COMP3230 Principles of Operating Systems
Programming Assignment One

Due date: Oct. 17, 2024, at 23:59

Total 13 points – Release Candidate Version 2

Programming Exercise – Implement a LLM Chatbot Interface

Objectives

1. An assessment task related to ILO 4 [Practicability] – "demonstrate knowledge in applying system software and tools available in the modern operating system for software development".
2. A learning activity related to ILO 2a.
3. The goals of this programming exercise are:

- To have hands-on practice in designing and developing a chatbot program, which involves the creation, management and coordination of processes.
- to learn how to use various important Unix system functions:
  - to perform process creation and program execution
  - to support interaction between processes by using signals and pipes
  - to get the processes's running status by reading the /proc file system
  - to configure the scheduling policy of the process via syscall

Tasks

Chatbots like ChatGPT or Poe are the most common user interfaces to large language models (LLMs). Compared with standalone inference programs, it provides a natural way to interact with LLMs. For example, after you enter "What is Fibonacci Number" and press Enter, the chatbot will base on your prompt and use LLM to generate, for example, "Fibonacci Number is a series of numbers whose value is sum of previous two…". But it's not the end, you could further enter prompt like "Write a Python program to generate Fibonacci Numbers." And the model would continue to generate based on the previous messages like "def fibonacci_sequence(n): …".

Moreover, in practice, we usually separate the *inference process* handles LLM from *main process* that handles user input and output, which leads to a separable design that facilitates in-depth control on inference process. For example, we can observe the status of the running process via reading the /proc file system or even control the scheduling policy of the inference process from the main process via relevant syscall.

Though understanding GPT structure is not required, in this assignment, we use Llama3, an open-source variation of GPT and we provide a complete single-thread LLM inference engine as the startpoint of your work. You need to use Unix Process API to create inference process that runs LLM, use pipe and signal to communicate between two processes, read /proc pseudo file system to monitor running status of the inference process, and use sched syscall to set the scheduler of the inference process and observe the performance changes.

Acknowledgement: The inference framework used in this assignment is based on the open-source project llama2.c by Andrej Karpathy. The LLM used in this assignment is based on SmolLM by HuggingfaceTB. Thanks open-source!

Specifications

a. Preparing Environment

Download start code – Download `start.zip` from course's Moodle, unzip to a folder with:

```
start
├── common.h          # common and helper macro defns, read through first
├── main_[UID].c      # [your task] template for main process implementation
├── inference_[UID].c # [your task] template for inference child process implementation
├── Makefile          # makefile for the project, update [UID] on line 5
├── model.h           # GPT model definition, modification not allowed
└── avg_cpu_use.py    # Utility to parse the log and calculate average cpu usage
```

Rename `[UID]` in `inference_[UID].c` and `main_[UID].c` with your UID, and open `Makefile`, rename `[UID]` at line 5 and make sure no space left after your uid.

Download the model files. There are two binary files required, `model.bin` for model weight and `tokenizer.bin` for tokenizer. Please use following instructions to download them:

```
make prepare # will download model.bin and tokenizer.bin if not existed
# or manually download via wget, will force repeated download, not recommended
wget -O model.bin https://huggingface.co/huangs0/smollm/resolve/main/model.bin
wget -O tokenizer.bin https://huggingface.co/huangs0/smollm/resolve/main/tokenizer.bin
```

Compile and run the inference program. The **initial** `inference_[UID].c` is a complete single-thread C inference program that can be compiled as follows:

```
make -B inference # -B := --always-make, force rebuild
# or manually
gcc -o inference inference_[UID].c -O3 -lm # replace [UID] with yours
```

Please use `-lm` flag to link math library and `-O3` flag to apply the best optimization allowed within C standard. Please stick to `-O3` and don't use other optimization level. Compiled program can be executed with an integer specifying the random seed and a series of string as prompts (up to 4 prompts allowed) supplied via command-line arguments, aka `argv`:

```
./inference <seed> "<prompt>" "<prompt>" # prompt must quoted with ""
# examples
./inference 42 "What's the answer to life the universe and everything?" # answer is 42!
./inference 42 "What's Fibonacci Number?" "Write a python program to generate Fibonaccis."
```

Upon invocation, the program will configure the random seed and begin sentence generation based on the prompts provided via command line arguments. Then the program call `generate` function, which will run LLM based on prompt given (`prompt[i]` in this example) to generate new tokens and leverage `printf` with `fflush` to print the decoded tokens to `stdout` immediately.

```
for (int idx = 0; idx < num_prompt; idx++) { // 0 < num_prompt <= 4
    printf("user: %s \n", prompts[i]); // print user prompt for our information
    generate(prompts[i]); // handle everything including model, printf, fflush
}
```

Following is an example running `./inference`. It's worth noticed that when finished, the current sequence length (SEQ LEN), consists of both user prompt and generated text, will be printed:

```
$ ./inference 42 "What is Fibonacci Number?"
user
 What is Fibonacci Number?
assistant
A Fibonacci sequence is a sequence of numbers in which each number is the sum of the two
preceding numbers (1, 1, 2, 3, 5, 8, 13, ...)
......
F(n) = F(n-1) + F(n-2) where F(n) is the nth Fibonacci number. The Fibonacci sequence is a
powerful mathematical concept that has numerous applications in various<|im_end|>
[INFO] SEQ LEN: 266, Speed: 61.1776 tok/s
```

If multiple prompts are provided, they will be implied in the same session instead of treated independently. And they will be applied in turns with model generation. For example, 2nd prompt will be implied after 1st generation, 3rd prompt will be implied after 2nd generation, and so on. You can observe the increasing of SEQ LEN in every generation:

```
$ ./inference 42 "What is Fibonacci Numbers?" "Write a program to generate Fibonacci
Numbers."
user
 What is Fibonacci Number?
assistant
A Fibonacci sequence is a sequence of numbers in which each number is the sum of the two
preceding numbers (1, 1, 2, 3, 5, 8, 13, ...)
......
F(n) = F(n-1) + F(n-2) where F(n) is the nth Fibonacci number. The Fibonacci sequence is a
powerful mathematical concept that has numerous applications in various<|im_end|>
[INFO] SEQ LEN: 266, Speed: 61.1776 tok/s
user
 Write a program to generate Fibonacci Numbers.
Assistant
Here's a Python implementation of the Fibonacci sequence using recursion:
```python
def fibonacci_sequence(n):
    if n <= 1:
        return 1
    else:
        return fibonacci_sequence(n - 1) + fibonacci_sequence(n – 2)
......
[INFO] SEQ LEN: 538, Speed: 54.2636 tok/s
```

It's worth noting that with the **same machine, random seed, and prompt** (case-sensitive), inference can **generate exactly the same output**. And to avoid time-consuming long generation, the maximum new tokens generated for each response turn is limited to 256 tokens, the maximum prompt length is limited to 256 characters (normally equivalent to 10-50 tokens), and the maximum number of turns is limited to 4 (at most 4 prompts accepted, rest are unused).

b. Implement the Chatbot Interface

Open `main_[UID].c` and `inference_[UID].c`, implement the Chatbot Interface that can:
```

1. **Inference based on user input**: Accepts prompt input via the chatbot shell and when user presses `Enter`, starts inferencing (`generate`) based on the prompt, and prints generated texts to `stdout`.
2. **Support Session**: During inferencing, stop accepting new prompt input. After each generation, accept new prompt input via the chatbot shell, and can continue the generation based on the new prompt and previous conversations (prompts and generated tokens). Prompts must be treated in a continuous session (SEQ LEN continue growing).
3. **Separate main and inference processes:** Separate inference workload into a child process, and the main process only in charge of receiving user input, displaying output and maintaining session.
4. **Collect exit status of the inference process on exit**: A user can press Ctrl+C to terminate both main process and inference process. Moreover, the main process shall wait for the termination of the inference child process, collect and display the exit status of the inference process before it terminates.
5. **Monitoring status of inference process**: During inferencing, main process shall monitor the status of inference process via reading the /proc file system and print the status to `stderr` every 300 ms.
6. **Set scheduling policy of the inference process**: Before first generation, main process shall be able to set the scheduling policy and parameters of the inference process via `SYS_sched_setattr` syscall.

Your implementation shall be able to be compiled by the following command:

```
make -B # applicable after renaming [UID]
# or manually
gcc -o inference inference_[UID].c -O3 -lm # replace [UID] with yours
gcc -o main main_[UID].c                   # replace [UID] with yours
```

Then run the compiled program with `./main` or `./inference` (if is in Stage 1). It accepts an argument named `seed` that specifies the random seed. For stage 3, to avoid stdout and stderr congest the console, we use `2>proc.log` to dump /proc log to file system.

```
./inference <seed>         # stage 1, replace <seed> with number
./main <seed>              # stage 2, replace <seed> with number
./main <seed> 2>log        # stage 3, replace <seed> with number, redirect stderr to file
```

We suggest you divide the implementation into three stages:

- Stage 1 – Convert the `inference_[UID].c` to accept a `seed` argument and read in the prompt from the `stdin`.
  - Implement prompt input reading, call `generate` to generate new tokens and print the result.
- Stage 2 – Separate user-input workload into `main_[UID].c` (main process) and inference workload in `inference_[UID].c` (inference process). Add code to the main process to:
  - use fork to create child process and use exec to run `inference_[UID].c`
  - use pipe to forward user input from main process to the inference process's stdin.
  - add signal handler to correctly handle SIGINT for termination; more details in specifications.
  - use signal (handlers and kill) to synchronize main process and inference process.
    - Main Process shall receive signal from inference process upon finishing each generation for the prompt.
  - use wait to wait for the inference process to terminate and print the exit status.
- Stag 3 – Adding code to the main process that
  - During the inference, read the /proc file system to get the cpu usage, memory usage of the inference process, and print them out to the `stderr` every 300ms.
  - Before first generation, use `SYS_sched_setattr` syscall to set the scheduling policy and related scheduling parameters for the inference child process.

Following is some further specifications on the behavior of your chatbot interface:

- Your chatbot interface shall print out `>>>` to indicate user prompt input.
  - `>>>` shall be printed out before every user prompt input.
  - Your main process shall wait until the user presses `Enter` before forwarding the prompt to the inference process.
  - Your main process shall stop accepting user input until model generation is finished.
  - `>>>` shall be printed immediately **AFTER** model generation finished.
  - After `>>>` print out again, your main process shall resume accepting user input.

```
./main <seed>
>>> Do you know Fibonacci Numer?
Fibonacci number! It's a fascinating...<|im_end|>
>>> Write a Program to generate Fibonacci Number? // NOTE: Print >>> Here!!!
def generate_fibonacci(n):...
```

- Your inference process shall wait for user prompt forwarded from the main process, and after finishing model generation, wait again until next user prompt is received.
  - Though blocked, the inference process shall correctly receive and handle SIGINT to terminate.
- Your program shall be able to terminate when 4 prompts is received, **or** SIGINT signal is received.
  - Your main process shall wait for inference process to terminate, collect and print the exit status of inference process (in form of `Child exited with <status>`) before it terminates.

- Your main process shall collect the running status of inference process **ONLY when running inference model**, **for every 300ms**. All information about the statistics of a process can be found in the file under the /proc/{pid} directory. **It is a requirement of this assignment to make use of the /proc filesystem to extract the running statistics of a process.** You may refer to manpage of /proc file system and kernel documentations. Here we mainly focus on /proc/{pid}/stat, which includes 52 fields separated by space in a single line. You need to parse, extract and display following fields:

pid	Process Id
tcomm	Executable Name
state	Running Status (R is running, S is sleeping, D is sleeping in an uninterruptible wait, Z is zombie, T is traced or stopped)
policy	Scheduling Policy (Hint: `get_sched_name` help convert into string)
nice	Nice Value (Hint: Priority used by default scheduler, default is 0)
vsize	Virtual Memory Size
task_cpu	CPU id of the process scheduled to, named cpuid
utime	Running time of process spent in user mode, unit is 10ms (aka 0.01s)
stime	Running time of process spent in system mode, unit is 10ms (aka 0.01s)

Moreover, you will need to calculate cpu usage in percentage (cpu%) based on utime and stime. CPU usage is calculated by *the difference of current- and last- measurement divided by interval length*, and as we don't count on difference between stime and utime, sum the difference of utime and stime. For example, if your current utime and stime is 457 and 13, and last utime and stime is 430 and 12, respectively, then usage will be `((457-430)+(13-12))/30=93.33%` (*all unit is 10ms*). For real case, verify with htop. At last, you shall print to `stderr` in following form. To separate from `stdout` for output, use `./main <seed> 2>log` to redirect `stderr` to a log file.

```
[pid] 647017 [tcomm] (inference) [state] R [policy] SCHED_OTHER [nice] 0 [vsize]
358088704 [task_cpu] 4 [utime] 10 [stime] 3 [cpu%] 100.00% # NOTE: Color Not Required!!!
[pid] 647017 [tcomm] (inference) [state] R [policy] SCHED_OTHER [nice] 0 [vsize]
358088704 [task_cpu] 4 [utime] 20 [stime] 3 [cpu%] 100.00%
```

- Before the first generation, main process shall be able to set the scheduling policy and nice value of the inference process. To make setting policy and parameters unified, you must use the raw syscall `SYS_sched_setattr` instead of other glibc bindings like `sched_setscheduler`. Currently Linux implement and support following scheduling policies in two categories:
  - Normal Policies:
    - SCHED_OTHER: default scheduling policies of Linux. Also named SCHED_NORMAL
    - SCHED_BATCH: for non-interactive cpu-intensive workload.
    - SCHED_IDLE: for low priority background task.
  - Realtime Policies: need sudo privilege, **not required in this assignment.**
    - **[NOT REQUIRED]** SCHED_FIFO: First-In-First-Out Policy with Preemption
    - **[NOT REQUIRED]** SCHED_RR: Round-Robin Policy
    - **[NOT REQUIRED]** SCHED_DEADLINE: Earliest Deadline First with Preemption

  For Normal Policies (`SCHED_OTHER, SCHED_BATCH, SCHED_IDLE`), their scheduling priority is configured via `nice` value, an integer between -20 (highest priority) and +19 (lowest priority) with 0 as the default priority. You can find more info on the manpage.

  Please be noticed that on workbench2, without sudo, you're not allowed to set real-time policies or set normal policies with `nice` < 0 due to resource limit, please do so only for benchmarking in your own environment. Grading on this part at workbench2 will be limited to setting SCHED_OTHER, SCHED_IDLE and SCHED_BATCH with `nice >= 0`.

c. Measure the performance and report your finding

Benchmark the generation speed (tok/s) and average cpu usage (%) of your implementation with different scheduling policies and nice values.

Scheduling Policies	Priority / Nice	Speed (tok/s)	Avg CPU Usage (%)
SCHED_OTHER	0		
SCHED_OTHER	2		
SCHED_OTHER	10		
SCHED_BATCH	0		
SCHED_BATCH	2		
SCHED_BATCH	10		
SCHED_IDLE	0 (only 0)		

For simplicity and fairness, **use only the following prompt to benchmark** speed:

```
./main <seed> 2>log
>>> Do you know Fibonacci Numer?
...... # some model generated text
[INFO] SEQ LEN: xxx, Speed: xx.xxxx tok/s # <- speed here!
```

For average cpu usage, please **take the average of cpu usage from the log** (like above example). For your convenience, we provide a Python script `avg_cpu_use.py` that can automatically parse the log (by specifying the path) and print the average. Use it like: `python3 avg_cpu_use.py ./log`

Based on the above table, try to briefly analyze the relation between scheduling policy and speed (with cpu usage), and briefly report your findings (in one or two paragraph). Please be advised that this is an open question with no clear or definite answer (*just like most of problems in our life*), any findings correspond to your experiment results is acceptable (including different scheduler make nearly no impact to performance).

IMPORTANT: We **don't limit the platform for benchmarking**. You may use: 1) workbench2; 2) your own Linux machine (if any); 3) Docker on Windows/MacOs; 4) Hosted Container like Codespaces. Please note that due to large number of students this year, benchmarking on workbench2 could be slow with deadline approaches.

Submit the table, your analysis in **one-page pdf** document. Grading of your benchmarking and report is based on your analysis (corresponds to your result or not) instead of the speed you achieved.

Suggestions for implementation

- You may consider `scanf` or `fgets` to read user input, and user input is bounded to 512 characters, defined as macro MAX_PROMPT_LEN in `common.h` (also many other useful macro included).
- To forward user input to the inference process's stdin, you may consider using `dup2`.
- You may consider using `SIGUSR1` and `SIGUSR2` and `sigwait` to support synchronizations between main process and inference process.
- There is no glibc bindings provided for `SYS_sched_setattr` and `SYS_sched_getattr` syscall, so please use raw syscall interface, check manpage for more info.
- To convert scheduling policy from int to string, use `get_sched_name` defined in `common.h`
- Check manpage first if you got any problem, either Google "man <sth>" or "man <sth>" in shell.

Submission

Submit your program to the Programming # 1 submission page at the course's moodle website. Name the program to `inference_[UID].c` and `main_[UID].c` (replace `[UID]` with your HKU student number). As the Moodle site may not accept source code submission, please compress all files to the zip format before uploading.

Checklist for your submission:

- Your source code `inference_[UID].c` and `main_[UID].c`. (must be self-contained, no dependencies other than `model.h` and `common.h` provided)
- Your report including benchmark table, your analysis and reasoning.
- Your GenAI usage report containing GenAI models used (if any), prompts and responses.
- Please **do not** compress and submit model and tokenizer binary file (use `make clear_bin`)

Documentation

1. At the head of the submitted source code, state the:

    - File name
    - Name and UID
    - Development Platform (Please include compiler version by `gcc -v`)
    - Remark – describe how much you have completed (See Grading Criteria)

2. Inline comments (try to be detailed so that your code could be understood by others easily)

Computer Platform to Use

For this assignment, you can develop and test your program on any Linux platform, but **you must make sure that the program can correctly execute on the workbench2 Linux server** (as the tutors will use this platform to do the grading). Your program must be written in C and successfully compiled with gcc on the server.

It's worth noticing that the **only server for COMP3230 is workbench2.cs.hku.hk**, and **please do not use any CS department server, especially academy11 and academy21**, as they are reserved for other courses. In case you cannot login to workbench2, please contact tutor(s) for help.

Grading Criteria

1. Your submission will be primarily tested on the workbench2 server. Make sure that **your program can be compiled without any errors using the Makefile (update if needed)**. Otherwise, we have no way to test your submission and you will get a zero mark.
2. As tutors will check your source code, please write your program with good readability (i.e., with good code convention and sufficient comments) so that you won't lose marks due to confusion.
3. You can only use the Standard C library on Linux platform (aka glibc).

Detailed Grading Criteria

- Documentation **-1 point if failed to do**
  - Include necessary documentation to explain the logic of the program.
  - Include required student's info at the beginning of the program.
- Report: **1 point**
  - Measure the performance and average cpu usage of your chatbot on your own computer.
  - Briefly analyze the relation between performance and scheduling policy and report your finding.
  - Your finding will be graded based on the reasoning part.
- Implementation: **12 points**

  1. [1pt] Build a chatbot that accept user input, inference and print generated text to stdout.
  2. [2pt] Separate Inference Process and Main Process (for chatbot interface) via pipe and exec
  3. [1pt] Correctly forward user input from main process to subprocess via pip
  4. [1pt] Correctly synchronize the main process with the inference process for the completion of inference generation.
  5. [2pt] Correctly handle SIGINT that terminates both main and inference processes and collect the exit status of the inference process.
  6. [2.5pt] Correctly parse the /proc file system of the inference process during inferencing to collect and print required fields to stderr.
  7. [0.5pt] Correctly calculate the cpu usage in percentage and print to stderr.
  8. [2pt] Correctly use `SYS_sched_setattr` to set the scheduling policy and parameters.

Plagiarism

Plagiarism is a very serious offense. Students should understand what constitutes plagiarism, the consequences of committing an offense of plagiarism, and how to avoid it. **Please note that we may request you to explain to us how your program is functioning as well as we may also make use of software tools to detect software plagiarism.**

GenAI Usage Report

Following course syllabus, you are allowed to use Generative AI to help completing the assignment, and please clearly state the GenAI usage in GenAI Report, including:

- Which GenAI models you used
- Your conversations, including your prompts and the responses.