

Name: Nguyễn Đại Hưng

ID: 24520601

Class: IT007.Q15.1

OPERATING SYSTEM LAB 3'S REPORT

SUMMARY

Task		Status	Page
Section 3.5	Ex 1	Hoàn thành	2
	Ex 2	Hoàn thành	11
	Ex 3	Hoàn thành	13
	Ex 4	Hoàn thành	15

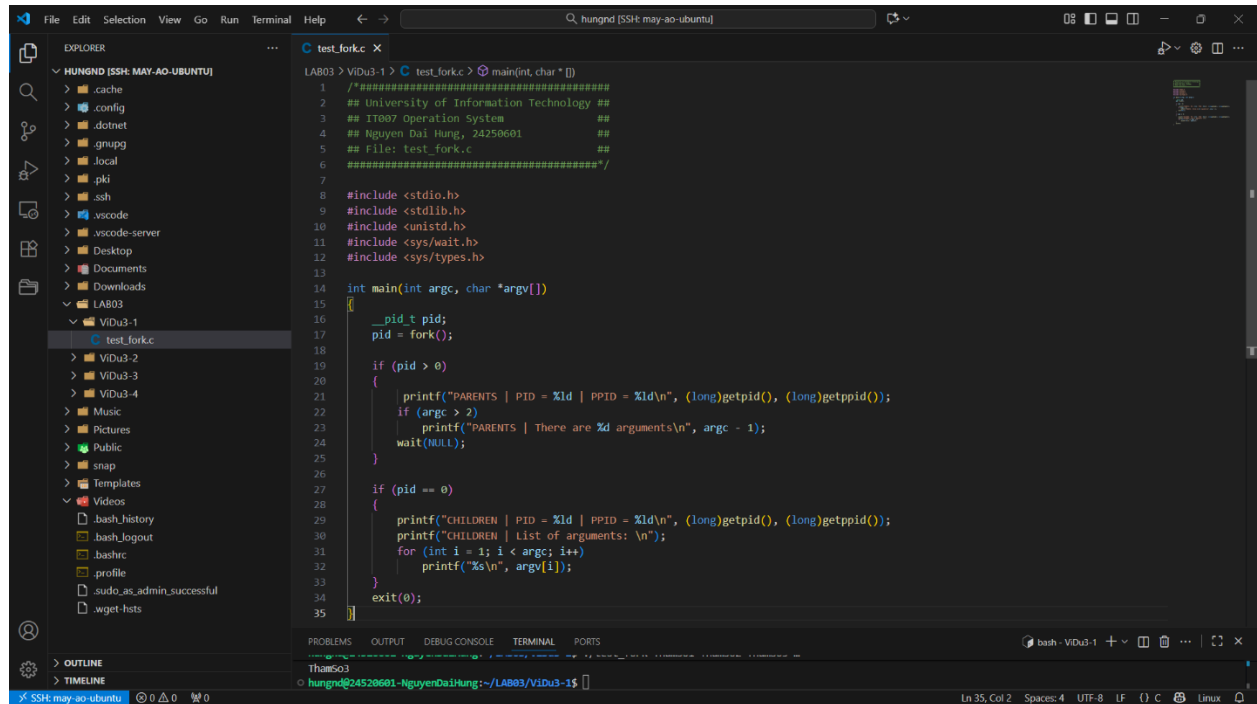
Self-scores: 10

Note: Export file to **PDF and name the file by following format:
Student ID_LABx.pdf*

Section 3.5

1. Thực hiện ví dụ 3-1, ví dụ 3-2, ví dụ 3-3, ví dụ 3-4. Giải thích code và kết quả nhận được.

a) Ví dụ 3-1.



```
LAB03 > VIDu3-1 > C test_fork.c > main(int, char *[])
1  /*=====
2  ## University of Information Technology ##
3  ## IT807 operation system ##
4  ## Nguyen Dai Hung, 24250601 ##
5  ## File: test_fork.c ##
6  =====*/
7
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include <unistd.h>
11 #include <sys/wait.h>
12 #include <sys/types.h>
13
14 int main(int argc, char *argv[])
15 {
16     _pid_t pid;
17     pid = fork();
18
19     if (pid > 0)
20     {
21         printf("PARENTS | PID = %ld | PPID = %ld\n", (long) getpid(), (long) getppid());
22         if (argc > 2)
23             printf("PARENTS | There are %d arguments\n", argc - 1);
24         wait(NULL);
25     }
26
27     if (pid == 0)
28     {
29         printf("CHILDREN | PID = %ld | PPID = %ld\n", (long) getpid(), (long) getppid());
30         printf("CHILDREN | List of arguments: \n");
31         for (int i = 1; i < argc; i++)
32             printf("%s\n", argv[i]);
33     }
34     exit(0);
35 }
```

Hình 1.1: Source code ví dụ 3-1

• Giải thích code:

- Đoạn mã sử dụng lời gọi hệ thống **fork()** để tạo ra một tiến trình con. Sau khi **fork()** được gọi, chương trình được nhân bản thành hai tiến trình (cha và con) chạy song song.
 - Logic của Tiến trình Cha (**pid > 0**): Tiến trình cha (có **pid** là ID của con) sẽ in ra PID và PPID của chính nó. Sau đó, nó gọi **wait(NULL)** để tạm dừng thực thi và chờ cho đến khi tiến trình con kết thúc.
 - Logic của Tiến trình Con (**pid == 0**): Tiến trình con (có **pid** bằng 0) cũng in ra PID và PPID của nó (PPID này chính là PID của tiến trình cha). Nhiệm vụ chính của tiến trình con là lặp và in ra danh sách các đối số dòng lệnh (command-line arguments) đã được truyền vào chương trình.
- Mục đích chính của mã là minh họa việc tạo và quản lý tiến trình con, đồng thời sử dụng **wait()** để đồng bộ hóa, đảm bảo tiến trình cha chỉ kết thúc sau khi tiến trình con đã hoàn thành.

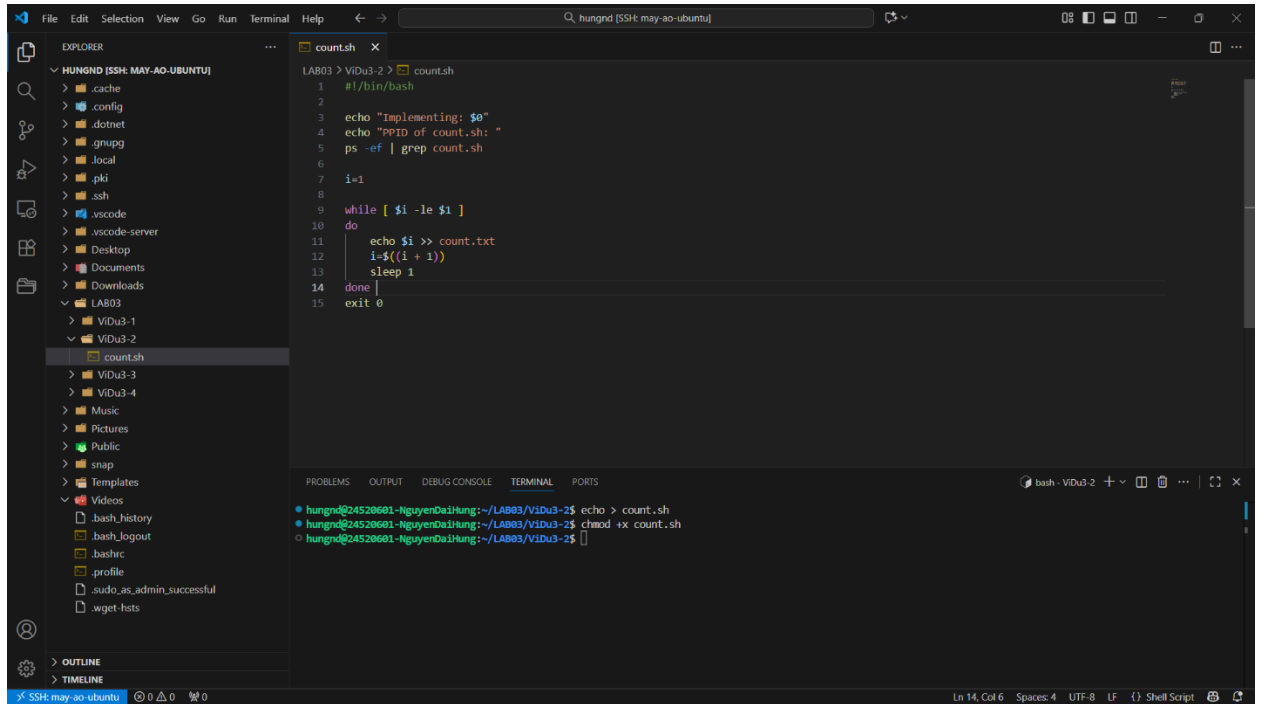
```
LAB03 > VIDu3-1 > C test_fork.c > main(int, char * [])
8 #include <stdio.h>
10 #include <unistd.h>
11 #include <sys/wait.h>
12 #include <sys/types.h>
13
14 int main(int argc, char *argv[])
15 {
16     _pid_t pid;
17     pid = fork();
18
19     if (pid > 0)
20     {
21         printf("PARENTS | PID = %ld | PPID = %ld\n", (long) getpid(), (long) getppid());
22         if (argc > 2)
23             printf("PARENTS | There are %d arguments\n", argc - 1);
24         wait(NULL);
25     }
26
27     if (pid == 0)
28     {
29         printf("CHILDREN | PID = %ld | PPID = %ld\n", (long) getpid(), (long) getppid());
30         printf("CHILDREN | List of arguments: \n");
31         for (int i = 1; i < argc; i++)
32             printf("%s\n", argv[i]);
33     }
34     exit(0);
35 }
```

```
hungnd@24528601-NguyenDaiHung:~/LAB03/VIDu3-1$ gcc test_fork.c -o test_fork
hungnd@24528601-NguyenDaiHung:~/LAB03/VIDu3-1$ ./test_fork ThamSo1 ThamSo2 ThamSo3
PARENTS | PID = 7174 | PPID = 3297
PARENTS | There are 3 arguments
CHILDREN | PID = 7175 | PPID = 7174
CHILDREN | List of arguments:
ThamSo1
ThamSo2
ThamSo3
```

Hình 1.2: Kết quả chương trình ví dụ 3-1

- **Giải thích kết quả:**
 - Dòng lệnh `gcc test_fork.c -o test_fork` : dùng để biên dịch mã nguồn ngôn ngữ C của file `test_fork.c` thành một chương trình có thể thực thi được (tên file là `test_fork`)
 - Dòng lệnh `./test_fork ThamSo1 ThamSo2 ThamSo3` : dùng để thực thi file `test_fork`, đồng thời truyền 3 tham số (`ThamSo1`, `ThamSo2`, `ThamSo3`) cho chương trình thực thi này.
 - Kết quả **PARENTS | PID = 7174 | PPID = 3297** : kết quả là **PARENTS**, tức kết quả trả về hiện tại của hàm `fork()` > 0 (hàm `fork()` đang ở trong tiến trình cha), PID của tiến trình cha là 7174, PPID của tiến trình cha là 3297.
 - Vì `argc = 4` / (“test_fork”, “ThamSo1”, “ThamSo2”, “ThamSo3”); $4 > 2$ nên chương trình sẽ in ra câu **PARENTS | There are 3 (argc – 1) arguments**. Nếu `argc` ≤ 2 thì câu lệnh này sẽ không được in ra.
 - Kết quả **CHILDREN | PID = 7175 | PPID = 7174: CHILDREN**, tức là hàm `fork()` hiện tại đang ở trong tiến trình con (`fork() = 0`). ID của tiến trình con này là 7175, ID của tiến trình cha của tiến trình con này là 7174.

b) Ví dụ 3-2.



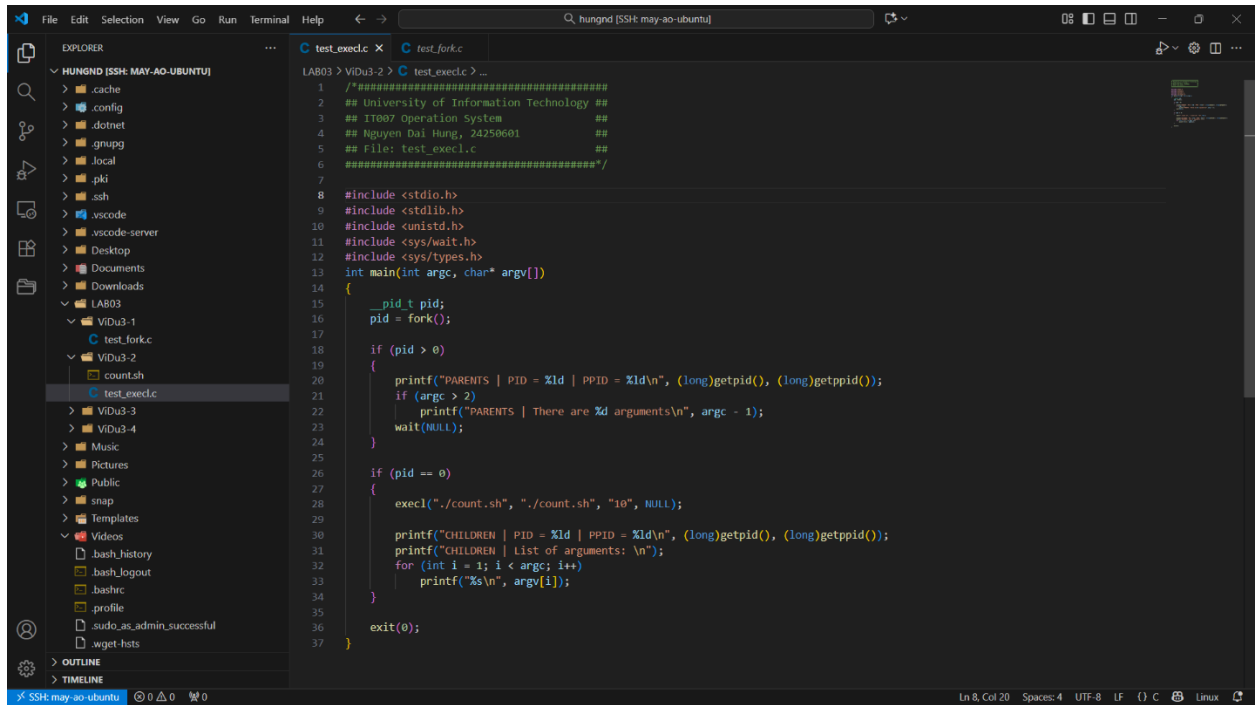
The screenshot shows a VS Code editor window with a file explorer on the left and a code editor in the center. The file explorer shows a directory structure with a file named `count.sh` under the `LAB03` directory. The code editor displays the content of `count.sh`, which is a shell script that implements a loop to count from 1 to 10, printing the current value and the PID of the script to a file named `count.txt`. The terminal at the bottom shows the execution of the script, with the output of the `echo > count.sh` and `chmod +x count.sh` commands.

```
count.sh
1 #!/bin/bash
2
3 echo "Implementing: $0"
4 echo "PID of count.sh: "
5 ps -ef | grep count.sh
6
7 i=1
8
9 while [ $i -le 10 ]
10 do
11     echo $i >> count.txt
12     i=$((i + 1))
13     sleep 1
14 done
15 exit 0
```

```
bash - ViDu3-2
* hungnd@24528691-NguyenDaiHung:~/LAB03/ViDu3-2$ echo > count.sh
* hungnd@24528691-NguyenDaiHung:~/LAB03/ViDu3-2$ chmod +x count.sh
* hungnd@24528691-NguyenDaiHung:~/LAB03/ViDu3-2$
```

Hình 2.1: Script của file *count.sh*

- Giải thích code file *count.sh*:
 - File *count.sh* là một file kịch bản để thực hiện một vòng lặp đếm số, đồng thời ghi lại kết quả ra file (*count.txt*) và in thông tin về tiến trình của chính nó.
 - *ps -ef | grep count.sh*: lệnh này liệt kê các tiến trình trên hệ thống mà chứa chuỗi “*count.sh*”.
 - *chmod +x count.sh*: lệnh này để cấp quyền thực thi (execute) cho file kịch bản *count.sh*.



Hình 2.2: Source code ví dụ 3-2

- **Giải thích code:**

- Đoạn mã sử dụng lời gọi hệ thống **fork()** để tạo ra một tiến trình con. Sau khi **fork()** được gọi, chương trình được nhân bản thành hai tiến trình (cha và con) chạy song song.
 - Logic của tiến trình Cha (**pid > 0**): Tiến trình cha (có **pid** là ID của con) sẽ in ra PID và PPID của chính nó. Sau đó, nó gọi **wait(NULL)** để tạm dừng thực thi và chờ cho đến khi tiến trình con kết thúc.
 - Logic của tiến trình Con (**pid == 0**): Nhiệm vụ chính của tiến trình con là gọi hàm **execl()** để thay thế mã lệnh của chính nó bằng một chương trình mới (là file script **./count.sh**). Nếu **execl()** thực thi thành công, các dòng **printf** bên dưới nó sẽ không bao giờ được chạy. Các dòng **printf** đó chỉ có tác dụng báo lỗi nếu **execl()** thất bại.
- Mục đích chính của mã là minh họa cách một tiến trình cha tạo ra một tiến trình con và sử dụng họ hàm **exec** để chạy một chương trình hoàn toàn khác. Lệnh **wait()** được dùng để đồng bộ hóa, đảm bảo tiến trình cha dọn dẹp tài nguyên sau khi tiến trình con đã hoàn thành nhiệm vụ.

```

8 #include <stdio.h>
10 #include <unistd.h>
11 #include <sys/wait.h>
12 #include <sys/types.h>
13 int main(int argc, char* argv[])
14 {
15     _pid_t pid;
16     pid = fork();
17     if (pid > 0)
18     {
19         printf("PARENTS | PID = %ld | PPID = %ld\n", (long) getpid(), (long) getppid());
20         if (argc > 2)
21             printf("PARENTS | There are %d arguments\n", argc - 1);
22         wait(NULL);
23     }
24     if (pid == 0)
25     {
26         execl("./count.sh", "./count.sh", "10", NULL);
27
28         printf("CHILDREN | PID = %ld | PPID = %ld\n", (long) getpid(), (long) getppid());
29         printf("CHILDREN | List of arguments: \n");
30         for (int i = 1; i < argc; i++)
31             printf("%s\n", argv[i]);
32     }
33     exit(0);
34 }

```

```

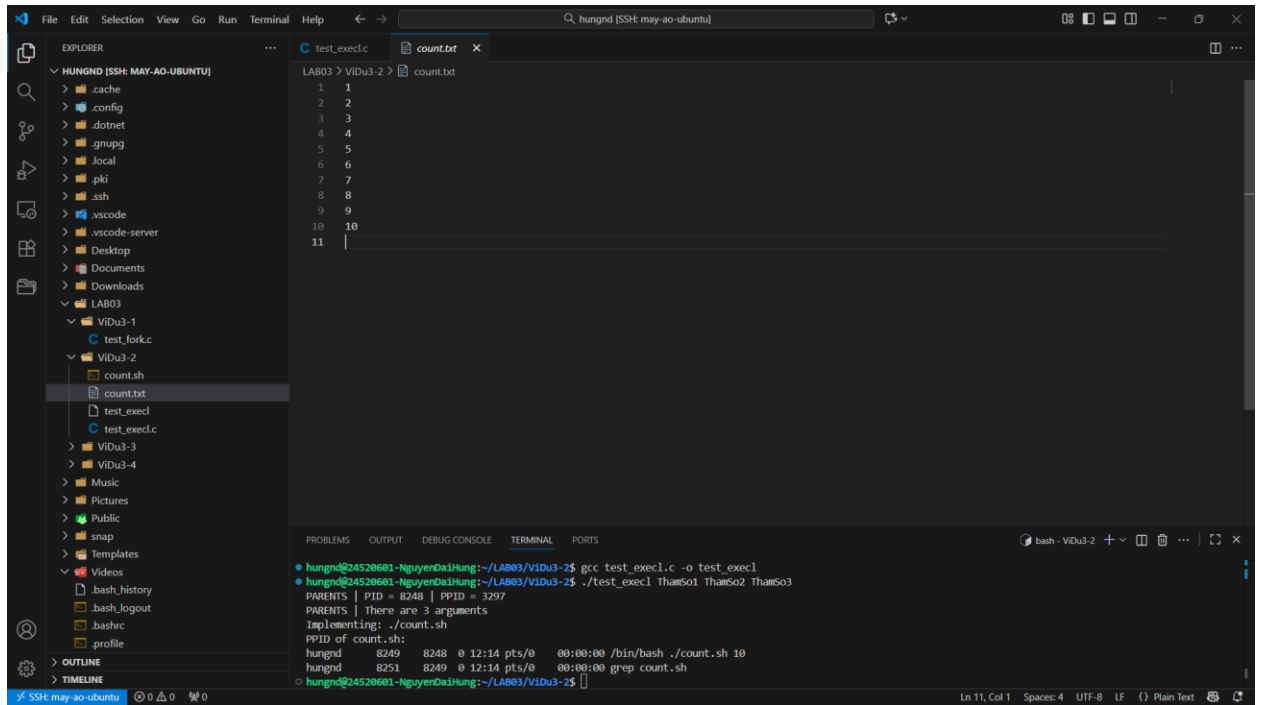
hungnd@24528681-NguyenDaiHung:~/LAB03/VIDu3-2$ gcc test_execl.c -o test_execl
hungnd@24528681-NguyenDaiHung:~/LAB03/VIDu3-2$ ./test_execl ThamSo1 ThamSo2 ThamSo3
PARENTS | PID = 8248 | PPID = 3297
PARENTS | There are 3 arguments
Implementing: ./count.sh
PPID of count.sh:
hungnd 8249 8248 0 12:14 pts/0 00:00:00 /bin/bash ./count.sh 10
hungnd 8251 8249 0 12:14 pts/0 00:00:00 ./count.sh
hungnd@24528681-NguyenDaiHung:~/LAB03/VIDu3-2$ grep count.sh

```

Hình 2.3: Kết quả chương trình ví dụ 3-2

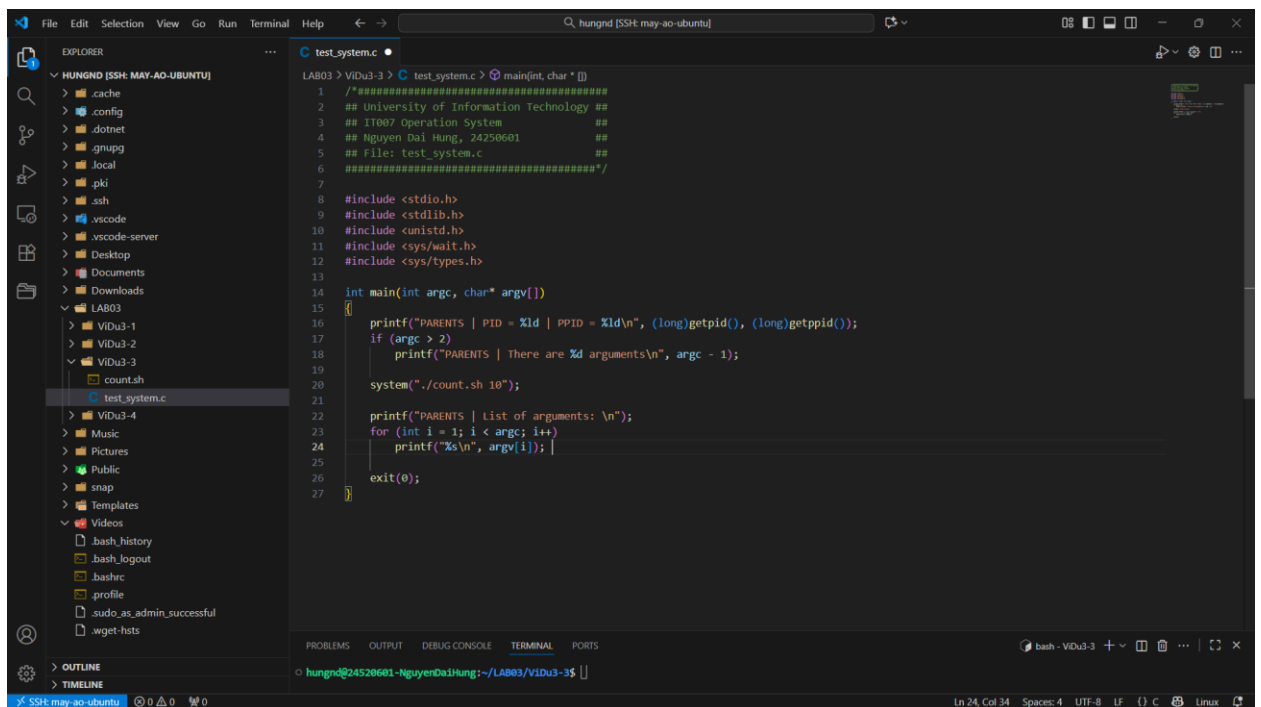
• **Giải thích code:**

- Khi ta ở trong tiến trình cha thì kết quả in ra lần lượt là
 - Dòng đầu tiên cho biết hiện tại đang ở tiến trình cha (**PARENTS**), **PID**(8248) và **PPID**(3297).
 - Dòng thứ hai cho biết tiến trình có 3 tham số được nạp vào (ThamSo1, ThamSo2, ThamSo3).
- `Wait(NULL)` được sử dụng để chờ đợi tiến trình con kết thúc và trả về thông tin về quá trình con đã kết thúc. Khi sử dụng `wait(NULL)` thì quá trình cha sẽ bị chặn (blocked) cho đến khi quá trình con kết thúc.
- Ở trong tiến trình con có dòng lệnh `execl("./count.sh", "./count.sh", "10", NULL)`.
 - “./count.sh” để in ra ở ngay sau “Implementing: “.
 - “10” được dùng để **count.sh** đếm từ 1 → 10.



Hình 2.4: Kết quả đếm được sẽ được ghi vào file count.txt (từ 1 → 10)

c) Ví dụ 3-3.



Hình 3.1: Source code ví dụ 3-3

- Giải thích code:
 - Dùng `printf()` để in ra **PARENTS**, **PID** và **PPID** của **PARENTS**.

- Xét nếu $argc > 2$ thì tiếp tục in **PARENTS** và số lượng tham số được truyền vào.
- Dòng `system("./count.sh 10")` để tạo mới hoàn toàn tiến trình `count.sh` cũng như truyền 2 tham số `$0` và `$1` cho `count.sh`.

```

1  /* ===== */
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <unistd.h>
5  #include <sys/wait.h>
6  #include <sys/types.h>
7
8  int main(int argc, char* argv[])
9  {
10     printf("PARENTS | PID = %d | PPID = %d\n", (long) getpid(), (long) getppid());
11     if (argc > 2)
12         printf("PARENTS | There are %d arguments\n", argc - 1);
13
14     system("./count.sh 10");
15
16     printf("PARENTS | List of arguments: \n");
17     for (int i = 1; i < argc; i++)
18         printf("%s\n", argv[i]);
19     exit(0);
20 }

```

```

hungnd@24528681-NguyenDaiHung:~/LAB03/VIDU3-3$ ./test_system ThamSo1 ThamSo2 ThamSo3
PARENTS | PID = 8630 | PPID = 3297
PARENTS | There are 3 arguments
Implementing: ./count.sh
PPID of count.sh:
hungnd 8631 8630 0 12:32 pts/0 00:00:00 sh -c -- ./count.sh 10
hungnd 8632 8631 0 12:32 pts/0 00:00:00 /bin/bash ./count.sh 10
hungnd 8634 8632 0 12:32 pts/0 00:00:00 grep count.sh
PARENTS | List of arguments:
ThamSo1
ThamSo2
ThamSo3

```

Hình 3.2: Kết quả chương trình ví dụ 3-3

• Giải thích code:

- In ra **PARENTS**, **PID(8630)** và **PPID(3297)** của tiến trình cha này.
- In ra **PARENTS**, số lượng tham số của tiến trình (3 tham số).
- Tham số `$0` đã được truyền cho `count.sh` nên dòng tiếp theo được in ra là **“Implementing: ./count.sh”**.
- Hàm `system("./count.sh 10")` đã được tạo mới hoàn toàn tiến trình `count.sh` nên tiến trình này sẽ được thực thi, đến khi kết thúc sẽ tiếp tục thực hiện những dòng code bên dưới, tức là in ra danh sách tham số (**PARENTS | List of arguments ...**) chứ không bị thay thế như ở ví dụ 3-2.

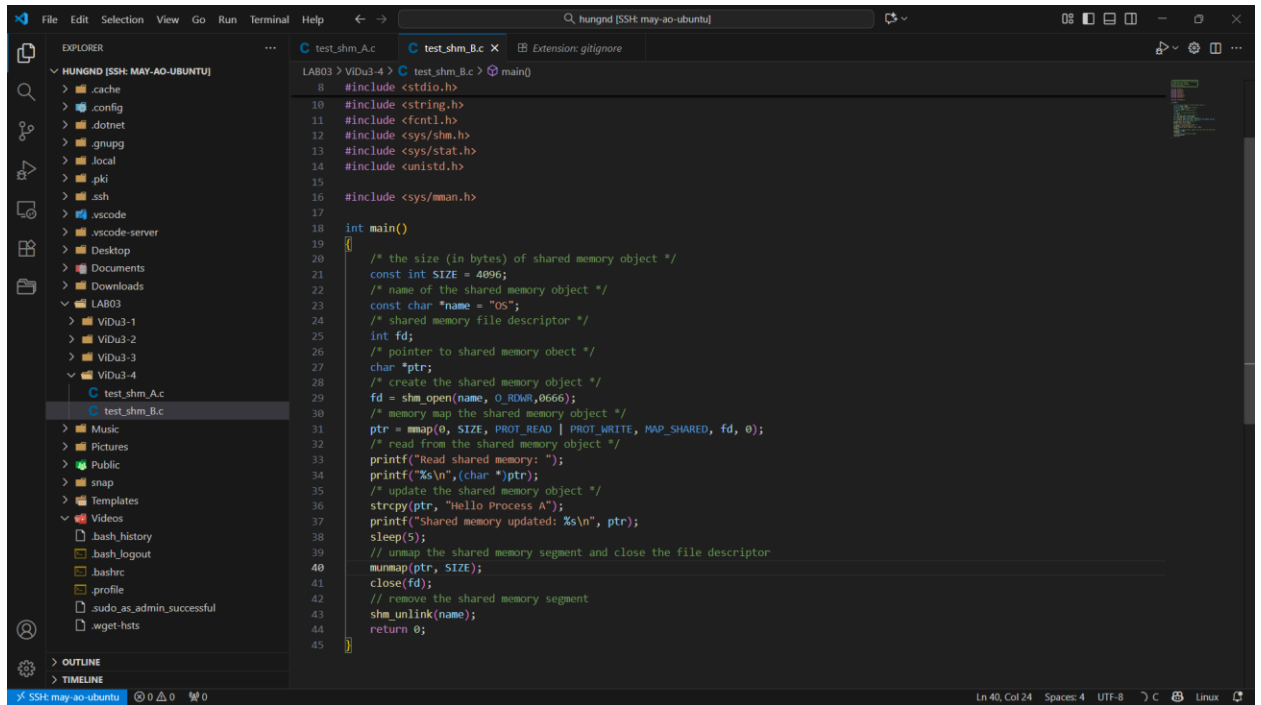
d) Ví dụ 3-4.

```
LAB03 > VIDu3-4 > test_shm_A.c > main()
8 #include <stdio.h>
13 #include <sys/stat.h>
14 #include <unistd.h>
15 #include <sys/mman.h>
16
17 int main()
18 {
19     /* the size (in bytes) of shared memory object */
20     const int SIZE = 4096;
21     /* name of the shared memory object */
22     const char *name = "OS";
23     /* shared memory file descriptor */
24     int fd;
25     /* pointer to shared memory object */
26     char *ptr;
27     /* create the shared memory object */
28     fd = shm_open(name, O_CREAT | O_RDWR, 0666);
29     /* configure the size of the shared memory object */
30     ftruncate(fd, SIZE);
31     /* memory map the shared memory object */
32     ptr = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
33     /* write to the shared memory object */
34     strcpy(ptr, "Hello Process B");
35     /* wait until Process B updates the shared memory segment */
36     while (strncmp(ptr, "Hello Process B", 15) == 0)
37     {
38         printf("Waiting Process B update shared memory\n");
39         sleep(1);
40     }
41     printf("Memory updated: %s\n", (char *)ptr);
42     /* unmap the shared memory segment and close the file descriptor */
43     munmap(ptr, SIZE);
44     close(fd);
45
46     return 0;
47 }
```

Hình 4.1: Source code Process A ví dụ 3-4

• **Giải thích code:**

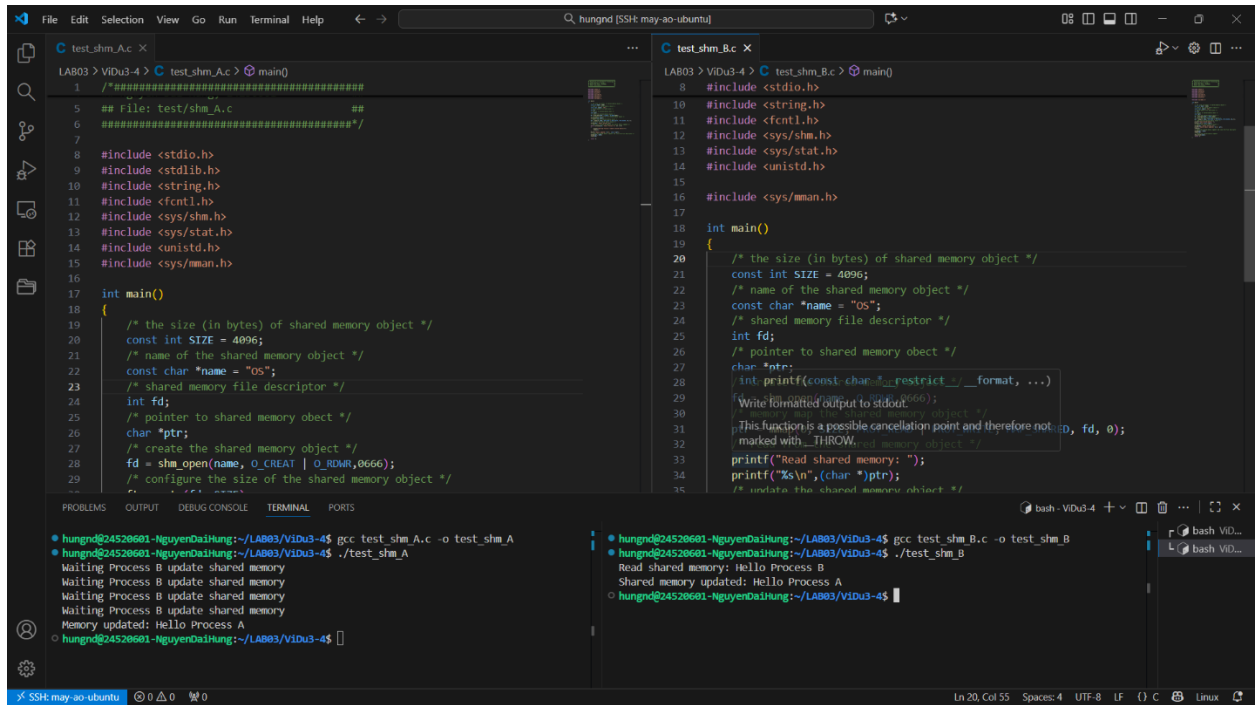
- Đoạn code này là Tiến trình A, một phần của cơ chế Giao tiếp Liên Tiến trình (IPC) sử dụng Vùng nhớ chia sẻ (Shared Memory), thay vì dùng fork().
 - Logic của Tiến trình A: Nó khởi tạo (shm_open với cờ O_CREAT) và thiết lập kích thước (ftruncate) cho một vùng nhớ chia sẻ có tên là "OS". Sau đó, nó ánh xạ (mmap) vùng nhớ đó vào không gian địa chỉ của mình để lấy con trỏ ptr.
 - Gửi dữ liệu: Nó ghi (strcpy) chuỗi "Hello Process B" vào vùng nhớ chia sẻ.
 - Đồng bộ hóa/Chờ: Nó lập tức vào một vòng lặp while để chờ (sleep). Vòng lặp này liên tục kiểm tra xem nội dung tại ptr có còn là "Hello Process B" hay không. Chừng nào nội dung chưa bị thay đổi, nó vẫn tiếp tục chờ.
 - Nhận dữ liệu: Khi một Tiến trình B (chạy riêng biệt) truy cập và ghi đè nội dung mới vào vùng nhớ "OS", điều kiện strncmp sẽ sai và vòng lặp while của Tiến trình A kết thúc. Tiến trình A sau đó in ra nội dung mới mà Tiến trình B đã cập nhật.
- Mục đích chính của mã là minh họa cách một tiến trình khởi tạo, ghi dữ liệu và sử dụng vòng lặp chờ (busy-waiting) để đồng bộ hóa và nhận dữ liệu từ một tiến trình khác thông qua Vùng nhớ chia sẻ



Hình 4.2: Source code Process B ví dụ 3-4

- **Giải thích code:**

- Đoạn mã này là Tiến trình B, chạy độc lập và song song với Tiến trình A để thực hiện giao tiếp qua Vùng nhớ chia sẻ (Shared Memory).
 - Logic của Tiến trình B: Nó mở (shm_open) vùng nhớ chia sẻ có tên "OS" (đã được tạo bởi Tiến trình A). Nó cũng ánh xạ (mmap) vùng nhớ đó vào không gian địa chỉ của mình để lấy con trỏ ptr.
 - Nhận dữ liệu: Nó đọc và in ra nội dung ban đầu mà Tiến trình A đã ghi ("Hello Process B").
 - Gửi dữ liệu: Ngay sau đó, nó ghi đè (strcpy) chuỗi "Hello Process A" vào lại vùng nhớ chia sẻ. Hành động này chính là tín hiệu "trả lời" mà vòng lặp while của Tiến trình A đang chờ đợi.
 - Dọn dẹp: Sau khi tạm dừng (sleep), Tiến trình B giải phóng (munmap, close) và quan trọng nhất là xóa bỏ (shm_unlink) vùng nhớ chia sẻ khỏi hệ thống.
- Mục đích chính của mã là minh họa cách một tiến trình thứ hai truy cập, đọc, và cập nhật một vùng nhớ chia sẻ đã tồn tại, hoàn thành chu trình giao tiếp hai chiều và thực hiện vai trò dọn dẹp tài nguyên hệ thống.

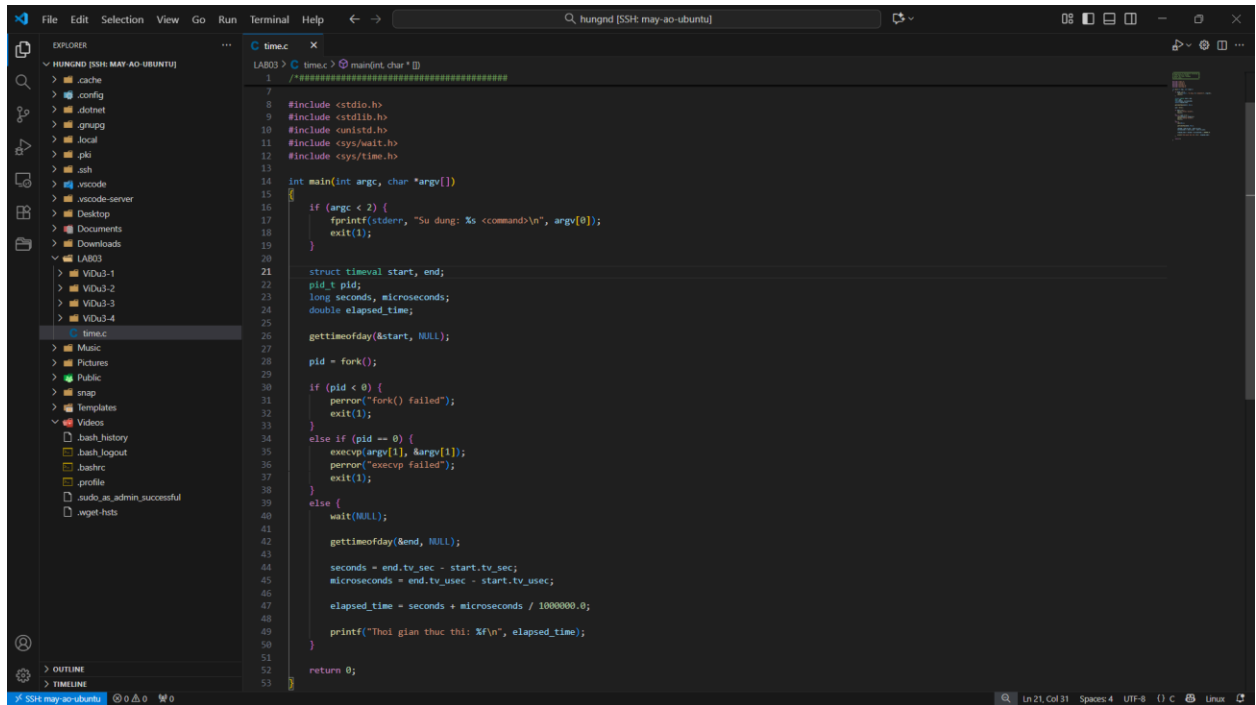


Hình 4.3: Kết quả chương trình ví dụ 3-4

- **Giải thích kết quả:**

- Biên dịch 2 file `test_shm_A.c` và `test_shm_B.c` rồi chia 2 terminal, một bên thực thi `test_shm_A`, một bên thực thi `test_shm_B`.
- Khi thực thi `test_shm_A`, màn hình sẽ liên tục in ra các dòng “Waiting Process B update shared memory” tức đang đợi tiến trình B vào để cập nhật ở shared memory.
- Ở phía bên phải, sau khi tiến hành thực thi `test_shm_B` màn hình sẽ xuất hiện 2 dòng chữ “Read shared memory: Hello Process B” và “Shared memory updated: Hello Process A”.
- Ngay lúc này, ở phía terminal bên trái cũng sẽ dừng và in ra dòng lệnh “Memory updated: Hello Process A” tức B đã vào và cập nhật shared memory thành “Hello Process A”.

2. Viết chương trình `time.c` thực hiện đo thời gian thực thi của một lệnh shell. Chương trình sẽ được chạy với cú pháp “`./time <command>`” với `<command>` là lệnh shell muốn đo thời gian thực thi.

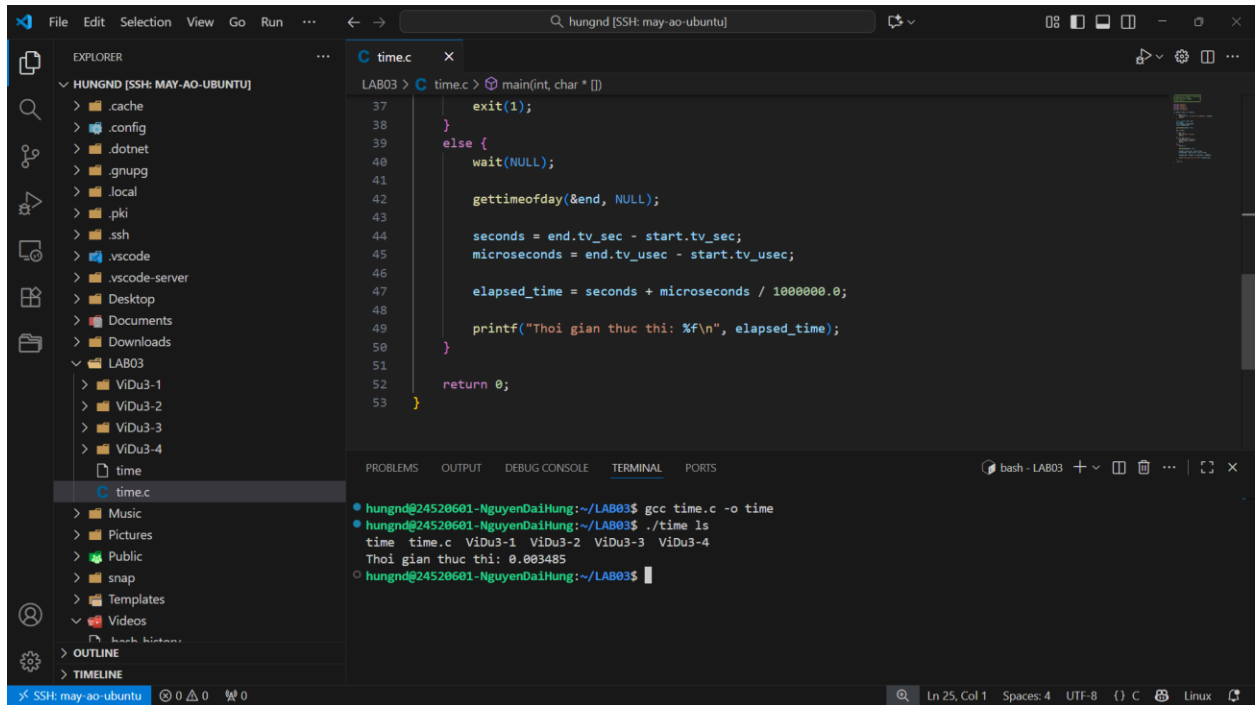


```
1 1
2 2
3 3
4 4
5 5
6 6
7 7
8 #include <stdio.h>
9 #include <stdlib.h>
10 #include <unistd.h>
11 #include <sys/wait.h>
12 #include <sys/time.h>
13
14 int main(int argc, char *argv[])
15 {
16     if (argc < 2) {
17         fprintf(stderr, "Su dung: %s <command>\n", argv[0]);
18         exit(1);
19     }
20
21     struct timeval start, end;
22     pid_t pid;
23     long seconds, microseconds;
24     double elapsed_time;
25
26     gettimeofday(&start, NULL);
27
28     pid = fork();
29
30     if (pid < 0) {
31         perror("fork() failed");
32         exit(1);
33     }
34     else if (pid == 0) {
35         execvp(argv[1], &argv[1]);
36         perror("execvp failed");
37         exit(1);
38     }
39     else {
40         wait(NULL);
41
42         gettimeofday(&end, NULL);
43
44         seconds = end.tv_sec - start.tv_sec;
45         microseconds = end.tv_usec - start.tv_usec;
46         elapsed_time = seconds + microseconds / 1000000.0;
47
48         printf("Thoi gian thuc thi: %f\n", elapsed_time);
49
50     }
51
52     return 0;
53 }
```

Hình 2.1: Source code time.c

- **Giải thích code:**

- Đoạn mã sử dụng lời gọi hệ thống `fork()` để tạo ra một tiến trình con, và dùng `gettimeofday()` để đo lường thời gian.
 - Logic của Tiến trình Cha (`pid > 0`): Tiến trình cha lấy mốc thời gian bắt đầu (`gettimeofday`) trước khi tạo con. Sau đó, nó gọi `wait(NULL)` để tạm dừng thực thi và chờ cho đến khi tiến trình con kết thúc. Ngay khi con kết thúc, cha lấy mốc thời gian kết thúc, tính toán khoảng thời gian đã trôi qua và in kết quả ra màn hình.
 - Logic của Tiến trình Con (`pid == 0`): Nhiệm vụ chính của tiến trình con là sử dụng `execvp()` để thay thế mã lệnh của chính nó bằng lệnh shell mà người dùng đã truyền vào (ví dụ: `ls` hoặc `sleep 5`). Tiến trình con chính là thứ "bị" đo thời gian.
- Mục đích chính của mã là minh họa việc sử dụng `fork` và `wait` để đo lường thời gian thực thi (wall-clock time) của một lệnh shell bên ngoài, bằng cách tính thời gian từ lúc cha chuẩn bị chạy lệnh cho đến khi lệnh đó hoàn thành.



The screenshot shows a VS Code editor window with a file explorer on the left and a code editor in the center. The file explorer shows a directory structure for 'HUNGND [SSH: MAY-AO-UBUNTU]' with subdirectories like '.cache', '.config', '.dotnet', '.gnupg', '.local', '.pki', '.ssh', '.vscode', '.vscode-server', 'Desktop', 'Documents', 'Downloads', 'LAB03', 'time', 'time.c', 'ViDu3-1', 'ViDu3-2', 'ViDu3-3', 'ViDu3-4', 'time', 'time.c', 'Music', 'Pictures', 'Public', 'snap', 'Templates', 'Videos', 'OUTLINE', and 'TIMELINE'. The code editor shows the following C code:

```
LAB03 > C time.c > main(int, char * [])
37     exit(1);
38 }
39 else {
40     wait(NULL);
41
42     gettimeofday(&end, NULL);
43
44     seconds = end.tv_sec - start.tv_sec;
45     microseconds = end.tv_usec - start.tv_usec;
46
47     elapsed_time = seconds + microseconds / 1000000.0;
48
49     printf("Thời gian thực thi: %f\n", elapsed_time);
50 }
51
52 return 0;
53 }
```

The terminal output shows the following commands and results:

```
hungnd@24520601-NguyenDaiHung:~/LAB03$ gcc time.c -o time
hungnd@24520601-NguyenDaiHung:~/LAB03$ ./time ls
time time.c ViDu3-1 ViDu3-2 ViDu3-3 ViDu3-4
Thời gian thực thi: 0.003485
hungnd@24520601-NguyenDaiHung:~/LAB03$
```

Hình 2.2: Kết quả chương trình *time.c*

• **Giải thích kết quả:**

- *Lệnh ./time ls: Người dùng thực thi chương trình time và truyền ls làm đối số (command).*
- *Output 1 (Danh sách file): Đây là kết quả từ tiến trình con. Tiến trình cha đã fork() thành công. Tiến trình con (pid == 0) đã gọi execvp() để thay thế chính nó bằng lệnh ls. Lệnh ls chạy, liệt kê các file trong thư mục (time, time.c, ViDu3-1...).*
- *Output 2 ("Thời gian thực thi..."): Đây là kết quả từ tiến trình cha. Sau khi fork(), tiến trình cha (pid > 0) gọi wait(NULL) và tạm dừng. Khi tiến trình con (lệnh ls) kết thúc, wait() trả về. Tiến trình cha tiếp tục chạy, tính toán thời gian chênh lệch (từ trước fork đến sau wait) và in ra 0.003485 giây, là thời gian thực thi của lệnh ls.*

3. Viết một chương trình làm bốn công việc sau theo thứ tự:

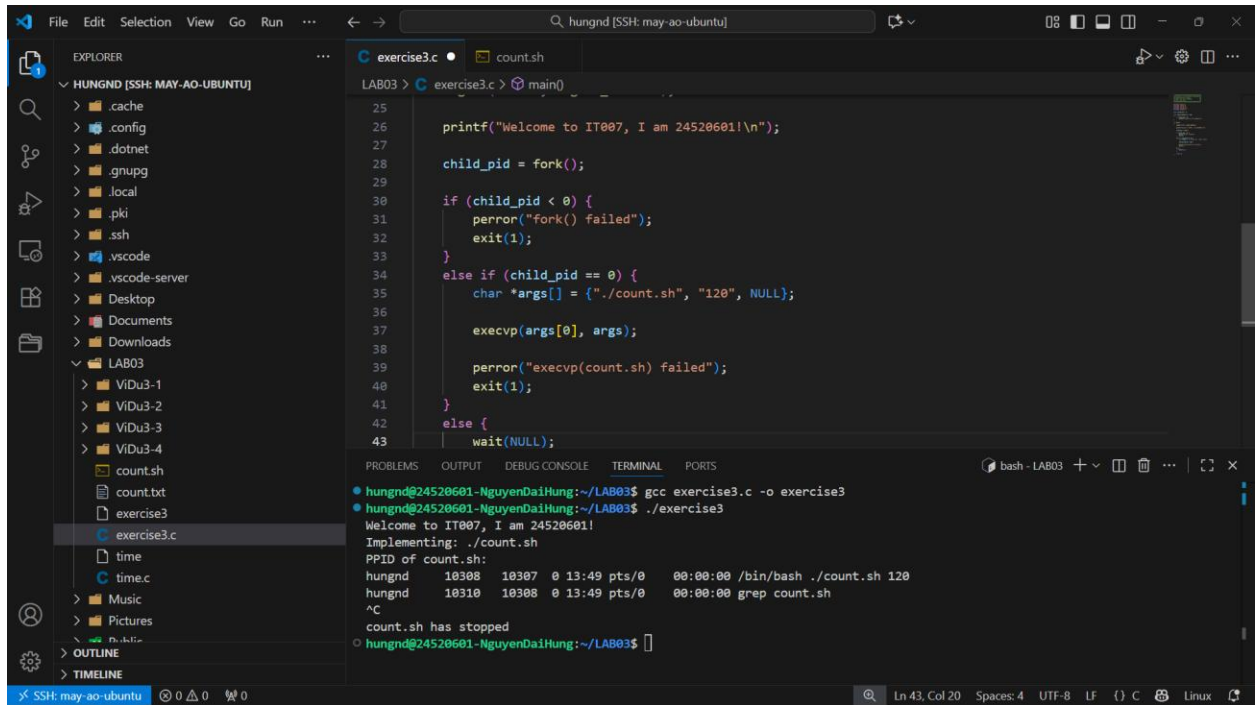
- **In ra dòng chữ: “Welcome to IT007, I am <your_Student_ID>!”**
- **Thực thi file script count.sh với số lần đếm là 120**
- **Trước khi count.sh đếm đến 120, bấm CTRL+C để dừng tiến trình này**
- **Khi người dùng nhấn CTRL+C thì in ra dòng chữ: “count.sh has stopped”**

```
LAB03 > exercise3.c
1 #include <stdio.h>
2 #include <unistd.h>
3 #include <sys/wait.h>
4 #include <signal.h>
5
6 pid_t child_pid = -1;
7
8 void sigint_handler(int sig)
9 {
10     if (child_pid > 0)
11         printf("\ncount.sh has stopped\n");
12 }
13
14 int main()
15 {
16     signal(SIGINT, sigint_handler);
17
18     printf("Welcome to IT007, I am 24520601!\n");
19
20     child_pid = fork();
21
22     if (child_pid < 0) {
23         perror("fork() failed");
24         exit(1);
25     }
26     else if (child_pid == 0) {
27         char *args[] = {"/count.sh", "120", NULL};
28
29         execvp(args[0], args);
30
31         perror("execvp(count.sh) failed");
32         exit(1);
33     }
34     else {
35         wait(NULL);
36     }
37
38     return 0;
39 }
```

Hình 3.1: Source code exercise3.c

• **Giải thích code:**

- Đoạn mã sử dụng lời gọi hệ thống `fork()` để tạo ra một tiến trình con và `signal()` để xử lý tín hiệu ngắt. Chương trình bắt đầu bằng việc in ra "Welcome to IT007, I am 24520601!".
 - Logic của Tiến trình Cha ($pid > 0$): Tiến trình cha, trước khi `fork`, đã đăng ký một hàm xử lý là `sigint_handler` để "bắt" tín hiệu CTRL+C (SIGINT). Sau khi `fork`, nó gọi `wait(NULL)` để tạm dừng và chờ cho đến khi tiến trình con kết thúc. Khi người dùng nhấn CTRL+C, tín hiệu này sẽ kích hoạt hàm `sigint_handler`, khiến tiến trình cha in ra "count.sh has stopped".
 - Logic của Tiến trình Con ($pid == 0$): Nhiệm vụ chính của tiến trình con là gọi hàm `execvp()` để thay thế toàn bộ mã lệnh của nó bằng file script `./count.sh`, đồng thời truyền vào tham số "120" (yêu cầu đếm 120 lần).
- Mục đích chính của mã là minh họa cách một tiến trình cha có thể chạy một chương trình khác (thông qua `fork` và `exec`) và đồng thời sử dụng `signal` để bắt một sự kiện ngắt (như CTRL+C), cho phép nó thực hiện một hành động tùy chỉnh (in thông báo) khi tiến trình con bị dừng.



```
25
26 printf("Welcome to IT007, I am 24520601!\n");
27
28 child_pid = fork();
29
30 if (child_pid < 0) {
31     perror("fork() failed");
32     exit(1);
33 }
34 else if (child_pid == 0) {
35     char *args[] = { "./count.sh", "120", NULL };
36
37     execvp(args[0], args);
38
39     perror("execvp(count.sh) failed");
40     exit(1);
41 }
42 else {
43     wait(NULL);
44 }
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

hungnd@24520601-NguyenDaiHung:~/LAB03\$ gcc exercise3.c -o exercise3

hungnd@24520601-NguyenDaiHung:~/LAB03\$./exercise3

Welcome to IT007, I am 24520601!

Implementing: ./count.sh

PPID of count.sh:

USER	PID	TID	TIME	PTS	TH	COMM
hungnd	10308	10307	0	13:49	pts/0	/bin/bash ./count.sh 120
hungnd	10310	10308	0	13:49	pts/0	00:00:00 grep count.sh

^C

count.sh has stopped

hungnd@24520601-NguyenDaiHung:~/LAB03\$

Hình 3.2: Kết quả chương trình exercise3

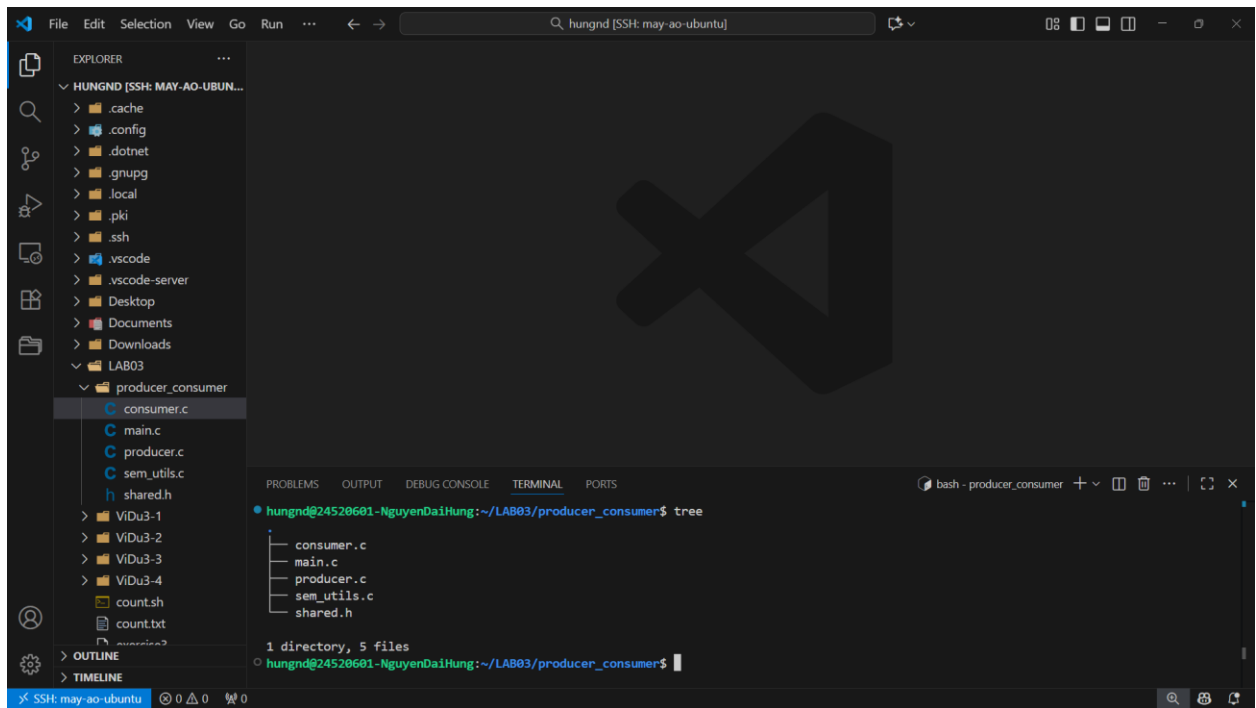
• Giải thích kết quả:

- Đầu tiên, biên dịch file `exercise3.c` với lệnh `gcc exercise3.c -o exercise3` để tạo file thực thi.
- Khi thực thi `./exercise3`, màn hình trước hết in ra dòng "Welcome to IT007, I am 24520601!". Đây là output từ tiến trình cha.
- Ngay sau đó, các dòng "Implementing: ./count.sh" và kết quả lệnh `ps` xuất hiện. Đây là output từ tiến trình con, cho thấy nó đã exec thành công script `count.sh` (PID 10308) và xác nhận PPID của nó là 10307 (chính là tiến trình cha).
- Khi người dùng nhấn ^C (tức CTRL+C), tín hiệu SIGINT được gửi đến cả hai tiến trình.
- Ngay lúc này, ở terminal, tiến trình con (`count.sh`) bị dừng (đây là hành vi mặc định khi nhận SIGINT). Tiến trình cha (đang wait) thì đã bắt được tín hiệu này, nó liền chạy hàm `sigint_handler` và in ra dòng "count.sh has stopped" rồi kết thúc chương trình.

4. Viết chương trình mô phỏng bài toán Producer – Consumer như sau:

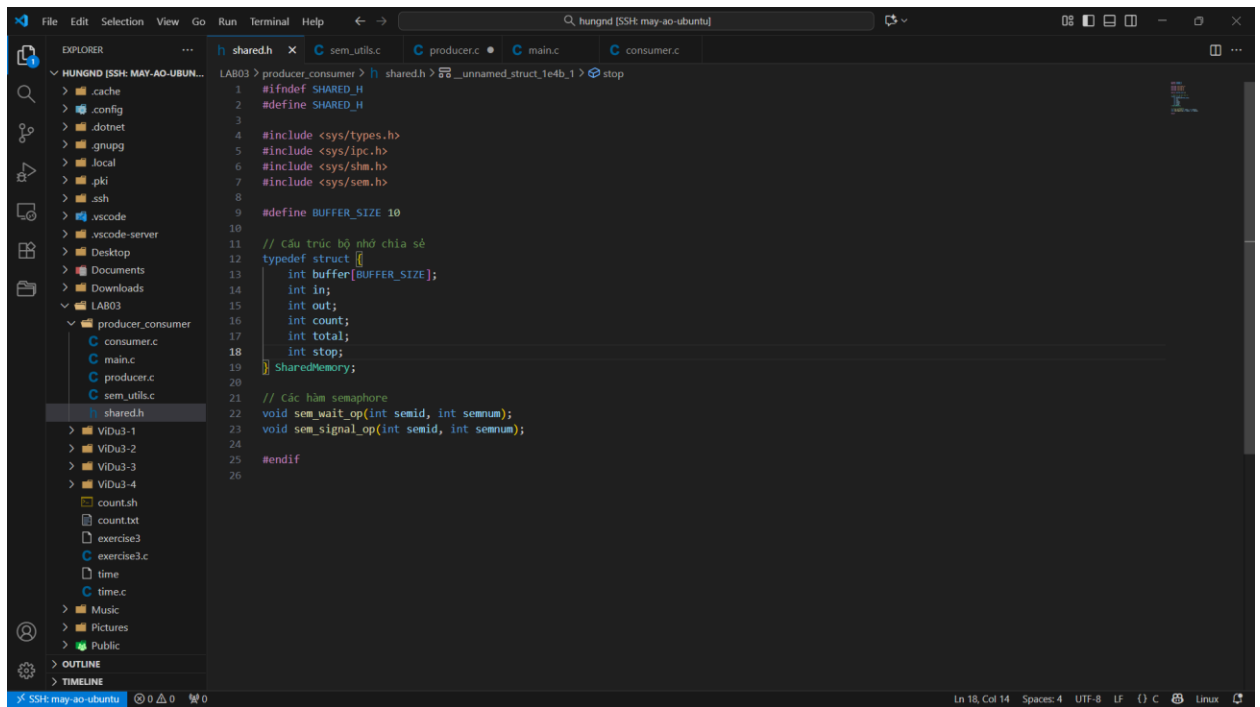
- Sử dụng kỹ thuật shared-memory để tạo một bounded-buffer có độ lớn là 10 bytes. 44
- Tiến trình cha đóng vai trò là Producer, tạo một số ngẫu nhiên trong khoảng [10, 20] và ghi dữ liệu vào buffer
- Tiến trình con đóng vai trò là Consumer đọc dữ liệu từ buffer, in ra màn hình và tính tổng

- Khi tổng lớn hơn 100 thì cả 2 dừng lại



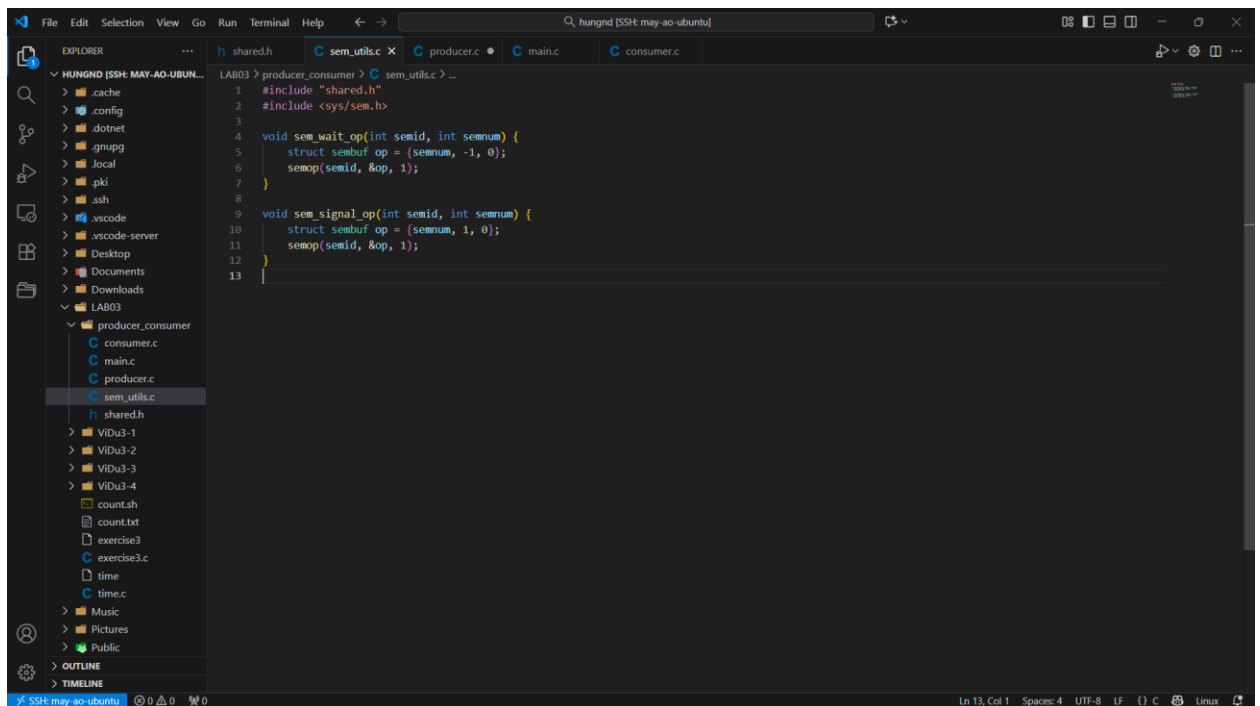
Hình 4.1: Cây thư mục *producer_consumer*

- Giải thích cấu trúc thư mục:
 - ***main.c***: chương trình chính tạo ra *shared memory*, *semaphore*, *fork* tiến trình.
 - ***producer.c***: code của tiến trình cha – *Producer*.
 - ***consumer.c***: code của tiến trình con – *Consumer*.
 - ***shared.h***: khai báo cấu trúc vùng nhớ chia sẻ và các hàm *semaphore*.
 - ***sem_utils.c***: cài đặt thao tác *semaphore*.



```
1 #ifndef SHARED_H
2 #define SHARED_H
3
4 #include <sys/types.h>
5 #include <sys/ipc.h>
6 #include <sys/shm.h>
7 #include <sys/sem.h>
8
9 #define BUFFER_SIZE 10
10
11 // Cấu trúc bộ nhớ chia sẻ
12 typedef struct {
13     int buffer[BUFFER_SIZE];
14     int in;
15     int out;
16     int count;
17     int total;
18     int stop;
19 } SharedMemory;
20
21 // Các hàm semaphore
22 void sem_wait_op(int semid, int semnum);
23 void sem_signal_op(int semid, int semnum);
24
25 #endif
26
```

Hình 4.2: Source code file shared.h



```
1 #include "shared.h"
2 #include <sys/sem.h>
3
4 void sem_wait_op(int semid, int semnum) {
5     struct sembuf op = {semnum, -1, 0};
6     semop(semid, &op, 1);
7 }
8
9 void sem_signal_op(int semid, int semnum) {
10     struct sembuf op = {semnum, 1, 0};
11     semop(semid, &op, 1);
12 }
13
```

Hình 4.3: Source code file sem_utils.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <time.h>
5 #include "shared.h"
6
7 void producer(sharedMemory *shm, int semid) {
8     srand(time(NULL) + getpid());
9
10    while (1) {
11        if (shm->stop) break;
12
13        int num = rand() % 11 + 10; // [10,20]
14
15        sem_wait_op(semid, 1); // wait empty
16        sem_wait_op(semid, 0); // wait mutex
17
18        if (shm->stop) {
19            sem_signal_op(semid, 0);
20            sem_signal_op(semid, 1);
21            break;
22        }
23
24        shm->buffer[shm->in] = num;
25        shm->in = (shm->in + 1) % BUFFER_SIZE;
26        shm->count++;
27
28        printf("Producer: produced %d\n", num);
29
30        sem_signal_op(semid, 0); // signal mutex
31        sem_signal_op(semid, 2); // signal full
32
33        sleep(1);
34    }
35 }
36

```

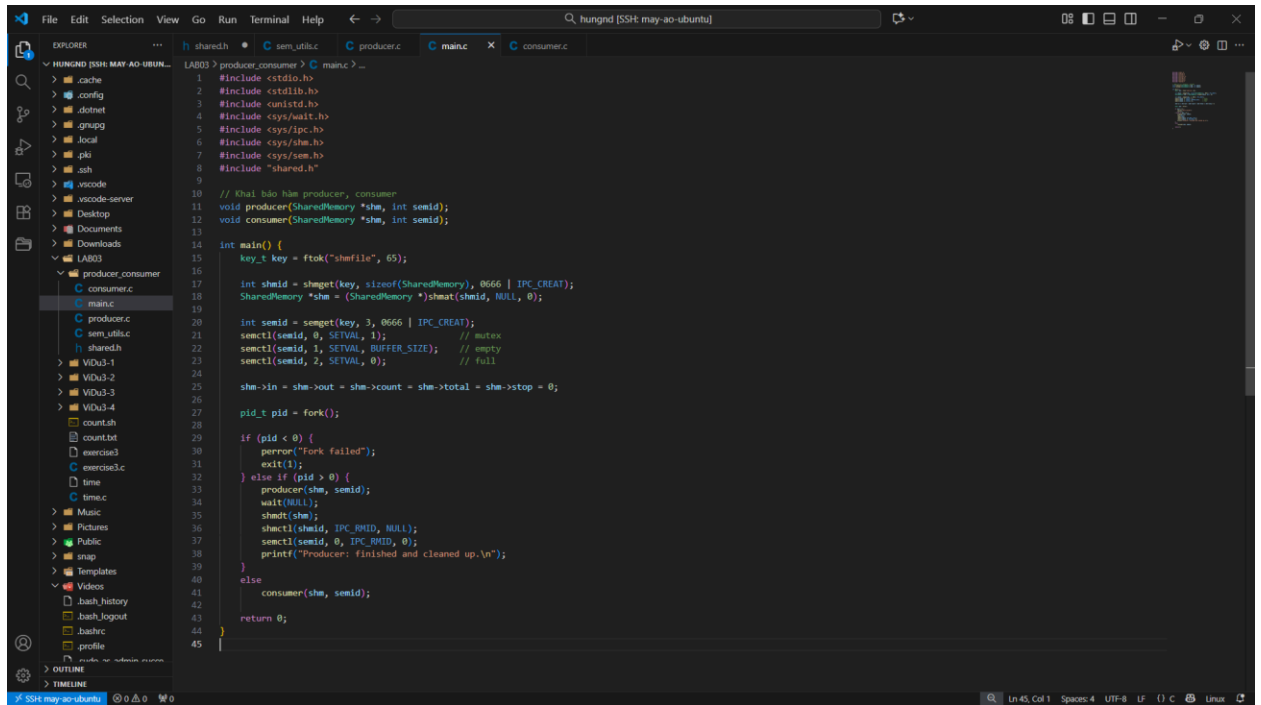
Hình 4.4: Source code file producer.c

```

1 #include <stdio.h>
2 #include <unistd.h>
3 #include "shared.h"
4
5 void consumer(sharedMemory *shm, int semid) {
6     while (1) {
7         sem_wait_op(semid, 2); // wait full
8         sem_wait_op(semid, 0); // wait mutex
9
10        if (shm->count > 0) {
11            int num = shm->buffer[shm->out];
12            shm->out = (shm->out + 1) % BUFFER_SIZE;
13            shm->count--;
14            shm->total += num;
15
16            printf("Consumer: consumed %d, total = %d\n", num, shm->total);
17
18            if (shm->total > 100) {
19                printf("consumer: total > 100, stopping...\n");
20                shm->stop = 1;
21                sem_signal_op(semid, 0);
22                sem_signal_op(semid, 1);
23                break;
24            }
25
26            sem_signal_op(semid, 0);
27            sem_signal_op(semid, 1);
28            sleep(1);
29        }
30    }
31 }
32

```

Hình 4.5: Source code file consumer.c



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <unistd.h>
4 #include <sys/wait.h>
5 #include <sys/ipc.h>
6 #include <sys/shm.h>
7 #include <sys/sem.h>
8 #include "shared.h"
9
10 // Khai báo hàm producer, consumer
11 void producer(SharedMemory *shm, int semid);
12 void consumer(SharedMemory *shm, int semid);
13
14 int main() {
15     key_t key = ftok("shmfile", 65);
16
17     int shmid = shmget(key, sizeof(SharedMemory), 0666 | IPC_CREAT);
18     SharedMemory *shm = (SharedMemory *)shmat(shmid, NULL, 0);
19
20     int semid = semget(key, 3, 0666 | IPC_CREAT);
21     semctl(semid, 0, SETVAL, 1); // mutex
22     semctl(semid, 1, SETVAL, BUFFER_SIZE); // empty
23     semctl(semid, 2, SETVAL, 0); // full
24
25     shm->in = shm->out = shm->count = shm->total = shm->stop = 0;
26
27     pid_t pid = fork();
28
29     if (pid < 0) {
30         perror("fork failed");
31         exit(1);
32     } else if (pid > 0) {
33         producer(shm, semid);
34         wait(NULL);
35         shmdt(shm);
36         shmctl(shmid, IPC_RMID, NULL);
37         semctl(semid, 0, IPC_RMID, 0);
38         printf("Producer: finished and cleaned up.\n");
39     } else {
40         consumer(shm, semid);
41     }
42     return 0;
43 }
44
45 }
```

Hình 4.6: Source code file main.c

• **Giải thích code:**

- *Đoạn mã sử dụng lời gọi hệ thống fork() để tạo ra hai tiến trình:*
 - *Tiến trình cha đóng vai trò là Producer, và tiến trình con đóng vai trò là Consumer.*
 - *Hai tiến trình này trao đổi dữ liệu thông qua vùng nhớ chia sẻ (shared memory), và được đồng bộ hóa bằng semaphore.*
- *Logic của Tiến trình Cha (pid > 0):*
 - *Tiến trình cha tạo và khởi tạo vùng nhớ chia sẻ (shmget, shmat) và bộ semaphore (semget, semctl).*
 - *Sau khi fork(), tiến trình cha đóng vai trò Producer, liên tục sinh số ngẫu nhiên trong khoảng [10, 20].*
 - *Trước khi ghi dữ liệu, Producer phải:*
 - *wait(empty) để đảm bảo buffer còn chỗ trống.*
 - *wait(mutex) để chiếm quyền truy cập vùng găng.*
 - *Sau đó ghi số vào buffer, cập nhật chỉ số in, rồi:*
 - *signal(mutex) để trả quyền truy cập.*
 - *signal(full) để báo cho Consumer biết có dữ liệu mới.*
 - *Khi chờ stop trong vùng nhớ được đặt (do Consumer báo dừng), Producer thoát vòng lặp và giải phóng bộ nhớ, semaphore.*

- *Logic của Tiến trình Con (pid == 0):*
 - *Tiến trình con đóng vai trò Consumer, đọc dữ liệu từ vùng nhớ chia sẻ.*
 - *Trước khi đọc, Consumer thực hiện:*
 - *wait(full) để đảm bảo có dữ liệu trong buffer.*
 - *wait(mutex) để truy cập vùng găng.*
 - *Sau khi đọc xong, cập nhật chỉ số out, giảm count, và cộng dồn vào total.*
 - *Nếu total > 100, Consumer đặt stop = 1 để báo hiệu cho Producer dừng lại.*
 - *Cuối cùng, giải phóng semaphore:*
 - *signal(mutex) và signal(empty) để cho phép Producer ghi thêm khi cần.*
- *Mục đích chính của mã là mô phỏng quá trình giao tiếp và đồng bộ giữa hai tiến trình bằng cơ chế IPC (Inter-Process Communication). Thông qua việc sử dụng shared memory và semaphore, chương trình đảm bảo rằng:*
 - *Hai tiến trình không truy cập đồng thời vào vùng dữ liệu chung (tránh race condition).*
 - *Dữ liệu được truyền đúng thứ tự và không bị ghi đè.*
 - *Khi tổng vượt quá 100, cả hai tiến trình đều dừng hoạt động an toàn.*
- **Cơ chế hoạt động:**
 - *Chương trình mô phỏng bài toán Producer – Consumer bằng bộ nhớ chia sẻ (shared memory) và semaphore để đồng bộ hai tiến trình.*
 - *Producer (tiến trình cha): sinh số ngẫu nhiên trong khoảng [10, 20] và ghi vào vùng nhớ chia sẻ nếu buffer còn trống.*
 - *Consumer (tiến trình con): đọc dữ liệu từ buffer, in ra màn hình và cộng dồn tổng.*
 - *Ba semaphore được sử dụng:*
 - *mutex: đảm bảo chỉ một tiến trình truy cập vùng nhớ tại một thời điểm.*
 - *empty: đếm số ô trống trong buffer.*
 - *full: đếm số ô đã có dữ liệu.*
 - *Producer chỉ ghi khi empty > 0, còn Consumer chỉ đọc khi full > 0. Hai tiến trình luân phiên hoạt động thông qua cơ chế wait/signal của semaphore.*

- Khi tổng các số đọc được vượt quá 100, Consumer đặt cờ stop = 1, báo cho Producer dừng, sau đó cả hai tiến trình kết thúc và vùng nhớ được giải phóng.

The screenshot shows a Visual Studio Code editor with the file `producer_consumer.c` open. The code defines a shared memory region and implements a producer-consumer algorithm using semaphores. The terminal output shows the program running successfully, with the producer and consumer processes alternating and producing/consuming items. The final output is:

```

Producer: produced 19
Consumer: consumed 19, total = 19
Producer: produced 10
Consumer: consumed 10, total = 29
Producer: produced 11
Consumer: consumed 11, total = 40
Producer: produced 20
Consumer: consumed 20, total = 60
Producer: produced 16
Consumer: consumed 16, total = 76
Producer: produced 12
Consumer: consumed 12, total = 88
Producer: produced 11
Consumer: consumed 11, total = 99
Producer: produced 11
Consumer: consumed 11, total = 110
Consumer: total > 100, stopping...
Producer: finished and cleaned up.

```

Hình 4.7: Kết quả chương trình producer_consumer

• Giải thích kết quả:

- Khi chương trình được biên dịch và chạy thành công bằng lệnh: **gcc main.c producer.c consumer.c sem_utils.c -o pc -Wall** và **./pc** kết quả hiện thị như hình cho thấy hai tiến trình cha và con hoạt động luân phiên nhau:

- Tiến trình Producer (cha) sinh ra các số ngẫu nhiên trong khoảng [10, 20] và ghi vào vùng nhớ chia sẻ (shared memory).

Producer: produced 19

- Tiến trình Consumer (con) đọc số từ buffer, in ra màn hình và cộng dồn tổng. Mỗi lần đọc sẽ hiển thị giá trị vừa tiêu thụ và tổng hiện tại:

Consumer: consumer 19, total = 19

- Hai tiến trình được đồng bộ hóa bằng semaphore, nhờ đó dữ liệu không bị ghi đè hoặc đọc sai thứ tự. Có thể thấy Producer và Consumer luân phiên nhau hoạt động rất ổn định:

Producer → Consumer → Producer → Consumer ...

- Khi tổng vượt quá 100, Consumer in ra thông báo:

Consumer: total > 100, stopping ...

- Đồng thời đặt cờ $stop = 1$ trong vùng nhớ chia sẻ để báo cho Producer dừng lại.
- Sau đó Producer nhận tín hiệu dừng và in ra:

Producer: finished and cleaned up.

⇒ Điều này xác nhận vùng nhớ chia sẻ và semaphore đã được giải phóng đúng cách, chương trình kết thúc an toàn.