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

HỆ ĐIỀU HÀNH (CO2017)

THỰC HÀNH 4

SCHEDULING

Lớp: L02 – Học Kỳ HK251
Ngày nộp: 11/12/2025

GVHD: Thầy Nguyễn Minh Tâm

Sinh Viên	MSSV
Phạm Công Võ	2313946

TP. Hồ Chí Minh, Tháng 12/2025

Mục lục

Danh sách hình vẽ

1 PROBLEM 1 - Cài đặt chính sách lập lịch FIFO

1.1 Giới thiệu

Trong hệ thống đa nhiệm, bộ lập lịch (scheduler) là thành phần quyết định việc phân phối tài nguyên xử lý cho các tác vụ đang chờ. Mặc dù BK TaskPool đã cung cấp cơ chế tạo worker, truyền tín hiệu và thực thi nhiệm vụ, chức năng lựa chọn worker vẫn được để trống nhằm giúp sinh viên tự hiện thực. Problem 1 yêu cầu xây dựng chính sách lập lịch **FIFO (First-In, First-Out)** – một thuật toán cơ bản nhưng giữ vai trò quan trọng trong việc đảm bảo tính công bằng và tính dự đoán của hệ thống.

Việc triển khai FIFO giúp sinh viên hiểu rõ cách scheduler theo dõi trạng thái worker, cách dispatcher tương tác với worker thông qua tín hiệu, cũng như cách hệ thống duy trì tiến trình xử lý tác vụ theo đúng thứ tự yêu cầu.

1.2 Yêu cầu của bài toán

- Xây dựng nội dung hàm `bkwrk_get_worker()` để lựa chọn worker theo đúng nguyên tắc FIFO.
- FIFO phải đảm bảo rằng worker được chọn là **worker rảnh đầu tiên** theo thứ tự tăng dần ID.
- Worker chỉ được xem là sẵn sàng khi `wrkid_busy[i] == 0`.
- Scheduler phải trả về ID hợp lệ để dispatcher gửi tín hiệu kích hoạt đúng worker.
- Khi worker hoàn tất nhiệm vụ, worker phải tự đánh dấu trạng thái rảnh để chuẩn bị nhận nhiệm vụ tiếp theo.

1.3 Cách thức hiện thực

Trong Problem 1, hệ thống cần xây dựng cơ chế lập lịch FIFO nhằm phân phối tác vụ cho các worker trong BK TaskPool. Việc hiện thực được tổ chức xoay quanh mô hình *worker-thread* kết hợp cơ chế đồng bộ hóa bằng tín hiệu (*signal-based synchronization*).

Kiến trúc tổng thể bao gồm bốn mô-đun chính: (1) khởi tạo worker pool, (2) vòng lặp xử lý của worker, (3) ánh xạ tác vụ vào worker, và (4) thuật toán lựa chọn worker theo FIFO. Các thành phần này phối hợp với nhau để hình thành một pipeline xử lý tác vụ nhẹ, ổn định và hiệu quả.

1. Khởi tạo worker pool bằng `clone()`

Worker pool được tạo trong hàm `bkwrk_create_worker()`, sử dụng system call `clone()` để sinh ra nhiều worker hoạt động song song. Mỗi worker được cấu hình với các đặc điểm sau:

- Stack độc lập, đảm bảo không gian thực thi riêng.
- Chung không gian địa chỉ thông qua cờ `CLONE_VM`, giúp việc chia sẻ thông tin giữa các worker và thread cha diễn ra tức thời.
- Chung bộ mô tả file nhờ cờ `CLONE_FILES`, hỗ trợ in ấn và giao tiếp hệ thống đơn giản.
- Nhận worker ID thông qua tham số truyền vào khi gọi `clone()`.

Sau khi khởi tạo, worker được thiết lập về trạng thái `IDLE` và trở thành một phần của *lightweight thread pool*. Cách tiếp cận này cho phép nhiều worker sẵn sàng xử lý tác vụ mà không tiêu tốn tài nguyên dư thừa.

2. Vòng lặp xử lý tác vụ trong worker

Hàm `bkwrk_worker()` định nghĩa vòng đời của mỗi worker. Worker hoạt động theo hai pha chính: **pha chờ tín hiệu** và **pha xử lý tác vụ**.

A - Chờ tín hiệu:

Worker block tại lệnh:

```
sigwait(&set, &sig);
```

Ưu điểm của cơ chế này:

- Worker hoàn toàn không tiêu tốn CPU khi không có nhiệm vụ.
- Worker chỉ được đánh thức khi nhận tín hiệu `SIGUSR1` từ dispatcher.
- Signal masking đảm bảo đồng bộ hóa tuyệt đối, tránh race condition.

B - Thực thi tác vụ:

Khi nhận tín hiệu đánh thức:

- Worker lấy hàm thực thi và tham số từ `worker[i].func` và `worker[i].arg`.
- Thực hiện tác vụ bằng lời gọi:

```
wrk->func(wrk->arg);
```

- Sau khi hoàn thành:
  - Đặt lại trạng thái về `IDLE` (`wrkid_busy[i] = 0`),
  - Xóa sạch thông tin task để tránh xử lý lặp lại.

3. Ánh xạ tác vụ vào worker

Việc gán tác vụ được thực hiện trong hàm `bktask_assign_worker()`, bao gồm:

- Truy xuất thông tin tác vụ từ task pool thông qua `bktask_get_byid()`.
- Sao chép hàm thực thi và tham số vào cấu trúc worker.
- Đánh dấu worker sang trạng thái `BUSY`.

Nhờ chia sẻ bộ nhớ (`CLONE_VM`), quá trình ánh xạ chỉ là thao tác ghi trực tiếp, nhanh và không yêu cầu cơ chế IPC.

4. Kích hoạt worker bằng tín hiệu hệ thống

Sau khi tác vụ được gán, worker được đánh thức bằng cách gọi:

```
syscall(SYS_tkill, tid, SIG_DISPATCH);
```

Lợi ích của dispatch bằng tín hiệu:

- Worker được đánh thức đúng thời điểm và đúng nhiệm vụ.
- Không sử dụng busy-waiting, giảm tiêu thụ CPU.
- Worker thoát khỏi `sigwait()` một cách an toàn và chính xác.

5. Cơ chế lập lịch FIFO – Trọng tâm của Problem 1

Thuật toán FIFO được hiện thực trong hàm:

```
for (int i = 0; i < MAX_WORKER; i++)
    if (wrkid_busy[i] == 0)
        return i;
```

Đây là một cài đặt tinh gọn và chính xác của *FIFO Worker Scheduling*, đặc trưng bởi:

- Duyệt worker theo thứ tự ID tăng dần.
- Worker trở về trạng thái `IDLE` sớm nhất sẽ được ưu tiên phục vụ task tiếp theo.
- Tránh hoàn toàn hiện tượng starvation.
- Phù hợp với worker pool đồng nhất.

6. Luồng thực thi tổng thể của Problem 1

Quá trình xử lý tác vụ trong hệ thống diễn ra theo pipeline:

1. **Main** khởi tạo task và worker pool.
2. **Scheduler** (FIFO) tìm worker rảnh đầu tiên.
3. **Assignment**: Task được ánh xạ vào worker.
4. **Dispatch**: Gửi tín hiệu đánh thức worker.
5. **Worker thực thi**:
   - Worker wake từ `sigwait()`,
   - Thực hiện hàm tác vụ,
   - Đặt lại trạng thái về `IDLE`.
6. Scheduler tiếp tục phân phối task tiếp theo.

1.4 Output đầu ra

Khi thực thi chương trình:

```
./mypool
```

Hệ thống khởi tạo BK TaskPool theo mô hình **worker–thread** sử dụng `clone()`, kết hợp cơ chế đồng bộ dựa trên tín hiệu (`sigwait()` và `tkill()`). Các dòng output phản ánh chính xác pipeline thực thi mà mã nguồn đã cài đặt.

```
PS C:\Users\ASUS\Desktop\LAB 251\LAB4_Scheduling\lab4-student\Problem1> wsl
vophamk23@VoPham:/mnt/c/Users/ASUS/Desktop/LAB 251/LAB4_Scheduling/lab4-student/Problem1$ make clean
rm -f mypool main.o bktpool.o bktask.o bkwrk.o *.o
vophamk23@VoPham:/mnt/c/Users/ASUS/Desktop/LAB 251/LAB4_Scheduling/lab4-student/Problem1$ make
gcc -pthread -D_GNU_SOURCE -DINFO -w -c main.c -o main.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -c bktpool.c -o bktpool.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -c bktask.c -o bktask.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -c bkwrk.c -o bkwrk.o
gcc -pthread -o mypool main.o bktpool.o bktask.o bkwrk.o
vophamk23@VoPham:/mnt/c/Users/ASUS/Desktop/LAB 251/LAB4_Scheduling/lab4-student/Problem1$ ./mypool
bkwrk_create_worker got worker 550
bkwrk_create_worker got worker 551
bkwrk_create_worker got worker 552
bkwrk_create_worker got worker 553
bkwrk_create_worker got worker 554
bkwrk_create_worker got worker 555
bkwrk_create_worker got worker 556
bkwrk_create_worker got worker 557
bkwrk_create_worker got worker 558
bkwrk_create_worker got worker 559
Assign tsk 0 wrk 0
worker wake 0 up
Task func - Hello from 1
Assign tsk 1 wrk 0 >>>>>>>>>> Activate asynchronously
Assign tsk 2 wrk 1 >>>>>>>>>> Activate asynchronously
worker wake 0 up
Task func - Hello from 2
worker wake 1 up
Task func - Hello from 5
```

Hình 1: *Kết quả thực thi Problem 1 (Worker scheduling bằng clone() – Thread Mode)*

Phân tích quá trình chạy và giải thích Output - Problem 1

1. Tạo worker - `clone()` sinh 10 worker thread và đưa vào trạng thái chờ

Ngay khi chương trình khởi động, hệ thống in ra:

```
bkwrk_create_worker got worker 586
bkwrk_create_worker got worker 587
bkwrk_create_worker got worker 588
bkwrk_create_worker got worker 589
bkwrk_create_worker got worker 590
bkwrk_create_worker got worker 591
bkwrk_create_worker got worker 592
bkwrk_create_worker got worker 593
bkwrk_create_worker got worker 594
bkwrk_create_worker got worker 595
```

Các dòng này xuất phát từ hàm `bkwrk_create_worker()` tại thời điểm mỗi worker được tạo bằng:

```
1    clone(bkwrk_worker, stack_top,
2          CLONE_VM | CLONE_FILES,
3          &worker_ids[i]);
```

Ý nghĩa kỹ thuật:

- Mỗi worker là một thread độc lập có stack riêng.
- Tất cả worker chia sẻ chung không gian bộ nhớ với master (nhờ `CLONE_VM`).
- Worker chuyển ngay vào `bkwrk_worker()` và block tại `sigwait()`, không tiêu tốn CPU khi không có tác vụ.

Kết quả: Worker pool gồm 10 thread ở trạng thái **IDLE**, sẵn sàng nhận nhiệm vụ.

2. Gán tác vụ đầu tiên - FIFO chọn worker 0

Output:

```
Assign tsk 0 wrk 0
```

Đây là kết quả của thuật toán FIFO:

```
for (i = 0; i < MAX_WORKER; i++)
    if (wrkid_busy[i] == 0)
        return i;
```

Worker 0 là worker rảnh đầu tiên nên được chọn. Master gán task bằng cách ghi trực tiếp vào bộ nhớ dùng chung:

```
worker[0].func = tsk->func;
worker[0].arg  = tsk->arg;
wrkid_busy[0]  = 1;
```

Không cần IPC: toàn bộ truyền thông chỉ là thao tác ghi bộ nhớ, cực nhanh và không tạo overhead.

3. Dispatch - master gửi tín hiệu đánh thức worker 0

Master đánh thức worker bằng:

```
syscall(SYS_tkill, tid, SIGUSR1);
```

Worker thoát khỏi `sigwait()` và in ra:

```
worker wake 0 up
Task func - Hello from 1
```

Ý nghĩa:

- Worker nhận đúng tín hiệu được gửi từ master.
- Worker đọc thông tin tác vụ từ cấu trúc worker.
- Tiến hành thực thi hàm tác vụ.
- Sau khi hoàn thành, đặt trạng thái về IDLE.

4. Scheduling bất đồng bộ - master tiếp tục lập lịch song song

Output tiếp theo:

```
Assign tsk 1 wrk 0 >>>>>>>>>> Activate asynchronously
Assign tsk 2 wrk 1 >>>>>>>>>> Activate asynchronously
```

Điều này phản ánh đúng logic của hệ thống:

- Task 1: tiếp tục được xếp cho worker 0, nhưng chỉ thực thi khi worker 0 rảnh.
- Task 2: được gán cho worker 1 vì worker 1 là worker rảnh kế tiếp theo FIFO.

Dòng "`Activate asynchronously`" cho biết master **vẫn lập lịch trong khi worker đang chạy**, thể hiện cơ chế **asynchronous scheduling** của mô hình thread.

5. Worker quay lại vòng lặp - tiếp tục xử lý tác vụ kế tiếp

Sau khi hoàn thành tác vụ và trở về `sigwait()`, các worker được kích hoạt:

```
worker wake 0 up
Task func - Hello from 2
worker wake 1 up
Task func - Hello from 5
```

Chu trình hoạt động lặp lại đúng thiết kế:

1. Worker block trong `sigwait()`.
2. Master gán task và gửi tín hiệu.
3. Worker đọc func/arg và thực thi.
4. Worker đặt trạng thái về IDLE.
5. Worker quay lại `sigwait()` chờ nhiệm vụ mới.

2 PROBLEM 2 - Chuyển đổi worker sang tiến trình (fork)

2.1 Giới thiệu

Trong phiên bản mặc định của BK TaskPool, mỗi worker được khởi tạo dưới dạng một thread thông qua `clone()` hoặc `pthread_create()`. Mặc dù mô hình thread rất nhẹ và thuận tiện, mô hình tiến trình (process) lại mang tính tách biệt mạnh mẽ hơn, giúp sinh viên hiểu sâu hơn về cơ chế quản lý bộ nhớ, PID độc lập và tín hiệu liên tiến trình. Problem 2 yêu cầu triển khai lại worker bằng cách sử dụng `fork()`, từ đó tạo ra mỗi worker như một tiến trình độc lập.

Việc áp dụng `fork()` giúp người học cảm nhận rõ ràng hơn sự khác biệt giữa thread và process, cũng như cách hệ điều hành thực hiện điều phối và gửi tín hiệu trong môi trường đa tiến trình.

2.2 Yêu cầu của bài toán

- Viết lại hàm tạo worker trong mục 3.1.1 sao cho mỗi worker được tạo bằng `fork()`.
- Mỗi worker là một **tiến trình riêng biệt**, hoạt động độc lập với PID của chính nó.
- Worker phải duy trì vòng lặp chờ tín hiệu (`sigwait`) và xử lý nhiệm vụ tương tự như phiên bản dùng thread.
- Cơ chế dispatch phải gửi tín hiệu đến đúng PID bằng `kill(pid, SIG_DISPATCH)`.
- Worker phải cập nhật trạng thái bận/rảnh tương thích với hệ thống TaskPool.

2.3 Cách thức hiện thực

Problem 2 mở rộng hệ thống sang mô hình đa tiến trình bằng cách thay thế toàn bộ worker-thread bằng worker-process được tạo bởi `fork()`. Sự chuyển đổi này làm thay đổi cơ chế chia sẻ dữ liệu và đồng bộ hóa, buộc hệ thống triển khai một kiến trúc mới dựa trên POSIX Shared Memory kết hợp Signal IPC, nhưng vẫn giữ nguyên mô hình lập lịch FIFO từ Problem 1.

Kiến trúc thực thi xoay quanh bốn thành phần chính:

1. Tạo worker bằng `fork()`.
2. Duy trì vùng nhớ dùng chung cho master và worker.
3. Worker hoạt động theo cơ chế event–driven với `sigwait()`.
4. Master dispatch nhiệm vụ bằng tín hiệu hệ điều hành.

1. Khởi tạo worker bằng `fork()`

Hệ thống tạo từng worker trong hàm `bkwrk_create_worker()` bằng:

```
1   pid = fork();
```

Đặc điểm của mô hình:

- Mỗi worker là một tiến trình độc lập, có không gian bộ nhớ riêng.
- Worker không truy cập được biến toàn cục của master.
- Master lưu lại PID trong `wrkid_tid[]` để gửi tín hiệu.

Quy trình khởi tạo:

- Master cố định `worker_id` để tránh race condition.
- Khoá tín hiệu `SIGUSR1` và `SIGQUIT` trước khi fork.
- Tiến trình con nhảy vào vòng lặp `bkwrk_worker()`.
- Tiến trình cha khởi tạo worker trong shared memory.

Kết quả: hệ thống hình thành một tập worker-process độc lập nhưng được điều khiển thống nhất.

2. POSIX Shared Memory

Do worker-process không thể truy cập biến toàn cục, hệ thống sử dụng hai vùng shared memory:

- **Busy Array**: lưu trạng thái rảnh/bận.
- **Worker Array**: chứa `func`, `arg`, và `bktaskid`.

Shared memory được tạo bằng:

```
mmap(... MAP_SHARED ...)
```

Ý nghĩa:

- Master cập nhật task và trạng thái → worker nhìn thấy tức thời.
- Không cần pipe, socket hay message queue.
- Tốc độ cao, đồng bộ thời gian thực.

3. Vòng đời worker-process

Worker-process vận hành trong `bkwrk_worker()` theo hai bước lặp:

(1) Chờ tín hiệu bằng `sigwait()`

```
sigwait(&set, &sig);
```

Đặc điểm:

- Worker không tiêu tốn CPU khi idle.
- Chỉ hoạt động khi master gửi `SIG_DISPATCH`.
- Loại bỏ hoàn toàn busy-waiting.

(2) Thực thi tác vụ

```
wrk->func(wrk->arg);
```

Sau khi hoàn tất:

- Đặt lại `busy = 0`.
- Xoá `func`, `arg`, `taskid`.
- Quay về trạng thái chờ trong `sigwait()`.

4. Gán tác vụ bằng shared memory

Trong `bktask_assign_worker()`:

- Master ghi trực tiếp `func`, `arg` vào worker array.
- Đặt trạng thái `busy = 1`.

5. Dispatch worker bằng tín hiệu

Worker được kích hoạt bằng:

```
kill(pid, SIG_DISPATCH);
```

Ưu điểm:

- Worker thoát khỏi `sigwait()` ngay lập tức.
- Không cần lock hay queue.
- IPC đơn giản nhưng hiệu quả.

6. Thuật toán lập lịch FIFO

Cơ chế FIFO được giữ nguyên:

```
for (i = 0; i < MAX_WORKER; i++)
    if (!busy[i]) return i;
```

Shared memory đảm bảo trạng thái worker được cập nhật chính xác, giúp:

- Tránh starvation,
- Duy trì tính công bằng,
- Phù hợp mô hình worker đồng nhất.

7. Pipeline tổng thể Problem 2

1. Master tạo worker-process bằng `fork()`.
2. Khởi tạo shared memory cho busy array và worker array.
3. Worker vào vòng `sigwait()`.
4. Master tạo task và chọn worker theo FIFO.
5. Ghi dữ liệu task vào shared memory.
6. Gửi tín hiệu `SIGUSR1` cho worker.
7. Worker thực thi hàm `func(arg)`.
8. Worker đánh dấu rảnh và quay lại `sigwait()`.

Pipeline này mô phỏng hoàn chỉnh mô hình thread scheduling trên kiến trúc đa tiến trình, đảm bảo tính độc lập, an toàn và hiệu năng cao.

2.4 Output đầu ra

Khi biên dịch Problem 2, hệ thống cung cấp hai chế độ thực thi:

- **Thread mode** (`make thread`): Worker hoạt động như các *thread* dựa trên `clone()` và chia sẻ toàn bộ không gian địa chỉ với tiến trình chính.
- **Fork mode** (`make fork`): Worker hoạt động như các *process độc lập*, sử dụng `fork()` kết hợp với POSIX Shared Memory để trao đổi dữ liệu.

Thread mode chủ yếu dùng để đối chiếu với Problem 1; phần trọng tâm của Problem 2 là cơ chế fork-mode.

1. Thread Mode (`make thread`) - Tương tự Problem 1

Khi chạy:

```
make thread
./mypool
```

quá trình xử lý tương tự hoàn toàn Problem 1 vì:

- Các worker được tạo bằng `clone()` với cờ `CLONE_VM`, cho phép chia sẻ bộ nhớ với master.
- Trạng thái worker (`wrkid_busy[]` và `worker[]`) là biến toàn cục nằm trong chung một không gian địa chỉ.
- Worker block tại `sigwait()` và được đánh thức bằng `tkill(tid, SIGUSR1)`.

Kết quả output :

```
PS C:\Users\ASUS\Desktop\LAB 251\LAB4_Scheduling\lab4-student\Problem2> wsl
vophamk23@VoPham:/mnt/c/Users/ASUS/Desktop/LAB 251/LAB4_Scheduling/lab4-student/Problem2$ make thread
rm -f mypool main.o bktpool.o bktask.o bkwrk.o bkshm.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -DWORK_THREAD -c main.c -o main.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -DWORK_THREAD -c bktpool.c -o bktpool.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -DWORK_THREAD -c bktask.c -o bktask.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -DWORK_THREAD -c bkwrk.c -o bkwrk.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -DWORK_THREAD -c bkshm.c -o bkshm.o
gcc -pthread -lrt -o mypool main.o bktpool.o bktask.o bkwrk.o bkshm.o
vophamk23@VoPham:/mnt/c/Users/ASUS/Desktop/LAB 251/LAB4_Scheduling/lab4-student/Problem2$ ./mypool
bkwrk_create_worker got worker 685
bkwrk_create_worker got worker 686
bkwrk_create_worker got worker 687
bkwrk_create_worker got worker 688
bkwrk_create_worker got worker 689
bkwrk_create_worker got worker 690
bkwrk_create_worker got worker 691
bkwrk_create_worker got worker 692
bkwrk_create_worker got worker 693
bkwrk_create_worker got worker 694
Assign tsk 0 wrk 0
worker wake 0 up
Task func - Hello from 1
Assign tsk 1 wrk 0 >>>>>>>>>>> Activate asynchronously
Assign tsk 2 wrk 1 >>>>>>>>>>> Activate asynchronously
worker wake 0 up
Task func - Hello from 2
worker wake 1 up
Task func - Hello from 5
```

Hình 2: *Kết quả thực thi Problem 2 ở chế độ Thread Mode (clone)*

2. Fork Mode (`make fork`)

Khi chạy:

```
make fork
./mypool
```

hệ thống chuyển sang mô hình worker-process độc lập. Điều này dẫn tới sự thay đổi hoàn toàn cơ chế chia sẻ dữ liệu và đồng bộ hoá: tất cả thông tin liên quan worker được đặt trong POSIX Shared Memory, và việc đánh thức tiến trình dùng lệnh `kill()`.

Kết quả output :

```
PS C:\Users\ASUS\Desktop\LAB 251\LAB4_Scheduling\lab4-student\Problem2> wsl
vophamk23@VoPham:/mnt/c/Users/ASUS/Desktop/LAB 251/LAB4_Scheduling/lab4-student/Problem2$ make clean
rm -f mypool main.o bktpool.o bktask.o bkwrk.o bkshm.o
vophamk23@VoPham:/mnt/c/Users/ASUS/Desktop/LAB 251/LAB4_Scheduling/lab4-student/Problem2$ make fork
rm -f mypool main.o bktpool.o bktask.o bkwrk.o bkshm.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -UWORK_THREAD -c main.c -o main.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -UWORK_THREAD -c bktpool.c -o bktpool.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -UWORK_THREAD -c bktask.c -o bktask.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -UWORK_THREAD -c bkwrk.c -o bkwrk.o
gcc -pthread -D_GNU_SOURCE -DINFO -w -UWORK_THREAD -c bkshm.c -o bkshm.o
gcc -pthread -lrt -o mypool main.o bktpool.o bktask.o bkwrk.o bkshm.o
vophamk23@VoPham:/mnt/c/Users/ASUS/Desktop/LAB 251/LAB4_Scheduling/lab4-student/Problem2$ ./mypool
bkwrk_create_worker got worker PID 1443
bkwrk_create_worker got worker PID 1444
bkwrk_create_worker got worker PID 1445
bkwrk_create_worker got worker PID 1446
bkwrk_create_worker got worker PID 1447
bkwrk_create_worker got worker PID 1448
bkwrk_create_worker got worker PID 1449
bkwrk_create_worker got worker PID 1450
bkwrk_create_worker got worker PID 1451
bkwrk_create_worker got worker PID 1452
Assign tsk 0 wrk 0
worker wake 0 up
Task func - Hello from 1
Assign tsk 1 wrk 0 >>>>>>>>>> Activate asynchronously
Assign tsk 2 wrk 1 >>>>>>>>>> Activate asynchronously
worker wake 0 up
Task func - Hello from 2
worker wake 1 up
Task func - Hello from 5
```

Hình 3: *Kết quả thực thi Problem 2 ở chế độ Fork Mode (worker-process + POSIX shared memory)*

1. Khởi tạo worker-process bằng `fork()`

Output:

```
bkwrk_create_worker got worker PID 1443
bkwrk_create_worker got worker PID 1444
bkwrk_create_worker got worker PID 1445
bkwrk_create_worker got worker PID 1446
bkwrk_create_worker got worker PID 1447
bkwrk_create_worker got worker PID 1448
bkwrk_create_worker got worker PID 1449
```

```
bkwrk_create_worker got worker PID 1450
bkwrk_create_worker got worker PID 1451
bkwrk_create_worker got worker PID 1452
```

Giải thích:

- Hàm `bkwrk_create_worker()` gọi `fork()` 10 lần để tạo 10 worker-process độc lập.
- Mỗi worker có PID riêng và chuyển ngay vào hàm `bkwrk_worker()`, sau đó block tại `sigwait()`.
- Vì tiến trình con không chia sẻ bộ nhớ với master, module `bkshm.c` tạo hai vùng shared memory:
  - **Busy array**: theo dõi trạng thái rảnh/bận.
  - **Worker array**: chứa `func`, `arg` và mã tác vụ.

Điều này xác nhận worker-process đã khởi tạo thành công.

2. Gán tác vụ đầu tiên bằng FIFO

Output:

```
Assign tsk 0 wrk 0
```

Ý nghĩa:

- Scheduler FIFO duyệt từ worker 0 và tìm worker rảnh đầu tiên.
- Master ghi trực tiếp vào shared memory:

```
1  wrkid_busy[0] = 1;
2  worker[0].func = tsk->func;
3  worker[0].arg  = tsk->arg;
```

Không có IPC khác; worker sẽ đọc dữ liệu này khi được đánh thức.

3. Đánh thức worker-process và thực thi tác vụ

Output:

```
worker wake 0 up
Task func - Hello from 1
```

Giải thích pipeline:

1. Worker 0 đang chờ tại `sigwait()`.
2. Master gọi:

```
1  kill(pid_of_worker_0, SIGUSR1);
```

3. Worker 0 thoát khỏi `sigwait()`, đọc `func` và `arg` từ shared memory.
4. Hàm tác vụ được thực thi và in ra thông báo.
5. Worker đặt lại trạng thái về IDLE và quay về `sigwait()`.

4. Scheduling bất đồng bộ

Output:

```
Assign tsk 1 wrk 0 >>>>>>>>>> Activate asynchronously
Assign tsk 2 wrk 1 >>>>>>>>>> Activate asynchronously
```

Giải thích:

- Task 1: worker 0 đã rảnh trở lại nên FIFO tiếp tục gán task cho worker 0.
- Task 2: worker 1 là worker rảnh tiếp theo.
- Cả hai tác vụ đều được gán trong khi master không chờ worker hoàn tất nhiệm vụ.

Thông tin ghi vào shared memory:

```
worker[0].arg = 2;
worker[1].arg = 5;
```

5. Worker-process xử lý các tác vụ tiếp theo

Output:

```
worker wake 0 up
Task func - Hello from 2
worker wake 1 up
Task func - Hello from 5
```

Pipeline hoạt động:

- Master gửi tín hiệu `SIGUSR1` tới từng PID tương ứng.
- Worker 0 đọc tham số = 2 và thực thi hàm.
- Worker 1 đọc tham số = 5 và thực thi hàm.

Mỗi worker sau khi hoàn thành đều đặt `busy = 0` và quay về `sigwait()`.

Tổng quát

Fork-mode triển khai đầy đủ pipeline xử lý bằng đa tiến trình:

1. Tạo worker-process bằng `fork()`.
2. Khởi tạo shared memory để chia sẻ dữ liệu giữa master và worker.
3. FIFO chọn worker rảnh đầu tiên.
4. Master gán task bằng thao tác ghi vào shared memory.
5. Master gửi tín hiệu đánh thức worker.
6. Worker đọc task, thực thi và trả về trạng thái IDLE.
7. Master tiếp tục lập lịch các tác vụ tiếp theo.

Mỗi dòng output đều phản ánh chính xác luồng xử lý được lập trình trong mã nguồn.

3 PROBLEM 3 - Xây dựng mô hình Fork–Join Framework

3.1 Mở đầu

Mô hình **Fork–Join** là một kiến trúc song song truyền thống, được áp dụng trong nhiều hệ thống đa luồng và thư viện tính toán hiệu năng cao như Java Fork/Join Pool, Intel TBB, OpenMP Task Parallelism và các hệ thống dựa trên POSIX Threads. Nguyên lý hoạt động của mô hình gồm ba pha:

- **Fork**: phân rã bài toán lớn thành tập hợp các tác vụ nhỏ độc lập.
- **Parallel Execution**: phân phối và thực thi tác vụ đồng thời trên nhiều worker.
- **Join**: đồng bộ, chờ tất cả tác vụ hoàn tất và hợp nhất kết quả.

Trong **Problem 3**, nhiệm vụ là xây dựng một Fork–Join Framework tối giản nhưng hoàn chỉnh, hoạt động ổn định trong môi trường đa luồng POSIX và đáp ứng các yêu cầu:

- Phân chia tác vụ thành các đơn vị độc lập;
- Thực thi song song trên nhóm worker thread bền vững;
- Đồng bộ chính xác quá trình hoàn tất bằng condition variable;
- Thu kết quả của từng tác vụ theo ID.

Bộ mã nguồn `bkfj.c / bkfj.h` sử dụng đầy đủ các cơ chế đồng bộ POSIX: mutex, semaphore, condition variable và memory barrier, đảm bảo không xảy ra busy-wait hay race condition.

3.2 Mục tiêu và Yêu cầu Thiết kế

Framework phải tận dụng tối đa khả năng song song của hệ thống thông qua một nhóm worker thread hoạt động độc lập. Mỗi worker phải:

3.2.1. Song song thực sự

Framework phải tận dụng tối đa khả năng song song của CPU:

- mỗi worker thread chạy độc lập;
- worker ngủ bằng `sem_wait()` và chỉ thức khi có task mới;
- các task được xử lý hoàn toàn song song.

3.2.2. Đồng bộ và nhất quán tuyệt đối

Framework phải ngăn chặn mọi race condition trên tài nguyên chia sẻ. Do đó, toàn bộ truy cập quan trọng đều được bảo vệ:

- `queue_mutex`: push/pop trong Task Queue,
- `completed_mutex`: Completed List,
- `task_id_mutex`: cấp phát ID an toàn,
- `count_mutex + all_done_cond`: xử lý join.

3.2.3. Tối ưu hiệu năng và tái sử dụng

Một trong các yêu cầu thiết kế quan trọng:

- Worker thread được tạo một lần, chạy liên tục và tái sử dụng cho tất cả chu kỳ fork–join.
- Không busy-wait, không dùng sleep thủ công.
- Join không tốn CPU, master block cho đến khi tất cả task hoàn tất.
- Hỗ trợ nhiều chu kỳ fork–join liên tiếp (được kiểm chứng bằng `test_multiple_cycles`).

3.2.4. API rõ ràng, dễ dùng

API	Chức năng
fj_pool_init(N)	Tạo pool gồm N worker thread
fj_fork(func, arg)	Tạo task và đưa vào Task Queue
fj_join()	Đồng bộ, đợi tất cả tác vụ hoàn tất
fj_get_result(task_id)	Lấy kết quả của task theo ID
fj_pool_destroy()	Giải phóng toàn bộ tài nguyên

3.3 Kiến trúc Tổng thể của Hệ thống

Kiến trúc Fork–Join Framework được tổ chức thành ba thành phần trung tâm, hoạt động phối hợp nhịp nhàng như một pipeline song song hoàn chỉnh.

3.3.1. Worker Pool

- Gồm tối đa `MAX_FJ_WORKERS` worker thread.
- Mỗi worker chạy vòng lặp vô hạn và ngủ trên semaphore.
- Worker chỉ dừng khi framework shutdown.

3.3.2. Task Queue (FIFO)

- Hàng đợi tác vụ dùng danh sách liên kết đơn.
- Đảm bảo tính FIFO đúng nghĩa.
- Truy cập được bảo vệ bởi `queue_mutex`.

3.3.3. Completed List

- Lưu từng task đã hoàn tất cùng với kết quả trả về `(task->result)`.
- Bảo vệ bằng `completed_mutex`.
- Hỗ trợ duyệt hoặc truy xuất theo ID.

Pipeline tổng thể:

$$\text{Master} \xrightarrow{\text{Fork}} \text{Task Queue} \xrightarrow{\text{Worker}} \text{Completed List} \xrightarrow{\text{Join}} \text{Master}$$

3.4 Worker Pool và Cơ chế Hoạt động

3.4.1. Khởi tạo Worker

Trong `fj_pool_init()`:

- Cấp phát `fj_pool_t`;
- Khởi tạo mutex, semaphore, condition variable;
- Tạo N worker thread:

```
pthread_create(&pool->workers[i], NULL, fj_worker_thread, pool);
```

Worker sau đó lập tức vào `sem_wait()`.

3.4.2. Vòng đời của Worker Thread

Mỗi worker thực hiện chu trình:

1. **Chờ nhiệm vụ**: sem_wait();
2. **Lấy task FIFO**: bảo vệ bằng `queue_mutex`;
3. **Thực thi task**:

```
result = task->func(task->arg);
```

4. **Lưu kết quả**: chuyển vào Completed List;
5. **Cập nhật active_tasks**: khi còn 0 → báo join;
6. **Quay về `sem_wait()` chờ nhiệm vụ mới**.

3.5 Hàng đợi Tác vụ (Task Queue)

Trong `fj_fork()`:

1. Cấp phát `fj_task_t`;
2. Gán task ID bằng hàm nguyên tử `get_next_task_id()`;
3. Thêm task vào đuôi queue (`queue_mutex`);
4. Tăng `active_tasks`;
5. Đánh thức worker: `sem_post()`.

Master hoàn toàn không bị block.

3.6 Cơ chế Join và Đồng bộ Hoàn tất

Join được xây dựng đúng chuẩn POSIX bằng condition variable:

- `active_tasks` đếm số task đang thực thi.
- Worker giảm biến này khi hoàn tất.
- Khi về 0: worker phát sự kiện:

```
pthread_cond_broadcast(&pool->all_done_cond);
```

Trong `fj_join()`:

```
while (active_tasks > 0)
    pthread_cond_wait(&all_done_cond, &count_mutex);
```

3.7 Completed List và Thu kết quả

Sau khi worker hoàn tất:

- lưu `task->result`;
- chuyển task vào Completed List (`completed_mutex`).

API hỗ trợ:

- `fj_get_result(id)`;
- `fj_get_all_results()`;
- `fj_free_result(id)`.

Pipeline xử lý và pipeline thu kết quả được tách biệt rõ ràng.

3.8 Cleanup và Hủy tài nguyên

Hàm `fj_pool_destroy()` thực hiện:

1. Đặt `shutdown = 1`;
2. Đánh thức toàn bộ worker bằng `sem_post()`;
3. Join tất cả worker thread;
4. Giải phóng queue, completed list;
5. Hủy mutex, semaphore, condition variable;
6. Giải phóng pool.

Đảm bảo không rò rỉ bộ nhớ và worker kết thúc an toàn.

3.9 Quy trình thực thi của hệ thống (Execution Workflow)

Dựa trên mã kiểm thử `test_forkjoin.c`, quá trình vận hành đầy đủ của Fork–Join Framework được mô tả theo sáu bước dưới đây. Quy trình thể hiện rõ chu trình *fork* → *execute* → *join* → *collect* đặc trưng của mô hình Fork–Join.

Bước 1 - Khởi tạo Pool

```
fj_pool_t *pool = fj_pool_init(4);
```

Lệnh trên tạo ra một *pool* gồm 4 worker thread. Sau khi được tạo, mỗi worker lập tức đi vào trạng thái chờ nhiệm vụ bằng `sem_wait()`, đảm bảo không tiêu tốn CPU cho đến khi có tác vụ mới.

Bước 2 - Fork (Gửi tác vụ vào Task Queue)

Ví dụ trong hàm kiểm thử `test_parallel_square()`:

```
fj_fork(pool, square_task, (void*)(intptr_t)i);
```

Khi master gọi `fj_fork()`:

- `fj_task_t` mới được cấp phát;
- task được gán *task ID* duy nhất;
- task được đưa vào cuối hàng đợi FIFO;
- biến đếm `active_tasks` tăng lên;
- một worker đang ngủ được đánh thức bằng `sem_post()`.

Việc gửi tác vụ hoàn toàn không làm master bị chặn, đúng với tinh thần *producer/consumer* song song.

Bước 3 - Worker xử lý song song

Mỗi worker, khi được đánh thức, thực hiện quy trình:

1. Lấy task đầu tiên trong Task Queue (bảo vệ bởi `queue_mutex`).
2. Gọi hàm xử lý tương ứng, ví dụ:
   - `square_task()`,
   - `fib_task()`.
3. Lưu kết quả vào `task->result` và chuyển task vào Completed List.
4. Giảm `active_tasks`. Nếu số task còn lại bằng 0, worker phát tín hiệu đánh thức master:

```
pthread_cond_broadcast(&pool->all_done_cond);
```

Tất cả worker chạy hoàn toàn độc lập → tạo ra mức độ song song tối đa.

Bước 4 - Join (Đợi toàn bộ tác vụ hoàn tất)

```
fj_join(pool);
```

Khi join được gọi, master thread:

- đi vào trạng thái block trên `all_done_cond`;
- chỉ được đánh thức khi `active_tasks == 0`.

Join không sử dụng busy-wait, đảm bảo hiệu năng tối ưu và CPU không bị lãng phí.

Bước 5 - Thu kết quả (tuỳ chọn)

Sau khi join hoàn tất, người dùng có thể truy xuất kết quả:

- theo *task ID*:

```
fj_get_result(id);
```

- hoặc duyệt toàn bộ Completed List để xử lý đồng loạt.

Thiết kế tách riêng pipeline xử lý và pipeline thu kết quả giúp hệ thống dễ mở rộng.

Bước 6 - Destroy (Giải phóng tài nguyên)

```
fj_pool_destroy(pool);
```

Hàm hủy pool thực hiện:

- đặt cờ `shutdown = 1`;
- đánh thức tất cả worker còn đang chờ;
- join từng worker thread;
- giải phóng Task Queue và Completed List;
- hủy mutex, semaphore và condition variable.

Toàn bộ tài nguyên được giải phóng an toàn, không rò rỉ và không để lại trạng thái lock.

3.10 Output đầu ra

Chương trình được biên dịch bằng lệnh `make` để tạo ra tệp thực thi `test_fj`. Sau đó, quá trình kiểm thử được tiến hành bằng cách chạy:

```
./test_fj
```

Chương trình lần lượt thực hiện ba nhóm kiểm thử: (1) tính bình phương song song, (2) tính dãy Fibonacci song song, và (3) vận hành nhiều chu kỳ Fork–Join liên tiếp.

Kết quả output :

```
PS C:\Users\ASUS\Desktop\LAB 251\LAB4_Scheduling\lab4-student\Problem3> wsl
vophamk23@VoPham:/mnt/c/Users/ASUS/Desktop/LAB 251/LAB4_Scheduling/lab4-student/Problem3$ make clean
rm -f test_fj bkfj.o test_forkjoin.o
vophamk23@VoPham:/mnt/c/Users/ASUS/Desktop/LAB 251/LAB4_Scheduling/lab4-student/Problem3$ make
gcc -pthread -Wall -Wextra -D_GNU_SOURCE -c bkfj.c -o bkfj.o
gcc -pthread -Wall -Wextra -D_GNU_SOURCE -c test_forkjoin.c -o test_forkjoin.o
gcc -pthread -o test_fj bkfj.o test_forkjoin.o
vophamk23@VoPham:/mnt/c/Users/ASUS/Desktop/LAB 251/LAB4_Scheduling/lab4-student/Problem3$ ./test_fj

========================================
  Fork-Join Framework Demonstration
========================================

=== TEST 1: Parallel Square Computation ===
Pool initialized with 4 workers

FORK: Submitting 10 tasks...

JOIN: Waiting for all tasks to complete...
  [Worker] 1^2 = 1
  [Worker] 3^2 = 9
  [Worker] 2^2 = 4
  [Worker] 4^2 = 16
  [Worker] 5^2 = 25
  [Worker] 6^2 = 36
  [Worker] 7^2 = 49
  [Worker] 8^2 = 64
  [Worker] 10^2 = 100
  [Worker] 9^2 = 81
All tasks completed in 0.001 seconds
=== TEST 2: Fibonacci Computation ===
Pool initialized with 4 workers

FORK: Computing fib(0) to fib(15)...
  [Worker] fib(0) = 0
  [Worker] fib(1) = 1
  [Worker] fib(2) = 1
  [Worker] fib(3) = 2
  [Worker] fib(4) = 3
  [Worker] fib(5) = 5
  [Worker] fib(6) = 8
  [Worker] fib(8) = 21
  [Worker] fib(7) = 13
  [Worker] fib(11) = 89
  [Worker] fib(12) = 144
  [Worker] fib(13) = 233
  [Worker] fib(14) = 377
  [Worker] fib(15) = 610
  [Worker] fib(9) = 34
  [Worker] fib(10) = 55

JOIN: Waiting for all tasks to complete...
All tasks completed in 0.000 seconds
=== TEST 3: Multiple Fork-Join Cycles ===
Pool initialized with 4 workers

--- Cycle 1 ---
FORK: Submitting 5 tasks (1^2 to 5^2)
JOIN: Waiting...
  [Worker] 1^2 = 1
  [Worker] 3^2 = 9
  [Worker] 4^2 = 16
  [Worker] 2^2 = 4
  [Worker] 5^2 = 25
Cycle 1 completed

--- Cycle 2 ---
FORK: Submitting 5 tasks (6^2 to 10^2)
JOIN: Waiting...
  [Worker] 6^2 = 36
  [Worker] 8^2 = 64
  [Worker] 7^2 = 49
  [Worker] 9^2 = 81
  [Worker] 10^2 = 100
Cycle 2 completed

========================================
  All tests completed successfully
========================================
```

Hình 4: *Kết quả chạy chương trình minh họa Fork–Join Framework*

* Phân tích Output - Cơ chế Fork–Join thể hiện qua kết quả chạy

Kết quả thực thi chương trình `test_fj` cho thấy rõ vòng đời hoạt động của một Fork–Join Framework: *fork tác vụ → worker thực thi song song → join đợi toàn bộ hoàn tất → thu kết quả → lặp lại nhiều chu kỳ*. Ba nhóm kiểm thử dưới đây minh họa trọn vẹn hành vi của hệ thống.

Test 1 - Parallel Square Computation

1. Giai đoạn khởi tạo

```
Pool initialized with 4 workers
```

Pool gồm 4 worker thread được tạo và ngay lập tức chuyển vào trạng thái chờ. Đây là giai đoạn chuẩn bị cho chu trình fork–join.

2. Giai đoạn Fork

```
FORK: Submitting 10 tasks...
```

Master gửi đồng thời 10 tác vụ vào hệ thống. Tại thời điểm này, toàn bộ tác vụ chỉ được đưa vào hàng đợi, chưa có xử lý thực tế.

3. Giai đoạn Join

```
JOIN: Waiting for all tasks to complete...
```

Master block trong lệnh `join`, chờ tất cả worker báo hoàn thành.

4. Worker thực thi song song

```
[Worker] 1^2 = 1
[Worker] 3^2 = 9
[Worker] 2^2 = 4
[Worker] 4^2 = 16
[Worker] 5^2 = 25
[Worker] 6^2 = 36
[Worker] 7^2 = 49
[Worker] 8^2 = 64
[Worker] 10^2 = 100
[Worker] 9^2 = 81
```

Các kết quả được in ra:

- Không theo thứ tự tuần tự (1–10), mà theo đúng thời điểm worker hoàn thành tác vụ.
- Chứng minh tính **song song thực sự**, mỗi worker lấy và xử lý tác vụ độc lập.

5. Hoàn tất chu kỳ

```
All tasks completed in 0.001 seconds
```

Join chỉ kết thúc khi toàn bộ 10 tác vụ đã được xử lý và báo hoàn thành. Điều này xác thực cơ chế đồng bộ hoạt động chính xác.

3.11 Test 2 - Fibonacci Computation

1. Khởi tạo lại Pool

```
Pool initialized with 4 workers
```

Một pool mới được tạo để đảm bảo quá trình kiểm thử độc lập.

2. Fork các tác vụ Fibonacci

```
FORK: Computing fib(0) to fib(15)...
```

Tổng cộng 16 tác vụ `fib()` được đẩy vào hàng đợi theo cơ chế FIFO.

3. Worker xử lý song song

Ví dụ các dòng tiêu biểu:

```
[Worker] fib(0) = 0
[Worker] fib(1) = 1
[Worker] fib(2) = 1
[Worker] fib(3) = 2
[Worker] fib(4) = 3
[Worker] fib(5) = 5
[Worker] fib(6) = 8
[Worker] fib(8) = 21
[Worker] fib(7) = 13
[Worker] fib(11) = 89
[Worker] fib(12) = 144
[Worker] fib(13) = 233
[Worker] fib(14) = 377
[Worker] fib(15) = 610
[Worker] fib(9) = 34
[Worker] fib(10) = 55
```

Nhận xét:

- Kết quả không theo thứ tự 0–15 vì mỗi tác vụ Fibonacci có độ phức tạp khác nhau.
- Worker nào rảnh trước sẽ xử lý trước, tạo ra thứ tự hoàn thành phi tuyến tính.
- Đây là hành vi đặc trưng của mô hình Fork–Join.

4. Hoàn tất Join

```
All tasks completed in 0.000 seconds
```

Join kết thúc khi toàn bộ 16 tác vụ đã hoàn thành, xác nhận tính đúng đắn của cơ chế đồng bộ.

3.12 Test 3 - Multiple Fork–Join Cycles

Kiểm thử này đánh giá khả năng tái sử dụng worker pool qua nhiều chu kỳ liên tiếp.

1. Chu kỳ 1 – Fork

```
FORK: Submitting 5 tasks (1^2 to 5^2)
```

2. Chu kỳ 1 – Worker xử lý

```
[Worker] 1^2 = 1
[Worker] 3^2 = 9
[Worker] 4^2 = 16
[Worker] 2^2 = 4
[Worker] 5^2 = 25
```

Thứ tự hoàn tất không trùng với thứ tự gửi vào queue, cho thấy worker xử lý hoàn toàn độc lập.

3. Chu kỳ 1 – Join

```
Cycle 1 completed
```

4. Chu kỳ 2 – Fork

```
FORK: Submitting 5 tasks (6^2 to 10^2)
```

5. Chu kỳ 2 – Worker xử lý

```
[Worker] 6^2 = 36
[Worker] 8^2 = 64
[Worker] 7^2 = 49
[Worker] 9^2 = 81
[Worker] 10^2 = 100
```

Worker từ chu kỳ 1 tiếp tục được sử dụng, chứng minh cơ chế **tái sử dụng pool** hoạt động ổn định.

6. Chu kỳ 2 – Join

```
Cycle 2 completed
```

Hai chu kỳ liên tiếp hoàn thành đầy đủ, xác nhận:

- Worker không bị treo.
- Không có rò rỉ tài nguyên.
- Pipeline Fork–Join vận hành đúng trong môi trường lặp.

Tài liệu