Practical 1

Alejandro Pérez Bueno

November 5th, 2022

Introduction

This practice proposes a series of activities with the aim the student can apply on a U system some of the concepts introduced in the subject. The student will have to do experiments and answer the proposed questions. You will also need to write a short program in *C* language. The practice can be developed on any *UNIX* system (the *UOC* facilitates **Ubuntu 14.04** distribution). It's recommended that while you are doing the experiments there are no other users working on the system because the result of some experiments may depend on the system load.

I will personally develop this practical activity from **Fedora Workstation 37** and **ArchLinux**.

Base Code

For the following questions, some code has been provided. It is all included in the `pr1so/` folder. Here is a brief description of what *some* of the files do:

- **`count1.c`** executes an infinite loop where each iteration increments a counter (initialized to `0`). When count1 receives the asynchronous notification indicating that it must end, it prints the counter value and ends. `count1.c` emulates an intensive computational process.
- **`count2.c`** executes an infinite loop where each iteration increase a counter (initialized to `0`) and waits a millisecond (blocking) before iterating again. When `count2` receives an asynchronous notification indicating that it should end, it prints the counter value and ends. `count2.c` emulates an interactive process.
- **`launch.sh`** starts the concurrent execution of `N` processes running the `count1` program and another `N` running the `count2` (`N` is the first parameter of the shellscript). By default, once 3 seconds have elapsed, it terminates all count processes. If we want the execution time to take a different amount of time, the shellscript supports a second optional parameter that indicates the desired execution time.

Question 1. Processes

1.1. Count Programs Study

Question

Once studied the behavior of `count1`, draw a graph of states with three nodes (one for each possible state of the process: **Ready**, **Run** and **Wait**) and with the arcs that reflect the state changes which can occur while a process is running `count1`.

Answer

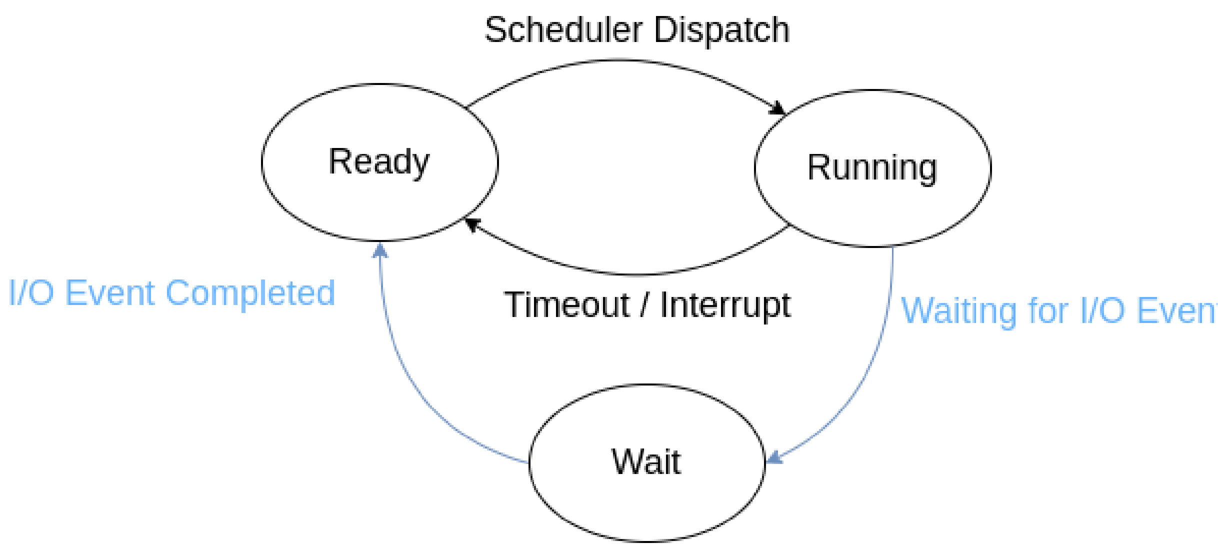


Figure 1: Graph for count1

Question

Similar to 1.1.1., draw the graph showing the state changes that a process running `count2` may happen.

Answer

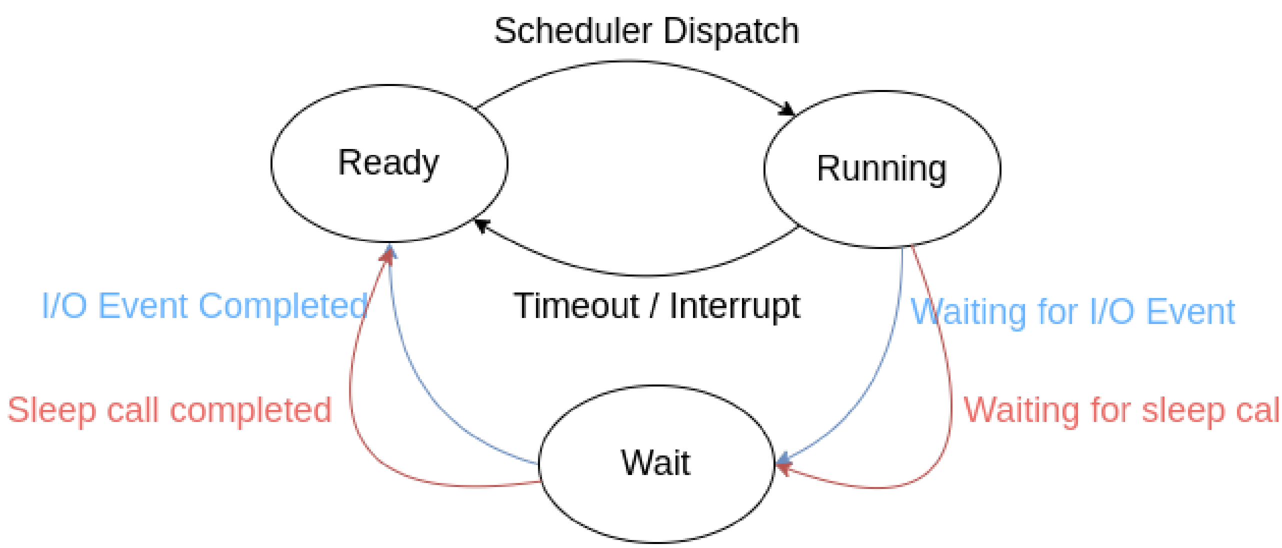


Figure 2: Graph for count2

1.2. Hardware Analysis

Question

Please indicate which specific processor model you are working on.

Answer

```
pixel@minishell ~ $ grep 'model name' /proc/cpuinfo | head -1
model name  : Intel(R) Core(TM) i5-8259U CPU @ 2.30GHz
pixel@minishell ~ $
```

Question

How many physical cores (cores) does your processor have? If your processor is Intel, you may want to check out Intel Ark.

Answer

My device has `8` **logical cores** resulting from `4` **cores** each with `2` **threads**. In other words,

`(4 physical cores) x (2 threads per core) = 8 logical cores`

My device has **Hyperthreading** capabilities, it takes advantage of up to `8` cores running at the same time.

Question

How many cores are you actually using? This number may be different from that obtained in 1.2.2. depending on how the operating system is set up, the virtual machine (in case you are using one) or if you have Hyperthreading enabled. To answer this question, count how many lines the following command is typed in and attach a screenshot of the result.

`grep processor /proc/cpuinfo`

If the result is less than the number you obtained in 1.2.2., Reconfigure your system/virtual machine to ensure that the number of kernels you use is at least equal to the number of kernels available. Please indicate how you reconfigured the system and show the result of `grep processor /proc/cpuinfo` again.

Answer

Screenshot of the command output:

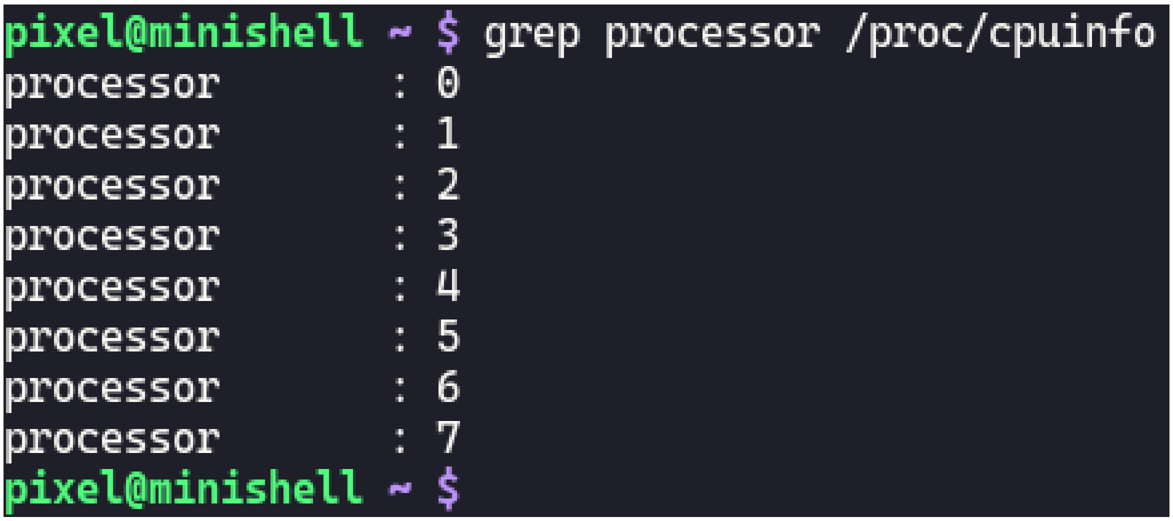


Figure 3: “grep processor /proc/cpuinfo” command

I have 8 processors resulting from the aforementioned **Hyperthreading**.

1.3. Running the Script

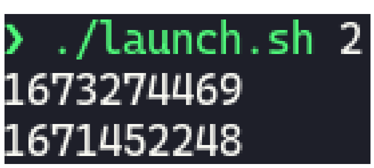
Question

Run the `launch.sh` script several times, changing the value of the parameter from 1 to twice the value you tested in section 1.2.3. Explain the trend of the counters that print the processes `count1` (the first `N` counters shown) as you increase the value of the parameter and relate it to the answers 1.2.2. and 1.2.3. Show screenshots of executions.

Answer

There is a clear trend: the more concurrent executions of the program are created, the more system load is created and thus thus the program `count1` can count up to a smaller number. My device has `8` **threads** in the `4` **physical** cores of the CPU. Depending on the scheduler of the CPU and my personal system configurations, some processes may be prioritized over other ones. Therefore it is very hard to find one clear trend in the executions. Perhaps it is interesting to evaluate what happens when we launch more threads than there are *actual* threads (i.e. CPU cores). Computers can launch many threads at once, certainly many more than `8`. Ideally, every thread would be assigned some core, but in reality is not handled in this way; the process and its threads might switch between different cores during execution, again, related to the scheduler and the specifics of every system.

Screenshot of outputs from `1` to `2N` (`N` being the number of processors on my system).



```
❯ ./launch.sh 3
1667424808
1667213408
1664891094
```

```
❯ ./launch.sh 4
1669883477
1671308336
1670950867
1669988654
```

```
❯ ./launch.sh 5
1658556246
1348726839
1671148369
1349377044
1661141070
```

```
❯ ./launch.sh 6
1582173766
1429910111
1436101266
1435914660
1481520903
1340123208
```

```
❯ ./launch.sh 7
1358910369
1328892165
1391539624
1383421419
1345187805
1361104284
1524286578
```

```
❯ ./launch.sh 8
1284315233
1314379449
1345397587
1322729081
1326050965
1337137763
1333072810
1332105321
```

```
❯ ./launch.sh 9
1130127090
1182919922
1086537436
1202133709
1191876178
1226266755
1158637297
1226799263
1166170225
```

```
❯ ./launch.sh 10
1016892036
1057748483
1064126323
1106643376
997202302
1018961419
1085352234
1104028397
1047989414
1012627497
```

```
❯ ./launch.sh 11
760773617
1110225751
905520143
1127202576
856513404
1184726090
802623874
757225910
956107900
1066407383
1020830450
```

```
❯ ./launch.sh 12
918605447
810485064
960935341
708808525
704679018
878101627
1059791659
928370814
799201641
962028921
967096807
881366710
```

```
❯ ./launch.sh 13
797186873
1061397465
815736871
825583939
755400850
804685841
869385803
766918317
797214783
793478351
649585517
926581142
715499835
```

```
❯ ./launch.sh 14
984418417
956382366
1049111304
879123176
601153961
714175521
610666308
785986805
677283419
613869782
611976158
746027894
653960555
661543756
```

```
❯ ./launch.sh 15
650751512
753549098
630213918
751922419
573119648
841781302
873080747
907613999
560859630
570029008
740250070
696831549
544549724
717888644
806135634
```

```
❯ ./launch.sh 16
558824910
597650457
554831001
640998774
612919578
665167537
693965965
589181346
681160971
591830781
566786113
883171200
730007874
748874919
566540721
887638069
```

These results are taken after several repetitions of the same command, to show the general output. These results vary a lot depending on the system usage and other running applications, as well as depending on how fast the computer is.

Question

Do the numbers that print the `count2` processes (the last `N` counters shown) follow the same trend? Justify the answer.

Answer

The output values for the `count2` command do not vary in any significant way, giving consistent values of around `3700`. This is because the constraint that keeps this value low is the `usleep` call of `1000ms`, and not so much the fact that the system is running many processes simultaneously. As we saw with the `count1` program, which had no constraint and the counter had no slowdowns, the system can count numbers far greater than `1600000000` in some circumstances. The **bottleneck** for `count2` is the delay before every iteration, rather than the number of running processes.

1.4. Execution of the top Command

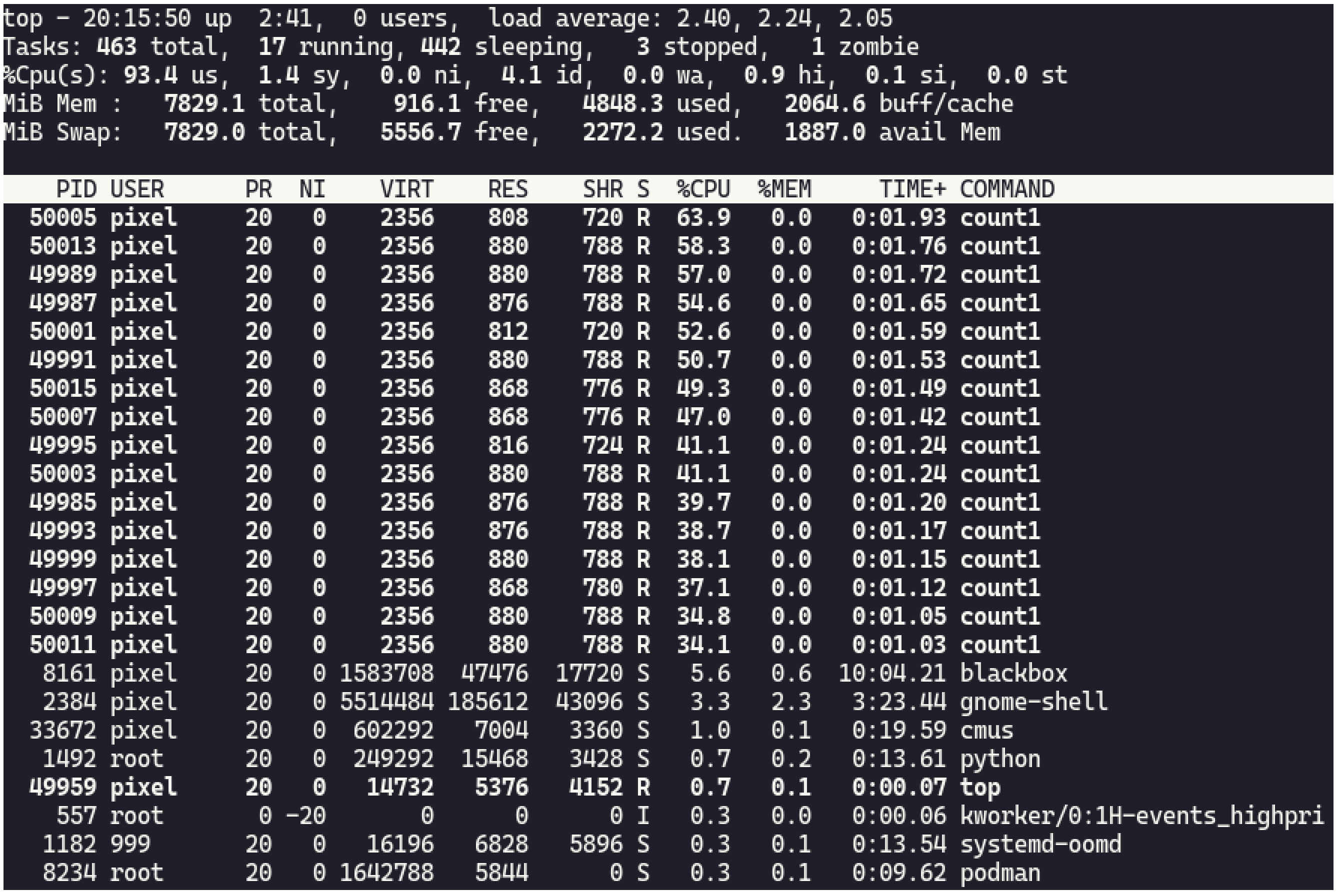
Question

From another window, run the `top` command, which shows information about *CPU* usage and which processes are consuming the most. From the original window, run `./launch.sh 1 20` and look at the information that shows top while launching. What can you conclude?

Answer

By default, the `top` command lists the running processes on the system, with the most CPU-intensive processes on the 'top' of the list. `count1` appears on the very beginning because it is supposed to emulate a heavy task using tons of system resources. `count2` on the other hand is much less CPU-intensive because of the delay introduced before counting and thus does not appear on top of the list. This does not mean that the processes are not createds or that they do not show up on the `top` command, it merely shows the `count2` processes much lower on the list, not visible at a glance at the command.

Screenshot of the `top` command during execution of `./stack 16`.



```
top - 20:15:50 up  2:41,  0 users,  load average: 2.40, 2.24, 2.05
Tasks: 463 total,  17 running, 442 sleeping,   3 stopped,   1 zombie
%Cpu(s): 93.4 us,  1.4 sy,  0.0 ni,  4.1 id,  0.0 wa,  0.9 hi,  0.1 si,  0.0 st
MiB Mem :   7829.1 total,    916.1 free,   4848.3 used,   2064.6 buff/cache
MiB Swap:   7829.0 total,   5556.7 free,   2272.2 used.   1887.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
  50005 pixel     20   0    2356    808    720 R  63.9   0.0   0:01.93 count1
  50013 pixel     20   0    2356    880    788 R  58.3   0.0   0:01.76 count1
  49989 pixel     20   0    2356    880    788 R  57.0   0.0   0:01.72 count1
  49987 pixel     20   0    2356    876    788 R  54.6   0.0   0:01.65 count1
  50001 pixel     20   0    2356    812    720 R  52.6   0.0   0:01.59 count1
  49991 pixel     20   0    2356    880    788 R  50.7   0.0   0:01.53 count1
  50015 pixel     20   0    2356    868    776 R  49.3   0.0   0:01.49 count1
  50007 pixel     20   0    2356    868    776 R  47.0   0.0   0:01.42 count1
  49995 pixel     20   0    2356    816    724 R  41.1   0.0   0:01.24 count1
  50003 pixel     20   0    2356    880    788 R  41.1   0.0   0:01.24 count1
  49985 pixel     20   0    2356    876    788 R  39.7   0.0   0:01.20 count1
  49993 pixel     20   0    2356    876    788 R  38.7   0.0   0:01.17 count1
  49999 pixel     20   0    2356    880    788 R  38.1   0.0   0:01.15 count1
  49997 pixel     20   0    2356    868    780 R  37.1   0.0   0:01.12 count1
  50009 pixel     20   0    2356    880    788 R  34.8   0.0   0:01.05 count1
  50011 pixel     20   0    2356    880    788 R  34.1   0.0   0:01.03 count1
   8161 pixel     20   0 1583708  47476  17720 S   5.6   0.6  10:04.21 blackbox
   2384 pixel     20   0 5514484 185612  43096 S   3.3   2.3   3:23.44 gnome-shell
  33672 pixel     20   0  602292   7004   3360 S   1.0   0.1   0:19.59 cmus
   1492 root      20   0  249292  15468   3428 S   0.7   0.2   0:13.61 python
  49959 pixel     20   0   14732   5376   4152 R   0.7   0.1   0:00.07 top
    557 root       0 -20       0      0      0 I   0.3   0.0   0:00.06 kworker/0:1H-events_highpri
   1182 999       20   0   16196   6828   5896 S   0.3   0.1   0:13.54 systemd-oomd
   8234 root      20   0 1642788   5844      0 S   0.3   0.1   0:09.62 podman
```

Figure 4: Example top command with ./stack 16

Question 2. Memory

We provide you with the `stack.c` program which supports a numeric parameter (1, 2, 3, 4, 5 or 6). Study its source code, compile it, and run it. When running it on your system, the sequence of numbers generated in each case may be different (longer or shorter) than the examples; However, in all cases, the operating system must abort the program (the message Segmentation fault is displayed).

2.1. Stack

Question

In all cases, the program ends up with invalid memory access and the operating system aborts the process. Indicate justifiably at which point in the program this invalid access is caused in each case.

Answer

Here are the reasons why the program (`rec1`, `rec2`, . . . , `rec6`) crashes with a segmentation faults occur.

- **`rec1`**: Reaches a stack overflow somewhere in between 52000 and 53000 recursions. That's why the program lists 52 right before crashing, which happens when the stack size is filled due to the endless amount of recursive calls.
- **`rec2`**: In this case an *int pointer* is initialized with 128 bytes of memory of reserved **on the heap** with `malloc`. In this case the stack overflow occurs sooner than before because the function calls and variable initializations are done **on the stack** of the program.

- **rec3**: This case is almost the same as the last one, but `malloc` reserves `256` bytes of memory. Executables run with a set of stack (static) memory and heap (dynamic) memory. Memory on the heap can exceed and start using resources as needed, while stack memory will only allow for so much memory. That is why after increasing stack memory beyond a certain limit on this recursive function, the program aborts. In summary, this example is no different from before because in both cases the real cause of the stack overflow is the initialization of the `int *` variable and the recursive function call, allocating more or less with dynamic memory has little impact on the abort result. In fact, if you just leave the pointer declaration and remove the `malloc` call (initializing the pointer to `NULL`), the abort sticll happens after printing `26`.
- **rec4**: In this case we are declaring an array of `16 int` variables, which are all allocated on the stack (including the pointer declaration). This causes the program to (once again) overflow, but way faster than before since there are many more variables on the stack than before, and because the function keeps calling itself again and again it never has time to destroy those variables allocated on the stack, making it larger and larger till it grows to its limit (the limit defined by the current heap memory, growing in the opposite direction of the stack).
- **rec5**: this time we are allocating an array of `32` integers. Previously the program crashed right after printing `8` on the console, and this time the program crashes after printing `5`. It does not exactly take half the time to crash (that would be printing `4` on the console), because previously we had `16` integers from the array, plus the pointer to the array. This time we are just allocating an extra `16` integers, thus causing the crash to occur almost (but not quite) half as fast as before.
- **rec6**: this time the crash occurs before the first `1000` recursions (that would be the equivalent of printing `1` in the previous functions), the fastest of all the `rec` functions. In each recursive call we are assigning a new value of a **global int array** of 1500 ints. The program crashes after printing the `2000`th iteration, however at some point we see that the program accesses indexes out of bounds of the array (indexes over `1500`). This is defined to have an undefined behavior, but not necessarily a crash, which explains how the program didn't crash and successfully printed `1600`, `1800` and `2000`. Having this in mind, the program aborts when it tries to access memory outside the bounds of the program.

Question

What's the cause of the time that the program takes to abort is different in each case? What does it depend on?

Answer

Generally, every case has a similar reason for aborting. As I previously mentioned, programs run with a predefined amount of memory and resources to run, with the possibility to dynamically ask the operating system for extra resources the stack and heap memory grow towards one another. Function calls and variable definitions inside them are stored on the stack and freed after the function finishes execution. On the previous examples we saw various cases of stack overflows due to infinite recursions, and the main factor determining how quickly the segmentation faults occurred was the number of variable declarations and initializations inside the recursive call. It is only because of the nature of recursivity that these issues occur, because every time we call the function we allocate on the stack all the local variables in it. In the last case, however, the segmentation fault happens not because of the variable declarations (there aren't any), but rather because we are endlessly accessing and modifying the value of a finite array, so at some point we will try to access points beyond the scope of the array, eventually causing a segmentation fault for trying to access memory outside the memory granted to the running file.

2.2. Code Exercise

Question

Type a program that reads keyboard strings until it detects that the same string has been entered twice. To implement this, you need to use dynamic memory management routines.

Your solution should store strings in a list where each item in the list will store the contents of one string and a pointer in the next item in the list. The list will grow dynamically as you read strings. To check if a string has already been read, you'll need to go through all the positions in the list and see if it already exists.

As a guide, we provide you with the structure of a possible solution (`mem3_base.c`). You can use it as a reference and implement the missing code.

You must provide the source code of the program and, if it works, a screenshot to prove it.

Some remarks on the code to implement:

- This is not the most efficient solution to this problem, but it is the solution we ask you to implement.
- The number of strings to read until a repeat is detected can be arbitrarily large.
- You can assume that the maximum size of each string will be `80` characters.
- The memory must be freed before the program can run dynamics that have been requested.
- To compare strings you must use the `strcmp()` routine.

Answer

The code can be found in the file `main.c`

To compile the code, you can run the following command in the appropriate directory:

```
gcc -Wall -Wextra -Werror main.c -o mem3
```

Note: The compilation flags `-Wall -Wextra -Werror` are optional, used to optimize the code and avoid having unused variables.

To run the code, you can simply run the following command:

```
./mem3
```

Here are some screenshots proving it works:

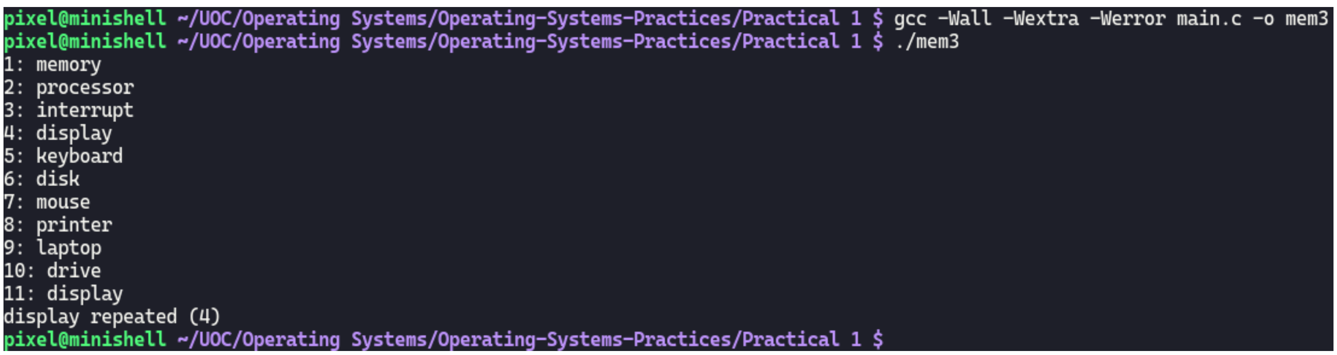


Figure 5: mem3 executable example

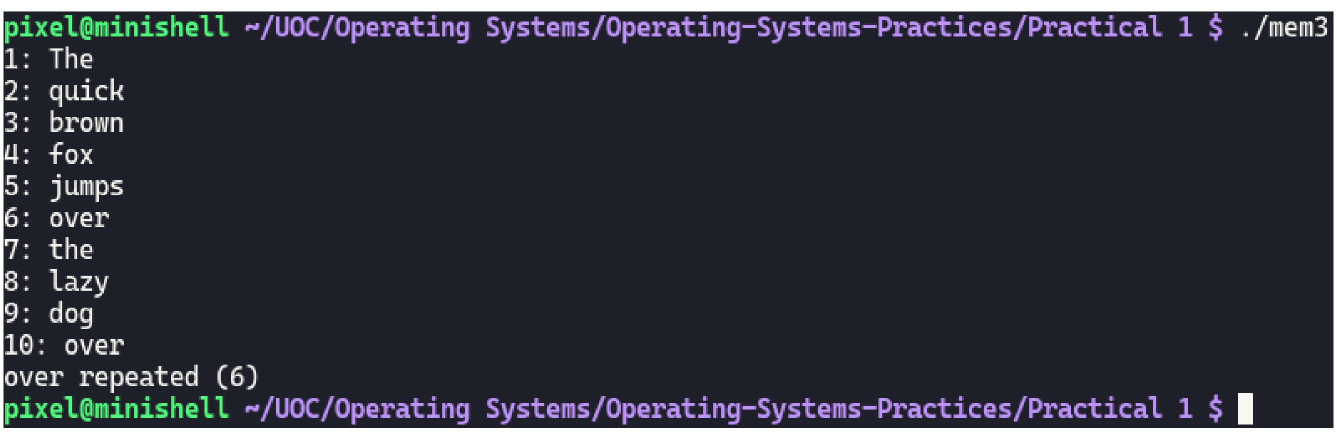


Figure 6: mem3 executable example

```
pixel@minishell ~/UOC/Operating Systems/Operating-Systems-Practices/Practical 1 $ valgrind --leak-check=full --show-leak-kinds=all -s -q ./mem3
1: Somewhere
2: over
3: the
4: rainbow
5: .
6: Way
7: up
8: high
9: .
. repeated (5)
==184816== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
pixel@minishell ~/UOC/Operating Systems/Operating-Systems-Practices/Practical 1 $
```

Figure 7: mem3 executable example

*Note how this third example is run with **valgrind**, a tool that helps check for code errors and leaks. No errors are reported, so we assume that there are **no memory leaks**.*

As an added bonus, the code I created passes the so-called `norminette`, a linter for `.c` and `.h` files I use to keep code clean.

```
pixel@minishell ~/UOC/Operating Systems/Operating-Systems-Practices/Practical 1 $ norminette main.c
main.c: OK!
pixel@minishell ~/UOC/Operating Systems/Operating-Systems-Practices/Practical 1 $
```

Figure 8: norminette passed

Question 3. In and Out

The `args.c` program shows the list of parameters it receives on the command line.

Several examples of execution are attached:

```
[enricm@willy dev]$  ./args
argc = 1
  argv[0]=./args
[enricm@willy dev]$  ./args a1 a2
argc = 3
  argv[0]=./args
  argv[1]=a1
  argv[2]=a2
[enricm@willy dev]$  ./args a1 a2 | wc
argc = 3
  argv[0]=./args
  argv[1]=a1
  argv[2]=a2
      0       0       0
[enricm@willy dev]$  ./args /bin/l*s
argc = 4
  argv[0]=./args
  argv[1]=/bin/less
  argv[2]=/bin/loadkeys
  argv[3]=/bin/ls
[enricm@willy dev]$
```

Figure 9: args Executable

Question

How is it that in the third example `argc` has the value 3 and not 5?

Answer

The correct number of arguments (`argc`) is 3 and not 5 because the command-line interpreter reads the pipe symbol `|` as a special character that, in this case, pipes the output of the command to the left of the pipe to the `stdin` of the command to the right of the pipe. Thus, the only things passed as arguments to the executable `arg` are `a1` and `a2` before the special character appears.

Question

How is it that in the fourth example `argc` has the value 4 and not 2?

Answer

The number of arguments is 4 and not two in this case because the command-line interpreter (shell) treats the star symbol `*` as a special character. This symbol expands the command-line arguments with new entries that match the expression. In the example the wildcard will look for any occurrence of `/bin/ls` where anything can appear between `l` and `s` (or nothing at all). In the screenshot we can see that the user that ran the command only had three files (command binaries) that matched what the wildcard was looking for, namely:

`/bin/less`

```
/bin/loadkeys
/bin/ls
```

Thus, the shell internally expanded the wildcard to the said matches, effectively increasing the number of arguments passed to the `args` executable, as follows:

```
./args /bin/less /bin/loadkeys /bin/ls
```

Question

Replace the two occurrences of `stderr` with `stdout` in `args.c`. Compile the program and run the third example again. Explain what the new observed behavior is due to.

Answer

Code to replace occurrences of `stderr`:

```
sed -i 's/stderr/stdout/g pr1so/args.c'
```

New output of third example:

```
pixel@minishell ~ $ ./args a1 a2 | wc
      4       6      52
pixel@minishell ~ $
```

This happends because pipes (`|`) will take the standard output (`stdout`) *only* and use it as standard input (`stdin`) for the command that follows the pipe. Initially the code printed on the standard error (`stderr`), which is usually used, as the name suggests, for printing error messages. Messages printed on the `stderr` will be displayed on the console but **will not be piped onto the next command**. That is why when printing on the `stdout` instead of on the `stderr`, the entire output of our `args` command is passed onto the `wc` command. This second command counts words, lines and characters from the standard input (`stdin`) or from a file (see `man wc`). That explains the three zeroes from example 3 when using `stderr`, as nothing was being passed as input.

Summary

All in all, this first practical was much more challenging than expected, and it helped clarify many aspects about Operating Systems I previously ignored.

November 5th, 2022