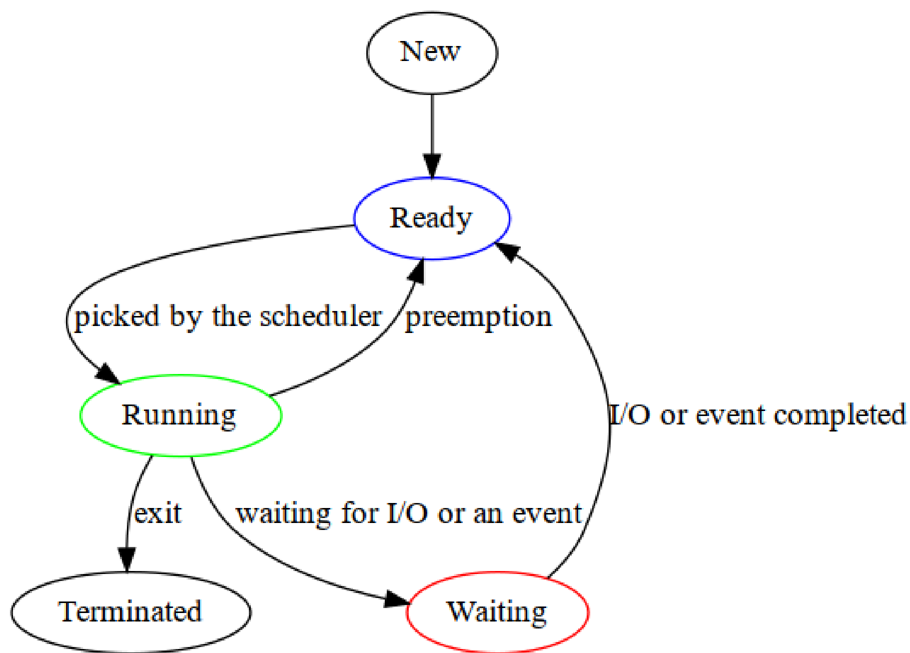

Figure 1: Process state graph

Process descriptor

SO stores information about the processes in the so-called descriptors. In particular the following are stored:

- process ID
- parent process ID
- used processor time
- virtual memory (page table)
- values of all registers, in particular the program counter (address of the instruction that is being executed)

- open files and connections

Process descriptor cont.

Under Linux the process descriptor is stored in the `task_struct` structure defined in `include/linux/sched.h`.

Most information about running processes can be found in the directory `/proc/<process ID>`.

Interrupts

Interrupts are a mechanism that informs the processor of an event that has happened and that has to be handled. Interrupt cause the currently run code to be, well, interrupted and a corresponding *interrupt handler* is being executed.

Interrupts cont.

Interrupt can be an effect of an external event – hardware interrupts (e.g., keyboard event) or an internal one that is sigalled by the processor – software interrupts (e.g., division by zero). Interrupt can be also triggered programatically – this is roughly th mechanism by which user programs communicate with the OS, i.e., calls *system calls*.

System call in assembly

```
mov rax, 0x3c  ; use the system call number 60 (_exit)
xor rdi, rdi   ; the only parameter (status code) is 0
syscall        ; perform the system call
```

Process preemption

To be able to preempt a process (i.e., forcibly take processor from a process) and to carry out other important tasks OSs use so-called timer interrupts, i.e., an interrupt that is triggered at specific points in time. In the past this used to happen with a specified frequency – *ticks* (e.g., 100 Hz). Moder kernel are *tickless* – timer interrupts are scheduled as needed.

Process preemption cont.

If during the handling of an interrupt (timer or other) the OS determines that the current process occupies he CPU for too long and there is another process that deserves it more (this decision is taken by the *scheduler*), then the current instruction counter is saved and another instruction counter is loaded (corresponding to a different process). A new process will begin executing when that interrupt handler finishes.

Process priority

Many parameters can influence the *scheduler* work. The main one is the process priority. Under Linux we manipulate the priority by setting onether paramter: **nice** in the range -19 to +20. Process with a larger nice value is nice *towards others*, i.e., has a lower priority. Setting a negative nice value requires root privileges.

. . .

On the command line *nice* can be set by the commands `nice` i `renice`.

scheduler – algorithms

scheduler

To see the problems encountered by the people writing OSs we will try to think about how a scheduler might decide and we will see how the CFS (completely fair scheduler) in Linux works.

Criteria

A good *scheduler*:

- decides very fast (small computational complexity)
- cares that the processor is not idle
- cares that processes which are more important for the user (interactive) get the processor whenever they need it
- cares that each process gets processor time eventually (so that it is not *starved*)

Activity bursts

Most processes switch between carrying out computations and waiting for data. A good *scheduler* should adjust to the character of a process – wheather it mostly carries out a lot of calculations (and accordingly can wait for the CPU) or rather small amount of computation in short bursts (which suggests interacting with the user and an instant need for CPU time).

FCFS

First come, first served – CPU time is allocated in the order the processes are created. No preemption. CPU can be taken from the processor only when it waits for I/O. The moment that I/O ends a process is put into the queue of processes waiting for CPU.

. . .

This is the algorithm usually used at the counter in a supermarket.

FCFS – pros and cons

Pros: simplicity

Cons:

- everyone waits for a single process tat is not willing to let go of the processor

SJF

Shortest job first – priority is given to processes that will finish their work first. We have two possibilities: either we do not preempt processes (shortest job first) or we do preempt them if a process comes that will finish sooner than the one currently being processed (shortest remaining time first).

SJF – Pros and cons

Pros: * this algorithm (in theory) minimizes the average time processes have to wait

Cons: * in general it is difficult to tell how long will a particular task take. There are some heuristics (based on past behaviour), but they are still heuristics. * has a potential for starving processes

According to priority

We divide the processes into classes according to their priority. Higher priority processes have priority. In a class we use another algorithm, e.g., FCFS.

Pros and cons

Pros: * important tasks can be carried out faster

Wada: * Low priority risks starvation (solution: raise priority artificially with time)

RR

Round-robin – Processes are put into a queue as the arrive. Each process in the queue is given an certain amaount of CPU time and afterwards it is preempted and put into the queue to wait again.

RR – Pros and cons

Pros: * simplicity

Cons: * Switching between tasks is not instantaneous – we loose quite a lot of time.

Real life example – CFS

Starting with version 2.6.23 Linux uses a novel scheduler – CFS (Completely Fair Scheduler). For each precess a virtual running time is kept, which more or less tracks the amount of time used by the process (e.g., adjusted for priority). At each moment the process with the least virtual time is scheduled for execution (i.e., the process that was treated the most unfairly). Information about processes running time is kept in a *red-black tree* (a self-balancing binary tree).

CFS cont.

This system looks fair, but only as long as there are no new processes. A freshly created process would have a zero virtual running time and hence an unwelcome advantage. To counter that, when a new process is inserted into the tree its virtual running time is adjusted according to the smallest virtual running time in the tree. This gives good results in practice.

`bash` scripts

Script

Script are programs written in `bash`.

Passing arguments

Command line arguments are accessed through the special variables `$1`, `$2`, …, `$9`, `${10}`, …

One can also get all the arguments by using the special variables `$@` and `$*`. The first one creates a list, the other is just one giant string containing the arguments.

Number of arguments is in the variable `$#`.

`shift`

The `shift [n]` command changes positional arguments `${n+1}`, `${n+2}`, … into `$1`, `$2`, …

By default `n=1`.

Functions

Functions are create as follows:

```
f() {
  #code goes here
}
```

One can write `function` before the function name, but this is not necessary. But empty parentheses are important. Arguments are accessed like command line arguments (`$1` etc.).

`read`

One can ask the user for a value using `read`. This is usually done in tandem with `echo` like this:

```
echo -n 'Enter value: ' && read variable_name
```

Subshell

It is sometimes very convenient to run commands in a shell that is the child of the shell executing the script (command substitution uses this mechanism). This allows, e.g., to change the working directory without worrying about going back.

Shebang

We can make a script more like a true executable if we change its permissions (add executable). Still, the system needs to know that it has a bash script on its hands, not a python script etc. To mark that in the first line we put

```
#!/bin/bash
```

The `#!` is called the *shebang*, the rest is just the path to the correct interpreter (usually a true executable).

Debugging

When writing programs sometimes there is a need for debugging. One can trace execution of a bash script by modifying the first line to:

```
#!/bin/bash -x
```

Options

Scripts often need arguments. One can either support them by hand or using the `getopts` builtin.

Options by hand

```
#!/bin/bash

OPTION_A=value
OPTION_B=false

while [[ $1 == -* ]]
do
  case $1 in
    "-a") OPTION_A="$2"
          shift 2
          ;;
    "-b") OPTION_B="true"
          shift
          ;;
  esac
done

echo We use the following values:
echo "OPTION_A=$OPTION_A"
echo "OPTION_B=$OPTION_B"
```

getopts

```
#!/bin/bash
OPTION_A=value
OPTION_B=false

while getopts "a:b" opt; do
  case $opt in
    a)
      OPTION_A="$OPTARG"
      ;;
    b)
      OPTION_B=true
      ;;
  esac
done

echo We use the following values:
echo "OPTION_A=$OPTION_A"
echo "OPTION_B=$OPTION_B"
```