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

Course: Operating Systems

Assignment

Simple Operating System

Advisors: Trương Tuấn Phát
Nguyễn Quang Hùng
Group: 4
Students: Trần Đăng Bảo - 2210443
Trần Đức Trí Cường - 2210270
Mã Hoàng Linh - 2211853
Nguyễn Vũ Quang Minh - 2212071
Nguyễn Như Thắng - 2213203

HO CHI MINH CITY, APRIL 2024

BÁO CÁO KẾT QUẢ LÀM VIỆC NHÓM

STT	Họ và tên	MSSV	Nhiệm vụ	Đóng góp
1	Trần Đăng Bảo	2210270	Soạn lý thuyết, làm Sched, TLB, report	35%
2	Trần Đức Trí Cường	2210443	Soạn lý thuyết, làm MMU, report, slide	35%
3	Mã Hoàng Linh	2211853		10%
4	Nguyễn Vũ Quang Minh	2212071		10%
5	Nguyễn Như Thắng	2213203		10%

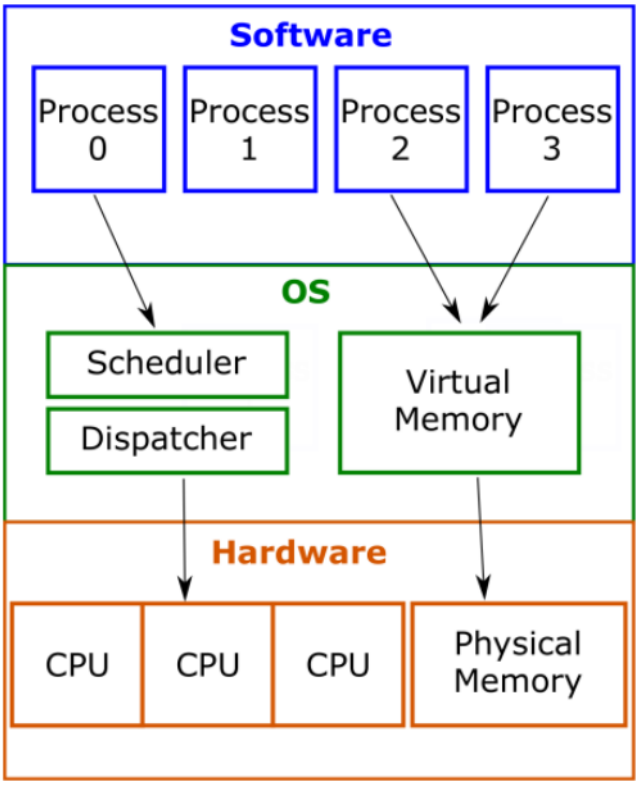
Họ và tên nhóm trưởng: **Trần Đăng Bảo**
Số ĐT: **0961501846**
Email: **bao.trandang@hcmut.edu.vn**
Nhận xét của GV: .....
.....

Giảng viên	**Nhóm trưởng**
(*Ký và ghi rõ họ, tên*)	(*Ký và ghi rõ họ, tên*)
Trương Tuấn Phát	

Mục lục

1 Giới thiệu

Mục tiêu của bài tập lớn này là mô phỏng các thành phần chính của một hệ điều hành đơn giản. Hình sau đây thể hiện kiến trúc tổng thể của hệ điều hành sắp được triển khai. Nói chung, hệ điều hành phải quản lý hai tài nguyên ảo là (các) CPU và RAM bằng hai thành phần cốt lõi:



Hình 1: Tổng quan về các module chính trong bài tập lớn này

- Scheduler (và Dispatcher): xác định tiến trình nào được phép chạy trên CPU nào.
- Virtual memory engine (VME): cô lập không gian bộ nhớ của từng tiến trình với nhau. Bộ nhớ vật lý được chia sẻ bởi nhiều tiến trình nhưng mỗi tiến trình không biết sự tồn tại của tiến trình khác. Điều này được thực hiện bằng cách cho phép mỗi tiến trình có không gian bộ nhớ ảo riêng và công cụ bộ nhớ ảo sẽ ánh xạ và dịch các địa chỉ ảo được cung cấp bởi các tiến trình thành các địa chỉ vật lý tương ứng.

Thông qua các module đó, hệ điều hành cho phép nhiều tiến trình do người dùng tạo ra chia sẻ và sử dụng chung.

2 Scheduler

2.1 Cơ sở lý thuyết

Trong bài tập lớn này, chúng ta sẽ sử dụng giải thuật MLQ (Multi Level Queue) để xác định tiến trình (process) sẽ được thực hiện trong quá trình định thời.

Giải thuật MLQ: là một giải thuật định thời CPU trong đó các tiến trình được phân thành nhiều hàng đợi khác nhau dựa trên mức độ ưu tiên của chúng. Các tiến trình ưu tiên cao được đặt trong các hàng đợi có mức độ ưu tiên cao hơn, trong khi các tiến trình ưu tiên thấp được đặt trong các hàng đợi có mức độ ưu tiên thấp hơn.

Các đặc trưng của giải thuật MLQ:

- Ready queue được chia thành nhiều hàng đợi riêng biệt theo một số tiêu chuẩn như đặc điểm, yêu cầu về định thời của process, foreground và background ...
- Process được gán cố định vào một hàng đợi, mỗi hàng đợi có thể sử dụng giải thuật định thời riêng.
- Hệ điều hành cần phải định thời cho các hàng đợi.
  - Fixed priority scheduling: phục vụ từ hàng đợi có độ ưu tiên cao đến thấp. Vấn đề: có thể có starvation.
  - Time slice: mỗi hàng đợi được nhận một khoảng thời gian chiếm CPU và phân phối cho các process trong hàng đợi khoảng thời gian đó. Ta gọi mỗi đơn vị thời gian là 1 timeslot. Trong bài tập lớn lần này, các hàng đợi sẽ được thực thi bằng giải thuật định thời Round Robin với số lượng phân bổ trong 1 lần chạy bằng max_priority trừ đi priority.

```
prio = 0        |         1          | ....    | MAX_PRIO - 1
slot = MAX_PRIO |   MAX_PRIO - 1     | ....    |      1
```

Hình 2: Time slice của queue

Trong bài tập lớn này, ta sẽ thiết kế định thời cho MLQ theo Time slice.

2.2 Trả lời câu hỏi

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

Answer: Ưu điểm của việc sử dụng hàng đợi ưu tiên so với các thuật toán lập lịch khác là nó cho phép ưu tiên và lựa chọn quy trình một cách hiệu quả dựa trên các mức ưu tiên của chúng. Một hàng đợi ưu tiên là một cấu trúc dữ liệu trong đó các phần tử được gán các mức ưu tiên, và phần tử có mức ưu tiên cao nhất luôn được chọn trước.

- Lập lịch dựa trên ưu tiên: Một hàng đợi ưu tiên cho phép lập lịch dựa trên ưu tiên, trong đó các quy trình có mức ưu tiên cao được thực thi trước. Điều này cho phép triển khai các chính sách dựa trên ưu tiên, như lập lịch can thiệp, nơi một quy trình có mức ưu tiên cao có thể ngắt quy trình đang thực thi có mức ưu tiên thấp hơn.

- Linh hoạt: Hàng đợi ưu tiên có thể dễ dàng cập nhật và điều chỉnh trong quá trình chạy. Khi các mức ưu tiên của các quy trình thay đổi hoặc các quy trình mới được tạo ra, hàng đợi có thể được điều chỉnh tương ứng. Tính linh hoạt này cho phép xử lý hiệu quả các mức ưu tiên thay đổi động trong các hệ thống thời gian thực.
- Lựa chọn hiệu quả: Với một hàng đợi ưu tiên, quy trình có mức ưu tiên cao nhất có thể được chọn trong thời gian hằng số, không phụ thuộc vào số lượng quy trình trong hàng đợi. Điều này đảm bảo lựa chọn hiệu quả cho quy trình tiếp theo được thực thi, giảm độ phức tạp thời gian của thuật toán lập lịch.
- Ưu tiên có thể tùy chỉnh: Hàng đợi ưu tiên cung cấp tính linh hoạt để gán và điều chỉnh mức ưu tiên dựa trên các tiêu chí hoặc chính sách cụ thể. Mức ưu tiên của một quy trình có thể được xác định bởi các yếu tố như quan trọng của quy trình, ràng buộc thời hạn, yêu cầu tài nguyên hoặc bất kỳ chỉ số cụ thể của ứng dụng nào.
- Sự công bằng và đáp ứng: Bằng cách gán các mức ưu tiên khác nhau cho các quy trình, một hàng đợi ưu tiên có thể đảm bảo sự công bằng và đáp ứng trong việc lập lịch. Các quy trình có mức ưu tiên cao nhận được nhiều thời gian CPU hơn, dẫn đến đáp ứng tốt hơn cho các nhiệm vụ quan trọng hoặc thao tác đòi hỏi thời gian cụ thể. Quan trọng nhấn mạnh rằng trong khi hàng đợi ưu tiên cung cấp những ưu điểm về lập lịch dựa trên ưu tiên, chúng có thể không phù hợp cho tất cả các tình huống. Lựa chọn thuật toán lập lịch phụ thuộc vào các yêu cầu cụ thể của hệ điều hành hoặc ứng dụng, xem xét các yếu tố như sự công bằng, công suất, thời gian phản hồi và tải hệ thống.

2.3 Hiện thực

2.3.1 Hiện thực queue.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "queue.h"

int empty(struct queue_t *q)
{
        if (q == NULL)
                return 1;
        return (q->size == 0);
}

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
         * */
        struct pcb_t *proc = NULL;
        if (q->size != 0)
```

```
        {
                proc = q->proc[0];
                q->size--;
                for (int i = 0; i < q->size; ++i)
                {
                        q->proc[i] = q->proc[i + 1];
                }
        }
        return proc;
}
```

2.3.2 Hiện thực sched.c

```

#include "queue.h"
#include "sched.h"
#include <pthread.h>

#include <stdlib.h>
#include <stdio.h>
static struct queue_t ready_queue;
static struct queue_t run_queue;
static pthread_mutex_t queue_lock;

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

int queue_empty(void)
{
#ifdef MLQ_SCHED
  unsigned long prio;
  for (prio = 0; prio < MAX_PRIO; prio++)
    if (!empty(&mlq_ready_queue[prio]))
      return -1;
#endif
  return (empty(&ready_queue) && empty(&run_queue));
}

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

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

#ifdef MLQ_SCHED
/*
 *  Stateful design for routine calling
 *  based on the priority and our MLQ policy
 *  We implement stateful here using transition technique
 *  State representation   prio = 0 .. MAX_PRIO, curr_slot = 0..(MAX_PRIO - prio)
 */
struct pcb_t *get_mlq_proc(void)
```

```
48 {
49   struct pcb_t *proc = NULL;
50   /*TODO: get a process from PRIORITY [ready_queue].
51    * Remember to use lock to protect the queue.
52    */
53   static int curr_prio = 0;
54   static int curr_slot = MAX_PRIO;
55   pthread_mutex_lock(&queue_lock);
56   int turn = MAX_PRIO;
57   while (turn--)
58   {
59     if ((&mlq_ready_queue[curr_prio])->size != 0)
60     {
61       proc = dequeue(&mlq_ready_queue[curr_prio]);
62       curr_slot--;
63       if (curr_slot <= 0)
64       {
65         curr_prio++;
66         if (curr_prio >= MAX_PRIO)
67           curr_prio = 0;
68         curr_slot = MAX_PRIO - curr_prio;
69       }
70       break;
71     }
72     curr_prio++;
73     if (curr_prio >= MAX_PRIO)
74       curr_prio = 0;
75     curr_slot = MAX_PRIO - curr_prio;
76   }
77   pthread_mutex_unlock(&queue_lock);
78   return proc;
79 }
80
81 void put_mlq_proc(struct pcb_t *proc)
82 {
83   pthread_mutex_lock(&queue_lock);
84   enqueue(&mlq_ready_queue[proc->prio], proc);
85   pthread_mutex_unlock(&queue_lock);
86 }
87
88 void add_mlq_proc(struct pcb_t *proc)
89 {
90   pthread_mutex_lock(&queue_lock);
91   enqueue(&mlq_ready_queue[proc->prio], proc);
92   pthread_mutex_unlock(&queue_lock);
93 }
94
95 struct pcb_t *get_proc(void)
96 {
97   return get_mlq_proc();
98 }
99
100 void put_proc(struct pcb_t *proc)
101 {
102   return put_mlq_proc(proc);
103 }
104
105 void add_proc(struct pcb_t *proc)
106 {
107   return add_mlq_proc(proc);
108 }
109 #else
```

```
110 struct pcb_t *get_proc(void)
111 {
112   struct pcb_t *proc = NULL;
113   /* TODO : get a process from [ ready_queue ].
114    * Remember to use lock to protect the queue .
115    * */
116   pthread_mutex_lock(&queue_lock);
117   if (ready_queue->size != 0)
118     proc = dequeue(&ready_queue);
119   pthread_mutex_unlock(&queue_lock);
120   return proc;
121 }
122
123 void put_proc(struct pcb_t *proc)
124 {
125   pthread_mutex_lock(&queue_lock);
126   enqueue(&run_queue, proc);
127   pthread_mutex_unlock(&queue_lock);
128 }
129
130 void add_proc(struct pcb_t *proc)
131 {
132   pthread_mutex_lock(&queue_lock);
133   enqueue(&ready_queue, proc);
134   pthread_mutex_unlock(&queue_lock);
135 }
136 #endif
```

2.4 Kết quả thực thi

2.4.1 Testcase sched

Input:

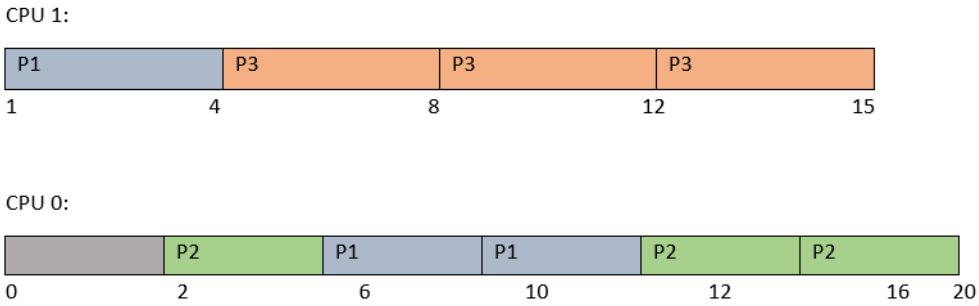
```
1 4 2 3
2 0 p1s 1
3 1 p2s 0
4 2 p3s 1
```

Output:

```
1  Time slot   0
2  ld_routine
3          Loaded a process at input/proc/p1s, PID: 1 PRIO: 1
4  Time slot   1
5          CPU 1: Dispatched process  1
6          Loaded a process at input/proc/p2s, PID: 2 PRIO: 0
7  Time slot   2
8          CPU 0: Dispatched process  2
9          Loaded a process at input/proc/p3s, PID: 3 PRIO: 1
10 Time slot   3
11 Time slot   4
12         CPU 1: Put process  1 to run queue
13         CPU 1: Dispatched process  3
14 Time slot   5
15 Time slot   6
16         CPU 0: Put process  2 to run queue
17         CPU 0: Dispatched process  1
18 Time slot   7
19 Time slot   8
20         CPU 1: Put process  3 to run queue
```

```
        CPU 1: Dispatched process  3
Time slot   9
Time slot  10
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  11
Time slot  12
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot  13
Time slot  14
Time slot  15
        CPU 1: Processed  3 has finished
Time slot  16
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
        CPU 1 stopped
Time slot  17
Time slot  18
Time slot  19
Time slot  20
        CPU 0: Processed  2 has finished
        CPU 0 stopped
```

Biểu đồ Gantt:



Giải thích kết quả:

- Dòng đầu tiên của input: 4 2 3 có nghĩa là mỗi lần dispatch (1 slot) sẽ có độ dài là 4 timeslot, có 2 CPU và có tổng cộng 3 processs. Các process là p1s, p2s, p3s và MAX_PRIO là 5
- Timeslot 0 : process 1 được load vào queue 1 và thời gian load lâu nên tốn 1 timeslot.
- Timeslot 1 : process 2 được load vào queue 0 và CPU 1 đang thực thi process 1.
- Timeslot 2 : process 2 được thực thi thực thi bởi CPU 0 và load process 3 vào queue 1.
- Timeslot 4 : hết 4 timeslot nên process 1 được lấy ra khỏi CPU 1 và trở về queue 1 (hết 1 / 4 lần dispatch của queue 1), tiếp tục lấy process 3 từ queue 1 để thực thi.
- Timeslot 6 : queue 2 hết 4 timeslot, process 2 được đưa về queue 0 (hết 1 / 5 lần dispatch của queue 0), hệ điều hành xuống queue 1 lấy process 1 vào CPU 0 thực thi.
- Timeslot 8 : Sau khi thực thi xong, process 3 trở về queue 1, hệ điều hành tiếp tục thực thi process 3 ở CPU 1.

- Timeslot 10 : process 1 thực thi xong và tiếp tục thực thi tiếp tại CPU 0.
- Timeslot 12 CPU thực xong process 1 và tiếp tục thực thi process 2, CPU 1 thực thi xong process 3 và tiếp tục thự thi process 3.
- Timeslot 15: CPU 1 thực thi xong process.
- Time slot 16: CPU 0 thực thi xong 1 dispatch của process 2 và tiếp tực thực thi, CPU 0 dừng hoạt động.
- Timeslot 20: CPU 1 thực thi xong process 2 và dừng hoạt động luôn sau khi thực thi.

2.4.2 Testcase sched_0

Input:

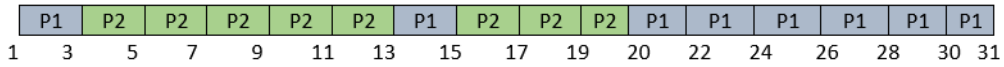
```
2 1 2
0 s0
4 s1
```

Output:

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 4
Time slot   1
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s0, PID: 2 PRIO: 0
Time slot   2
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   4
Time slot   5
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   6
Time slot   7
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   8
Time slot   9
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  10
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  12
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
Time slot  14
Time slot  15
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot  16
Time slot  17
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot  18
```

```
Time slot   19
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
Time slot   20
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  1
Time slot   21
Time slot   22
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   23
Time slot   24
        CPU 0: Put process  1 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  1
Time slot   27
Time slot   28
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   29
Time slot   30
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot   31
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Biểu đồ Gantt:



Giải thích kết quả:

- Dòng đầu tiên của input: 2 1 2 có nghĩa là mỗi lần dispatch (1 slot) sẽ có độ dài là 2 timeslot, có 1 CPU và có tổng cộng 2 processs. Các process là s0, s1 với tổng thời gian thực thi lần lượt là 15, 15. Các process có priority lớn nhất là 4 nên MAX_PRIO là 5.
- Timeslot 0 : process 1 được load vào queue 4 và thời gian load lâu nên tốn 1 timeslot.
- Timeslot 1 : process 2 được load vào queue 0, tuy nhiên CPU đang được sử dụng bởi process 1.
- Timeslot 3 : process 1 thực thi xong và trở về queue 4, hết 1 / 1 lần dispatch của queue 4, hệ điều hành quay lại queue 0 và lấy process 2 vào CPU để thực thi.
- Timeslot 5 : hết 2 timeslot nên process 2 được lấy ra khỏi CPU và trở về queue 0 (hết 1 / 5 lần dispatch của queue 0), tiếp tục lấy process 2 từ queue 0 để thực thi.
- Timeslot 13 : queue 0 hết 5 / 5 lần dispatch, process 2 được đưa về queue 0, hệ điều hành xuống queue 4 lấy process 1 vào CPU thực thi.
- Timeslot 15 : Sau khi thực thi xong, process 1 trở về queue 4, hệ điều hành quay về queue 0 và tiếp tục dispatch nhiều lần liên tiếp ở đây.
- Timeslot 20 : process 2 đã thực hiện xong, queue 0 trống nên hệ điều hành xuống queue 4 lấy process 1 và đưa nó thực thi đến khi xong vào timeslot 31.

2.4.3 Testcase sched_1

Input:

```
2 1 4
0 s0
4 s1
6 s2
7 s3
```

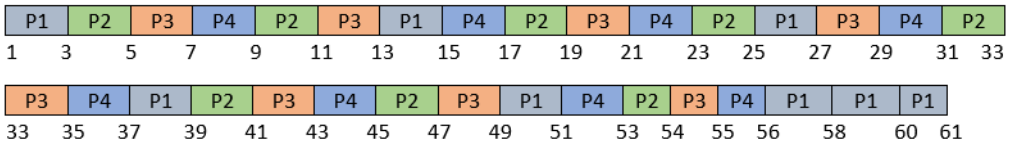
Output:

```
Time slot   0
ld_routine
        Loaded a process at input/proc/s0, PID: 1 PRIO: 4
Time slot   1
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/s0, PID: 2 PRIO: 0
Time slot   2
        Loaded a process at input/proc/s0, PID: 3 PRIO: 0
Time slot   3
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
        Loaded a process at input/proc/s0, PID: 4 PRIO: 0
Time slot   4
Time slot   5
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot   6
Time slot   7
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot   8
Time slot   9
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  10
Time slot  11
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  12
Time slot  13
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  14
Time slot  15
        CPU 0: Put process  1 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  2
Time slot  18
Time slot  19
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  20
Time slot  21
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  22
Time slot  23
        CPU 0: Put process  4 to run queue
```

```
        CPU 0: Dispatched process  2
Time slot  24
Time slot  25
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
Time slot  26
Time slot  27
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  3
Time slot  28
Time slot  29
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  30
Time slot  31
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  32
Time slot  33
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  34
Time slot  35
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  36
Time slot  37
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  1
Time slot  38
Time slot  39
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot  40
Time slot  41
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  42
Time slot  43
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  4
Time slot  44
Time slot  45
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  46
Time slot  47
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  3
Time slot  48
Time slot  49
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  1
Time slot  50
Time slot  51
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
Time slot  52
Time slot  53
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  2
Time slot  54
```

```
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  3
Time slot  55
        CPU 0: Processed  3 has finished
        CPU 0: Dispatched process  4
Time slot  56
        CPU 0: Processed  4 has finished
        CPU 0: Dispatched process  1
Time slot  57
Time slot  58
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  59
Time slot  60
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  1
Time slot  61
        CPU 0: Processed  1 has finished
        CPU 0 stopped
```

Biểu đồ Gantt:



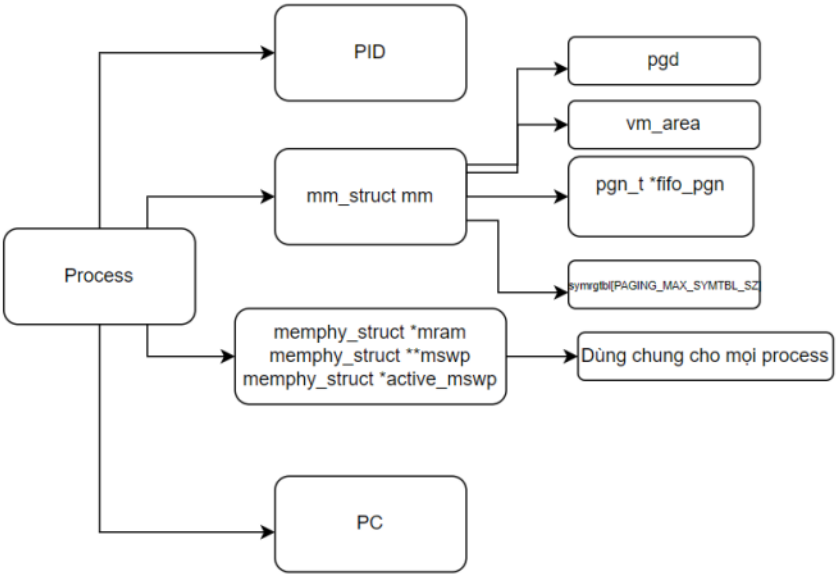
Giải thích kết quả:

- Dòng đầu tiên của input: 2 1 4, nghĩa là mỗi timeslice sẽ là 2 timeslot, có 1 CPU là CPU0 và có tổng cộng 4 process. Các process đều được định nghĩa s0 nên thời gian thực thi của mỗi process là 15s. Các process có độ ưu tiên thấp nhất là 4 nên MAX_PRIO sẽ là 5.
- Timeslot 0: process 1 được load vào queue 4, thời gian load khá lâu nên chưa thể thực thi được.
- Timeslot 1 : process 2 được load vào queue 0, tuy nhiên CPU đang được sử dụng bởi process 1
- Timeslot 2: Process 3 được đưa vào queue 0.
- Timeslot 3: process 1 đã thực hiện xong 1 lần dispatch nên sẽ được lấy ra khỏi CPU và đưa về queue 4 (hết 1/1 lần dispatch của queue 4). Queue 0 đang có process 2, process này được load vào CPU và thực thi, sau đó process 4 được đưa vào queue 0 đang trống.
- Timeslot 5, process 2 ra khỏi CPU, đưa về queue 0 sau process 4 và hệ điều hành tiếp tục lấy process từ queue 0 (lần 2 / 5 lần dispatch), lúc này process 3 được đưa vào CPU và thực thi.
- Timeslot 7: process 3 ra khỏi CPU, đưa về queue 0 sau process 2 và hệ điều hành tiếp tục lấy process từ queue 0 (lần 3 / 5 lần dispatch), lúc này process 4 được đưa vào CPU và thực thi.
- Timeslot 9, process 4 ra khỏi CPU, đưa về queue 0 sau process 3 và hệ điều hành tiếp tục lấy process từ queue 0 (lần 4 / 5 lần dispatch), lúc này process 2 được đưa vào CPU và thực thi.

- Timeslot 11, process 2 ra khỏi CPU, đưa về queue 0 sau process 4 và hệ điều hành tiếp tục lấy process từ queue 0 (lần 5 / 5 lần dispatch), lúc này process 3 được đưa vào CPU và thực thi.
- Timeslot 13, sau khi đủ 5 lần dispatch của queue 0, tiếp tục tìm các process ở các queue priority thấp hơn để thực hiện, hệ điều hành xuống queue 4 và đưa process 1 vào CPU.
- Timeslot 15, sau khi process 1 thực hiện xong (lần 1 / 1 lần dispatch) queue 4, hệ điều hành sẽ lại quay về dispatch 5 lần trên queue 0 và cứ tiếp tục như thế đến khi các process hoàn thành.

3 Memory Management

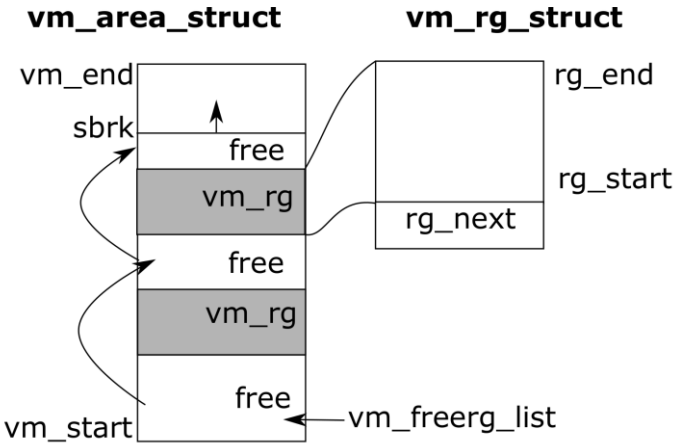
Những thành phần chính của process:



Hình 2: Một số thành phần chính của process

3.1 Cơ sở lý thuyết

Trong Bài tập lớn này, kỹ thuật Paging được sử dụng để quản lý bộ nhớ. Máy tính sử dụng kỹ thuật này để lưu trữ và truy xuất dữ liệu từ bộ nhớ thứ cấp để đưa vào bộ nhớ chính để có thể sử dụng. Dưới đây là các mô tả về các thành phần được sử dụng cho kỹ thuật Paging:



Cấu trúc của vm area và region

3.1.1 Virtual memory area

Khi chương trình chạy, mỗi process sẽ được phân bố một phần không gian bộ nhớ riêng gọi là "Memory Area". Khu vực này chứa những phần nhỏ liên tục hoặc không liên tục, được gọi là "Memory Region" hay "Page". Mỗi process có một page address (hoặc logical address) lưu trong PCB, bao gồm số trang (pgn) là chỉ số của page table lưu base address của mỗi phần bộ nhớ của process và "Page offset" kết hợp với base address để xác định physical address trong bộ nhớ vật lý.

Vm_area: Mỗi khu vực bộ nhớ kéo dài liên tục trong khoảng [vm start, vm end]. Mặc dù không gian trải dài trên toàn bộ khoảng này, nhưng khu vực có thể sử dụng thực tế được giới hạn bởi đỉnh chỉ vào sbrk. Trong khu vực giữa vm start và sbrk, có nhiều vùng được ghi nhận bởi cấu trúc vm rg struct và các khoảng trống được theo dõi bởi danh sách vm freerg list.

vm_rg: Những khu vực này thực sự được xem như là các biến trong mã nguồn của chương trình có thể đọc được bởi con người. Do thực tế hiện tại nằm ngoài phạm vi đề cập, chúng ta chỉ đơn giản là khái quát khái niệm không gian tên trong phạm vi chỉ mục. Chúng ta tạm thời tưởng tượng những khu vực này như một tập hợp các khu vực có số lượng hạn chế. Chúng ta quản lý chúng bằng cách sử dụng một mảng symrgtbl[PAGING MAX SYMTBL SZ]. Kích thước của mảng được cố định bởi một hằng số, PAGING MAX SYMTBL SZ, chỉ định số lượng biến được phép trong mỗi chương trình. Memory mapping: Các khu vực bộ nhớ trong một khu vực bộ nhớ liền kề riêng biệt. Trong mỗi cấu trúc ánh xạ bộ nhớ, nhiều khu vực bộ nhớ được chỉ ra bởi struct vm area struct *mmap list. Trường quan trọng tiếp theo là pgd, đây là thư mục bảng trang, chứa tất cả các mục nhập bảng trang. Mỗi mục nhập là một ánh xạ giữa số trang và số khung trong hệ thống quản lý bộ nhớ trang. Symrgtbl là một triển khai đơn giản của bảng ký hiệu. Các trường khác chủ yếu được sử dụng để theo dõi một hoạt động cụ thể của người dùng ví dụ như người gọi, trang fifo (để tham khảo).

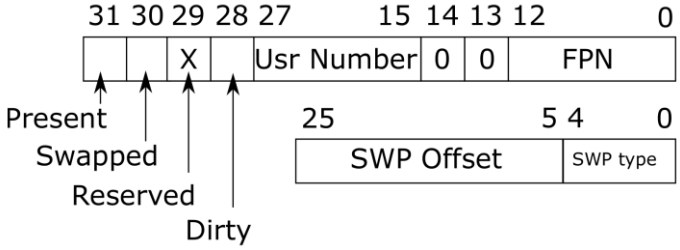
3.1.2 Physical memory

Tất cả các quá trình đều có các bản đồ bộ nhớ riêng biệt của chúng, nhưng tất cả các bản đồ này đều nhắm đến một thiết bị vật lý đơn lẻ. Có hai loại thiết bị là RAM và SWAP. Cả hai có thể được triển khai bởi cùng một thiết bị vật lý trong mm-memphy.c với các cài đặt khác nhau. Các cài đặt được hỗ trợ bao gồm truy cập bộ nhớ ngẫu nhiên, truy cập bộ nhớ tuần tự/serial và dung lượng lưu trữ.

Mặc dù có nhiều cấu hình có thể, việc sử dụng logic của các thiết bị này có thể được phân biệt. Thiết bị RAM, thuộc về hệ thống bộ nhớ chính, có thể được truy cập trực tiếp từ bus địa chỉ CPU, tức là, có thể đọc/ghi bằng các lệnh CPU. Trong khi đó, SWAP chỉ là một thiết bị bộ nhớ thứ cấp, và tất cả các thao tác dữ liệu lưu trữ của nó phải được thực hiện bằng cách di chuyển chúng đến bộ nhớ chính. Do thiếu truy cập trực tiếp từ CPU, hệ thống thường trang bị một SWAP lớn với chi phí nhỏ và thậm chí có nhiều hơn một phiên bản. Trong cài đặt của nhóm, phần cứng được cài đặt với một thiết bị RAM và tối đa 4 thiết bị SWAP.

3.1.3 Paging-based address translation scheme

Mỗi virtual page của process đều có một page table giúp process truy xuất "frame" vật lý tương ứng. Bảng này chứa các mục (Page Table Entries - PTE), mỗi PTE có giá trị 32-bit, được định nghĩa về dữ liệu và cấu trúc như sau:



Nội dung của page table entry

- Bits 0-4: Swap type if swapped
- Bits 5-25: Swap offset if swapped
- Bits 0-12: Page frame number (PFN) if present
- Bits 13-14: Zero if present
- Bits 15-27: User-defined numbering if present
- Bit 28: Dirty
- Bit 29: Reserved
- Bit 30: Swapped
- Bit 31: Present

Memory swapping: Khi dữ liệu trong RAM đã đầy, nếu có một tiến trình yêu cầu alloc thì việc này không thể thực hiện được. Tuy nhiên với virtual memory và sự giúp đỡ của Memory swapping thì việc này có thể. Việc hoán đổi có thể giúp di chuyển nội dung của khung vật lý giữa MEMRAM và MEMSWAP. Hoán đổi là cơ chế thực hiện sao chép nội dung của khung từ bên ngoài vào bộ nhớ chính RAM. Ngược lại, hoán đổi ra cố gắng di chuyển nội dung của khung trong MEMRAM sang MEMSWAP. Trong bối cảnh điển hình, việc hoán đổi giúp chúng ta thu được khung RAM trống vì kích thước của thiết bị SWAP thường đủ lớn. Tuy nhiên, trên thực tế, quá trình này chiếm thời gian lớn hơn rất nhiều so với đọc từ RAM nên với những RAM nhỏ thì việc swap thường xuyên khiến cho việc chạy chương trình trở nên rất chậm. Basic memory operations in paging-based system:

- ALLOC: trong hầu hết trường hợp, cung cấp một Alloc tương ứng với khu vực có sẵn. Nếu không có không gian thích hợp, chúng ta cần nâng giới hạn sbrk lên và vì nó chưa từng được sử dụng, có thể cần cung cấp một số khung vật lý và sau đó ánh xạ chúng bằng Bảng mục trang
- FREE: không gian lưu trữ liên quan đến id khu vực. Vì chúng ta không thể thu hồi lại khung vật lý đã lấy, điều này có thể gây ra lỗ hổng bộ nhớ, chúng ta chỉ giữ không gian lưu trữ đã thu thập trong danh sách miễn phí để yêu cầu cấp phát tiếp theo.
- READ/WRITE: yêu cầu trang phải được hiển thị trong bộ nhớ chính. Bước tiêu tốn nhiều tài nguyên nhất là hoán đổi trang. Nếu trang đó ở trong thiết bị MEMSWAP, cần phải đưa trang đó trở lại thiết bị MEMRAM (hoán đổi vào) và nếu thiếu không gian, chúng ta cần trả lại một số trang cho thiết bị MEMSWAP (hoán đổi ra) để tạo thêm chỗ.

3.2 Trả lời câu hỏi

Question: 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?

Answer: Việc sử dụng multiple memory segments trong khai báo mã nguồn của hệ điều hành đơn giản mang lại một số lợi ích.

- Tổ chức Bộ nhớ: Bằng cách chia bộ nhớ thành nhiều segments, thiết kế cung cấp một bố cục cấu trúc và có tổ chức cho các loại dữ liệu và mã khác nhau. Mỗi segment có thể được dành riêng cho một mục đích cụ thể, như the text segment để lưu trữ chỉ thị chương trình, data segment để lưu trữ dữ liệu được khởi tạo và đoạn heap để cấp phát bộ nhớ động. Tổ chức này cải thiện việc quản lý và truy cập chung của các loại bộ nhớ khác nhau.
- Bảo vệ Bộ nhớ: Việc sử dụng multiple segments cho phép bảo vệ và cô lập bộ nhớ. Mỗi segment có thể có các quyền truy cập riêng, như chỉ đọc, đọc-viết hoặc chỉ thực thi. Điều này đảm bảo rằng các process hoặc segment không thể không cố ý sửa đổi hoặc truy cập vào các khu vực bộ nhớ mà chúng không được phép, nâng cao tính bảo mật và ổn định của hệ thống.
- Tính Mô-đun và Linh hoạt: Thiết kế của multiple memory segment cho phép tính mô-đun và linh hoạt trong việc quản lý bộ nhớ. Nó cung cấp một cơ chế để cấp phát và giải phóng bộ nhớ một cách linh hoạt, cho phép sử dụng bộ nhớ hiệu quả và tránh sự phân mảnh. Nó cũng cho phép thay đổi kích thước động của các segment cụ thể dựa trên nhu cầu của các tiến trình hoặc ứng dụng.
- Virtual Memory Mapping: Hệ thống điều hành đơn giản có maps virtual addressses chuyển đổi các địa chỉ ảo do các process cung cấp thành các địa chỉ vật lý tương ứng. Thiết kế của nhiều memory segment hỗ trợ quá trình mapping này bằng cách cung cấp ranh giới và mapping giữa bộ nhớ ảo và vật lý. Điều này cho phép dịch địa chỉ và quản lý bộ nhớ hiệu quả.
- Đọc hiểu Mã và Dễ bảo trì: Bằng cách khai báo rõ ràng nhiều memory segment trong mã nguồn, thiết kế cải thiện độ dễ đọc và bảo trì của mã. Nó cung cấp một hiểu biết rõ ràng về bố cục và việc sử dụng memory, làm cho việc gỡ lỗi và sửa đổi mã trong tương lai trở nên dễ dàng hơn.

Tổng thể, thiết kế của multiple memory segment trong hệ điều hành đơn giản cải thiện tổ chức, bảo vệ, tính mô-đun và tính bảo trì của bộ nhớ, góp phần vào hiệu suất và độ tin cậy tổng thể của hệ điều hành.

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

Answer: Nếu chúng ta chia địa chỉ thành nhiều hơn 2 cấp độ trong hệ thống quản lý bộ nhớ phân trang, bộ nhớ RAM và thiết bị SWAP, việc chia địa chỉ thành nhiều cấp độ có thể có một số yếu tố cần xem xét bổ sung: Nếu chúng ta chia địa chỉ thành nhiều hơn 2 cấp độ trong hệ thống quản lý bộ nhớ phân trang, nó cho phép biểu diễn một không gian địa chỉ lớn hơn. Mỗi cấp độ phân trang bổ sung cung cấp nhiều bit hơn cho việc địa chỉ hóa, cho phép một số lượng lớn hơn các virtual and physic pages.

Việc chia địa chỉ thành nhiều cấp độ tăng tính linh hoạt và khả năng mở rộng của hệ thống quản lý bộ nhớ. Nó cho phép ánh xạ chi tiết hơn của các trang ảo sang các trang vật lý, có thể cải thiện việc sử dụng bộ nhớ và giảm phân mảnh. Nó cũng cho phép hệ thống xử lý lượng bộ nhớ lớn một cách hiệu quả.

Tuy nhiên, việc tăng số cấp độ trong hệ thống phân trang cũng đưa ra thêm sự phức tạp và chi phí tăng thêm. Mỗi cấp độ đòi hỏi bộ nhớ bổ sung để lưu trữ các bảng trang hoặc mục thư mục trang, và nó thêm các cấp độ trung gian trong quá trình dịch địa chỉ, điều này có thể ảnh hưởng đến hiệu suất.

Tổng thể, việc chia địa chỉ thành nhiều hơn 2 cấp độ trong hệ thống quản lý bộ nhớ phân trang có thể mang lại lợi ích về khả năng địa chỉ và việc sử dụng bộ nhớ, nhưng nó cũng đi kèm với sự phức tạp và những đối thoại về hiệu suất tiềm ẩn. Quyết định sử dụng nhiều cấp độ trong hệ thống phân trang nên xem xét các yêu cầu cụ thể và ràng buộc của hệ điều hành và kiến trúc phần cứng.

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

Answer: Segmentation with paging kết hợp các lợi ích của cả hai kỹ thuật quản lý bộ nhớ segmentation và paging. Dưới đây là các ưu điểm và nhược điểm của Segmentation with paging:

- **Advantage:**
  - Linh hoạt trong Quản lý Bộ nhớ: Segmentation cho phép phân chia không gian địa chỉ thành các segments có kích thước biến đổi, có thể đại diện cho các phần khác nhau của một chương trình, như code, dữ liệu, stack, v.v. Điều này cung cấp một lược đồ cấp phát bộ nhớ linh hoạt hơn so với một mô hình bộ nhớ phẳng.
  - Bảo vệ và Chia sẻ: Segmentation cho phép mỗi phân đoạn có quyền truy cập riêng.
  - Mở rộng Không gian Địa chỉ: Paging cho phép không gian địa chỉ lớn hơn so với bộ nhớ vật lý có sẵn. Nó cho phép cấp phát bộ nhớ theo các đơn vị cố định nhỏ gọi là pages. Điều này giúp sử dụng bộ nhớ vật lý một cách hiệu quả và hỗ trợ không gian địa chỉ lớn hơn.
  - Virtual Memory: Segmentation with paging cho phép hệ thống triển khai bộ nhớ ảo, cho phép các tiến trình sử dụng nhiều bộ nhớ hơn so với sự có sẵn vật lý. Nó cung cấp ảo hóa của một không gian địa chỉ lớn hơn cho các tiến trình bằng cách thực hiện thông minh việc hoán đổi các trang vào và ra khỏi bộ nhớ vật lý.
- **Disadvantage:**
  - Tăng sự Phức tạp: Segmentation with paging đưa ra sự phức tạp bổ sung cho hệ thống quản lý bộ nhớ. Nó yêu cầu quản lý cả segment tables và page tables, tăng thêm chi phí cho việc dịch địa chỉ và truy cập bộ nhớ.

- Tăng Overhead: Việc sử dụng cả segmentation và paging đưa ra sự tăng chi phí overhead. Mỗi segmentation và paging đều đòi hỏi siêu dữ liệu bổ sung dưới dạng segment tables and page tables, tiêu tốn tài nguyên bộ nhớ.
- Phân mảnh: Kết hợp segmentation và paging có thể dẫn đến phân mảnh bộ nhớ. Phân mảnh ngoại vi có thể xảy ra do các segments có kích thước biến đổi, và phân mảnh nội vi có thể xảy ra do các pages có kích thước cố định.
- Ảnh Hưởng đến hiệu suất: Các cấp độ trung gian bổ sung trong quá trình dịch địa chỉ có thể xảy ra overhead và ảnh hưởng đến hiệu suất hệ thống. Truy cập bộ nhớ thông qua nhiều cấp độ bảng đòi hỏi thêm thời gian chờ đợi cho các truy cập bộ nhớ.

Tổng thể, Segmentation with paging cung cấp linh hoạt, bảo vệ, chia sẻ và hỗ trợ cho bộ nhớ ảo. Tuy nhiên, nó cũng đưa ra sự phức tạp, phân mảnh, overhead và những vấn đề về hiệu suất tiềm ẩn. Quyết định sử dụng segmentation with paging nên xem xét các yêu cầu cụ thể và những trade-off của hệ thống và kiến trúc phần cứng.

3.3 Hiện thực

3.3.1 Hiện thực mm-memphy.c

Hàm MEMPHY_dump

Chức năng: dump nội dung từ mp->storage ra màn hình. Ở đây, vì kích cỡ bộ nhớ lớn, nên ta sẽ chỉ in ra những địa chỉ có nội dung không phải 0 (sẽ có trường hợp giá trị được ghi vào bộ nhớ là 0, khi đó không thể in ra được giá trị; để tránh trường hợp đó, các giá trị được ghi vào bộ nhớ sẽ khác 0 để dễ theo dõi). Ở bài tập lớn này ta chỉ quan tâm đến trạng thái của RAM nên hàm MEMPHY_dump() sẽ được điều chỉnh sao cho phù hợp với yêu cầu đề bài:

```
int MEMPHY_dump(struct memphy_struct *mp) {
  /*TODO dump memphy content mp->storage
   *     for tracing the memory content
   */
  for(int i=0;i<mp->maxsz;i++){
    BYTE data;

    MEMPHY_read(mp, i, &data);
    if(data!=0)
    printf("Addr 0x%x: %d\n", i, (uint32_t)data);
  }
  #ifdef MEM_TO_FILE
  FILE *ptr;
  char file[100] = "\0";
  char *num[10];
  sprintf(num, "%d", numFile);
  strcat(file, "output");
  strcat(file, num);
  strcat(file, ".txt");
  ptr = fopen(file, "w");
  fprintf(ptr, "Memphy content:\n");
  for (int i = 0; i < mp->maxsz; i++) {
    BYTE data;
    MEMPHY_read(mp, i, &data);
    fprintf(ptr, "Addr 0x%x: %d\n", i, (uint32_t)data); // %x is hex
  }
  numFile++;
  fclose(ptr);
  #endif
  return 0;
```

```
}
```

3.3.2 Hiện thực mm-vm.c

Hàm __alloc Hàm __alloc có nhiệm vụ cấp phát vùng nhớ cho một tiến trình trong hệ thống dựa trên quản lý bộ nhớ phân trang. Nó kiểm tra và cấp phát một vùng nhớ mới cho tiến trình, hoặc thực hiện các biện pháp khác nếu không có đủ không gian.

```
int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *alloc_addr)
    {
  /*Allocate at the toproof */
  struct vm_rg_struct rgnode;

  // Check if there is free room in list vm_freerg_list of vma
  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;
    struct framephy_struct *frame = NULL;
    int n_page = (PAGING_PAGE_ALIGNSZ(rgnode.rg_end) - PAGING_PAGE_ALIGNSZ( rgnode
    .rg_start))/PAGING_PAGESZ;
    if(alloc_pages_range(caller,n_page, &frame) < 0){
      return -1;
    }
    vmap_page_range(caller,caller->mm->symrgtbl[rgid].rg_start, n_page,frame,&
    caller->mm->symrgtbl[rgid]);

    *alloc_addr = rgnode.rg_start;
      #ifdef EX
  printf("\n\tRun ALLOC %d: Process %2d\n", size,  caller->pid);
  printf("\tRange ID: %d, Start: %lu, End: %lu\n", rgid, rgnode.rg_start, rgnode.
    rg_end);
    print_rg_memphy(caller, rgnode);
  printf("\tUsed Region List: \n");
  for(int rgit = 0 ; rgit < PAGING_MAX_SYMTBL_SZ; rgit++){
      if(caller->mm->symrgtbl[rgit].rg_start == 0 && caller->mm->symrgtbl[rgit].
    rg_end == 0){
        continue;
      }
      printf("\trg[%ld->%ld]\n", caller->mm->symrgtbl[rgit].rg_start, caller->mm->
    symrgtbl[rgit].rg_end);
    }
  printf("\tFree Region List:\n");
  print_list_rg(caller->mm->mmap->vm_freerg_list);
  #endif
    return 0;
  }

  /* TODO get_free_vmrg_area FAILED handle the region management (Fig.6)*/

  /*Attempt to increate limit to get space */
  if( caller->mm->mmap->sbrk + size < caller->mm->mmap->vm_end){
    caller->mm->symrgtbl[rgid].rg_start = caller->mm->mmap->sbrk;
    caller->mm->symrgtbl[rgid].rg_end = caller->mm->mmap->sbrk + size;
    caller->mm->mmap->sbrk += size;
    *alloc_addr = caller->mm->symrgtbl[rgid].rg_start;
      #ifdef EX
  printf("\n\tRun ALLOC %d: Process %2d\n", size,  caller->pid);
  printf("\tRange ID: %d, Start: %lu, End: %lu\n", rgid, caller->mm->symrgtbl[rgid
    ].rg_start, caller->mm->symrgtbl[rgid].rg_end);
    print_rg_memphy(caller, caller->mm->symrgtbl[rgid]);
  printf("\tUsed Region List: \n");
```

```
47   for(int rgit = 0 ; rgit < PAGING_MAX_SYMTBL_SZ; rgit++){
48       if(caller->mm->symrgtbl[rgit].rg_start == 0 && caller->mm->symrgtbl[rgit].
       rg_end == 0){
49           continue;
50       }
51       printf("\trg[%ld->%ld]\n", caller->mm->symrgtbl[rgit].rg_start, caller->mm->
       symrgtbl[rgit].rg_end);
52     }
53   printf("\tFree Region List:\n");
54   print_list_rg(caller->mm->mmap->vm_freerg_list);
55   #endif
56     return 0;
57   }
58
59
60   struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
61   int inc_sz = PAGING_PAGE_ALIGNSZ(size);
62   // int inc_limit_ret
63   int old_sbrk;
64
65   old_sbrk = cur_vma->sbrk;
66
67   /* TODO INCREASE THE LIMIT
68    * inc_vma_limit(caller, vmaid, inc_sz)
69    */
70   if (inc_vma_limit(caller, vmaid, inc_sz) < 0) {
71     printf("Failed to increase virtual memory area limit!\n");
72     return -1;
73   }
74
75   /*Successful increase limit */
76   caller->mm->symrgtbl[rgid].rg_start = old_sbrk;
77   caller->mm->symrgtbl[rgid].rg_end = old_sbrk + size;
78
79   *alloc_addr = old_sbrk;
80
81   cur_vma->sbrk += inc_sz;
82     #ifdef EX
83   printf("\n\tRun ALLOC %d: Process %2d\n", size,  caller->pid);
84   printf("\tRange ID: %d, Start: %lu, End: %lu\n", rgid, caller->mm->symrgtbl[rgid
       ].rg_start, caller->mm->symrgtbl[rgid].rg_end);
85   print_rg_memphy(caller, caller->mm->symrgtbl[rgid]);
86   printf("\tUsed Region List: \n");
87   for(int rgit = 0 ; rgit < PAGING_MAX_SYMTBL_SZ; rgit++){
88       if(caller->mm->symrgtbl[rgit].rg_start == 0 && caller->mm->symrgtbl[rgit].
       rg_end == 0){
89           continue;
90       }
91       printf("\trg[%ld->%ld]\n", caller->mm->symrgtbl[rgit].rg_start, caller->mm->
       symrgtbl[rgit].rg_end);
92     }
93   printf("\tFree Region List:\n");
94   print_list_rg(caller->mm->mmap->vm_freerg_list);
95   #endif
96   return 0;
97 }
```

Hàm enlist_vm_freerg_list

Để việc thêm bộ nhớ mới hoạt động tốt hơn thì cần sửa lại hàm enlist_vm_freerg_list như sau để có thể xử lý được hết tất cả trường hợp có thể xảy ra:

```
1 int enlist_vm_freerg_list(struct mm_struct *mm, struct vm_rg_struct rg_elmt) {
```

```
  pthread_mutex_lock(&vm_lock);
  struct vm_rg_struct *rg_node = mm->mmap->vm_freerg_list;

  if (rg_elmt.rg_start >= rg_elmt.rg_end){
    pthread_mutex_unlock(&vm_lock);
    return -1;
  }

  struct vm_rg_struct *new_rg = (struct vm_rg_struct *)malloc(sizeof(struct
    vm_rg_struct));
  new_rg->rg_start = rg_elmt.rg_start;
  new_rg->rg_end = rg_elmt.rg_end;
  new_rg->rg_next = NULL;

  if (rg_node != NULL)
    new_rg->rg_next = rg_node;

  /* Enlist the new region */

  mm->mmap->vm_freerg_list = new_rg;
  pthread_mutex_unlock(&vm_lock);
  return 0;
}
```

Hàm __free Hàm __free được thiết kế để giải phóng một vùng nhớ đã được cấp phát trước đó cho một quá trình (process). Hàm này nhận vào thông tin về quá trình gọi (caller), mã quản lý bộ nhớ ảo (vmaid), và số nhận dạng vùng nhớ cần giải phóng (rgid). Sau đó, hàm thực hiện các bước như kiểm tra tính hợp lệ của rgid, lấy thông tin về vùng nhớ cần giải phóng, tạo một cấu trúc mới để đại diện cho vùng nhớ đã giải phóng và thêm nó vào danh sách các vùng nhớ không sử dụng thông qua hàm enlist_vm_freerg_list

```
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;
  int i;
  int page_start = PAGING_PGN(caller->mm->symrgtbl[rgid].rg_start);
  int page_end = PAGING_PGN(caller->mm->symrgtbl[rgid].rg_end);
  for ( i = page_start; i <= page_end; i++)
  {
    /* code */
    caller->mm->pgd[i] = 0;
  }

  /* 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;

  caller->mm->symrgtbl[rgid].rg_start = 0;
  caller->mm->symrgtbl[rgid].rg_end = 0;

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

  #ifdef EX
  printf("\n\tRun FREE: Process %2d\n", caller->pid);
  printf("\tRange ID: %d, Start: %lu, End: %lu\n", rgid, rgnode.rg_start, rgnode.
    rg_end);
  print_rg_memphy(caller, rgnode);
  printf("\tUsed Region List: \n");
```

```
  for(int rgit = 0 ; rgit < PAGING_MAX_SYMTBL_SZ; rgit++){
      if(caller->mm->symrgtbl[rgit].rg_start == 0 && caller->mm->symrgtbl[rgit].
    rg_end == 0){
        continue;
      }
      printf("\trg[%ld->%ld]\n", caller->mm->symrgtbl[rgit].rg_start, caller->mm->
    symrgtbl[rgit].rg_end);
    }
  printf("\tFree Region List:\n");
  print_list_rg(caller->mm->mmap->vm_freerg_list);
  #endif

  return 0;
}
```

Hàm pg_getpage

Hàm pg_getpage thực hiện việc quản lý trang (paging) trong hệ thống bộ nhớ ảo. Hàm này nhận vào một cấu trúc quản lý bộ nhớ (mm_struct), số thứ tự trang (pgn), con trỏ đến số khung trang (fpn), và con trỏ đến quá trình gọi (caller). Hàm kiểm tra trạng thái của trang và thực hiện các bước như swap trang từ bộ nhớ chính (MEMRAM) sang bộ nhớ trang đổi (MEMSWP) nếu cần thiết, sau đó cập nhật bảng trang và danh sách theo dõi trang

```
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);

    vicpte = mm->pgd[vicpgn];

    vicfpn = PAGING_FPN(vicpte);

    /* 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, vicfpn, caller->active_mswp, swpfpn);
    /* Copy target frame from swap to mem */
    __swap_cp_page(caller->active_mswp, tgtfpn, caller->mram, vicfpn);

    /* Update page table */
    pte_set_swap(&vicpte, 0, swpfpn);

    /* Update its online status of the target page */
    // pte_set_fpn() & mm->pgd[pgn];
    pte_set_fpn(&pte, vicfpn);

    /* Update fifo_pgn of process */
    enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
  }
  // printf("PTE %08x\n",pte);
```

```
  *fpn = (pte)&0xFFF;

  return 0;
}
```

3.3.3 Hiện thực mm.c

Hàm vmap_page_range

Chức năng: Ánh xạ các khối trang vật lý tới một không gian địa chỉ ảo cho process.

```
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
  // uint32_t *pte = malloc(sizeof(uint32_t));
  struct framephy_struct *fpit = frames;
  int fpn;
  int pgit;

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

  /* TODO map range of frame to address space
   *      [addr to addr + pgnum*PAGING_PAGESZ
   *      in page table caller->mm->pgd[]
   */
  pthread_mutex_lock(&mem_lock);
  for (pgit = 0; pgit < pgnum; pgit++) {
    fpn = fpit->fpn;
    int pgn = PAGING_PGN((addr + pgit * PAGING_PAGESZ));
    ret_rg->rg_end += PAGING_PAGESZ;
    pte_set_fpn(&caller->mm->pgd[pgn], fpn);
    enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
    fpit = fpit->fp_next;

  }
  pthread_mutex_unlock(&mem_lock);

  return 0;
}
```

Hàm alloc_pages_range

Chức năng: Cấp vùng nhớ vật lý (physical) nhằm phục vụ cho việc map vùng nhớ tới vùng nhớ ảo, thuộc tính in_RAM được thêm vào để cung cấp thông tin về việc vùng nhớ đó nằm ở RAM hay SWAP

```
int alloc_pages_range(struct pcb_t *caller, int req_pgnum, struct framephy_struct
    **frm_lst) {
  int pgit, fpn;
  struct framephy_struct *newfp_str=NULL;

  pthread_mutex_lock(&mem_lock);
  for (pgit = 0; pgit < req_pgnum; pgit++) {
    if (MEMPHY_get_freefp(caller->mram, &fpn) == 0) {
      newfp_str = malloc(sizeof(struct framephy_struct));
      newfp_str->owner = caller->mm;
      if(frm_lst == NULL)
```

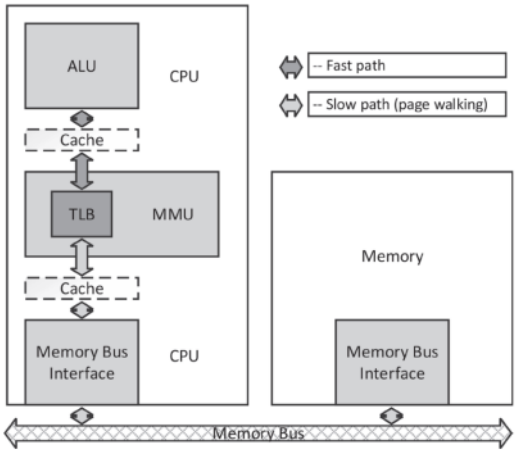
```
    {
      newfp_str->fp_next = NULL;
      newfp_str->fpn = fpn;
      *frm_lst = newfp_str;
    }
     // case 2: frame list is not empty
     else
     {
       newfp_str->fp_next = *frm_lst;
       newfp_str->fpn = fpn;
       *frm_lst = newfp_str;
     }
     MEMPHY_put_fp(caller->mram, fpn);

   } else { // ERROR CODE of obtaining somes but not enough frames
     int vicpgn, swfpn;

     /* Get victim frame from victim page*/
     find_victim_page(caller->mm,&vicpgn);
     fpn = PAGING_FPN(caller->mm->pgd[vicpgn]);

     /* Get free frame in MEMSWP */
     MEMPHY_get_freefp(caller->active_mswp, &swfpn);
     /* Copy victim frame to swap */
     __swap_cp_page(caller->mram, fpn, caller->active_mswp, swfpn);
     /* Update page table */
     pte_set_swap(&caller->mm->pgd[vicpgn], 0, swfpn);

     newfp_str = malloc(sizeof(struct framephy_struct));
     newfp_str->owner = caller->mm;

     if(frm_lst == NULL)
    {
      newfp_str->fp_next = NULL;
      newfp_str->fpn = fpn;
      *frm_lst = newfp_str;
    }
     // case 2: frame list is not empty
     else
     {
       newfp_str->fp_next = *frm_lst;
       newfp_str->fpn = fpn;
       *frm_lst = newfp_str;
     }

   }

 }
 pthread_mutex_unlock(&mem_lock);

 return 0;
}
```

3.4 Translation Lookaside Buffer (TLB)

3.4.1 Cơ sở lý thuyết

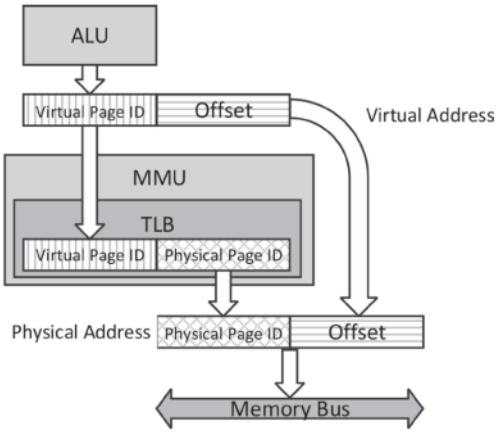
Translation Lookaside Buffer (TLB), hay còn gọi là bộ đệm chuyển đổi, đóng vai trò là bộ nhớ đệm cho bảng phân trang. Vì chi phí sản xuất bộ nhớ đệm cao nên TLB có dung lượng nhỏ. Sau khi tổng hợp hai phần trên, ta đã có được một hệ điều hành đơn giản với một góc nhìn khái

quát qua sơ đồ sau:



Hình ảnh mô phỏng TLB

Tương tự như các bộ nhớ đệm, mỗi phần tử của TLB sẽ có các thành phần là bit hợp lệ, nhãn, và vùng dữ liệu. Vùng dữ liệu trong các phần tử của TLB chứa các chỉ số trang vật lý được đệm lại từ bảng phân trang. Ngoài ra, các phần tử trong bảng phân trang còn chứa thêm các thông tin khác. Do đó, các khối của TLB cũng chứa những thông tin này trong vùng dữ liệu hoặc trong một vùng riêng nhưng ý nghĩa của nó cũng tương tự như vùng dữ liệu.



Dịch địa chỉ sử dụng TLB

3.4.2 Trả lời câu hỏi

Question: 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?
Answer: Trong hệ thống đa lõi, mỗi lõi CPU có MMU và TLB riêng, điều này đòi hỏi hệ điều hành và các giải pháp quản lý bộ nhớ phải thích nghi để tận dụng hiệu quả các cấu hình phần cứng này.

- Tăng đa luồng: Với mỗi lõi CPU có thể hoạt động trong ngữ cảnh khác nhau, có tiềm năng tăng đa luồng trong việc thực hiện các nhiệm vụ. Tuy nhiên, để khai thác tối đa tính đa luồng này, thuật toán lập lịch của hệ điều hành cần được tối ưu hóa để phân phối các nhiệm vụ một cách hiệu quả giữa các lõi.
- Tăng chi phí quản lý bộ nhớ: Mỗi lõi có MMU và TLB riêng, có thể dẫn đến tăng chi phí quản lý bộ nhớ. Điều này bởi vì các mục nhập TLB của mỗi lõi phải được duy trì đồng bộ với các thay đổi về ánh xạ bộ nhớ. Để đảm bảo tính đồng nhất giữa nhiều TLB, cần phải có sự phối hợp bổ sung và có thể gây ra mất hiệu suất.
- TLB Shootdowns: Khi ánh xạ bộ nhớ được sửa đổi, các mục nhập TLB tương ứng trong tất cả các lõi phải bị vô hiệu hóa hoặc cập nhật để phản ánh sự thay đổi. Quá trình này, được gọi là TLB shootdown, có thể tạo ra chi phí thêm, đặc biệt là trong các hệ thống có sự thay đổi thường xuyên về ánh xạ bộ nhớ.
- Tính nhất quán của TLB: Để duy trì tính nhất quán giữa các TLB trong hệ thống đa lõi, cần có các cơ chế phần cứng như giao thức TLB shootdown giữa các lõi. Bên cạnh đó, các phương pháp dựa trên phần mềm như giao thức vô hiệu hóa TLB có thể cần thiết để đảm bảo tính nhất quán của TLB qua các lõi.
- Các giải pháp dịch thích đa dạng: Với các kiến trúc CPU hiện đại có TLB 2 cấp, các giải pháp dịch thích cần phải thích nghi để tận dụng hiệu quả các cấu trúc này. Các giải pháp dịch thích thích nghi có thể điều chỉnh độ tổng hợp dịch thích dựa trên mẫu truy cập và việc sử dụng TLB để giảm thiểu chi phí dịch thích và cải thiện hiệu suất.

Tóm lại, trong khi hệ thống đa lõi với MMU và TLB riêng cho mỗi lõi cung cấp cơ hội tăng đa luồng, chúng cũng đặt ra thách thức về chi phí quản lý bộ nhớ và duy trì tính nhất quán. Sự phối hợp hiệu quả giữa các thành phần phần cứng và phần mềm là rất quan trọng để khai thác tối đa các lợi ích của các cấu hình phần cứng bộ nhớ này.

3.4.3 Cấu trúc

TLB được sử dụng để chứa các thông tin PID, VALID, SWAP, PGN, FRN. Vì mỗi ô trong storage có kích thước 1 BYTE, nên cần nhiều ô liền kề để tạo được một entry, ở đây nhóm đã sử dụng 5 BYTE liên tiếp. Cấu trúc cụ thể như sau:

3.4.3.1 tlb-alloc

Đây là một trong những hàm quan trọng nhất của TLB, hàm này thực hiện việc cấp phát bộ nhớ cho process, sau đó cập nhật vùng nhớ mới vào TLB. Việc cấp phát về cơ bản cũng giống với mem thông thường tuy nhiên sau khi bộ nhớ được cấp phát, nó sẽ được kết hợp với một số giá trị khác như : pid, pgn, valid,... và được lưu vào TLB để thuận tiện cho việc sử dụng.

3.4.3.2 tlb-free

Hàm này thực hiện việc giải phóng bộ nhớ đã cung cấp đang được chứa trong thanh ghi, việc giải phóng này sẽ đi với đó là giải phóng vùng TLB đang được sử dụng (nếu có). Sau khi dữ liệu được giải phóng.

3.4.3.3 tlb-read

Thực hiện việc tìm kiếm giá trị cần đọc trong TLB, nếu giá trị tồn tại thì đọc trực tiếp từ bộ nhớ, nếu không tồn tại thì sẽ tìm kiếm từ pagetable và cập nhật lại TLB.

3.4.3.4 tlb-write

Thực hiện việc tìm kiếm giá trị cần ghi trong TLB, nếu giá trị tồn tại thì ghi trực tiếp từ bộ nhớ, nếu không tồn tại thì sẽ tìm kiếm từ pagetable và cập nhật lại TLB.

3.4.3.5 Một số thay đổi trong thư viện mm.h

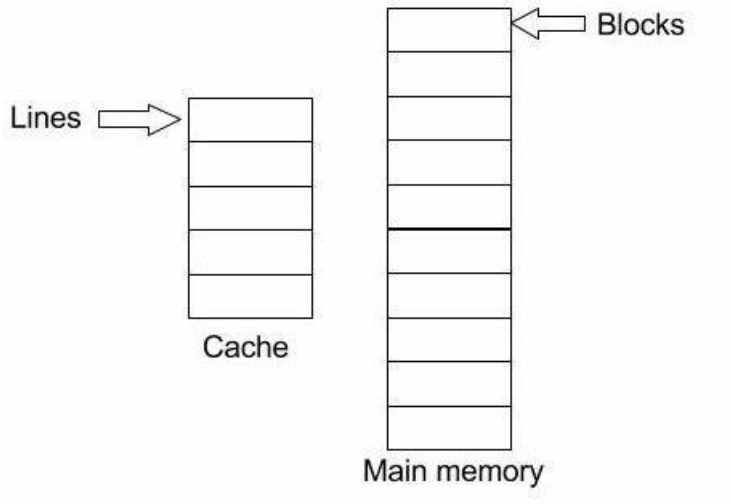
Thêm một số hàm:

```
int tlb_cache_write(struct memphy_struct* mp, int pid, int pgnum, int value);
int tlb_cache_read(struct memphy_struct* mp, int pid, int pgnum, int* value);
uint32_t tlb_create_value(uint32_t pte,uint32_t pgn,int valid); // Create a TLB
    value
int tlb_get_pid(struct memphy_struct* mp,int addr); // Get PID from TLB value
int tlb_empty(struct memphy_struct* mp, int addr); // Empty TLB value
uint32_t tlb_get_pgn(struct memphy_struct* mp, int addr); // Get PGN from TLB
    value
int tlb_set_value(struct memphy_struct* mp, int addr, uint32_t value,int pid); //
    Set TLB value
int tlb_get_addr(struct memphy_struct* mp, int pid, int pgn); // Get TLB address
int tlb_get_fpn(struct memphy_struct* mp, int addr); // Get FPN from TLB value
uint32_t tlb_get_value(struct memphy_struct* mp, int addr); // Get TLB value
```

3.4.4 Cách thức truy cập TLB

Ở bài tập này, nhóm thực hiện chế độ để có thể truy cập TLB là Direct. Ở đây, nhóm sử dụng hàm hash từ pid và pgn để ánh xạ ra địa chỉ vật lý của TLB.

- Ưu điểm: Sử dụng direct map sẽ giúp việc truy xuất TLB trở nên nhanh chóng và đơn giản.
- Nhược điểm: Không tận dụng được hết TLB, có thể xảy ra những trường hợp các process chiếm cùng 1 vùng dẫn đến khi đè nhau thường xuyên làm giảm hiệu xuất TLB, tăng thời gian cho việc truy xuất dữ liệu. Với các mô hình truy cập bộ nhớ phức tạp, direct mapped TLB có thể gây ra nhiều xung đột và giới hạn khả năng truy cập vào các trang bộ nhớ ảo.Một số trang bộ nhớ ảo có thể không được sử dụng hiệu quả trong direct mapped TLB, khiến cho TLB không tận dụng được toàn bộ khả năng lưu trữ của nó.



Cache Direct Mapping

3.4.5 Chiến lược cập nhật TLB

Khi TLB chưa đầy (tồn tại entry mà biến valid = 0) thì việc cập nhật là tìm frame đầu tiên còn trống và ghi giá trị vào đó. Khi TLB đầy thì việc cập nhật entry được diễn ra direct map ghi đè giá trị mới lên ô hiện tại.

3.5 Hiện thực

Hàm tlballoc

```
    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()*/
  int n_page = (PAGING_PAGE_ALIGNSZ(proc->mm->symrgtbl[reg_index].rg_end) -
    PAGING_PAGE_ALIGNSZ(proc->mm->symrgtbl[reg_index].rg_start))/PAGING_PAGESZ;
  for(int i=0;i<n_page;i++){
      int pgn = PAGING_PGN(proc->mm->symrgtbl[reg_index].rg_start)+i;
      // printf("TLB PGN : %d\n",pgn);

      tlb_set_value(proc->tlb,
                    tlb_get_addr(proc->tlb,proc->pid,pgn),
                    tlb_create_value(proc->mm->pgd[pgn],pgn,1),
                    proc->pid);
      // printf("DOC RA %08x\n %d",tlb_get_value(proc->tlb,tlb_get_addr(proc->tlb,
    proc->pid,pgn)),tlb_get_addr(proc->tlb,proc->pid,pgn));
  }
  TLBMEMPHY_dump(proc->tlb);
  return val;
}
```

Hàm tlbfree_data

```
int tlbfree_data(struct pcb_t *proc, uint32_t reg_index)
{
  int start_addr = PAGING_PGN(proc->mm->symrgtbl[reg_index].rg_start);
  int end_addr = PAGING_PGN(proc->mm->symrgtbl[reg_index].rg_end);

  __free(proc, 0, reg_index);
  for(int i=start_addr;i<=end_addr;i++){
      // tlb_set_value(proc->tlb,tlb_get_addr(proc->tlb,proc->pid,PAGING_PGN(proc
    ->mm->symrgtbl[reg_index].rg_start)+i),tlb_create_value(reg_index,addr+i,1),
    proc->pid);

      tlb_set_value(proc->tlb,tlb_get_addr(proc->tlb,proc->pid,PAGING_PGN(proc->mm
    ->symrgtbl[reg_index].rg_start)+i),tlb_create_value(0,0,0),proc->pid);
  }
  /* TODO update TLB CACHED frame num of freed page(s)*/
  /* by using tlb_cache_read()/tlb_cache_write()*/
  printf("FREE\n");
  TLBMEMPHY_dump(proc->tlb);

  return 0;
}
```

Hàm tlbread

```
int tlbread(struct pcb_t * proc, uint32_t source,
            uint32_t offset,  uint32_t destination)
{
  BYTE data;
  int frmnum = -1;
  int val =0;
  /* 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*/
  int page = PAGING_PGN((proc->mm->symrgtbl[source].rg_start + offset));
  int off = PAGING_OFFST((proc->mm->symrgtbl[source].rg_start + offset));
  frmnum = tlb_cache_read(proc->tlb, proc->pid, page, &val);
  if(frmnum<0){
    val = __read(proc, 0, source, offset, &data);
  }else{
    int addr =( frmnum << PAGING_ADDR_FPN_LOBIT )+ off;
    val = MEMPHY_read(proc->mram,addr,&data);
    printf("READ DATA: %d\n",data);
  }
#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);
  TLBMEMPHY_dump(proc->tlb);
#endif

  /* TODO update TLB CACHED with frame num of recent accessing page(s)*/
  /* by using tlb_cache_read()/tlb_cache_write()*/
  // tlb_cache_write(proc->tlb, proc->pid, source + offset, val);
  return val;
}
```

Hàm tlbwrite

```
int tlbwrite(struct pcb_t * proc, int data_,
             uint32_t destination, uint32_t offset)
{
  int val = 0;
  int frmnum = -1;
  BYTE data = data_;
  /* 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*/
  int t=0;
  /* 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*/
  int page = PAGING_PGN((proc->mm->symrgtbl[destination].rg_start + offset));
  int off = PAGING_OFFST((proc->mm->symrgtbl[destination].rg_start + offset));
  printf("PAGE %d\n",page);
  frmnum = tlb_cache_read(proc->tlb, proc->pid, page, &t);
  if(frmnum<0){
    val = __write(proc, 0, destination, offset,data);
  }else{
    int addr = (frmnum << PAGING_ADDR_FPN_LOBIT )+ off;
    val = MEMPHY_write(proc->mram,addr,data);
```

```
    printf("WRITE DATA: %d\n",data);
  }
#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);
  TLBMEMPHY_dump(proc->tlb);
#endif

  return val;
}
```

Hàm tlb_change_all_page_tables_of

```
int tlb_change_all_page_tables_of(struct pcb_t *proc,  struct memphy_struct * mp)
{
  /* TODO update all page table directory info
   *      in flush or wipe TLB (if needed)
   */
    tlb_flush_tlb_of(proc, mp);
    return 0;
}
```

Hàm tlb_flush_tlb_of

```
int tlb_flush_tlb_of(struct pcb_t *proc, struct memphy_struct * mp)
{
    for(int i=0; i<mp->maxsz; i++){
        if(tlb_get_pid(mp, i) == proc->pid){
            tlb_set_value(mp,i,tlb_create_value(0,0,0),-1);
        }
    }
    return 0;
}
```

Hàm tlb_cache_read

```
int tlb_cache_read(struct memphy_struct * mp, int pid, int pgnum, int* value)
{
   /* TODO: the identify info is mapped to
    *      cache line by employing:
    *      direct mapped, associated mapping etc.
    */
    if (mp == NULL || pgnum < 0 || pgnum >= mp->maxsz)
        return -1;
    int addr = tlb_get_addr(mp, pid, pgnum);
    int pid_tlb = tlb_get_pid(mp, addr);
    int pgnum_tlb = tlb_get_pgn(mp, addr);
    if(pid!=pid_tlb) return -1;
    if(pgnum!=pgnum_tlb) return -1;
    *value = tlb_get_fpn(mp, addr);
    return *value;
    return 0;
}
```

Hàm TLBMEMPHY_dump

```c
int TLBMEMPHY_dump(struct memphy_struct * mp)
{
   /*TODO dump memphy contnt mp->storage
    *     for tracing the memory content
    */
    if (mp == NULL) {
        printf("Error: Invalid memory physical structure\n");
        return -1;
    }
    for(int i = 0; i < mp->maxsz/5; i++){
        if(!tlb_empty(mp,i))
        printf("Memory physical content: %d: %08x %02x\n", i, tlb_get_value(mp,i), tlb_get_pid(mp,i));
    }

    return 0;
}
```

3.6 Kết quả thực thi

3.6.1 Testcase 1 CPU

Input:

```
6 1 3
1048576 16777216 0 0 0
0 p0s 0
0 p0s 0
2 p1s 15
```

Với các process p0s và p1s lần lượt là:

- p0s

```
1 10
calc
alloc 300 0
alloc 300 4
free 0
alloc 100 1
write 100 1 20
read 1 20 20
write 103 3 20
read 3 20 20
free 4

```

- p1s

```
1 10
calc
calc
calc
calc
calc
calc
calc
calc
calc
calc

```

Output:

```
Time slot   0
ld_routine
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
Time slot   1
        CPU 0: Dispatched process  1
        Loaded a process at input/proc/p0s, PID: 2 PRIO: 0
Time slot   2
SO TRANG DUOC CUNG CAP VA TLB PGN: 2 page 0 1
Memory physical content 10144: 80000001 01
Memory physical content 10145: 80002000 01
        Loaded a process at input/proc/p1s, PID: 3 PRIO: 15
Time slot   3
SO TRANG DUOC CUNG CAP VA TLB PGN: 2 page 2 3
Memory physical content 10144: 80000001 01
Memory physical content 10145: 80002000 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot   4
FREE 1 page
Time slot   5
SO TRANG DUOC CUNG CAP VA TLB PGN: 1 page 0
Memory physical content 10144: 80000004 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot   6
PAGE 0
WRITE DATA: 100
TLB hit at write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000004
00000004: 00000000
00000008: 80000003
00000012: 80000002
Memory physical content 10144: 80000004 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot   7
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  2
Time slot   8
SO TRANG DUOC CUNG CAP VA TLB PGN: 2 page 0 1
Memory physical content 10144: 80000004 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot   9
SO TRANG DUOC CUNG CAP VA TLB PGN: 2 page 2 3
Memory physical content 10144: 80000004 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot  10
FREE 1 page
Time slot  11
SO TRANG DUOC CUNG CAP VA TLB PGN: 1 page 0
Memory physical content 10144: 80000004 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot  12
PAGE 0
WRITE DATA: 100
TLB hit at write region=1 offset=20 value=100
```

```
print_pgtbl: 0 - 1024
00000000: 80000009
00000004: 00000000
00000008: 80000008
00000012: 80000007
Memory physical content 10144: 80000004 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot  13
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  1
READ DATA: 100
TLB hit at read region=1 offset=20
print_pgtbl: 0 - 1024
00000000: 80000004
00000004: 00000000
00000008: 80000003
00000012: 80000002
Memory physical content 10144: 80000004 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot  14
PAGE 0
WRITE DATA: 103
TLB hit at write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000004
00000004: 00000000
00000008: 80000003
00000012: 80000002
Memory physical content 10144: 80000004 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot  15
READ DATA: 103
TLB hit at read region=3 offset=20
print_pgtbl: 0 - 1024
00000000: 80000004
00000004: 00000000
00000008: 80000003
00000012: 80000002
Memory physical content 10144: 80000004 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot  16
FREE 1 page
Time slot  17
        CPU 0: Processed  1 has finished
        CPU 0: Dispatched process  2
READ DATA: 100
TLB hit at read region=1 offset=20
print_pgtbl: 0 - 1024
00000000: 80000009
00000004: 00000000
00000008: 80000008
00000012: 80000007
Memory physical content 10144: 80000004 01
Time slot  18
PAGE 0
WRITE DATA: 103
TLB hit at write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
```

```
00000000: 80000009
00000004: 00000000
00000008: 80000008
00000012: 80000007
Memory physical content 10144: 80000004 01
Time slot  19
READ DATA: 103
TLB hit at read region=3 offset=20
print_pgtbl: 0 - 1024
00000000: 80000009
00000004: 00000000
00000008: 80000008
00000012: 80000007
Memory physical content 10144: 80000004 01
Time slot  20
FREE 1 page
Time slot  21
        CPU 0: Processed  2 has finished
        CPU 0: Dispatched process  3
Time slot  22
Time slot  23
Time slot  24
Time slot  25
Time slot  26
Time slot  27
        CPU 0: Put process  3 to run queue
        CPU 0: Dispatched process  3
Time slot  28
Time slot  29
Time slot  30
Time slot  31
        CPU 0: Processed  3 has finished
        CPU 0 stopped
```

Biểu đồ Gantt

Giải thích kết quả

Dòng đầu tiên của input: 6 1 3, nghĩa là mỗi timeslice sẽ là 6 timeslot, có 1 CPU là CPU 0 và có tổng cộng 3 process.
Dòng thứ 2 của input: 1048576 16777216 0 0 0, kích thước của MEMRAM và MEMSWAP lần lượt là 1048576 và 16777216. Kết quả của từng timeslot được giải thích như sau:

- Timeslot 0: Process p0s được load với pid=1.
- Process 1 thực hiện lệnh CALC (không in gì ra màn hình). Process p0s được load với pid=2.

- Timeslot 2: Process 1 thực hiện lệnh ALLOC 300 0, thực hiện cấp phát 300 byte và lưu địa chỉ vào thanh ghi 0. Vì một trang có kích thước là 256 byte nên số trang cần được cung cấp là 2. Page number được gán là 0 và 1.
- Timeslot 3: Process 1 thực hiện lệnh ALLOC 300 4, thực hiện cấp phát 300 byte và lưu địa chỉ vào thanh ghi 4. Tương tự như trên, số trang cần được cung cấp là 2. Page number được gán là 2.
- Timeslot 4: Process 1 thực hiện lệnh FREE 0. Giải phóng vùng nhớ được cấp phát tại địa chỉ lưu tại thanh ghi 0.
- Timeslot 5: Process 1 thực hiện lệnh ALLOC 100 1, thực hiện cấp phát 100 byte và lưu địa chỉ vào thanh ghi 1. Số trang cần được cung cấp là 1. Page number được gán là 0.
- Timeslot 6: Process 1 thực hiện lệnh WRITE 100 1 20, thực hiện ghi giá trị 100 vào địa chỉ được lưu ở thanh ghi 1 cộng với offset 20. Lúc này chương trình tìm thấy địa chỉ trong TLB và thông báo 1 TLB hit.
- Timeslot 7: Process 1 đã chạy hết 6 timeslot cho phép nên process 1 được đặt vào run queue, process 2 thực hiện lệnh CALC.
- Timeslot 8 - Timeslot 12: Process 2 thực hiện các lệnh tương tự như process 1
- Timeslot 13: Process 2 đã chạy hết 6 timeslot cho phép nên process 2 được đặt vào run queue, process 1 thực hiện lệnh READ 1 20 20, thực hiện đọc 1 byte dữ liệu tại địa chỉ được lưu tại thanh ghi 1 cộng với offset 20, sau đó lưu dữ liệu vào thanh ghi 20. Lúc này chương trình tìm thấy địa chỉ trong TLB và thông báo 1 TLB hit. Do trước đó chương trình đã thực hiện lệnh ghi giá trị 100 vào địa chỉ cần đọc, nên giá trị đọc được là 100.
- Timeslot 14: Process 1 thực hiện lệnh WRITE 103 3 20, thực hiện ghi giá trị 103 vào địa chỉ được lưu ở thanh ghi 3 cộng với offset 20. Lúc này chương trình tìm thấy địa chỉ trong TLB và thông báo 1 TLB hit.
- Timeslot 15: Process 1 thực hiện lệnh READ 3 20 20, thực hiện đọc 1 byte dữ liệu tại địa chỉ được lưu tại thanh ghi 3 cộng với offset 20, sau đó lưu dữ liệu vào thanh ghi 20. Lúc này chương trình tìm thấy địa chỉ trong TLB và thông báo 1 TLB hit. Do trước đó chương trình đã thực hiện lệnh ghi giá trị 103 vào địa chỉ cần đọc, nên giá trị đọc được là 103.
- Timeslot 16: Process 1 thực hiện lệnh FREE 4, thực hiện giải phóng vùng nhớ được cấp phát tại địa chỉ lưu ở thanh ghi 4.
- Timeslot 17: Kết thúc process 1. Process 2 tiếp tục thực thi lệnh tiếp theo (tương tự như process 1)
- Timeslot 18 - Timeslot 20: Process 2 thực hiện các lệnh tương tự như process 1.
- Timeslot 21: Kết thúc process 2. Process 3 được đưa vào thực thi.
- Timeslot 21 - Timeslot 30: Process 3 thực hiện các lệnh CALC. Tại timeslot 27, do đã hết 6 timeslot cho process 3 nên process 3 được đưa vào run queue, sau đó lại tiếp tục đưa đưa ra thực thi.
- Timeslot 31: Kết thúc process 3. Kết thúc chương trình.

3.6.2 Testcase 2 CPU

Input:

```
6 2 3
1048576 16777216 0 0 0
0 p0s 0
0 p0s 0
2 p1s 15
```

Với các process p0s và p1s như đã trình bày ở phần trên.

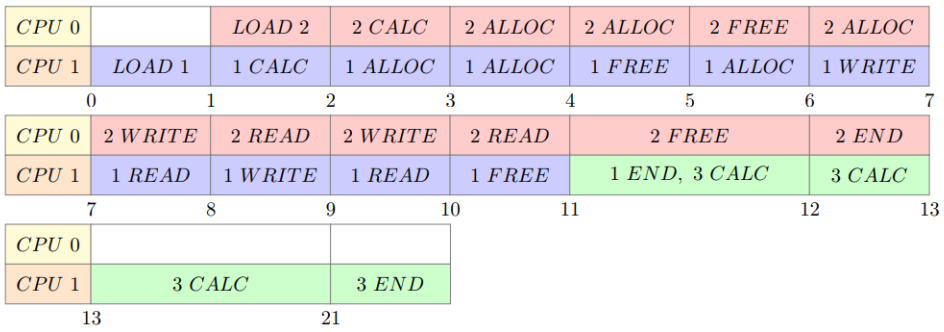
Output:

```
Time slot   0
ld_routine
        Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
        CPU 1: Dispatched process  1
Time slot   1
        Loaded a process at input/proc/p0s, PID: 2 PRIO: 0
Time slot   2
        CPU 0: Dispatched process  2
SO TRANG DUOC CUNG CAP VA TLB PGN: 2 page 0 1
Memory physical content 10144: 80000001 01
Memory physical content 10145: 80002000 01
        Loaded a process at input/proc/p1s, PID: 3 PRIO: 15
Time slot   3
SO TRANG DUOC CUNG CAP VA TLB PGN: 2 page 0 1
SO TRANG DUOC CUNG CAP VA TLB PGN: 2 page 2 3
Memory physical content 10144: 80000001 01
Memory physical content 10145: 80002000 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Memory physical content 10144: 80000001 01
Memory physical content 10145: 80002000 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
FREE 1 page
Time slot   4
SO TRANG DUOC CUNG CAP VA TLB PGN: 2 page 2 3
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot   5
FREE 1 page
SO TRANG DUOC CUNG CAP VA TLB PGN: 1 page 0
Memory physical content 10144: 80000008 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot   6
PAGE 0
WRITE DATA: 100
TLB hit at write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000008
00000004: 00000000
00000008: 80000003
00000012: 80000002
SO TRANG DUOC CUNG CAP VA TLB PGN: 1 page 0
Memory physical content 10144: 80000008 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Memory physical content 10144: 80000008 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
```

```
Time slot   7
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
READ DATA: 100
TLB hit at read region=1 offset=20
print_pgtbl: 0 - 1024
00000000: 80000008
00000004: 00000000
00000008: 80000003
00000012: 80000002
PAGE 0
WRITE DATA: 100
TLB hit at write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000009
00000004: 00000000
00000008: 80000007
00000012: 80000006
Memory physical content 10144: 80000008 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Memory physical content 10144: 80000008 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot   8
PAGE 0
WRITE DATA: 103
TLB hit at write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000008
00000004: 00000000
00000008: 80000003
00000012: 80000002
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
READ DATA: 100
TLB hit at read region=1 offset=20
print_pgtbl: 0 - 1024
00000000: 80000009
00000004: 00000000
00000008: 80000007
00000012: 80000006
Memory physical content 10144: 80000008 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Memory physical content 10144: 80000008 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot   9
READ DATA: 103
TLB hit at read region=3 offset=20
print_pgtbl: 0 - 1024
00000000: 80000008
00000004: 00000000
00000008: 80000003
00000012: 80000002
PAGE 0
WRITE DATA: 103
TLB hit at write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000009
00000004: 00000000
```

```
00000008: 80000007
00000012: 80000006
Memory physical content 10144: 80000008 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Memory physical content 10144: 80000008 01
Memory physical content 10146: 80004003 01
Memory physical content 10147: 80006002 01
Time slot  10
READ DATA: 103
TLB hit at read region=3 offset=20
print_pgtbl: 0 - 1024
00000000: 80000009
00000004: 00000000
00000008: 80000007
00000012: 80000006
FREE 1 page
Memory physical content 10144: 80000008 01
Time slot  11
FREE 1 page
        CPU 1: Processed  1 has finished
        CPU 1: Dispatched process  3
Time slot  12
        CPU 0: Processed  2 has finished
        CPU 0 stopped
Time slot  13
Time slot  14
Time slot  15
Time slot  16
Time slot  17
        CPU 1: Put process  3 to run queue
        CPU 1: Dispatched process  3
Time slot  18
Time slot  19
Time slot  20
Time slot  21
        CPU 1: Processed  3 has finished
        CPU 1 stopped
```

Biểu đồ Gantt



Giải thích kết quả

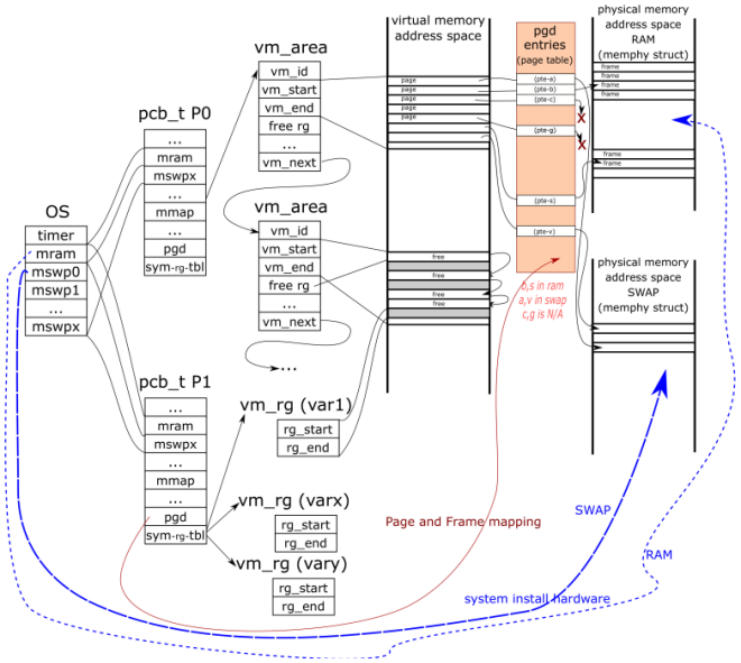
Input tương tự như testcase ở phần 4.4.8a. Chỉ khác là testcase này sử dụng 2 CPU là CPU 0 và CPU 1. Kết quả tại từng timeslot được giải thích như sau:

- Timeslot 0: Load process 1 (p0s) vào CPU 1. Và được thực thi luôn lệnh CALC ngay trong timeslot này.

- Timeslot 1: process 2 (p0s) được load vào CPU 0.
- Timeslot 2: Process 1 thực hiện lệnh ALLOC 300 0, thực hiện cấp phát 300 byte và lưu địa chỉ vùng nhớ vào thanh ghi 0. Đồng thời, process 2 thực hiện lệnh CALC. Lúc này, process 3 (p1s) cũng được load và đưa vào run queue.
- Timeslot 3: Process 1 thực hiện lệnh ALLOC 300 4, thực hiện cấp phát 300 byte và lưu địa chỉ vùng nhớ vào thanh ghi 4. Đồng thời, process 2 thực hiện lệnh ALLOC 300 0 (tương tự process 1).
- Timeslot 4: Process 1 thực hiện lệnh FREE 0, thực hiện giải phóng vùng nhớ tại địa chỉ lưu ở thanh ghi 0. Đồng thời, process 2 thực hiện lệnh ALLOC 300 4 (tương tự process 1).
- Timeslot 5: Process 1 thực hiện lệnh ALLOC 100 1, thực hiện cấp phát 100 byte và lưu địa chỉ vào thanh ghi 1. Số trang cần được cung cấp là 1. Page number được gán là 0. Đồng thời, process 2 thực hiện lệnh FREE 0 (tương tự process 1)
- Timeslot 6: Process 1 thực hiện lệnh WRITE 100 1 20, thực hiện ghi giá trị 100 vào địa chỉ được lưu ở thanh ghi 1 cộng với offset 20. Lúc này chương trình tìm thấy địa chỉ trong TLB và thông báo 1 TLB hit. Đồng thời, process 2 thực hiện lệnh ALLOC 100 1 (tương tự process 1)
- Timeslot 7: Process 1 thực hiện lệnh READ 1 20 20, thực hiện đọc 1 byte dữ liệu tại địa chỉ được lưu tại thanh ghi 1 cộng với offset 20, sau đó lưu dữ liệu vào thanh ghi 20. Lúc này chương trình tìm thấy địa chỉ trong TLB và thông báo 1 TLB hit. Do trước đó chương trình đã thực hiện lệnh ghi giá trị 100 vào địa chỉ cần đọc, nên giá trị đọc được là 100. Đồng thời, process 2 thực hiện lệnh WRITE 100 1 20 (tương tự process 1)
- Timeslot 8: Process 1 thực hiện lệnh WRITE 103 3 20, thực hiện ghi giá trị 103 vào địa chỉ được lưu ở thanh ghi 3 cộng với offset 20. Lúc này chương trình tìm thấy địa chỉ trong TLB và thông báo 1 TLB hit. Đồng thời, process 2 thực hiện lệnh READ 1 20 20 (tương tự process 1)
- Timeslot 9: Process 1 thực hiện lệnh READ 3 20 20, thực hiện đọc 1 byte dữ liệu tại địa chỉ được lưu tại thanh ghi 3 cộng với offset 20, sau đó lưu dữ liệu vào thanh ghi 20. Lúc này chương trình tìm thấy địa chỉ trong TLB và thông báo 1 TLB hit. Do trước đó chương trình đã thực hiện lệnh ghi giá trị 103 vào địa chỉ cần đọc, nên giá trị đọc được là 103. Đồng thời, process 2 thực hiện lệnh WRITE 103 3 20 (tương tự process 1)
- Timeslot 10: Process 1 thực hiện lệnh FREE 4, thực hiện giải phóng vùng nhớ được cấp phát tại địa chỉ lưu ở thanh ghi 4. Đồng thời, process 2 thực hiện lệnh READ 3 20 20 (tương tự process 1)
- Timeslot 11: Kết thúc process 1. Lúc này, process 3 sẽ được load vào CPU 1 và thực hiện lệnh CALC đầu tiên. Đồng thời, process 2 thực hiện lệnh FREE 4 (tương tự process 1)
- Timeslot 12: Kết thúc process 2. Lúc này không còn process nào trong hàng đợi nữa nên CPU 0 dừng lại. Đồng thời, prcess 3 thực hiện lệnh CALC thứ hai.
- Timeslot 13 - Timeslot 20: Process 3 thực hiện các lệnh CALC còn lại.
- Timeslot 21: Process 3 kết thúc. CPU 1 cũng dừng lại. Kết thúc chương trình.

4 Put It All Together

4.1 Cơ sở lý thuyết



Hình 3: Tổng hợp các module

4.2 Trả lời câu hỏi

Question: 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. Note: You need to run two versions of your simple OS: the program with/without synchronization, then observe their performance based on demo results and explain their diff.

Answer: Trong một hệ điều hành multi-tasking, đồng bộ hóa là rất quan trọng để đảm bảo rằng nhiều process hoặc thread không can thiệp vào việc thực thi lẫn nhau và truy cập vào tài nguyên được chia sẻ. Nếu đồng bộ hóa không được xử lý đúng cách, nó có thể dẫn đến tình trạng race conditions, deadlocks và các sự cố khác có thể khiến hệ thống trở nên không ổn định hoặc không phản hồi.

Ví dụ: Nếu hai tác vụ cố gắng truy cập đồng thời cùng một tài nguyên mà không đồng bộ hóa phù hợp, điều đó có thể dẫn đến tình trạng chạy đua trong đó hành vi của hệ thống trở nên khó đoán. Hãy xem xét một tình huống trong đó hai process, process 1 và process 2 đều cùng cố gắng truy cập vào một tài nguyên được chia sẻ, chẳng hạn như một biến toàn cục, mà không có sự đồng bộ hóa thích hợp. Cả hai process đều đọc giá trị của biến và tăng giá trị đó, sau đó ghi giá trị mới trở lại biến. Nếu process 1 và process 2 thực hiện đồng thời và cả hai đều đọc giá trị của biến cùng một lúc, cả hai sẽ nhận được cùng một giá trị. Sau đó, cả hai sẽ tăng giá trị này và ghi lại vào biến. Tuy nhiên, vì cả hai tác vụ đều tăng giá trị lên 1, nên giá trị cuối cùng của

biến lẽ ra phải được tăng lên 2, nhưng nó chỉ được tăng lên 1 vì mỗi tác vụ chỉ nhìn thấy giá trị trước đó.

Tài liệu

[1] Slide bài giảng thuộc sở hữu của Trường Đại học Bách Khoa.

[2] Sách Operating System - Abraham Silberschatz