OS DRY 2

22/05/2021

Dani Bondar - 206560856

Gur Telem - 206631848

Part 1.

(1) By itself, the yes command outputs 'y' or whatever is specified as an argument, followed by a newline repeatedly until stopped by the user or otherwise killed; when piped into a command, it will continue until the pipe breaks (i.e., the program completes its execution).

If yes get an argument, then it is the argument that will be repeated rather than 'y'.

Kinda like an echo if it was stuck in an infinite loop.

(2) We use the yes with '' to send newlines to the make oldconfig command. This will discard the need for a user interaction with the process, choosing the default choices. In most scripts, merely entering a newline chooses the option capitalized. e.g. are you should (Y/n): simply hitting enter will choose y. While running the command make oldconfig without the yes it did the same this as if we used yes. I assume the script has changed or some caching has taken place and it should have prompted the user for input.

(3) the GRUB_TIMEOUT parameter indicates the amount of time in seconds the grub process waits for the user before automagically choosing the default boot option.

The pros of increasing the timeout are that now the user has more time to choose to pick a different boot option or to modify parameters in the grub, e.g. which version of the kernel to boot with.

The cons are that most of the time the user wants the default options (if not, then the user should change the default to something else). Thus, the boot time will either increase or the user will need to manually interact with the grub to boot faster. I usually prefer a timeout of 1 or 2 seconds for a normal installation and around 5 seconds for a virtual machine (for a virtual machine, you need to capture the keyboard before being able to interact with the vm so I need more time. Also, it's more rare to reboot a vm unless you actually play with the kernel because you usually suspend when you're done with it).

(4) TL;LD: User space and kernel space has a different set of functions. Neither space can use a function from the "other side".

The execve() is part of the standard c library and its purpose is to envelop the calling of the syscall command - which transitions to the kernel space and execute the do_execve() function which then dose the actual switch of the programs. the run_init_process() function is located in the kernel space and isn't aware to the C standard library (and also doesn't need to).

(5) The syscall () function envelops the calling of the syscall assembly instruction. It receives at list one argument (the number of the system call) and any number of optional arguments, depending on the system call - for example, the added parameters of the sys_write system call are the address and length of the string. It is most commonly used to invoke syscalls that don't have a pretty wrapper.

The syscall () function moves the system call's number (first parameter) to the RAX register and if any other parameter were added it passes them via predefined registers. It calls the syscall assembly command and then waits to the outcome of the system call. On return, the function sets back the registers and saves the result in register RAX.

The implementation of syscall () is in the glibc library. It is a userspace function after all. Its prototype is included with unistd.h.

(6) It calls for the system call getpid() which corresponds to the number 39. A more readable code may look like this:

```
int main(){
        printf("sys_hello returnd %ld\n", getpid());
        return 0;
}
```

Since the answer needs more content and I think this is explanation enough, not sure what else I need to add that isn't here. Maybe I can mention that the getpid is assigned to syscall 39 on line 48. I really don't no what else should be added here. Maybe some Lorem Ipsum as a space filler: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc ac scelerisque nunc. Interdum et malesuada fames ac ante ipsum primis in faucibus. Donec rutrum aliquet mauris vel congue. Etiam quis tortor nulla. Suspendisse egestas hendrerit ipsum ac blandit.

(7) There is no file called test.c. The file provided was test.cxx. And the file tests whether the first two new syscalls, get_weight and set_weight work as expected for valid input as well as the default value for a process's weight. As part of the test, it uses the wrappers provided to us. The test first checks the default weight value, then sets the weight to 5 and test the change took. It also tests the return value of the set_weight function to be 0 for the valid parameter 5.

Part 2.

(1) Answers 1 and 5 are correct. E

(a) Every scheduling method that doesn't uses preemption can suffer from the conveyer-belt effect - the reason is that if a long task is first one to enter the system, then it blocks all of the other processes that can come after it; i.e. there is a conveyer belt effect.

Because the FCFS algorithm is a non-preemption algorithm then is also suffers from the conveyer-belt effect

For sane values of RR, there's no convoy effect.

(b)
  (i) All tasks arrive at the same time
  (ii) Their runtime is known from the beginning
  (iii) A single-core system while all the jobs are serial (meaning they run from beginning to the end without using IO or pausing).

(2) We'll prove that $Q_i = \left(\frac{W_i}{\sum_j W_j}\right) \cdot t$ with $t = sched\_latency$.

Proof. It is given that $\sum_i Q_i = t$ (meaning the total of the quantums is exactly the sched_latency and is constant).

Since a process runs once per epoch and they have the same vtime after each epoch, then the $\frac{c}{W_i} \cdot Q_i$ is constant (i.e. the virtual time we add per epoch is the same).

So we can mark $\forall i : \frac{c}{W_i} \cdot Q_i = v$. So we get $Q_i = \frac{W_i}{c} v$. And from that we conclude

$$\sum_i Q_i = \sum_i \frac{W_i}{c} v = \frac{v}{c} \sum_i W_i = t \Longrightarrow v = \frac{t \cdot c}{\sum_i W_i}$$

And not, we can substitute v

$$Q_i = \frac{W_i}{\not{c}} \cdot \frac{t \cdot \not{c}}{\sum_i W_i} =$$

$$Q_i = \frac{W_i}{\sum_i W_i} \cdot t$$

Q.E.D □

(3) CFS improvements
  (a) Like discussed in the tutorial, lets say that we have two tasks (A and B). Task B uses some lengthy IO operation (10 sec). according to Hila's suggestion, task B will rejoin the tree with the same virtual-time. This means that now task B will run 10 sec, while task A will "starve". So Hila's suggestion will promote long IO operation.
  (b) Using a list DS will increase the time complexity from $O(\log n)$ to $O(n)$. Getting the task with the minimal VT will take $O(1)$, but after every insertion to the DS we will need to go through all of the tasks in the list to find where to insert the task with the updated value.
  (c) In case of large number of tasks in the system, with lets say same priority, the quantum for each task will be relatively small. this will result in increased number of context switches - which means the percentage of time been spent on context switch will increase substantially and perhaps will overshadow the time been spent on actual calculations. Or in other words, a potential big increase in the overhead.
  (d) Will result in increase epoch time and a potential starvation of tasks with low priority. Will also cause the processes to feel less responsive due to the longer epoch time.
  (e) Will cause the difference between priorities to be less edivent on top of non linear.
  (f) Potential stravation of old/existing tasks due to new tasks "cutting in line" (this method is aka an Israeli Queue).

(4) The user can use large number of threads in his/hers program. Because all of the threads use the same memory space they will receive a high weight score via the new scoring method, which result in prioritization of this program over the other tasks. So instead of changing the weight of the task via a singular way (the value of the NICE parameter), the user is now able to bypass the kernel and change the effective program's priority by simply splitting the program into multiple threads.

In addition, assuming a user program has two threads, the core would be dedicated to that user program until one all the threads but one finish. This is because after a task (i.e. a thread) finishes, we will run the other threads (even if they already ran during this epoch. Assuming they're in a ready state and alive). This effectively locks a core to a single user program if it has two or more threads. Thus, starving ALL other tasks running on the same core (i.e. CPU).

(5)
  (a)

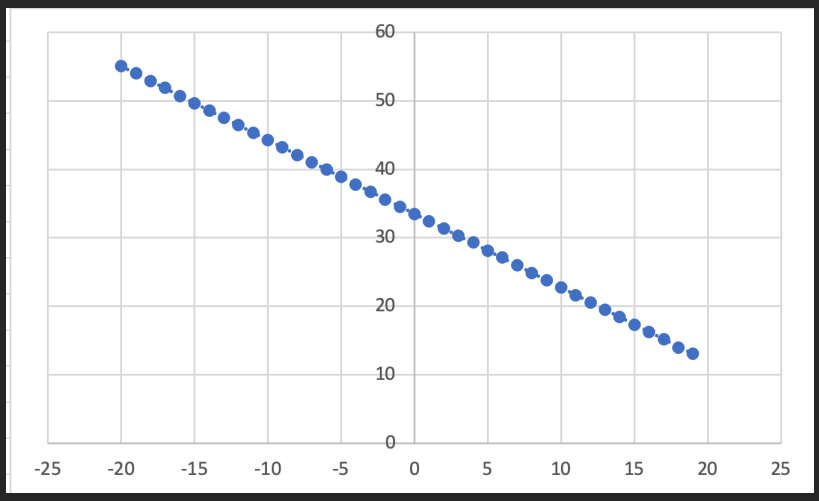

FIGURE 0.1. $f\,(nice) = \log_{1.23} weight\,(nice)$

(b) The connection between the nice value and the weight value is

$$\log_{1.23}(weight) = -1.078 \cdot NICE + 33.486$$

Meaning, the weight decreases exponentially with the nice value.