

# Operating Systems

Example solutions for Theoretical Exercise 1

Michael Engel

# 1.1 Unix processes and the shell

- a. `init` is the process at the top of the Unix process hierarchy. Explain why `init` has to run all the time when a Unix system is running.
- `init` has to "catch the zombies", i.e. it becomes the new parent process of processes
  - However, `init` traditionally has additional tasks, e.g.
    - Respawn `getty` processes on terminals when a user logs out
    - Change *runlevels* of the system, which define the set of activities to start and run in a certain system state
      - e.g. single user mode, text mode, GUI mode
    - Activities for a run level are defined in `/etc/inittab`
  - Today, "modern" `init` systems redistribute some of `init`'s original functionality among more/other processes

# 1.1 Unix processes and the shell

b. Describe the function of `exec1` in your own words

(Hint: read the man page).

- The `exec` family of functions replaces the current process image with a new process image. `exec1` is a function that allows the caller to pass command line parameters to the executed program, which appear as `argv` array in `main`.
- Also describe the content of the second parameter passed to `exec1`
- The second parameter is given as `const char *arg0`. Like `printf`, `exec1` is a function with a variable number of arguments. The first one ("path") is always the Unix path of the process to execute, the subsequent ones are a list of arguments passed as `argv[0]`, `argv[1]`, ...
- The second parameter, accessible as `argv[0]` in the called program, by convention always contains the name of the executed program (e.g., "vim").
- The command line parameters given, e.g., in the shell appear as `argv[1]` ...

# 1.1 Unix processes and the shell

- c. Explain the output of the following command in your own words. Which data is transferred through the pipeline and what operation does the `grep` command perform here?

```
ls | grep -vc .pdf
```

- This command lists all files in the current directory (since no directory name is passed to `ls`) and passes this list as text (one line per file) through the pipe to the `grep` command.
- `grep` reads this list from the pipe line by line (separated by `\n`). The options to `grep` mean
  - `-v`: output lines **not** containing the given pattern `.pdf`
  - `-c`: output the count of lines instead of the contents
- So the command counts the number of files in the current directory that have names **not** ending in `.pdf`

# 1.1 Unix processes and the shell

- d. Try to find a shorter form of the following shell command that does not require a pipeline:

```
cat /etc/passwd | grep root | cat > /tmp/x
```

- One solution is

```
grep root < /etc/passwd > /tmp/x
```

- I/O redirection is a powerful feature and you can supply multiple redirections per command. We will see later that the standard operators `<` and `>` redirect the *standard input* and *standard output* channels (text streams) of a program, but there are additional channels that can be redirected (e.g., *standard error* output using `"2>"`).
- The solution using `cat` is less efficient since three processes (2 x `cat`, 1 x `grep`) have to be created instead of only one.
- See also

<https://stackoverflow.com/questions/11710552/useless-use-of-cat>

# 1.2 fork

Consider the following line of C code:

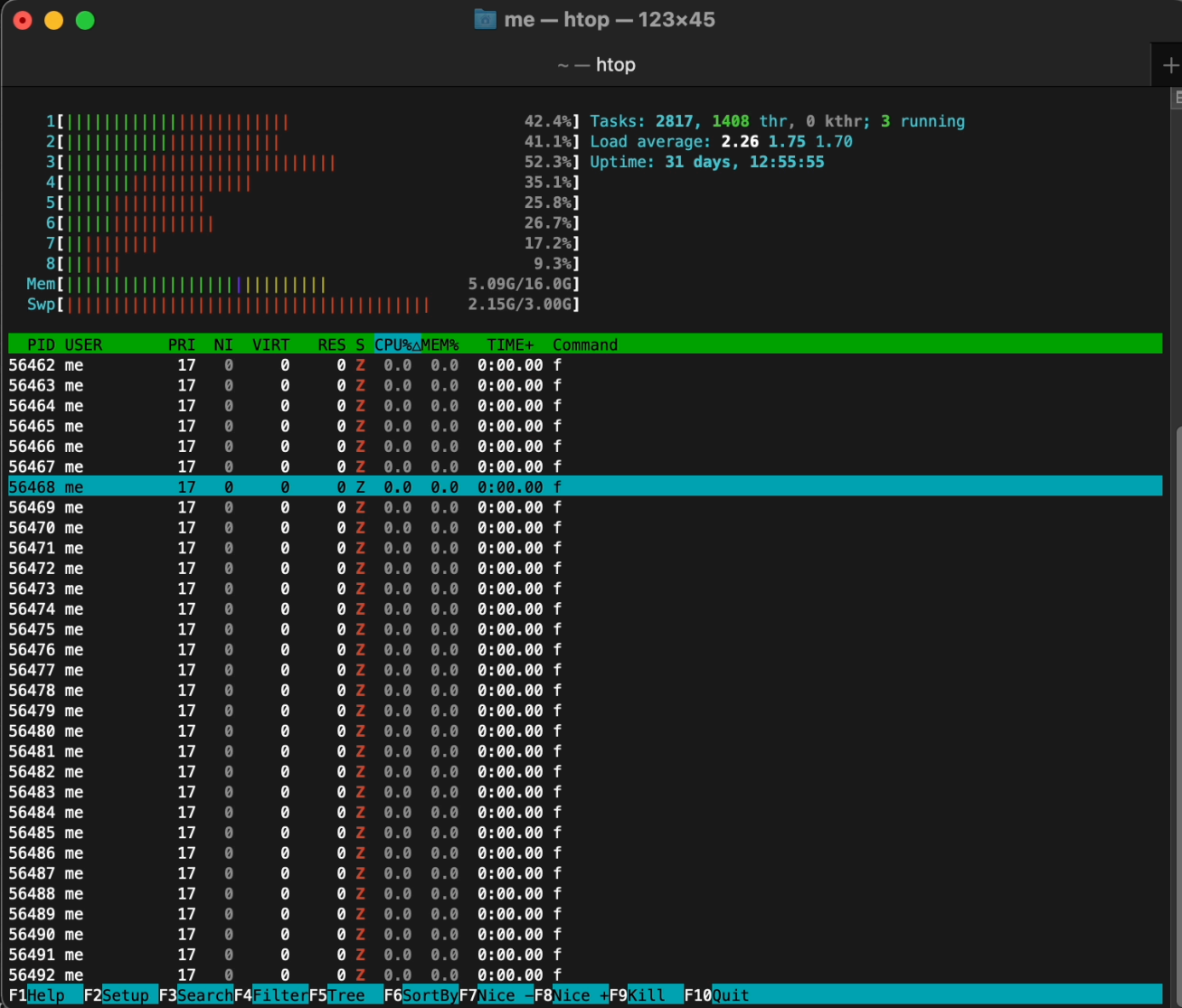
*(Caution: Do not try to execute this!)*

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

- a. Describe the program behavior after 1, 2, 3 and n iterations of the while loop.
  - The first iteration creates one child process, resulting in two processes
  - The child process then exits the while loop, since 0 is returned to it by `fork`
  - The second iteration of the parent process then creates another child process... and so on
  - So in theory we only have one process created per loop iteration which terminates immediately (note: we didn't specify this since we only gave the one line of code...)
  - However, since there is no parent `wait()`ing for the termination of the child process, our process table fills up with zombie processes, potentially making the system unusable!

# 1.2 fork

- Here's the "invasion of the zombies" shown in htop
- We can see that this keeps the CPU cores quite busy...



```
me — htop — 123x45
~ — htop

1[|||||] 42.4% Tasks: 2817, 1408 thr, 0 kthr; 3 running
2[|||||] 41.1% Load average: 2.26 1.75 1.70
3[|||||] 52.3% Uptime: 31 days, 12:55:55
4[|||||] 35.1%
5[|||||] 25.8%
6[|||||] 26.7%
7[|||||] 17.2%
8[|||||] 9.3%
Mem[|||||] 5.09G/16.0G
Swp[|||||] 2.15G/3.00G

PID USER   PRI  NI  VIRT  RES  S  CPU% MEM%  TIME+  Command
56462 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56463 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56464 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56465 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56466 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56467 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56468 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56469 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56470 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56471 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56472 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56473 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56474 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56475 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56476 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56477 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56478 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56479 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56480 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56481 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56482 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56483 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56484 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56485 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56486 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56487 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56488 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56489 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56490 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56491 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
56492 me      17   0    0     0  Z  0.0  0.0  0:00.00 f
F1Help F2Setup F3Search F4Filter F5Tree F6SortBy F7Nice -F8Nice +F9Kill F10Quit
```

# 1.2 fork

- Consider the following line of C code: (Caution: Do not try to execute this!) `while (fork());`
- b. The behavior of a program like this can lead to problems. Describe the problems that can occur. Try to find a way to avoid the problem in Unix (without changing the program above).
- The `ulimit` command (actually a shell built-in command) can restrict the resource use of a specific user. This includes not only the number of processes, but also CPU time, file size, number of open file descriptors, ...

```
[me@Proton ~ % ulimit -a
-t: cpu time (seconds)          unlimited
-f: file size (blocks)          unlimited
-d: data seg size (kbytes)      unlimited
-s: stack size (kbytes)         8176
-c: core file size (blocks)     0
-v: address space (kbytes)      unlimited
-l: locked-in-memory size (kbytes) unlimited
-u: processes                   2666
-n: file descriptors            2560
me@Proton ~ % █
```



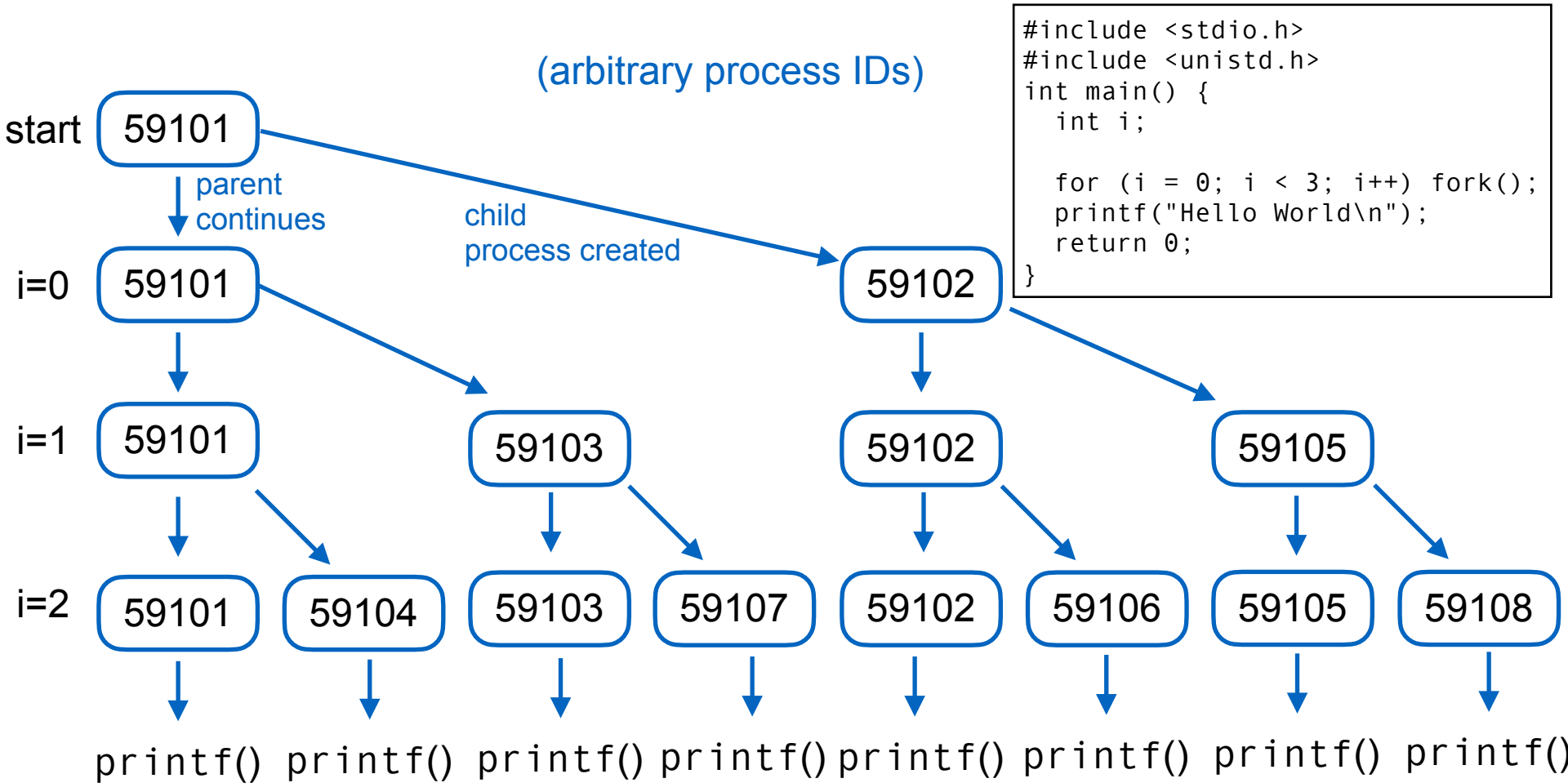
# 1.3 Process execution order

- How many times does the following program print “Hello World”? Draw a simple tree diagram to show the parent-child hierarchy of the spawned processes.
- "Hello World" is printed 8 times.
- fork returns to the *exact* place in the program at which it was called. *All local and global variables keep their values!*
- So the number of processes created increases with the value of the loop variable *i*, each process creates an additional child in each iteration it exists.

```
#include <stdio.h>
#include <unistd.h>
int main() {
    int i;

    for (i = 0; i < 3; i++) fork();
    printf("Hello World\n");
    return 0;
}
```

# 1.3 Process execution order



This example also shows that the order of process execution after fork is not specified, the order implemented is a decision of the scheduler!