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, memory pages are expanded by multiples of the page size which is 4k. However, this may not be an exact value hence why we see slightly weird values in the screenshots above.

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.[[1]](#footnote-1)
* 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.[[2]](#footnote-2) In this case, the subshells will let the script do parallel processing, thus executing tasks in parallel.

1. https://stackoverflow.com/questions/5547787/running-shell-script-in-parallel [↑](#footnote-ref-1)
2. https://tldp.org/LDP/abs/html/subshells.html [↑](#footnote-ref-2)