Simple Operating System

Báo cáo bài tập lớn Hệ điều hành

Nhóm L08_SysCallers

GVHD: ThS. Hoàng Lê Hải Thanh

TRƯỜNG ĐẠI HỌC BÁCH KHOA
ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

Ngày 21 tháng 5 năm 2025

Danh sách thành viên và nhiệm vụ

STT	Họ và tên	MSSV	Nhiệm vụ
1	Dương Gia Bảo	2310207	- Hiện thực Scheduler - Trả lời câu hỏi
2	Nguyễn Đình Khôi	2311681	- Hiện thực System Call - Trả lời câu hỏi
3	Bùi Nhật Quí	2312864	- Hiện thực Memory Management - Trả lời câu hỏi
4	Trần Hoàng Uyên	2313854	- Hiện thực Memory Management - Trả lời câu hỏi

- Mục tiêu bài tập lớn: xây dựng OS mô phỏng với ba thành phần chính:
  - Bộ lập lịch (Scheduler)
  - Quản lý bộ nhớ ảo (Memory Management)
  - Lời gọi hệ thống (System Call)
- Tài nguyên mô phỏng: CPU và RAM ảo
- Cách chạy mô phỏng:
  - Biên dịch: `make all`
  - Thực thi: `./os [configure_file]`

Tổng quan

- Trong bài tập lớn này, Scheduler là thành phần chịu trách nhiệm quản lý tiến trình và phân phối CPU. Mô hình sử dụng thuật toán "Multi-Level Queue" (MLQ)
- Hệ thống chạy theo nguyên tắc round-robin trong mỗi hàng đợi và cấp CPU cho tiến trình theo mức ưu tiên từ cao đến thấp (ưu tiên cao có giá trị nhỏ hơn). Mỗi hàng đợi chỉ có một số lượng slot cố định để sử dụng CPU, sau đó nhường quyền lại cho hàng đợi khác.

Thao tác hàng đợi (queue.c)

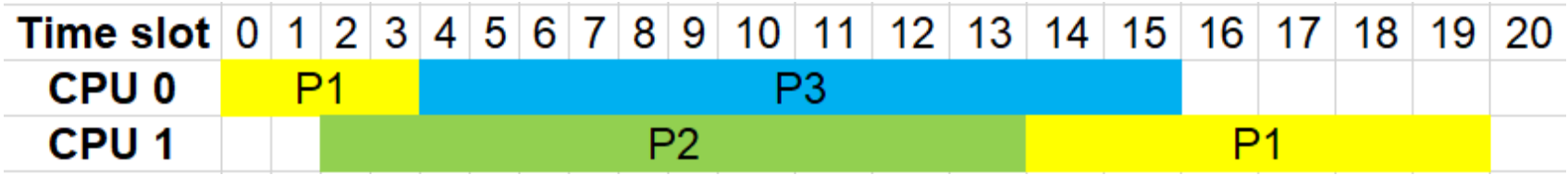
- Hàm `enqueue`: thêm `pcb_t` vào cuối queue.
- Hàm `dequeue`: lấy tiến trình có độ ưu tiên cao nhất:
  - Duyệt toàn bộ `q->proc[]` tìm chỉ số có `priority` nhỏ nhất.
  - Di chuyển các phần tử sau lên trước, giảm `q->size`.
- Đơn giản, phù hợp cho MLQ với kích thước queue giới hạn.

Lập lịch MLQ (sched.c)

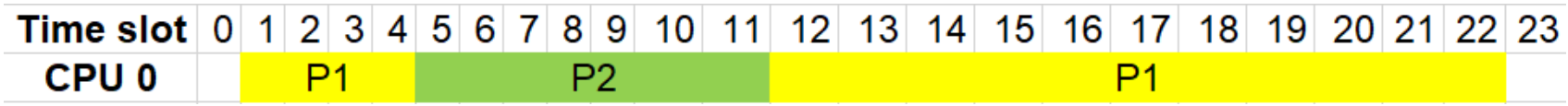
- `get_mlq_proc()`:
  - Duyệt từ ưu tiên cao (0) đến thấp (MAX_PRIO-1).
  - Nếu `slot[prio]` > 0 và queue không rỗng, gọi dequeue.
  - Giảm `slot[prio]` và trả về `pcb_t *`.
  - Reset về trạng thái ban đầu nếu toàn bộ slot của các tiến trình đều là 0.
- `add_proc()` vs `put_proc()`:
  - `add_proc()`: gọi một lần khi load tiến trình mới.
  - `put_proc()`: gọi sau mỗi time slice, enqueue vào running_list.

Phân tích output

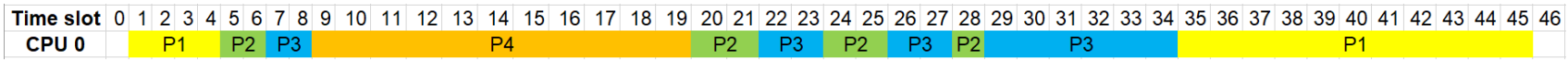
Sơ đồ Gantt cho các output:



Hình 1: Sơ đồ Gantt khi chạy input sched



Hình 2: Sơ đồ Gantt khi chạy input sched_0



Hình 3: Sơ đồ Gantt khi chạy input sched_1

Trả lời câu hỏi

Question: What are the advantages of using the scheduling algorithm described in this assignment compared to other scheduling algorithms you have learned?

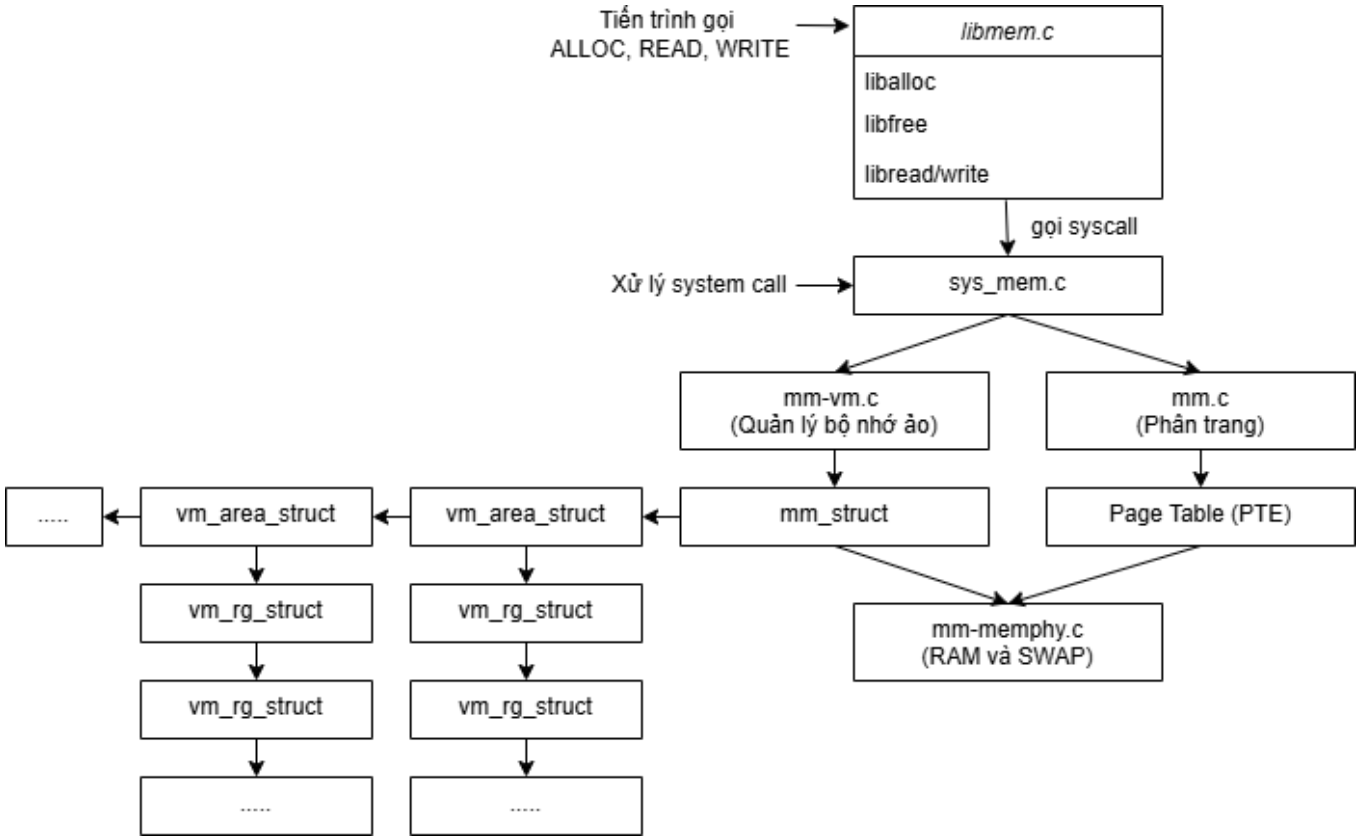
Ưu điểm của thuật toán MLQ so với các thuật toán khác:

- Mỗi mức ưu tiên được gán cho một hàng đợi riêng biệt, giúp dễ dàng quản lý và chọn tiến trình có độ ưu tiên cao hơn để thực thi trước.
- Hệ thống có thể vận hành trên nhiều CPU một cách hiệu quả, mỗi CPU có thể chọn tiến trình từ hàng đợi theo nguyên tắc vòng tròn (round-robin), tối ưu hiệu suất xử lý.
- Các tiến trình trong cùng một hàng đợi được xử lý theo vòng tròn, giúp đảm bảo tính công bằng và tránh tình trạng "starvation" trong một mức ưu tiên.

Tuy nhiên, do không có cơ chế phản hồi (feedback) như Multilevel Feedback Queue, thuật toán trong bài không cho phép tiến trình thay đổi mức ưu tiên trong suốt thời gian thực thi, dễ dẫn đến "starvation" ở tiến trình có mức ưu tiên thấp.

Tổng quan

Trong một hệ điều hành, việc quản lý bộ nhớ là một trong những nhiệm vụ cốt lõi và phức tạp nhất.



Tổng quan

Các phân lớp quản lý bộ nhớ từ trên xuống có vai trò như sau:

- Thư viện người dùng (`libmem.c`): Cấp phát, ghi/đọc (alloc, write/read) bộ nhớ cho tiến trình. Gọi syscall để thực thi.
- Xử lý System Call (`sys_mem.c`): Nhận và xử lý syscall từ tiến trình, gọi vào các phân lớp mm-vm và mm để thực thi quản lý bộ nhớ.
- Bộ nhớ ảo (`mm-vm.c`): Quản lý vùng nhớ của tiến trình.
- Phân trang (`mm.c`): Thực hiện ánh xạ page – frame qua bảng trang, cấp phát khung RAM, hoán trang.
- Bộ nhớ vật lý (`mm-memphy.c`): thực hiện thao tác thô trên bộ nhớ vật lý, mô phỏng RAM, SWAP: cấp phát frame, đọc/ghi các ô nhớ thực.

Hiện thực code

Memory Management — Thiết kế libmem và Syscall:

- Libmem: `alloc`, `free`, `read/write`, `pg_getpage`, `pg_getval` và `pg_setval`.
- Syscall mã 17: `SYSMEM_INC_OP`, `SWP_OP`, `IO_READ/WRITE`.

Memory Management — Phân lớp ảo và Phân trang:

- mm-vm: quản lý vùng ảo (`vm_area`, `free list`), mở rộng sbrk.
- mm: ánh xạ page sang frame, FIFO victim, swap in/out.
- mm-memphy: quản lý khung vật lý, đọc/ghi dữ liệu.

Thư viện người dùng (libmem.c)

- Hàm `__alloc`:
  - Tìm vùng trống trong VM area (get_free_vmrg_area).
  - Nếu không đủ, gọi syscall SYSMEM_INC_OP để mở rộng.
  - Cập nhật symrgtbl và trả về địa chỉ ảo.
- Hàm `__free`: đánh dấu region thành trống, đưa vào freerg_list.
- Chức năng truy xuất:
  - `pg_getpage`: đảm bảo page có mặt trong RAM, xử lý swap.
  - `pg_getval`/`pg_setval`: đọc/ghi dữ liệu qua syscall hoặc MEMPHY.

Xử lý syscall (sys_mem.c)

- Hàm `__sys_memmap`: trap handler cho opcode 17.
- Định nghĩa các thao tác:
  - SYSMEM_INC_OP: inc_vma_limit
  - SYSMEM_SWP_OP: __mm_swap_page (swap out/in)
  - SYSMEM_IO_READ/WRITE: MEMPHY_read/write
- Vai trò: phân phối yêu cầu từ libmem đến các module mm-vm, mm, mm-memphy an toàn.

Quản lý bộ nhớ ảo (mm-vm.c)

- Cấu trúc `vm_area_struct`: quản lý vùng địa chỉ, danh sách vùng trống.
- Hàm `get_vm_area_node_at_brk`: tạo vm_rg_struct ở sbrk hiện tại.
- Hàm `validate_overlap_vm_area`: kiểm tra không trùng lặp vùng mới.
- Hàm `inc_vma_limit`: mở rộng vùng, gọi vm_map_ram, cập nhật freerg_list.

Phân trang (mm.c)

- Cơ chế phân trang hai lớp:
  - `init_mm`: khởi tạo pgd, VMA, bảng free region.
  - `alloc_pages_range`/`vmap_page_range`: cấp phát frame vật lý, ánh xạ page->frame.
- Quản lý PTE:
  - `init_pte` để set flag present, valid, frame number.
  - FIFO queue: enlist_pgn_node cho chiến lược thay thế.

Bộ nhớ vật lý (mm-memphy.c)

- Mô phỏng RAM và SWAP:
  - Quản lý khung trống: `MEMPHY_get_freefp`.
  - Đọc ghi byte: `MEMPHY_read/write`.
- Hàm `MEMPHY_dump`: in nội dung storage để debug.
- Tính toàn vẹn dữ liệu: đảm bảo swap và RAM đồng bộ.

Phân tích output

Do đầu ra của các tập tin kiểm thử khá dài, để đảm bảo tính súc tích cho báo cáo, nhóm chỉ trình bày chi tiết output của ba testcase tiêu biểu: `os_0_mlq_paging`, `os_1_mlq_paging_small_1K` và `os_1_singleCPU_mlq`. Các output còn lại sẽ được nhóm chuẩn bị sẵn, sẵn sàng trình bày theo yêu cầu của giảng viên trong buổi thuyết trình hoặc có thể kiểm tra thông qua mã nguồn (thư mục `team_output`) nhóm đã nộp.

Trả lời câu hỏi

Question: In this simple OS, we implement a design of multiple memory segments or memory areas in source code declaration. What is the advantage of the proposed design of multiple segments?

- **Phân chia logic**
  - Code, Data, Stack riêng biệt
  - Hỗ trợ OOP: mỗi lớp/đối tượng một phân đoạn
- **Truy cập hiệu quả**
  - Dùng cặp (segment, offset) để đọc/ghi trực tiếp
- **Đảm bảo cô lập**
  - Mỗi tiến trình/người dùng không gian riêng, tăng an toàn
- **Mở rộng địa chỉ phần cứng**
  - X86 real mode: 16-bit segment + 16-bit offset $\rightarrow$ 1 MB
  - Khi vượt 64 KB, cấp thêm phân đoạn mới mà không ảnh hưởng
- **Ưu điểm so với phân trang**
  - Ít phân mảnh nội bộ, kích thước vừa khít yêu cầu
  - Bảng phân đoạn đơn giản, giảm overhead quản lý
  - Dễ tái định vị từng phân đoạn độc lập

Trả lời câu hỏi

Question: What will happen if we divide the address to more than 2 levels in the paging memory management system?

- **Ưu điểm**
  - **Giảm chi phí bộ nhớ:** Bảng trang nhỏ hơn, chỉ tạo khi cần → tiết kiệm RAM cho không gian ảo lớn.
  - **Cấp phát bảng trang linh hoạt:** Tạo bảng trang động theo nhu cầu chương trình, tránh lãng phí.
  - **Sử dụng hiệu quả bộ nhớ phân mảnh:** Các bảng nhỏ rải rác tận dụng vùng trống tốt hơn so với bảng lớn.
  - **Truy cập bảng trang nhanh hơn:** Bảng nhỏ hơn → tra cứu nhanh, giảm thời gian tìm kiếm.
- **Nhược điểm**
  - **Tăng thời gian dịch địa chỉ:** Mỗi cấp bảng trang gây thêm lần truy cập bộ nhớ → độ trễ tăng.
  - **Cài đặt và quản lý phức tạp:** Cơ chế quản lý bảng và xử lý lỗi nhiều cấp khó thiết kế, bảo trì.

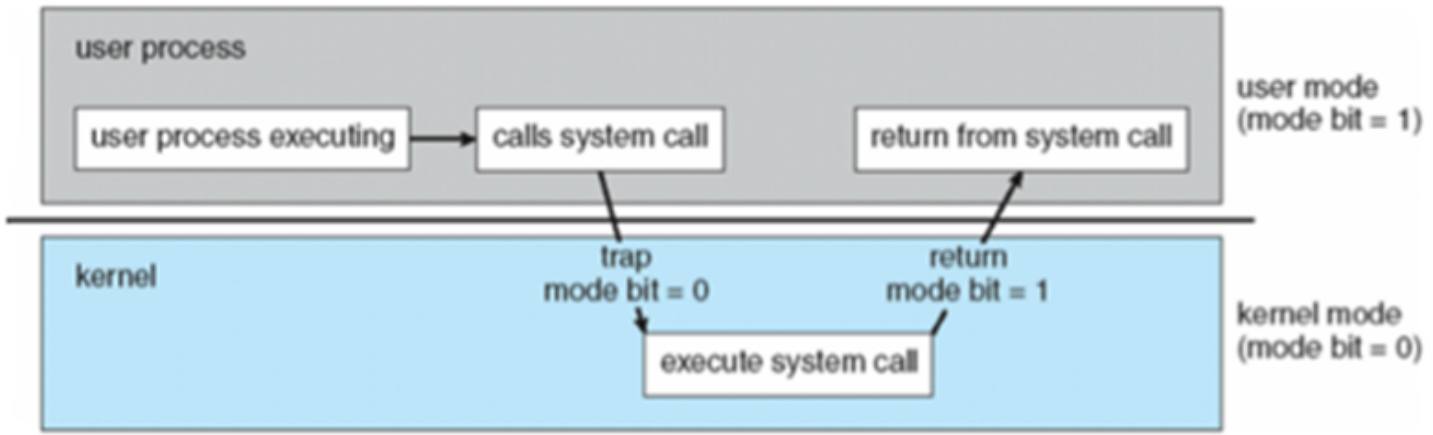
Trả lời câu hỏi

Question: What are the advantages and disadvantages of segmentation with paging?

- **Ưu điểm**
  - **Linh hoạt trong quản lý bộ nhớ:** Phân đoạn logic + cấp phát trang vừa khít → tối ưu kích thước.
  - **Bảo vệ và chia sẻ tốt hơn:** Quyền truy cập theo đoạn, code có thể chia sẻ giữa tiến trình.
  - **Mở rộng không gian địa chỉ:** Trang trên RAM, trang còn lại trên đĩa → vượt giới hạn vật lý.
  - **Hỗ trợ bộ nhớ ảo:** Hoán đổi trang giữa RAM và đĩa giúp ứng dụng chạy mượt dù RAM hạn chế.
- **Nhược điểm**
  - **Phức tạp trong quản lý:** Kết hợp cả bảng phân đoạn và bảng trang → nhiều bước truy xuất.
  - **Tốn tài nguyên hơn:** Lưu cả hai loại bảng → chiếm thêm không gian bộ nhớ.
  - **Gây phân mảnh bộ nhớ:**
    - Ngoài: khoảng trống giữa các đoạn kích thước khác nhau.
    - Trong: trang cố định có thể không dùng hết.
  - **Ảnh hưởng hiệu suất:** Phải tra bảng đoạn rồi tra bảng trang → độ trễ tăng.

Tổng quan

System Call (lời gọi hệ thống) là một phương thức để các ứng dụng sử dụng các dịch vụ của hệ điều hành như truy cập và quản lý các tập tin (file management) hay tạo và quản lý các tiến trình hệ thống (process management). System Call rất cần thiết để hệ điều hành hoạt động ổn định vì chúng chuẩn hóa cách thức truy cập vào tài nguyên hệ thống của các ứng dụng máy tính.



Xử lý killall (sys_killall.c)

- Hàm `__sys_killall`:
  - Đọc tên tiến trình (proc_name) từ vùng nhớ người dùng bằng `libread`.
  - Tìm trong danh sách running_list và các MLQ queue.
  - Gọi `free_process_memory` để giải phóng toàn bộ region với `libfree`.
- Hàm `free_process_memory`: duyệt `symrgtbl`, gọi `libfree` cho mỗi region.
- Sử dụng `pthread_mutex_t syscall_lock` để đồng bộ khi thao tác trên queue.

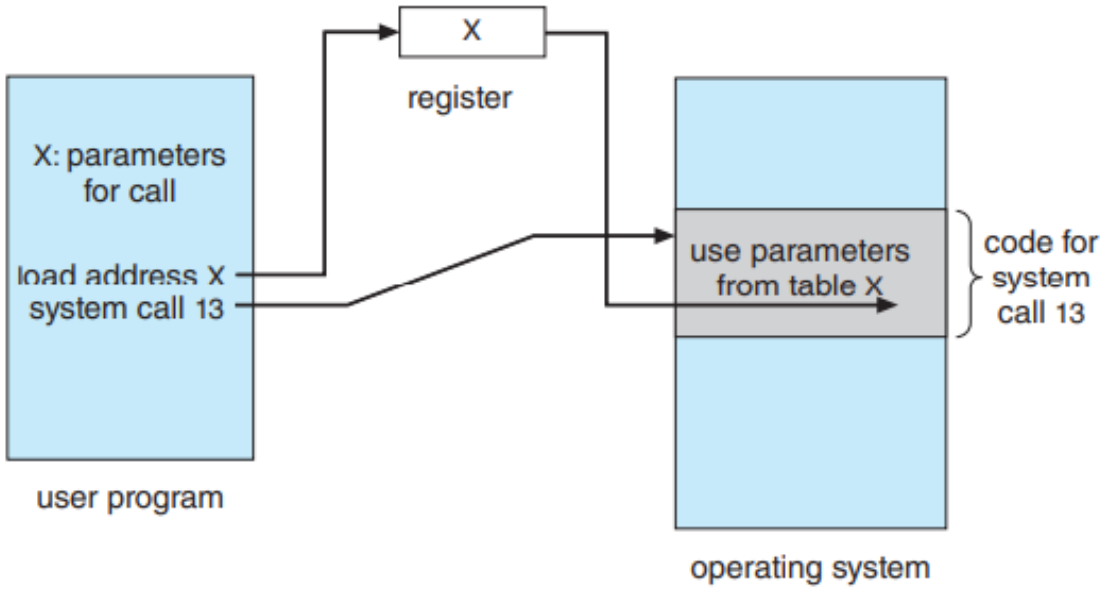
Tương tác giữa các module trong System Call

- Module quản lý bộ nhớ: System Call `killall` sử dụng hàm `libread` để đọc tên các tiến trình từ vùng nhớ được chỉ định bằng tham số `region_id`.
- Module điều khiển tiến trình của OS: System Call `killall` duyệt qua các tiến trình trong hệ thống và so sánh tên của các tiến trình đó. Nếu tìm thấy các tiến trình khớp tên thì System Call `killall` sẽ kết thúc tiến trình và giải phóng bộ nhớ.
- Module quản lý hàng đợi: System Call `killall` sẽ duyệt qua 2 hàng đợi `running_list` và `mlq_running_list` để giải phóng các tiến trình khớp tên đã cung cấp.

Trả lời câu hỏi

Question: What is the mechanism to pass a complex argument to a system call using the limited registers?

Thông thường, các tham số cần thiết để System Call hoạt động được truyền trực tiếp vào các thanh ghi. Tuy nhiên, trong trường hợp số lượng các tham số lớn hơn số lượng thanh ghi hiện có hoặc tham số truyền vào phức tạp thì phương pháp truyền này không khả thi. Do đó, để giải quyết trường hợp trên, các tham số cần truyền vào được lưu vào 1 block và truyền địa chỉ của block đó vào thanh ghi để System Call sử dụng.



- **Kiểm tra tính hợp lệ:** Kernel (memory manager) xác thực địa chỉ và quyền truy cập trước khi đọc block.
- **Quá trình thực thi:**
  - Kernel đọc thanh ghi, sao chép dữ liệu từ block vào vùng kernel.
  - Cập nhật PCB với trạng thái, tham chiếu tham số.
  - Nếu là IO blocking → chuyển tiến trình sang `BLOCKING`, sau khi hoàn tất đưa về Ready Queue.
- **Đối tượng tham số:** `struct student { int mssv; char name[100]; int age; }`
- **Handler:**
  - Khởi tạo `test_student`, gán giá trị.
  - Lưu địa chỉ vào `regs->a2`.
  - Thay đổi trực tiếp qua con trỏ → thể hiện qua thông báo tuổi & tên trước/sau.

```
1  ld_routine
2  Loaded a process at input/proc/sc3, PID: 1 PRIO:
       15
3  CPU 0: Dispatched process  1
4  Gia tri cua thanh ghi a2 la: 0x7fea259dcd20
5  Tuoi truoc khi thay doi: 20
6  Ten truoc khi thay doi: 10. He Dieu Hanh
7  Tuoi sau khi thay doi: 21
8  Ten sau khi thay doi: Tot nghiep xuat sac
9  CPU 0: Processed  1 has finished
10 CPU 0 stopped
```

- **Kết quả:**
  - In ra địa chỉ block và giá trị tham số trước/sau sửa đổi.
  - CPU dispatch → thực thi syscall → trả kết quả, CPU dừng khi hết tiến trình.
- **Lợi ích:** Hạn chế rào cản về số lượng & loại tham số, cho phép syscall xử lý struct/phức hợp.

Trả lời câu hỏi

Question: What happens if the syscall job implementation takes too long execution time?

- Khi gọi System Call:
  - Tiến trình tạm dừng, chờ System Call hoàn tất
  - Kernel lưu thông tin vào PCB
  - Chuyển trạng thái sang `BLOCKING`/`WAITING`
- Hậu quả khi System Call kéo dài:
  - Tiến trình "kẹt" ở kernel mode, không tiếp tục thực thi
  - Bị loại khỏi hàng đợi ready $\rightarrow$ không được lập lịch
  - CPU có thể nhàn rỗi do thiếu tiến trình sẵn sàng
  - System Call lâu còn ảnh hưởng đến tiến trình khác

Hiện thực Synchronization

Trong môi trường đa CPU, có khả năng nhiều tiến trình truy cập cùng một tài nguyên (như ready queue, bảng trang, bộ nhớ vật lý) cùng lúc. Để xử lý điều này, nhóm đã sử dụng `pthread_mutex_t` để khóa (lock) các vùng tài nguyên chia sẻ, đảm bảo rằng tại một thời điểm chỉ một tiến trình được phép truy cập vào vùng đó.
Các vùng sử dụng lock gồm:

- `queue_lock`: bảo vệ `mlq_ready_queue` trong quá trình `enqueue/dequeue`.
- `mmvm_lock`: bảo vệ các thao tác cấp phát, hoán đổi bộ nhớ.
- `syscall_lock`: bảo vệ các thao tác trên danh sách tiến trình (`running_list`, `mlq_ready_queue`) trong các syscall như `killall`, tránh xung đột giữa các CPU khi đồng thời huỷ và truy xuất tiến trình.

Trả lời câu hỏi

Question: What happens if the synchronization is not handled in your Simple OS? Illustrate the problem of your simple OS (assignment outputs) by example if you have any.

- Nhiều CPU cùng truy cập chung:
  - Hàng đợi tiến trình (mlq_ready_queue)
  - Bảng trang và free-list bộ nhớ
- Cần mutex lock để:
  - Ngăn race condition
  - Đảm bảo tính nhất quán và đúng đắn của mô phỏng
- Thiếu đồng bộ $\rightarrow$ lỗi khó lường, kết quả sai lệch

- **Ready queue:**
  - Hai CPU cùng `dequeue()` nhịp song song
  - Cùng PCB được dispatch & finish hai lần → Gantt chart trùng lặp
- **Free-list bộ nhớ:**
  - Hai CPU cùng `__free()` hoặc `libfree()`
  - Chèn/gộp vùng nhớ chồng lấn, mất liên kết
- **Hoán đổi trang (`pg_getpage()`):**
  - Cùng chọn victim & swap-out/swap-in không khóa
  - Bits `present`/`swapped` trong page table không nhất quán → segmentation fault

Kết luận chung

- **Scheduler:** MLQ + round-robin cân bằng hiệu suất & công bằng.
- **Memory Management:** Phân trang ảo, ánh xạ virtual→physical, hoán trang giữa RAM và SWAP, đảm bảo không gian địa chỉ riêng & an toàn.
- **System Call:** Cơ chế syscall (killall, listsyscall, memmap) mô phỏng giao diện kernel.
- **Synchronization:** Dùng `pthread_mutex_t` khoá ready queue, page table, PCB trong đa CPU → ngăn race & deadlock.

Hướng phát triển

- **MLFQ Scheduler:** Thêm feedback tự điều chỉnh ưu tiên, tránh starvation, cải thiện độ phản hồi.
- **Swap & Cache:** Cơ chế swap priority, cache page để giảm chi phí hoán trang.

Tài liệu tham khảo I

[1] A. Silberschatz, J. L. Peterson, and P. B. Galvin, *Operating system concepts*.
Addison-Wesley Longman Publishing Co., Inc., 1991.

[2] N. Weizer and G. Oppenheimer, "Virtual memory management in a paging environment," in *Proceedings of the May 14-16, 1969, spring joint computer conference*, pp. 249–256, 1969.

[3] B. W. Kernighan and D. M. Ritchie, *The C programming language*.
prentice-Hall, 1988.

[4] R. C. Seacord, *Secure Coding in C and C++*.
Addison-Wesley, 2013.

Thank You