Process and Signal: Simulated Parallel Processing

Overview

Parallel processing is a powerful technique used in computing to enhance performance and efficiency. It involves breaking down a large job into smaller, independent tasks that can be executed concurrently on multiple cores of a CPU. This concurrent execution of tasks significantly accelerates the overall job completion time. The CPU scheduler plays a crucial role in this process, efficiently assigning tasks to different cores. Once all tasks are completed, the results are aggregated to form the final output. This method of job execution optimizes CPU utilization, thereby improving system performance and enabling efficient handling of large and complex jobs. In this project, we will simulate a simplified version of this process with the concept of process, pipe, and signal learned from the course.

The project assignment will exercise your understanding of process manipulation, communication, and signal handling in C. This is an inherently hard project so please make sure to start early. You need to write the code from scratch on your own.

Learning Objectives

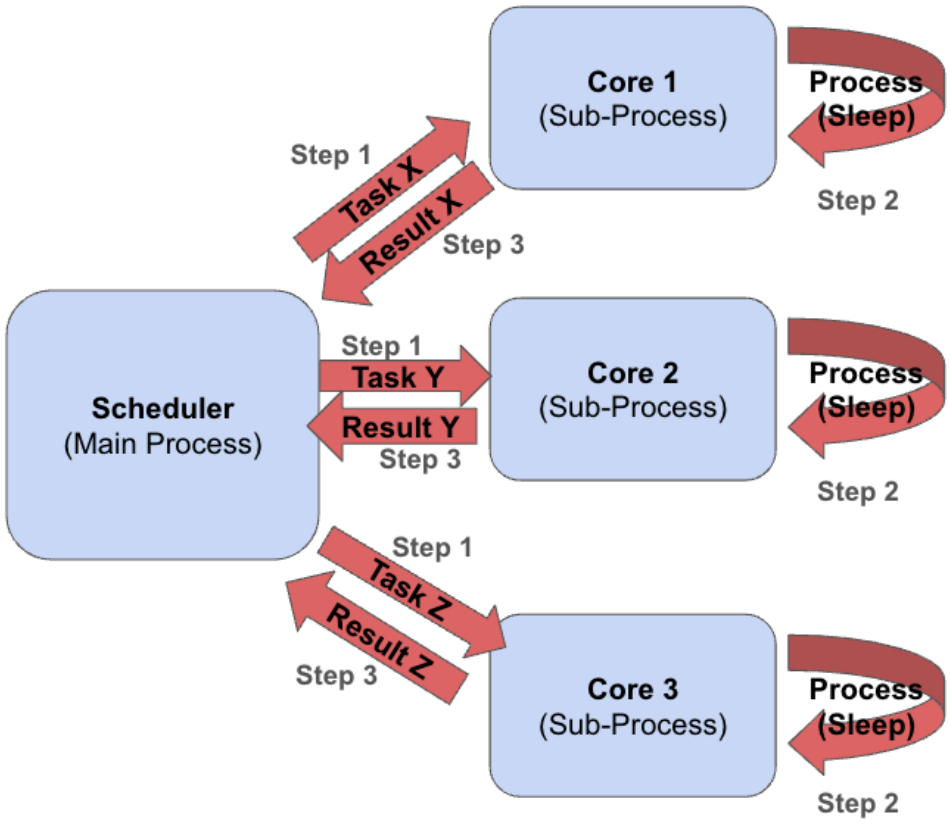
1. Creating processes and pipes
2. Communication between processes using pipes
3. Writing code for signal handling

Instructions

Overview

This project aims to provide a simple simulation of parallel processing across multiple processes. Parallel processing is the process of breaking a job into smaller tasks and executing them on multiple cores. It involves the following steps:

1. **Job Decomposition:** The initial job is broken down into smaller, independent tasks. This process is known as decomposition or partitioning. The goal is to create tasks that can be executed concurrently without interfering with each other.
2. **Task Assignment:** Once the tasks have been identified, they are assigned to different cores of the CPU. This is done by the scheduler, which uses algorithms to determine the most efficient distribution of tasks.
3. **Concurrent Execution:** Each core executes its assigned tasks concurrently. This means that multiple tasks are being processed simultaneously, significantly speeding up the overall execution time of the job.
4. **Aggregation:** Once all tasks have been completed, the results are aggregated. This involves combining the results from each task to form the final output of the job.



In this project, you simulate these steps with the concepts of process, pipe, and signal you learned from the class. More specifically, three sub-processes are created to act as the cores of the CPU. The main process decomposes the job into independent tasks and assigns them to the created subprocesses. Once a subprocess receives a new assignment, it executes the task and gets back to the main process once it's done. The main process will aggregate the returned result and might assign a new task to the sub-process depending on the remaining unsolved tasks. The program should take two command-line arguments: **the total number of tasks to be executed** and **the maximum time that any task can process**. These arguments will determine the number of independent tasks the main process will decompose the job into and the range of values each task will handle.

The document will go over all the necessary details needed for finishing the project. Let's get started!

Part 0: Project Startup

For this project, you need to write all the code from scratch. Unlike the previous projects, no starter code has been provided. You start by creating one or multiple C files for coding. You also need to make your own Makefile to compile the program you wrote.

Part 1: Create Pipes and Core Subprocesses

You need to create a main function as always in your C file. This is the entrance of the program. Once the program starts to run, the main process is created on the operating system and runs from the main function. As stated before, we want the main process to coordinate the parallel processing. Notice that the main function is not equivalent to the main process even though they have similar names.

Before the main process decomposes the task, we would like to begin with some initialization work of the system. One important task we need to do is to use the fork system call to create subprocesses. Each subprocess acts as a core of the CPU. **For simplicity, we assume there are three cores in the simulated CPU. Thus, we only need three subprocesses (children processes) for the simulation.**

After finishing creating the user subprocesses, you may notice that we haven't built communication between the subprocesses and the main process. We use pipes as a communication tool. Pipe is a one-way communication mechanism, but we need two-way communication because the subprocess needs to receive a task from the main process and return the result to the main process. Thus, every core needs two pipes to send and receive

messages to and from the main process. For example, two pipes should be created for core 1 (int main_to_core1[2], core1_to_main[2];). Now try to create the pipes so that each subprocess owns two pipes that can communicate with the main process back and forth. **Be very careful about closing the unused write/read end of the pipe.** For example, Core 1 should close the write end of the main_to_core1 pipe and the read end of the core1_to_main pipe. Core 1 should also close all the write end and read end of the other coreX_to_main pipe and main_to_CoreX pipe if they exist (X is not equal to 1). Furthermore, the main process should close the read end of the main_to_core1 pipe and the write end of the core1_to_main pipe.

Just to provide some hint: one potential structure of your code might be similar to the following pseudocode. Other implementations are also acceptable.

```
int main()
{
    ……
    For each core process you want to create:
        Create pipe()
        Do fork()
        Close unused end of pipes
    ……
}
```

Part 2: Task Definition and Scheduling

By now, we have created three cores for task completion and each core can communicate with the main process in a two-way manner. Before the core starts to execute its tasks, we need to first know what the task is and how we decompose a job into tasks. In this project, for simplicity, we use a trivial task: **The main process repeatedly assigns a new task to an idle core with the task ID till all the tasks are finished. Once a core accepts a new task, the only job it needs to do for this task is to read the task ID, sleep for a random amount of time (up to the second command-line argument), and then send the task ID back as the result of the completion of the task.** You can use the rand() function to randomize the sleep time for each task, ensuring that each task is completed in a different amount of time.

```
int sleep_time = rand() % max_sleep_time + 1;
```

Although it may not cause any problems, we encourage you to use the provided no_interrupt_sleep function instead of the sleep function in the standard library. This is because the original sleep function may be interrupted by a signal and not paused for the given amount of time.

```
#include <stdlib.h>
#include <time.h>

void no_interrupt_sleep(int sec)
{
    // * advanced sleep which will not be interfered by signals
    struct timespec req, rem;

    req.tv_sec = sec;  // The time to sleep in seconds
    req.tv_nsec = 0; // Additional time to sleep in nanoseconds

    while(nanosleep(&req, &rem) == -1)
        if(errno == EINTR)
            req = rem;
}
```

The number of tasks the main process assigns and maximum time that any task can process are given by the user via two integer command-line arguments (e.g. *$ ./main 30 5*). Now, alter your main function to accept an integer argument. As a hint, you can search for the usage instructions for **argc** and **argv**. Don't forget to handle illegal input from the user.

As for the strategy for selecting an idle core, task scheduling is a well-researched area aimed at optimizing efficiency. However, in this project, we will implement a simple approach: the main process will perform a linear search through the list of subprocesses (cores) and assign a new task to the first idle core it encounters. This ensures that tasks are distributed in the order that cores become available.

Part 3: Assign Task to Core

Once the core subprocess starts to run, it waits for a new task by calling the read system call to read from the main_to_coreK pipe, where K represents the ID of the core. Notice that, by default, the read function will block if the pipe is empty but still open. Due to this feature, the core subprocess will block until a new task comes. When a new task arrives, the core subprocess reads the task ID, executes the task, sends back the result, and returns to the wait status again as a loop.

The main process assigns a task to core K by calling the write system call and sending the task ID to the main_to_coreK pipe.

Part 4: Return Result to Main Process

Now we have created three core subprocesses, defined the job and its subtasks, designed the scheduling algorithm, and set up how each task is assigned to a core via pipes. The only missing component is how to return the result.

An intuitive implementation is to implement a similar mechanism as Part 3: After assigning a task to core K, the main process calls the read system call to read from the coreK_to_main pipe waiting for the result from core K. Once core K finishes the task, it will send the result back via the coreK_to_main pipe. As a result, the main process will receive the result from the read system call.

This is easy, but unfortunately, it is unsuitable for the project scenario. As we mentioned before, the read system call will block by default if there is no data inside the pipe. After the main process calls read, it will block and wait for the result from core K. During this period, it cannot do anything. This is undesirable since other cores might finish their tasks but the main process cannot assign a new task to them immediately due to the block making the computation not parallel anymore.

One solution is to use unblocked read, but it is beyond the scope of this course. Alternatively, we can use *sigaction*. When core K finishes its task, it writes the result into the coreK_to_main pipe. Also, it sends a signal to the main process to inform the new inbounding result. The main process no longer blocks and waits for the result after it assigns a task. Instead, it only reads the pipe for the result when a signal comes, indicating that the result from core K is already pushed to the pipe. In this case, the signal will interrupt the program flow making the main process run the signal handler function, which processes the new result. Here is a pseudocode for this process (core 1 is used as an example):

```
Handle_new_result_from_core1()
    Handle the new result…

Main_process()
    Register SIGUSR1 with signal handler Handle_new_result_from_core1
    While there are still unsolved tasks
        Assign an unsolved and unassigned task to the next idle core

Core(core_id=1)
    While the main_to_core1 pipe is not closed
        Wait for a new task
        Process the new task
        Push the result to core1_to_main and send a signal SIGUSR1 to main process
```

This idea seems not difficult. However, there are two traps that you might fall into.

Trap 1: Race condition in the signal handler

It is super tempting to read the result from the core1_to_main pipe in the signal handler (see the following code snippet), but this is incorrect! The signal handler may be interrupted by another signal. For example, the read function in your handler might be interrupted by another signal causing race condition problems. Thus, in a signal handler, you are only allowed to do some asynchronous safe operations. The general rule is to keep the operations in a signal handler as simple and atomic as possible.

```
Handle_new_result_from_core1()
    Read the new result from the core1_to_main pipe
    Aggregate the result
```

For this reason, we define a global variable msg_num_core1 of type ***volatile sig_atomic_t*** (just think of it as a variable of type ***int***) and initialize it as 0. In the signal handler, the only line is *++msg_num_core1*. In each iteration of the while loop of the main process, check if *msg_num_core1* is larger than 0. If yes, this indicates an incoming result from core 1, so safely read the data via the core1_to_main pipe. Then decrement *msg_num_core1* by 1. In this case, every time a new result is pushed to the pipe, the signal handler interrupts the main process, and the msg_num_core1 increases by 1 indicating a new inbounding result. After completing the signal handler function, the while loop in the main process will check the value of msg_num_core1 and read the new message from the pipe as soon as possible.

```
volatile sig_atomic_t msg_num_core1 = 0

Handle_new_msg_from_core1()
    ++msg_num_core1

Main_process()
    Register SIGUSR1 with signal handler Handle_new_msg_from_user1
    While there are still unsolved tasks
        Assign an unsolved and unassigned task to the next idle core if there is any idle core exist
        If msg_num_core1 > 0:
            Read the result from the core1_to_main pipe
            --msg_num_core1
            Aggregate the result, and set the corresponding task as solved
            Put core 1 to idle status
```

Trap 2: loss of signals

If you implement the above pseudocode, your program should work in some cases, but you might notice the loss of signals occasionally. This usually happens when two signals are sent at nearly the same time. This is because a signal will be eaten up when triggered during the processing of the signal handler for the same signal. One solution to this is to use "***real-time***

signals". Just replace SIGUSR1 with a signal between SIGRTMIN and SIGRTMAX[1], and the problem is fixed. The integer values of SIGRTMIN and SIGRTMAX are defined in *signal.h* so don't forget to include this header file when you want to pick a real-time signal to use. We don't require you to understand the concept of real-time signals, which is beyond the scope of the course, but in case you are interested in that, you can check this website.

Now, try to finish the project based on the pseudocode. Notice that we only used one core as an example, but you should be able to extend your program to three core subprocesses. Notice that each core should use a unique real-time signal.

Part 5: Post-processing

After Part 5, our program should work as expected: the main process assigns new tasks via pipes whenever there is an idle core subprocess. It also aggregates the result from a core as soon as possible when a signal from a core arrives. The core subprocess repeatedly waits for a new task, processes it, returns the result via pipes, and sends a signal to the main process informing the readiness of a new result (this also indicates the idleness of the core after finishing the task and bringing back the result).

The last part of the project is post-processing. When all the tasks are done with results, the main process quits the loop of assigning tasks and does necessary post-processing including closing all the pipes for communication with cores, reaping the child processes, and freeing the allocated memory.

It is worth mentioning that closing the pipes is vital. Since the main function already has the results for every task, we can infer that every core is currently idle. Thus, every core is blocked by the read system call and waiting for a new task. The closure of the pipes from the main process side stops the blocking, making the read function return 0. This tells the core subprocess that there will be no more new incoming tasks. Thus, each core can safely break its loop, do post-processing including closing all the pipes, freeing the allocated memory, etc.

For grading, please add a counter to each core subprocess to count the number of tasks it processes. Output the number of tasks each core subprocess processed at the end of the program in the following format:

```
Core 1 = A
Core 2 = B
Core 3 = C
```

[1] The real-time signals, ranging from SIGRTMIN to SIGRTMAX, are a set of signals that can be used for application-defined purposes. One of the primary differences between normal signals and real-time signals is that multiple instances of individual real-time signals can be queued.

Also, please output log information in important stages for both the main process and the subprocesses, for example: the creation of the core subprocess, the assignment of a new task, the closure of the main_to_coreK pipe, etc.

Grading Breakdown

1. Project Requirements (60 points)
    a. User processes are correctly created as stated in the requirement (4 pts)
    b. Processes are reaped (3 pts)
    c. Pipes are created to support two-way communication correctly (3 pts)
    d. The corresponding ends of pipes are correctly closed on both the parent and child process ends (8 pts)
    e. All pipes are correctly closed in the clean-up stage (4 pts)
    f. Correctly calls the write function to send data to the pipe (4 pts)
    g. Correctly call the read function to read data from the pipe (4 pts)
    h. Choose appropriate signals to register (3 pts)
    i. Correctly use sigaction to register signal handlers (3 pts)
    j. Signal handler should be correctly implemented. Its operation should keep atomic. (6 pts)
    k. Use the proper type of global variable as the indicator of new results (2 pts)
    l. Signal should be properly sent out when a message is sent out (2 pts)
    m. The core subprocess can detect the close of pipes and quit the status of the wait (3 pts)
    n. Error handling on all the functions and system calls including read/write/fork/pipe... (4 pts)
    o. Implement the command line arguments to allow user to control the number of total tasks (4 pts)
    p. Do not cause memory leakage. (3 pts)
2. Design and Implementation (12 points)
    a. Functions are declared and used properly. Variable and function naming is clear and helps understand the program's purpose. (3 pts)
    b. Global variables are minimized, declared, and used properly. (3 pts)
    c. Control flow (e.g., if statements, looping constructs, function calls) is used properly. (3 pts)
    d. There is no inefficient implementation such as no unnecessary blocking in the program, unreachable code, confusing or misleading constructions, or extra loop. (3 pts)
3. Coding Style and Comments (18 points)
    a. Code is written in a consistent style. For example, curly brace placement is consistent across all if/then/else and looping constructors. (3 pts)
    b. Proper and consistent indenting and spacing are adhered to across the entire implementation. (3 pts)

    c. Proper variable names are used making the code understandable and readable. (3 pts)
    d. Comments in the code are used to document algorithms. (3 pts)
    e. Variables are documented such that they aid the reader in understanding your code. (3 pts)
    f. Functions are documented to indicate the purpose of the parameters and return values. (3 pts)
4. README.txt and Makefile (9 points)
    a. The README.txt file provides an overview of your implementation. (3 pts)
    b. The README.txt file explains how your code satisfies each of the requirements. (3 pts)
    c. The code is compilable using Makefile. (3 pts)
5. GradeScope Submission (1 point)
    a. You automatically get a point for submitting to Gradescope. (1 pts)

Submission

You must submit the following to Gradescope by the assigned due date:

- **Makefile** - the build file to use with the **make** program. You can download the sample makefile here.
- **Source Files** - source file that provides the solution to this project.
- **README.txt** - this is a text file containing an overview/description of your submission highlighting the important parts of your implementation. You should also explain where in your implementation your code satisfies each of the requirements of the project or any requirements that you did not satisfy. The goal should make it easy and obvious for the person grading your submission to find the important rubric items.

Make sure you submit this project to the correct assignment on Gradescope!