1. **Task 2:** Write a c program to find all your current shell's processes and issue a kill to each one. You may want to create some background jobs to give a good demo here.

**Answer:**

For the purpose of this task, I started out by writing a C program that finds all the current shell’s running processes, stores their PIDs into a file, then reads the file line-by-line while issuing a kill to each one of the processes in the file. Now, in order to provide a good demo for my program, I started out by creating 15 different background jobs as the screenshot below shows.

A screenshot of a computer

Description automatically generated

Furthermore, to make sure that the background jobs have been created and are running in the background, I ran the “jobs” command and the screenshot below shows the result of running this command.

A screenshot of a computer

Description automatically generated

Then, I went ahead and ran the C program I wrote and it indeed did what it was supposed to do, which is kill all the jobs/processes running in the background in my current shell. The screenshot below shows the result of running my program.

A screenshot of a computer

Description automatically generated

Finally, I ran the “jobs” command once more to make sure that all of the background jobs/processes within this shell have been killed and the screenshot below shows the results.

A screenshot of a computer

Description automatically generated with medium confidence

Therefore, as we can see from the screenshot above, the program I wrote turned out to be rather successful as it did exactly what it was supposed to do, which is killing all the current shell’s processes.

1. **Task 3:** Write a c program to demonstrate whether malloc and calloc differ on your system.

**Answer:**

While doing my research on the differences between malloc and calloc, I was able to find plenty. Some of these differences are trivial and can be observed while writing code while others need a little bit of plumbing into the internals of the operating system to notice them. Nonetheless, here are some of the differences I was able to find:

* **Syntactic differences:** when working with malloc and calloc, noticing the syntactic differences between the two is rather trivial. While both calloc and malloc return a void pointer to the location in memory where the allocated memory resides, malloc takes in one parameter as input whereas calloc takes in two parameters as input. This difference can be seen below:
* int \*pointer1 = (int \*) malloc(10 \* sizeof(int));

Or according to the documentation: void \*malloc(size\_t n);

* int \*pointer2 = (int \*) calloc(10, sizeof(int));

Or according to the documentation: void \*calloc(size\_t n, size\_t size);

* **Time efficiency:** according to multiple resources, malloc tends to be quicker/faster than calloc when it comes to the time it takes the function to execute (the function’s runtime)[[1]](#footnote-1).
* **Naming:**
* malloc: stands for memory allocation.
* calloc: stands for contiguous allocation.

As we can see above, the names here are rather important. This is because malloc implies that the reserved block of memory is not necessarily contiguous whereas calloc implies that the reserved block of memory is always contiguous. However, after doing some research, it appears that this is heavily dependent on the operating system and is not always the case.

* **Memory initialization:** the key, and most notable, difference between malloc and calloc comes from the fact that one of them initializes the memory block it allocates whereas the other one does not. Now, this does vary depending on the operating system and the machine being used, but here is what I got on my particular machine running Ubuntu 18.4:
* malloc, on my particular machine, does not initialize the allocated/reserved memory block. That is, on my machine, malloc returns a pointer to an uninitialized block of memory (which usually contains garbage values).
* calloc, on my particular machine, does initialize the reserved memory with zero-values. That is, calloc returns a pointer to reserved block of memory whose values were initialized to zero (performs zero-initialization).

With that said, please note that the c program that shows these differences is in the Task\_3 > Task\_3.1 folder so please feel free to check it out. Furthermore, below is a screenshot of the results I obtained after running this program:

Text

Description automatically generated

* **Lazy vs Immediate memory allocation:** while doing my research on this particular subject, there seemed to be a lot of disagreement within the OS community on which of the two methods performs lazy allocation of memory and which performs immediate allocation of memory. From what I read; this difference seems to be highly dependent on the machine as well as the operating system it is running rather than being set-in-stone.With that said, I quickly became very curious to see what the difference is, if any, on my particular machine which runs Ubuntu 18.4. To accomplish this, I wrote two different programs: the first declares a pointer, mallocs memory space for it, then sleeps the system whereas the second declares a pointer, callocs memory space for it, then sleeps the system. These two programs can be found in the folders titled Task\_3.2 and Task\_3.3 (under the Task\_3 folder) so please feel free to check them out to learn more about my code. Now, after running the two programs, one after the other, here is what I found:
* As the screenshot below shows, malloc performs lazy memory allocation. This is apparent because for task\_3.2.out (which is the output file for the program that creates a pointer and mallocs space for it), we can see that the value of VIRT (which is all the allocated space be it used or just reserved) >> RES (the subset of the virtual address space representing the non-swapped physical memory a task is currently using) = USED (non-swapped out physical memory that the process is using). Therefore, this implies that malloc had allocated a block of memory but this block of memory is yet to be touched not to mention that no memory pages have been created yet. This will change once we start adding values to that block of memory but as far as malloc goes, the block of memory had been allocated but no memory pages have been created yet. Therefore, this verifies that malloc does indeed perform lazy allocation of memory.

A screenshot of a computer

Description automatically generated with medium confidence

* As the screenshot below shows, calloc performs immediate memory allocation. This is apparent because for task\_3.3.out (which is the output file for the program that creates a pointer then callocs space for that pointer), we can see that the value of VIRT ≈ RES = USED. Therefore, this implies that calloc had allocated memory space for the process, initialized that memory space, and created memory pages for it. Hence, this verifies that calloc does indeed perform lazy allocation of memory.

A screenshot of a computer

Description automatically generated

Please note that this might be specific to my machine and operating system. Again, this behavior might vary from one machine to another and from one operating system to another. With that said, as far as my machine goes, this difference in behavior is true and is as indicated above.

1. **Task 4:** Write a c program to show whether malloc has lazy allocation, or immediate allocation, of physical memory/pages. Can you determine whether pages are allocated as a multiple of some fixed heap-expansion size?

**Answer:**

In order to demonstrate whether malloc has lazy or immediate allocation of physical memory/pages, I had to split up this task into two smaller ones: Task\_4.1 and Task\_4.2.

Now, in Task\_4.1, I basically wrote a main method in which I initialize a pointer of type char, malloc space for that pointer, then sleep the system (the code can be found in Task\_4 > Task\_4.1 folder so feel free to give it read). Note that here I never actually use that malloced memory space nor do I write to it or even attempt to access it, therefore, if malloc does indeed have lazy memory allocation then the values of RES (how much actual physical memory the process is consuming) and USED (the non-swapped physical memory a task has used, RES, plus the non-resident portion of its address space, SWAP) must be less than VIRT (The total amount of virtual memory used by the task. It includes all code, data and shared libraries plus pages that have been swapped out and pages that have been mapped **but not used**). Now, after running top to investigate the previously mentioned values, I observed that RES = USED <<< VIRT for this process, which indicates that malloc does indeed perform lazy allocation of memory/pages. The screenshot below further demonstrates my observations:

A computer screen capture

Description automatically generated with medium confidence

Now, in Task\_4.2, I basically wrote a main method in which I initialize a pointer of type char, malloc space for that pointer, assign a value to each index of memory that the pointer maps to (using a for loop), and finally sleep the system (the code can be found in Task\_4 > Task\_4.2 folder so feel free to give it read). Note that now I do access/use the malloced memory/pages as I write to just about every single index in that malloced memory. Now, if malloc does indeed perform lazy allocation of memory/pages, then we would expect that values of USED, VIRT, and RES to be almost equal because the memory has been accessed and populated and therefore malloc will have to actually touch that memory. With that being said, running top on this process did indeed give the expected results (RES = USED ≈ VIRT) hence further solidifying that malloc has lazy allocation of memory/pages. The screenshot below further demonstrates this fact:

Graphical user interface

Description automatically generated with medium confidence

Finally, by looking at the VIRT values in the screenshot above, we can see that all of these values are divisible by 4. Therefore, it would be safe to assume that the allocated virtual memory for each one of the processes above is some multiple of 4 thus indicating that the fixed-heap expansion should be some multiple of 4 (which comes from the fact that page sizes are 4k).

1. **Task 6:** Start with this program that takes in a command and executes it calling execv/execvp/execvpe, i.e. that creates a basic shell program. Change the program so it takes two commands at a time, then executes them both, serially as background jobs. Next, change it so it executes both commands concurrently in the background. Search google to discover what you might use to execute these jobs in parallel, if you know you have two cores/processors?

**Answer:**

The code to create a shell that takes in two commands at a time, then executes them both serially as background jobs can be found in the Task\_6 > Task\_6.1 folder (please feel free to check it out for more detail on my code). Now, after running that code on my virtual machine (which runs Ubuntu 18.24), the two commands were indeed being executed serially, one by one, as background jobs and the screenshot below proves the correct (expected) behavior of my code:

A screenshot of a computer

Description automatically generated with medium confidence

The code to create a shell that takes in two commands at a time, then executes them both concurrently in the background can be found in the Task\_6 > Task\_6.2 folder (please feel free to check it out for more detail on my code). Now, after running that code on my virtual machine (which runs Ubuntu 18.24), the two commands were indeed being executed concurrently in the background (as child 2 did not have to wait for child 1 to finish execution before it started to execute, or vice versa, but instead, both children were started/started working at the same time) and the screenshot below proves the correct (expected) behavior of my code:

A screenshot of a computer

Description automatically generated with medium confidence

Now, regarding what might be used to execute these jobs in parallel (perhaps on different cores/processors), there appears to be a lot of different options that we can choose from, each with its pros and cons obviously.

* The first tool we can use to execute shell commands in parallel is the GNU Parallel tool. Simply put, GNU Parallel can split a given input and pipe it into commands in parallel.[[2]](#footnote-2)
* We can also use Bash’s basic parallelization tools (& and wait) however those are not up to bar when it comes to more complex problem.
* Using Bash subshells which allow us to run parts of a script in parallel.[[3]](#footnote-3) In this case, the subshells will let the script do parallel processing, thus executing tasks in parallel.

1. https://cs-fundamentals.com/tech-interview/c/difference-between-malloc-and-calloc [↑](#footnote-ref-1)
2. https://stackoverflow.com/questions/5547787/running-shell-script-in-parallel [↑](#footnote-ref-2)
3. https://tldp.org/LDP/abs/html/subshells.html [↑](#footnote-ref-3)