Ho Chi Minh University of Technology
Faculty of Computer Science & Engineering

OPERATING SYSTEM

ASSIGNMENT

Simple Operating System

Teacher: Trần Trương Tuấn Phát
Class: L12
Students: Tạ Quang Thái - 2213117
Lê Thị Thu Thuỷ - 2213379
Trần Quang Huy - 2211287
Trần Phước Nhật - 2212412
Trần Văn Mạnh - 2212003

Ho Chi Minh City, 5/2024

Mục lục

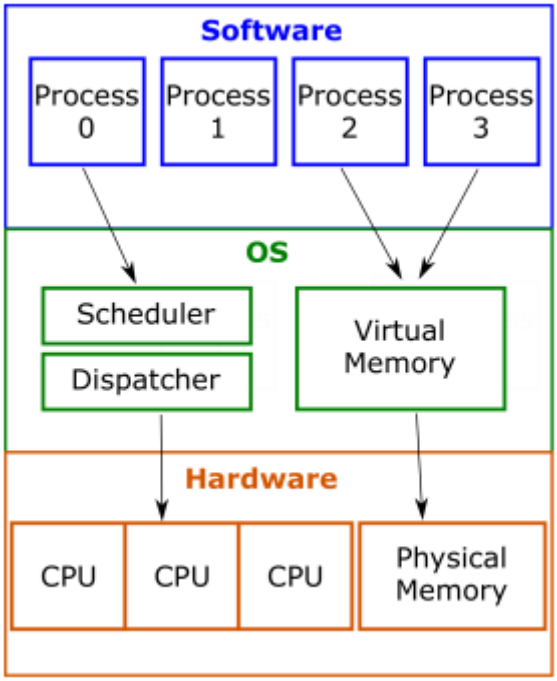
PHÂN CÔNG CÔNG VIỆC

Họ và tên	**MSSV**	**Công việc**	**Đóng góp**
Tạ Quang Thái	2213117	Viết code + báo cáo phần queue.c, sched.c, trả lời câu hỏi	20%
Lê Thị Thu Thuỷ	2213379	Viết code + báo cáo phần mm.c	20%
Trần Quang Huy	2211287	Viết code + báo cáo phần tlb	20%
Trần Phước Nhật	2212412	Viết code + báo cáo phần tlb , trả lời câu hỏi	20%
Trần Văn Mạnh	2212003	Viết code + báo cáo phần mm-memphy.c	20%

1 Giới thiệu

Mục đích của bài tập lớn này là mô phỏng một hệ điều hành đơn giản, giúp sinh viên hiểu kiến thức cơ bản về định thời (Scheduling), quá trình đồng bộ (Synchronization) và quản lý bộ nhớ (Memory Management). Hình 1.1 thể hiện tổng quan cấu trúc của hệ điều hành sẽ được hiện thực trong bài tập lớn. Về cơ bản, hệ điều hành sẽ quản lý 2 tài nguyên ảo: CPU(s) và RAM, sử dụng 2 thành phần:

- Scheduler (and Dispatcher): quyết định quá trình nào sẽ được thực thi trên CPU nào đó.
- Virtual Memory Engine (VME): cô lập không gian bộ nhớ của mỗi quá trình khỏi những quá trình khác. Mặc dù RAM được chia sẻ bởi nhiều quá trình, mỗi quá trình không biết sự tồn tại của các quá trình khác. Điều này được thực hiện bằng cách cho phép mỗi quá trình có không gian bộ nhớ ảo riêng và công cụ bộ nhớ ảo sẽ ánh xạ và dịch các địa chỉ logic được cung cấp bởi các quy trình sang các địa chỉ vật lý tương ứng.



Hình 1.1: *Hình ảnh mô phỏng các thành phần chính trong bài tập lớn*

Thông qua các module trên, hệ điều hành cho phép nhiều quá trình được tạo ra bởi người dùng chia sẻ và sử dụng các tài nguyên. Do đó, bài tập lớn này sẽ tiến hành hiện thực các thành phần Scheduler/Dispatcher và VME.

2 Hiện thực

2.1 Scheduler

2.1.1 Hiện thực Multi-level Queue

Trước tiên, chúng ta sẽ hiện thực scheduler. Mặc dù Hệ điều hành được thiết kế để hoạt động trên nhiều bộ xử lý, nhưng trong phần bài tập này, chúng ta giả sử hệ thống có nhiều bộ xử lý. Hệ điều hành sử dụng **ready_queue_system** để xác định quy trình nào sẽ được thực thi khi CPU khả dụng. Thiết kế bộ lập lịch dựa trên thuật toán "multilevel queue" được sử dụng trong Linux kernel.

Đối với mỗi chương trình mới, trình tải sẽ tạo một process mới và gán một PCB mới cho nó. Sau đó, bộ nạp sẽ đọc và sao chép nội dung của chương trình vào đoạn văn bản của process mới (được trỏ bởi con trỏ mã trong PCB của process). Cuối cùng, PCB của process được đẩy vào **ready_queue** và đợi CPU. CPU chạy các tiến trình theo thuật toán round-robin. Mỗi quá trình được phép chạy trong một khoảng thời gian nhất định. Sau đó, CPU buộc phải đưa process vào **ready_queue** ưu tiên thích hợp. Sau đó, CPU chọn một process khác từ **ready_queue** và tiếp tục chạy. Phần còn lại thuộc về mã quản lý CPU để thực thi nó trong **ready_queue**.

Trong hệ thống này, chúng ta sẽ triển khai Multi-level Queue (MLQ). Hệ thống chứa các mức ưu tiên **MAX_PRIO**. Mặc dù hệ thống thực, tức là Linux kernal, có thể nhóm các mức này thành tập hợp con, chúng ta vẫn sẽ giữ thiết kế trong đó mỗi mức độ ưu tiên được giữ bởi một **ready_queue** để đơn giản hóa vấn đề. Chúngta đơn giản hóa **add_queue** và **put_proc** như đặt proc vào **ready_queue** thích hợp bằng cách so khớp độ ưu tiên. Thiết kế chính thuộc về MLQ được triển khai bởi **get_proc** để tìm nạp một proc để phân phối tới CPU.

Mô tả về MLQ policy: bước duyệt qua danh sách hàng đợi sẵn sàng là một số có công thức cố định dựa trên mức độ ưu tiên, tức là vị trí = (**MAX PRIO** - ưu tiên)

MLQ policy chỉ đi qua bước cố định để duyệt qua tất cả hàng đợi trong danh sách **ready_queue** ưu tiên. Trong phần này, chúng ta sẽ thực hiện thuật toán này bằng cách hoàn thành các chức năng sau:

- **enqueue()** và **dequeue()** (trong queue.c): hoàn thiện struct **queue_t** để giúp đưa PCB mới vào hàng đợi và nhận PCB với mức ưu tiên cao nhất trong hàng đợi.
- **get proc()** (trong sched.c): nhận PCB của một process chở đợi tại **ready_queue**. Nếu hàng đợi trống tại thời điểm hàm được gọi, ta sẽ di chuyển tất cả PCB của các process đang chờ trong **run_queue** trở lại **ready_queue**.

Nội dung hiện thực như sau:

```
void enqueue(struct queue_t *q, struct pcb_t *proc){
    if (q->size >= MAX_QUEUE_SIZE)
        return;
    if (q->size >= 0){
#ifdef MLQ_SCHED
        if(proc->priority != proc->prio){
            proc->priority = proc->prio;
        }
#endif
        q->proc[q->size] = proc;
        q->size++;
    }
}

struct pcb_t *dequeue(struct queue_t *q){
    if (!q->size)
        return NULL;
    int id = 0;
    for (int i = 0; i < q->size; i++)
        if (q->proc[id]->priority < q->proc[i]->priority)
            id = i;
    struct pcb_t *temp = q->proc[id];
    q->size--;
    for (int i = id; i < q->size; i++)
        q->proc[i] = q->proc[i + 1];
    return temp;
}
```

Program 1: *Hiện thực* ***enqueue()*** *và* ***dequeue()***

```
struct pcb_t *get_mlq_proc(void) {
    if(mlq_ready_queue[curr_prio_queue].slot > 0){
    unsigned long i;
    for(i = curr_prio_queue; i < MAX_PRIO;i++){
      if(mlq_ready_queue[i].size > 0){
        pthread_mutex_lock(&queue_lock);
        proc = dequeue(&mlq_ready_queue[i]);
        mlq_ready_queue[i].slot --;
        pthread_mutex_unlock(&queue_lock);
        return proc;
      }
    }
```

```
13   }else{
14     mlq_ready_queue[curr_prio_queue].slot = MAX_PRIO - curr_prio_queue;
15     ++curr_prio_queue;
16     if(curr_prio_queue == MAX_PRIO) curr_prio_queue = 0;
17     unsigned long i;
18         for(i = curr_prio_queue; i < MAX_PRIO;i++){
19             if(mlq_ready_queue[i].size > 0){
20                 pthread_mutex_lock(&queue_lock);
21                 proc = dequeue(&mlq_ready_queue[i]);
22                 mlq_ready_queue[i].slot --;
23                 pthread_mutex_unlock(&queue_lock);
24                 return proc;
25             }
26         }
27   }
28   return proc;
29 }
```

Program 2: *Hiện thực* ***get_mlq_proc()***

2.1.2 Biểu đồ Gantt

Trong Bài tập lớn lần này, input của chương trình theo mẫu dưới đây:

[time quantum] [N = Number of CPU] [M = Number of Processes]
[time 0] [process 0] [[priority 0]
[time 1] [process 1] [[priority 1]
...
[time M-1] [process M-1] [priority M-1]

Process có định dạng như dưới đây:

[priority] [N = number of instructions]
instruction 0
instruction 1
...
instruction N-1

Lưu ý: Giá trị priority của process chỉ là giá trị mặc định, nó có thể được ghi đè bởi giá trị priority "trực tiếp" trong quá trình tải nap process (nếu có) - cũng là giá trị priority trong file input.
Dựa trên các testcase đã cho trong bài tập lớn này, chúng ta có sơ đồ Gantt sau: *(với giả định rằng CPU*

chạy đầu tiên là CPU 0, process bắt đầu tại thời điểm 0)

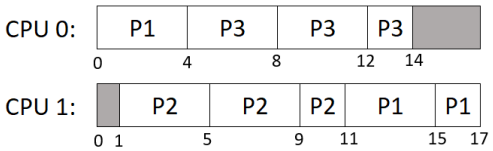
- sched:

Process	PID	PRIO	Priority	Arrival time	Code size
$p1s$	1	1	1	0	10
$p1s$	2	0	1	1	10
$p1s$	3	0	1	2	10

Time slice: 4

Number of CPU(s): 2

Gantt diagram:



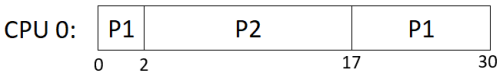
- sched_0:

Process	PID	PRIO	Priority	Arrival time	Code size
$s0$	1	4	12	0	15
$s0$	2	0	12	1	15

Time slice: 2

Number of CPU(s): 1

Gantt diagram:



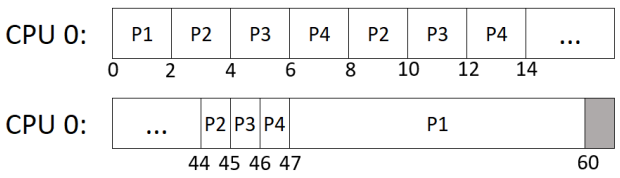
- sched_1:

Process	PID	PRIO	Priority	Arrival time	Code size
$s0$	1	4	12	0	15
$s0$	2	0	12	1	15
$s0$	3	0	12	2	15
$s0$	4	0	12	3	15

Time slice: 2

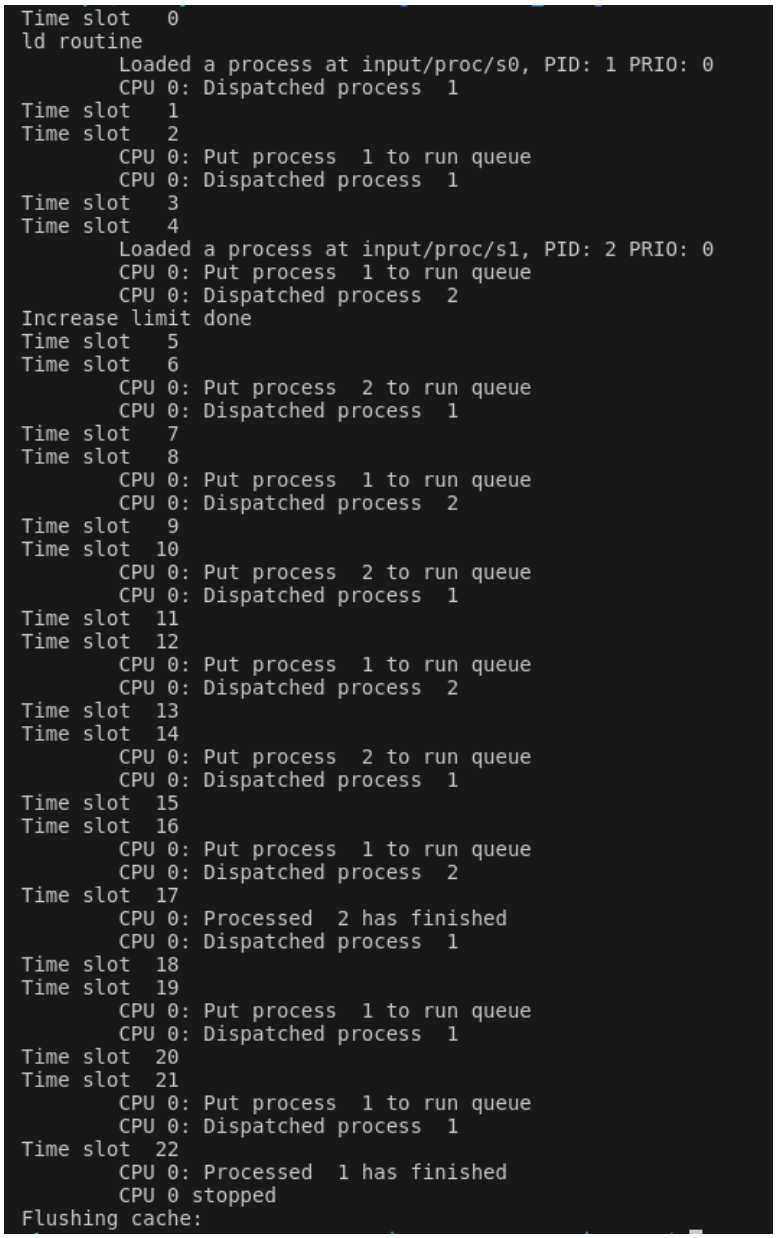
Number of CPU(s): 1

Gantt diagram:



2.1.3 Biên dịch và thực thi chương trình

- Chạy file sched_0:



Kết quả thực thi test case sched_0

- Chạy file sched_1:

```
Time slot   0
ld routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 0
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   4
        Loaded a process at input/proc/s1, PID: 2 PRIO: 0
Time slot   5
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Increase limit done
Time slot   6
        Loaded a process at input/proc/s2, PID: 3 PRIO: 0
Time slot   7
        CPU 0: Put process  2 to run queue
        Loaded a process at input/proc/s3, PID: 4 PRIO: 0
        CPU 0: Dispatched process  1
Time slot   8
Time slot   9
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  10
Time slot  11
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  14
Time slot  15
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  1
Time slot  16
Time slot  17
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  18
Time slot  19
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  20
Time slot  21
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  22
Time slot  23
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  1
Time slot  24
Time slot  25
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  26
```

Kết quả thực thi test case sched_1 (1/2)

```
Time slot  21
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  22
Time slot  23
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  1
Time slot  24
Time slot  25
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  26
Time slot  27
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  2
Time slot  28
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  4
Time slot  29
Time slot  30
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  1
Time slot  31
Time slot  32
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  33
Time slot  34
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  35
Time slot  36
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  1
Time slot  37
Time slot  38
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  39
Time slot  40
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  41
Time slot  42
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  1
Time slot  43
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  3
Time slot  44
Time slot  45
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  4
Time slot  46
        CPU 0: Processed  4 has finished
        CPU 0 stopped
Flushing cache:
```

Kết quả thực thi test case sched_1 (2/2)

So sánh kết quả thực thi và kết quả phân tích lý thuyết cho thấy các kết quả này phù hợp với nhau. Có thể kết luận các hàm phục vụ chức năng scheduling đã được hiện thực chính xác.

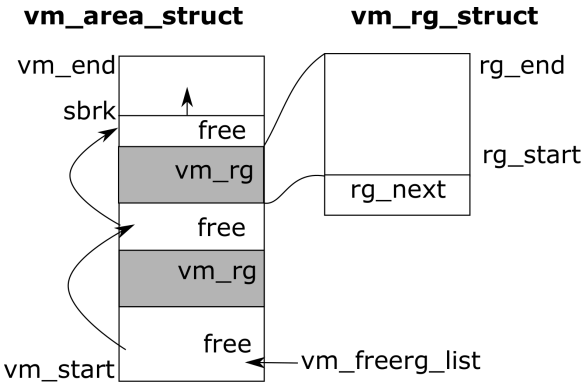
2.2 Memory Management

2.2.1 Ánh xạ vùng nhớ ảo trong mỗi process

Không gian bộ nhớ ảo được tổ chức dưới dạng bản đồ bộ nhớ cho từng process PCB (Process Control Block). Từ goc nhin cua process, địa chỉ ảo bao gồm nhiều vùng nhớ ảo (liền kề). Trong thực tế, mỗi

vùng có thể hoạt động như một đoạn mã (code), ngăn xếp (stack) hoặc heap. Do đó, quá trình giữ một con trỏ đến nhiều vùng nhớ liền kề trong PCB cua nó.

Memory area Mỗi khu vực bộ nhớ nằm trong phạm vi liên tục `[vm_start, vm_end]`. Mặc dù không gian bao phủ toàn bộ phạm vi này, khu vực thực sự có thể sử dụng bị giới hạn bởi đỉnh trỏ tới `sbrk`. Trong khoảng giữa `[vm_start và sbrk]`, có nhiều vùng được xác định bằng cấu trúc `vm_rg_struct` và các vị trí trống được theo dõi bởi danh sách `vm_freerg_list`. Qua thiết kế này, chúng ta chỉ thực hiện phân bố thực tế của bộ nhớ vật lý trong khu vực có thể sử dụng.



Hình 2.1: *Cấu trúc của vm area và region*

```
1  //From include/os-mm.h
2  /*
3  * Memory region struct
4  */
5  struct vm_rg_struct {
6      unsigned long rg_start;
7      unsigned long rg_end;
8      struct vm_rg_struct *rg_next;
9  };
10 /*
11 * Memory area struct
12 */
13 struct vm_area_struct {
14     unsigned long vm_id;
15     unsigned long vm_start;
16     unsigned long vm_end;
17     unsigned long sbrk;
18 /*
19 * Derived field
20 * unsigned long vm_limit = vm_end - vm_start
21 */
22     struct mm_struct *vm_mm;
23     struct vm_rg_struct *vm_freerg_list;
```

```
    struct vm_area_struct *vm_next;
};

```

Memory region Tạm thời, tưởng tượng rằng những khu vực này là một tập hợp các khu vực có số lượng giới hạn. Quản lý chúng bằng cách sử dụng một mảng `symrgtbl[PAGING_MAX_SYMTBL_SZ]`. Kích thước mảng được cố định bằng một hằng số, `PAGING_MAX_SYMTBL_SZ`, chỉ số lượng biến được phép trong mỗi chương trình. Tóm lại, chúng tôi sử dụng cấu trúc `vm_rg_struct symrgtbl` để lưu trữ điểm bắt đầu và kết thúc của khu vực, và con trỏ `rg_next` được dành cho việc theo dõi các bộ thiết lập trong tương lai.

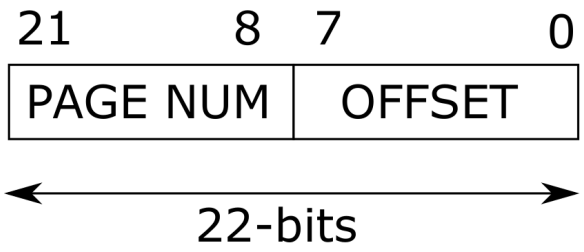
```
//From include/os-mm.h
/*
* Memory mapping struct
*/
struct mm_struct {
    uint32_t *pgd;
    struct vm_area_struct *mmap;
/* Currently we support a fixed number of symbol */
    struct vm_rg_struct symrgtbl[PAGING_MAX_SYMTBL_SZ];
    struct pgn_t *fifo_pgn;
    sem_t memlock;
};

```

Memory mapping được biểu diễn bằng cấu trúc `mm_struct`, theo dõi tất cả các khu vực bộ nhớ được đề cập trong một khu vực bộ nhớ liên tục riêng biệt. Trong mỗi cấu trúc memory mapping, nhiều khu vực bộ nhớ được chỉ định bởi `struct vm_area_struct *mmap list`. Trường quan trọng tiếp theo là `pgd`, đó là thư mục bảng trang (page table directory), chứa tất cả các mục bảng trang. Mỗi mục là một bản đồ giữa số trang và số khung trong hệ thống quản lý bộ nhớ phân trang. `Symrgtbl` là một hiên thực đơn giản của bảng ký hiệu (symbol table).

CPU addresses địa chỉ được CPU tạo ra để truy cập vào một vị trí bộ nhớ cụ thể. Trong hệ thống paging-based, nó được chia thành:

- Page number (p): được dùng như index vào 1 page table giữ địa chỉ cơ sở cho mỗi page trong bộ nhớ vật lý.
- Page offset (d): kết hợp với địa chỉ cơ sở để định nghĩa địa chỉ vật lý mà được gửi tới Memory Management Unit.

Không gian địa chỉ vật lý của 1 process có thể không liền kề. Ta chia bộ nhớ vật lý thành các fixed-sized block (the frames) với 2 kích thước là 256B hoặc 512B. Ta đề xuất các kết hợp cài đặt khác nhau ở Bảng 1 và kết thúc với cấu hình được đánh dấu. Đây là một cài đặt tham khảo và có thể modified hoặc re-selected ở các mô phỏng khác. Dựa theo cấu hình của CPU 22-bit và 256B page size, địa chỉ CPU được tổ chức như ở Hình 2.6.



Hình 2.2: *CPU address*

Trong bản tóm tắt, tất cả cấu trúc hỗ trợ VM được đặt trong module `mm_vm.c`.

CPU Bus	PAGE size	PAGE bit	No pg entry	PAGE Entry sz	PAGE TBL	OFFSET bit	PGT mem	MEMPHY	fram bit
20	256B	12	~4000	4 byte	16KB	8	2MB	1MB	12
22	256B	14	~16000	4 byte	64KB	8	8MB	1MB	12
22	512B	13	~8000	4 byte	32KB	9	4MB	1MB	11
22	512B	13	~8000	4 byte	32KB	9	4MB	128KB	8
16	512B	8	256	4 byte	1KB	9	128K	128KB	4

2.2.2 Bộ nhớ vật lý hệ thống

Hình 1 cho thấy phần cứng bộ nhớ được cài đặt dưới dạng toàn bộ hệ thống. Tất cả các process đều sở hữu các memory mapping riêng biệt của chung, nhưng tất cả các mapping này đều nhắm vào một thiết bị vật lý duy nhất. Có hai loại thiết bị là RAM và SWAP. Cả hai loại này có thể được triển khai bằng cùng một thiết bị vật lý, như trong `mm-memphy.c` với các setting khác nhau. Các setting được hỗ trợ bao gồm truy cập bộ nhớ ngẫu nhiên, truy cập bộ nhớ tuần tự/serial, và dung lượng lưu trữ.

Mặc dù có nhiều cấu hình khả thi, việc sử dụng logic của các thiết bị này có thể được phân biệt. Thiết bị RAM, thuộc về hệ thống bộ nhớ chính, có thể truy cập trực tiếp từ bus địa chỉ của CPU, tức là có thể đọc/ghi bằng các lệnh CPU. Trong khi đó, SWAP chỉ là một thiết bị bộ nhớ phụ, và tất cả các thao tác xử lý dữ liệu được lưu trữ trên SWAP phải được thực hiện bằng cách chuyển chúng vào bộ nhớ chính. Vì thiếu khả năng truy cập trực tiếp từ CPU, hệ thống thường trang bị một SWAP lớn với chi phí thấp và thậm chí có nhiều hơn một phiên bản. Trong cài đặt nay, chúng ta hỗ trợ phần cứng được cài đặt với một thiết bị RAM và tối đa 4 thiết bị SWAP.

struct framephy_struct chủ yếu được sử dụng để lưu trữ số khung (frame number).

struct memphy_struct có các trường cơ bản storage và size. Trường `rdmflg` xác định truy cập bộ nhớ là truy cập ngẫu nhiên hay tuần tự. Các trường `free_fp_list và used_fp_list` lần lượt dành để lưu giữ các khung bộ nhớ không sử dụng và đã sử dụng.

```c
//From include/os-mm.h
/*
* FRAME/MEM PHY struct
*/
struct framephy_struct {
    int fpn;
    struct framephy_struct *fp_next;
    /* Resereed for tracking allocated framed */
    struct mm_struct *owner;
};

struct memphy_struct{
   /* Basic field of data and size */
   BYTE *storage;
   int maxsz;
   sem_t MEMPHY_lock;
   /* Sequential device fields */
   int rdmflg;
   int cursor;
   /* Management structure */
   struct framephy_struct *free_fp_list;
   struct framephy_struct *used_fp_list;
   struct TLB_node *tlb_head;
   struct TLB_node *tlb_tail;
};
```

2.2.3 Phân tích trạng thái của RAM qua các câu lệnh trong test case

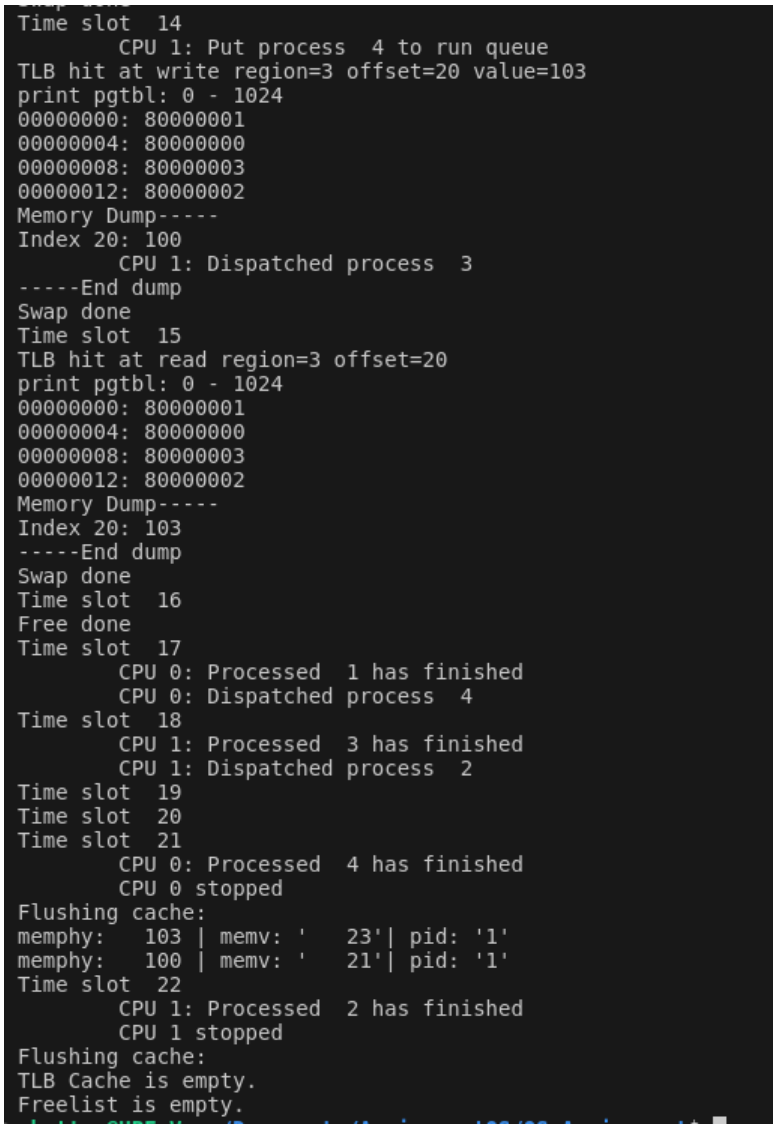
- Phân tích input của testcase:
  - Dòng đầu tiên <**time slot**> <**num of cpus**> <**num of processes**> :
    - <**time slot**>: Thời gian tối đa mỗi process có thể chạy trong mỗi time slot
    - <**num of cpus**>: Số lượng cpu
    - <**num of processes**: số lượng process cần chạy trong testcase
  - Kích thước TLB <**tlb size**>: kích thước cache được cấp cho tlb
  - Kích thước memory và swaps(tối đa là 4) <**ram memory**> <**swap memory**>**x4**: cung cấp vùng nhớ cho ram và swap
  - Thông tin về process <**start time**> <**process name**> <**priority**>:
    - <**start time**> : thời gian process tiến vào hàng đợi.
    - <**process name**> : tên process
    - <**priority**> (nếu có): độ ưu tiên của process trong hàng đợi.
- Chức năng lệnh trong mỗi process:
  - CALC : Đóng vai trò là một phép tính luôn trả kết quả là 0.
  - ALLOC <size> <register index> : Khởi tạo vùng nhớ và lưu địa chỉ vùng nhớ thông quá register index để các process có thể sử dụng.
  - FREE <register index> : Giải phóng vùng nhớ tại các địa chỉ được lưu thông qua register index.
  - READ <source register index> <offset> <destination register index> : Đọc dữ liệu từ vùng nhớ lưu trong [source register + offset] và lưu địa chỉ vùng nhớ vừa được đọc vào [destination register] trong process
  - WRITE <data> <destination register index> <offset> : Ghi dữ liệu vào vùng nhớ có địa chỉ được lưu trong [destination register + offset]

2.2.4 Biên dịch và thực thi chương trình

- Chạy os_0_mlq_paging:

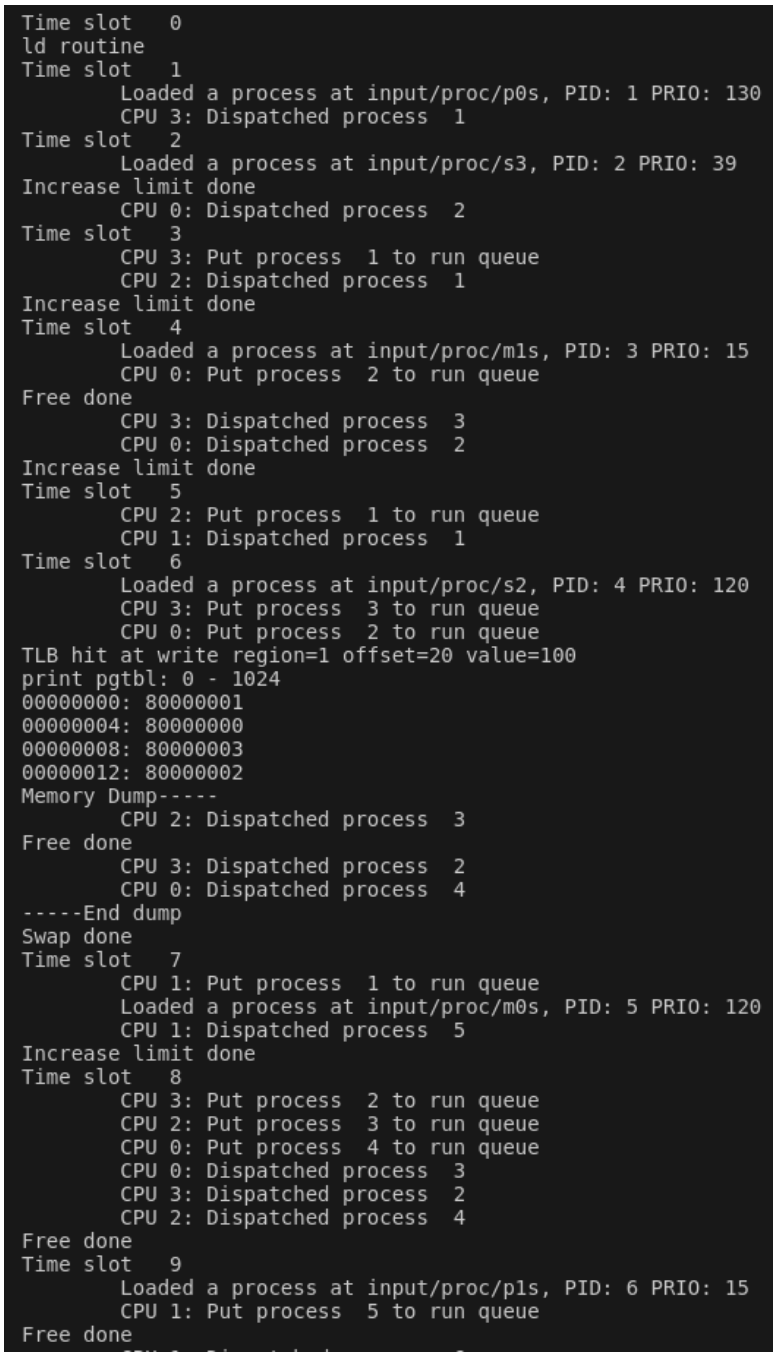
```
Time slot   0
ld routine
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
        Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
Increase limit done
        CPU 1: Dispatched process  2
Time slot   3
        Loaded a process at input/proc/p1s, PID: 3 PRIO: 0
Increase limit done
Time slot   4
Free done
        Loaded a process at input/proc/p1s, PID: 4 PRIO: 0
Time slot   5
Time slot   6
TLB hit at write region=1 offset=20 value=100
print pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Memory Dump-----
-----End dump
Swap done
Time slot   7
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot   8
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  4
Time slot   9
Time slot  10
Time slot  11
Time slot  12
Time slot  13
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
TLB hit at read region=1 offset=20
print pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Memory Dump-----
Index 20: 100
-----End dump
Swap done
Time slot  14
        CPU 1: Put process  4 to run queue
TLB hit at write region=3 offset=20 value=103
print pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Memory Dump-----
```

Kết quả thực thi test case os_0_mlq_paging (1/2)



Kết quả thực thi test case os_0_mlq_paging (2/2)

- Chạy os_1_mlq_paging:



Kết quả thực thi test case os_1_mlq_paging (1/4)

```
Time slot   9
        Loaded a process at input/proc/p1s, PID: 6 PRIO: 15
        CPU 1: Put process  5 to run queue
Free done
        CPU 1: Dispatched process  6
Time slot  10
        CPU 3: Put process  2 to run queue
        CPU 0: Put process  3 to run queue
        CPU 2: Put process  4 to run queue
        CPU 3: Dispatched process  3
        CPU 0: Dispatched process  2
Free done
        CPU 2: Dispatched process  5
Free done
Time slot  11
        Loaded a process at input/proc/s0, PID: 7 PRIO: 38
Free done
        CPU 1: Put process  6 to run queue
        CPU 1: Dispatched process  6
Time slot  12
        CPU 3: Processed  3 has finished
        CPU 0: Put process  2 to run queue
        CPU 2: Put process  5 to run queue
        CPU 3: Dispatched process  7
        CPU 0: Dispatched process  2
        CPU 2: Dispatched process  4
Time slot  13
        CPU 1: Put process  6 to run queue
        CPU 0: Processed  2 has finished
        CPU 1: Dispatched process  6
        CPU 0: Dispatched process  5
TLB hit at write region=1 offset=20 value=102
print pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Memory Dump-----
Index 20: 100
-----End dump
Swap done
Time slot  14
        CPU 3: Put process  7 to run queue
TLB hit at write region=2 offset=1000 value=1
        CPU 2: Put process  4 to run queue
print pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Memory Dump-----
Index 20: 100
Index 64: 102
        CPU 3: Dispatched process  7
        CPU 2: Dispatched process  4
-----End dump
Swapping
Swap done
Time slot  15
        CPU 0: Put process  5 to run queue
        CPU 1: Put process  6 to run queue
        CPU 0: Dispatched process  6
```

Kết quả thực thi test case os_1_mlq_paging (2/4)

```
        CPU 1: Dispatched process  5
TLB hit at write region=0 offset=0 value=0
print pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
Memory Dump-----
Index 20: 100
Index 64: 102
Index 232: 1
-----End dump
Swap done
Time slot  16
        Loaded a process at input/proc/s1, PID: 8 PRIO: 0
        CPU 3: Put process  7 to run queue
        CPU 1: Processed  5 has finished
        CPU 2: Put process  4 to run queue
        CPU 3: Dispatched process  8
Increase limit done
        CPU 1: Dispatched process  7
        CPU 2: Dispatched process  4
Time slot  17
        CPU 0: Put process  6 to run queue
        CPU 0: Dispatched process  6
Time slot  18
        CPU 3: Put process  8 to run queue
        CPU 2: Put process  4 to run queue
        CPU 1: Put process  7 to run queue
        CPU 3: Dispatched process  8
        CPU 2: Dispatched process  7
        CPU 1: Dispatched process  4
Time slot  19
        CPU 0: Processed  6 has finished
        CPU 0: Dispatched process  1
TLB miss at read region=1 offset=20
print pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Memory Dump-----
Index 20: 100
Index 64: 102
Index 232: 1
-----End dump
Swap done
Time slot  20
        CPU 3: Put process  8 to run queue
        CPU 2: Put process  7 to run queue
TLB hit at write region=3 offset=20 value=103
print pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
        CPU 1: Processed  4 has finished
00000008: 80000003
00000012: 80000002
Memory Dump-----
Index 20: 100
Index 64: 102
```

Kết quả thực thi test case os_1_mlq_paging (3/4)

```
Index 232: 1
        CPU 1 stopped
Flushing cache:
memphy:   100 | memv: '   21'| pid: '1'
memphy:     0 | memv: '    0'| pid: '5'
memphy:     2 | memv: ' 1002'| pid: '5'
memphy:     1 | memv: '   21'| pid: '5'
        CPU 3: Dispatched process  7
        CPU 2: Dispatched process  8
-----End dump
Swap done
Error
Time slot  21
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Error!
TLB miss at read region=3 offset=20
print pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Memory Dump-----
Index 20: 103
Index 64: 102
Index 232: 1
-----End dump
Swap done
Error
Time slot  22
        CPU 3: Put process  7 to run queue
Invalid!
        CPU 2: Put process  8 to run queue
        CPU 3: Dispatched process  8
        CPU 2: Dispatched process  7
Time slot  23
        CPU 3: Processed  8 has finished
        CPU 0: Processed  1 has finished
        CPU 3 stopped
Flushing cache:
TLB Cache is empty.
Freelist is empty.
        CPU 0 stopped
Flushing cache:
TLB Cache is empty.
Freelist is empty.
Time slot  24
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  7
Time slot  25
Time slot  26
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  7
Time slot  27
        CPU 2: Processed  7 has finished
        CPU 2 stopped
Flushing cache:
TLB Cache is empty.
```

Kết quả thực thi test case os_1_mlq_paging (4/4)

2.2.5 Translation Lookaside Buffer (TLB)

TLB là một phần của bộ nhớ cache trong một hệ thống máy tính, thường được sử dụng trong kiến trúc bộ vi xử lý và hệ điều hành. Chức năng chính của TLB là tăng tốc độ truy xuất bộ nhớ bằng cách lưu trữ các ánh xạ giữa các địa chỉ bộ nhớ ảo (Virtual Mermory) và các địa chỉ bộ nhớ vật lý (MEMPHY). Khi một chương trình yêu cầu truy cập đến một địa chỉ bộ nhớ ảo, TLB sẽ kiểm tra xem liệu ánh xạ tương ứng đã được lưu trong bộ nhớ cache này chưa. Nếu có, TLB sẽ cung cấp địa chỉ bộ nhớ vật lý tương ứng mà không cần truy cập tới bảng trang (page table) của hệ điều hành, giúp tăng tốc độ truy cập và giảm tải cho bộ vi xử lý.

Trong BTL này, chúng ta nghiên cứu về cách cài đặt TLB trong một hệ điều hành, thông qua cơ chế hoạt động của nó (hit/miss, tlbread, tlbwrite, tlballoc, tlbfree).

Trước hết, ta cần cài đặt TLB vào trong hệ điều hành qua 2 struct:

```
struct TLB_node
{
   uint32_t MEMPHY; // respresent physical mermory
   uint32_t MEMVIR; // respresent virtual mermory
   uint32_t pid;    // process ID.
   int id;
   uint32_t indexOfRegister;
   int isWrite; // is TLB cache written? 0:1
   struct TLB_node *next;
   struct TLB_node *prev;
};

struct TLB_cache
{
   int capacity; // number of nodes TLB can hold.
   int num;      // number of nodes TLB has now.
   struct TLB_node *head;
   struct TLB_node *tail;
   struct TLB_node *freehead;
   struct TLB_node *freetail;
};
```

Program 3: *Struct TLB_Node và TLB_cache*

Trong đó:

- **Struct "TLB_node"** là cấu trúc dữ liệu đại diện cho mỗi nút trong TLB. Các thành phần của nó bao gồm:
  - MEMPHY: Địa chỉ bộ nhớ vật lý.

  - MEMVIR: Địa chỉ bộ nhớ ảo.
  - pid: ID của quy trình sử dụng địa chỉ bộ nhớ.
  - id: ID của nút trong TLB (có thể được sử dụng cho mục đích quản lý nút).
  - indexOfRegister: Chỉ số của thanh ghi liên quan đến việc truy cập đến bộ nhớ.
  - isWrite: Chỉ định xem dữ liệu trong TLB đã được ghi hay không (0: chưa được ghi, 1: đã được ghi).
  - next và prev: Con trỏ đến nút tiếp theo và trước đó trong danh sách liên kết.
- **Struct "TLB_cache"** là cấu trúc dữ liệu đại diện cho TLB tổng thể. Các thành phần chính bao gồm:
  - capacity: Số lượng nút tối đa mà TLB có thể chứa.
  - num: Số lượng nút hiện tại trong TLB.
  - head và tail: Con trỏ đến nút đầu tiên và cuối cùng trong danh sách liên kết của TLB.
  - freehead và freetail: Con trỏ đến nút đầu tiên và cuối cùng trong danh sách liên kết của các nút được sử dụng để tái sử dụng trong TLB.

  Cấu trúc trên cho phép quản lý cũng như lưu trữ các thuộc tính liên quan đến việc ánh xạ địa chỉ bộ nhớ ảo sang bộ nhớ vật lý, giúp tăng tốc độ truy cập của CPU.

- **tlb_cache_read**
  - **Mục đích:** Hàm này có trách nhiệm đọc dữ liệu từ bộ đệm TLB dựa trên địa chỉ bộ nhớ ảo (MEMVIR), ID process (pid) được cung cấp, và lưu địa chỉ bộ nhớ vật lý (MEMPHY) tương ứng vào biến được trỏ đến bởi temp.
  - **Thuật toán:**
    * Hàm trước tiên kiểm tra xem bộ đệm TLB có rỗng không. Nếu có, nó in ra thông báo lỗi và trả về -1.
    * Sau đó, nó duyệt qua danh sách liên kết của các nút TLB cho đến khi tìm thấy một nút phù hợp với địa chỉ bộ nhớ ảo, ID quy trình và được đánh dấu là đã ghi (isWrite == 1).
    * Nếu tìm thấy một nút phù hợp, địa chỉ MEMPHY được lưu vào temp.
    * Nếu nút tìm thấy không phải là nút đầu tiên, nó sẽ được di chuyển đến đầu của bộ đệm TLB để tối ưu hóa việc truy cập sau này.
    * Cuối cùng, hàm trả về 0 để biểu thị việc thu thập địa chỉ bộ nhớ vật lý thành công, hoặc -1 nếu dữ liệu không được tìm thấy trong bộ đệm TLB.
- **tlb_cache_write**
  - **Mục đích: Hàm này có trách nhiệm đọc dữ liệu từ bộ đệm TLB dựa trên địa chỉ bộ nhớ ảo (MEMVIR), ID process (pid) được cung cấp, và lưu địa chỉ bộ nhớ vật lý (MEMPHY) tương ứng vào biến được trỏ đến bởi temp.**
  - **Thuật toán:**

    * Tương tự như tlb_cache_read, tlb_cache_write nó duyệt qua danh sách liên kết của các nút TLB để tìm một nút phù hợp với địa chỉ bộ nhớ ảo, ID quy trình và được đánh dấu là đã ghi (isWrite == 1).
    * Nếu tìm thấy một nút phù hợp, nó cập nhật địa chỉ bộ nhớ vật lý (MEMPHY) với giá trị được cung cấp và di chuyển nút đó đến đầu của bộ đệm TLB để tối ưu hóa.
    * Nếu không tìm thấy nút phù hợp, một nút trống (isWrite == 0) được tái sử dụng, hoặc thay thế nút ít được sử dụng nhất trong bộ đệm TLB.
    * Cuối cùng, hàm trả về 0 để chỉ ra việc ghi dữ liệu vào bộ đệm TLB thành công, hoặc -1 nếu thất bại.

- **tlballoc** được sử dụng để cấp phát một nút mới trong bộ đệm TLB để lưu trữ ánh xạ từ địa chỉ bộ nhớ ảo đến địa chỉ bộ nhớ vật lý bằng cách tạo ra một nút TLB mới với các thông số được chỉ định.

- **tlbwrite** được sử dụng để cập nhật hoặc ghi thông tin ánh xạ mới vào bộ đệm TLB, đảm bảo TLB được duy trì theo một cơ chế đệm hợp lý nhằm tối ưu hóa hiệu suất truy cập.

- **tlbread** dùng để đọc thông tin ánh xạ từ bộ nhớ ảo sang bộ nhớ vật lý từ TLB cache.

- **tlbfree** được sử dụng để giải phóng TLB cache khi không cần thiết nữa.

Kết quả kiểm thử khi chạy ./os tlb:

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s1, PID: 1 PRIO: 3
Time slot   1
        CPU 0: Dispatched process  1
Time slot   2
        Loaded a process at input/proc/s2, PID: 2 PRIO: 2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/p0s, PID: 3 PRIO: 4
Time slot   4
        Loaded a process at input/proc/p0s, PID: 4 PRIO: 4
Time slot   5
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   6
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  10
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  14
Time slot  15
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
Time slot  16
Time slot  17
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  18
Time slot  19
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  20
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  3
Time slot  21
Increase limit done
```

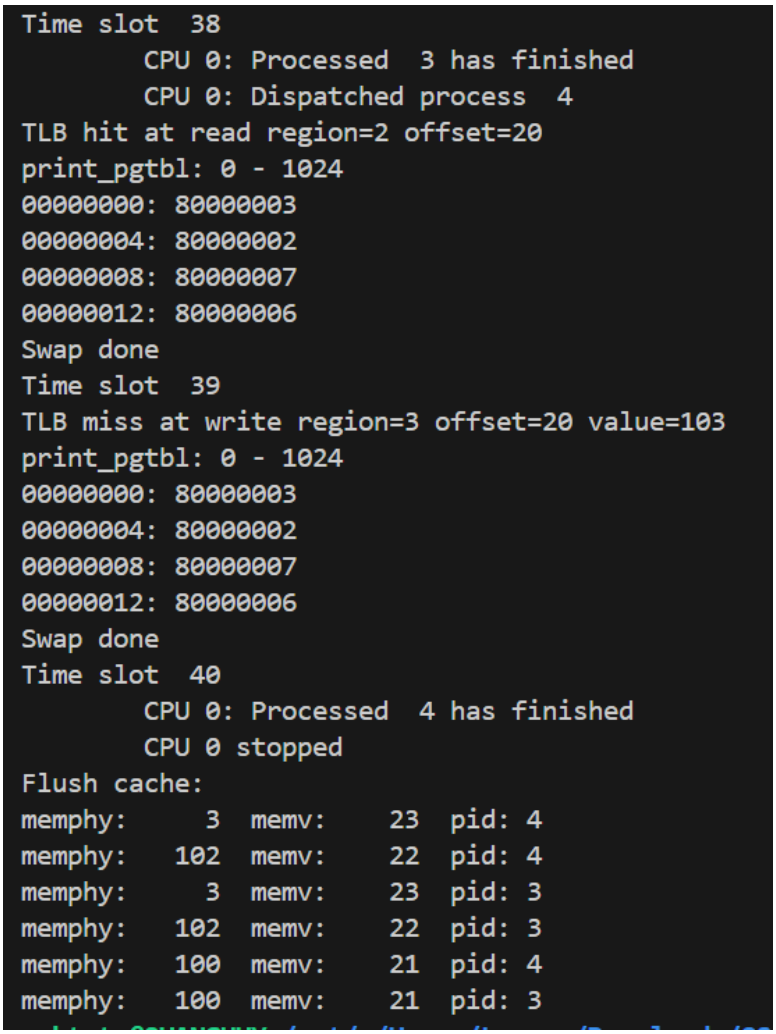
Kết quả kiểm thử khi chạy ./os tlb (1/4)

```
Time slot  22
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  23
Increase limit done
Time slot  24
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Increase limit done
Time slot  25
Free done
Time slot  26
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Increase limit done
Time slot  27
Free done
Time slot  28
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
Time slot  29
TLB miss at write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000005
00000012: 80000004
Swap done
Time slot  30
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  31
TLB miss at write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000003
00000004: 80000002
00000008: 80000007
00000012: 80000006
Swap done
Time slot  32
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
TLB hit at read region=1 offset=20
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000005
00000012: 80000004
Swap done
```

Kết quả kiểm thử khi chạy ./os tlb (2/4)

```
Time slot  33
TLB miss at write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000005
00000012: 80000004
Swap done
Time slot  34
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
TLB hit at read region=1 offset=20
print_pgtbl: 0 - 1024
00000000: 80000003
00000004: 80000002
00000008: 80000007
00000012: 80000006
Swap done
Time slot  35
TLB miss at write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000003
00000004: 80000002
00000008: 80000007
00000012: 80000006
Swap done
Time slot  36
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  3
TLB hit at read region=2 offset=20
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000005
00000012: 80000004
Swap done
Time slot  37
TLB miss at write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000005
00000012: 80000004
Swap done
```

Kết quả kiểm thử khi chạy ./os tlb (3/4)



Kết quả kiểm thử khi chạy ./os tlb (4/4)

2.3 Config

- **Sử dụng cho các testcase Sched_0, Sched_1 và os_1_singleCPU_mlq.**

```
#ifndef OSCFG_H
#define OSCFG_H

#define MLQ_SCHED 1
#define MAX_PRIO 140

#define CPU_TLB
#define MM_PAGING
#define CPUTLB_FIXED_TLBSZ
#define MM_FIXED_MEMSZ
#define VMDBG 1
#define MMDBG 1
#define IODUMP 1
#define PAGETBL_DUMP 1

#endif
```

Hình 2.3: *Config dùng để đọc process thông thường*

- **Sử dụng cho các testcase os_0_mlq_paging, os_1_mlq_paging, os_1_mlq_paging_small_1K, os_1_paging_small_4K và os_1_singleCPU_mlq_paging.**

```
#ifndef OSCFG_H
#define OSCFG_H

#define MLQ_SCHED 1
#define MAX_PRIO 140

#define CPU_TLB
#define MM_PAGING
#define CPUTLB_FIXED_TLBSZ
// #define MM_FIXED_MEMSZ
#define VMDBG 1
#define MMDBG 1
#define IODUMP 1
#define PAGETBL_DUMP 1

#endif
```

Hình 2.4: *Config dùng để đọc input có khai báo vùng nhớ ram và swap*

- **Sử dụng cho các testcase os_1_tlb_singleCPU_mlq.**

```
#ifndef OSCFG_H
#define OSCFG_H

#define MLQ_SCHED 1
#define MAX_PRIO 140

#define CPU_TLB
#define MM_PAGING
// #define CPUTLB_FIXED_TLBSZ
#define MM_FIXED_MEMSZ
#define VMDBG 1
#define MMDBG 1
#define IODUMP 1
#define PAGETBL_DUMP 1

#endif
```

Hình 2.5: *Config dùng để đọc input có khai báo vùng nhớ tlb*

- **Sử dụng cho các testcase sched_0, sched_1.**

  * Có thể bật OPTION 1 nếu input có 3 giá trị ở mỗi process.

```
src > C os.c > read_config(const char *)
173  static void read_config(const char * path) {
232          strcat(ld_processes.path[i], "input/proc/");
233          char proc[100];
234  #ifdef MLQ_SCHED
235
236  // /*OPTION_1*/ fscanf(file, "%lu %s %lu\n", &ld_processes.start_time[i], proc, &ld_processes.prio[i]
237  /*OPTION_2*/ fscanf(file, "%lu %s\n", &ld_processes.start_time[i], proc);
238  #else
```

Hình 2.6: *Config dùng để đọc các input không có độ ưu tiên*

2.4 Tổng hợp toàn bộ hệ thống

- Khi hiện thực scheduler và memory, nhóm đã hiện thực đồng bộ tại các nơi có thể xảy ra hiện tượng *race condition* trong hệ thống khi chạy bộ định thời và bộ nhớ.
- Kết quả chạy test case phù hợp với lý thuyết cho thấy không có hiện tượng bị sung đột trong hệ thống. Vậy nhóm đã hiện thực thành công việc đồng bộ trong toàn hệ thống.

3 Trả lời các câu hỏi

3.1 Ưu và nhược điểm của giải thuật được sử dụng trong bài so với các giải thuật khác đã học

Trong bài làm, **priority queue** là giải thuật đã được sử dụng.Ta có một vài so sánh ưu và nhược điểm của priority queue so với các giải thuật khác như sau:

1. First In First Out (FIFO)
    - Lợi ích:
        - FIFO không bao gồm bất kỳ logic phức tạp nào, khá đơn giản và dễ thực hiện.
        - Mỗi process đều có cơ hội để thực hiện, vì thế hiện tượng starvation sẽ không xảy ra.
    - Bất lợi:
        - Các process có thời gian thực hiện ít bị tác động, tức là thời gian chờ đợi thường khá lâu.
        - Ưu tiên process liên kết CPU sau đó ưu tiên process liên kết I/O.
        - Process đầu tiên sẽ nhận CPU trước, các process khác chỉ có thể nhận CPU sau khi process hiện tại kết thúc quá trình thực thi. Bây giờ, giả sử process đầu tiên có brust time lớn và các process khác có brust time ít hơn, thì các process sẽ phải chờ nhiều hơn một cách không cần thiết, điều này sẽ dẫn đến thời gian chờ trung bình nhiều hơn (hiệu ứng truyền tải).
        - Hiệu ứng này dẫn đến việc sử dụng CPU và thiết bị thấp hơn.
        - Thuật toán FIFO đặc biệt rắc rối đối với các hệ thống chia sẻ thời gian, trong đó điều quan trọng là mỗi người dùng phải nhận được một phần CPU theo định kỳ.
2. Round Robin (RR)
    - Lợi ích:
        - Mọi process đều nhận được một phần bằng nhau của CPU.
        - RR hoạt động tương tự như FCFS, nhưng quyền ưu tiên được thêm vào để cho phép hệ thống chuyển đổi giữa các process. RR phân bổ CPU cho từng process trong một lượng thời gian, vì vậy tất cả các process đều được ưu tiên như nhau.
        - Nếu process không thoát khỏi CPU trước khi thời gian của nó hết hạn, thì process đó sẽ được ưu tiên và được đưa trở lại ready_queue, một process khác được lên lịch để chạy trong một lượng thời gian.
        - Starvation không xảy ra vì đối với mỗi chu kỳ quay vòng, mọi process đều có thời gian cố định để thực hiện, không có process nào bị bỏ lại phía sau.
    - Bất lợi:
        - Thời gian chờ trung bình của RR thường dài.
        - Thông lượng trong RR phần lớn phụ thuộc vào việc lựa chọn độ dài của thời gian. Nếu

thời gian dài hơn mức cần thiết, nó có xu hướng hoạt động giống như FCFS.

- Nếu định lượng thời gian ngắn hơn mức cần thiết, số lần CPU chuyển từ process này sang process khác sẽ tăng lên. Điều này dẫn đến giảm hiệu suất của CPU.

3. Priority Queue

- Lợi ích:
  - Giải thuật Priority Queue linh hoạt hơn, cho phép các process khác nhau di chuyển giữa các hàng đợi khác nhau.
  - Cho phép aging, do đó không bị starvation.
  - Priority Scheduling cung cấp một cơ chế tốt trong đó priority của mỗi process có thể được xác định chính xác.
- Bất lợi:
  - Đây cũng là thuật toán phức tạp nhất, vì việc xác định bộ lập lịch tốt nhất yêu cầu một số phương tiện để chọn giá trị cho tất cả các tham số:
    * Số lượng hàng đợi.
    * Thuật toán scheduling cho mỗi hàng đợi.
    * Phương pháp được sử dụng để xác định thời điểm nâng cấp một process lên hàng đợi có mức ưu tiên cao hơn.
    * Phương pháp được sử dụng để xác định khi nào hạ cấp một process xuống hàng đợi có mức ưu tiên thấp hơn.
    * Phương pháp được sử dụng để xác định process sẽ vào hàng đợi nào khi quá trình đó cần thực thi.

Từ những phân tích tổng quan các giải thuật ở trên, chúng ta có thể nhận thấy rằng mỗi thuật toán sẽ được sử dụng hiệu quả trong một số trường hợp nhất định. Tuy nhiên, chúng ta cũng có thể nhận ra rằng Priority Queue có nhiều ưu điểm và tính linh hoạt hơn. Nó khắc phục được nhược điểm của các thuật toán khác như hiệu ứng Convey. Nó cũng chọn các quy trình cần được thực thi trước và có thể đợi các quy trình khác hoàn thành.
Ngoài ra, ta có thể kết hợp nhiều thuật toán định thời khác vào để phát huy được ưu điểm của chúng như Round Robin, First Come First Serve (FCFS),...

3.2 Ưu điểm và nhược điểm của thiết kế nhiều phân khúc (multiple segments)

Có một số lợi ích như sau:

- Quản lý bộ nhớ hiệu quả và tránh tình trạng mảng bên ngoài (external fragmentation): Quá trình sẽ được chia thành một số đoạn có các mục đích khác nhau. Tuy nhiên, điều này không ảnh hưởng đến toàn bộ chương trình. Do đó, tối ưu hóa việc sử dụng bộ nhớ được đảm bảo và giảm thiểu nguy cơ mảng bộ nhớ.
- Bảo vệ bộ nhớ: Hệ điều hành có thể thực hiện bảo vệ bộ nhớ khi có nhiều đoạn bộ nhớ bằng cách cung cấp mỗi đoạn một mức độ truy cập riêng biệt. Ví dụ, hệ điều hành có thể chỉ định một đoạn

chỉ để đọc cho mã và một đoạn để đọc và ghi cho dữ liệu, ngăn chặn sự sửa đổi mã không cố ý hoặc độc hại. Điều này giúp người dùng bảo vệ dữ liệu quan trọng khỏi truy cập hoặc sửa đổi trái phép.

- Tính linh hoạt: Hệ điều hành dễ dàng tạo ra các đoạn mới khi cần thiết để đáp ứng các loại dữ liệu và yêu cầu bộ nhớ khác nhau (mở rộng nếu cần), và cũng có thể loại bỏ các đoạn không còn cần thiết nữa. Cách kiểm tra lỗi hoặc lỗi có thể chỉ xảy ra trong một đoạn mà không ảnh hưởng đến các đoạn khác, tăng tính ổn định và đáng tin cậy của các hệ điều hành.

3.3 Ưu điểm và nhược điểm của việc chia địa chỉ thành nhiều hơn 2 cấp

Nếu chúng ta chia địa chỉ thành nhiều hơn 2 cấp độ trong hệ thống quản lý bộ nhớ phân trang, thì một số lợi ích chính của việc triển khai nó có thể nhìn thấy như một cấu trúc dữ liệu cây với nhiều cấp độ. Mỗi cấp độ có bảng trang riêng của nó, chứa địa chỉ của các bảng trang cấp độ tiếp theo hoặc các địa chỉ bộ nhớ vật lý. Điều này dẫn đến kiểm soát chính xác về việc ánh xạ vì mỗi bảng trang chỉ cần lưu trữ các địa chỉ cho một phần nhỏ không gian địa chỉ ảo.
Tuy nhiên, có một số nhược điểm khi sử dụng hơn 2 cấp độ trong quản lý bộ nhớ phân trang:

- Tăng độ phức tạp và chi phí: càng có nhiều bảng trang cần quản lý, hệ điều hành cần sử dụng càng nhiều bộ nhớ và công suất xử lý để duy trì các bảng trang và thực hiện các bước dịch chuyển cần thiết giữa địa chỉ ảo và vật lý.
- Mối đe dọa về bảo mật tăng lên: Với nhiều bảng trang hơn để quản lý, việc xác định và giảm thiểu các loại tấn công như tràn bộ đệm (buffer overflow) hoặc làm nhiễu bảng trang (page table poisoning) có thể trở nên khó khăn hơn đối với hệ điều hành.

3.4 Ưu điểm và nhược điểm của phương pháp segmentation with paging

Phương pháp Segmentation with paging là sự kết hợp giữa hai phương pháp segmentation và paging. Do đó, để có thể thấy rõ được ưu điểm và nhược điểm của segmentation with paging, trước tiên ta cần phân tích hai phương pháp nêu trên

1. Paging

   Phân trang là một khái niệm logic cho phép Hệ điều hành truy xuất các quy trình vào bộ nhớ chính từ bộ nhớ phụ. Nó cung cấp khả năng truy cập dữ liệu nhanh hơn khi các quy trình được lưu trữ dưới dạng các trang.

   Bộ nhớ hệ thống chính được chia thành các khung, tức là các khối bộ nhớ vật lý nhỏ có kích thước cố định trong đó kích thước của các khung bằng với kích thước của các trang. Điều này cung cấp khả năng sử dụng tối đa bộ nhớ hệ thống chính và giúp tránh phân mảnh ngoại.

   - Ưu điểm:
     - Thuật toán quản lý bộ nhớ dễ sử dụng.
     - Hạn chế phân mảnh ngoại.

  - Trao đổi giữa các trang có kích thước bằng nhau và khung trang trở nên dễ dàng hơn.
- Nhược điểm:
  - Phân mảnh nội.
  - Mức tiêu tốn bộ nhớ tăng theo theo bảng Trang.
  - Chi phí tham chiếu bộ nhớ cao do phân trang đa cấp (multi-level paging).

2. Segmentation

Phân đoạn tương tự như phân trang, ngoại trừ độ dài của các đoạn có thể thay đổi và các trang có kích thước cố định. Phân đoạn của chương trình bao gồm chức năng chính của chương trình, cấu trúc dữ liệu, chức năng tiện ích, v.v.

Tất cả thông tin này về các process được hệ điều hành duy trì dưới dạng bảng bản đồ phân đoạn (segment map table). Bảng này bao gồm danh sách số phân đoạn, khối bộ nhớ trống, kích thước và vị trí bộ nhớ của chúng trong bộ nhớ chính hoặc bộ nhớ ảo.

- Ưu điểm:
  - Cung cấp sự bảo vệ trong các phân đoạn.
  - Các phân đoạn tham chiếu nhiều process có thể giúp các process chia sẻ bộ nhớ.
  - Không tồn tại phân mảnh nội.
  - So với phân trang, bảng phân đoạn sử dụng ít bộ nhớ hơn.
- Nhược điểm:
  - Việc tách không gian bộ nhớ trống thành các phần nhỏ có thể gây ra phân mảnh ngoại.
  - Tốn kém về mặt chi phí.

Có thể thấy mỗi phương pháp đều có những ưu điểm và nhược điểm của riêng mình. Phương pháp segmentation with paging là sự kết hợp giữa cả hai phương pháp, do đó nó khắc phục được những nhược điểm còn tồn tại kể trên, tuy vậy nó vẫn tồn tại những mặt hạn chế của riêng mình:

- Ưu điểm:
  - Giảm thiểu hao phí bộ nhớ, khắc phục việc bảng phân trang có kích thước quá lớn.
  - Xử lý triệt để vấn đề phân mảnh ngoại.
  - Đơn giản hóa việc cấp phát vùng nhớ cho các process.
  - Bảo vệ dữ liệu thông qua cấu trúc nhiều lớp.
- Nhược điểm:
  - Còn tồn tại phân mảnh nội.
  - Mức độ phức tạp sẽ cao hơn nhiều so với phân trang.
  - Các bảng trang cần được lưu trữ liên tục trong bộ nhớ.

Mỗi phương pháp đều có cho mình những ưu điểm và nhược điểm riêng, không có phương pháp

nào là hoàn hảo cho mọi trường hợp. Do đó, trong thực tế chúng ta cần phải xem xét chọn lựa phương pháp cho phù hợp với điều kiện thực tế của hệ thống mà chúng ta đang làm việc.

3.5 Nếu hệ thống đa lõi có mỗi lõi CPU có thể chạy trong ngữ cảnh khác nhau, và mỗi lõi có riêng MMU của nó và một phần của lõi (TLB), thì điều gì sẽ xảy ra? Trong CPU hiện đại, TLB 2 cấp đã trở nên phổ biến, tác động của cấu hình phần cứng bộ nhớ mới này đối với các kế hoạch dịch của chúng ta là gì?

Hệ thống đa lõi với MMU và TLB riêng biệt: Ảnh hưởng đến lược đồ dịch

Hệ thống đa lõi với MMU và TLB riêng biệt cho mỗi nhân CPU mang đến nhiều lợi ích cho hiệu suất quản lý bộ nhớ ảo. Việc triển khai này có tác động đáng kể đến lược đồ dịch, cải thiện hiệu quả và tốc độ truy cập bộ nhớ.

Lợi ích:

- **Mỗi nhân hoạt động độc lập:** Mỗi nhân có thể chạy một tiến trình riêng biệt với không gian địa chỉ ảo riêng. MMU của mỗi nhân thực hiện dịch địa chỉ ảo sang địa chỉ bộ nhớ vật lý tương ứng. TLB lưu trữ các bản dịch gần đây để tăng tốc quá trình này.

- **TLB hai cấp:** Cấu trúc phân cấp này tối ưu hóa hiệu quả dịch địa chỉ:

  - TLB cấp 1 (TLB1): Kích thước nhỏ, tốc độ truy cập nhanh, lưu trữ các bản dịch truy cập thường xuyên.

  - TLB cấp 2 (TLB2): Kích thước lớn hơn, tốc độ truy cập chậm hơn, lưu trữ nhiều bản dịch hơn.

- **Giảm thời gian tra cứu:** Kích thước nhỏ của TLB1 giúp tra cứu nhanh hơn so với TLB đơn lớn hơn. Nhờ vậy, thời gian trung bình để tìm bản dịch được rút ngắn đáng kể.

- **Tăng tỷ lệ trúng cache:** Hệ thống hai cấp cho phép lưu trữ nhiều bản dịch hơn. Khả năng tìm thấy bản dịch cần thiết trong TLB (trúng cache) cao hơn, tránh quá trình truy cập bảng trang chậm hơn trong bộ nhớ.

- **Nâng cao hiệu suất tổng thể:** Việc tra cứu nhanh hơn và tỷ lệ trúng cache cao hơn dẫn đến dịch địa chỉ ảo sang địa chỉ bộ nhớ vật lý nhanh hơn, giúp chương trình thực thi mượt mà và hệ thống phản hồi tốt hơn.

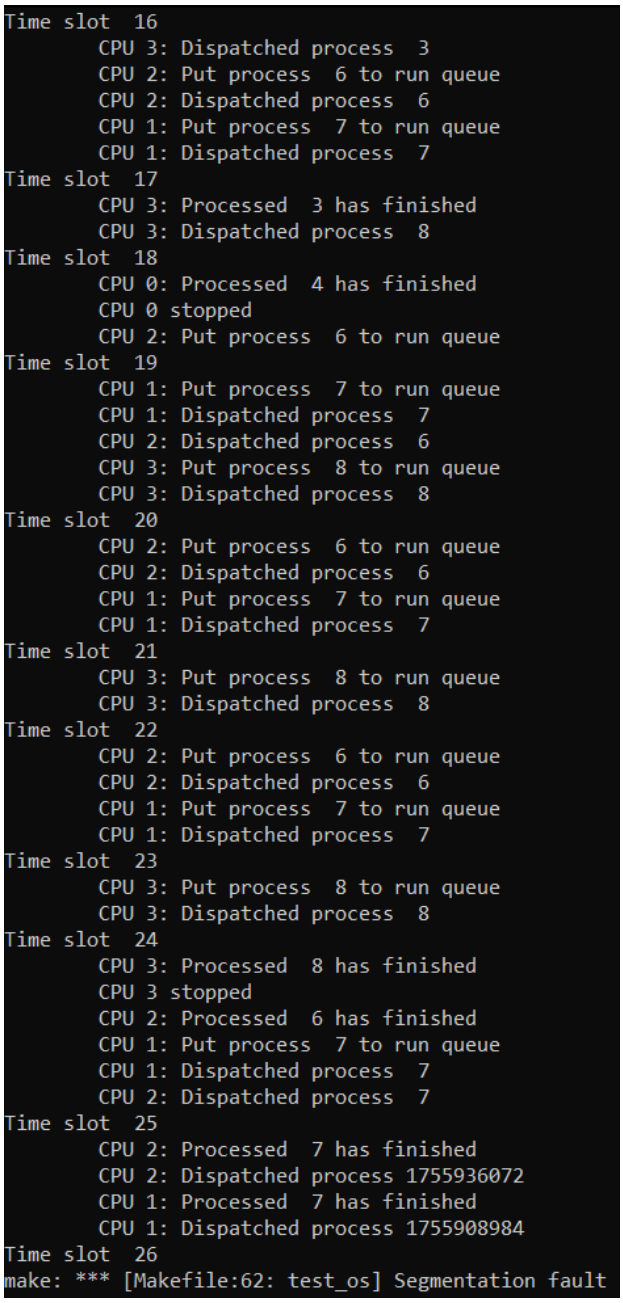
- **Lưu ý:** Kích thước và cấu trúc cụ thể của các cấp TLB có thể thay đổi tùy theo kiến trúc CPU và mục tiêu thiết kế.Mặc dù TLB hai cấp phổ biến, một số CPU có thể có nhiều cấp hơn để tối ưu

hóa hơn nữa.

Nhìn chung, hệ thống đa lõi với MMU và TLB riêng biệt cho mỗi nhân mang lại giải pháp quản lý bộ nhớ ảo hiệu quả trong nền tảng máy tính hiện đại. Cách tiếp cận phân cấp này giúp giảm thời gian tra cứu, tăng tỷ lệ trúng cache và ultimately nâng cao hiệu suất tổng thể của hệ thống.

3.6 Hiện tượng khi hệ thống không được xử lý vấn đề đồng bộ hóa

Nếu không sử dụng kỹ thuật đồng bộ hóa, các hàm trong chương trình khi chạy thông qua các thread sẽ xảy ra xung đột, dẫn tới việc cấp phát và bộ nhớ không được luân phiên tại cùng 1 địa chỉ bộ nhớ. Điều này có thể gây ra xung đột khi 2 process đi vào cùng 1 vùng địa chỉ bộ nhớ hoặc cùng thực hiện giải phóng vùng nhớ (deallocate) tại cùng 1 địa chỉ dẫn tới lỗi Segmentation fault.



Hình 3.1: *Minh họa hiện tượng khi không sử dụng đồng bộ hóa*

Tài liệu

[1] https://www.geeksforgeeks.org/paged-segmentation-and-segmented-paging/

[2] https://www.javatpoint.com/os-segmented-paging

[3] https://data-flair.training/blogs/paging-vs-segmentation-in-operating-system/

[4] https://www.geeksforgeeks.org/multilevel-queue-mlq-cpu-scheduling/