CS236 Spring 2025

Lab Quiz 3

4 questions, 20 marks

Instructions

Before you begin, please check that you can see the following files on the Desktop:

- This question paper `labquiz3.pdf`, with 4 questions
- A tarball `labquiz3_code.tgz`
- The OSTEP chapter on pthreads API.

Untar the code file as shown below:
`tar -zxvf labquiz3_code.tgz`

You will now find a folder `labquiz3_code` in the same directory, which has the following template code:

- For Q1, you are given `q1/sequence.c`
- For Q2, you are given `q2/goldbach.c`
- For questions 3 and 4, you are given xv6 tarball cs236-labquiz3-xv6-public.tgz, which has all the necessary changes needed for both questions.

Now, create your submission folder titled `submission_rollnumber` in the Desktop directory, where you must replace the string `rollnumber` above with your own roll number (e.g., your directory name should look like `submission_12345678`). Place the files you wish to submit in this folder, in separate subdirectories (named `q1`, `q2`, `q3`, `q4`) for each question. When you finish the quiz, please ensure that the submission folder has all your solution code in the following directory structure:

- For Q1, you must submit `q1/sequence.c`
- For Q2, you must submit `q2/goldbach.c`
- For questions 3 and 4, you must submit only the file `vm.c`, and not the entire xv6 folder. You must solve both questions separately, and submit `q3/vm.c` for Q3, and `q4/vm.c` for Q4.

Submission Instructions

1. Once you are ready to submit, please run the command `check` from your terminal. This command will create a tarball of your submission folder. If the folder is not in the proper place or is empty, this script will throw an error. In that case, please fix the errors and try again.

2. Next, call one of the TAs and ask them to run the `submit` command from your terminal. The TA will enter the password to upload your submission tarball to our remote submission server. Please note that we will only be grading the files that are submitted in this manner. So please ensure that you submit the correct files.

3. **IMPORTANT NOTE: Please ensure that you are submitting the correct files with the correct filenames. Test your code properly before submission. Strictly NO code changes will be allowed after the exam.**

Instructions related to xv6

- You are provided with an xv6 source directory that includes a simple test case in `simple.c`, and several other test cases for the other questions. All these test cases have already been added to the Makefile given to you.
- For each question on xv6, create a separate new xv6 directory to preserve the unmodified original copy, and make changes to this new directory.
- For both xv6 questions, we have made all the changes needed to add the system calls. You only need to write code in `vm.c` and submit this file separately for each question. You must not modify any other file.
- To compile and run xv6 code, run `make`, followed by `make qemu` or `make-qemu-nox`.

Questions

1. **[4 marks]** In this question, you are given the file `sequence.c`, which creates N=3 threads. Each thread i prints the string "I am thread i" to the screen in an infinite loop. You must add synchronization between the threads using locks and conditon variables so that the threads print one after the other in the order 0, 1, 2, 0, 1, 2, ... You must submit your final code in the file `sequence.c`.

   Please note that threads must NOT simply do busy-waiting, but must wait in a condition variable queue until their turn to print comes. We will read your code to check that you have indeed achieved synchronization using one or more condition variables.

   A sample execution of the code is shown below.

```
I am thread 0
I am thread 1
I am thread 2
I am thread 0
I am thread 1
I am thread 2
I am thread 0
I am thread 1
I am thread 2
...
```

2. **[6 marks]** In this question, we will consider a program with a long computation, and improve its run time by using multi-threading, in order to understand how real systems parallelize and speedup compute-intensive code.

   You are given a program `goldbach.c` that verifies the Goldbach conjecture up to N=100K numbers. The Goldbach conjecture states that every even number greater than 2 can be expressed as a sum of two prime numbers. The program given to you does the following. It first computes all primes up to N using the "Sieve of Eratosthenes" algorithm, and stores this information in a boolean array `primes`. That is, `primes[i]` is true if i is prime, and false otherwise.

   Now the program begins its task of verifying the Goldbach conjecture. It iterates over all even N, and counts the number of "Goldbach pairs", i.e., primes p_1 and p_2 that add up to N. It stores this count in an array `gb_count`. The value of `gb_count[i]` is the number of prime pairs that add up to N for even i, and is 0 for odd i. The program computes the time taken for this calculation of the counts, by recording timestamps at the start and end of this calculation, and prints out the total time elapsed. Most systems should take a few seconds to finish this calculation. Finally, the program writes the values of the number of prime pairs to a file `output.txt` so that we can verify the conjecture.

   Now, your job is to speed up this calculation of the number of Goldbach pairs for each N using multithreading. Your code must create 10 threads, assign different ranges of N to these threads, so that the threads can calculate the number of Goldbach pairs for different values of N in parallel. You must create threads after we have recorded the starting timestamp, and the threads must finish their calculation and join before we record the ending timestamp. In this way, we can measure the amount of time taken to compute the number of Goldbach pairs in parallel across 10 threads, and compare it with the time taken in the single-threaded program.

   The calculation of the number of Goldbach pairs for each value of N should be done by exactly one thread. You can distribute the work between the threads in any way you see fit. You must only parallelize the filling up of the `gb_count` array, and you need not parallelize the calculation of the primes itself (that happens before the timer is started). You can move around the code you have to parallelize to a separate function, and provide it as the start function for the various threads. You must provide suitable arguments to the threads so that each thread knows which range of values of N it must compute on. You may need to make some of the data structures like the `primes` and `gb_count` arrays as global variables so that they can be accessed easily by all threads. Think carefully on whether you need locking if the different threads access different parts of the array corresponding to different ranges of N. Please do not tamper with other parts of the code not relevant to you, like the time calculation.

   We expect you to submit a parallel version of the same program `goldbach.c` which runs a few times faster than the single-threaded version. A sample execution of both the single threaded code given to you, and a multi-threaded version you must write, and shown below. Note that we have to use the "-lm" flag when compiling to use the math library, and "-lpthread" flag to use the pthread library. We have also given you the output your code must generate in `expected-output.txt`.

   Single threaded execution:

```
$ gcc goldbach.c -lm
$ ./a.out
Computed primes upto 100000, count = 9592
```

```
elapsed time: 3448984 microseconds
$ tail output.txt
99982 608
99984 1216
99986 603
99988 736
99990 1855
99992 638
99994 651
99996 1303
99998 605
100000 810
$ diff output.txt expected-output.txt
$
```

Multi-threaded execution:

```
$ gcc goldbach.c -lpthread -lm
$ ./a.out
Computed primes upto 100000, count = 9592
elapsed time: 931387 microseconds
$ diff output.txt expected-output.txt
$
```

Note that in the solution written by us, the multi-threaded version with 10 threads runs about 4 times faster, and produces identical output, i.e., correctly computes the number of Goldbach pairs. The exact run times and speedup you may see may differ.

3. **[4 marks]** In this question, you will add the following two memory-related system calls to xv6. The plumbing required to execute the system calls has already been done for you, and the test cases have also been provided, along with an updated Makefile. You only need to make changes to, and submit, `vm.c`. The system calls you must implement are as follows.

   - `getptsize` returns the size of the page table of the process in terms of pages. That is, it returns the number of pages the OS allocates to the page table of the process. This count includes the outer page directory, as well as the inner page table pages allocated, to store all page table entries corresponding to the user and kernel parts of the virtual address space.
   - `getnumpages` returns the number of pages in the user part of the virtual address space of the process. This count must include all pages for the user code, stack, guard page, and heap.

   Hints:

   - You can get a pointer to the PCB of the current process as follows.
     `struct proc *p = myproc();`
     From here, you can access all information about the process, i.e., `p->pgdir` is the pointer to the page directory, and so on.
   - The page directory of a process contains page table entries corresponding to the inner page table pages. You can access these entries using an array notation, e.g., `pgdir[i]`.
   - You can check if a page table entry is present/valid or not by checking if the PTE_P flag is set. You can see an example of this usage in other functions like `walkpgdir`.
   - You can look through other functions in `vm.c` to understand how to implement these system calls, but you must only add code to the functions corresponding to these system calls that are given to you at the end of `vm.c`.

   You are given two test cases, and a sample execution of your completed code should look like this. We will test your code with other test cases beyond those provided to you here.

```
$ test-ptsize
getptsize 66
getptsize 66
getptsize 69
$ test-numpages
getnumpages 3
getnumpages 6
getnumpages 3078
```

4. **[6 marks]** In this question, you will be implementing a simplified IPC mechanism of shared memory between parent and child processes in xv6. The parent process memory maps a shared page into its virtual address space, and the child process inherits this page during fork. The shared page maps to the same physical memory in both parent and child processes, unlike other pages in the virtual address space that map to separate physical memory across both processes. The parent and child can then read and write into this shared page and see each other's changes.

   This mechanism is implemented via the following system calls. The scaffolding for the system calls is already provided in the xv6 code given to you. You must fill in the code for these system calls in `vm.c`. You can also modify any other functions as required in this file.

   - `mapshared()` maps one page at the end of the user virtual address space of the parent process, and returns the starting virtual address of the new shared page. You may assume that the parent does not perform any `sbrk` system calls while using the shared page, so that the shared page always remains at the end of the address space. This shared page must be identified by a separate flag in the page table entry. We have already defined the flag `PTE_S` for you in `mmu.h`, and you must tag the shared page with this flag in the page table.
   - `getshared()` returns the starting virtual address of the shared page that is already present in the address space of a process. You may assume that there is only one shared page in the address space, and return the address of the page that has the shared flag set.
   - `unmapshared()` unmaps the shared page from the address space of the process. It must return a negative value if there was some error during the unmap, and a non-negative value otherwise. You may assume that the parent process that mapped the shared page will invoke this system call to unmap the shared page before either parent or child exits. Once the shared page is unmapped, it must not be usable from both the parent and child processes.

   To correctly implement these system calls, you may have to change other functions in `vm.c` besides the code invoked by the system call itself. For example, you may need to change the code of `copyuvm` so that the child maps the same physical page as the parent for the shared page.

   You may also need to change the code in `deallocuvm` so that the shared pages are not freed up twice in the parent and child. Of course, our test cases try to ensure that both parent and child exit only after the shared page is unmapped, by using sleep in the parent and child processes. But these sleeps may not work sometimes, and a process may exit before unmapping the shared page, resulting in a shared page being freed up multiple times from the parent and child memory images. You will see an error like this in such cases: `lapicid 0: panic: kfree`. Your code should gracefully handle such cases, and not panic.

   You are given two test cases, and a sample execution of your completed code should look like this. We will test your code with other test cases beyond those provided to you here.

```
$ test-shared1
child 42
parent 53
$ test-shared2
child 42
parent 53
child again 43
parent again 54
```