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

HỆ ĐIỀU HÀNH

BÁO CÁO BÀI TẬP LỚN

Hiện thực Hệ điều hành đơn giản

GVHD: Trương Tuấn Phát
SV: Nguyễn Trung Nguyên - 2011710
Hứa Hoàng Nhật - 2111918
Phạm Như Thuần - 2213349

TP. HỒ CHÍ MINH, THÁNG 12/2023

Mục lục

Tóm tắt

Trong quá trình thực hiện bài tập lớn "Hiện thực Hệ điều hành đơn giản", nhóm đã thành công trong việc xây dựng một hệ điều hành đơn giản. Nhờ hoàn thành bài tập lớn này, nhóm đã học được rất nhiều về cấu trúc đơn giản của một hệ điều hành, cũng như cách thức hoạt động của một hệ điều hành trong thực tiễn.

Trước khi bắt tay vào làm bài tập lớn, nhóm đã phải xem lại lý thuyết về bộ môn này, đồng thời củng cố kiến thức về quá trình (process), định thời (scheduling), bộ nhớ (memory)... vả cả ngôn ngữ lập trình C để hiện thực hệ điều hành đơn giản này.

Bắt tay vào làm, nhóm đã gặp khó khăn vì cho dù được gọi là đơn giản, hệ điều hành này vẫn rất phức tạp đối với sinh viên. Do là lý thuyết khác xa thực tiễn, ban đầu nhóm không biết bắt đầu như thế nào. Nhưng sau khi duyệt lại kiến thức và đặc tả bài tập lớn, nhóm đã có một phương hướng nhất định. Tuy có nhiều vấn đề như lỗi cú pháp do không quen với ngôn ngữ C, không nắm rõ được mã nguồn, và lỗi liên quan đến giải thuật, nhưng nhóm đã hoàn tất vượt qua được chướng ngại.

Hoàn thành xong bài tập lớn, nhóm đã đúc kết được rằng, hiện thực một hệ điều hành "đơn giản" lại không hề đơn giản. Để có thể hiện thực một hệ điều hành, cần kiến thức sâu rộng về hệ thống, phần mềm cũng như phần cứng, kỹ năng lập trình và thuật toán. Nhưng dù thế, qua bài tập này nhóm đã có thể hiểu hơn về hệ điều hành và ứng dụng kiến thức đã học vào thực tiễn.

Bảng thành viên nhóm

Họ tên	MSSV	Phân công	Tiến độ
Nguyễn Trung Nguyên	2011710	Hiện thực bộ nhớ, mô phỏng	0%
Hứa Hoàng Nhật	2111918	Hiện thực bộ nhớ, mô phỏng	0%
Phạm Như Thuần	2213349	Hiện thực định thời, viết báo cáo	100%

1 Giới thiệu đề tài

Hệ điều hành (tiếng Anh: Operating system - OS) là phần mềm hệ thống quản lý tài nguyên phần cứng máy tính, phần mềm và cung cấp các dịch vụ chung cho các chương trình máy tính. Chức năng chính của hệ điều hành là định thời CPU, kiểm soát và phân phối tài nguyên phần cứng, hỗ trợ giao diện người dùng.

Bài tập lớn "Hiện thực Hệ điều hành đơn giản" giúp sinh viên ứng dụng và nắm rõ những kiến thức đã học, đồng thời rút được kinh nghiệm từ việc hiện thực.

1.1 Nền tảng

Những thứ sinh viên phải trang bị cho bài tập lớn:

- Đã cài đặt Ubuntu, trình biên dịch C và chạy chương trình thành công
- Kiến thức về tiến trình (Process), luồng (Thread) và hệ đa nhiệm
- Kiến thức về định thời CPU
  - Phân biệt bộ điều phối (Dispatcher) và bộ định thời (Scheduler)
  - Nắm được các tiêu chí định thời (Turnaround time, Waiting time, ...)
  - Nắm rõ các giải thuật định thời, nhất là Multilevel Queue và ưu, nhược điểm của mỗi giải thuật
  - Giản đồ Gantt
- Kiến thức về các công cụ đồng bộ (mutex, conditional variable, ...) và ưu nhược điểm của chúng
- Kiến thức về bộ nhớ
  - Hạn chế của cơ chế cấp phát liên tục và ưu điểm của cơ chế cấp phát không liên tục
  - Địa chỉ luận lý, địa chỉ vật lý và bảng phân trang
  - Chuyển đổi địa chỉ luận lý sang địa chỉ vật lý thông qua khối bộ nhớ MMU (memory management unit)
  - Cơ chế phân trang (paging), hoán đổi (swapping) và các giải thuật thay thế trang

1.2 Mục tiêu

Sinh viên phải hiện thực hệ điều hành đơn giản này trong ngôn ngữ C, bằng cách đọc hiểu đặc tả bài tập lớn và mã nguồn, so sánh kết quả chương trình sau khi chạy với kết quả cho trước.

Sau khi làm xong bài tập lớn, sinh viên phải đạt được những tiêu chí sau:

- L.O.1 - Mô tả cách ứng dụng các kiến thức nền tảng của máy tính và toán học trong hệ điều hành
- L.O.1.1 - Định nghĩa được các chức năng và cấu trúc của một hệ điều hành hiện đại theo nhu cầu cụ thể nào đó
- L.O.1.2 - Giải thích bộ nhớ ảo và hiện thực cả về phần cứng lẫn phần mềm
- L.O.2 - Mối quan hệ giữa hiệu suất và các hạn chế về tài nguyên và công nghệ trong một thiết kế của một hệ điều hành

- L.O.2.1 - So sánh và làm nổi bật các giải thuật thông dụng để định thời công việc trong hệ điều hành
- L.O.2.2 - So sánh và làm nổi bật các giải pháp khác nhau về tổ chức hệ thống tập tin, đưa ra được các ưu / khuyết điểm của mỗi giải pháp
- L.O.3 - Mô tả các khái niệm cơ bản của hệ điều hành cùng các đặc điểm hữu ích nhằm hiện thực hệ thống đồng thời và mô tả tiện ích của mỗi đặc điểm.
- L.O.3.1 - So sánh và làm nổi bật các phương pháp đồng bộ quá trình

2 Cơ sở lý thuyết

2.1 Định thời

Là quá trình xác định thứ tự thực hiện các công việc hoặc quy trình trên đơn vị xử lý trung tâm (CPU) của máy tính. Mục tiêu chính của định thời là tối ưu hóa sử dụng tài nguyên hệ thống, cải thiện hiệu suất hệ thống và cung cấp môi trường thực thi công bằng và hiệu quả cho nhiều quá trình.

Trong hệ điều hành đơn giản này, chúng ta sử dụng giải thuật định thời Multilevel Queue, được dựa trên giải thuật định thời trong Linux.

Bộ định thời hoạt động như sau: mỗi khi có chương trình mới, trình tải (loader) sẽ tạo một quá trình mới và gán một PCB (Process Control Block - Khối điều khiển quá trình) cho nó. Sau đó, trình tải đọc và sao chép nội dung của chương trình vào đoạn văn bản (text segment) của quá trình mới. PCB của quá trình được đẩy vào *ready_ queue_ list* có cùng độ ưu tiên (prio). Sau đó, trình tải đợi CPU. CPU chạy các quá trình theo kiểu Round Robin. Mỗi quá trình được phép chạy trong một đoạn thời gian cố định. Sau đó, CPU buộc phải đưa quá trình này vào *ready_ queue_ list*. CPU sau đó chọn một quá trình khác từ *ready_ queue_ list* và tiếp tục chạy.

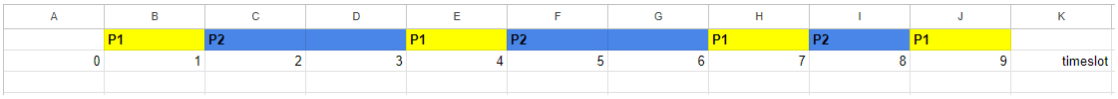
Trong hệ thống này, chúng ta triển khai giải thuật Multi-Level Queue (MLQ). Hệ thống bao gồm MAX_PRIO mức ưu tiên. Chúng ta đơn giản hóa việc thêm vào hàng đợi và đặt quá trình bằng cách đặt quá trình vào hàng đợi sẵn sàng phù hợp dựa trên độ ưu tiên. Giải thuật MLQ sẽ sử dụng *get_ proc* để lấy một quá trình và sau đó giao cho CPU xử lý.

Mô tả về giải thuật MLQ: ta lặp qua *ready_ queue_ list* được xác định dựa trên ưu tiên (prio), tức là **slot = (MAX PRIO - prio)**. Mỗi *ready_ queue* chỉ có một số lượng slot nhất định để sử dụng CPU và khi đã sử dụng hết, hệ thống phải chuyển tài nguyên sang quá trình khác trong *ready_ queue* tiếp theo và để lại công việc còn lại cho slot kế tiếp trong tương lai, ngay cả khi phải lặp lại 1 vòng của *ready_ queue_ list*.
Ví dụ: giả sử ta có cấu hình như sau:

- MAX_PRIO: 2
- num processes: 2
- num cpus: 1
- 1 time quantum = 1 slot

Process	Arrival time	Burst time	Priority
P1	0	4	1
P2	1	5	0

Khi đó ta có giản đồ Gantt như sau:



Hình 1: Giản đồ Gantt của giải thuật định thời MLQ

Giải thích: prio 0 có MAX PRIO - prio = 2 - 0 = 2 slot, prio 1 có 1 slot. CPU chạy các quá trình theo kiểu Round Robin, tức là chạy mỗi process trong thời gian là time quantum. Vì prio càng cao thì càng được chạy nhiều, nên queue 0 nhận được 2 slot để chạy, mỗi slot chạy trong khoảng thời gian time quantum.

2.2 Bộ nhớ

Bộ nhớ ảo trong hệ điều hành là một khái niệm quan trọng giúp tối ưu hóa việc sử dụng bộ nhớ của máy tính. Nó cho phép các chương trình chạy trên hệ điều hành có cảm giác như chúng có một không gian bộ nhớ lớn hơn so với bộ nhớ vật lý thực tế có sẵn trên máy. Trong hệ thống này, không gian bộ nhớ ảo sẽ bao gồm memory area và memory region. Ta có thể coi memory area là một memory segment

- Memory area: kéo dài liên tục trong khoảng [vm_start, vm_end]. Mặc dù vậy, chỉ có thể sử dụng từ vm_start đến con trỏ sbrk. Dùng để cấp phát và quản lý các memory region
- Memory region: Là nơi lưu trữ các biến trong mã nguồn của chương trình mà con người có thể đọc được.

Bộ nhớ chính trong hệ điều hành là một phần quan trọng của hệ thống, nơi lưu trữ dữ liệu và chương trình đang hoạt động. Không gian bộ nhớ thực sẽ bao gồm RAM và SWAP

- Bộ nhớ truy cập ngẫu nhiên RAM (Random Access Memory), được sử dụng để lưu trữ dữ liệu tạm thời và các phần mềm đang chạy. Khi một chương trình được khởi chạy, nó sẽ được nạp từ ổ đĩa cứng vào bộ nhớ chính để thực thi.
- Bộ nhớ SWAP, còn được gọi là không gian swap, là một khu vực trên ổ đĩa cứng của máy tính được sử dụng như một phần mở rộng của RAM. Khi RAM vật lý đầy và hệ thống cần thêm nguồn tài nguyên bộ nhớ để chạy ứng dụng hoặc quy trình, hệ điều hành (OS) sẽ đưa dữ liệu từ RAM sang không gian swap trên ổ đĩa.

Kĩ thuật phân trang là một kỹ thuật quản lý bộ nhớ giúp tối ưu hóa việc sử dụng bộ nhớ vật lý của máy tính. Thay vì phải cố gắng tìm một khối liên tục và lớn trong bộ nhớ, hệ điều hành chia bộ nhớ thành các trang có kích thước cố định. Mỗi trang có kích thước nhỏ và có thể tồn tại ở bất kỳ vị trí nào trong bộ nhớ vật lý.

Bảng phân trang là một cấu trúc cho phép một quá trình trong không gian người dùng xác định khung trang nào được ánh xạ cho mỗi trang. Bảng phân trang chứa một giá trị 32-bit cho mỗi trang, bao gồm dữ liệu sau đây:

```
* Bits 0-12  page frame number (PFN) if present
* Bits 13-14 zero if present
* Bits 15-27 user-defined numbering if present
* Bits 0-4   swap type if swapped
* Bits 5-25  swap offset if swapped
* Bit  28    dirty
* Bits 29    reserved
* Bit  30    swapped
* Bit  31    presented
```

Hình 2: Page table entry

,

Kĩ thuật hoán đổi (Swapping): là quá trình di chuyển một quá hoặc một phần của quá trình giữa bộ nhớ chính (RAM) và bộ nhớ phụ, thường là không gian SWAP trong ổ đĩa cứng. Nhờ sự lân cận về không gian và thời gian (spatial locality, temporal locality) mà quá trình có thể chạy khi chỉ cần một phần không gian bộ nhớ của process đó, thay vì phải tải lên toàn bộ, từ đó giúp tiết kiệm bộ nhớ cho các quá trình khác, và có khả năng chạy đượng các quá trình tiêu tốn nhiều bộ nhớ.

3 Hiện thực và kết quả

3.1 Hiện thực các hàm

Ta sẽ đi qua các file và các hàm cần hiện thực, ý nghĩa và cách hiện thực nó:
queue.c:

- **enqueue**: thêm PCB vào *ready_queue*, là một array có size là *MAX_QUEUE_SIZE*. Sẽ có trường hợp khi *ready_queue* đầy thì ta không cần thêm, ngoài ra thì ta chỉ đơn giản thêm vào cuối mảng và cập nhật biến size.
- **dequeue**: lấy PCB ra khỏi *ready_queue*. Ta chỉ cần lấy phần tử đầu tiên ra, dịch các phần tử ở sau lên 1 vị trí và cập nhật biến size

sched.c:

- **get_mlq_proc**: Hàm này dùng để lấy process tiếp theo trong *ready_queue_list*. Do phải giữ trạng thái đang sử dụng *ready_queue* nào và đang còn bao nhiêu *slot*, ta có thể khai báo biến toàn cục (global variable) hoặc khai báo biến tĩnh (static variable) đối với hàm. Nhóm đã chọn phương án 2, và hiện thực sau đó diễn ra như sau:
  - Khoá khoá đồng bộ, việc này để tránh data race khi có nhiều CPU
  - kiểm tra xem *ready_queue_list* có trống hay không, nếu trống thì mở khoá khoá đồng bộ và thoát chương trình
  - Nếu *ready_queue* hiện tại còn 0 *slot*, ta di chuyển đến *ready_queue* không trống tiếp theo và cập nhật biến *slot*.
  - Lấy PCB từ *ready_queue* và cập nhật biến *slot*
  - Mở khoá khoá đồng bộ và trả về kết quả

mm-vm.c:

- **__alloc**: Hàm dùng để cấp phát memory region trong một memory area. Hiện thực diễn ra như sau:
  - Tìm một region còn trống. Nếu có, dùng nó để cấp phát
  - Nếu không có, tức là area đã hết region còn trống. Ta sẽ tăng giới hạn của memory area, khi đó sẽ có region trống để cấp phát.
  - Trường hợp region trống đó lớn hơn kích cỡ cần cấp phát, ta sẽ chia region ban đầu làm 2, một dùng để cấp phát, một là region trống.
- **__free**: Hàm dùng để thu hồi memory cấp phát. Ta chỉ đơn giản là thêm region đó vào danh sách region trống, và vô hiệu hoá region trong bảng symrgtbl.
- **validate_overlap_vm_area** Kiểm tra xem memory area có đè lên memory area đã có hay không, tránh lỗi truy cập bộ nhớ. Ta đơn giản là lặp qua tất cả memory area để kiểm tra
- **find_victim_page** Hàm dùng trong giải thuật thay trang, khi ta đưa khung trang từ trong SWAP vào RAM, cũng có nghĩa là phải chọn khung trang để đưa từ RAM sang SWAP. Ta dùng giải thuật FIFO (first in first out) để thay trang.
- **pg_getpage**: Hàm để lấy khung trang từ số trang. Hiện thực diễn ra như sau:

- Từ số trang, ta lấy được entry trong bảng trang
- Từ entry ta kiểm tra xem khung trang có trong bộ nhớ không. Nếu có ta đơn giản lấy khung trang đó và trả về.
- Nếu không ta sẽ dùng giải thuật thay trang để hoán đổi khung trang cần sử dụng với khung trang nào đó dựa trên giải thuật thay trang của chúng ta
- Trả về khung trang cần tìm

mm.c

- **vmap_page_range**: Hàm dùng để ánh xạ các trang bắt đầu từ một địa chỉ nhất định trong bảng trang đến các khung trang. Ta sẽ lặp qua các entry trong bảng bắt đầu từ *addr* và ánh xạ nó đến khung trang được cho trước theo thứ tự.

mm-memphy.c

- **MEMPHY_dump**: dùng để dump memory (in thông tin bộ nhớ) để theo dõi thông tin bộ nhớ vật lý. Ta sẽ in mỗi hàng là 4 byte trong bộ nhớ vật lý tại mỗi địa chỉ (là một con số chia hết cho 4).

3.2 Kết quả chạy chương trình

Kết quả chạy các testcase sẽ ở trong thư mục our_output

Vì có nhiều testcase, nhóm sẽ phân tích 1 testcase là *os_0_mlq_paging* (có tinh chỉnh so với testcase gốc)

Input

```
%== os_0_mlq_paging
    6 2 2
    1048576 16777216 0 0 0
    0 p0s 0
    2 p1s 15

%== p0s
    1 10
    calc
    alloc 300 0
    alloc 300 4
    free 0
    alloc 100 1
    write 100 1 20
    read 1 20 20
    write 102 2 20
    read 2 20 20
    write 103 3 20
    read 3 20 20
    calc
    free 4
    calc

%== p1s
    1 10
    calc
```

```
27          calc
28          calc
29          calc
30          calc
31          calc
32          calc
33          calc
34          calc
35          calc
36          calc
```

Output

```
1       Time slot  0
2       ld_routine
3        Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
4        CPU 1: Dispatched process 1
5       Time slot  1
6        Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
7       Time slot  2
8        CPU 0: Dispatched process 2
9       Time slot  3
10      Time slot  4
11      Time slot  5
12      write region=1 offset=20 value=100
13      print_pgtbl: 0 - 512
14      00000000: 80000001
15      00000004: 80000000
16      ---MEM DUMP---
17      Time slot  6
18       CPU 1: Put process 1 to run queue
19       CPU 1: Dispatched process 1
20      read region=1 offset=20 value=100
21      print_pgtbl: 0 - 512
22      00000000: 80000001
23      00000004: 80000000
24      ---MEM DUMP---
25      000000dc: 00000064
26      Time slot  7
27      write region=2 offset=20 value=102
28      print_pgtbl: 0 - 512
29      00000000: 80000001
30      00000004: 80000000
31      ---MEM DUMP---
32      000000dc: 00000064
33      Time slot  8
34       CPU 0: Put process 2 to run queue
35       CPU 0: Dispatched process 2
36      read region=2 offset=20 value=102
37      print_pgtbl: 0 - 512
38      00000000: 80000001
```

```
00000004: 80000000
---MEM DUMP---
00000014: 00000066
000000dc: 00000064
Time slot  9
write region=3 offset=20 value=103
print_pgtbl: 0 - 512
00000000: 80000001
00000004: 80000000
---MEM DUMP---
00000014: 00000066
000000dc: 00000064
Time slot 10
 CPU 1: Processed 1 has finished
 CPU 1 stopped
Time slot 11
Time slot 12
 CPU 0: Processed 2 has finished
 CPU 0 stopped
```

Giải thích

- Chúng ta sẽ bắt đầu mô phỏng với time quantum = 6 time slot, 2 CPU và 2 process p0s và p1s
- p0s có arrival time là 0 và priority là 0, p1s có arrival time là 2 và priority là 15
- Time slot 0: p0s được load bởi loader, được gán PID là 1 và thực thi ngay lập tức bởi CPU 1
- Time slot 1: p1s được load bởi loader, được gán PID là 2
- Time slot 2: p1s bắt đầu thực thi
- Time slot 6: ta bắt đầu ghi vào region 1, offset là 20 giá trị 100 và in bảng trang và nội dung RAM. Do quá trình ghi bắt đầu sau khi in nên thông tin được in ra trước khi ghi. Cùng lúc đó P1 (process có PID là 1 nhưng ta gọi tắt) kết thúc thời gian thực thi (thực thi ở timeslot 1, 1 time quantum = 6 time slot nên sẽ bị thu hồi CPU vào time slot 0 + 6 = 6), và CPU chọn process tiếp theo để thực thi. Nhưng trong hàng đợi *ready_ queue_ list* Chỉ có mỗi P1 vừa được đặt vào nên lấy P1 ra thực thi tiếp.
- Time slot 7: Chúng ta tiếp tục in ra thông tin bảng trang và RAM trước khi thực hiện ghi. Để ý thấy RAM có nội dung 0x00000064 ở địa chỉ 0x000000dc. 0x00000064 trong hệ thập phân là 100, 0x00000000dc trong hệ thập phân là 220. Đây là giá trị được ghi ở time slot 6, vì region 1 có địa chỉ bắt đầu là 200, thêm offset 20 sẽ ra địa chỉ 220.
- Time slot 8: P2 đã hết thời gian được phép thực thi (thực thi ở time slot 2, time quantum = 6 time slot nên kết thúc ở time slot 2 + 6 = 8). CPU 0 tìm process kế và chỉ thấy P2 nên lấy thực thi tiếp. Ta thực hiện đọc ở region 2 và offset 20 được giá trị là 102 (đúng giá trị được ghi ở time slot 7). Tiếp đến ta in ra thông tin bảng trang và RAM. Nhận thấy RAM có giá trị mới là 0x0000000066 ở địa chỉ 0x00000014, trong hệ thập phân là giá trị 102 ở địa chỉ 20. Dễ dàng thấy đây là giá trị được ghi ở time slot 7, region 2 có địa chỉ bắt đầu là 0 và offset là 20 nên địa chỉ được ghi là 0 + 20 = 20.

- Time slot 9: Ta lại tiếp tục thực hiện ghi ở region 3, offset là 20 và giá trị là 103, nội dung bảng trang và RAM vẫn như cũ.
- Time slot 10: vì P1 đã xong (thực thi xong lệnh cuối cùng) và không còn chương trình nào được đưa vào nữa nên CPU dừng.
- Time slot 12. Tương tự với P1, P2 hoàn thành và CPU 0 dừng.

3.3 Kết luận

Thông qua testcase này ta đã kiểm tra được tính đúng đắn về định thời và bộ nhớ. Tuy vậy, hiện thực này vẫn có thể có sai sót, do số testcase là giới hạn. Hạn chế của hiện thực này là vẫn còn tồn tại internal fragmentation do các memory region trống liền kề không được gộp lại và tồn tại memory leak (page list và page frame, region list, area list không được free khi process hoàn thành).

4 Trả lời câu hỏi

4.1 What is the advantage of using priority queue in comparison with other scheduling algorithms you have learned?

Dịch: Lợi thế của việc sử dụng hàng đợi ưu tiên so với các giải thuật định thời khác là gì?
Trả lời:

So với các giải thuật định thời khác (FCFS, SJF, SRTF, ...), giải thuật hàng đợi ưu tiên trong hệ điều này có lợi thế:

- Phân bổ tài nguyên cho các quá trình một cách hiệu quả: định thời bằng hàng đợi ưu tiên giúp tối ưu CPU cho các quá trình có độ ưu tiên cao để thực thi kịp thời, đồng thời cho phép các quá trình có độ ưu tiên thấp hơn được thực thi, tránh hiện tượng đói tài nguyên (starvation)
- Tính công bằng: Do thực thi dựa trên round robin, mỗi quá trình đều có cơ hội được thực thi.
- Nhờ tính công bằng, giải thuật định thời này hỗ trợ tốt cho các tác vụ thời gian thực và các tác vụ có thời hạn ngắn.

4.2 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

Dịch: Trong hệ điều hành đơn giản này, chúng ta hiện thực thiết kế nhiều phân đoạn bộ nhớ hoặc khu vực bộ nhớ trong mã nguồn. Lợi ích của việc thiết kế nhiều phân đoạn là gì?
Trả lời

- Linh hoạt: có thể chia chương trình thành các mô-đun nhất định với kích thước tuỳ biến, giúp dễ dàng quản lý hơn
- So với phân trang thì phân đoạn giúp tiết kiệm bộ nhớ hơn (bảng phân đoạn bé hơn bảng phân trang)
- Khi đi kèm với phân trang sẽ hạn chế phân mảnh ngoại (vấn đề mà khi chỉ có phân đoạn sẽ gặp phải)
- Dễ dàng chia sẻ với các quá trình khác (ví dụ như shared library) hơn so với phân trang

4.3 What will happen if we divide the address to more than 2-levels in the paging memory management system

Dịch: Chuyện gì sẽ xảy ra nếu ta chia địa chỉ hơn 2 mức trong hệ thống phân trang bộ nhớ.
Trả lời:

Khi ta chia địa chỉ luận lý thành hơn 2 mức trong hệ thống phân trang thì:

- Giảm chi phí bộ nhớ cho bảng phân trang: chúng ta chỉ lưu các entry của bảng trang mà chương trình sử dụng hoặc đã load vào.
- Truy xuất bộ nhớ chậm hơn do mỗi lần truy xuất một mức của bảng phân trang đều phải truy xuất bộ nhớ.

- Phức tạp hơn trong việc hiện thực
- Có thể gây phân mảnh bộ nhớ do các bảng thuộc các mức không liền kề nhau

4.4 What is the advantage and disadvantage of segmentation with paging?

Dịch: ưu điểm và nhược điểm của việc sử dụng phân đoạn cùng với phân trang?
Trả lời:

Ưu điểm

- Linh hoạt hơn trong việc chia quá trình thành các phần khác nhau nhờ phân đoạn
- Giảm thiểu phân mảnh nội (internal fragmentation) nhờ phân trang
- Phân đoạn cho phép bảo vệ và chia sẻ tài nguyên giữa các quá trình

Nhược điểm

- Phức tạp trong việc hiện thực và quản lý
- Phân mảnh ngoại (external fragmentation) do phân đoạn thành các phần có kích thước khác nhau
- Tăng kích thước bộ nhớ dành cho bảng phân đoạn và bảng phân trang

4.5 What will happen if the synchronization is not handled in your simple OS? Illustrate by example the problem of your simple OS if you have any.

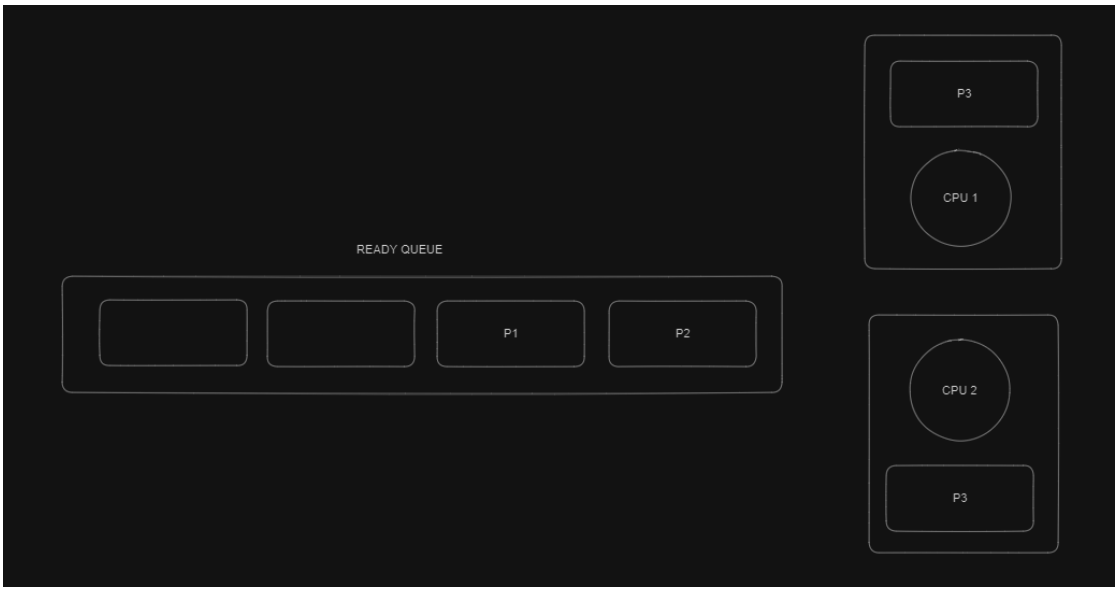
Dịch: Sẽ xảy ra vấn đề gì nếu ta không xử lý việc đồng bộ trong hệ điều hành đơn giản này? Nêu ví dụ nếu có.
Trả lời:

Do hệ điều hành sử dụng nhiều CPU, việc không xử lý đồng bộ sẽ xảy một số vấn đề đồng bộ phổ biến:

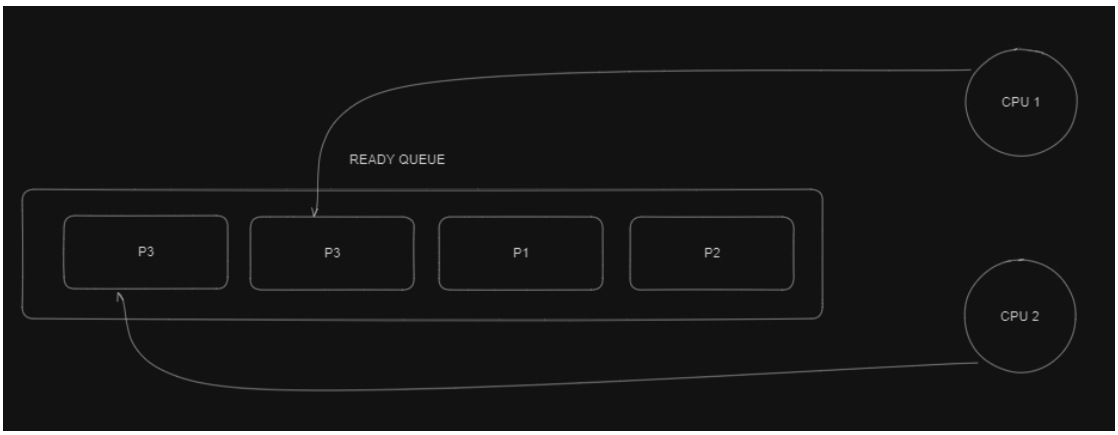
- Data race: xảy ra khi các process hoặc thread cùng thực hiện đọc và ghi lên một khối dữ liệu, sẽ dẫn đến tính không nhất quán của dữ liệu
- Race condition: trường hợp thứ tự sử dụng của nhiêu process hoặc thread có thể dẫn đến kết quả không như mong muốn
- Deadlock: xảy ra khi nhiều luồng đều chờ lẫn nhau để truy cập tài nguyên. Việc này sẽ khiến process hoặc thread bị ngưng vô thời hạn.

Sau đây là ví dụ một trường hợp có thể xảy ra bằng hình ảnh, có 2 CPU và 1 *ready_ queue*

Hình 3: Do không có xử lý đồng bộ, cả 2 CPU lấy process đầu tiên của *ready_ queue* cùng lúc, cụ thể là P3



Hình 4: Cả 2 CPU thực thi P3 (cho rằng trong quá trình này không xảy ra data race để dễ hình dung)



Hình 5: Cả 2 CPU đưa P3 về *ready_ queue* nhưng vì CPU 2 chậm hơn 1 chút nên giờ trong *ready_ queue* đã có 2 P3