**ĐẠI HỌC QUỐC GIA TPHCM**

**TRƯỜNG ĐẠI HỌC BÁCH KHOA**

****



**BÁO CÁO BÀI TẬP LỚN**

**Môn: OPERATION SYSTEM**

**Đề tài: SIMPLE OPERATION SYSTEM**

**GVHD: NGUYỄN THANH QUÂN**

**Lớp: L03**

|  |  |  |  |
| --- | --- | --- | --- |
| **STT** | **MSSV** | **Họ và tê** | **Nhiệm vụ được phân công** |
| **1** | **2333021** | **Phạm Nguyễn Đức Thành** | **Thiết kế slide và nội dung cho bài báo cáo** |
| **2** | **2114485** | **Võ Mai Phương** | **Thực hiện nội dung phần định thời** |
| **3** | **2212607** | **Dương Hồ Hoàng Phúc** | **Thực hiện nội dung phần quản lý bộ nhớ** |

***Thành Phố Hồ Chí Minh, Tháng 12-2023***

Mục lục

[**I.** **Chính sách về định thời:** 3](#_Toc153830484)

[**1.** **Tổng quan:** 3](#_Toc153830485)

[**2.** **Hiện thực:** 3](#_Toc153830486)

[**3.** **Phân tích kết quả:** 7](#_Toc153830487)

[**3.1.** **Kết quả output 2:** 7](#_Toc153830488)

[**4.** **Trả lời câu hỏi:** 11](#_Toc153830489)

[**II.** **Quản lý bộ nhớ** 12](#_Toc153830490)

[**1.** **Cơ chế quản lý bộ nhớ của hệ điều hành đơn giản:** 12](#_Toc153830491)

[**2.** **Hiện thực** 13](#_Toc153830492)

[**2.1.** **Cấp phát bộ nhớ:** 13](#_Toc153830493)

[**2.2.** **Thu hồi bộ nhớ:** 14](#_Toc153830494)

[**2.3.** **Đọc/ghi bộ nhớ:** 15](#_Toc153830495)

[**3.** **Phân tích kết quả kiểm thử:** 18](#_Toc153830496)

[**4.** **Trả lời câu hỏi:** 24](#_Toc153830497)

1. **Chính sách về định thời:**
2. **Tổng quan:**

Chính sách định thời mà hệ điều hành sử dụng là Multi Level Queue (MLQ) kết hợp Round Robin. Trong hệ thống, mỗi process có một giá trị prio (độ ưu tiên) nhất định thể hiện độ ưu tiên của một process (tiến trình) và được dùng để xác định ready\_queue (hàng đợi sẵn sàng) nào sẽ được dùng để chứa process đó, prio càng thấp thể hiện cho độ ưu tiên càng cao.

Ban đầu các hàng đợi có độ ưu tiên khác nhau sẽ có số lần sử dụng CPU khác nhau (thể hiện qua giá trị slot, tính bằng công thức MAX\_PRIO - prio), hàng đợi có độ ưu tiên càng cao thì càng có nhiều lượt sử dụng CPU.Hàng đợi có độ ưu tiên cao nhất sẽ được chọn để thi thực trước cho đến khi đã sử dụng hết số slot cho phép.

Lúc này, CPU sẽ chuyển sang thực thi các process trong hàng đợi có độ ưu tiên thấp hơn. Tất cả hàng đợi sẽ được áp dụng cơ chế Round Robin, tức là việc thực thi một hàng đợi chính xác là thay phiên nhau thực thi các process trong hàng đợi ấy theo time slice cho trước, một lần thay phiên được tính là một lần sử dụng CPU.

Đến một thời điểm mà tất cả các hàng đợi đều sử dụng hết số slot ban đầu thì hệ thống sẽ cấp phát lại số slot cho từng hàng đợi theo công thức ban đầu sau đó lại tiếp thực hiện công việc định thời.

1. **Hiện thực:**

Trước hết, ta định nghĩa cấu trúc của một hàng đợi (queue) gồm ba thuộc tính là mảng lưu trữ các tiến trình, số lượng các tiến trình trong hàng đợi hiện tại và vị trí mới tiếp theo trong mảng mà tiến trình có thể được thêm vào hoặc lấy ra.

struct queue\_t {

    struct pcb\_t \* proc[MAX\_QUEUE\_SIZE];

    int size;

    int slot;

};

Trong file sched.c, ta bổ sung thêm file sched.h, hai hàm dequeue và enqueue như sau:

void enqueue(struct queue\_t \* q, struct pcb\_t \* proc) {

        /\* TODO: put a new process to queue [q] \*/

    if (q->size == MAX\_QUEUE\_SIZE) return;

    q->proc[q->size] = proc;

    q->size++;

}

struct pcb\_t \* dequeue(struct queue\_t \* q) {

    #ifdef MLQ\_SCHED

    // Return process at index = 0;

    if (empty(q)) return NULL;

    struct pcb\_t \* proc = q->proc[0];

    for (int i = 0; i < q->size - 1; i++) {

        q->proc[i] = q->proc[i + 1];

    }

    q->size--;

    q->slot--;

    return proc;

    #else

    /\* TODO: return a pcb whose prioprity is the highest

    \* in the queue [q] and remember to remove it from q

    \* \*/

    if (empty(q)) return NULL;

    if (q->size == 1) {

        q->size--;

        return q->proc[0];

    }

    int min\_priority = q->proc[0]->priority;

    int rt\_index = 0;

    for (int i = 1; i < q->size; i++) {

        if (q->proc[i]->priority < min\_priority) {

            min\_priority = q->proc[i]->priority;

            rt\_index = i;

        }

    }

    for (int i = rt\_index; i < q->size - 1; i++) {

        q->proc[i] = q->proc[i + 1];

    }

    q->size--;

    return q->proc[rt\_index];

    #endif

}

Hàm enqueue có nhiệm vụ đưa tiến trình mới đến vào cuối hàng đợi

Hàm dequeue thì thực hiện theo hai trường hợp:

* Đầu tiên, nếu có sử dụng module MLQ\_SCHED, dequeue sẽ lấy ra tiến trình đầu tiên trong hàng đợi và trừ đi 1 lần sử dụng của hàng đợi đó. Khi lấy ra tiến trình ở đầu và đặc một tiến trình mới vào cuối là do hàng đợi này đang áp dụng Round Robin.
* Ngược lại nếu không sử dụng module MLQ\_SCHED, hệ thống sẽ áp dụng chính sách định thời cho hàng đợi ưu tiên, khi này hàm dequeue sẽ trả về tiến trình có độ ưu tiên cao nhất (dựa theo giá trị của thuộc tính priority của tiến trình được lưu trong pcb\_t).

Kế đến, ta sẽ sửa lại hàm init\_scheduler() và hoàn thiện hàm get\_mlq\_proc() như sau:

void init\_scheduler(void) {

#ifdef MLQ\_SCHED

    int i ;

    for (i = 0; i < MAX\_PRIO; i ++) {

        mlq\_ready\_queue[i].size = 0;

        mlq\_ready\_queue[i].slot = MAX\_PRIO - i;

    }

#endif

    ready\_queue.size = 0;

    run\_queue.size = 0;

    pthread\_mutex\_init(&queue\_lock, NULL);

}

struct pcb\_t \* get\_mlq\_proc(void) {

    struct pcb\_t \* proc = NULL;

    pthread\_mutex\_lock(&queue\_lock);

    int iterator;

    int num\_empty\_queue = 0;

    for (iterator = 0; iterator < MAX\_PRIO; iterator++) {

        if (!empty(&mlq\_ready\_queue[iterator])) {

            if (mlq\_ready\_queue[iterator].slot > 0) {

                proc = dequeue(&mlq\_ready\_queue[iterator]);

                break;

            }

        }

        else num\_empty\_queue++;

    }

    if (num\_empty\_queue == MAX\_PRIO) goto rt\_proc;

    if (iterator == MAX\_PRIO) {

        for (int i = 0; i < MAX\_PRIO; i++) {

            mlq\_ready\_queue[i].slot = MAX\_PRIO - i;

        }

        iterator = 0;

        while (iterator < MAX\_PRIO) {

            if (!empty(&mlq\_ready\_queue[iterator])) {

                proc = dequeue(&mlq\_ready\_queue[iterator]);

                break;

            }

            iterator++;

        }

    }

    rt\_proc:

    pthread\_mutex\_unlock(&queue\_lock);

    return proc;

}

Trong trường hợp có module MLQ\_SCHED, hàm init\_scheduler() sẽ thực hiện thêm việc khởi tạo các giá trị cho kích thước và slot của tất cả các hàng đợi theo công thức cho trước.

Đối với hàm get\_mlg\_proc(), việc thực hiện lấy ra 1 tiến trình từ hàng đợi mlq\_ready\_queue. Đầu tiên một con trỏ lần lượt duyệt qua các hàng đợi, bắt đầu từ hàng đợi có giá trị ưu tiên cao nhất(prio = 0), nếu lúc này nó không tìm thấy giá trị hợp lệ là không được rỗng và giá trị của thuộc tính slot phải > 0 ( số lần sử dụng CPU) thì ta mới sử dụng hàm dequeue cho tiến trình đó và kết thúc hàm.

Tuy nhiên, sau khi đã duyệt qua hết tất cả các hàng đợi mà vẫn không tìm thấy hàng đợi hợp lệ thì ta tiếp tục xét các trường sau:

* Nếu tất cả các hàng đều rỗng, ta sẽ kết thúc hàm bằng cách trả về NULL.
* Ngược lại, tức là tất cả hàng đợi đều có số slot = 0 thì ta khởi tạo lại số slot và thực hiện lại quá trình duyệt hàng và dequeue cho tiến này.

1. **Phân tích kết quả:**
   1. **Kết quả output 2:**

Test case được thực hiện trong tệp "os\_mlq\_2" có nội dung như sau:

**2 2 8**

**1 s4 4**

**2 s3 3**

**4 m1 2**

**6 s2 3**

**7 m0 3**

**9 p1 2**

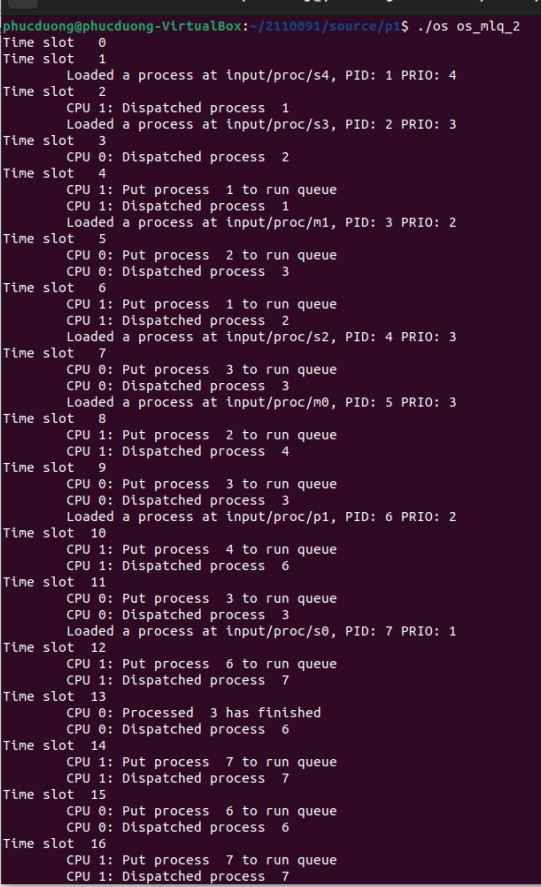
**11 s0 1**

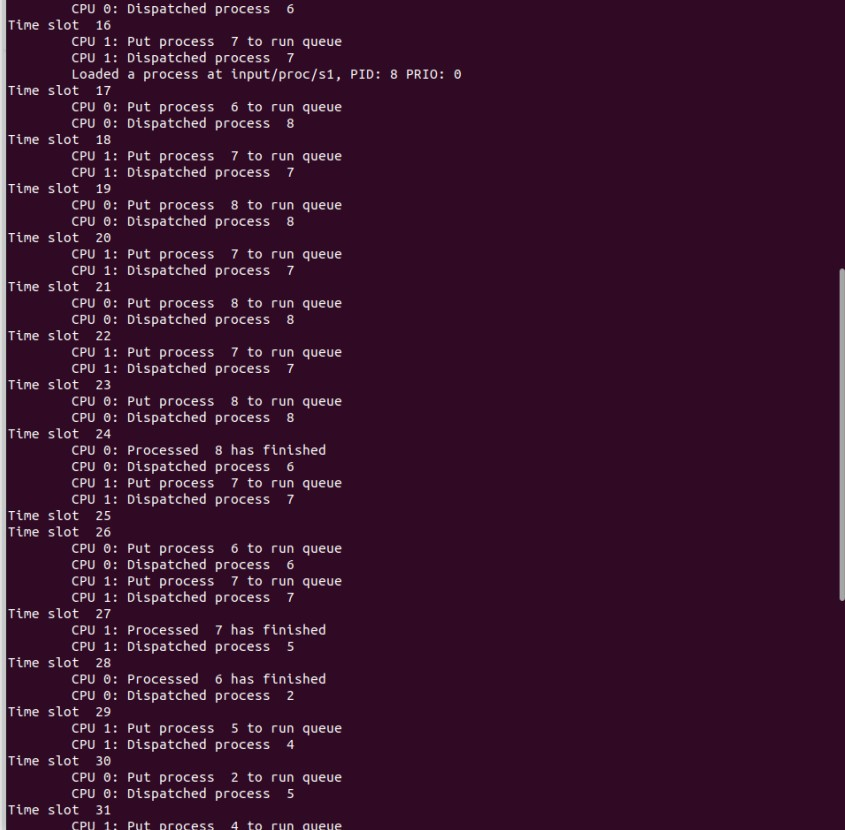
**16 s1 0**

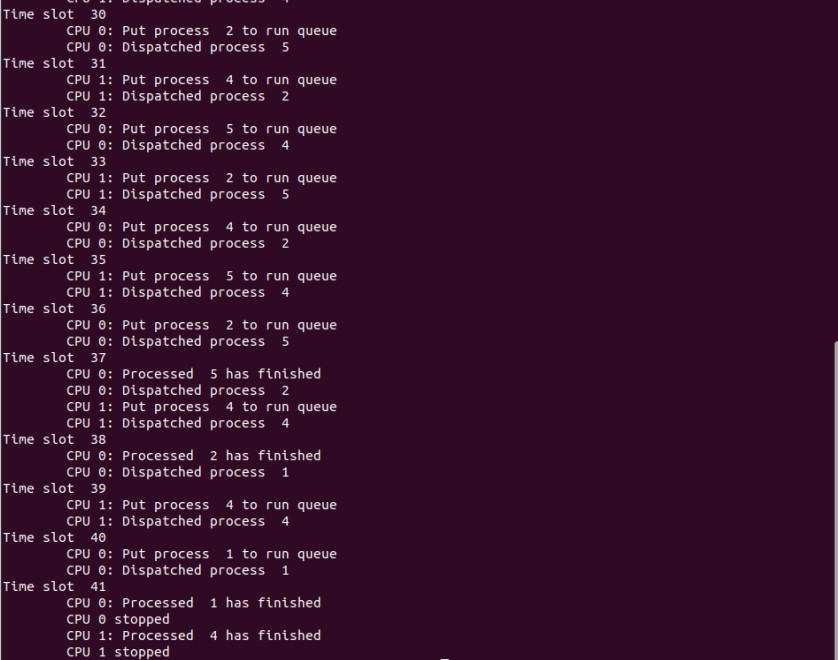
Trong test case này, hệ điều hành sử dụng 2 CPU áp dụng Round Robin với time slice = 2 để định thời 8 tiến trình. Thông tin tổng quan của 8 tiến trình này như sau:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Tiến trình** | **Số chu kì thực thi** | **Độ ưu tiên** | **Thời điểm đến** |
| **1** | **s4** | **7** | **4** | **1** |
| **2** | **s3** | **11** | **3** | **2** |
| **3** | **m1** | **8** | **2** | **4** |
| **4** | **s2** | **12** | **3** | **6** |
| **5** | **m0** | **7** | **3** | **7** |
| **6** | **p1** | **10** | **2** | **9** |
| **7** | **s0** | **15** | **1** | **11** |
| **8** | **s1** | **7** | **0** | **16** |

**Kết quả:**





**Gantt chart diagram:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **S4** | | | **S3** | **S2** |
|  |  | **S3** | **M1** | | |

**1 2 3 4 5 6 7 8 9 10**

|  |  |  |  |
| --- | --- | --- | --- |
| **P1** | **S0** | | |
| **M1** | | **P1** | **S1** |

**10 11 12 13 14 15 16 17 18 19**

|  |  |  |
| --- | --- | --- |
| **S0** | | **M0** |
| **S1** | **P1** | |

**19 20 21 22 23 24 25 26 27 28**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **M0** | **S2** | | **S3** | | **M0** | | **S2** | |
| **S3** | | **M0** | | **S2** | | **S3** | | **M0** |

**28 29 30 31 32 33 34 35 36 37**

|  |  |
| --- | --- |
| **S3** | **S4** |
| **S2** | |

**37 38 39 40 41**

**CPU1**

**CPU2**

1. **Trả lời câu hỏi:**

**Câu hỏi: Ưu điểm của việc sử dụng hàng đợi ưu tiên so với các thuật toán lập lịch khác mà bạn đã học là gì ?**

Hàng đợi ưu tiên có một số ưu điểm chính sau đây:

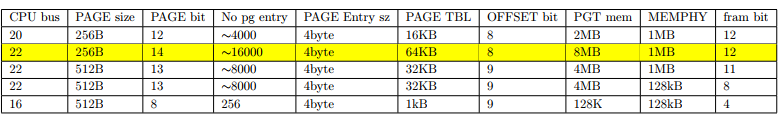
* Ưu tiên các công việc quan trọng: Hàng đợi ưu tiên cho phép xác định mức độ ưu tiên của các công việc. Các công việc quan trọng hơn có thể được ưu tiên xử lý trước, giúp đảm bảo tính cấp thiết của các tác vụ quan trọng.
* Đáp ứng nhanh các yêu cầu ưu tiên: Ưu tiên các công việc quan trọng: Hàng đợi ưu tiên cho phép xác định mức độ ưu tiên của các công việc. Các công việc quan trọng hơn có thể được ưu tiên xử lý trước, giúp đảm bảo tính cấp thiết của các tác vụ quan trọng.
* Ưu tiên các công việc quan trọng: Hàng đợi ưu tiên cho phép xác định mức độ ưu tiên của các công việc. Các công việc quan trọng hơn có thể được ưu tiên xử lý trước, giúp đảm bảo tính cấp thiết của các tác vụ quan trọng.

Tuy nhiên, giải thuật này cũng tồn tại vấn đề là ưu tiên quá mức đối với process ưu tiên cao dẫn đến một hoặc một vài process ưu tiên thấp hơn có thể không được thực thi trong thời gian dài. Đây là hiện tượng Starvation, để giảiquyết vấn đề này ta có thể dùng đến kỹ thuật Aging đó là tăng dần độ ưu tiên của các process chưa được thực thi trong ready queue.

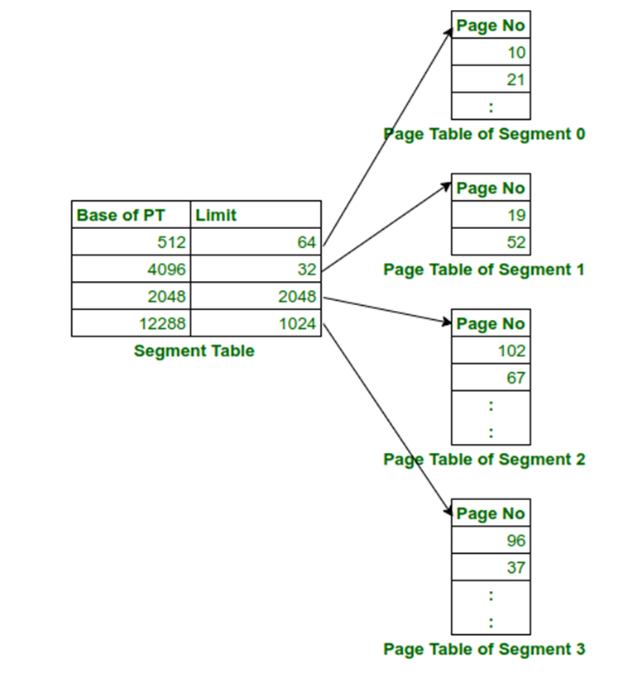
1. **Quản lý bộ nhớ**
2. **Cơ chế quản lý bộ nhớ của hệ điều hành đơn giản:**

Hệ điều hành mà nhóm mô phỏng sẽ quản lý bộ nhớ với kỹ thuật phân đoạn kết hợp phân trang (Segmentation with paging hoặc Segmented paging) mục đích là tận dụng được lợi thế của cả 2 phương pháp này.

Đây là cơ chế mà ở đó không gian bộ nhớ của mỗi process sẽ được chia thành một sốMemory segment (hay Memory area) có thể khác nhau về kích thước để phân loại các dữ liệu (code segment, data segment, heap segment, stack segment ,...) và được mỗi process quản lý thông qua bảng phân đoạn (Segment table). Các phân đoạn (Memory segment) này lại được hệ thống chia thành các trang được quản lý qua bảng phân trang (Page table). Mỗi phân đoạn sẽ sở hữu một bảng phân trang riêng. Sau đây là các thông số cấu hình cho hệ điều hành môphỏng (dòng được tô đậm):



Ảnh 1: Các thông số cho 1 hệ điều hành.



Ảnh 2: Segment kết hợp với paging.

1. **Hiện thực**
   1. **Cấp phát bộ nhớ:**

int \_\_alloc(struct pcb\_t \*caller, int vmaid, int rgid, int size, int \*alloc\_addr)

{

  /\*Allocate at the toproof \*/

  struct vm\_rg\_struct rgnode;

  if (get\_free\_vmrg\_area(caller, vmaid, size, &rgnode) == 0)

  {

    caller->mm->symrgtbl[rgid].rg\_start = rgnode.rg\_start;

    caller->mm->symrgtbl[rgid].rg\_end = rgnode.rg\_end;

    \*alloc\_addr = rgnode.rg\_start;

    return 0;

  }

  /\* TODO get\_free\_vmrg\_area FAILED handle the region management (Fig.6)\*/

  /\*Attempt to increate limit to get space \*/

  struct vm\_area\_struct \*cur\_vma = get\_vma\_by\_num(caller->mm, vmaid);

  int inc\_sz = PAGING\_PAGE\_ALIGNSZ(size);

  //int inc\_limit\_ret

  int old\_sbrk ;

  old\_sbrk = cur\_vma->sbrk;

  /\* TODO INCREASE THE LIMIT

   \* inc\_vma\_limit(caller, vmaid, inc\_sz)

   \*/

  if (inc\_vma\_limit(caller, vmaid, inc\_sz) != 0) {

    return -1;

  }

  /\*Successful increase limit \*/

  caller->mm->symrgtbl[rgid].rg\_start = old\_sbrk;

  caller->mm->symrgtbl[rgid].rg\_end = old\_sbrk + size;

  \*alloc\_addr = old\_sbrk;

  return 0;

}

Thông qua \_\_alloc() hệ thống sẽ cấp phát một vùng nhớ theo yêu cầu của process caller. Ban đầu hệ thống sẽ tìm kiếm vùng nhớ (Memory region) phù hợp trong số các vùng trống mà phân đoạn hiện tại (xác định qua biến vmaid) đang có nếu tìm thấy hệ thống sẽ cấp phát vùng nhớ này cho caller. Ngược lại, tức là không có vùng nào đủ kích thước để đáp ứng yêu cầu hiện tại hệ thống sẽ phải tăng giới hạn của phân đoạn qua hàm inc\_vma\_limit().

Các hành vi của hàm inc\_vma\_limit() như sau:

* Tạo vùng nhớ mới (bắt đầu từ địa chỉ của con trỏ sbrk) có kích thước bằng với kích thước mà caller yêu cầu.
* Tạo vùng nhớ mới (bắt đầu từ địa chỉ của con trỏ sbrk) có kích thước bằng với kích thước mà caller yêu cầu.
* Tạo vùng nhớ mới (bắt đầu từ địa chỉ của con trỏ sbrk) có kích thước bằng với kích thước mà caller yêu cầu.
  1. **Thu hồi bộ nhớ:**

int \_\_free(struct pcb\_t \*caller, int vmaid, int rgid)

{

  struct vm\_rg\_struct rgnode;

  if(rgid < 0 || rgid > PAGING\_MAX\_SYMTBL\_SZ)

    return -1;

  /\* TODO: Manage the collect freed region to freerg\_list \*/

  rgnode.rg\_start = caller->mm->symrgtbl[rgid].rg\_start;

  rgnode.rg\_end = caller->mm->symrgtbl[rgid].rg\_end;

  /\*enlist the obsoleted memory region \*/

  enlist\_vm\_freerg\_list(caller->mm, rgnode);

  return 0;

}

Hàm \_\_free() thực hiện công việc thu hồi vùng nhớ rgid, với hàm này hệ thống chỉ đưa vùng nhớ này vào danh sách các vùng nhớ trống để chờ đợi yêu cầu cấp phát tiếp theo.

* 1. **Đọc/ghi bộ nhớ:**

Đầu tiên, để thực hiện đọc hay ghi vào một vùng nhớ hệ thống phải xác định được trang đích để thực hiện thao tác. Để thu được trang này, hệ điều hành mô phỏng của nhóm sử dụng hàm sau đây:

int pg\_getpage(struct mm\_struct \*mm, int pgn, int \*fpn, struct pcb\_t \*caller)

{

  uint32\_t pte = mm->pgd[pgn];

  if (!PAGING\_PAGE\_PRESENT(pte))

  { /\* Page is not online, make it actively living \*/

    int vicpgn, swpfpn;

    //int vicfpn;

    //uint32\_t vicpte;

    int tgtfpn = PAGING\_SWP(pte);//the target frame storing our variable

    /\* TODO: Play with your paging theory here \*/

    /\* Find victim page \*/

    find\_victim\_page(caller->mm, &vicpgn);

    /\* Get free frame in MEMSWP \*/

    MEMPHY\_get\_freefp(caller->active\_mswp, &swpfpn);

    /\* Do swap frame from MEMRAM to MEMSWP and vice versa\*/

    /\* Copy victim frame to swap \*/

    \_\_swap\_cp\_page(caller->mram, vicpgn, caller->active\_mswp, swpfpn);

    /\* Copy target frame from swap to mem \*/

     \_\_swap\_cp\_page(\*caller->mswp, tgtfpn, caller->mram, vicpgn);

    /\* Update page table \*/

    //pte\_set\_swap() &mm->pgd;

    pte\_set\_swap(&mm->pgd[vicpgn], PAGING\_MAX\_MMSWP, swpfpn);

    /\* Update its online status of the target page \*/

    //pte\_set\_fpn() & mm->pgd[pgn];

    pte\_set\_fpn(&mm->pgd[pgn], vicpgn);

    // pte\_set\_fpn(&pte, tgtfpn);

    enlist\_pgn\_node(&caller->mm->fifo\_pgn,pgn);

  }

  \*fpn = PAGING\_FPN(pte);

  return 0;

}

Nếu trang hiện tại đã hiện diện trong RAM (thể hiện qua bit present) ta có thể trực tiếp thu được trang này nhưng trong trường hợp ngược lại (page fault), ta cần phải thực hiện hoán đổi (swap) để đưa trang vào RAM sau đó mới truy xuất vào trang. Các bước cụ thể như sau:

* Tìm trang thay thế qua hàm find\_victim\_page() nếu không tìm được trang khả thi hệ thống sẽ dừng lập tức
* Tìm khung (frame) trống trong bộ nhớ SWAP qua hàm MEMPHY\_get\_freefp().
* Sau khi đã tìm được cả trang thay thế và khung trống, hệ thống sẽ thực hiện swap out và swap in. Đầu tiên hệ thống sẽ sao chép nội dung của trang thay thế từ RAM sang SWAP. Sau đó sẽ sao chép nội dung khung chứa dữ liệu cần truy xuất từ SWAP sang lại RAM.
* Cập nhật lại nội dung của Page table entry (PTE) tương ứng với hai đối tượng đã được swap.

int find\_victim\_page(struct mm\_struct \*mm, int \*retpgn)

{

  struct pgn\_t \*pg = mm->fifo\_pgn;

  /\* TODO: Implement the theorical mechanism to find the victim page \*/

  if (pg == NULL) {

    return -1;  // No page found

  }

  \*retpgn = pg->pgn;

  mm->fifo\_pgn = pg->pg\_next;

  free(pg);

  return 0;

}

Trên đây là hàm được sử dụng để tìm trang thay thế theo nguyên tắc First in, first out (FIFO).

Sau khi thu được trang đích chung ta đã có thể đọc (hoặc ghi) dữ liệu vào trang mong muốn. Bên dưới là pg\_getval() và pg\_setval() lần lượt hỗ trợ công việc đọc và ghi dữ liệu vào bộ nhớ:

int pg\_getval(struct mm\_struct \*mm, int addr, BYTE \*data, struct pcb\_t \*caller)

{

  int pgn = PAGING\_PGN(addr);

  int off = PAGING\_OFFST(addr);

  int fpn;

  /\* Get the page to MEMRAM, swap from MEMSWAP if needed \*/

  if(pg\_getpage(mm, pgn, &fpn, caller) != 0)

    return -1; /\* invalid page access \*/

  int phyaddr = (fpn << PAGING\_ADDR\_FPN\_LOBIT) + off;

  MEMPHY\_read(caller->mram,phyaddr, data);

  return 0;

}

int pg\_setval(struct mm\_struct \*mm, int addr, BYTE value, struct pcb\_t \*caller)

{

  int pgn = PAGING\_PGN(addr);

  int off = PAGING\_OFFST(addr);

  int fpn;

  /\* Get the page to MEMRAM, swap from MEMSWAP if needed \*/

  if(pg\_getpage(mm, pgn, &fpn, caller) != 0)

    return -1; /\* invalid page access \*/

  int phyaddr = (fpn << PAGING\_ADDR\_FPN\_LOBIT) + off;

  MEMPHY\_write(caller->mram,phyaddr, value);

   return 0;

}

1. **Phân tích kết quả kiểm thử:**

Test case được thực hiện trên file “os\_1\_singleCPU\_mlq\_paging” có nội dung như sau:

**2 1 8**

**1048576 16777216 0 0 0**

**1 s4 4**

**2 s3 3**

**4 m1s 2**

**6 s2 3**

**7 m0s 3**

**9 p1s 2**

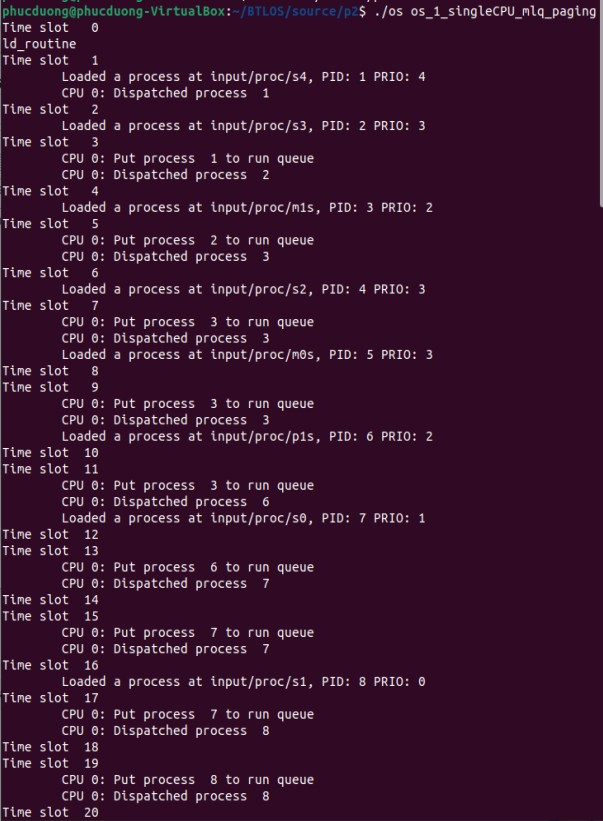
**11 s0 1**

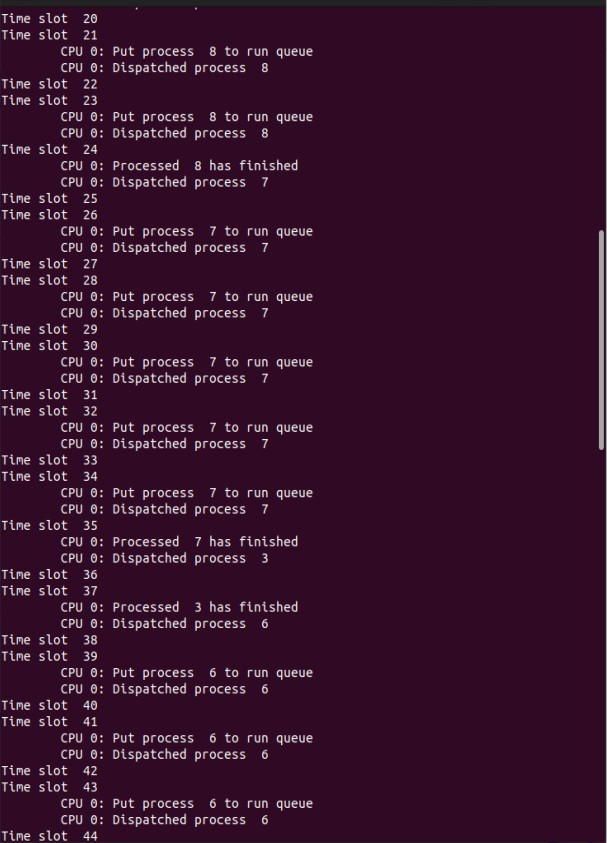
**16 s1 0**

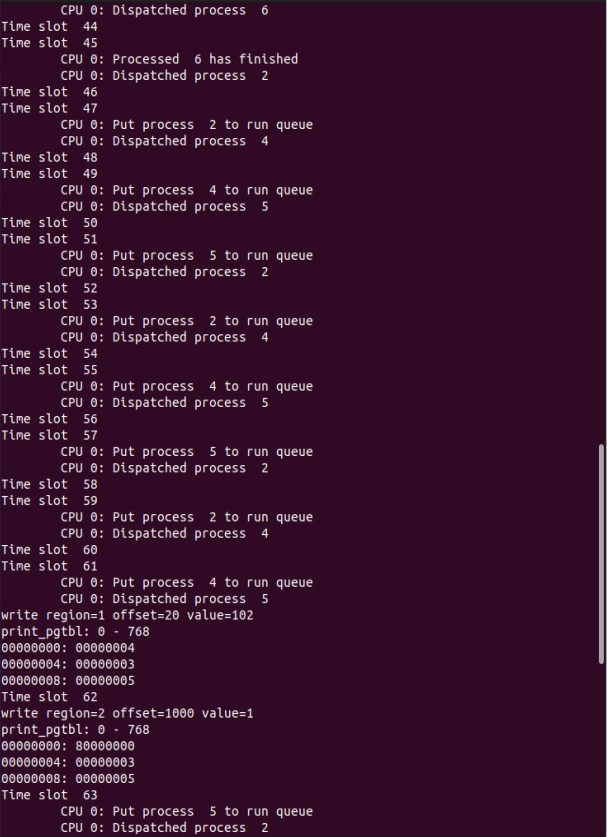
Trong test case này, hệ điều hành sử dụng 1 CPU áp dụng Round Robin với time slice = 2 để định thời 8 tiến trình. Thông tin tổng quan của 6 tiến trình này như sau:

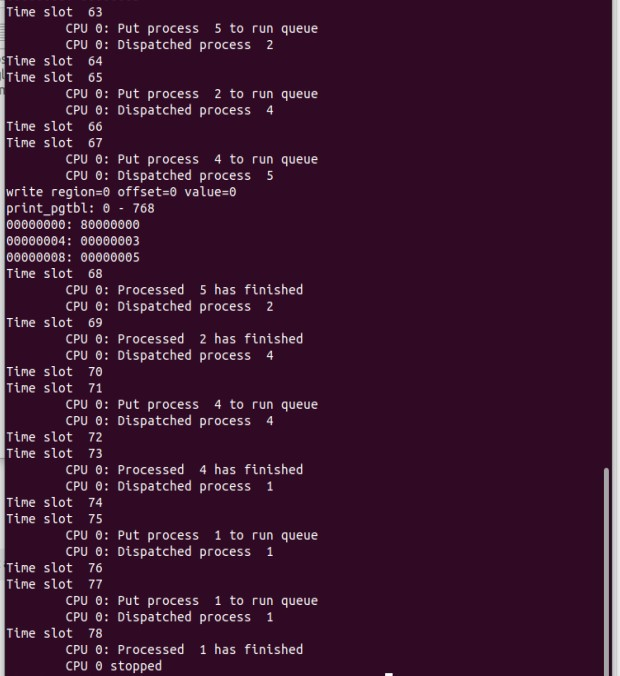
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ID** | **Tiến trình** | **Số chu kỳ thực hiện** | **Độ ưu tiên** | **Thời gian đến** |
| **1** | **s4** | **7** | **4** | **1** |
| **2** | **s3** | **11** | **3** | **2** |
| **3** | **m1s** | **8** | **2** | **4** |
| **4** | **s2** | **12** | **3** | **6** |
| **5** | **m0s** | **7** | **3** | **7** |
| **6** | **p1s** | **10** | **2** | **9** |
| **7** | **s0** | **15** | **1** | **11** |
| **8** | **s1** | **7** | **0** | **16** |

Kết quả:









\***Gant chart diagram:**

|  |  |  |
| --- | --- | --- |
| **s4** | **s3** | **m1s** |

**1 2 3 4 5 6 7 8 9 10**

|  |  |  |  |
| --- | --- | --- | --- |
| **m1s** | **p1s** | **s0** | **s1** |

**10 11 12 13 14 15 16 17 18 19**

|  |  |
| --- | --- |
| **s1** | **s0** |

**19 20 21 22 23 24 25 26 27 28**

|  |  |
| --- | --- |
| **s0** | **m1s** |

**28 29 30 31 32 33 34 35 36 37**

|  |  |
| --- | --- |
| **p1s** | **s3** |

**37 38 39 40 41 42 43 44 45 46**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **s3** | **s2** | **m0s** | **s3** | **s2** |

**46 47 48 49 50 51 52 53 54 55**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **m0s** | **s3** | **s2** | **m0s** | **s3** |

**55 56 57 58 59 60 61 62 63 64**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **s3** | **s2** | **m0s** | **s3** | **s2** |

**64 65 66 67 68 69 70 71 72 73**

|  |
| --- |
| **s4** |

**73 74 75 76 77 78**

1. **Trả lời câu hỏi:**

**Câu 1: Hệ điều hành được mô phỏng có thiết kế gồm nhiều Memory Segment hoặc Memory Area (MA). Vậy tác dụng của kiểu thiết kế này là gì ?**

* Tối ưu hóa việc sử dụng bộ nhớ: Bằng cách chia bộ nhớ thành các phân đoạn hoặc khu vực có kích thước khác nhau, mỗi phân đoạn có thể được cấp phát cho một tiến trình, giúp tối ưu hóa việc sử dụng bộ nhớ.
* Hỗ trợ đa chương trình (Multiprogramming): Bằng cách đẩy memory vào các khối cố định, hệ thống có thể tải cùng lúc nhiều tiến trình hơn vào bộ nhớ chính, làm tăng mức độ Đa lập trình cũng như sử dụng CPU.
* Loại bỏ vấn đề phân mảnh bên ngoài (External Fragmentation): Kỹ thuật paging giúp loại bỏ vấn đề phân mảnh bên ngoài, khi memory được đẩy vào các khối cố định3.
* Giảm chi phí đi kèm với việc nén dữ liệu trong quá trình di dời: Các chi phí đi kèm với việc nén dữ liệu trong quá trình di dời đều được loại bỏ.
* Bảo vệ các tiến trình người dùng với nhau và chống việc thay đổi mã lệnh và dữ liệu của hệ điều hành: Sử dụng thanh ghi tái định vị (relocation register) và thanh ghi giới hạn (limit register) để bảo vệ các tiến trình người dùng với nhau, và để chống việc thay đổi mã lệnh và dữ liệu của hệ điều hành.

Vì vậy, việc sử dụng Memory Segment hoặc Memory Area trong thiết kế hệ điều hành giúp tối ưu hóa việc quản lý và sử dụng bộ nhớ.

**Câu 2: Sẽ ra sao nếu ta chia địa chỉ nhiều hơn 2 bậc trong hệ thống quản lý phân trang ?**

* Nếu chia địa chỉ nhiều hơn 2 bậc trong hệ thống quản lý phân trang, chúng ta sẽ tạo ra một hệ thống phân trang đa cấp (multilevel paging). Điều này có thể giúp quản lý hiệu quả hơn đối với không gian địa chỉ lớn.
* Trong hệ thống phân trang đa cấp, địa chỉ ảo được chia thành nhiều phần, mỗi phần tương ứng với một cấp độ trong bảng trang. Ví dụ, trong một máy tính 32-bit địa chỉ, sử dụng một bảng trang nhị cấp, địa chỉ ảo có thể được phân bổ như sau: 9 bit dành cho bảng trang cấp 1, 11 bit cho bảng trang cấp 2, còn lại dành cho offset.

Tuy nhiên, việc sử dụng phân trang đa cấp cũng có nhược điểm là tốn kém về mặt thời gian do phải truy cập bộ nhớ nhiều lần để lấy địa chỉ thựctừ địa chỉ ảo. Ngoài ra, nó cũng tốn kém về không gian bộ nhớ do cần lưu trữ nhiều bảng trang.

**Câu 3: Đâu là ưu thế và khó khăn của kỹ thuật Segmentation kết hợp Paging ?**

* **Ưu thế:**
* Tối ưu hóa việc sử dụng bộ nhớ: Segmentation cho phép lưu trữ các đoạn có kích thước khác nhau, giúp tận dụng không gian bộ nhớ hiệu quả. Paging giúp quản lý không gian bộ nhớ bằng cách chia nó thành các khối cố định.
* Bảo vệ và cô lập: Mỗi đoạn trong Segmentation được bảo vệ và cô lập từ các đoạn khác, giúp ngăn chặn các tiến trình truy cập trái phép vào nhau.
* Hỗ trợ chia sẻ và tái sử dụng: Segmentation cho phép chia sẻ và tái sử dụng các đoạn mã và dữ liệu giữa các tiến trình.
* **Khó khăn:**
* Phân mảnh bộ nhớ: Segmentation có thể dẫn đến phân mảnh bộ nhớ, khi các đoạn có kích thước khác nhau được cấp phát và giải phóng.
* Chi phí phần cứng: Paging có thể làm tăng chi phí phần cứng, vì các địa chỉ của trang bộ nhớ được ánh xạ tới phần cứng.
* Chi phí bảng chỉ mục: Một hạn chế lớn của Segmentation kết hợp Paging là toàn bộ bảng chỉ mục phải nạp vào bộ nhớ trong suốt thời gian làm việc của hệ thống, điều này sẽ làm tốn thời gian nạp bảng chỉ mục của hệ điều hành và làm lãng phí không gian bộ nhớ của hệ thống.

**Câu 4: Điều gì sẽ xảy ra nếu hệ điều hành không thực hiện đồng bộ ?**

* Race Condition: Điều này xảy ra khi nhiều tiến trình cố gắng truy cập hoặc thay đổi dữ liệu chung cùng một lúc1. Kết quả của việc thực thi có thể phụ thuộc vào thứ tự mà các tiến trình được lên lịch1.
* Không đảm bảo tính nguyên tử (Atomicity): Trong một số trường hợp, một tiến trình cần thực hiện một chuỗi các hoạt động mà không bị gián đoạn. Nếu hệ điều hành không thực hiện đồng bộ, thì các hoạt động này có thể bị gián đoạn, dẫn đến kết quả không như mong đợi1.
* Không đảm bảo tính nhất quán (Consistency): Khi nhiều tiến trình cùng lúc truy cập và thay đổi dữ liệu chung, dữ liệu có thể trở nên không nhất quán nếu không có sự đồng bộ.
* Deadlock: Đây là tình huống mà mỗi tiến trình trong một tập hợp các tiến trình đang chờ một tài nguyên mà một tiến trình khác trong tập hợp đang giữ. Nếu không có sự đồng bộ, các tiến trình có thể rơi vào tình trạng bế tắc, khiến cho hệ thống không thể tiếp tục hoạt động.

Ví dụ trong thực thi định thời của hệ điều hành trong bài tập lớn này, vùng tranh chấp (critical section) giữa các process là mlq\_ready\_queue việc truy cập vào vùng này cần phảiđược đồng bộ (cụ thể là nhóm sử dụng biến queue\_lock) đảm bảo rằng trong một thời điểm chỉ một process được truy cập hay chỉnh sửa mlq\_ready\_queue. Xét trường hợp tại cùng thời điểm ban đầu cả hai process (cùng độ ưu tiên) được đưa vào hệ thống mà không có sự đồng bộ. Lúc này hệ thống sẽ đồng thời đưa hai process này vào cùng một hàng đợi và điều này tồn tại khả năng hàng đợi đó sẽ chỉ chứa một trong hai process mà không phải là cả hai như vậy sẽ làm mất mát dữ liệu và ảnh hưởng đến sự đúng đắn của hệ thống