ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC BÁCH KHOA

KHOA KHOA HỌC VÀ KĨ THUẬT MÁY TÍNH

MÔN : HỆ ĐIỀU HÀNH (CO2017)

BÁO CÁO BÀI TẬP LỚN

SIMPLE OPERATING SYSTEM

Giảng viên hướng dẫn: **Nguyễn Quang Hùng**

Nhóm SV thực hiện: **Nhóm D**

Nguyễn Thiên Hải – 2252189

Nguyễn Lê Minh Huân – 2252242

Võ Nguyễn Gia Huy – 2252270

Lê Hải Long – 2252441

Vương Thanh Phương - 2252658

Mục lục

Giới thiệu

Mục tiêu của bài tập lớn là mô phỏng một hệ điều hành đơn giản để giúp sinh viên hiểu các khái niệm cơ bản về định thời, đồng bộ hóa và quản lý bộ nhớ. Qua đó hiểu được nguyên lí cơ bản của hệ điều hành.

1. Định thời CPU

1.1 Cơ sở lý thuyết

Định thời CPU là quá trình quyết định việc phân phối thời gian xử lý của CPU cho các quá trình đang chạy trong hệ thống máy tính. Nhiệm vụ chính của định thời CPU là quản lý và điều chỉnh việc chuyển đổi giữa các quá trình để đảm bảo tối ưu hoá sử dụng tài nguyên và cung cấp hiệu suất cao nhất cho hệ thống.

Multilevel Queue Scheduling là một phương pháp kết hợp giữa priority scheduling, round-robin scheduling và phân chia single queue thành nhiều queue với độ ưu tiên riêng biệt.

Priority scheduling là một cấu trúc dữ liệu dùng để sắp xếp độ ưu tiên. Trong đó, mỗi process gắn với một độ ưu tiên nhất định và CPU sẽ phân bổ process có độ ưu tiên cao nhất. Priority scheduling có hai loại: pre-emptive và nonpre-emptive. Pre-emptive nghĩa là khi một process mới đến, CPU sẽ so với các process khác, kể cả process đang chạy và sẽ lấy process có độ ưu tiên cao nhất để tiến hành. Còn nonpre-emptive thì process chỉ đơn giản là đưa thẳng vào hàng đợi sẵn sàng.

Round-robin scheduling là một phương pháp định thời theo thời gian thực. Trong một chu kỳ, mỗi process được gán một thời gian giữ CPU nhất định. Sau đó, nó sẽ chuyển qua một process các vào chu kỳ mới theo quy tắc vòng tròn.

+ **Question:** What is the advantage of the scheduling strategy used in this assignment in comparison with other scheduling algorithms you have learned?

- Lợi thế của việc sử dụng Multilevel Queue Scheduling so với các phương pháp khác là:

→ Phân loại tác vụ: Chiến lược này cho phép phân loại các tác vụ vào các hàng đợi riêng biệt dựa trên thuộc tính của chúng, ví dụ như tương tác người dùng, xử lý thời gian thực, hoặc công việc nền. Điều này giúp đảm bảo rằng mỗi loại tác vụ nhận được mức độ ưu tiên và tài nguyên phù hợp.

→ Tiết kiệm thời gian : do các quy trình được phân bổ tương ứng với phương pháp thích hợp giúp tối ưu thời gian một cách linh hoạt.

→ Công bằng : Phân bổ công bằng cho nhiều loại process khác nhau, tránh được tình trạng starvation.

→ Tính linh hoạt : Khả năng tùy biến tốt vì kết hợp nhiều loại phương pháp nên dễ tránh, đề phòng được các vấn đề xảy ra. Phân loại tác vụ: Chiến lược này cho phép phân loại các tác vụ vào các hàng đợi riêng biệt dựa trên thuộc tính của chúng, ví dụ như tương tác người dùng, xử lý thời gian thực, hoặc công việc nền. Điều này giúp đảm bảo rằng mỗi loại tác vụ nhận được mức độ ưu tiên và tài nguyên phù hợp.

→ Tối ưu hóa hiệu suất : Bằng cách nhóm các tác vụ tương tự lại với nhau, các hàng đợi có thể được tối ưu hóa cho các loại tác vụ cụ thể. Ví dụ, các tác vụ thời gian thực có thể được ưu tiên cao hơn so với các tác vụ nền không quan trọng**.**

1.2 Hiện thực định thời áp dụng chính sách MLQ

Thực hiện 3 hàm sau : enqueue() ; dequeue() ; get_mlq_proc().

Hàm enqueue() đưa 1 process vào cuối hàng chờ có độ ưu tiên tương ứng :

```
void enqueue(struct queue_t *q, struct pcb_t *proc)
{
        /* TODO: put a new process to queue [q] */
        if (q && proc)
        {
                if(q->size < MAX_QUEUE_SIZE){
                        q->proc[q->size++]=proc;
                }
        }
}
```

Code hàm enqueue()

Hàm dequeue() lấy 1 process từ đầu hàng chờ có độ ưu tiên chỉ định :

```c
struct pcb_t *dequeue(struct queue_t *q)
{
        /* TODO: return a pcb whose prioprity is the highest
         * in the queue [q] and remember to remove it from q
         * */
        if(!q || empty(q))
                return NULL;
        struct pcb_t *tmp=q->proc[0];
        int n=q->size;
        for(int i=0;i<n-1;i++){
                q->proc[i]=q->proc[i+1];
        }
        q->size--;
        return tmp;
}
```

Code hàm dequeue()

Hàm get_mlq_proc lấy 1 process từ mlq_ready_queue.

Mỗi queue đều có slot tương ứng (slot = MAX_PRIOR – prior).Khi 1 queue đã sử dụng hết số slot thì CPU sẽ chuyển sang những hàng đợi tiếp theo để tìm process.

Đầu tiên,ta chạy vòng lặp while để kiểm tra xem các hàng đợi không rỗng của mlq_ready_queue có full slot hay không,nếu có ta reset slot_count lại bằng 0.

Tiếp theo,ta chạy vòng lặp từ prior 0 đến prior 140,nếu queue nào không rỗng và còn slot ta lẽ lấy 1 process bằng dequeue và break vòng lặp . Mỗi lần lấy được process ta lẽ tăng số slot_count của queue tương ứng.

```c
struct pcb_t *get_mlq_proc(void) {
    /*TODO: get a process from PRIORITY [ready_queue].
     * Remember to use lock to protect the queue.
     */

    pthread_mutex_lock(&queue_lock);
    struct pcb_t *proc = NULL;
    int all_full=1;
    for(int i=0;i<MAX_PRIO;i++){
        if(!empty(&mlq_ready_queue[i]) && mlq_ready_queue[i].slot_count<MAX_PRIO-i){ // check xem cac queue co full het slot ko
            all_full=0;
            break;
        }
    }
    if(all_full){
        printf("MLQ da full slot!\n");
        for(int i=0;i<MAX_PRIO;i++)
            mlq_ready_queue[i].slot_count = 0;
    }
    for(int t=0;t<MAX_PRIO;t++){
        if(!empty(&mlq_ready_queue[t]) && mlq_ready_queue[t].slot_count<MAX_PRIO-t){
            mlq_ready_queue[t].slot_count++;
            proc = dequeue(&mlq_ready_queue[t]);
            //printf("Process ID la %2d va Prior la %u  va dang o queue  %d  va slot count hien tai la %d gau gau gau\n",proc->pid,
            break;
        }
    }
    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

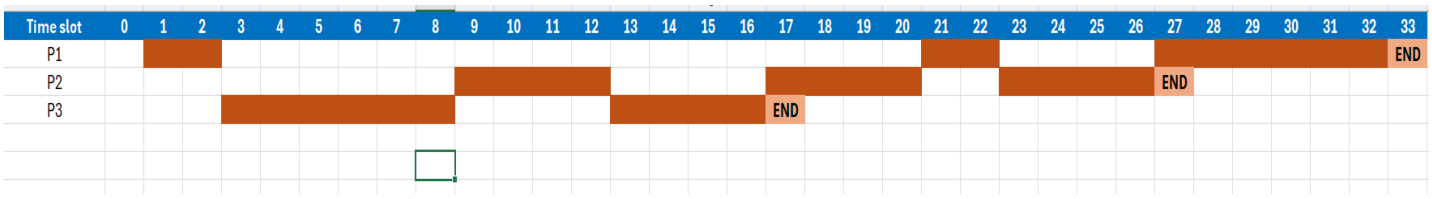
Code hàm get_mlq_proc()

1.3 Biểu đồ Gantt

Case 1: 1 CPU , Time slot = 2 :

```
2 1 3
0 p1s 139
1 p2s 138
2 p1s 137
```

Ta có được biểu đồ Gantt như sau:

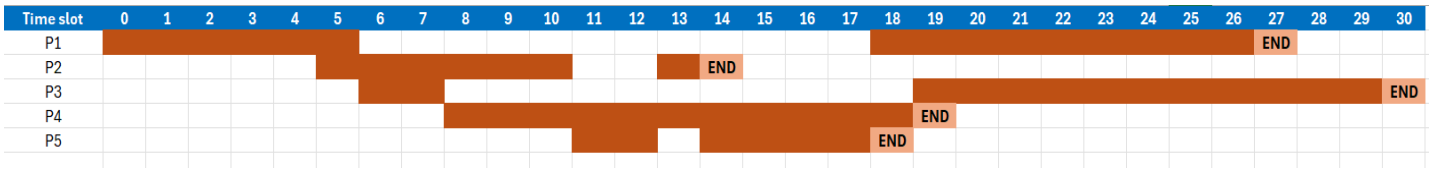


Biểu đồ Gantt case 1

Case 2: 2 CPU , Time slot = 2 :

```
2 2 5
0 s0 25
4 s1 3
6 s2 25
7 s3 0
10 s4 3
```

Ta có được biểu đồ Gantt như sau:



Biểu đồ Gantt case 2

2. Quản lí bộ nhớ

2.1 Cơ sở lý thuyết

Segmentation là một kỹ thuật quản lý bộ nhớ, trong đó bộ nhớ được chia thành các phần có kích thước thay đổi. Mỗi phần được gọi là một Segmentation có thể được phân bổ cho một quy trình.

Paging là một phương thức quản lý bộ nhớ giúp loại bỏ nhu cầu phải cấp phát liên tục cho bộ nhớ vật lý. Là quá trình truy xuất các tiến trình dưới dạng các trang từ bộ nhớ phụ trợ vào bộ nhớ chính.

Mapping là quá trình ánh xạ hoặc liên kết giữa các thành phần khác nhau để định vị và truy cập vào các tài nguyên trong hệ thống.

Translation Lookaside Buffer (TLB) là một cấu trúc bộ nhớ đệm quan trọng trong việc quản lý bộ nhớ của máy tính. Nó hoạt động như một bảng nhanh lưu trữ thông tin dịch địa chỉ từ địa chỉ ảo sang địa chỉ vật lý, giúp giảm thiểu thời gian truy cập bộ nhớ. TLB giữ một số lượng hạn chế của các mục từ bảng trang, cho phép truy cập nhanh chóng đến các địa chỉ thường xuyên được sử dụng.

+ **Question 1:** 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?

-Thiết kế hệ điều hành với nhiều phân đoạn bộ nhớ mang lại một số lợi ích sau:

→ Quản lý bộ nhớ hiệu quả hơn: Bằng cách chia nhỏ bộ nhớ thành các phân đoạn, hệ điều hành có thể quản lý bộ nhớ một cách linh hoạt và hiệu quả hơn. Điều này giúp giảm thiểu lãng phí bộ nhớ do việc cấp phát và giải phóng không hiệu quả.

→ Dễ dàng mở rộng: Khi cần thêm bộ nhớ, bạn chỉ cần thêm phân đoạn mới mà không cần phải thay đổi cấu trúc bộ nhớ hiện tại.

→ Tăng cường bảo mật: Với việc chia bộ nhớ thành nhiều phân đoạn, mỗi phân đoạn có thể được kiểm soát và bảo vệ riêng. Điều này giúp ngăn chặn các lỗi hoặc tấn công từ một phân đoạn ảnh hưởng đến toàn bộ hệ thống.

→ Cải thiện hiệu suất: Việc sử dụng các phân đoạn có thể giúp hệ điều hành tối ưu hóa việc truy cập và sử dụng bộ nhớ. Ví dụ, hệ điều hành có thể dễ dàng theo dõi và ưu tiên các phân đoạn quan trọng hơn, giúp cải thiện hiệu suất tổng thể.

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

- Việc phân chia như thế tạo ra một cấu trúc gọi là **Multi-level Paging.**

Multi-level paging là một kỹ thuật trong quản lý bộ nhớ ảo, trong đó địa chỉ ảo được chia thành nhiều cấp trang (page levels) để tìm đến địa chỉ vật lý tương ứng. Thay vì sử dụng một bảng trang phẳng, multi-level paging sử dụng một hệ thống phân cấp các bảng trang.

Trong multi-level paging, địa chỉ ảo được chia thành nhiều phần. Mỗi phần đại diện cho một cấp trong hệ thống phân cấp. Ví dụ, trong một hệ thống hai cấp, địa chỉ ảo có thể được chia thành ba phần: chỉ mục của bảng trang thứ nhất, chỉ mục của bảng trang thứ hai và một phần bù (offset). Hệ thống sẽ sử dụng chỉ mục đầu tiên để tìm bảng trang thứ hai, sau đó dùng chỉ mục thứ hai để tìm trang vật lý cụ thể.

Một số hệ quả của multi-level paging :

→ Tăng độ phức tạp: Việc chia địa chỉ thành nhiều cấp hơn làm tăng độ phức tạp của hệ thống. Điều này có thể dẫn đến việc thiết kế bảng trang trở nên khó hiểu hơn và khó bảo trì hơn.

→ Hiệu suất chậm hơn: Với nhiều cấp độ hơn, hệ thống sẽ phải thực hiện nhiều bước hơn để dịch địa chỉ ảo sang địa chỉ vật lý. Điều này có thể làm chậm hiệu suất truy cập bộ nhớ, đặc biệt nếu không sử dụng bộ nhớ đệm (TLB) một cách hiệu quả.

→ Tiết kiệm không gian: Mặc dù phức tạp hơn, nhưng việc chia địa chỉ thành nhiều cấp có thể giúp tiết kiệm không gian. Điều này có thể đặc biệt hữu ích trong các hệ thống có địa chỉ lớn, vì nó giúp giảm bớt kích thước của bảng trang.

→ Tiết kiệm không gian: Mặc dù phức tạp hơn, nhưng việc chia địa chỉ thành nhiều cấp có thể giúp tiết kiệm không gian. Điều này có thể đặc biệt hữu ích trong các hệ thống có địa chỉ lớn, vì nó giúp giảm bớt kích thước của bảng trang.

+ **Question 3:** What is the advantage and disadvantage of segmentation with paging?

- Điểm mạnh của cơ chế segmentation with paging là:

→ Việc sử dụng bộ nhớ được tối ưu hơn và giảm thiểu khả năng bị phân mảnh ở ngoài.

→ Quản lý bộ nhớ linh hoạt: Kết hợp phân đoạn và phân trang giúp hệ thống quản lý bộ nhớ hiệu quả hơn. Phân đoạn cho phép chia bộ nhớ thành các phần logic tương ứng với các thành phần của chương trình, trong khi phân trang giúp quản lý không gian bộ nhớ một cách hiệu quả.

→ Bảo mật tốt hơn: Kết hợp hai kỹ thuật này giúp tăng cường bảo mật, vì các đoạn và trang có thể được kiểm soát độc lập. Điều này giúp ngăn chặn lỗi hoặc tấn công trong một phân đoạn hoặc trang không ảnh hưởng đến các phần khác của hệ thống.

Hạn chế của cơ chế này là:

→ Không hạn chế được nguy cơ phân mảnh nội.

→ Độ lớn của process bị giới hạn hơn.

→ Phức tạp hơn: Thiết kế hệ thống kết hợp hai kỹ thuật này làm tăng độ phức tạp, cả trong việc lập trình và trong quản lý bộ nhớ.

→ Tốn tài nguyên: Việc duy trì cả bảng phân đoạn và bảng trang có thể tốn tài nguyên, đặc biệt là với các hệ thống có không gian địa chỉ lớn hoặc nhiều đoạn.

+ **Question 4:** What will happen if the multi-core system has each CPU core can be run in a different context, and each core has its own MMU and its part of the core (the TLB)? In modern CPU, 2-level TLBs are common now, what is the impact of these new memory hardware configurations to our translation schemes?

- Hệ thống đa lõi hiện đại với các bộ quản lý bộ nhớ (MMU) và bộ nhớ đệm (TLB) riêng cho mỗi lõi mang lại nhiều lợi ích và thách thức đối với các cơ chế dịch địa chỉ:

→ Hiệu suất tốt hơn: Mỗi lõi CPU có MMU và TLB riêng sẽ cải thiện hiệu suất dịch địa chỉ vì nó giảm thiểu sự cạnh tranh tài nguyên giữa các lõi. Điều này cũng cho phép các lõi hoạt động độc lập, không bị ảnh hưởng bởi nhau khi thực hiện dịch địa chỉ.

→ Ngữ cảnh độc lập: Khi mỗi lõi có thể chạy trong một ngữ cảnh khác nhau (không gian địa chỉ khác hoặc luồng khác), điều này cho phép đa nhiệm hiệu quả hơn. Mỗi lõi có thể độc lập dịch địa chỉ ảo sang địa chỉ vật lý, cải thiện khả năng thực thi song song.

→ Tăng độ phức tạp: Việc có nhiều TLB và MMU độc lập sẽ làm tăng độ phức tạp của hệ thống, đặc biệt trong việc đồng bộ hóa thông tin giữa các lõi. Hệ thống cần phải đảm bảo rằng các thông tin về dịch địa chỉ được cập nhật đồng bộ, đặc biệt khi các lõi làm việc trên cùng một không gian địa chỉ.

→ 2-level TLB : Với TLB 2 cấp, quá trình dịch địa chỉ có thể nhanh hơn nhờ có một cấp TLB nhỏ hơn, nhanh hơn và một cấp lớn hơn, chậm hơn. Điều này cho phép hệ thống tối ưu hóa việc dịch địa chỉ, nhưng cũng yêu cầu quản lý hiệu quả giữa hai cấp TLB để tránh bị lỗi TLB quá nhiều.

2.2 Hiện thực quản lý bộ nhớ

mm.c :

Trong file này, ta thực hiện các hàm : vmap_page_range() , alloc_pages_range().

- Hàm vmap_page_range():

Hàm này ánh một danh sách các frame vào không gian địa chỉ ảo của một tiến trình.

Đặt địa chỉ bắt đầu và kết thúc của vùng cần được ánh xạ.

Vòng lặp duyệt qua các frame, ánh xạ từng frame bộ nhớ vào bảng trang của tiến trình,thêm các node vào danh sách pgn_node để theo dõi.

```
int vmap_page_range(struct pcb_t *caller,           // process call
                    int addr,                       // start address which is aligned to pagesz
                    int pgnum,                      // num of mapping page
                    struct framephy_struct *frames, // list of the mapped frames
                    struct vm_rg_struct *ret_rg)    // return mapped region, the real mapped fp
{                                                   // no guarantee all given pages are mapped
  struct framephy_struct *fpit;
  int pgit = 0;
  int pgn = PAGING_PGN(addr);

  ret_rg->rg_start = addr; // at least the very first space is usable
  ret_rg->rg_end = ret_rg->rg_start + pgnum * PAGING_PAGESZ;

  /* TODO map range of frame to address space
   *      [addr to addr + pgnum*PAGING_PAGESZ
   *      in page table caller->mm->pgd[]
   */
  for (; pgit < pgnum; ++pgit)
  {
    fpit = frames;
    pte_set_fpn(&caller->mm->pgd[pgn + pgit], fpit->fpn);
    frames = frames->fp_next;
    free(fpit);

    /* Tracking for later page replacement activities (if needed)
     * Enqueue new usage page */
    enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit);
  }

  return 0;
}
```

Code hàm vmap_page_range()

- Hàm alloc_pages_range() :

Hàm này cấp phát 1 dãy các frame trong bộ nhớ cho 1 tiến trình.

Vòng lặp qua số lượng trang yêu cầu,cấp phát từng frame.

Nếu còn frame trong bộ nhớ, hàm sẽ cấp phát .

Nếu không còn frame trống, hàm sẽ cố gắng tìm 1 trang thay thế (victim page) đẩy ra MEMSWAP. Do có tới 4 MEMSWAP nên ta cần xác định MEMSWAP nào còn không gian trống để chứa victim page.

Sau khi thay thế cập nhật mục trong bảng trang để phản ánh việc swap victim page.

Thêm các frame đã được cấp phát vào danh sách frame và danh sách frame đã sử dụng của bộ nhớ RAM.

```
int alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct framephy_struct **frm_lst)
{
  int pgit, fpn;
  // struct framephy_struct *newfp_str;
  struct framephy_struct *newframe;
  for (pgit = 0; pgit < req_pgnum; pgit++)
  {
    if (MEMPHY_get_freefp(caller->mram, &fpn) == 0)
    {
      newframe = (struct framephy_struct *)malloc(sizeof(struct framephy_struct));
      newframe->fpn = fpn;
      newframe->owner = caller->mm;
    }
    else
    { // ERROR CODE of obtaining somes but not enough frames
      // return -1;
      int vicpgn, vicfpn;
      int vicpte;
      int swpfpn;
      if (find_victim_page(caller->mm, &vicpgn) < 0)
        return -1;
      vicpte = caller->mm->pgd[vicpgn];
      vicfpn = PAGING_FPN(vicpte);
      newframe = (struct framephy_struct *)malloc(sizeof(struct framephy_struct));
      newframe->fpn = vicfpn;
      newframe->owner = caller->mm;
      if (MEMPHY_get_freefp(caller->active_mswp, &swpfpn) == 0)
      {
        __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
      }
      else
      {
        struct memphy_struct *new_mswp = (struct memphy_struct *)caller->mswp;
        for (int i = 0; i < PAGING_MAX_MMSWP; i++)
        {
```

```
    else
    {
      struct memphy_struct *new_mswp = (struct memphy_struct *)caller->mswp;
      for (int i = 0; i < PAGING_MAX_MMSWP; i++)
      {

        if (MEMPHY_get_freefp(new_mswp + i, &swpfpn) == 0)
        {
          __swap_cp_page(caller->mram, vicfpn, new_mswp + i, swpfpn);
          caller->active_mswp = new_mswp + i;
          break;
        }
      }
    }
    struct memphy_struct *mswp = (struct memphy_struct *)caller->mswp;
    int swp_index = 0;
    for (swp_index = 0; swp_index < PAGING_MAX_MMSWP; swp_index++)
    {
      if (mswp + swp_index == caller->active_mswp)
        break;
    }
    pte_set_swap(&caller->mm->pgd[vicpgn], swp_index, swpfpn);
  }
  if (!*frm_lst)
    *frm_lst = newframe;
  else
  {
    newframe->fp_next = *frm_lst;
    *frm_lst = newframe;
  }
  newframe->fp_next = caller->mram->used_fp_list;
  caller->mram->used_fp_list = newframe;
}

  return 0;
}
```

Code hàm alloc_pages_range()

mm-memphy.c :

Hàm MEMPHY_dump :

Hàm này hiển thị nội dung của bộ nhớ vật lý được lưu trữ trong cấu trúc memphy_struct.

Hàm sử dụng vòng lặp for để duyệt qua tất cả các phần tử trong mảng storage và in ra các phần tử khác 0.

```c
int MEMPHY_dump(struct memphy_struct *mp)
{
    /*TODO dump memphy contnt mp->storage
     *     for tracing the memory content
     */


    printf("Tracing memory content - (index,content): ");
    if (mp && mp->storage)
    {
        for (int i = 0; i < mp->maxsz; i++)
        {
            if (mp->storage[i] != (char)0)
                printf("(%d ; %d) ", i, mp->storage[i]);
        }
        printf("\n");
    }
    return 0;
}
```

Code hàm MEMPHY_dump()

mm-vm.c :

- Hàm __alloc() :

Hàm này cấp phát một vùng nhớ cho một tiến trình.

Đầu tiên , hàm này cố gắng lấy một vùng nhớ trống đủ lớn cho kích thước yêu cầu.Nếu thành công , cập nhật địa chỉ bắt đầu và kết thúc vùng nhớ của vùng cần cấp phát.

Nếu thất bại ,hàm sẽ tính toán kích thước cần tăng để phù hợp với kích thước trang. Nếu old_sbrk + size lớn hơn vm_end, nghĩa là vùng nhớ hiện tại không đủ lớn, thì tăng giới hạn vùng nhớ ảo bằng cách gọi inc_vma_limit.Cập nhật bảng ký hiệu với địa chỉ bắt đầu và kết thúc của vùng nhớ mới.

Kiểm tra xem có còn không gian trống trong vùng nhớ ảo sau khi cấp phát không.Nếu có,tạo 1 node free_rg để lưu vùng trống đó và thêm nó vào danh sách vm_freerg.

Trả về 0 để chỉ ra rằng cấp phát thành công.

```c
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *alloc_addr)
{
  /*Allocate at the toproof */
  pthread_mutex_lock(&vm_lock);
  struct vm_rg_struct rgnode;
  while(0);
  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;

    pthread_mutex_unlock(&vm_lock);
    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)
   */
  inc_vma_limit(caller, vmaid, inc_sz);

  /*Successful increase limit */
  caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
  caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
```

```
  /*Successful increase limit */
  caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
  caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;

  *alloc_addr = old_sbrk;

  struct vm_area_struct *remain_rg = get_vma_by_num(caller->mm, vmaid);
  if (old_sbrk + size < remain_rg->sbrk)
  {
    struct vm_rg_struct *free_rg = malloc(sizeof(struct vm_rg_struct));
    free_rg->rg_start = old_sbrk + size;
    free_rg->rg_end = remain_rg->sbrk;
    enlist_vm_freerg_list(caller->mm, free_rg);
  }

  pthread_mutex_unlock(&vm_lock);
  return 0;
}
```

Code hàm __alloc()

- Hàm __free() :

Hàm này giải phóng một vùng nhớ của một tiến trình.

Đầu tiên , kiểm tra nếu rgid nằm ngoài phạm vi hợp lệ, trả về -1.

Lấy vùng nhớ dựa trên ID vùng nhớ từ bảng ký hiệu.Nếu vùng nhớ đã được giải phóng trước đó (cả rg_start và rg_end bằng 0), trả về -1.

Tạo một vùng nhớ mới free_rg để lưu lại các giá trị của vùng nhớ cần giải phóng.

Đặt lại giá trị của rgnode để chỉ ra rằng vùng nhớ đã được giải phóng.

Thêm free_rg vào danh sách vùng nhớ trống.

```c
int __free(struct pcb_t *caller, int vmaid, int rgid)
{
  // pthread_mutex_lock(&mmvm_lock);

  if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
  {
    // pthread_mutex_unlock(&mmvm_lock);
    return -1;
  }

  /* TODO: Manage the collect freed region to freerg_list */
  struct vm_rg_struct *rgnode = get_symrg_byid(caller->mm, rgid);

  if (rgnode->rg_start == 0 && rgnode->rg_end == 0)
  {
    // pthread_mutex_unlock(&mmvm_lock);
    return -1;
  }
  struct vm_rg_struct free_rg;
  free_rg.rg_start = rgnode->rg_start;
  free_rg.rg_end = rgnode->rg_end;
  free_rg.rg_next = NULL;

  rgnode->rg_start = rgnode->rg_end = 0;
  rgnode->rg_next = NULL;

  /*enlist the obsoleted memory region */
  enlist_vm_freerg_list(caller->mm, free_rg);

  // pthread_mutex_unlock(&mmvm_lock);
  return 0;
}
```

Code hàm __free()

- Hàm pg_getpage() :

Hàm này lấy trang từ RAM dựa trên số trang (pagenum) và trả về số khung (framenum).

Đầu tiên hàm sẽ tìm trang trong TLB,nếu tìm được hàm sẽ trở về 0 để báo thành công .

Nếu không hàm sẽ tiếp tục tìm kiếm trong page table.

Nếu trang không có trong page table,hàm sẽ ra không gian MEMSWAP tìm kiếm frame number cần thiết để thay vào. Macro PAGING_SWP để tìm frame num cần thiết, macro PAGING_SWPTYP để xác định frame num đó đang ở MEMSWP nào

Kiểm tra xem bộ nhớ chính RAM còn frame trống không,nếu còn thì chỉ việc thay frame number cần thiết vào.Nếu không thì phải dùng hàm find_victim_page để tìm trang thay thế,thay victim page ra MEMSWP và thế frame number cần thiết vào RAM.

Nếu MEMSWAP hiện tại (active_mswp) không còn chỗ trống,hàm sẽ tìm kiếm MEMSWAP nào còn trống để thay victim page ra,sau đó cập nhật active_mswp.

Sau đó chuyển frame num từ MEMSWP vào bộ nhớ chính RAM.

Cập nhật bảng trang với frame num mới và cập nhật trang bị thay thế ra MEMSWP.

Cập nhật frame mới vào TLB và kết thúc hàm.

```c
int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller)
{
#ifdef CPU_TLB
  if (tlb_cache_read_123(caller->pid, pgn, fpn) != -1)
  {
    return 0;
  }
#endif
  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 tgtswp_typ = PAGING_SWPTYP(pte);
    int tgtfpn = PAGING_SWP(pte); // the target frame storing our variable
    /* TODO: Play with your paging theory here */
    /* Find victim page */
    // MOI ADD
    struct memphy_struct *mswp = (struct memphy_struct *)caller->mswp;
    if (MEMPHY_get_freefp(caller->mram, &swpfpn) == 0)
    {
      __swap_cp_page(mswp + tgtswp_typ, tgtfpn, caller->mram, swpfpn);
      pte_set_fpn(&pte, tgtfpn);
      enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
      *fpn = PAGING_FPN(pte);
#ifdef CPU_TLB
      /* Update its online status of TLB (if needed) */
      tlb_cache_write_123(caller->pid, pgn, *fpn);
#endif
      return 0;
    }
    // MOI ADD
    if (find_victim_page(caller->mm, &vicpgn) < 0)
      return -1;
    while (vicpgn == pgn)
```

```c
while (vicpgn == pgn)
{
  if (find_victim_page(caller->mm, &vicpgn) < 0)
    return -1;
}
vicpte = caller->mm->pgd[vicpgn];
vicfpn = PAGING_FPN(vicpte);
/* Get free frame in MEMSWP */
/* Do swap frame from MEMRAM to MEMSWP and vice versa*/
/* Copy victim frame to swap */
//__swap_cp_page();

if (MEMPHY_get_freefp(caller->active_mswp, &swpfpn) == 0)
{
  __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
}
else
{
  struct memphy_struct **mswpid = caller->mswp;
  for (int i = 0; i < PAGING_MAX_MMSWP; i++)
  {
    struct memphy_struct *new_mswp = (struct memphy_struct *)(mswpid);
    if (MEMPHY_get_freefp(new_mswp + i, &swpfpn) == 0)
    {
      __swap_cp_page(caller->mram, vicfpn, new_mswp + i, swpfpn);
      caller->active_mswp = new_mswp + i;
      break;
    }
  }
}
/* Copy target frame from swap to mem */
//__swap_cp_page();
__swap_cp_page(mswp + tgtswp_typ, tgtfpn, caller->mram, vicfpn);
MEMPHY_put_freefp(mswp + tgtswp_typ, tgtfpn);
/* Update page table */
// pte_set_swap() &mm->pgd;
```

Ln 23

```
    __swap_cp_page(mswp + tgtswp_typ, tgtfpn, caller->mram, vicfpn);
    MEMPHY_put_freefp(mswp + tgtswp_typ, tgtfpn);
    /* Update page table */
    // pte_set_swap() &mm->pgd;
    int swp_index = 0;
    for (swp_index = 0; swp_index < PAGING_MAX_MMSWP; swp_index++)
    {
      if (mswp + swp_index == caller->active_mswp)
        break;
    }
    pte_set_swap(&caller->mm->pgd[vicpgn], swp_index, swpfpn);
    // pte_set_fpn(&caller->mm->pgd[pgn],vicfpn); // trong code mau
    /* Update its online status of the target page */
    // pte_set_fpn() & mm->pgd[pgn];
    pte_set_fpn(&pte, tgtfpn);

    enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
  }

  *fpn = PAGING_FPN(pte);
#ifdef CPU_TLB
  /* Update its online status of TLB (if needed) */
  tlb_cache_write_123(caller->pid, pgn, *fpn);
#endif
  return 0;
}
```

Code hàm pg_getpage()

- Hàm validate_overlap_vm_area :

Hàm này kiểm tra xem vùng nhớ dự định cấp phát có bị chồng lấn với bất kỳ vùng nhớ ảo hiện có nào không.

Dùng vòng lặp for để kiểm tra chồng lấn. Nếu ID của vùng nhớ ảo hiện tại khác với ID của vùng nhớ ảo dự định cấp phát (vmaid), Sử dụng macro OVERLAP để kiểm tra xem vùng nhớ dự định (vmastart, vmaend) có chồng lấn với vùng nhớ hiện tại (vma->vm_start, vma->vm_end) hay không.

Tiếp tục kiểm tra các trường hợp biên, chẳng hạn như khu vực mới cần cấp phát có bắt đầu hoặc kết thúc ngay tại ranh giới của một khu vực bộ nhớ hiện có hay không. Nếu có, hàm trả về -1 để báo hiệu trùng lặp.

Chuyển đến vùng nhớ ảo tiếp theo trong danh sách và lặp lại quá trình kiểm tra. Nếu không có sự chồng lấn nào được phát hiện, hàm trả về 0 để chỉ ra rằng vùng nhớ dự định hợp lệ.

```
int validate_overlap_vm_area(struct pcb_t *caller, int vmaid, int vmastart, int vmaend)
{
  struct vm_area_struct *vma = caller->mm->mmap;

  /* TODO validate the planned memory area is not overlapped */
  while (vma != NULL)
  {
    if (vmaid != vma->vm_id)
    {
      if (OVERLAP(vmastart, vmaend, vma->vm_start, vma->vm_end))
      {
        if ((vmastart != vma->vm_end || vmaend == vma->vm_start) && (vmastart == vma->vm_end || vmaend != vma->vm_start))
          return -1;
      }
    }
    vma = vma->vm_next;
  }
  return 0;
}
```

Code hàm validate_overlap_vm_area()

- Hàm find_victim_page() :

Hàm này tìm trang nạn nhân trong bộ nhớ chính để thay thế theo cơ chế FIFO.

```
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)
    return -1;
  while (pg->pg_next && pg->pg_next->pg_next)
    pg = pg->pg_next;
  if (!pg->pg_next)
  {
    *retpgn = pg->pgn;
    free(pg);
    pg = mm->fifo_pgn = NULL;
  }
  else
  {
    *retpgn = pg->pg_next->pgn;
    free(pg->pg_next);
    pg->pg_next = NULL;
  }

  return 0;
}
```

Code hàm find_victim_page()

2.3 Hiện thực TLB

Ở phần này,ta sẽ tập trung vào việc xây dựng cơ chế quản lý trang TLB.

Đầu tiên , khai báo 1 struct tlb_entry , 1 mảng động global tlb_table và các hàm cần thiết :

```
typedef struct
{
   int page;
   int frame;
   int valid;
   uint32_t pid;
   int last_used;
} tlb_entry;
```

```
extern int TLB_SIZE;
extern tlb_entry* tlb_table;
void init_tlb_table(int tlbsz);
void print_check_tlb_table(void);
int check_in_tlb(int pid,int pgn);
int tlb_cache_read_123(int pid, int pgn, int *fpn);
int tlb_cache_write_123(int pid, int pgn, int fpn);
```

Mục đích dùng mảng động là để dễ lắp đặt,quản lý các trang chung cho tất cả các process . Ở hàm khởi tạo tlb_table ,ta tính kích thước TLB_SIZE dựa theo công thức :

cpu-tlb-cache.c :

- Hàm init_tlb_table() :

```
static int time_access = 0;
int TLB_SIZE;
tlb_entry* tlb_table;
void init_tlb_table(int tlbsz)
{
   TLB_SIZE= tlbsz/PAGING_PAGESZ;
   tlb_table = (tlb_entry*) malloc(TLB_SIZE * sizeof(tlb_entry));
    if (tlb_table == NULL) {
        perror("Failed to allocate memory for TLB table");
        exit(EXIT_FAILURE);
    }
   for (int i = 0; i < TLB_SIZE; ++i)
   {
      tlb_table[i].page = -1;
      tlb_table[i].frame = -1;
      tlb_table[i].valid = 0;
      tlb_table[i].pid = -1;
      tlb_table[i].last_used = 0;
   }
}
```

Ta xây dựng TLB quản lý trang theo cơ chế Least Recently Used (LRU) nên có 1 biến tĩnh time_access chỉ thời gian được truy cập gần đây nhất.

- Hàm check_in_tlb() : Dùng để kiểm tra xem page và pid tương ứng đã có trong TLB hay không

```
int check_in_tlb(int pid,int pgn){
    for(int i=0;i<TLB_SIZE;i++){
        if(tlb_table[i].valid == 1 && tlb_table[i].page == pgn && tlb_table[i].pid == pid ){
            return i;
        }
    }
    return -1;
}
```

- Hàm tlb_cache_read_123() :

Hàm này tìm trang cần đọc trong TLB,nếu tìm thấy sẽ cho biết frame number tương ứng,đồng thời cập nhật thời gian truy cập gần đây (last_used).

```
int tlb_cache_read_123(int pid, int pgn, int *fpn)
{
    for (int i = 0; i < TLB_SIZE; i++)
    {
        if (tlb_table[i].valid == 1 && tlb_table[i].page == pgn && tlb_table[i].pid == pid)
        {
            *fpn = tlb_table[i].frame;
            tlb_table[i].last_used = ++time_access;
            return i; // return ve vi tri doc dc
        }
    }
    return -1;
}
/*
```

- Hàm tlb_cache_write_123() :

Hàm này cập nhật trang mới được truy cập vào TLB.

Đầu tiên hàm kiểm tra xem trang đã có sẵn trong TLB hay không ,nếu có chỉ cần cập nhật last_used và return 0 để báo thành công.

Nếu không,hàm dùng vòng lặp for để tìm kiếm trang có last_used nhỏ nhất (đồng nghĩa với việc chưa được truy cập lâu nhất) và cập nhật pgn,fpn mới vào vị trí trang đó.

```c
int tlb_cache_write_123(int pid, int pgn, int fpn)
{
    int k=check_in_tlb(pid,pgn);
    if(k!=-1){
        tlb_table[k].last_used= ++time_access;
        return 0;
    }
    // LRU
    int lru_id = 0;
    int min_time_access = tlb_table[0].last_used;
    for (int i = 1; i < TLB_SIZE; i++)
    {
        if (tlb_table[i].last_used < min_time_access)
        {
            lru_id = i;
            min_time_access = tlb_table[i].last_used;
        }
    }
    tlb_table[lru_id].page = pgn;
    tlb_table[lru_id].frame = fpn;
    tlb_table[lru_id].pid = pid;
    tlb_table[lru_id].valid = 1;
    tlb_table[lru_id].last_used = ++time_access;
    return 0;
}
```

- Hàm print_check_tlb_table() : Hiển thị các trang được lưu trong TLB

```c
void print_check_tlb_table(){
   printf("Check tlb table :\n");
   for(int i=0;i<TLB_SIZE;i++){
      if(tlb_table[i].valid){
         printf("PID: %u; PGNUM: %d; FRMNUM: %d; LRU: %d\n",tlb_table[i].pid,tlb_table[i].page,tlb_table[i].frame,tlb_table[i].last_used)
      }
   }
}
```

cpu-tlb.c :

- Hàm tlballoc() :

Hàm tlballoc thực hiện cấp phát một vùng nhớ cho tiến trình proc với kích thước size . Sử dụng hàm __alloc để cấp phát thực tế.

Tính số page tương ứng với size cần cấp phát,sau đó lưu thông tin các page đó vào TLB.

```
int tlballoc(struct pcb_t *proc, uint32_t size, uint32_t reg_index)
{
  int addr, val;

  /* By default using vmaid = 0 */
  val = __alloc(proc, 0, reg_index, size, &addr);

  /* TODO update TLB CACHED frame num of the new allocated page(s)*/
  /* by using tlb_cache_read()/tlb_cache_write()*/

  if (val < 0)
    return -1;
  pthread_mutex_lock(&tlb_lock);
  int pgn = PAGING_PGN(addr);

  int pgnum = PAGING_PAGE_ALIGNSZ(size) / PAGING_PAGESZ;
  for (int i = 0; i < pgnum; i++)
  {
    uint32_t pte = proc->mm->pgd[pgn + i];
    int fpn = PAGING_FPN(pte);
    tlb_cache_write_123(proc->pid, pgn + i, fpn);
  }
  pthread_mutex_unlock(&tlb_lock);
  return 0;
}
```

- Hàm tlbfree() :

Hàm tlbfree_data giải phóng vùng nhớ đã được cấp phát cho một tiến trình.Gọi hàm __free() để giải phóng vùng nhớ thực tế cho tiến trình proc dựa trên reg_index.

Dựa trên chỉ số reg_index, hàm xác định địa chỉ bắt đầu (start) và kết thúc (end) của vùng nhớ cần giải phóng. Từ đó, tính toán số trang (page) cần giải phóng.

Duyệt qua các trang và kiểm tra xem trang đó có trong TLB hay không. Nếu có, các mục tương ứng trong TLB sẽ được đặt lại giá trị không hợp lệ.

```
int tlbfree_data(struct pcb_t *proc, uint32_t reg_index)
{

  __free(proc, 0, reg_index);

  /* TODO update TLB CACHED frame num of freed page(s)*/
  /* by using tlb_cache_read()/tlb_cache_write()*/

  pthread_mutex_lock(&tlb_lock);
  int start = proc->mm->symrgtbl[reg_index].rg_start;
  int end = proc->mm->symrgtbl[reg_index].rg_end;
  int size = end = start;
  int pgn = PAGING_PGN(start);
  int pgnum = PAGING_PAGE_ALIGNSZ(size) / PAGING_PAGESZ;
  for (int i = 0; i < pgnum; i++)
  {
    int t = check_in_tlb(proc->pid, pgn + i);
    if (t >= 0)
    {
      tlb_table[i].page = -1;
      tlb_table[i].frame = -1;
      tlb_table[i].last_used = 0;
      tlb_table[i].valid = 0;
    }
  }
  pthread_mutex_unlock(&tlb_lock);
  return 0;
}
```

- Hàm tlbread() :

Hàm tlbread đọc dữ liệu từ vùng nhớ của một tiến trình. Tính địa chỉ cần đọc dựa trên source và offset.

Sử dụng tlb_cache_read_123 để kiểm tra xem trang này có trong TLB không. Nếu có, frmnum sẽ là số khung (frame) của trang trong TLB. In ra kết quả kiểm tra TLB: Nếu IODUMP được định nghĩa, in ra kết quả kiểm tra.

Gọi hàm __read để đọc dữ liệu từ bộ nhớ và lưu vào destination.

```
int tlbread(struct pcb_t *proc, uint32_t source,
            uint32_t offset, uint32_t destination)
{
  BYTE data;
  int frmnum = -1;
  /* TODO retrieve TLB CACHED frame num of accessing page(s)*/
  /* by using tlb_cache_read()/tlb_cache_write()*/
  /* frmnum is return value of tlb_cache_read/write value*/

  pthread_mutex_lock(&tlb_lock);
  struct vm_rg_struct *currg = get_symrg_byid(proc->mm, source);
  int addr = currg->rg_start + offset;
  int pgn = PAGING_PGN(addr);
  tlb_cache_read_123(proc->pid, pgn, &frmnum);
  print_check_tlb_table();
#ifdef IODUMP
  if (frmnum >= 0)
    printf("TLB hit at read region=%d offset=%d\n",
           source, offset);
  else
    printf("TLB miss at read region=%d offset=%d\n",
           source, offset);
#ifdef PAGETBL_DUMP
  print_pgtbl(proc, 0, -1); // print max TBL
#endif
  MEMPHY_dump(proc->mram);
#endif
  pthread_mutex_unlock(&tlb_lock);
  int val = __read(proc, 0, source, offset, &data);

  destination = (uint32_t)data;

  /* TODO update TLB CACHED with frame num of recent accessing page(s)*/
  /* by using tlb_cache_read()/tlb_cache_write()*/

  return val;
}
```

- Hàm tlbwrite() :

Hàm tlbwrite ghi dữ liệu vào vùng nhớ của một tiến trình. Tính địa chỉ cần ghi dựa trên destination và offset.

Sử dụng tlb_cache_read_123 để kiểm tra xem trang này có trong TLB không. Nếu có, frmnum sẽ là số khung (frame) của trang trong TLB.In ra kết quả kiểm tra TLB: Nếu IODUMP được định nghĩa, in ra kết quả kiểm tra.

Gọi hàm __write để ghi dữ liệu vào bộ nhớ.

```c
int tlbwrite(struct pcb_t *proc, BYTE data,
             uint32_t destination, uint32_t offset)
{
  int val;
  int frmnum = -1;
  /* TODO retrieve TLB CACHED frame num of accessing page(s))*/
  /* by using tlb_cache_read()/tlb_cache_write()
  frmnum is return value of tlb_cache_read/write value*/
  pthread_mutex_lock(&tlb_lock);
  struct vm_rg_struct *currg = get_symrg_byid(proc->mm, destination);
  int addr = currg->rg_start + offset;
  int pgn = PAGING_PGN(addr);
  tlb_cache_read_123(proc->pid, pgn, &frmnum);
  print_check_tlb_table();
#ifdef IODUMP
  if (frmnum >= 0)
    printf("TLB hit at write region=%d offset=%d value=%d\n",
           destination, offset, data);
  else
    printf("TLB miss at write region=%d offset=%d value=%d\n",
           destination, offset, data);
#ifdef PAGETBL_DUMP
  print_pgtbl(proc, 0, -1); // print max TBL
#endif
  MEMPHY_dump(proc->mram);
#endif
  pthread_mutex_unlock(&tlb_lock);
  val = __write(proc, 0, destination, offset, data);

  /* TODO update TLB CACHED with frame num of recent accessing page(s)*/
  /* by using tlb_cache_read()/tlb_cache_write()*/

  return val;
}
```

Testcase os_1_tlbsz_singleCPU_mlq :

input > os_1_tlbsz_singleCPU_mlq

```
2 1  8
40000
1 s4   4
2 s3   3
4 m1s  2
6 s2   3
7 m0s  3
9 p1s  2
11 s0  1
16 s1  0

```

Output của testcase :

```
Time slot  60
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
Check tlb table :
PID: 3; PGNUM: 0; FRMNUM: 1; LRU: 4
PID: 3; PGNUM: 1; FRMNUM: 0; LRU: 3
PID: 8; PGNUM: 0; FRMNUM: 3; LRU: 5
PID: 8; PGNUM: 1; FRMNUM: 2; LRU: 6
PID: 5; PGNUM: 0; FRMNUM: 5; LRU: 10
PID: 5; PGNUM: 1; FRMNUM: 4; LRU: 11
TLB hit at write region=1 offset=20 value=102
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Tracing memory content - (index,content):
Time slot  61
Check tlb table :
PID: 3; PGNUM: 0; FRMNUM: 1; LRU: 4
PID: 3; PGNUM: 1; FRMNUM: 0; LRU: 3
PID: 8; PGNUM: 0; FRMNUM: 3; LRU: 5
PID: 8; PGNUM: 1; FRMNUM: 2; LRU: 6
PID: 5; PGNUM: 0; FRMNUM: 5; LRU: 10
PID: 5; PGNUM: 1; FRMNUM: 4; LRU: 12
TLB miss at write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Tracing memory content - (index,content): (1088 ; 102)
Time slot  62
        CPU 0: Put process  5 to run queue
```

```
00000004: 80000004
Tracing memory content - (index,content): (1088 ; 102)
Time slot  62
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  2
Time slot  63
Time slot  64
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  4
Time slot  65
Time slot  66
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
Check tlb table :
PID: 3; PGNUM: 0; FRMNUM: 1; LRU: 4
PID: 3; PGNUM: 1; FRMNUM: 0; LRU: 3
PID: 8; PGNUM: 0; FRMNUM: 3; LRU: 5
PID: 8; PGNUM: 1; FRMNUM: 2; LRU: 6
PID: 5; PGNUM: 0; FRMNUM: 5; LRU: 14
PID: 5; PGNUM: 1; FRMNUM: 4; LRU: 12
PID: 5; PGNUM: 3; FRMNUM: 0; LRU: 13
TLB hit at write region=0 offset=0 value=0
print_pgtbl: 0 - 512
00000000: 80000005
00000004: 80000004
Tracing memory content - (index,content): (232 ; 1) (1088 ; 102)
Time slot  67
        CPU 0: Processed  5 has finished
        CPU 0: Dispatched process  2
Time slot  68
```

Theo output ta có thể thấy có 2 lần TLB hit và 1 lần TLB miss. Vậy hit ratio là 2/3 .

Đánh giá :

- Hiệu suất hiện tại của TLB với hit ratio là 2/3 là khá tốt, có thể giảm số lần tra cứu page table và và làm tăng hiệu suất tổng thể của hệ thống.
- Có 1 số phương pháp cải tiến hiệu suất TLB như : tăng kích thước mảng TLB,sử dụng thuật toán thay thế hiệu quả hơn như LFU (Least Frequently Used) hoặc ARC (Adaptive Replacement Cache,kết hợp giữa LRU và LFU).

3. Put it all together

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

- Nếu việc đồng bộ hóa không được xử lý trong hệ điều hành đơn giản của bạn, nhiều vấn đề có thể xảy ra, gây ảnh hưởng đến hoạt động của hệ thống. Một số vấn đề có thể phát sinh bao gồm:

1. Race Conditions: Các điều kiện tranh chấp xảy ra khi hai hoặc nhiều tiến trình cố gắng truy cập hoặc sửa đổi một tài nguyên chung cùng lúc mà không có biện pháp đồng bộ hóa. Điều này có thể dẫn đến lỗi hoặc dữ liệu không nhất quán. Ví dụ, nếu hai tiến trình cùng tăng giá trị của một biến chung, kết quả cuối cùng có thể không chính xác nếu không có khóa bảo vệ.
2. Deadlock: Deadlock xảy ra khi hai hoặc nhiều tiến trình đang chờ nhau giải phóng tài nguyên và không có tiến trình nào có thể tiếp tục. Điều này thường xảy ra khi các tiến trình chiếm giữ tài nguyên theo thứ tự khác nhau mà không có cơ chế đồng bộ hóa phù hợp. Ví dụ, tiến trình A giữ khóa X và chờ khóa Y, trong khi tiến trình B giữ khóa Y và chờ khóa X, dẫn đến bế tắc.
3. Inconsistent State: Nếu không có đồng bộ hóa, các tài nguyên chung có thể trở nên không nhất quán. Ví dụ, nếu nhiều tiến trình truy cập một hàng đợi chia sẻ mà không khóa, các mục có thể bị ghi đè hoặc bị mất, dẫn đến trạng thái không chính xác hoặc bị hỏng.

→ Để giải quyết các vấn đề trên,ta có thể sử dụng các cơ chế đồng bộ hóa như mutex-lock,semaphore,spin-lock…

4. Kết luận

Thông qua bài tập lớn, nhóm đã hiểu được cơ chế của một hệ điều hành đơn giản, đặc biệt là ở các cơ chế định thời CPU, việc phân chia phân đoạn xen lẫn paging để quản lí, tối ưu bộ nhớ, việc cấp phát và thu hồi bộ nhớ và khả năng đồng bộ để quản lí dữ liệu một cách hiệu quả.

Trong quá trình thực hiện bài tập lớn, tuy đã được học hỏi và thấu hiểu nhiều hơn về các cơ chế của hệ điều hành, sự thiếu kinh nghiệm, hiểu biết và gây ra sai sót là khó tránh khỏi. Do đó, việc bài tập lớn không được hoàn hảo là tất yếu cộng thêm việc lí luận các câu hỏi sẽ không được tốt. Ở khía cạnh này, nhóm em mong sẽ nhận được góp ý và từ đó tích lũy kinh nghiệm để trở nên hoàn thiện, hiểu biết hơn và hoàn thành tốt những bài tập sau này.

Cuối cùng, nhóm em xin thành thật cảm ơn các giảng viên, các bạn sinh viên đã cùng nhau giúp đỡ, giải đáp các khúc mắc và cung cấp kiến thức để có thể hoàn thành tốt bài tập lớn của môn Hệ Điều Hành học kì này.

5. Video thuyết trình

Assignment

Lab 4

6. Tài liệu tham khảo

[1]: Thầy Nguyễn Quang Hùng - Slide Hệ Điều Hành, học kì 232

[2]: Abraham Silberschatz - Operating System Concepts – 10th

Bảng phân công công việc:

Họ và tên	Công việc	Đóng góp
Nguyễn Thiên Hải	Làm report + trả lời câu hỏi	100%
Nguyễn Lê Minh Huân	Làm report + trả lời câu hỏi	100%
Võ Nguyễn Gia Huy	Code phần quản lí bộ nhớ	100%
Lê Hải Long	Code phần quản lí bộ nhớ	100%
Vương Thanh Phương	Code phần Scheduling	100%