VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HCM UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

OPERATING SYSTEM (CO2017)

Assignment

Simple Operating System

TA: La Quoc Nhut Huan
Group: Do Chi Khai - 1913769
Mai Nhu Anh Khoa - 2110274
Class: L04

Ho Chi Minh City, 11/2022

Member list and contribution:

No	Name	ID Number	Contribution
1	Do Chi Khai	1913769	Scheduling
2	Mai Nhu Anh Khoa	2110274	Memory

Contents

1 Scheduling

1.1 Question - Multi-level Queue

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

Answer:

Giải thuật MLQ (Multi-level queue) sử dụng cơ chế chia các process với các độ ưu tiên khác nhau vào mỗi ready_queue tương ứng, với ready_queue có độ ưu tiên thấp nhất là MAX_PRIO:

- *mlq_ready_queue*: một danh sách (array) với phần tử là các queue với độ ưu tiên từ cao đến thấp là 0 -> MAX_PRIO.
- *mlq_ready_queue[prio]*: hàng đợi với độ ưu tiên của từng phần tử là prio.
- *curr_slot*: một array lưu các slot thực thi còn lại của mỗi hàng đợi, với slot = MAX_PRIO - prio.

Từ đó ta có thể thấy ưu điểm của giải thuật MLQ đó là:

- Sử dụng cơ chế slot, tránh được starvation, nhưng đồng thời vẫn đảm bảo được các process có độ ưu tiên cao hơn sẽ được thực thi thường xuyên hơn.
- Với việc mỗi process được gán cố định vào một hàng đợi nhất định, việc enqueue cũng như dequeue sẽ tốn ít chi phí hơn.

1.2 Implementation

Enqueue và Dequeue

- Với enqueue, ta kiểm tra xem queue đã đạt giới hạn chưa, nếu chưa, ta thêm vào vị trí cuối cùng.
- Với dequeue, trước tiên kiểm tra xem queue có rỗng hay không, rồi ta lấy ra phần tử đầu tiên của queue, sau đó cập nhật lại vị trí của từng phần tử.

Hiện thực:

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

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(!empty(q)) {
      struct pcb_t *proc = q->proc[0];
      for(int i = 0; i < q->size-1; i++) {
         q->proc[i] = q->proc[i+1];
      }
      q->proc[q->size-1] = NULL;
      q->size--;
      return proc;
   }
   return NULL;
}
```

Get process

- Ta hiện thực giải thuật MLQ bằng cách, định nghĩa 1 danh sách chứa các slot còn lại của các hàng đợi.
- Hàm *refresh_slot()* dùng để init danh sách và refresh slot của các queue khi tất cả các queue đều đã hết slot.

Hiện thực:

```
#ifdef MLQ_SCHED
static struct queue_t mlq_ready_queue[MAX_PRIO];
static int curr_slot[MAX_PRIO];
#endif
```

```
void refresh_slot(void) {
   for(int i = 0; i < MAX_PRIO; i++)
      curr_slot[i] = MAX_PRIO - i;
}

void init_scheduler(void) {
#ifdef MLQ_SCHED
    int i ;

   for (i = 0; i < MAX_PRIO; i ++)
      mlq_ready_queue[i].size = 0;
   refresh_slot();
#endif
   ready_queue.size = 0;
   run_queue.size = 0;
   pthread_mutex_init(&queue_lock, NULL);
}
```

- Đối với hàm *get_proc()*, trước tiên ta kiểm tra xem hàng đợi nào có độ ưu tiên cao nhất, và lưu prio vào biến nonEmptyQ.
- Sau đó kiểm tra xem hàng đợi đó còn slot để thực thi không, nếu không, qua hàng đợi kế tiếp, nếu còn, thì trừ 1 slot và trả về process đó.
- Nếu sau khi duyệt tất cả queue nhưng đều hết slot, thì ta refresh_slot, và lấy process từ hàng đợi có độ ưu tiên cao nhất vẫn còn slot.

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

   int  nonEmptyQ = -1;
   int i;
   pthread_mutex_lock(&queue_lock);
   for(i = 0; i < MAX_PRIO; i++){
      //Check if queue is empty
      if(empty(&mlq_ready_queue[i]))
```

```
            continue;

        //If queue isnt empty, store the highest priority idx
        if(nonEmptyQ == -1){
            nonEmptyQ = i;
        }

        //Check if queue still have slot;
        if(curr_slot[i] == 0)
            continue;

        proc = dequeue(&mlq_ready_queue[i]);
        curr_slot[i]--;
        pthread_mutex_unlock(&queue_lock);
        return proc;
    }

    //If all queue are empty or all out of slots, refresh slots
    refresh_slot();

    //Check if there exist a queue with high priority and not empty
    if(nonEmptyQ != -1)
    {
        proc = dequeue(&mlq_ready_queue[nonEmptyQ]);
        curr_slot[nonEmptyQ]--;
    }

    pthread_mutex_unlock(&queue_lock);
    return proc;
}
```

Chạy thử

```
------ SCHEDULING TEST 1 --------------------------------------------
./os sched_1
Time slot  0
       Loaded a process at input/proc/s0, PID: 1 PRIO: 4
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: 3
Time slot  5
       CPU 0: Put process 1 to run queue
       CPU 0: Dispatched process 2
Time slot  6
       Loaded a process at input/proc/s2, PID: 3 PRIO: 3
Time slot  7
       CPU 0: Put process 2 to run queue
       CPU 0: Dispatched process 3
       Loaded a process at input/proc/s3, PID: 4 PRIO: 0
Time slot  8
Time slot  9
       CPU 0: Put process 3 to run queue
       CPU 0: Dispatched process 4
Time slot 10
Time slot 11
       CPU 0: Put process 4 to run queue
       CPU 0: Dispatched process 4
Time slot 12
Time slot 13
       CPU 0: Put process 4 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 4
Time slot 16
Time slot 17
       CPU 0: Put process 4 to run queue
       CPU 0: Dispatched process 4
Time slot 18
Time slot 19
       CPU 0: Put process 4 to run queue
       CPU 0: Dispatched process 4
Time slot 20
```

```
        CPU 0: Processed 4 has finished
        CPU 0: Dispatched process 2
Time slot 21
Time slot 22
        CPU 0: Put process 2 to run queue
        CPU 0: Dispatched process 3
Time slot 23
Time slot 24
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 1
Time slot 25
Time slot 26
        CPU 0: Put process 1 to run queue
        CPU 0: Dispatched process 2
Time slot 27
Time slot 28
        CPU 0: Put process 2 to run queue
        CPU 0: Dispatched process 3
Time slot 29
Time slot 30
        CPU 0: Put process 3 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 2
Time slot 33
        CPU 0: Processed 2 has finished
        CPU 0: Dispatched process 3
Time slot 34
Time slot 35
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 1
Time slot 36
Time slot 37
        CPU 0: Put process 1 to run queue
        CPU 0: Dispatched process 3
Time slot 38
Time slot 39
```

```
        CPU 0: Put process 3 to run queue
        CPU 0: Dispatched process 3
Time slot 40
Time slot 41
        CPU 0: Processed 3 has finished
        CPU 0: Dispatched process 1
Time slot 42
Time slot 43
        CPU 0: Put process 1 to run queue
        CPU 0: Dispatched process 1
Time slot 44
Time slot 45
        CPU 0: Put process 1 to run queue
        CPU 0: Dispatched process 1
Time slot 46
        CPU 0: Processed 1 has finished
        CPU 0 stopped
```

1.3 Gantt Diagram

	p1				p2		p3		p4		
0	1	2	3	4	5	6	7	8	9	10	11

p4								p2		p3	
12	13	14	15	16	17	18	19	20	21	22	23

p1		p2		p3		p1		p2	p3		p1
24	25	26	27	28	29	30	31	32	33	34	35

p1	p3				p1						
36	37	38	39	40	41	42	43	44	45		

2 Memory

2.1 Question - Segmentation with paging

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

Answer:

Ưu điểm của giải thuật:

- Tiết kiệm bộ nhớ, sử dụng bộ nhớ hiệu quả.
- Mang các ưu điểm của giải thuật phân trang: đơn giản việc cấp phắt vùng nhớ và khắc phục được phân mảnh ngoại.
- Giải quyết vấn đề phân mảnh ngoại của giải thuật phân đoạn bằng cách phân trang trong mỗi đoạn.

Nhược điểm của giải thuật:

- Phân mảnh nội của giải thuật phân trang vẫn còn.

2.2 Result - Status of RAM

Requirement: Show the status of RAM after each memory allocation and deallocation function call.

Dưới đây là kết quả của quá trình ghi log sau mỗi lệnh allocation và deallocation trong chương trình, cụ thể ghi lại trạng thái của RAM trong chương trình ở mỗi bước.

Test 0:

——————Allocation——————

000: 00000-003ff - PID: 01 (idx 000, nxt: 001)

001: 00400-007ff - PID: 01 (idx 001, nxt: 002)

002: 00800-00bff - PID: 01 (idx 002, nxt: 003)

003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)

004: 01000-013ff - PID: 01 (idx 004, nxt: 005)

005: 01400-017ff - PID: 01 (idx 005, nxt: 006)

006: 01800-01bff - PID: 01 (idx 006, nxt: 007)

007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)

008: 02000-023ff - PID: 01 (idx 008, nxt: 009)

009: 02400-027ff - PID: 01 (idx 009, nxt: 010)

010: 02800-02bff - PID: 01 (idx 010, nxt: 011)

011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)

012: 03000-033ff - PID: 01 (idx 012, nxt: 013)

013: 03400-037ff - PID: 01 (idx 013, nxt: -01)

————————Allocation————————

000: 00000-003ff - PID: 01 (idx 000, nxt: 001)

001: 00400-007ff - PID: 01 (idx 001, nxt: 002)

002: 00800-00bff - PID: 01 (idx 002, nxt: 003)

003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)

004: 01000-013ff - PID: 01 (idx 004, nxt: 005)

005: 01400-017ff - PID: 01 (idx 005, nxt: 006)

006: 01800-01bff - PID: 01 (idx 006, nxt: 007)

007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)

008: 02000-023ff - PID: 01 (idx 008, nxt: 009)

009: 02400-027ff - PID: 01 (idx 009, nxt: 010)

010: 02800-02bff - PID: 01 (idx 010, nxt: 011)

011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)

012: 03000-033ff - PID: 01 (idx 012, nxt: 013)

013: 03400-037ff - PID: 01 (idx 013, nxt: -01)

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)

015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)

————————Deallocation————————

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)

015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)

————————Allocation————————

000: 00000-003ff - PID: 01 (idx 000, nxt: 001)

001: 00400-007ff - PID: 01 (idx 001, nxt: -01)

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)

015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)

————————Allocation————————

000: 00000-003ff - PID: 01 (idx 000, nxt: 001)

001: 00400-007ff - PID: 01 (idx 001, nxt: -01)

002: 00800-00bff - PID: 01 (idx 000, nxt: 003)

003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)

004: 01000-013ff - PID: 01 (idx 002, nxt: 005)

005: 01400-017ff - PID: 01 (idx 003, nxt: 006)

006: 01800-01bff - PID: 01 (idx 004, nxt: -01)

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)

015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)

Test 1:

——————Allocation——————

000: 00000-003ff - PID: 01 (idx 000, nxt: 001)

001: 00400-007ff - PID: 01 (idx 001, nxt: 002)

002: 00800-00bff - PID: 01 (idx 002, nxt: 003)

003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)

004: 01000-013ff - PID: 01 (idx 004, nxt: 005)

005: 01400-017ff - PID: 01 (idx 005, nxt: 006)

006: 01800-01bff - PID: 01 (idx 006, nxt: 007)

007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)

008: 02000-023ff - PID: 01 (idx 008, nxt: 009)

009: 02400-027ff - PID: 01 (idx 009, nxt: 010)

010: 02800-02bff - PID: 01 (idx 010, nxt: 011)

011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)

012: 03000-033ff - PID: 01 (idx 012, nxt: 013)

013: 03400-037ff - PID: 01 (idx 013, nxt: -01)

——————Allocation——————

000: 00000-003ff - PID: 01 (idx 000, nxt: 001)

001: 00400-007ff - PID: 01 (idx 001, nxt: 002)

002: 00800-00bff - PID: 01 (idx 002, nxt: 003)

003: 00c00-00fff - PID: 01 (idx 003, nxt: 004)

004: 01000-013ff - PID: 01 (idx 004, nxt: 005)

005: 01400-017ff - PID: 01 (idx 005, nxt: 006)

006: 01800-01bff - PID: 01 (idx 006, nxt: 007)

007: 01c00-01fff - PID: 01 (idx 007, nxt: 008)

008: 02000-023ff - PID: 01 (idx 008, nxt: 009)

009: 02400-027ff - PID: 01 (idx 009, nxt: 010)

010: 02800-02bff - PID: 01 (idx 010, nxt: 011)

011: 02c00-02fff - PID: 01 (idx 011, nxt: 012)

012: 03000-033ff - PID: 01 (idx 012, nxt: 013)

013: 03400-037ff - PID: 01 (idx 013, nxt: -01)

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)

015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)

——————Deallocation——————

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)

015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)

——————Allocation——————

000: 00000-003ff - PID: 01 (idx 000, nxt: 001)

001: 00400-007ff - PID: 01 (idx 001, nxt: -01)

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)

015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)

——————Allocation——————

000: 00000-003ff - PID: 01 (idx 000, nxt: 001)

001: 00400-007ff - PID: 01 (idx 001, nxt: -01)

002: 00800-00bff - PID: 01 (idx 000, nxt: 003)

003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)

004: 01000-013ff - PID: 01 (idx 002, nxt: 005)

005: 01400-017ff - PID: 01 (idx 003, nxt: 006)

006: 01800-01bff - PID: 01 (idx 004, nxt: -01)

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)

015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)

——————Deallocation——————

002: 00800-00bff - PID: 01 (idx 000, nxt: 003)

003: 00c00-00fff - PID: 01 (idx 001, nxt: 004)

004: 01000-013ff - PID: 01 (idx 002, nxt: 005)

005: 01400-017ff - PID: 01 (idx 003, nxt: 006)

006: 01800-01bff - PID: 01 (idx 004, nxt: -01)

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)

015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)

————————Deallocation————————

014: 03800-03bff - PID: 01 (idx 000, nxt: 015)

015: 03c00-03fff - PID: 01 (idx 001, nxt: -01)

————————Deallocation————————

2.3 Implementation

Tìm bảng phân trang từ segment

Trong asignment này, mỗi địa chỉ được biểu diễn bởi 20 bits, trong đó 5 bits đầu tiên là segment, 5 bits tiếp theo là page, và 10 bits cuối cùng là offset.

Chức năng này nhận vào 5 bits segment index và bảng phân đoạn seg_table, cần tìm ra bảng phân trang res của segment tương ứng trong bảng phân đoạn nói trên.

Do bảng phân đoạn seg_table là một danh sách gồm các phần tử u có cấu trúc (v_index, trans_table_t), trong đó v_index là 5 bits segment của phần tử u và trans_table_t là bảng phân trang tương ứng của segment đó. Vì vậy để tìm được res, ta chỉ cần duyệt trên bảng phân đoạn này, phần tử u nào có v_index bằng index cần tìm, ta trả về trans_table tương ứng.

Dưới đây là phần hiện thực cho chức năng trên.

```
static struct trans_table_t * get_trans_table(
        addr_t index, // Segment level index
        const struct seg_table_t * seg_table) { // first level table
    if (seg_table==NULL) return NULL;

    int i;

    for (i = 0; i < seg_table->size; i++) {
        if (seg_table->table[i].v_index==index) return seg_table->table[i].next_lv;
    }

    return NULL;
```

```
}
```

Ánh xạ địa chỉ ảo thành địa chỉ vật lý

Do mỗi địa chỉ gồm 20 bits với cách tổ chức như nói ở trên, do đó để tạo được địa chỉ vật lý, ta lấy 10 bits đầu (segment và page) nối với 10 bits cuối (offset). Mỗi trans_table_t lưu các phần tử có p_index là 10 bits đầu đó. do đó để tạo được địa chỉ vật lý, ta chỉ cần dịch trái 10 bits đó đi 10 bits offset rồi or (|) hai chuỗi này lại.

Dưới đây là phần hiện thực của chức năng trên.

```
    static int translate(
        addr_t virtual_addr,  // Given virtual address
        addr_t * physical_addr, // Physical address to be returned
        const struct pcb_t * proc) { // Process uses given virtual address

    /* Offset of the virtual address */
    addr_t offset = get_offset(virtual_addr);
    /* The first layer index */
    addr_t first_lv = get_first_lv(virtual_addr);
    /* The second layer index */
    addr_t second_lv = get_second_lv(virtual_addr);

    /* Search in the first level */
    struct trans_table_t * trans_table = NULL;
    trans_table = get_trans_table(first_lv, proc->seg_table);
    if (trans_table == NULL) {
        return 0;
    }

    int i;
    for (i = 0; i < trans_table->size; i++) {
        if (trans_table->table[i].v_index == second_lv) {
            *physical_addr = ((offset) | ((trans_table->table[i].p_index)<<OFFSET_LEN));
            return 1;
        }
    }
    return 0;
}
```

Cấp phát memory

Các bước thực hiện

- Kiểm tra xem memory có sẵn sàng cả trên bộ nhớ vật lý và bộ nhớ luận lý hay không.
- Duyệt trên vùng nhớ vật lý, tìm các trang trống, gán trang này được process sử dụng.
- Trên vùng nhớ luận lý, dựa trên địa chỉ cấp phát, tính từ địa chỉ bắt đầu và vị trí thứ tự trang cấp phát, ta tìm được các segment, page của nó. Từ đó cập nhật các bảng phân trang, phân đoạn tương ứng

Dưới đây là phần thực hiện chi tiết:

```
addr_t alloc_mem(uint32_t size, struct pcb_t * proc) {
pthread_mutex_lock(&mem_lock);
addr_t ret_mem = 0;

if ((proc->seg_table!=NULL) && (proc->seg_table->size)>=(1 << FIRST_LV_LEN)){
    proc->seg_table->size=0;
}
/* TODO: Allocate [size] byte in the memory for the
 * process [proc] and save the address of the first
 * byte in the allocated memory region to [ret_mem].
 * */

uint32_t num_pages = (size+PAGE_SIZE-1) / PAGE_SIZE; // Number of pages we will use
int mem_avail = 0; // We could allocate new memory region or not?
/* First check if the amount of free memory in
 * virtual address space and physical address space is
 * large enough to represent the amount of required
 * memory. If so, set 1 to [mem_avail].
 * */
int pages_avail=0;
for (int i=0; i<NUM_PAGES; ++i) pages_avail+=(_mem_stat[i].proc==0);
mem_avail = (pages_avail>=num_pages) && (proc->bp + num_pages*PAGE_SIZE<RAM_SIZE) ? 1 : 0;

if (mem_avail) {
    /* We could allocate new memory region to the process */
    ret_mem = proc->bp;
```

```
addr_t vir_add = ret_mem;
proc->bp += num_pages * PAGE_SIZE;

int pre_page=-1;
int index_cnt=0;
for (int i=0; i<NUM_PAGES && num_pages>0; ++i) if (_mem_stat[i].proc==0){
    _mem_stat[i].proc=proc->pid;
    _mem_stat[i].index=index_cnt++;
    --num_pages;
    if (pre_page!=-1){
        _mem_stat[pre_page].next=i;
    }
    pre_page=i;

    addr_t first_lv = get_first_lv(vir_add);
    addr_t second_lv = get_second_lv(vir_add);

    struct trans_table_t * trans_table = NULL;
    trans_table = get_trans_table(first_lv, proc->seg_table);
    if (trans_table == NULL) {
        if (proc->seg_table==NULL){
            proc->seg_table = (struct seg_table_t *) malloc(sizeof(struct seg_table_t *));
            proc->seg_table->size=0;
        }
        int idx = proc->seg_table->size;
        proc->seg_table->table[idx].v_index=first_lv;

        trans_table = proc->seg_table->table[idx].next_lv
        = (struct trans_table_t *) malloc(sizeof(struct trans_table_t));
        trans_table->size=0;

        proc->seg_table->size++;
    }
    trans_table->size++;
    trans_table->table[trans_table->size-1].v_index=second_lv;
    trans_table->table[trans_table->size-1].p_index=i;
    vir_add+= (PAGE_SIZE);
}
_mem_stat[pre_page].next=-1;
```

```
    }
    pthread_mutex_unlock(&mem_lock);

    return ret_mem;
}
```

Thu hồi memory

Thu hồi địa chỉ vật lý

Chuyển địa chỉ luận lý từ process thành vật lý, sau đó dựa trên giá trị next của mem, ta cập nhật lại chuỗi địa chỉ tương ứng đó

```
    int free_mem(addr_t address, struct pcb_t * proc) {
    pthread_mutex_lock(&mem_lock);

    addr_t phy_add = 0;
    addr_t vir_add = address;

    if (!translate(address, &phy_add, proc)) return 1;

    int num_pages=0;
    addr_t index=(phy_add>>OFFSET_LEN);

    while(1==1){
        ++num_pages;
        _mem_stat[index].proc=0;// clear physical page in memory


        //clear virtual page in process
        addr_t vir_seg = get_first_lv(vir_add);
        addr_t vir_page = get_second_lv(vir_add);
        struct trans_table_t * trans_table = get_trans_table(vir_seg, proc->seg_table);
        if (trans_table==NULL){
            continue;
        }
        int j;
        for (j=0; j<trans_table->size; ++j){
```

```
            if (trans_table->table[j].v_index==vir_page){
                int last = --trans_table->size;
                trans_table->table[j] = trans_table->table[last];
                break;
            }
        }
        if (trans_table->size==0){
            remove_entries(vir_seg, proc->seg_table);
        }

        if (_mem_stat[index].next==-1) break;

        //move to next page
        index=_mem_stat[index].next;
        vir_add+=PAGE_SIZE;
    }
    pthread_mutex_unlock(&mem_lock);

    //update break pointer
    addr_t v_segment_page = address >> OFFSET_LEN;
    if (v_segment_page + num_pages*PAGE_SIZE == proc->bp){
        free_mem_bp(proc);
    }
    return 0;
}
```

Cập nhật địa chỉ luận lý

Dựa trên số trang đã xóa trên block của địa chỉ vật lý, ta tìm lần lượt các trang trên địa chỉ luận lý, dựa trên địa chỉ, ta tìm được segment, page tương ứng. Sau đó cập nhật lại bảng phân trang, sau quá trình cập nhật, nếu bảng trống thì xóa bảng này trong segment đi.

```
    void remove_entries(addr_t vir_seg, struct seg_table_t * seg_table){
    if (seg_table==NULL) return;

    int i;
```

```
    for (i=0; i<seg_table->size; ++i){
        if (seg_table->table[i].v_index == vir_seg){
            int idx = seg_table->size-1;
            seg_table->table[i] =seg_table->table[idx];
            seg_table->table[idx].v_index=0;
            free(seg_table->table[idx].next_lv);
            seg_table->size--;
            return ;
        }
    }
}
```

Cập nhật break point

Chỉ thực hiện khi block cuối cùng trên địa chỉ luận lý được xóa, sau đó từ đó duyệt lần lượt ngược lại các trang, đến khi đến trang đang được sử dụng thì dừng.

```
    void free_mem_bp(struct pcb_t * proc){
    while (proc->bp>=PAGE_SIZE){
        addr_t last_addr = proc->bp - PAGE_SIZE;
        addr_t last_segment = get_first_lv(last_addr);
        addr_t last_page = get_second_lv(last_addr);
        struct trans_table_t * trans_table = get_trans_table(last_segment, proc->seg_table);
        if (trans_table == NULL) return ;
        while (last_page>=0){
            int i;
            for (i=0; i<trans_table->size; ++i){
                if (trans_table->table[i].v_index == last_page){
                    proc->bp-=PAGE_SIZE;
                    last_page--;
                    break;
                }
            }
            if (i==trans_table->size) break;
        }
        if (last_page>=0) break;
    }
}
```

3 Overall

3.1 Question

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

Answer:

Nếu hệ thống không có cơ chế đồng bộ hóa, việc truy xuất và sử dụng dữ liệu sẽ gặp delay, và dữ liệu sẽ không chính xác do race condition.

Ví dụ, một biến dùng chung đang được truy xuất và sử dụng bởi một thread, và giữa lúc thực hiện phép tính, thì một thread khác cũng đồng thời thay đổi nó (mà không có cơ chế bảo vệ nào), sẽ dẫn đến những output sai lệch và không mong muốn. Để ngăn điều này, ta phải có cơ chế bảo vệ các việc truy xuất đến những biến dùng chung bằng những kỹ thuật đồng bộ hóa, tạo và định nghĩa những critical section - những vùng không được truy xuất đồng thời, từ đó đảm bảo được tính đúng đắn của các phép toán.

3.2 Test

make test_ all

```
   ------ MEMORY MANAGEMENT TEST 0 -----------------------------------
./mem input/proc/m0
000: 00000-003ff - PID: 01 (idx 000, nxt: -01)
       003e8: 15
001: 00400-007ff - PID: 01 (idx 000, nxt: 002)
002: 00800-00bff - PID: 01 (idx 001, nxt: 003)
003: 00c00-00fff - PID: 01 (idx 002, nxt: 004)
004: 01000-013ff - PID: 01 (idx 003, nxt: -01)
013: 03400-037ff - PID: 01 (idx 000, nxt: -01)
       03414: 66
```

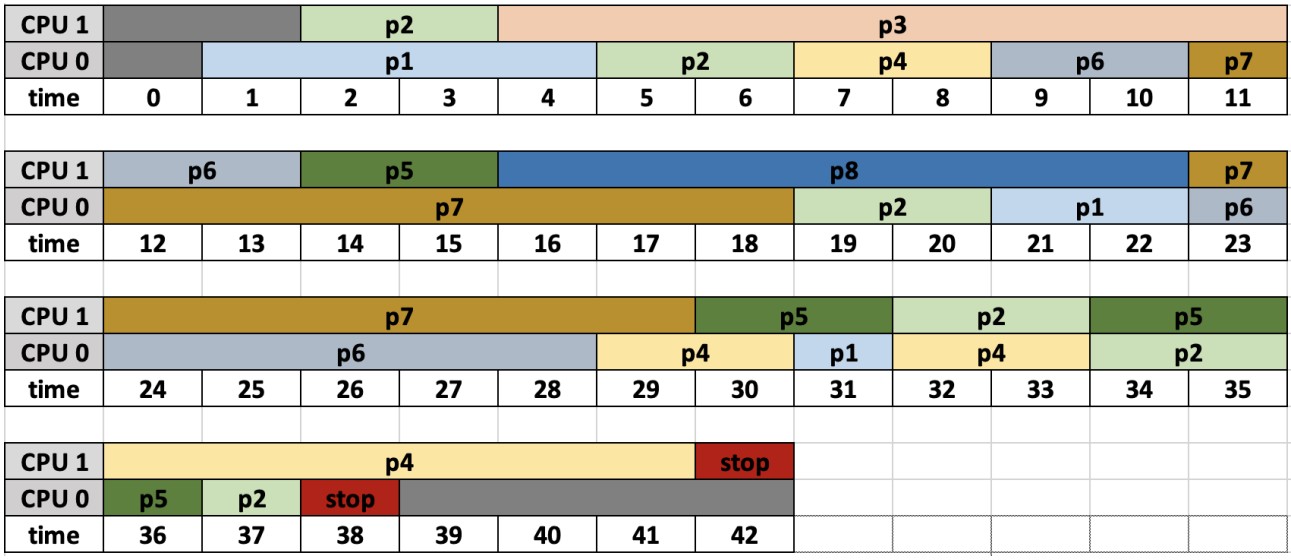
```
   ----- OS TEST 0 -------------------------------------------------
./os os_mlq_2
```

```
Time slot  0
Time slot  1
       Loaded a process at input/proc/s4, PID: 1 PRIO: 4
       CPU 0: Dispatched process 1
Time slot  2
       Loaded a process at input/proc/s3, PID: 2 PRIO: 3
       CPU 1: Dispatched process 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/m1, PID: 3 PRIO: 2
       CPU 1: Put process 2 to run queue
       CPU 1: Dispatched process 3
Time slot  5
       CPU 0: Put process 1 to run queue
       CPU 0: Dispatched process 2
Time slot  6
       Loaded a process at input/proc/s2, PID: 4 PRIO: 3
       CPU 1: Put process 3 to run queue
       CPU 1: Dispatched process 3
Time slot  7
       Loaded a process at input/proc/m0, PID: 5 PRIO: 3
       CPU 0: Put process 2 to run queue
       CPU 0: Dispatched process 4
Time slot  8
       CPU 1: Put process 3 to run queue
       CPU 1: Dispatched process 3
Time slot  9
       Loaded a process at input/proc/p1, PID: 6 PRIO: 2
       CPU 0: Put process 4 to run queue
       CPU 0: Dispatched process 6
Time slot 10
       CPU 1: Put process 3 to run queue
       CPU 1: Dispatched process 3
Time slot 11
       Loaded a process at input/proc/s0, PID: 7 PRIO: 1
       CPU 0: Put process 6 to run queue
       CPU 0: Dispatched process 7
```

```
Time slot 12
        CPU 1: Processed 3 has finished
        CPU 1: Dispatched process 6
Time slot 13
        CPU 0: Put process 7 to run queue
        CPU 0: Dispatched process 7
Time slot 14
        CPU 1: Put process 6 to run queue
        CPU 1: Dispatched process 5
Time slot 15
        CPU 0: Put process 7 to run queue
        CPU 0: Dispatched process 7
Time slot 16
        Loaded a process at input/proc/s1, PID: 8 PRIO: 0
        CPU 1: Put process 5 to run queue
        CPU 1: Dispatched process 8
Time slot 17
        CPU 0: Put process 7 to run queue
        CPU 0: Dispatched process 7
Time slot 18
        CPU 1: Put process 8 to run queue
        CPU 1: Dispatched process 8
Time slot 19
        CPU 0: Put process 7 to run queue
        CPU 0: Dispatched process 2
Time slot 20
        CPU 1: Put process 8 to run queue
        CPU 1: Dispatched process 8
Time slot 21
        CPU 0: Put process 2 to run queue
        CPU 0: Dispatched process 1
Time slot 22
        CPU 1: Put process 8 to run queue
        CPU 1: Dispatched process 8
Time slot 23
        CPU 1: Processed 8 has finished
        CPU 1: Dispatched process 7
        CPU 0: Put process 1 to run queue
        CPU 0: Dispatched process 6
```

```
Time slot 24
Time slot 25
        CPU 1: Put process 7 to run queue
        CPU 1: Dispatched process 7
        CPU 0: Put process 6 to run queue
        CPU 0: Dispatched process 6
Time slot 26
Time slot 27
        CPU 0: Put process 6 to run queue
        CPU 0: Dispatched process 6
        CPU 1: Put process 7 to run queue
        CPU 1: Dispatched process 7
Time slot 28
Time slot 29
        CPU 1: Put process 7 to run queue
        CPU 1: Dispatched process 7
        CPU 0: Processed 6 has finished
        CPU 0: Dispatched process 4
Time slot 30
        CPU 1: Processed 7 has finished
        CPU 1: Dispatched process 5
Time slot 31
        CPU 0: Put process 4 to run queue
        CPU 0: Dispatched process 1
Time slot 32
        CPU 1: Put process 5 to run queue
        CPU 1: Dispatched process 2
        CPU 0: Processed 1 has finished
        CPU 0: Dispatched process 4
Time slot 33
Time slot 34
        CPU 1: Put process 2 to run queue
        CPU 1: Dispatched process 5
        CPU 0: Put process 4 to run queue
        CPU 0: Dispatched process 2
Time slot 35
Time slot 36
        CPU 1: Put process 5 to run queue
        CPU 1: Dispatched process 4
```

```
        CPU 0: Put process 2 to run queue
        CPU 0: Dispatched process 5
Time slot 37
        CPU 0: Processed 5 has finished
        CPU 0: Dispatched process 2
Time slot 38
        CPU 1: Put process 4 to run queue
        CPU 1: Dispatched process 4
        CPU 0: Processed 2 has finished
        CPU 0 stopped
Time slot 39
Time slot 40
        CPU 1: Put process 4 to run queue
        CPU 1: Dispatched process 4
Time slot 41
Time slot 42
        CPU 1: Processed 4 has finished
        CPU 1 stopped
```



Hình 1: Biểu đồ gantt sau khi chạy make test_all

4 References

[1] Các slides và tài liệu thầy up trên BK-el

[2] ABRAHAM SILBERSCHATZ, PETER BAER GALVIN, GREG GAGNE (2018) *Operating System Concepts (10th edition)*, Wiley.

[3] Multilevel Queue (MLQ) CPU Scheduling, *geeksforgeeks.org*, link