ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC & THUẬT MÁY TÍNH

HỆ ĐIỀU HÀNH (CO2017)

THỰC HÀNH 2

PROCESS

Lớp: L02 – Học Kỳ HK251
Ngày nộp: 11/11/2025

GVHD: Thầy Nguyễn Minh Tâm

Sinh Viên	MSSV
Phạm Công Võ	2313946

Ho Chi Minh, Tháng 11/2025

Mục lục

Danh sách hình vẽ

1 PROBLEM 1

1.1 Mô tả bài toán

Mục tiêu của bài tập là xây dựng một chương trình có khả năng tính toán điểm trung bình của các bộ phim dựa trên một tập dữ liệu lớn, đồng thời ứng dụng cơ chế giao tiếp liên tiến trình (Inter-Process Communication – IPC) thông qua **bộ nhớ dùng chung** (Shared Memory).

Tập dữ liệu đầu vào gồm 100.000 lượt đánh giá, được thực hiện bởi 943 người dùng cho 1.682 bộ phim khác nhau. Mỗi dòng dữ liệu có cấu trúc như sau:

`userID <tab> movieID <tab> rating <tab> timeStamp`

Ví dụ:

```
196     242     3    881250949
186     302     3    891717742
22      377     1    878887116
244     51      2    880606923
166     346     1    886397596
298     474     4    884182806
115     265     2    881171488
253     465     5    891628467
......
```

Để mô phỏng quá trình xử lý song song dữ liệu lớn, tập dữ liệu được chia thành hai tệp:

- `movie-100k_1.txt`
- `movie-100k_2.txt`

Chương trình sẽ tạo ra hai tiến trình con (child processes), mỗi tiến trình chịu trách nhiệm đọc một tệp dữ liệu riêng, tính toán tổng điểm (Total Ratings) và số lượt đánh giá (Count) cho từng bộ phim. Kết quả tính toán của mỗi tiến trình được ghi vào vùng bộ nhớ dùng chung (Shared Memory), nơi tiến trình cha (Parent Process) sẽ thu thập, tổng hợp và hiển thị kết quả cuối cùng ra màn hình.

1.2 Cách thức thực hiện

Bước 1: Khởi tạo vùng bộ nhớ dùng chung (Shared Memory Segment)

Tiến trình cha gọi hệ thống `shmget()` để khởi tạo một vùng bộ nhớ chia sẻ có kích thước đủ lớn nhằm lưu trữ thông tin của toàn bộ 1.682 bộ phim. Mỗi phần tử trong vùng nhớ này tương ứng với một bộ phim và bao gồm các trường dữ liệu:

- **movieID:** Mã định danh duy nhất của bộ phim.
- **totalRatings:** Tổng điểm đánh giá của bộ phim từ tất cả người dùng.
- **count:** Số lượt đánh giá mà bộ phim nhận được.
- **average:** Điểm trung bình của bộ phim (được tính sau khi hai tiến trình con hoàn thành công việc).

Bước 2: Tạo và thực thi hai tiến trình con

Tiến trình cha sử dụng hàm `fork()` để tạo ra hai tiến trình con song song:

- **Child 1:** Đọc và xử lý dữ liệu trong `movie-100k_1.txt`.
- **Child 2:** Đọc và xử lý dữ liệu trong `movie-100k_2.txt`.

Mỗi tiến trình con thực hiện các thao tác sau:

1. Mở tệp dữ liệu được chỉ định.
2. Đọc từng dòng dữ liệu và tách các giá trị `movieID` và `rating`.
3. Cộng dồn giá trị `rating` vào `totalRatings[movieID]`.
4. Tăng biến đếm `count[movieID]` thêm 1.
5. Ghi kết quả tạm thời vào vùng `Shared Memory` để tiến trình cha có thể truy cập.

Nhờ đó, hai tiến trình con có thể **xử lý dữ liệu song song**, giúp tăng hiệu quả và rút ngắn thời gian tính toán.

Bước 3: Tổng hợp và hiển thị kết quả

Sau khi cả hai tiến trình con hoàn tất (được xác nhận thông qua hàm `wait()`), tiến trình cha sẽ:

1. Truy cập vùng Shared Memory để đọc dữ liệu tổng hợp từ hai tiến trình con.
2. Tính điểm trung bình cho từng bộ phim theo công thức:

$$\text{average}[\text{movieID}] = \frac{\text{totalRatings}[\text{movieID}]}{\text{count}[\text{movieID}]}$$

3. Tính điểm trung bình toàn hệ thống (overall average):

$$\text{overallAverage} = \frac{\sum \text{totalRatings}}{\sum \text{count}}$$

4. Hiển thị kết quả cuối cùng ra màn hình dưới dạng bảng thống kê.

Bảng 1: ***Ý nghĩa các trường dữ liệu trong kết quả đầu ra***

Trường dữ liệu	Ý nghĩa
MovieID	Mã định danh duy nhất của bộ phim.
Total Ratings	Tổng số điểm đánh giá mà bộ phim nhận được từ tất cả người dùng.
Count	Số lượt đánh giá hợp lệ mà bộ phim nhận được.
Average	Điểm trung bình của bộ phim, được tính theo công thức: $$\text{Average} = \frac{\text{Total Ratings}}{\text{Count}}$$

1.3 Hình Ảnh Kết Quả Đầu Ra

```
======================================
MOVIE RATING ANALYZER WITH SHARED MEMORY
Lab 2 - Problem 1
======================================
[Parent - PID 680] Starting...
File 1: movie-100k_1.txt
File 2: movie-100k_2.txt
======================================

✓ Shared memory created (ID: 0, Size: 26916 bytes)
✓ Shared memory attached at: 0x718afdd95000

✓ Shared memory initialized

Creating child processes...
[Child 1 - PID 681] Started reading movie-100k_1.txt

[Parent] Waiting for child processes to complete...

[Child 2 - PID 682] Started reading movie-100k_2.txt
[Child 1] Progress: 10000 lines processed...
[Child 2] Progress: 10000 lines processed...
[Child 1] Progress: 20000 lines processed...
[Child 2] Progress: 20000 lines processed...
[Child 1] Progress: 30000 lines processed...
[Child 2] Progress: 30000 lines processed...
[Child 1] Progress: 40000 lines processed...
[Child 2] Progress: 40000 lines processed...
[Child 1] Progress: 50000 lines processed...
[Child 1 - PID 681] Completed: Read 50177 lines from movie-100k_1.txt
[Child 2 - PID 682] Completed: Read 49823 lines from movie-100k_2.txt
✓ Child 1 (PID 681) finished
✓ Child 2 (PID 682) finished
```

Hình 1: *Thông tin tiến trình và quá trình xử lý*

Ngay khi chương trình bắt đầu, tiến trình cha (`Parent`) khởi tạo một vùng bộ nhớ dùng chung (*Shared Memory*) và sau đó tạo ra hai tiến trình con (`Child 1` và `Child 2`). Ba tiến trình này cùng tham gia vào quá trình tính toán, nhưng mỗi tiến trình đảm nhận một vai trò khác nhau:

- **Parent - PID 680:** Là tiến trình cha. Nhiệm vụ chính là khởi tạo vùng *Shared Memory*, tạo hai tiến trình con, chờ cả hai hoàn tất công việc, rồi đọc dữ liệu đã tổng hợp để in bảng thống kê.
- **Child 1 - PID 681:** Đọc tệp dữ liệu `movie-100k_1.txt`. Khi gặp dòng có `movieID = 1`, tiến trình này sẽ tăng giá trị `sum_ratings` và `count` của phim 1 trong vùng nhớ dùng chung.
- **Child 2 - PID 682:** Đọc tệp dữ liệu `movie-100k_2.txt`. Nếu dòng có `movieID = 1`, tiến trình sẽ tiếp tục cộng dồn điểm và số lượt đánh giá vào đúng vị trí của phim 1 trong *Shared Memory*.

Cả hai tiến trình con đều truy cập cùng vùng nhớ chung. Để tránh *race condition*, chương trình sử dụng cơ chế khóa đơn giản thông qua hai hàm:

```
acquire_lock()
release_lock()
```

Khi đọc và xử lý dữ liệu, chương trình hiển thị thông báo tiến độ:

```
Progress: 10000 lines processed...
```

Khi hoàn tất đọc tệp:

```
Completed: Read 50177 lines
Completed: Read 49823 lines
```

Sau khi cả hai tiến trình con kết thúc (`Child 1 finished`, `Child 2 finished`), tiến trình cha tổng hợp dữ liệu từ vùng nhớ dùng chung.

========== MOVIE AVERAGE RATINGS ==========

MovieID	Total Ratings	Count	Average
1	1753	452	3.8783
2	420	131	3.2061
3	273	90	3.0333
4	742	209	3.5502
5	284	86	3.3023
6	93	26	3.5769
7	1489	392	3.7985
8	875	219	3.9954
9	1165	299	3.8963
10	341	89	3.8315
11	908	236	3.8475
12	1171	267	4.3858
13	629	184	3.4185
14	726	183	3.9672
15	1107	293	3.7782
16	125	39	3.2051
17	287	92	3.1196
18	28	10	2.8000
19	273	69	3.9565
20	246	72	3.4167
21	232	84	2.7619
22	1233	297	4.1515
23	750	182	4.1209
24	600	174	3.4483
25	1009	293	3.4437
26	252	73	3.4521
27	177	57	3.1053
28	1085	276	3.9312
29	304	114	2.6667
30	146	37	3.9459
31	559	154	3.6299
32	307	81	3.7901
33	335	97	3.4536
34	19	7	2.7143
35	24	11	2.1818

36	28	13	2.1538
37	18	8	2.2500
38	361	120	3.0083
39	284	87	3.2644
40	165	57	2.8947
41	114	37	3.0811
42	563	148	3.8041
43	120	40	3.0000
44	264	79	3.3418
45	324	80	4.0500
46	96	27	3.5556
47	479	133	3.6015
48	479	117	4.0940
49	269	81	3.3210
50	2541	583	4.3585
51	280	81	3.4568
52	343	91	3.7692
53	378	128	2.9531
54	337	104	3.2404
55	552	149	3.7047
56	1600	394	4.0609
57	160	40	4.0000
58	638	175	3.6457
59	337	83	4.0602
60	257	64	4.0156
61	228	59	3.8644
62	399	127	3.1417
63	254	82	3.0976
64	1258	283	4.4452
65	407	115	3.5391
66	575	162	3.5494
67	314	103	3.0485
68	458	134	3.4179
69	1237	321	3.8536
70	919	251	3.6614
71	832	220	3.7818
72	412	129	3.1938
73	444	128	3.4688

74	21	7	3.0000
75	15	5	3.0000
76	184	54	3.4074
77	495	151	3.2781
78	81	33	2.4545
79	1359	336	4.0446
80	188	68	2.7647
81	391	110	3.5545
82	971	261	3.7203
83	715	176	4.0625
84	54	18	3.0000
85	176	58	3.0345
86	591	150	3.9400
87	539	138	3.9058
88	754	213	3.5399
89	1138	275	4.1382
90	290	95	3.0526
91	513	143	3.5874
92	376	104	3.6154
93	417	112	3.7232
94	423	137	3.0876
95	835	219	3.8128
96	1182	295	4.0068
97	971	256	3.7930
98	1673	390	4.2897
99	638	172	3.7093
100	2111	508	4.1555
101	238	73	3.2603
102	169	54	3.1296
103	28	15	1.8667
104	7	5	1.4000
105	190	74	2.5676
106	205	71	2.8873
107	147	42	3.5000
108	200	65	3.0769
109	446	130	3.4308
110	78	31	2.5161
111	948	272	3.4853

112	49	20	2.4500
113	37	9	4.1111
114	298	67	4.4478
115	56	15	3.7333
116	478	125	3.8240
117	1396	378	3.6931
118	942	293	3.2150
119	18	4	4.5000
120	150	67	2.2388
121	1475	429	3.4382
122	248	106	2.3396
123	372	115	3.2348
124	758	187	4.0535
125	868	244	3.5574
126	369	97	3.8041
127	1769	413	4.2833
128	230	65	3.5385
129	493	129	3.8217
130	52	23	2.2609
131	354	95	3.7263
132	1003	246	4.0772
133	662	171	3.8713
134	850	198	4.2929
135	1028	259	3.9691
136	433	105	4.1238
137	667	171	3.9006
138	49	19	2.5789
139	139	50	2.7800
140	196	61	3.2131
141	252	72	3.5000
142	175	57	3.0702
143	836	222	3.7658
144	941	243	3.8724
145	159	65	2.4462
146	25	10	2.5000
147	642	185	3.4703
148	410	128	3.2031
149	69	23	3.0000

150	601	157	3.8280
151	1184	326	3.6319
152	298	82	3.6341
153	935	247	3.7854
154	652	174	3.7471
155	304	98	3.1020
156	590	148	3.9865
157	469	127	3.6929
158	157	60	2.6167
159	326	101	3.2277
160	239	69	3.4638
161	766	220	3.4818
162	377	106	3.5566
163	327	92	3.5543
164	542	151	3.5894
165	263	64	4.1094
166	239	58	4.1207
167	198	67	2.9552
168	1285	316	4.0665
169	527	118	4.4661
170	505	121	4.1736
171	252	65	3.8769
172	1543	367	4.2044
173	1352	324	4.1728
174	1786	420	4.2524
175	794	208	3.8173
176	1121	284	3.9472
177	529	137	3.8613
178	543	125	4.3440
179	864	221	3.9095
180	894	221	4.0452
181	2032	507	4.0079
182	893	226	3.9513
183	1174	291	4.0344
184	398	116	3.4310
185	980	239	4.1004
186	963	251	3.8367
187	875	209	4.1866

. . . . . .

```
1650        4                   1           4.0000
1651        4                   1           4.0000
1652        5                   3           1.6667
1653        5                   1           5.0000
1654        1                   1           1.0000
1655        2                   1           2.0000
1656        7                   2           3.5000
1657        3                   1           3.0000
1658        9                   3           3.0000
1659        1                   1           1.0000
1660        2                   1           2.0000
1661        1                   1           1.0000
1662        5                   2           2.5000
1663        2                   1           2.0000
1664        13                  4           3.2500
1665        2                   1           2.0000
1666        2                   1           2.0000
1667        3                   1           3.0000
1668        3                   1           3.0000
1669        2                   1           2.0000
1670        3                   1           3.0000
1671        1                   1           1.0000
1672        4                   2           2.0000
1673        3                   1           3.0000
1674        4                   1           4.0000
1675        3                   1           3.0000
1676        2                   1           2.0000
1677        3                   1           3.0000
1678        1                   1           1.0000
1679        3                   1           3.0000
1680        2                   1           2.0000
1681        3                   1           3.0000
1682        3                   1           3.0000
================================================================
Total movies with ratings: 1682 / 1682
Total ratings processed: 100000
Overall average rating: 3.5299
================================================================
```

Hình 2: *Bảng "MOVIE AVERAGE RATINGS*

Phân tích bảng kết quả "MOVIE AVERAGE RATINGS"

Bảng kết quả hiển thị danh sách các bộ phim kèm theo thông tin về điểm đánh giá của người xem. Đây là kết quả cuối cùng mà chương trình in ra sau khi hai tiến trình con đã hoàn tất việc xử lý **100.000 dòng dữ liệu** và tiến trình cha đã tổng hợp dữ liệu từ vùng nhớ dùng chung (*Shared Memory*).

Bảng gồm bốn cột chính, mỗi cột mang một ý nghĩa cụ thể như sau:

Cột	Tên đại lượng	Giải thích ý nghĩa
MovieID	Mã định danh phim	Là số thứ tự duy nhất đại diện cho từng bộ phim trong hệ thống (từ 1 đến 1682). Mỗi bộ phim chỉ có một mã ID duy nhất.
Total Ratings	Tổng điểm đánh giá	Là tổng cộng tất cả các điểm (rating) mà người dùng đã chấm cho bộ phim đó, được tổng hợp từ hai tệp dữ liệu `movie-100k_1.txt` và `movie-100k_2.txt`.
Count	Số lượt đánh giá	Biểu thị số lần (hoặc số người dùng) đã chấm điểm cho bộ phim đó.
Average	Điểm trung bình	Được tính theo công thức: $$\text{Average} = \frac{\text{Total Ratings}}{\text{Count}}$$ Đây là mức điểm trung bình phản ánh mức độ yêu thích của khán giả đối với bộ phim.

Bảng 2: *Ý nghĩa các cột trong bảng kết quả Movie Average Ratings*

Ví dụ minh họa - Phim số 1

Dòng đầu tiên trong bảng kết quả thể hiện thông tin về bộ phim có **MovieID = 1** như sau:

```
1    1753    452    3.8783
```

Phân tích chi tiết từng đại lượng:

- **MovieID:**

1 – là mã định danh của bộ phim đầu tiên trong danh sách gồm 1 682 phim. Mỗi bộ phim chỉ có một mã số duy nhất dùng để nhận dạng trong toàn bộ hệ thống dữ liệu.

- **Total Ratings:**

1753 – là tổng điểm mà tất cả người dùng đã chấm cho bộ phim này. Ví dụ: nếu có các lượt đánh giá 4, 5, 3, . . . thì chương trình sẽ cộng dồn tất cả lại để thu được giá trị tổng là 1753.

- **Count:**

452 – là số lượt đánh giá, tương ứng với 452 người dùng khác nhau đã chấm điểm cho bộ phim số 1.

- **Average:**

3.8783 – là điểm trung bình của phim, được tính theo công thức:

$$\text{Average} = \frac{\text{Total Ratings}}{\text{Count}} = \frac{1753}{452} = 3.8783$$

Giá trị này phản ánh mức độ yêu thích trung bình của khán giả đối với bộ phim.

Nhận xét:

Điểm trung bình **3.8783/5** cho thấy phim số 1 được đánh giá khá cao và nằm trong nhóm những bộ phim được yêu thích nhất trong tập dữ liệu. Nói cách khác, nếu cộng tất cả 452 lượt đánh giá lại, trung bình mỗi người dùng chấm khoảng **3.88 trên thang điểm 5**, thể hiện mức độ hài lòng tích cực của người xem.

Ví dụ minh họa - Phim số 2

Dòng thứ hai trong bảng kết quả thể hiện thông tin về bộ phim có **MovieID = 2** như sau:

```
2    420    131    3.2061
```

Phân tích chi tiết từng đại lượng:

- **MovieID:**

2 – là mã định danh duy nhất của bộ phim thứ hai trong danh sách gồm 1 682 phim. Giá trị này giúp hệ thống xác định chính xác phim cần tính toán và tổng hợp dữ liệu đánh giá.

- **Total Ratings:**

420 – là tổng điểm mà tất cả người dùng đã chấm cho bộ phim này. Ví dụ: nếu có người đánh giá 4, 3, 5, . . . thì tổng cộng các giá trị đó lại được 420 điểm.

- **Count:**

131 – là tổng số lượt đánh giá (tức là có 131 người dùng khác nhau) đã chấm điểm cho bộ phim số 2. Số lượng này thấp hơn so với phim 1, cho thấy mức độ phổ biến của phim 2 có thể ít hơn.

- **Average:**

3.2061 – là điểm trung bình của phim, được tính theo công thức:

$$\text{Average} = \frac{\text{Total Ratings}}{\text{Count}} = \frac{420}{131} \approx 3.2061$$

Điểm trung bình này cho thấy bộ phim được đánh giá ở mức **trung bình khá**, không quá cao nhưng vẫn trên ngưỡng 3/5.

2 PROBLEM 2

2.1 Mô tả bài toán

Bài toán yêu cầu lập trình một chương trình tính tổng của dãy số tự nhiên liên tiếp từ 1 đến n, được biểu diễn dưới dạng:

$$\text{sum}(n) = 1 + 2 + 3 + \cdots + n$$

Đây là một bài toán cơ bản trong lập trình, tuy nhiên khi n rất lớn (ví dụ 10^6, 10^8 hoặc hơn), việc tính toán tuần tự sẽ tiêu tốn nhiều thời gian xử lý CPU. Do đó, mục tiêu đặt ra là xây dựng hai phiên bản chương trình:

- **Phiên bản tuần tự (Serial Version):** chỉ sử dụng một luồng (*thread*) để thực hiện phép cộng lần lượt từ 1 đến n.
- **Phiên bản song song (Multi-thread Version):** chia bài toán thành nhiều phần nhỏ và tính toán đồng thời trên nhiều luồng nhằm rút ngắn thời gian xử lý.

Sau khi hoàn thành, hai phiên bản sẽ được so sánh hiệu năng (*speed-up*) để minh họa lợi ích của lập trình song song.

Về mặt toán học, giá trị tổng có thể được tính chính xác bằng công thức Gauss:

$$\text{sum}(n) = \frac{n(n+1)}{2}$$

Công thức này được dùng để kiểm chứng tính đúng đắn của kết quả mà hai phiên bản chương trình tính toán được.

2.2 Cách thức thực hiện

2.2.1 Phương pháp tiếp cận

(a) Phiên bản tuần tự (Serial Version).

Phiên bản này thực hiện một vòng lặp duy nhất, cộng dồn từng giá trị từ 1 đến n để thu được tổng cuối cùng. Toàn bộ quá trình chỉ chạy trên một luồng, do đó mọi phép cộng được thực hiện tuần tự.

Để đo thời gian thực thi, chương trình sử dụng hàm `clock()` trong thư viện `<time.h>`. Sau khi hoàn tất, kết quả được so sánh với giá trị lý thuyết (theo công thức Gauss) để xác minh tính chính xác.

Cấu trúc chương trình gồm ba phần chính:

1. Kiểm tra và đọc tham số đầu vào (giá trị n);
2. Thực hiện phép cộng tuần tự;
3. Đo và hiển thị thời gian chạy cùng kết quả kiểm chứng.

(b) Phiên bản đa luồng (Multi-thread Version).

Phiên bản này áp dụng mô hình **chia miền dữ liệu (data partitioning)** để khai thác khả năng xử lý song song của CPU đa nhân.

Toàn bộ tập dữ liệu $[1, n]$ được chia thành nhiều đoạn con gần bằng nhau, tùy theo số lượng luồng `numThreads`. Mỗi luồng sẽ chịu trách nhiệm tính tổng trên một đoạn con cụ thể như sau:

$$\begin{aligned}
&\text{Thread 1: } 1 \to \frac{n}{\text{numThreads}} \\
&\text{Thread 2: } \frac{n}{\text{numThreads}} + 1 \to \frac{2n}{\text{numThreads}} \\
&\qquad\vdots \\
&\text{Thread k: } (k-1) \cdot \frac{n}{\text{numThreads}} + 1 \to n
\end{aligned}$$

Mỗi luồng lưu kết quả riêng của mình vào biến `partial_sum`, nằm trong một cấu trúc dữ liệu `ThreadData`. Sau khi tất cả luồng hoàn tất (được đồng bộ bằng hàm `pthread_join()`), chương trình sẽ cộng gộp tất cả các giá trị `partial_sum` lại để thu được tổng cuối cùng `total_sum`.

Thư viện `<pthread.h>` được sử dụng cho các thao tác tạo, thực thi và đồng bộ luồng, trong khi `<time.h>` tiếp tục được sử dụng để đo thời gian xử lý.

Quá trình thực thi của phiên bản đa luồng bao gồm các bước:

1. **Nhập tham số đầu vào**: chương trình nhận vào số lượng luồng `numThreads` và giá trị `n`.
2. **Chia miền dữ liệu**: toàn bộ dãy số $[1, n]$ được chia thành các đoạn con gần bằng nhau, mỗi đoạn có kích thước
$$\text{chunk\_size} = \frac{n}{\text{numThreads}}$$
để mỗi luồng xử lý một phần riêng.
3. **Tạo và khởi chạy các luồng**: sử dụng hàm `pthread_create()` để tạo luồng và gán cho mỗi luồng phạm vi dữ liệu cần tính toán.
4. **Tính toán song song**: mỗi luồng thực hiện hàm `calculate_partial_sum()` để tính tổng của đoạn dữ liệu được phân công, lưu kết quả vào biến `partial_sum` riêng trong cấu trúc `ThreadData`.
5. **Đồng bộ hóa luồng**: sử dụng `pthread_join()` để chờ tất cả luồng hoàn tất, đảm bảo kết quả được thu thập đầy đủ trước khi tiếp tục.
6. **Tổng hợp kết quả**: cộng gộp tất cả các giá trị `partial_sum` từ từng luồng để thu được tổng cuối cùng `total_sum`.
7. **Kiểm chứng và đo thời gian**: so sánh kết quả cuối cùng với công thức Gauss và hiện thời gian.

2.2.2 Đánh giá và quan sát

Khi thực hiện chương trình với cùng một giá trị n (ví dụ $n = 1{,}000{,}000$):

- Phiên bản tuần tự chỉ sử dụng một luồng, do đó thời gian thực thi tăng tuyến tính theo kích thước n.
- Phiên bản đa luồng chia tải công việc cho nhiều luồng, nhờ đó tận dụng được nhiều nhân xử lý của CPU, dẫn đến giảm đáng kể thời gian tính toán.

Hiệu quả tăng tốc (*Speed-up*) được định nghĩa bởi công thức:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}}$$

Trong đó:

- T_{serial}: thời gian thực thi của phiên bản tuần tự;
- T_{parallel}: thời gian thực thi của phiên bản đa luồng.

Kết quả thực nghiệm cho thấy, đối với các giá trị n lớn, phiên bản đa luồng giúp giảm đáng kể thời gian chạy, minh chứng cho hiệu quả của lập trình song song trong việc tối ưu hiệu năng chương trình.

2.3 Hình Ảnh Kết Quả Đầu Ra

(a) Phiên bản tuần tự (Sum Serial).

Sau khi chạy chương trình **phiên bản tuần tự (Serial Version)** với đầu vào $n = 500{,}000$, kết quả hiển thị như sau:

```
vopham05@DESKTOP-2HD73M6:~/LAB2/lab2_problem2$ ./sum_serial 500000

      1. SERIAL SUM CALCULATOR

Calculating sum(1..500000)

========================================
               RESULTS
========================================
Result:              125000250000
Time taken:          0.000103 seconds
Expected (formula):  125000250000
Verification:        ✓ CORRECT
========================================
```

Hình 3: *Serial Sum Calculator ($n = 500{,}000$)*

Một số nhận xét chi tiết:

- **Result: 125000250000** - Đây là tổng được tính bằng phương pháp cộng tuần tự các số từ 1 đến 500,000. Kết quả hoàn toàn chính xác và hợp lý.
- **Expected (formula): 125000250000** - Giá trị này được tính theo công thức Gauss:

$$\text{sum}(n) = \frac{n(n+1)}{2} \Rightarrow \text{sum}(500{,}000) = 125{,}000{,}250{,}000$$

  Việc trùng khớp giữa giá trị tính toán và lý thuyết chứng minh chương trình hoạt động đúng.
- **Time taken: 0.000103 seconds** - Thời gian thực thi được đo bằng hàm `clock()`. Với $n = 500{,}000$, thời gian vẫn rất nhỏ, chỉ khoảng 0.1 milli-giây. Tuy nhiên, nếu n lớn hơn nhiều, ví dụ 10^7 hoặc 10^8, thời gian sẽ tăng tuyến tính.
- **Verification: ✓CORRECT** - Kết quả tính toán được xác nhận là chính xác, không xảy ra sai sót hay lỗi logic.

Tổng thể, chương trình **Serial Sum Calculator** với $n = 500{,}000$ hoạt động chính xác và hiệu quả. Tuy nhiên, đối với các giá trị n lớn hơn, thời gian tính toán sẽ tăng theo cấp số tuyến tính, đây là lý do để so sánh với phiên bản **đa luồng (Multi-thread Version)** nhằm đánh giá lợi ích của lập trình song song.

(b) Phiên bản đa luồng (Sum Multi Thread).

Sau khi chạy chương trình **phiên bản đa luồng** với $n = 8{,}000{,}000$ và 10 luồng, kết quả hiển thị:

```
vopham05@DESKTOP-2HD73M6:~/LAB2/lab2_problem2$ ./sum_multi-thread 10 8000000
      2. MULTI-THREAD SUM CALCULATOR

Number of threads: 10
Calculating sum(1..8000000)

Creating 10 threads...
Waiting for all threads to complete...

========================================
             THREAD DETAILS
========================================
Thread   Range Start          Range End            Partial Sum
----------------------------------------
1        1                    800000               320000400000
2        800001               1600000              960000400000
3        1600001              2400000              1600000400000
4        2400001              3200000              2240000400000
5        3200001              4000000              2880000400000
6        4000001              4800000              3520000400000
7        4800001              5600000              4160000400000
8        5600001              6400000              4800000400000
9        6400001              7200000              5440000400000
10       7200001              8000000              6080000400000

========================================
             FINAL RESULTS
========================================
Total sum:          32000004000000
Time taken:         0.005356 seconds
Expected (formula): 32000004000000
Verification:       ✓ CORRECT
========================================
```

Hình 4: *Multi thread Sum Calculator (n = 8,000,000, numThreads = 10)*

Một số nhận xét chi tiết:

- **Thread Details** - Bảng hiển thị phạm vi số và tổng từng đoạn mà mỗi luồng xử lý. Mỗi luồng chịu trách nhiệm tính tổng của khoảng 800,000 số liên tiếp. Việc chia đều công việc giúp tận dụng hiệu quả khả năng xử lý song song của CPU.
- **Partial Sum** - Mỗi luồng lưu kết quả tính toán riêng. Khi cộng tất cả các `partial_sum` từ 10 luồng, thu được `Total sum` chính xác.
- **Total sum: 32,000,004,000,000** - Tổng cuối cùng khớp hoàn toàn với giá trị lý thuyết, chứng minh tính chính xác của thuật toán đa luồng.
- **Expected (formula): 32,000,004,000,000** - Giá trị tính theo công thức Gauss:

$$\text{sum}(n) = \frac{n(n+1)}{2} \Rightarrow \text{sum}(8{,}000{,}000) = 32{,}000{,}004{,}000{,}000$$

- **Time taken: 0.005356 seconds** - Thời gian thực thi được đo bằng `clock()`. So với phiên bản tuần tự, phiên bản đa luồng xử lý nhanh hơn đáng kể nhờ việc chia tải cho nhiều luồng.
- **Verification: ✓CORRECT** - Kết quả cuối cùng được xác nhận chính xác, hoàn toàn phù hợp với giá trị lý thuyết.

Nhìn chung, phiên bản đa luồng cho thấy lợi ích rõ rệt của lập trình song song: giảm đáng kể thời gian thực thi, đặc biệt với các giá trị n lớn, đồng thời kết quả vẫn đảm bảo chính xác.

(c) So sánh hiệu năng - Serial vs Multi-thread Sum Calculator

Thử nghiệm chương trình với $n = 100{,}000{,}000$ và 10 luồng cho kết quả như sau:

```
vopham05@DESKTOP-2HD73M6:~/LAB2/lab2_problem2$ ./sum_serial 100000000
./sum_multi-thread 10 100000000

      1. SERIAL SUM CALCULATOR

Calculating sum(1..100000000)

========================================
               RESULTS
========================================
Result:             5000000050000000
Time taken:         0.020298 seconds
Expected (formula): 5000000050000000
Verification:       ✓ CORRECT
========================================


      2. MULTI-THREAD SUM CALCULATOR

Number of threads: 10
Calculating sum(1..100000000)

Creating 10 threads...
Waiting for all threads to complete...

========================================
             THREAD DETAILS
========================================
Thread   Range Start          Range End            Partial Sum
----------------------------------------
1        1                    10000000             50000005000000
2        10000001             20000000             150000005000000
3        20000001             30000000             250000005000000
4        30000001             40000000             350000005000000
5        40000001             50000000             450000005000000
6        50000001             60000000             550000005000000
7        60000001             70000000             650000005000000
8        70000001             80000000             750000005000000
9        80000001             90000000             850000005000000
10       90000001             100000000            950000005000000

========================================
             FINAL RESULTS
========================================
Total sum:          5000000050000000
Time taken:         0.004505 seconds
Expected (formula): 5000000050000000
Verification:       ✓ CORRECT
========================================
```

Hình 5: *Serial và Multi-thread Sum Calculator (n = 100, numThreads = 10)*

Một số nhận xét quan trọng:

- **Độ chính xác:**
  - Cả hai phiên bản Serial và Multi-thread đều tính đúng tổng của dãy số từ 1 đến 100,000,000.
  - Kết quả cuối cùng `Total sum = 5,000,000,050,000,000` trùng khớp với giá trị lý thuyết

theo công thức Gauss:

$$\text{sum}(n) = \frac{n(n+1)}{2} = 5{,}000{,}000{,}050{,}000{,}000$$

- Xác minh thành công: ✓CORRECT.

- **Hiệu năng:**
  - **Serial Version:** thời gian thực thi $T_{\text{serial}} = 0.020298$ giây.
  - **Multi-thread Version:** thời gian thực thi $T_{\text{parallel}} = 0.004505$ giây.
  - Hiệu quả tăng tốc (Speed-up) được tính bằng:

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \frac{0.020298}{0.004505} \approx 4.5$$

  - Như vậy, Multi-thread nhanh hơn Serial khoảng 4.5 lần, nhờ tận dụng 10 luồng xử lý song song.

- **Nguyên nhân khác biệt:**
  - Phiên bản Multi-thread chia dãy số thành nhiều đoạn con, mỗi luồng xử lý song song trên một đoạn riêng biệt, nhờ đó tận dụng khả năng đa nhân của CPU.
  - Phiên bản Serial thực hiện tính toán tuần tự trên một luồng duy nhất, không tận dụng được song song.
  - Khi n lớn, chi phí tạo và đồng bộ luồng trở nên rất nhỏ so với tổng thời gian, dẫn đến cải thiện đáng kể tốc độ xử lý.

- **Nhận xét chung:**
  - Phiên bản Multi-thread tỏ ra hiệu quả hơn rõ rệt khi xử lý dữ liệu lớn, đồng thời vẫn đảm bảo kết quả chính xác.
  - Phiên bản Serial phù hợp với dữ liệu nhỏ, vì khi đó overhead của Multi-thread có thể làm thời gian chạy dài hơn.
  - Tăng số lượng luồng giúp speed-up tiến gần tới số nhân vật lý của CPU, nhưng luôn bị giới hạn bởi chi phí tạo và đồng bộ luồng.

3 PROBLEM 3

3.1 Mô tả bài toán tổng quan

3.1.1 Giới thiệu

Bài toán yêu cầu xây dựng một hệ thống giao tiếp hai chiều giữa hai tiến trình (*Process A* và *Process B*) trên hệ điều hành Linux, sử dụng *System V Message Queue* làm kênh trao đổi thông điệp. Hệ thống mô phỏng một ứng dụng nhắn tin cơ bản, cho phép người dùng nhập và nhận tin nhắn theo thời gian thực giữa hai tiến trình độc lập.

Hệ thống được thiết kế với các yêu cầu sau:

- **Giao tiếp song song, đa luồng**: mỗi tiến trình có hai *thread* độc lập - một *thread gửi* (*send*) và một *thread nhận* (*receive*) - để người dùng có thể gửi và nhận tin nhắn đồng thời mà không bị chặn.
- **Quản lý tài nguyên an toàn**: mỗi tiến trình chịu trách nhiệm tạo và xóa *message queue* mà nó quản lý, đảm bảo tránh rò rỉ tài nguyên hệ thống.
- **Xử lý *signal* và kết thúc an toàn**: hệ thống có cơ chế nhận *signal* từ OS (`SIGINT`, `SIGTERM`) để dừng *threads* và dọn dẹp queue một cách có kiểm soát.
- **Cơ chế đồng bộ khởi động**: tiến trình B phải đợi tiến trình A tạo queue trước khi bắt đầu gửi/nhận, tránh lỗi khi queue chưa tồn tại.

3.1.2 Kiến trúc tổng thể

Hệ thống bao gồm ba thành phần chính:

1. **Hai tiến trình độc lập (Process A và Process B):**
   - Mỗi tiến trình chạy trong không gian địa chỉ riêng biệt.
   - Giao tiếp thông qua các *message queues* do kernel quản lý.
2. **Hai hàng đợi thông điệp (Message Queues):**
   - `QUEUE_A_TO_B`: Truyền dữ liệu từ Process A sang Process B.
   - `QUEUE_B_TO_A`: Truyền dữ liệu từ Process B sang Process A.
   - Việc tách riêng hai queue giúp đảm bảo tính toàn vẹn và hướng truyền **độc lập cho mỗi chiều dữ liệu**.
3. **Hai luồng xử lý trong mỗi tiến trình:**
   - **Sender Thread**: Đọc dữ liệu từ bàn phím, đóng gói vào cấu trúc thông điệp, và gửi đi qua queue.
   - **Receiver Thread**: Liên tục lắng nghe queue nhận và hiển thị nội dung ra màn hình.

Nhờ cấu trúc này, mỗi tiến trình có thể gửi và nhận **song song**, hình thành kênh truyền hai chiều hoạt động đồng bộ, đồng thời vẫn đảm bảo tách biệt luồng dữ liệu và tránh xung đột.

3.2 Cơ chế hoạt động của hệ thống

Hệ thống được thiết kế dựa trên nguyên lý **giao tiếp liên tiến trình không đồng bộ (Asynchronous IPC)** thông qua các **hàng đợi thông điệp** do kernel quản lý. Cơ chế hoạt động được tổ chức theo bốn giai đoạn chính:

1. **Khởi tạo hệ thống**
   (a) **Process A:** Khởi động đầu tiên và tạo hai hàng đợi thông điệp:
      - `QUEUE_A_TO_B`: truyền dữ liệu từ A sang B.
      - `QUEUE_B_TO_A`: truyền dữ liệu từ B sang A.
      - Cả hai được tạo bằng hàm `msgget()` với cờ `IPC_CREAT`.

   (b) **Process B:** Khởi động sau, kiểm tra (polling) sự tồn tại của `QUEUE_A_TO_B` tối đa 30 giây. Khi hàng đợi sẵn sàng:
      - Kết nối vào hệ thống.
      - Tạo hoặc mở lại `QUEUE_B_TO_A` để đảm bảo kênh gửi dữ liệu.

2. **Truyền thông điệp**
   (a) Người dùng nhập dữ liệu tại Process A hoặc Process B.
   (b) **Luồng gửi (`thread_send`):** đóng gói dữ liệu vào cấu trúc thông điệp:

```
struct message {
    long mtype;
    char text[256];
    char sender[50];
};
```

      - `mtype`: định danh loại thông điệp.
      - `sender`: xác định nguồn phát.

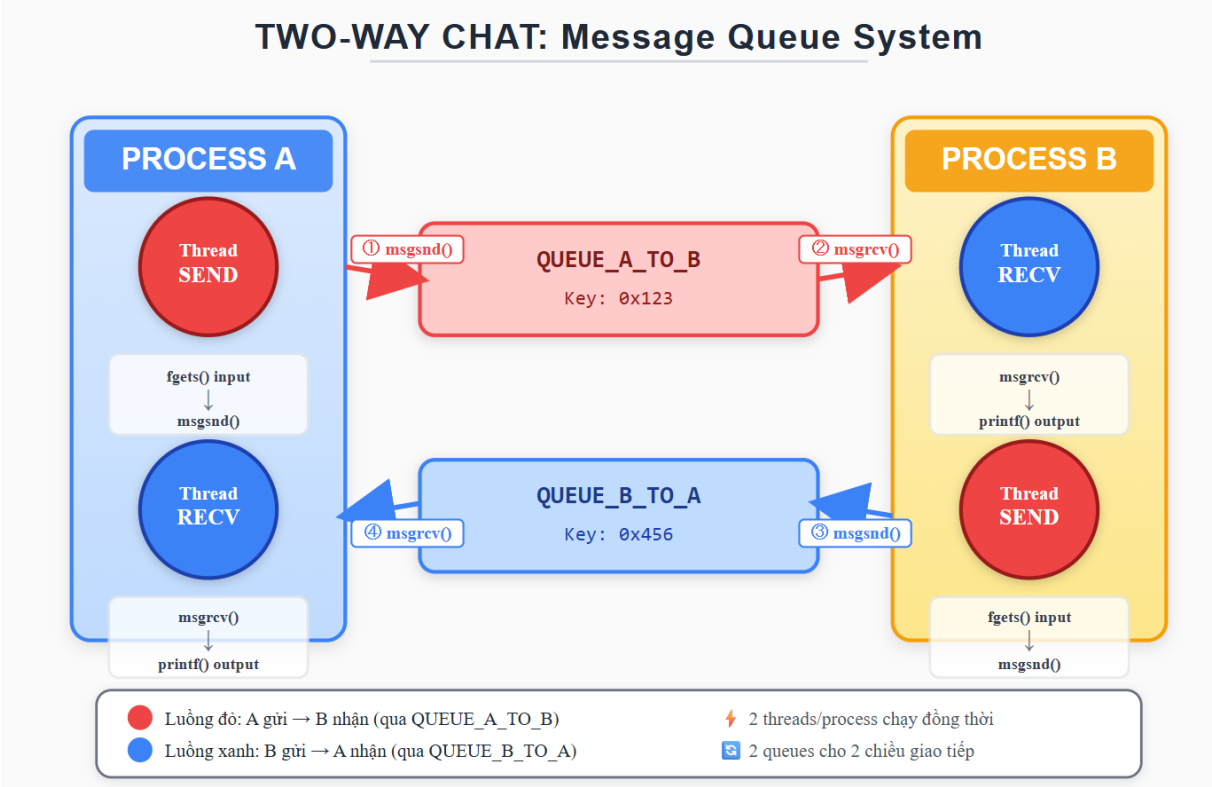
   (c) Thông điệp được gửi qua `msgsnd()` và lưu trong kernel cho đến khi tiến trình đích đọc.

3. **Nhận và hiển thị thông điệp**
   (a) **Luồng nhận (`thread_recv`):** liên tục gọi `msgrcv()` để nhận thông điệp từ hàng đợi.
   (b) Nội dung thông điệp được hiển thị ra màn hình kèm tên người gửi.
   (c) Nếu thông điệp chứa từ khóa `"quit"`, hệ thống hiểu là yêu cầu kết thúc phiên và kích hoạt quy trình dọn dẹp tài nguyên.

4. **Kết thúc và dọn dẹp (Cleanup)**
   (a) Khi người dùng thoát hoặc nhấn Ctrl+C, hàm `cleanup()` được gọi.
   (b) **Xóa hàng đợi:**
      - Process A xóa `QUEUE_A_TO_B`.
      - Process B chỉ xóa `QUEUE_B_TO_A`.

   (c) Các luồng được hủy an toàn, mutex giải phóng, và toàn bộ hàng đợi trong kernel được dọn sạch.

Lưu ý: Mỗi tiến trình chịu trách nhiệm duy nhất với hàng đợi của mình, giúp tránh **race condition** và lỗi truy cập khi một tiến trình kết thúc trước tiến trình còn lại.



Hình 6: *Message Queue System*

Process A - Khởi tạo giao tiếp

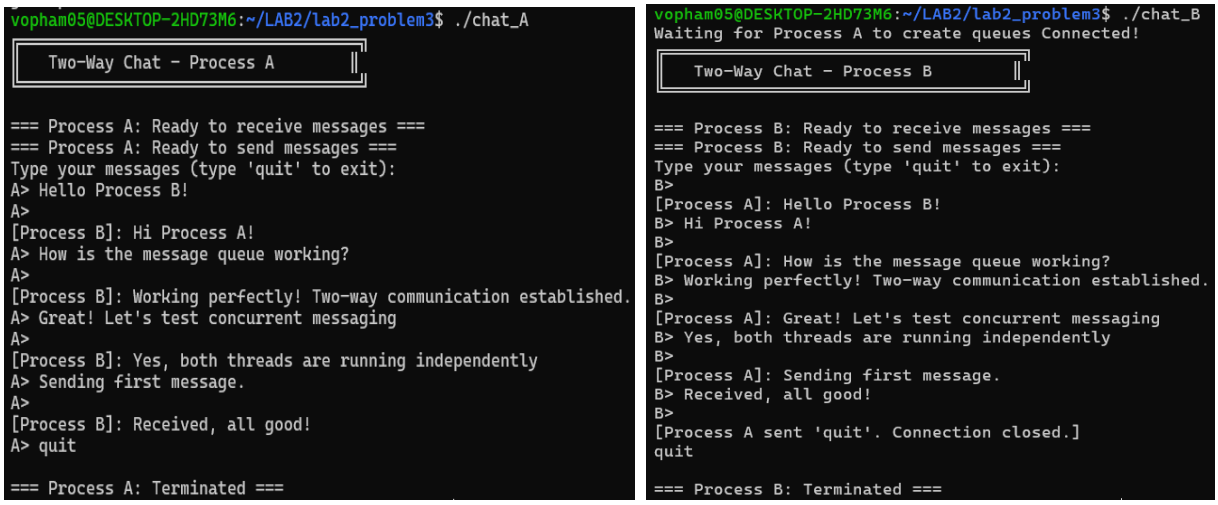
- **SEND (Gửi đi)**
  - Thu thập dữ liệu từ bàn phím và đóng gói thông điệp gồm: loại, người gửi và nội dung.
  - Gửi thông điệp vào hàng đợi `QUEUE_A_TO_B`.
  - **Vai trò:** Khởi tạo giao tiếp, chủ động truyền dữ liệu đến Process B.
- **RECV (Nhận về)**
  - Lắng nghe hàng đợi `QUEUE_B_TO_A` để nhận phản hồi từ Process B.
  - Hiển thị thông điệp ngay khi nhận, cho phép A vừa gửi vừa nhận song song.
  - **Đặc điểm:** Giao tiếp không chặn, đảm bảo tính liên tục.

Process B - Phản hồi

- **RECV (Nhận về)**
  - Lắng nghe hàng đợi `QUEUE_A_TO_B` từ Process A.
  - Hiển thị nội dung và thông tin người gửi, chuẩn bị phản hồi tiếp theo.
  - **Vai trò:** B luôn sẵn sàng nhận và xử lý dữ liệu từ A.
- **SEND (Gửi đi)**
  - Nhập dữ liệu phản hồi và đóng gói thành thông điệp.
  - Gửi trở lại Process A qua `QUEUE_B_TO_A`.
  - **Kết quả:** Hoàn tất chu kỳ trao đổi hai chiều, duy trì tính liên tục và đối xứng.

3.3 Hình Ảnh Kết Quả Đầu Ra

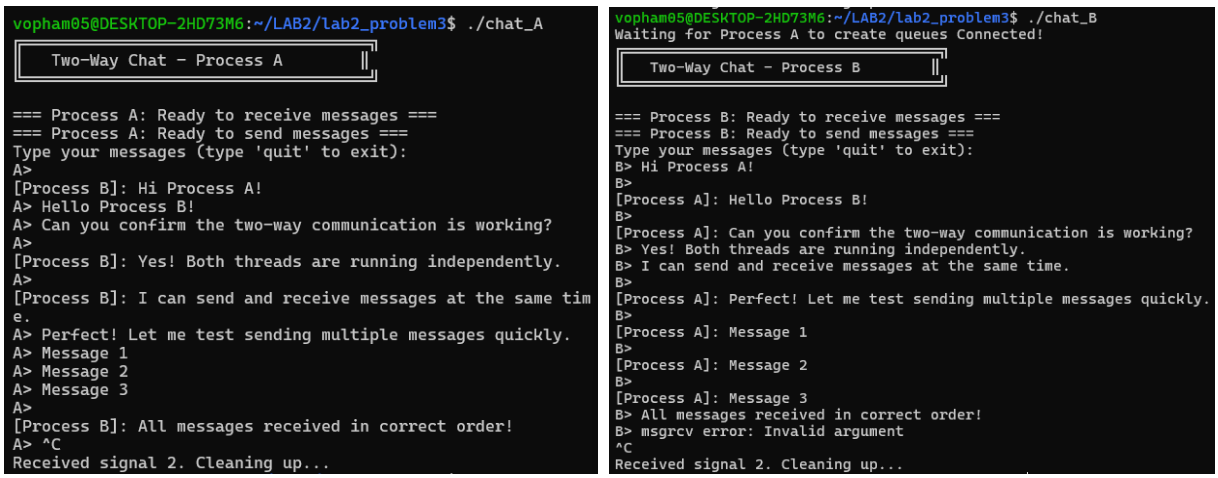
3.2.1. Testcase 1:



Hình 7: *Implementation Results*

- **Khởi chạy Process A**
  - Process A được kích hoạt và hiển thị tiêu đề: *"Two-Way Chat – Process A"*.
  - Hai luồng **SEND** và **RECV** được tạo, sẵn sàng để gửi và nhận tin nhắn.
  - Người dùng có thể gửi tin nhắn hoặc thoát bằng lệnh `quit`.
- **Khởi chạy Process B**
  - Process B khởi động sau Process A, chờ kết nối với *message queue* do A tạo.
  - Khi kết nối thành công, Process B hiển thị tiêu đề *"Two-Way Chat – Process B"* và trạng thái các luồng **SEND/RECV**.
- **Trao đổi tin nhắn**
  - Hai tiến trình gửi và nhận tin nhắn song song, đảm bảo giao tiếp không bị chặn.
  - Hàng đợi tin nhắn hoạt động ổn định, các luồng **SEND/RECV** chạy độc lập, cho phép cả hai tiến trình trao đổi liên tục mà không trễ.
- **Kết thúc phiên chat**
  - Khi Process A gửi lệnh kết thúc, Process B nhận thông báo đóng kết nối.
  - Cả hai tiến trình hiển thị thông báo *"Terminated"* và dọn dẹp tài nguyên an toàn.

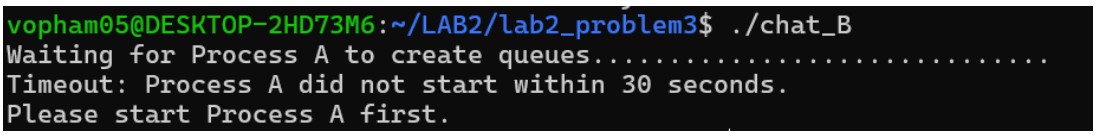
3.2.2. Testcase 2:



Hình 8: *SIGNAL HANDLING - Ctrl+C Cleanup*

- **Process A khởi chạy**
  - Hiển thị tiêu đề chương trình *"Two-Way Chat – Process A"* và thông báo các luồng **SEND** và **RECV** đã sẵn sàng.
  - Người dùng có thể gửi tin nhắn hoặc thoát phiên chat bằng lệnh `quit`.
- **Process B khởi chạy**
  - Chờ Process A tạo hàng đợi, kết nối thành công và hiển thị tiêu đề cùng trạng thái các luồng **SEND/RECV**.
  - Các luồng **SEND/RECV** của B sẵn sàng trao đổi dữ liệu với A.
- **Trao đổi tin nhắn**
  - Tin nhắn được gửi và nhận liên tục giữa A và B, đảm bảo giao tiếp hai chiều thời gian thực.
  - Luồng **SEND** và **RECV** chạy song song, không bị chặn.
  - Hàng đợi tin nhắn hoạt động ổn định, các tin nhắn được nhận đúng thứ tự, đảm bảo đáng tin cậy.
- **Xử lý thoát**
  - Khi nhấn `Ctrl+C` hoặc A gửi lệnh kết thúc, cả hai tiến trình nhận tín hiệu, in thông báo và dọn dẹp tài nguyên.
  - Đảm bảo không để lại rác hệ thống hay tin nhắn trong hàng đợi.

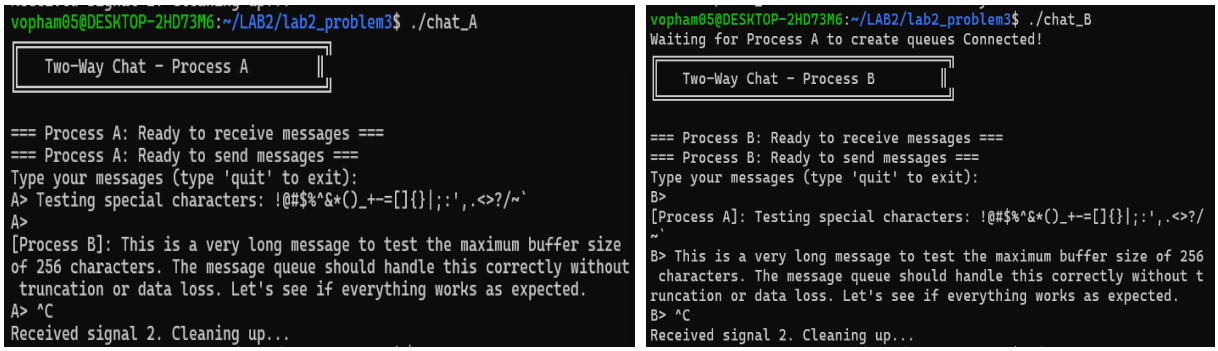
3.2.3. Testcase 3:



Hình 9: *Error Handling - Process B Timeout*

- **Tình huống:** Người dùng khởi chạy **Process B** trước **Process A**. Process B cần nhận tin nhắn từ hàng đợi `QUEUE_A_TO_B`, nhưng hàng đợi này chưa tồn tại vì Process A chưa chạy.
- **Hành vi của Process B:**
  - Kiểm tra sự tồn tại của hàng đợi.
  - Nếu hàng đợi chưa tồn tại, báo *Waiting for Process A to create queues.*
  - Nếu sau 30 giây queue vẫn chưa có, hiển thị lỗi và dừng thực thi.
- **Kết luận:** Process B phụ thuộc hoàn toàn vào Process A; phải chạy Process A trước thì Process B mới hoạt động bình thường.

3.2.4. Testcase 4:



Hình 10: *SPECIAL CHARACTERS and LONG MESSAGE*

- Gửi ký tự đặc biệt (`!@$%*()_+-=[]{}|;:',.<>?/ ‘`) để đảm bảo hệ thống xử lý đúng tất cả các ký tự ASCII mà không gây lỗi hoặc mất dữ liệu.
- Gửi văn bản rất dài (gần hoặc tối đa kích thước buffer, ví dụ 256 ký tự) để kiểm tra khả năng xử lý message dài của message queue, đảm bảo không bị cắt bớt hoặc tràn buffer.

Ý nghĩa của kết quả:

- Process B và A nhận đầy đủ toàn bộ nội dung, không mất dữ liệu → xác nhận buffer handling hoạt động ổn định.
- Kết quả này minh chứng hệ thống có thể xử lý các trường hợp biên (edge cases) một cách an toàn.

4 PROBLEM 4

4.1 Mô tả bài toán tổng quan

4.1.1 Mục tiêu bài toán

Vận dụng hàm hệ thống `mmap()` trong ngôn ngữ C để xây dựng cơ chế chia sẻ dữ liệu giữa hai tiến trình (Writer và Reader) thông qua một vùng nhớ dùng chung (shared memory) được ánh xạ trực tiếp từ file.

Cụ thể, chương trình cần thực hiện các nhiệm vụ sau:

- Ánh xạ file vào không gian địa chỉ cục bộ (local address space) của tiến trình, cho phép truy cập dữ liệu như thao tác với bộ nhớ trong RAM.
- Cho phép hai tiến trình độc lập cùng đọc và ghi dữ liệu trên cùng một vùng nhớ đã ánh xạ, thể hiện khả năng chia sẻ dữ liệu theo thời gian thực.
- Minh họa cơ chế giao tiếp liên tiến trình (IPC) thông qua bộ nhớ dùng chung dựa trên file, mà không cần sử dụng các phương thức truyền thống như `pipe`, `socket` hay `message queue`.

4.1.2 Nguyên lý hoạt động

- Hàm `mmap()` cho phép ánh xạ một file từ đĩa vào không gian địa chỉ ảo của tiến trình, giúp tiến trình truy cập dữ liệu như thao tác với vùng nhớ trong RAM mà không cần đọc/ghi file thủ công.
- Khi hai tiến trình cùng ánh xạ một file với cờ `MAP_SHARED`, cả hai sẽ chia sẻ cùng một vùng bộ nhớ vật lý do kernel quản lý, đảm bảo mọi thay đổi từ tiến trình này sẽ tức thời hiển thị ở tiến trình kia.
- Dữ liệu được đồng bộ qua `page cache` của hệ điều hành, giúp giao tiếp liên tiến trình (IPC) nhanh và hiệu quả, mà không cần sử dụng các cơ chế truyền thống như `pipe`, `socket` hay `message queue`.
- Để tránh xung đột khi đọc/ghi đồng thời, tiến trình thường sử dụng biến trạng thái (`status`) trong vùng nhớ chung để thông báo tình trạng dữ liệu (ví dụ: `READY`, `UPDATED`, `READ_AND_MODIFIED`)

4.1.3 Cấu trúc chương trình

Gồm ba tệp nguồn chính, mỗi tệp đảm nhận một vai trò riêng trong cơ chế chia sẻ dữ liệu:

- **shared_data.h**: Định nghĩa cấu trúc dữ liệu chung (`SharedData`), kích thước vùng nhớ và tên file chia sẻ (`SHARED_FILE`).
- **mmap_writer.c**: Tiến trình Writer tạo/mở file, ánh xạ vào bộ nhớ, ghi dữ liệu (biến đếm, thông điệp, mảng, trạng thái) và cập nhật cờ báo hiệu dữ liệu đã sẵn sàng.
- **mmap_reader.c**: Tiến trình Reader ánh xạ cùng file, đọc dữ liệu Writer ghi, hiển thị thông tin và có thể ghi phản hồi ngược lại (ví dụ: cập nhật trạng thái).

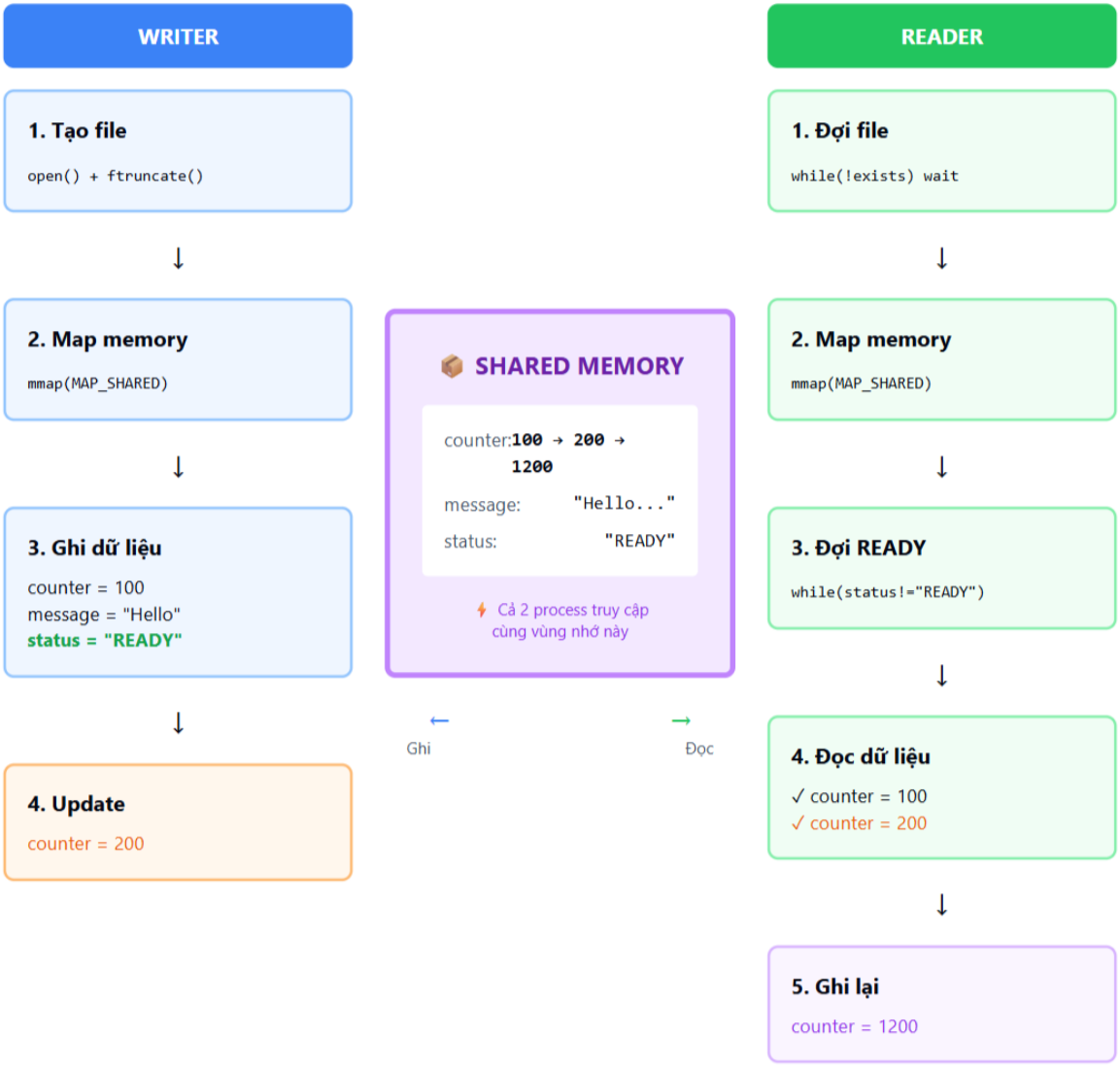
4.2 Quy trình thực hiện

Mô phỏng giao tiếp giữa hai tiến trình (Writer và Reader) thông qua bộ nhớ dùng chung (shared memory) ánh xạ từ file.

Quy trình thực hiện được chia thành các bước chính sau:

1. **Chuẩn bị vùng nhớ dùng chung**
   - Writer tạo hoặc mở file chia sẻ dữ liệu và đặt kích thước file đủ lớn để chứa toàn bộ thông tin cần chia sẻ (`ftruncate()`).
   - File được ánh xạ vào không gian địa chỉ ảo của tiến trình bằng `mmap()` với cờ `MAP_SHARED`, cho phép truy cập dữ liệu như vùng nhớ thông thường.
   - Cơ chế này đảm bảo mọi thay đổi từ một tiến trình sẽ tức thời phản ánh đến tiến trình khác và được kernel quản lý thông qua page cache.
2. **Khởi tạo dữ liệu và đồng bộ trạng thái**
   - Writer khởi tạo biến trạng thái (`status`) = "INIT" để báo Reader rằng dữ liệu chưa sẵn sàng.
   - Ghi dữ liệu ban đầu vào vùng nhớ, bao gồm:
     - Biến đếm (`counter`)
     - Thông điệp (`message`)
     - Mảng số nguyên (`data`)
     - Mảng số thực (`values`)
   - Sau khi ghi xong, Writer cập nhật trạng thái `status = "READY"`, báo hiệu Reader có thể bắt đầu đọc dữ liệu.
3. **Reader đọc dữ liệu**
   - Reader đợi file chia sẻ tồn tại và ánh xạ cùng file vào bộ nhớ (`mmap()` với `MAP_SHARED`).
   - Reader kiểm tra biến trạng thái (`status`) để chỉ đọc dữ liệu khi Writer đã chuẩn bị xong (`status = "READY"`).
   - Reader đọc toàn bộ dữ liệu từ vùng nhớ: biến đếm, thông điệp, mảng số nguyên và mảng số thực.
   - Reader có thể ghi phản hồi ngược lại vào vùng nhớ, ví dụ cập nhật `status` hoặc `counter`, minh họa khả năng giao tiếp hai chiều.
4. **Cập nhật dữ liệu trong thời gian thực**
   - Writer hoặc Reader có thể cập nhật dữ liệu đã ánh xạ trong khi tiến trình còn hoạt động:
     - Writer: cập nhật `counter`, `message`, `status = "UPDATED"`
     - Reader: ghi phản hồi, ví dụ `status = "READ_AND_MODIFIED"`
   - Các thay đổi được phản ánh tức thời nhờ cơ chế chia sẻ bộ nhớ `MAP_SHARED`.
5. **Giải phóng tài nguyên**
   - Khi hoàn tất, cả hai tiến trình gọi `munmap()` để hủy ánh xạ vùng nhớ, đóng file bằng `close()` và có thể xóa file chia sẻ nếu không còn sử dụng.
   - Quy trình này đảm bảo tài nguyên hệ thống được giải phóng an toàn và không gây rò rỉ bộ nhớ.

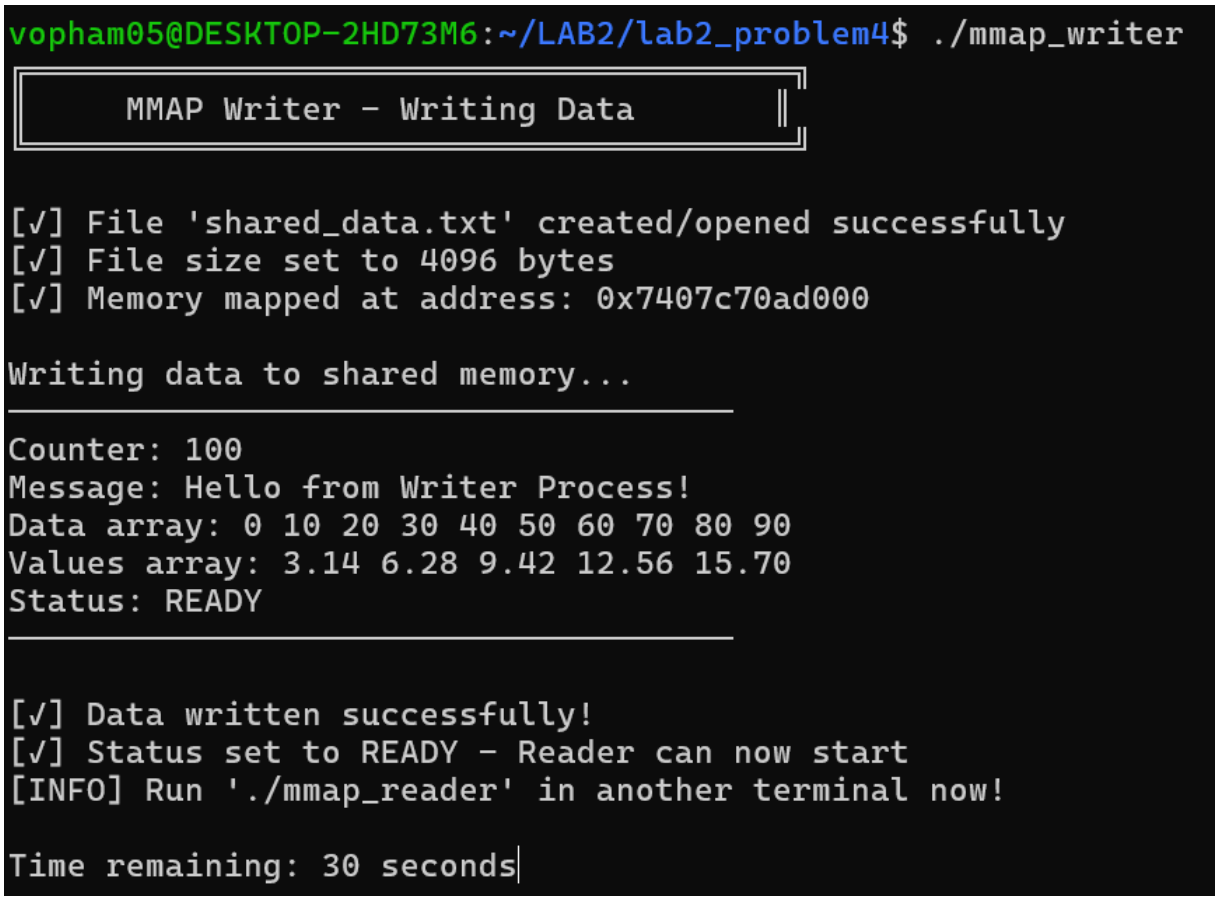
Tiến trình	Quy trình
Writer	Tạo/mở file → ánh xạ vào bộ nhớ → ghi dữ liệu → cập nhật `status = "READY"` → chờ Reader → cập nhật `"UPDATED"` → giải phóng vùng nhớ.
Reader	Đợi file tồn tại → ánh xạ vào bộ nhớ → kiểm tra `status = "READY"` → đọc dữ liệu → ghi phản hồi (tùy chọn) → giải phóng vùng nhớ.
Cơ chế chia sẻ	Dữ liệu được truyền trực tiếp qua page cache nhờ `MAP_SHARED`, không cần IPC truyền thống (pipe, socket, message queue).



Hình 11: *Sơ Đồ Minh Họa MMap - Chia sẽ dữ liệu giữa 2 Process*

4.3 Kết quả đầu ra và phân tích

4.3.1 Writer Process - Khởi tạo và ghi dữ liệu ban đầu



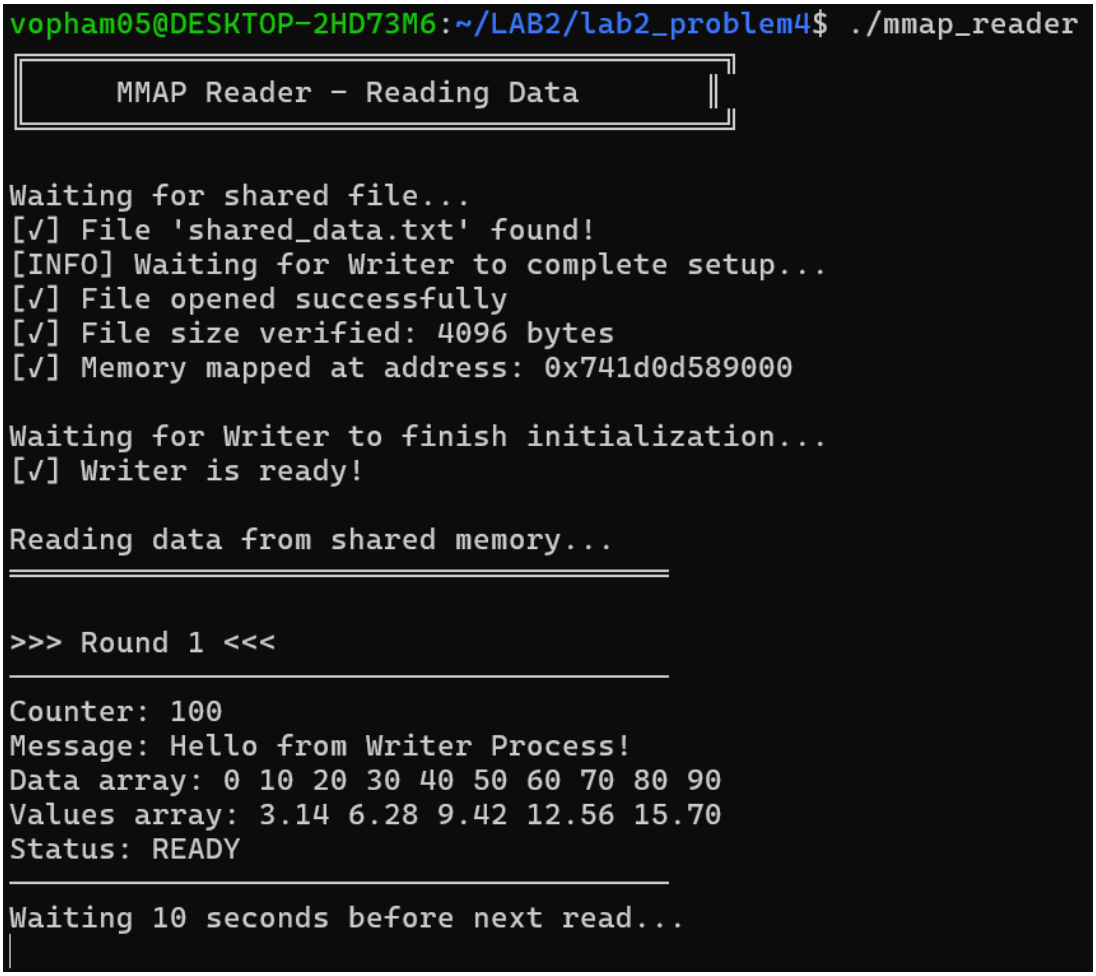
Hình 12: *Quá trình khởi tạo và ghi dữ liệu ban đầu của Writer Process*

Trong giai đoạn khởi tạo, tiến trình **Writer** tạo (hoặc mở lại nếu đã tồn tại) tệp `shared_data.txt`, thiết lập kích thước **4096 bytes**, và ánh xạ vùng nhớ của tệp này vào không gian địa chỉ tiến trình tại `0x7afc661a0000`. Sau đó, Writer ghi dữ liệu ban đầu gồm:

- **Counter** = 100
- **Message** = "Hello from Writer Process!"
- **Data array**: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90
- **Values array**: 3.14, 6.28, 9.42, 12.56, 15.70
- **Status** = READY

Dữ liệu khởi tạo hợp lệ, trạng thái `READY` cho phép Reader truy cập. Tiến trình Writer tạm dừng 30 giây để chờ tiến trình Reader kết nối và đọc dữ liệu.

4.3.2 Reader Process – Kết nối và đọc dữ liệu đầu tiên



Hình 13: *Reader Process – Kết nối và đọc dữ liệu (vòng đọc 1)*

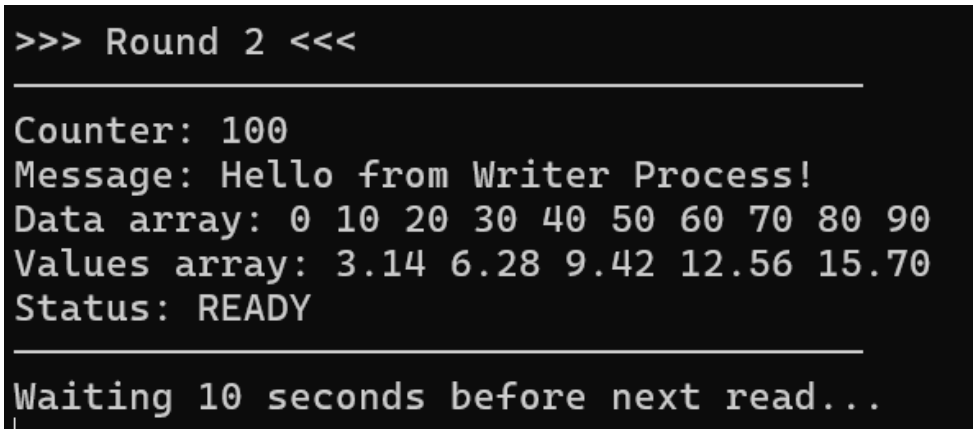
Trong giai đoạn này (sau hơn 10s từ khi Writer Process khởi tạo) , tiến trình **Reader** mở cùng tệp `shared_data.txt`, kiểm tra kích thước hợp lệ và ánh xạ vùng nhớ của tệp vào không gian địa chỉ của mình tại `0x7a629e71f000`. Sau khi tiến trình **Writer** thiết lập trạng thái `READY`, Reader bắt đầu truy cập và đọc toàn bộ dữ liệu khởi tạo, bao gồm:

- **Counter** = 100
- **Message** = "Hello from Writer Process!"
- **Data array**: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90
- **Values array**: 3.14, 6.28, 9.42, 12.56, 15.70

Phân tích:

- Mặc dù vùng nhớ được ánh xạ tại địa chỉ khác nhau trong hai tiến trình, Reader vẫn truy cập chính xác cùng vùng dữ liệu vật lý do Writer tạo ra.
- Toàn bộ dữ liệu được đọc ra hoàn toàn trùng khớp với nội dung mà Writer đã ghi, thể hiện tính toàn vẹn và ổn định của cơ chế chia sẻ bộ nhớ.
- Kết quả chứng minh rằng cơ chế `memory mapping` bảo đảm sự nhất quán dữ liệu giữa các tiến trình độc lập, đồng thời loại bỏ nhu cầu trao đổi thông qua cơ chế I/O truyền thống.

4.3.3 Reader Round 2



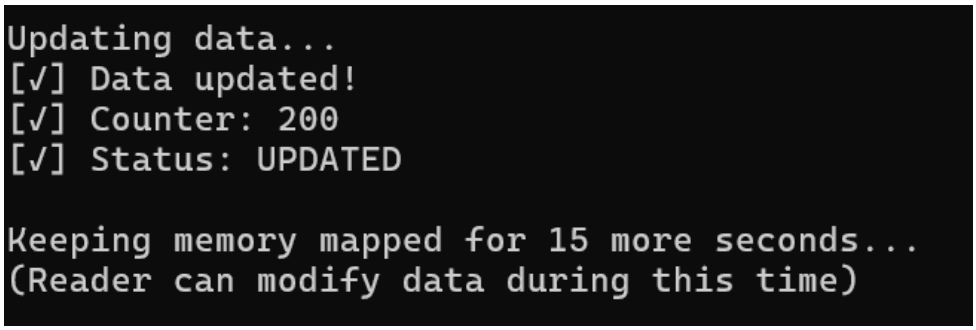
Hình 14: *Reader Round 2 – Dữ liệu ổn định trong thời gian chờ*

Sau khoảng 10 giây (tương ứng với vòng đọc thứ hai), tiến trình **Reader** tiếp tục truy cập vùng nhớ chia sẻ và nhận được cùng một tập dữ liệu như trước:

- **Counter** = 100, **Message** = "Hello from Writer Process!"
- **Data array**: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90
- **Values array**: 3.14, 6.28, 9.42, 12.56, 15.70
- **Status** = READY

Tại thời điểm này, tiến trình **Writer** vẫn đang trong giai đoạn tạm dừng 30 giây và chưa thực hiện bất kỳ thao tác ghi hoặc cập nhật nào lên vùng nhớ.

4.3.4 Writer Update Data



Hình 15: *Writer cập nhật dữ liệu sau 30s*

Sau khi kết thúc khoảng thời gian chờ ban đầu (30 giây), tiến trình **Writer** tiến hành cập nhật nội dung trong vùng bộ nhớ dùng chung (`shared memory`) như sau:

- **Counter**: 100 → 200
- **Message**: "Updated message!"
- **Status**: `READY` → `UPDATED`

Sau khi ghi dữ liệu mới, Writer tiếp tục duy trì ánh xạ bộ nhớ thêm 15 giây để tiến trình **Reader** có thể phát hiện và đọc được sự thay đổi này.

4.3.5 Reader Round 3

Hình 16: *Reader Round 3 – Phát hiện thay đổi từ Writer*

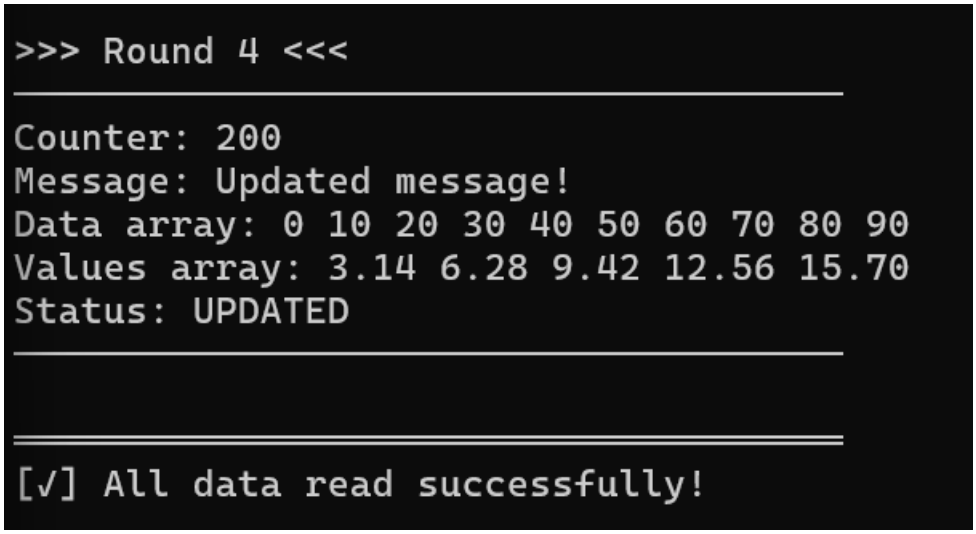
Trong vòng đọc thứ ba, tiến trình **Reader** phát hiện các giá trị trong vùng bộ nhớ chia sẻ đã được cập nhật:

- **Counter**: 100 → 200
- **Message**: "Hello from Writer Process!" → "Updated message!"
- **Status**: `READY` → `UPDATED`

Kết quả này chứng minh rằng Reader nhận được các thay đổi từ Writer theo thời gian thực, mặc dù hai tiến trình hoạt động độc lập.

Đây là kết quả trọng tâm, minh chứng khả năng đồng bộ dữ liệu giữa các tiến trình. Reader có thể tự động nhận thấy dữ liệu mới mà không cần thao tác đọc lại tệp từ đĩa, thể hiện hiệu quả của bộ nhớ chia sẻ (*shared memory*).

4.3.6 Reader Round 4



Hình 17: *Reader Round 4 – Xác nhận dữ liệu mới*

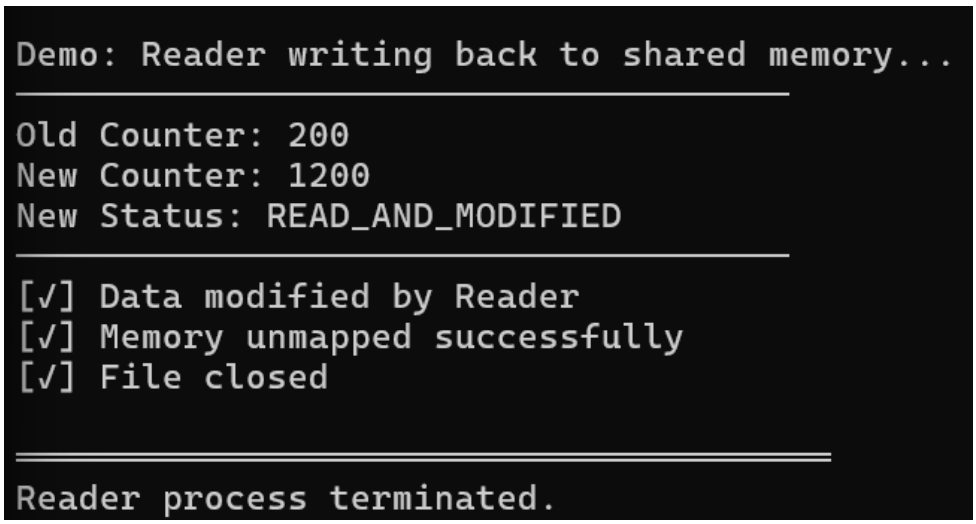
Trong vòng đọc thứ tư, tiến trình **Reader** tiếp tục truy cập vùng bộ nhớ chia sẻ và nhận được các giá trị đã được cập nhật:

- **Counter** = 200
- **Message**: Updated message!
- **Status** = UPDATED

Kết quả này xác nhận rằng các giá trị mới được duy trì ổn định trong vùng nhớ chia sẻ, không bị thay đổi hay mất mát.

Dữ liệu trong `shared memory` được lưu trữ một cách nhất quán giữa các tiến trình, đảm bảo tính toàn vẹn dữ liệu. Không xảy ra hiện tượng mất dữ liệu hay rollback, minh chứng cơ chế *memory mapping* duy trì trạng thái ổn định ngay cả khi Reader và Writer hoạt động độc lập.

4.3.7 Bidirectional Communication



Hình 18: *Bidirectional Communication – Reader ghi lại dữ liệu*

Để minh chứng khả năng truyền thông hai chiều, tiến trình **Reader** thực hiện ghi ngược lại vào vùng bộ nhớ chia sẻ (`shared memory`) với các giá trị mới:

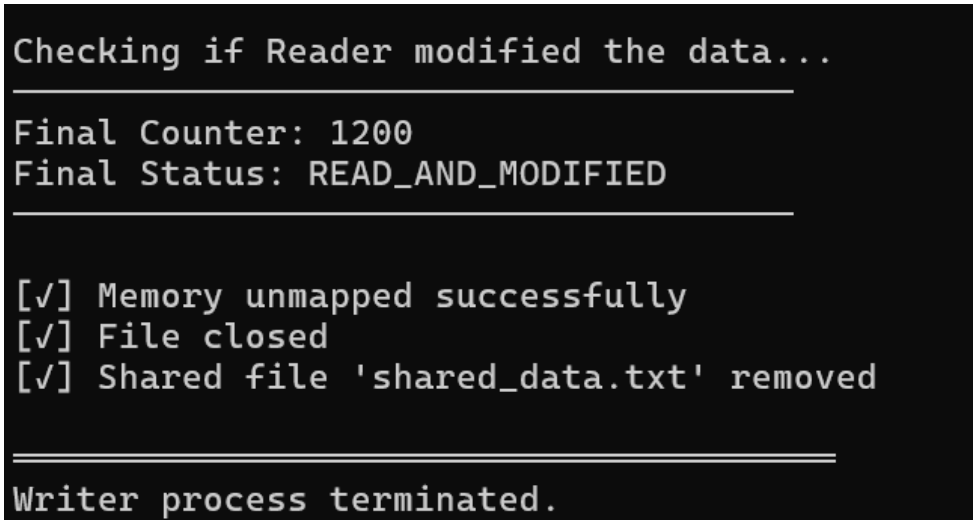
- **Counter**: 200 → 1200
- **Status**: `UPDATED` → `READ_AND_MODIFIED`

Sau khi thực hiện ghi dữ liệu, Reader thông báo cho Writer rằng nội dung đã được chỉnh sửa.

Giải thích:

- Reader có quyền ghi dữ liệu, không chỉ đơn thuần đọc, thể hiện khả năng tương tác hai chiều (*bidirectional communication*) giữa các tiến trình.
- Cơ chế này minh chứng rằng `shared memory` cho phép cả hai tiến trình trao đổi dữ liệu qua cùng một vùng nhớ vật lý mà không cần sử dụng thêm cơ chế IPC bổ sung.
- Việc Writer nhận thông báo từ Reader xác nhận dữ liệu đã được chỉnh sửa cho thấy sự đồng bộ và quản lý trạng thái hiệu quả giữa các tiến trình.

4.3.8 Writer nhận thay đổi từ Reader



Hình 19: *Writer nhận thay đổi từ Reader*

Sau khi Reader ghi dữ liệu trở lại, tiến trình **Writer** kiểm tra vùng bộ nhớ chia sẻ và nhận thấy:

- **Counter** = 1200 (thay vì 200)
- **Status** = `READ_AND_MODIFIED`

Việc này xác nhận rằng Reader đã thực hiện ghi ngược dữ liệu thành công. Sau khi kiểm tra, Writer tiến hành **cleanup** để kết thúc thí nghiệm:

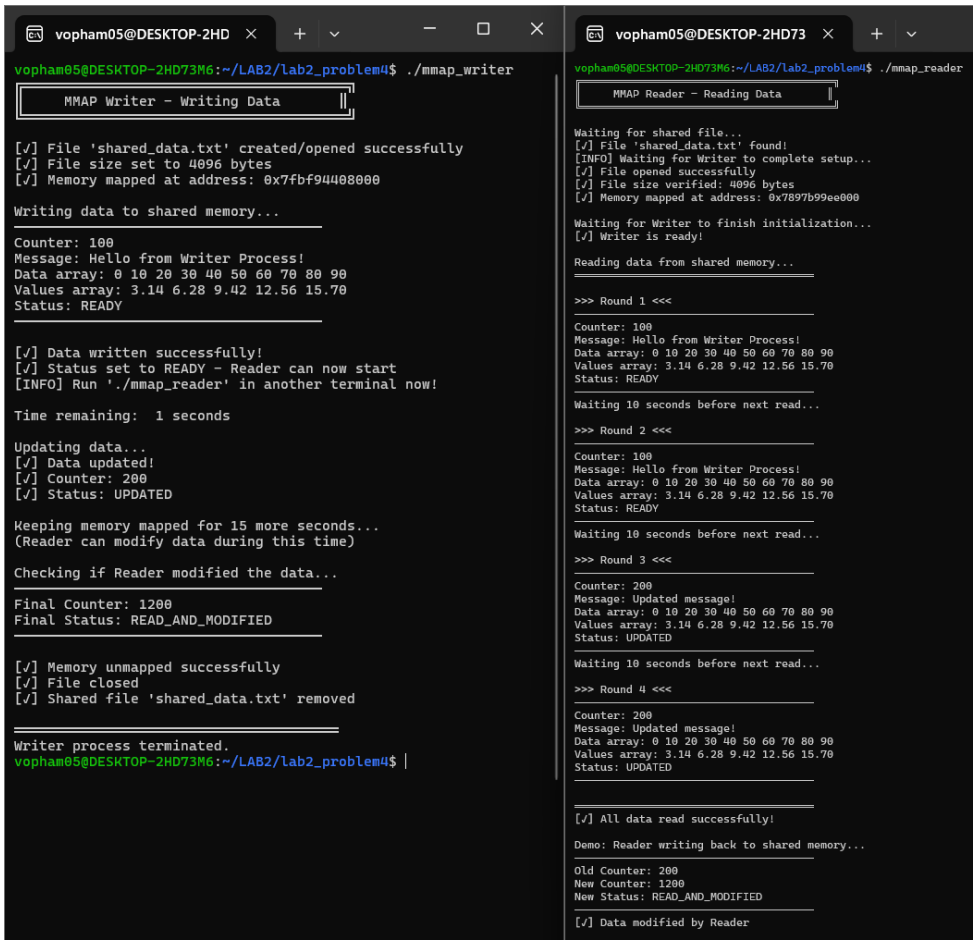
- Hủy ánh xạ vùng nhớ (`unmap`)
- Đóng file `shared_data.txt`
- Xóa file khỏi hệ thống

Giải thích:

- Writer quan sát được thay đổi do Reader thực hiện, chứng minh rằng giao tiếp hai chiều (*bidirectional communication*) qua shared memory đã hoạt động thành công.
- Quá trình cleanup đảm bảo giải phóng bộ nhớ và tài nguyên hệ thống một cách an toàn, tránh rò rỉ tài nguyên sau khi kết thúc thí nghiệm.

Bảng 3: *Timeline tổng hợp quá trình giao tiếp giữa Writer và Reader*

Thời điểm	Tiến trình	Sự kiện chính
0 s	Writer	Khởi tạo, tạo file, map memory và ghi Counter=100, Status=READY
12 s	Reader	Vòng 1 – Đọc dữ liệu ban đầu (Counter=100)
22 s	Reader	Vòng 2 – Dữ liệu chưa thay đổi
30 s	Writer	Cập nhật dữ liệu: Counter=200, Message="Updated message!", Status=UPDATED
32 s	Reader	Vòng 3 – Phát hiện thay đổi từ Writer
42 s	Reader	Vòng 4 – Xác nhận dữ liệu mới (Counter=200)
42 s	Reader	Ghi lại dữ liệu vào shared memory: Counter=1200, Status=READ_AND_MODIFIED
45 s	Writer	Phát hiện dữ liệu do Reader ghi lại (Counter=1200)
50 s	Writer	Cleanup: unmap memory, đóng file, xóa file shared_data.txt



Hình 20: *Kết quả tổng hợp*

Tài liệu