TRƯỜNG ĐẠI HỌC BÁCH KHOA
ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

HỆ ĐIỀU HÀNH (CO2018)

BÀI TẬP LỚN

"Simple Operating System"

Giáo viên hướng dẫn: Bùi Xuân Giang

Sinh viên: Nguyễn Tấn Phát - 2352888 *(CN01, **Nhóm trưởng**)*
Vũ Hà Như Ngọc - 2352818 *(CN01)*
Lê Diệu Quỳnh - 2353036 *(CN01)*
Lương Đức Huy - 2352384 *(CN02)*
Phạm Trần Nguyên Chương - 2352142 *(CN01)*

THÀNH PHỐ HỒ CHÍ MINH, THÁNG 4 NĂM 2025

Mục lục

Listings

Danh sách thành viên và phân công công việc

STT	Họ Tên	MSSV	Nhiệm vụ	% Hoàn thành
1	Nguyễn Tấn Phát	2352888	- Viết báo cáo LaTeX - Code Memory	100%
2	Vũ Hà Như Ngọc	2352818	- Code Scheduler	100%
3	Lê Diệu Quỳnh	2353036	- Viết báo cáo LaTeX - Code Memory	100%
4	Lương Đức Huy	2352384	- Code System Call	100%
5	Phạm Trần Nguyên Chương	2352142	- Code Memory	100%

Bảng 1: Danh sách thành viên và phân công công việc

Lời cảm ơn

1. Lời cảm ơn

Trong quá trình thực hiện nghiên cứu và mô phỏng hệ điều hành đơn giản, chúng em đã nhận được sự giúp đỡ và hỗ trợ từ nhiều cá nhân, tổ chức và nguồn tài liệu giá trị. Đây là một cơ hội để chúng em bày tỏ lòng biết ơn chân thành đến tất cả những đóng góp đó.

2. Ghi nhận công cụ và tài nguyên hỗ trợ

Đầu tiên, nhóm chúng em xin gửi lời cảm ơn sâu sắc đến các tác giả đã cung cấp nền tảng lý thuyết và các bài nghiên cứu trước đây đã đóng vai trò quan trọng trong việc định hướng và giải quyết bài toán này. Trong quá trình triển khai bài tập, nhóm chúng em đã tham khảo và sử dụng một số công cụ và tài nguyên sau:

- **Visual Studio Code** – môi trường lập trình và gỡ lỗi.
- **GCC + Makefile** – công cụ biên dịch và tự động hoá build hệ thống mô phỏng.
- **LaTeX + Overleaf** – trình bày báo cáo và sơ đồ chuyên nghiệp.
- **TikZ** – hỗ trợ vẽ biểu đồ Gantt và sơ đồ luồng system call.
- **Tài liệu tham khảo**: OSDev Wiki, Operating System Concepts (Silberschatz), Modern Operating Systems (Tanenbaum), và các bài viết từ GeeksforGeeks.

Nhờ những công cụ này, nhóm chúng em đã có thể triển khai bài tập một cách hiệu quả, trực quan và tiết kiệm thời gian trong cả lập trình lẫn trình bày báo cáo.

3. Cảm ơn các cá nhân và tổ chức

Chúng em xin bày tỏ lòng biết ơn đến giảng viên đã hướng dẫn, cung cấp ý kiến phản hồi quý giá và định hướng cho chúng em trong suốt quá trình thực hiện bài toán này. Ngoài ra, chúng em cũng xin chân thành cảm ơn Khoa Khoa học và Kỹ thuật Máy tính, Trường Đại học Bách Khoa – Đại học Quốc gia TP.HCM đã tạo điều kiện thuận lợi để nhóm có cơ hội học tập, nghiên cứu và áp dụng các kiến thức vào mô phỏng hệ điều hành đơn giản.

Một lần nữa, nhóm xin chân thành cảm ơn tất cả các cá nhân, đơn vị và nguồn tài liệu đã đóng góp vào sự thành công của bài tập lớn này.

1. Lập lịch tiến trình với Multi-Level Queue (MLQ)

1.1. Giới thiệu

Thuật toán lập lịch **Multi-Level Queue (MLQ)** là một kỹ thuật quản lý tiến trình bằng cách chia các tiến trình thành nhiều hàng đợi dựa trên mức độ ưu tiên. Mỗi hàng đợi có thể sử dụng một thuật toán lập lịch riêng biệt, nhưng trong hệ điều hành mô phỏng này, tất cả các hàng đợi đều sử dụng **Round Robin** với `time slice` cố định.

1.2. Cơ chế hoạt động trong hệ thống mô phỏng

Hệ thống có:

- `MAX_PRIO = 140`: tổng số mức ưu tiên.
- Mỗi tiến trình có thuộc tính `prio` (mức ưu tiên) được xác định khi nạp từ file mô tả.
- Mỗi hàng đợi tương ứng với một mức `prio`, tạo nên một danh sách 140 hàng đợi.
- CPU duyệt các hàng đợi theo thứ tự từ ưu tiên cao đến thấp (từ `prio = 0`).

Một khái niệm đặc biệt là **slot**:

- Slot là số lượt phục vụ CPU mà một hàng đợi được cấp trong mỗi vòng lặp.
- Slot được tính bằng công thức:

$$\texttt{slot} = \texttt{MAX\_PRIO} - \texttt{prio}$$

- Điều này đảm bảo rằng các tiến trình có mức ưu tiên cao hơn (`prio` thấp) được cấp CPU nhiều hơn.

1.3. Vận hành hàm `get_proc()`

Hàm `get_proc()` trong `sched.c` duyệt qua danh sách hàng đợi theo thứ tự slot. Tại mỗi hàng đợi, nếu có tiến trình đang chờ, nó được đưa ra khỏi hàng đợi và cấp CPU để thực thi trong một `time slice`.
Nếu tiến trình chưa hoàn tất sau time slice, nó được đưa trở lại hàng đợi tương ứng với cùng mức `prio`.

1.4. Ví dụ minh họa

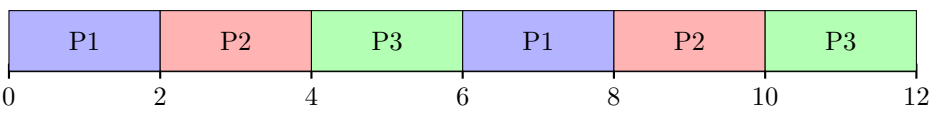
Giả sử có 3 tiến trình P1 (prio=0), P2 (prio=1), P3 (prio=5) với time slice = 2s.

Tiến trình	Mức ưu tiên (prio)	Slot = 140 - prio
P1	0	140
P2	1	139
P3	5	135

Tiến trình P1 sẽ luôn được chọn phục vụ trước P2 và P3, theo chính sách slot.

1.5. Biểu đồ Gantt minh họa

Biểu đồ sau minh họa việc lập lịch CPU cho 3 tiến trình với mức ưu tiên khác nhau theo chính sách MLQ và `time slice = 2s`:



Như minh họa, CPU luôn ưu tiên tiến trình ở hàng đợi có độ ưu tiên cao hơn. Sau mỗi `time slice`, CPU kiểm tra lại từ đầu để chọn tiến trình tiếp theo theo chính sách MLQ.

1.6. Ưu điểm và nhược điểm của MLQ

Ưu điểm:

- Phân loại tiến trình rõ ràng theo mức độ ưu tiên.
- Dễ kiểm soát tài nguyên và phục vụ tiến trình thời gian thực.
- Có thể điều chỉnh linh hoạt chính sách lập lịch giữa các hàng đợi.

Nhược điểm:

- Tiến trình ở hàng đợi ưu tiên thấp dễ bị đói CPU.
- Việc cấu hình số lượng hàng đợi và thuật toán từng hàng đòi hỏi hiểu sâu.
- Không có cơ chế tự động thay đổi mức ưu tiên theo thời gian (nếu không tích hợp feedback).

1.7. Code minh họa cho MLQ

1.7.1. File queue.c

`queue.c` cung cấp cấu trúc dữ liệu hàng đợi (queue) được sử dụng trong hệ điều hành mô phỏng để quản lý danh sách tiến trình và các đối tượng hệ thống khác. Hàng đợi này là hàng đợi liên kết đơn, hỗ trợ các thao tác như khởi tạo, thêm phần tử vào cuối, lấy phần tử đầu, xóa phần tử, và kiểm tra trạng thái rỗng.

Listing 1: queue.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.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) {
        if(q->size < MAX_QUEUE_SIZE){
                q->proc[q->size] = proc;
                q->size++;
        }
}

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

Giải thích về các hàm

`empty`

- **Chức năng:** Kiểm tra xem hàng đợi có trống hay không.

- **Tham số:**
  - `q`: Một con trỏ tới hàng đợi kiểu `struct queue_t*`.
- **Hoạt động:**
  1. Nếu `q` là `NULL`, hàm trả về 1 (hàng đợi trống).
  2. Nếu không, hàm trả về 1 nếu `q->size` bằng 0 (hàng đợi trống), và 0 nếu không (hàng đợi không trống).
- **Giá trị trả về:** 1 nếu hàng đợi trống, 0 nếu không.
- **Mục đích:** Cho phép kiểm tra xem có tiến trình nào đang chờ trong hàng đợi hay không.

`enqueue`

- **Chức năng:** Thêm một tiến trình vào cuối hàng đợi.
- **Tham số:**
  - `q`: Một con trỏ tới hàng đợi kiểu `struct queue_t*`.
  - `proc`: Một con trỏ tới tiến trình kiểu `struct pcb_t*` cần thêm vào hàng đợi.
- **Hoạt động:**
  1. Kiểm tra xem kích thước của hàng đợi (`q->size`) có nhỏ hơn kích thước tối đa (`MAX_QUEUE_SIZE`) hay không.
  2. Nếu có, hàm sẽ thêm tiến trình `proc` vào cuối hàng đợi (vị trí `q->size` trong mảng `q->proc`).
  3. Sau đó, hàm tăng kích thước của hàng đợi (`q->size++`).
- **Giá trị trả về:** Không có (void).
- **Mục đích:** Thêm một tiến trình vào hàng đợi để nó có thể được xử lý sau.
- **Lưu ý:** Nếu hàng đợi đã đầy (kích thước bằng `MAX_QUEUE_SIZE`), hàm sẽ không thêm tiến trình vào.

`dequeue`

- **Chức năng:** Lấy ra tiến trình có độ ưu tiên cao nhất từ hàng đợi.
- **Tham số:**
  - `q`: Một con trỏ tới hàng đợi kiểu `struct queue_t*`.
- **Hoạt động:**
  1. Kiểm tra xem hàng đợi có `NULL` hoặc trống (kích thước bằng 0) hay không. Nếu có, hàm trả về `NULL`.
  2. Nếu không, hàm tìm tiến trình có độ ưu tiên cao nhất trong hàng đợi. Biến `highest` lưu trữ chỉ số của tiến trình có độ ưu tiên cao nhất hiện tại. Ban đầu, nó được đặt thành 0.
  3. Hàm duyệt qua tất cả các tiến trình trong hàng đợi (từ chỉ số 1 đến `q->size - 1`).
  4. Với mỗi tiến trình, hàm so sánh độ ưu tiên của nó với độ ưu tiên của tiến trình có chỉ số `highest`. Nếu độ ưu tiên của tiến trình hiện tại cao hơn (giá trị `priority` nhỏ hơn, vì độ ưu tiên thấp hơn tương ứng với giá trị số lớn hơn), thì `highest` được cập nhật thành chỉ số của tiến trình hiện tại.
  5. Sau khi tìm thấy tiến trình có độ ưu tiên cao nhất, hàm lưu trữ con trỏ tới tiến trình đó vào biến `proc`.
  6. Hàm sau đó loại bỏ tiến trình khỏi hàng đợi bằng cách dịch chuyển tất cả các tiến trình phía sau tiến trình có độ ưu tiên cao nhất một vị trí về phía trước.
  7. Cuối cùng, hàm giảm kích thước của hàng đợi (`q->size-`) và trả về con trỏ tới tiến trình có độ ưu tiên cao nhất (`proc`).

- **Giá trị trả về:** Một con trỏ tới tiến trình có độ ưu tiên cao nhất (`struct pcb_t*`) hoặc `NULL` nếu hàng đợi trống.
- **Mục đích:** Lấy ra tiến trình có độ ưu tiên cao nhất từ hàng đợi để nó có thể được chạy.
- **Lưu ý:** Việc dịch chuyển các phần tử mảng trong quá trình dequeue có thể không hiệu quả đối với các hàng đợi lớn. Các triển khai hàng đợi ưu tiên khác, chẳng hạn như sử dụng heap, có thể hiệu quả hơn.

1.7.2. File sched.c

`sched.c` hiện thực bộ lập lịch tiến trình (Scheduler) theo thuật toán đa mức (Multi-Level Queue - MLQ). Các tiến trình được phân chia theo mức độ ưu tiên và được lập lịch thực thi lần lượt từ hàng đợi có mức ưu tiên cao xuống thấp. Module này xử lý logic lựa chọn tiến trình kế tiếp, tính toán thời gian thực thi, và quản lý chuyển đổi trạng thái tiến trình.

Listing 2: sched.c

```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;
static struct queue_t running_list;
#ifdef MLQ_SCHED
static struct queue_t mlq_ready_queue[MAX_PRIO];
static int slot[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;
                slot[i] = MAX_PRIO - i;
        }
#endif
        ready_queue.size = 0;
        run_queue.size = 0;
        pthread_mutex_init(&queue_lock, NULL);
}

#ifdef MLQ_SCHED
struct pcb_t * get_mlq_proc(void) {
        struct pcb_t * proc = NULL;
    pthread_mutex_lock(&queue_lock);

        for (int prio = 0; prio < MAX_PRIO; prio++) {
                if (!empty(&mlq_ready_queue[prio])) {
                        if (slot[prio] > 0) {
                                proc = dequeue(&mlq_ready_queue[prio]);
                                slot[prio]--;
                                break;
                        }
                }
        }
```

```
53
54          if (proc == NULL) {
55                  for (int i = 0; i < MAX_PRIO; i++) {
56                          slot[i] = MAX_PRIO - i;
57                  }
58
59                  for (int prio = 0; prio < MAX_PRIO; prio++) {
60                          if (!empty(&mlq_ready_queue[prio])) {
61                                  proc = dequeue(&mlq_ready_queue[prio]);
62                                  slot[prio]--;
63                                  break;
64                          }
65                  }
66          }
67
68          pthread_mutex_unlock(&queue_lock);
69          return proc;
70  }
71
72  void put_mlq_proc(struct pcb_t * proc) {
73          if (proc == NULL || proc->prio < 0 || proc->prio >= MAX_PRIO) return;
74          pthread_mutex_lock(&queue_lock);
75          enqueue(&mlq_ready_queue[proc->prio], proc);
76          pthread_mutex_unlock(&queue_lock);
77  }
78
79  void add_mlq_proc(struct pcb_t * proc) {
80          if (proc == NULL || proc->prio < 0 || proc->prio >= MAX_PRIO) return;
81          pthread_mutex_lock(&queue_lock);
82          enqueue(&mlq_ready_queue[proc->prio], proc);
83          pthread_mutex_unlock(&queue_lock);
84  }
85
86  struct pcb_t * get_proc(void) {
87          return get_mlq_proc();
88  }
89
90  void put_proc(struct pcb_t * proc) {
91          if (proc == NULL || proc->prio < 0 || proc->prio >= MAX_PRIO) return;
92
93          pthread_mutex_lock(&queue_lock);
94          enqueue(&running_list, proc);
95          pthread_mutex_unlock(&queue_lock);
96
97          put_mlq_proc(proc);
98  }
99
100 void add_proc(struct pcb_t * proc) {
101         if (proc == NULL || proc->prio < 0 || proc->prio >= MAX_PRIO) return;
102
103         pthread_mutex_lock(&queue_lock);
104     enqueue(&running_list, proc);
105     pthread_mutex_unlock(&queue_lock);
106
107         add_mlq_proc(proc);
108 }
109 #else
110 struct pcb_t * get_proc(void) {
111         struct pcb_t * proc = NULL;
112         pthread_mutex_lock(&queue_lock);
113     if (!empty(&ready_queue)) {
114         proc = dequeue(&ready_queue);
115     }
116     pthread_mutex_unlock(&queue_lock);
117
118     return proc;
119 }
120
121
122 void put_proc(struct pcb_t * proc) {
123         if (proc == NULL) return;
124
```

```
125         pthread_mutex_lock(&queue_lock);
126         enqueue(&running_list, proc);
127         pthread_mutex_unlock(&queue_lock);
128
129         pthread_mutex_lock(&queue_lock);
130         enqueue(&run_queue, proc);
131         pthread_mutex_unlock(&queue_lock);
132 }
133
134 void add_proc(struct pcb_t * proc) {
135         if (proc == NULL) return;
136
137         pthread_mutex_lock(&queue_lock);
138         enqueue(&running_list, proc);
139         pthread_mutex_unlock(&queue_lock);
140
141         pthread_mutex_lock(&queue_lock);
142         enqueue(&ready_queue, proc);
143         pthread_mutex_unlock(&queue_lock);
144 }
145 #endif
```

Giải thích về các hàm

queue_empty

- **Chức năng:** Kiểm tra xem tất cả các hàng đợi có trống hay không.
- **Hoạt động:**
  - Nếu `MLQ_SCHED` được định nghĩa, hàm sẽ duyệt qua tất cả các hàng đợi ưu tiên (`mlq_ready_queue`) và trả về -1 nếu có bất kỳ hàng đợi nào không trống.
  - Nếu `MLQ_SCHED` không được định nghĩa, hàm sẽ kiểm tra xem cả `ready_queue` và `run_queue` đều trống hay không.
- **Mục đích:** Xác định xem có còn tiến trình nào đang chờ xử lý hay không, thường được sử dụng để quyết định xem hệ thống có thể ở trạng thái nghỉ (idle) hay không.

init_scheduler

- **Chức năng:** Khởi tạo bộ lập lịch.
- **Hoạt động:**
  - Nếu `MLQ_SCHED` được định nghĩa, hàm sẽ khởi tạo từng hàng đợi ưu tiên trong `mlq_ready_queue` bằng cách đặt kích thước của chúng về 0 và gán số lượng slot ban đầu cho mỗi mức độ ưu tiên dựa trên công thức `MAX_PRIO - i`. Điều này có nghĩa là các hàng đợi có độ ưu tiên cao hơn sẽ có nhiều slot hơn, cho phép chúng chạy nhiều tiến trình hơn trước khi các hàng đợi có độ ưu tiên thấp hơn có cơ hội.
  - Nếu `MLQ_SCHED` không được định nghĩa, hàm sẽ chỉ khởi tạo `ready_queue` và `run_queue` bằng cách đặt kích thước của chúng về 0.
  - Hàm cũng khởi tạo `queue_lock` mutex để bảo vệ các hàng đợi.
- **Mục đích:** Chuẩn bị bộ lập lịch cho hoạt động, thiết lập các cấu trúc dữ liệu và khóa cần thiết.

get_mlq_proc (Khi MLQ_SCHED được định nghĩa)

- **Chức năng:** Lấy một tiến trình từ hàng đợi ưu tiên cao nhất có slot còn lại.
- **Hoạt động:**
  1. Hàm khóa `queue_lock` để bảo vệ các hàng đợi.

2. Hàm duyệt qua các hàng đợi ưu tiên từ cao đến thấp (từ 0 đến `MAX_PRIO - 1`).
3. Với mỗi hàng đợi, hàm kiểm tra xem nó có trống hay không và liệu số lượng slot còn lại của nó có lớn hơn 0 hay không.
4. Nếu cả hai điều kiện đều đúng, hàm sẽ lấy một tiến trình từ hàng đợi bằng cách sử dụng hàm `dequeue`, giảm số lượng slot còn lại của hàng đợi và trả về tiến trình đó.
5. Nếu không tìm thấy tiến trình nào trong lần duyệt đầu tiên, hàm sẽ đặt lại số lượng slot của tất cả các hàng đợi về giá trị ban đầu của chúng (`MAX_PRIO - i`) và thử lại. Điều này đảm bảo rằng các hàng đợi có độ ưu tiên thấp hơn cuối cùng cũng sẽ có cơ hội chạy, ngay cả khi có rất nhiều tiến trình có độ ưu tiên cao hơn đang chờ xử lý.
6. Hàm mở khóa `queue_lock` và trả về tiến trình.

- **Mục đích:** Chọn tiến trình tiếp theo để chạy dựa trên độ ưu tiên và chính sách chia sẻ thời gian theo độ ưu tiên.

`put_mlq_proc` (Khi `MLQ_SCHED` được định nghĩa)

- **Chức năng:** Thêm một tiến trình trở lại hàng đợi MLQ.
- **Hoạt động:**
  1. Hàm khóa `queue_lock` để bảo vệ các hàng đợi.
  2. Hàm thêm tiến trình vào hàng đợi ưu tiên tương ứng với độ ưu tiên của tiến trình bằng cách sử dụng hàm `enqueue`.
  3. Hàm mở khóa `queue_lock`.
- **Mục đích:** Đặt một tiến trình trở lại hàng đợi sẵn sàng sau khi nó đã sử dụng hết lượng thời gian của mình hoặc bị chặn vì một số lý do.

`add_mlq_proc` (Khi `MLQ_SCHED` được định nghĩa)

- **Chức năng:** Tương tự như `put_mlq_proc`, nhưng có thể được sử dụng để thêm một tiến trình mới vào hàng đợi MLQ lần đầu tiên.
- **Hoạt động:** Tương tự như `put_mlq_proc`.
- **Mục đích:** Thêm một tiến trình mới vào hàng đợi sẵn sàng.

`get_proc`

- **Chức năng:** Lấy một tiến trình để chạy.
- **Hoạt động:**
  - Nếu `MLQ_SCHED` được định nghĩa, hàm sẽ gọi `get_mlq_proc` để lấy một tiến trình từ hàng đợi MLQ.
  - Nếu `MLQ_SCHED` không được định nghĩa, hàm sẽ khóa `queue_lock`, lấy một tiến trình từ `ready_queue` bằng cách sử dụng hàm `dequeue`, mở khóa `queue_lock` và trả về tiến trình đó.
- **Mục đích:** Chọn tiến trình tiếp theo để chạy, sử dụng MLQ hoặc FCFS/Priority Scheduling tùy thuộc vào cấu hình.

put_proc

- **Chức năng:** Đặt một tiến trình trở lại hàng đợi.
- **Hoạt động:**
  - Hàm khóa `queue_lock`, thêm tiến trình vào `running_list`, và mở khóa `queue_lock`.
  - Nếu `MLQ_SCHED` được định nghĩa, hàm sẽ gọi `put_mlq_proc` để đặt tiến trình trở lại hàng đợi MLQ.
  - Nếu `MLQ_SCHED` không được định nghĩa, hàm sẽ khóa `queue_lock`, thêm tiến trình vào `run_queue`, và mở khóa `queue_lock`.
- **Mục đích:** Đặt một tiến trình trở lại hàng đợi sẵn sàng sau khi nó đã sử dụng hết lượng thời gian của mình hoặc bị chặn.

add_proc

- **Chức năng:** Thêm một tiến trình mới vào hệ thống.
- **Hoạt động:**
  - Hàm khóa `queue_lock`, thêm tiến trình vào `running_list`, và mở khóa `queue_lock`.
  - Nếu `MLQ_SCHED` được định nghĩa, hàm sẽ gọi `add_mlq_proc` để thêm tiến trình vào hàng đợi MLQ.
  - Nếu `MLQ_SCHED` không được định nghĩa, hàm sẽ khóa `queue_lock`, thêm tiến trình vào `ready_queue`, và mở khóa `queue_lock`.
- **Mục đích:** Thêm một tiến trình mới vào hệ thống và đặt nó vào hàng đợi sẵn sàng.

2. Quản lý bộ nhớ

2.1. Cấu trúc bộ nhớ ảo của tiến trình

Mỗi tiến trình trong hệ điều hành mô phỏng có một không gian bộ nhớ ảo riêng biệt. Không gian này được quản lý bởi các cấu trúc sau:

- `vm_area_struct`: đại diện cho một vùng bộ nhớ ảo liên tục trong tiến trình. Gồm các trường:
  - `vm_start`, `vm_end`: địa chỉ bắt đầu và kết thúc của vùng.
  - `sbrk`: con trỏ trỏ tới ranh giới hiện tại của heap, dùng để mở rộng vùng nhớ nếu cần.
  - `vm_freerg_list`: danh sách các vùng trống hiện có, hỗ trợ cấp phát lại sau khi giải phóng.
- `vm_rg_struct`: mô tả từng vùng nhỏ (region) trong vm_area, tương ứng với một biến/đối tượng được cấp phát.
- `mm_struct`: quản lý tổng thể bộ nhớ của tiến trình, bao gồm bảng trang, vùng nhớ và danh sách region.

2.2. Cơ chế cấp phát bộ nhớ (ALLOC)

Khi một tiến trình thực hiện lệnh:

```
alloc 128 2
```

Nghĩa là hệ điều hành cần cấp phát 128 byte và lưu địa chỉ kết quả vào thanh ghi số 2.
Hệ thống thực hiện như sau:

1. Duyệt danh sách `vm_freerg_list` để tìm một vùng trống có đủ kích thước.
2. Nếu không tìm thấy, tăng con trỏ `sbrk` để tạo thêm không gian mới trong heap.
3. Tạo một vùng `vm_rg_struct` mới cho vùng vừa cấp phát.
4. Ghi địa chỉ bắt đầu vào thanh ghi 2.
5. Cập nhật bảng trang (page table) nếu vùng nhớ chưa được ánh xạ.

2.3. Cơ chế giải phóng bộ nhớ (FREE)

Với lệnh:

```
free 2
```

Hệ thống sẽ thực hiện:

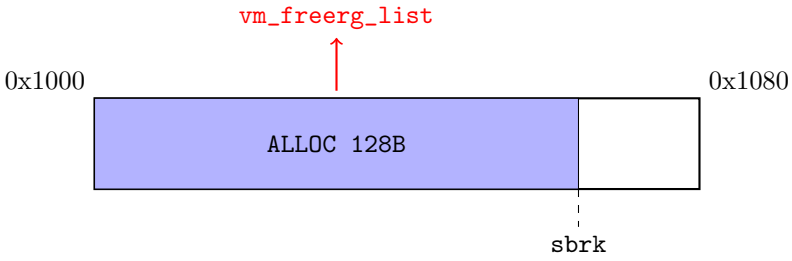
1. Lấy địa chỉ từ thanh ghi 2.
2. Tìm vùng tương ứng trong danh sách region của tiến trình.
3. Đánh dấu vùng đó là "free" và đưa vào danh sách `vm_freerg_list`.
4. Không giải phóng khung vật lý ngay (tránh tạo memory hole), mà giữ để tái sử dụng.

2.4. Ví dụ minh họa cụ thể

Giả sử vùng heap của tiến trình bắt đầu từ địa chỉ 0x1000 và có `sbrk = 0x1000`.

- Lệnh `alloc 128 2`:
  - Không có vùng trống → hệ thống nâng `sbrk` lên 0x1080.
  - Ghi địa chỉ `0x1000` vào thanh ghi 2.
  - Vùng cấp phát: [0x1000, 0x107F].
- Lệnh `free 2`:
  - Đánh dấu vùng [0x1000, 0x107F] là vùng trống.
  - Đưa vào `vm_freerg_list`.
  - `sbrk` không thay đổi, vùng này có thể được cấp phát lại về sau.

2.5. Sơ đồ minh họa vùng nhớ



2.6. Code minh họa cho Memory

2.6.1. File libmem.c

`libmem.c` mô phỏng các lời gọi hệ thống liên quan đến cấp phát bộ nhớ động trong tiến trình người dùng như `malloc()` và `free()`. Module này hoạt động trên các vùng nhớ ảo đã được quản lý bởi `mm.c`, và cung cấp API thân thiện cho tiến trình thực thi trong môi trường mô phỏng hệ điều hành.

Listing 3: libmem.c

```
#include "string.h"
#include "mm.h"
#include "syscall.h"
#include "libmem.h"
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include "mem.h"

static pthread_mutex_t mmvm_lock = PTHREAD_MUTEX_INITIALIZER;

void dump_byte(struct memphy_struct *mp, int addr, BYTE data)
{
    printf("BYTE %08X: %d\n", addr, data);
}

int vm_map_page_range(struct pcb_t *caller, int vstart, int vend, int pstart, int num_pages
    ) {
    int vaddr = vstart;
    int paddr = pstart;

    for (int i = 0; i < num_pages; i++) {
        int pgn = PAGING_PGN(vaddr);
        int fpn;

        if (MEMPHY_get_freefp(caller->mram, &fpn) != 0) {
            printf("vm_map_page_range: No free frame for page %d\n", pgn);
```

```
            return -1;
        }

        pte_set_fpn(&caller->mm->pgd[pgn], fpn);

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

        vaddr += PAGING_PAGESZ;
        paddr += PAGING_PAGESZ;
    }

    return 0;
}

int enlist_vm_freerg_list(struct mm_struct *mm, struct vm_rg_struct *rg_elmt)
{
    struct vm_rg_struct *rg_node = mm->mmap->vm_freerg_list;

    if (rg_elmt->rg_start >= rg_elmt->rg_end)
        return -1;

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

    mm->mmap->vm_freerg_list = rg_elmt;

    return 0;
}

struct vm_rg_struct *get_symrg_byid(struct mm_struct *mm, int rgid)
{
    if (rgid < 0 || rgid >= PAGING_MAX_SYMTBL_SZ)
        return NULL;

    return &mm->symrgtbl[rgid];
}

int __alloc(struct pcb_t *caller, int vmaid, int rgid, int size, int *alloc_addr)
{
    pthread_mutex_lock(&mmvm_lock);

    struct vm_rg_struct rgnode;

    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);

    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;
        int sz = rgnode.rg_end - rgnode.rg_start;
        int num_pages = PAGING_PAGE_ALIGNSZ(sz) / PAGING_PAGESZ;
        vm_map_page_range(caller, rgnode.rg_start, rgnode.rg_end, rgnode.rg_start,
            num_pages);

        *alloc_addr = rgnode.rg_start;
        pthread_mutex_unlock(&mmvm_lock);
        return 0;
    }

    if (!cur_vma) {
        pthread_mutex_unlock(&mmvm_lock);
        return -1;
    }

    int inc_sz = PAGING_PAGE_ALIGNSZ(size);

    if (inc_vma_limit(caller, vmaid, inc_sz) == -1)
    {
        pthread_mutex_unlock(&mmvm_lock);
        return -1;
    }

```

```
98      if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
99      {
100         caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
101         caller->mm->symrgtbl[rgid].rg_end   = rgnode.rg_end;
102         int sz = rgnode.rg_end - rgnode.rg_start;
103         int num_pages = PAGING_PAGE_ALIGNSZ(sz) / PAGING_PAGESZ;
104         vm_map_page_range(caller, rgnode.rg_start, rgnode.rg_end, rgnode.rg_start,
                num_pages);
105
106         *alloc_addr = rgnode.rg_start;
107         pthread_mutex_unlock(&mmvm_lock);
108         return 0;
109     }
110
111     int old_sbrk = cur_vma->sbrk;
112     int new_sbrk = old_sbrk + inc_sz;
113     struct sc_regs regs;
114     regs.a1 = SYSMEM_INC_OP;
115     regs.a2 = old_sbrk;
116     regs.a3 = new_sbrk;
117
118     syscall(caller, 17,  s );
119
120     cur_vma->sbrk = new_sbrk;
121     cur_vma->vm_end = new_sbrk;
122
123     cur_vma->vm_freerg_list = init_vm_rg(old_sbrk, new_sbrk);
124
125     if (get_free_vmrg_area(caller, vmaid, size, &rgnode) == 0)
126     {
127         caller->mm->symrgtbl[rgid].rg_start = rgnode.rg_start;
128         caller->mm->symrgtbl[rgid].rg_end   = rgnode.rg_end;
129
130         int sz = rgnode.rg_end - rgnode.rg_start;
131         int num_pages = PAGING_PAGE_ALIGNSZ(sz) / PAGING_PAGESZ;
132         vm_map_page_range(caller, rgnode.rg_start, rgnode.rg_end, rgnode.rg_start,
                num_pages);
133
134         *alloc_addr = rgnode.rg_start;
135         pthread_mutex_unlock(&mmvm_lock);
136         return 0;
137     }
138
139     pthread_mutex_unlock(&mmvm_lock);
140     return -1;
141 }
142
143 int __free(struct pcb_t *caller, int vmaid, int rgid) {
144     struct vm_rg_struct *rgnode =
145         (struct vm_rg_struct *)malloc(sizeof(struct vm_rg_struct));
146
147     if (rgid < 0 || rgid > PAGING_MAX_SYMTBL_SZ)
148       return -1;
149
150     pthread_mutex_lock(&mmvm_lock);
151
152     rgnode->rg_start = caller->mm->symrgtbl[rgid].rg_start;
153     rgnode->rg_end = caller->mm->symrgtbl[rgid].rg_end;
154     caller->mm->symrgtbl[rgid].rg_start = caller->mm->symrgtbl[rgid].rg_end;
155
156     enlist_vm_freerg_list(caller->mm, rgnode);
157
158     pthread_mutex_unlock(&mmvm_lock);
159
160     return 0;
161   }
162
163 int liballoc(struct pcb_t *proc, uint32_t size, uint32_t reg_index)
164 {
165     if (reg_index >= PAGING_MAX_SYMTBL_SZ) {
166         return -1;
167     }
```

```
168
169     int addr;
170     int ret = __alloc(proc, 0, reg_index, size, &addr);
171     if (ret != 0) {
172         printf("Allocation failed for PID=%d region=%u size=%u\n",
173                 proc->pid, reg_index, size);
174         return -1;
175     }
176
177 #ifdef IODUMP
178     printf("===== PHYSICAL MEMORY AFTER ALLOCATION =====\n");
179     printf("PID=%d - Region=%u - Address=%08x - Size=%u byte\n",
180             proc->pid, reg_index, addr, size);
181     print_pgtbl(proc, 0, -1);
182     printf("================================================================\n");
183 #endif
184
185     return 0;
186 }
187
188 int libfree(struct pcb_t *proc, uint32_t reg_index)
189 {
190     if (reg_index >= PAGING_MAX_SYMTBL_SZ) {
191         return -1;
192     }
193
194     int ret = __free(proc, 0, reg_index);
195     if (ret != 0) {
196         printf("Freeing region %u failed for PID=%d\n", reg_index, proc->pid);
197         return -1;
198     }
199
200 #ifdef IODUMP
201     printf("===== PHYSICAL MEMORY AFTER DEALLOCATION =====\n");
202     printf("PID=%d - Region=%u\n", proc->pid, reg_index);
203
204     print_pgtbl(proc, 0, -1);
205
206     printf("================================================================\n");
207 #endif
208
209     return 0;
210 }
211
212 int pg_getpage(struct mm_struct *mm, int pgn, int *fpn, struct pcb_t *caller) {
213     uint32_t pte = mm->pgd[pgn];
214
215     if (!PAGING_PAGE_PRESENT(pte)) {
216         int vicpgn, swpfpn;
217         uint32_t vicpte;
218
219         find_victim_page(caller->mm, &vicpgn);
220         vicpte = mm->pgd[vicpgn];
221         int vicfpn = PAGING_FPN(vicpte);
222
223         if (MEMPHY_get_freefp(caller->active_mswp, &swpfpn) != 0) {
224             printf("ERROR: No free frame available in swap memory\n");
225             return -1;
226         }
227
228         struct sc_regs regs_out;
229         regs_out.a1 = SYSMEM_SWP_OP;
230         regs_out.a2 = vicpgn;
231         regs_out.a3 = swpfpn;
232         if (syscall(caller, 17,  s_out ) < 0) {
233             printf("ERROR: Swap-out syscall failed\n");
234             return -1;
235         }
236
237         int tgtfpn = PAGING_PTE_SWP(pte);
238         struct sc_regs regs_in;
239         regs_in.a1 = SYSMEM_SWP_OP;
```

```
240         regs_in.a2 = tgtfpn;
241         regs_in.a3 = vicfpn;
242         if (syscall(caller, 17,  s_in ) < 0) {
243             printf("ERROR: Swap-in syscall failed\n");
244             return -1;
245         }
246 
247         pte_set_swap(&mm->pgd[vicpgn], 0, swpfpn);
248         pte_set_fpn(&mm->pgd[pgn], vicfpn);
249         enlist_pgn_node(&caller->mm->fifo_pgn, pgn);
250     }
251 
252     *fpn = PAGING_FPN(mm->pgd[pgn]);
253     return 0;
254 }
255 
256 int pg_getval(struct mm_struct *mm, int addr, BYTE *data, struct pcb_t *caller)
257 {
258     int pgn = PAGING_PGN(addr);
259     int off = PAGING_OFFST(addr);
260     int fpn;
261 
262     if (pg_getpage(mm, pgn, &fpn, caller) != 0)
263         return -1;
264 
265     int phyaddr = fpn * PAGING_PAGESZ + off;
266 
267     BYTE tmp;
268     if (MEMPHY_read(caller->mram, phyaddr, &tmp) == 0) {
269         *data = tmp;
270         return 0;
271     } else {
272         return -1;
273     }
274 }
275 
276 int pg_setval(struct mm_struct *mm, int addr, BYTE value, struct pcb_t *caller)
277 {
278     int pgn = PAGING_PGN(addr);
279     int off = PAGING_OFFST(addr);
280     int fpn;
281 
282     if (pg_getpage(mm, pgn, &fpn, caller) != 0)
283         return -1;
284 
285     int phyaddr = fpn * PAGING_PAGESZ + off;
286 
287     MEMPHY_write(caller->mram, phyaddr, value);
288     return 0;
289 }
290 
291 
292 int __read(struct pcb_t *caller, int vmaid, int rgid, int offset, BYTE *data) {
293   pthread_mutex_lock(&mmvm_lock);
294   struct vm_rg_struct *currg = get_symrg_byid(caller->mm, rgid);
295   struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
296 
297   if (currg == NULL || cur_vma == NULL)
298     return -1;
299 
300   pg_getval(caller->mm, currg->rg_start + offset, data, caller);
301   pthread_mutex_unlock(&mmvm_lock);
302 
303   return 0;
304 }
305 
306 int libread(struct pcb_t *proc, uint32_t source, uint32_t offset, uint32_t* destination)
307 {
308     BYTE data;
309     int ret = __read(proc, 0, source, offset, &data);
310     *destination = (uint32_t)data;
311 
```

```
#ifdef IODUMP
    int effective_addr = proc->mm->symrgtbl[source].rg_start + offset;
    int pgn = PAGING_PGN(effective_addr);
    int off = PAGING_OFFST(effective_addr);
    int fpn = PAGING_FPN(proc->mm->pgd[pgn]);
    int pa = fpn * PAGING_PAGESZ + off;

    printf("===== PHYSICAL MEMORY AFTER READING =====\n");
    printf("read region=%d offset=%d value=%d\n", source, offset, data);
#ifdef PAGETBL_DUMP
    print_pgtbl(proc, 0, -1);
    printf("================================================================\n");
    printf("===== PHYSICAL MEMORY DUMP =====\n");
#endif
    dump_byte(proc->mram, pa, data);
    printf("===== PHYSICAL MEMORY END-DUMP =====\n");
    printf("================================================================\n");
#endif


    return ret;
}

int __write(struct pcb_t *caller, int vmaid, int rgid, int offset, BYTE value) {

  pthread_mutex_lock(&mmvm_lock);
  struct vm_rg_struct *currg = get_symrg_byid(caller->mm, rgid);
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);

  if (currg == NULL || cur_vma == NULL)
    return -1;

  pg_setval(caller->mm, currg->rg_start + offset, value, caller);
  pthread_mutex_unlock(&mmvm_lock);
  return 0;
}
int libwrite(struct pcb_t *proc, BYTE data, uint32_t destination, uint32_t offset)
{
#ifdef IODUMP
    int effective_addr = proc->mm->symrgtbl[destination].rg_start + offset;
    int pgn = PAGING_PGN(effective_addr);
    int off = PAGING_OFFST(effective_addr);
    int fpn = PAGING_FPN(proc->mm->pgd[pgn]);
    int pa = fpn * PAGING_PAGESZ + off;

    printf("===== PHYSICAL MEMORY AFTER WRITING =====\n");
    printf("write region=%d offset=%d value=%d\n", destination, offset, data);
#ifdef PAGETBL_DUMP
    print_pgtbl(proc, 0, -1);
    printf("================================================================\n");
    printf("===== PHYSICAL MEMORY DUMP =====\n");
#endif
    dump_byte(proc->mram, pa, data);
    printf("===== PHYSICAL MEMORY END-DUMP =====\n");
    printf("================================================================\n");
#endif

    return __write(proc, 0, destination, offset, data);
}


int free_pcb_memph(struct pcb_t *caller)
{
    int pagenum, fpn;
    uint32_t pte;

    for(pagenum = 0; pagenum < PAGING_MAX_PGN; pagenum++)
    {
        pte = caller->mm->pgd[pagenum];

        if (!PAGING_PAGE_PRESENT(pte))
        {
```

```
            fpn = PAGING_PTE_FPN(pte);
            MEMPHY_put_freefp(caller->mram, fpn);
        } else {
            fpn = PAGING_PTE_SWP(pte);
            MEMPHY_put_freefp(caller->active_mswp, fpn);
        }
    }

    return 0;
}

int find_victim_page(struct mm_struct *mm, int *retpgn)
{
    if (mm == NULL || retpgn == NULL)
        return -1;

    struct pgn_t *head = mm->fifo_pgn;
    if (head == NULL)
        return -1;

    *retpgn        = head->pgn;
    mm->fifo_pgn   = head->pg_next;

    free(head);

    return 0;
}

int get_free_vmrg_area(struct pcb_t *caller, int vmaid, int size,
                       struct vm_rg_struct *newrg) {
  struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);

  struct vm_rg_struct *rgit = cur_vma->vm_freerg_list;

  if (rgit == NULL)
    return -1;

  newrg->rg_start = newrg->rg_end = -1;

  while (rgit != NULL) {
    if (rgit->rg_start + size <=
        rgit->rg_end) {
      newrg->rg_start = rgit->rg_start;
      newrg->rg_end = rgit->rg_start + size;

      if (rgit->rg_start + size < rgit->rg_end) {
        rgit->rg_start = rgit->rg_start + size;
      } else {
        struct vm_rg_struct *nextrg = rgit->rg_next;

        if (nextrg != NULL) {
          rgit->rg_start = nextrg->rg_start;
          rgit->rg_end = nextrg->rg_end;

          rgit->rg_next = nextrg->rg_next;

          free(nextrg);
        } else {
          rgit->rg_start = rgit->rg_end;
          rgit->rg_next = NULL;
        }
      }
      break;
    } else {
      rgit = rgit->rg_next;
    }
  }
  if (newrg->rg_start == -1)
    return -1;

  return 0;
}
```

Giải thích các hàm

vm_map_page_range

- **Chức năng:** Ánh xạ một dải các trang ảo tới các trang vật lý.
- **Tham số:**
  - `caller`: PCB của tiến trình gọi.
  - `vstart`: Địa chỉ ảo bắt đầu.
  - `vend`: Địa chỉ ảo kết thúc.
  - `pstart`: Địa chỉ vật lý bắt đầu (thực tế không được sử dụng, mà cấp phát động).
  - `num_pages`: Số lượng trang cần ánh xạ.
- **Hoạt động:**
  1. Duyệt qua từng trang trong dải địa chỉ ảo.
  2. Lấy số trang (PGN) từ địa chỉ ảo hiện tại.
  3. Lấy một frame vật lý trống (FPN) từ RAM. Nếu không có frame trống, hàm in ra thông báo lỗi và trả về -1.
  4. Cập nhật bảng trang (page table) của tiến trình để ánh xạ PGN tới FPN. Hàm `pte_set_fpn` được sử dụng để đặt FPN trong entry của bảng trang.
  5. Thêm PGN vào danh sách FIFO (First-In, First-Out) của tiến trình để theo dõi thứ tự sử dụng trang.
  6. Tăng địa chỉ ảo và địa chỉ vật lý lên kích thước trang tiếp theo.
- **Mục đích:** Thiết lập ánh xạ giữa bộ nhớ ảo và bộ nhớ vật lý.

enlist_vm_freerg_list

- **Chức năng:** Thêm một vùng bộ nhớ ảo tự do vào danh sách các vùng tự do của một `mm_struct`.
- **Tham số:**
  - `mm`: Memory module (cấu trúc quản lý bộ nhớ của tiến trình).
  - `rg_elmt`: Vùng bộ nhớ ảo tự do cần thêm.
- **Hoạt động:**
  1. Kiểm tra tính hợp lệ của vùng (start phải nhỏ hơn end).
  2. Thêm `rg_elmt` vào đầu danh sách `vm_freerg_list` của `mm`.
- **Mục đích:** Quản lý danh sách các vùng bộ nhớ ảo tự do, để có thể cấp phát khi cần.

get_symrg_byid

- **Chức năng:** Lấy một vùng bộ nhớ ảo (region) từ bảng ký hiệu (symbol table) dựa trên ID vùng.
- **Tham số:**
  - `mm`: Memory module.
  - `rgid`: ID của vùng cần tìm.
- **Hoạt động:**
  1. Kiểm tra tính hợp lệ của `rgid`.
  2. Trả về con trỏ tới vùng có ID tương ứng trong bảng `symrgtbl`.
- **Mục đích:** Truy cập nhanh vào một vùng bộ nhớ đã được cấp phát trước đó.

__alloc

- **Chức năng:** Cấp phát một vùng bộ nhớ ảo. Đây là hàm lõi để cấp phát bộ nhớ.
- **Tham số:**
  - `caller`: PCB của tiến trình gọi.
  - `vmaid`: ID của vùng ảo (Virtual Memory Area - VMA).
  - `rgid`: ID của vùng bộ nhớ (region ID), sử dụng làm chỉ số trong bảng ký hiệu (symbol table).
  - `size`: Kích thước cần cấp phát.
  - `alloc_addr`: Con trỏ để trả về địa chỉ ảo đã cấp phát.
- **Hoạt động:**
  1. Khóa mutex `mmvm_lock` để đảm bảo an toàn luồng.
  2. Lấy VMA từ `caller->mm` dựa trên `vmaid`.
  3. Gọi `get_free_vmrg_area` để tìm một vùng tự do đủ lớn trong VMA.
  4. Nếu tìm thấy vùng tự do:
     (a) Cập nhật thông tin vùng trong bảng ký hiệu (`caller->mm->symrgtbl[rgid]`).
     (b) Tính số trang cần thiết và ánh xạ dải trang ảo này tới các trang vật lý bằng `vm_map_page_range`.
     (c) Trả về địa chỉ ảo đã cấp phát qua `alloc_addr`.
     (d) Mở khóa mutex và trả về 0.
  5. Nếu không tìm thấy vùng tự do, hàm sẽ cố gắng tăng giới hạn của VMA bằng cách gọi `inc_vma_limit`.
  6. Nếu việc tăng giới hạn VMA thành công, hàm sẽ thử lại `get_free_vmrg_area`.
  7. Nếu vẫn không thành công, hàm sẽ gọi syscall để tăng giới hạn bộ nhớ ảo của tiến trình bằng `SYSMEM_INC_OP`.
  8. Sau khi tăng giới hạn bằng syscall, hàm sẽ thử lại `get_free_vmrg_area` lần cuối.
  9. Nếu tất cả các lần thử đều thất bại, hàm sẽ mở khóa mutex và trả về -1.
- **Mục đích:** Cấp phát bộ nhớ ảo cho tiến trình, sử dụng các vùng tự do hiện có hoặc tăng giới hạn VMA nếu cần.

__free

- **Chức năng:** Giải phóng một vùng bộ nhớ ảo đã được cấp phát.
- **Tham số:**
  - `caller`: PCB của tiến trình gọi.
  - `vmaid`: ID của VMA.
  - `rgid`: ID của vùng cần giải phóng trong bảng ký hiệu.
- **Hoạt động:**
  1. Khóa mutex `mmvm_lock`.
  2. Lấy thông tin vùng từ bảng ký hiệu dựa trên `rgid`.
  3. Cập nhật bảng ký hiệu để đánh dấu vùng là không còn sử dụng (thường bằng cách đặt `rg_start` = `rg_end`).
  4. Thêm vùng đã giải phóng vào danh sách các vùng tự do (free list) của VMA bằng cách gọi `enlist_vm_freerg_list`.
  5. Mở khóa mutex.
- **Mục đích:** Giải phóng bộ nhớ ảo, cho phép nó được cấp phát lại sau này.

liballoc

- **Chức năng:** Hàm giao diện (wrapper) cho `__alloc`. Cấp phát vùng nhớ dựa trên cơ chế paging, thường được gọi từ các thư viện hoặc ứng dụng.
- **Tham số:**
  - `proc`: Tiến trình gọi.
  - `size`: Kích thước vùng nhớ cần cấp phát.
  - `reg_index`: Region ID trong symbol table.
- **Hoạt động:**
  1. Kiểm tra tính hợp lệ của `reg_index`.
  2. Gọi `__alloc` để thực hiện cấp phát thực tế.
  3. In thông tin gỡ lỗi (nếu `IODUMP` được định nghĩa).
- **Mục đích:** Cung cấp một giao diện đơn giản để cấp phát bộ nhớ cho các ứng dụng.

libfree

- **Chức năng:** Hàm giao diện (wrapper) cho `__free`. Giải phóng vùng nhớ đã cấp phát, thường được gọi từ các thư viện hoặc ứng dụng.
- **Tham số:**
  - `proc`: Tiến trình gọi.
  - `reg_index`: Region ID trong symbol table.
- **Hoạt động:**
  1. Kiểm tra tính hợp lệ của `reg_index`.
  2. Gọi `__free` để thực hiện giải phóng thực tế.
  3. In thông tin gỡ lỗi (nếu `IODUMP` được định nghĩa).
- **Mục đích:** Cung cấp một giao diện đơn giản để giải phóng bộ nhớ cho các ứng dụng.

pg_getpage

- **Chức năng:** Lấy một trang từ RAM. Nếu trang không có trong RAM, hàm sẽ thực hiện swap.
- **Tham số:**
  - `mm`: Memory module.
  - `pgn`: Page number.
  - `fpn`: Con trỏ để trả về frame number.
  - `caller`: Tiến trình gọi.
- **Hoạt động:**
  1. Lấy PTE (Page Table Entry) từ bảng trang.
  2. Kiểm tra xem trang có hiện diện trong RAM không (`PAGING_PAGE_PRESENT`).
  3. Nếu trang không có trong RAM (page fault):
     (a) Tìm một trang "nạn nhân"để swap ra (sử dụng `find_victim_page`).
     (b) Lấy một frame trống trong swap.
     (c) Thực hiện syscall để swap trang nạn nhân ra swap và trang hiện tại vào RAM.
     (d) Cập nhật PTE của trang nạn nhân và trang hiện tại.
  4. Trả về FPN của trang (đã có trong RAM hoặc vừa được swap vào).
- **Mục đích:** Đảm bảo rằng trang cần thiết có mặt trong RAM để truy cập.

pg_getval

- **Chức năng:** Đọc một byte từ một địa chỉ ảo.
- **Tham số:**
  - `mm`: Memory module.
  - `addr`: Địa chỉ ảo cần đọc.
  - `data`: Con trỏ để trả về giá trị đọc được.
  - `caller`: Tiến trình gọi.
- **Hoạt động:**
  1. Tính PGN và offset từ địa chỉ ảo.
  2. Gọi `pg_getpage` để đảm bảo trang có trong RAM và lấy FPN.
  3. Tính địa chỉ vật lý từ FPN và offset.
  4. Đọc byte từ địa chỉ vật lý bằng `MEMPHY_read`.
  5. Trả về giá trị đọc được qua `data`.
- **Mục đích:** Đọc dữ liệu từ bộ nhớ ảo.

pg_setval

- **Chức năng:** Ghi một byte vào một địa chỉ ảo.
- **Tham số:**
  - `mm`: Memory module.
  - `addr`: Địa chỉ ảo cần ghi.
  - `value`: Giá trị cần ghi.
  - `caller`: Tiến trình gọi.
- **Hoạt động:**
  1. Tính PGN và offset từ địa chỉ ảo.
  2. Gọi `pg_getpage` để đảm bảo trang có trong RAM và lấy FPN.
  3. Tính địa chỉ vật lý từ FPN và offset.
  4. Ghi byte vào địa chỉ vật lý bằng `MEMPHY_write`.
- **Mục đích:** Ghi dữ liệu vào bộ nhớ ảo.

__read

- **Chức năng:** Đọc một byte từ một vùng nhớ ảo đã được cấp phát.
- **Tham số:**
  - `caller`: PCB của tiến trình gọi.
  - `vmaid`: ID của VMA.
  - `rgid`: ID của vùng trong bảng ký hiệu.
  - `offset`: Offset trong vùng đã cấp phát.
  - `data`: Con trỏ để trả về giá trị đọc được.
- **Hoạt động:**
  1. Lấy vùng nhớ ảo từ bảng ký hiệu bằng `get_symrg_byid`.
  2. Gọi `pg_getval` để đọc byte từ địa chỉ ảo (base + offset).
- **Mục đích:** Thực hiện đọc từ vùng nhớ đã cấp phát, đảm bảo tính hợp lệ của vùng.

__write

- **Chức năng:** Ghi một byte vào một vùng nhớ ảo đã được cấp phát.
- **Tham số:**
  - `caller`: PCB của tiến trình gọi.
  - `vmaid`: ID của VMA.
  - `rgid`: ID của vùng trong bảng ký hiệu.
  - `offset`: Offset trong vùng đã cấp phát.
  - `value`: Giá trị cần ghi.
- **Hoạt động:**
  1. Lấy vùng nhớ ảo từ bảng ký hiệu bằng `get_symrg_byid`.
  2. Gọi `pg_setval` để ghi byte vào địa chỉ ảo (base + offset).
- **Mục đích:** Thực hiện ghi vào vùng nhớ đã cấp phát, đảm bảo tính hợp lệ của vùng.

find_victim_page

- **Chức năng:** Tìm một trang "nạn nhân"để swap ra khỏi RAM (sử dụng thuật toán FIFO).
- **Tham số:**
  - `mm`: Memory module.
  - `retpgn`: Con trỏ để trả về PGN của trang nạn nhân.
- **Hoạt động:**
  1. Lấy phần tử đầu tiên từ danh sách FIFO `fifo_pgn`.
  2. Trả về PGN của trang đầu tiên (trang lâu đời nhất).
  3. Xóa trang đầu tiên khỏi danh sách FIFO.
- **Mục đích:** Chọn trang để swap ra khi cần thêm không gian trong RAM.

get_free_vmrg_area

- **Chức năng:** Tìm một vùng trống có kích thước đủ lớn trong danh sách các vùng tự do của một VMA.
- **Tham số:**
  - `caller`: PCB của tiến trình gọi.
  - `vmaid`: ID của VMA.
  - `size`: Kích thước cần cấp phát.
  - `newrg`: Con trỏ để trả về vùng mới (nếu tìm thấy).
- **Hoạt động:**
  1. Duyệt qua danh sách các vùng tự do `vm_freerg_list` của VMA.
  2. Nếu tìm thấy một vùng có kích thước đủ lớn:
     (a) Cập nhật thông tin của vùng tự do (chia nhỏ vùng nếu cần).
     (b) Trả về thông tin của vùng mới qua `newrg`.
  3. Nếu không tìm thấy vùng nào đủ lớn, trả về -1.
- **Mục đích:** Tìm một vùng nhớ tự do để cấp phát cho tiến trình.

2.6.2. File mm.c

mm.c là phần trung gian giữa quản lý bộ nhớ ảo và bộ nhớ vật lý. Nó xử lý quá trình ánh xạ từ địa chỉ ảo sang địa chỉ vật lý, cấp phát bộ nhớ cho tiến trình, và tương tác với các thành phần như mm-vm.c và mm-memphy.c. Đây là nơi hiện thực các hàm quản lý trang (paging), cấp phát/trả trang, và ánh xạ vùng nhớ ảo sang RAM.

Listing 4: mm.c

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

int init_pte(uint32_t *pte,
             int pre,
             int fpn,
             int drt,
             int swp,
             int swptyp,
             int swpoff)
{
    if (pre != 0) {
        if (swp == 0) {
            if (fpn == 0)
                return -1;

            SETBIT(*pte, PAGING_PTE_PRESENT_MASK);
            CLRBIT(*pte, PAGING_PTE_SWAPPED_MASK);
            CLRBIT(*pte, PAGING_PTE_DIRTY_MASK);

            SETVAL(*pte, fpn, PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);
        }
        else
        {
            SETBIT(*pte, PAGING_PTE_PRESENT_MASK);
            SETBIT(*pte, PAGING_PTE_SWAPPED_MASK);
            CLRBIT(*pte, PAGING_PTE_DIRTY_MASK);

            SETVAL(*pte, swptyp, PAGING_PTE_SWPTYP_MASK, PAGING_PTE_SWPTYP_LOBIT);
            SETVAL(*pte, swpoff, PAGING_PTE_SWPOFF_MASK, PAGING_PTE_SWPOFF_LOBIT);
        }
    }

    return 0;
}

int pte_set_swap(uint32_t *pte, int swptyp, int swpoff)
{
    SETBIT(*pte, PAGING_PTE_PRESENT_MASK);
    SETBIT(*pte, PAGING_PTE_SWAPPED_MASK);

    SETVAL(*pte, swptyp, PAGING_PTE_SWPTYP_MASK, PAGING_PTE_SWPTYP_LOBIT);
    SETVAL(*pte, swpoff, PAGING_PTE_SWPOFF_MASK, PAGING_PTE_SWPOFF_LOBIT);

    return 0;
}

int pte_set_fpn(uint32_t *pte, int fpn)
{
   *pte = 0;
    SETBIT(*pte, PAGING_PTE_PRESENT_MASK);
    CLRBIT(*pte, PAGING_PTE_SWAPPED_MASK);

    SETVAL(*pte, fpn, PAGING_PTE_FPN_MASK, PAGING_PTE_FPN_LOBIT);

    return 0;
}

int vmap_page_range(struct pcb_t *caller,
                    int addr,
                    int pgnum,
                    struct framephy_struct *frames,
```

```
                    struct vm_rg_struct *ret_rg)
{
    int pgit = 0;
    int pgn = PAGING_PGN(addr);

    ret_rg->rg_start = addr;
    ret_rg->rg_end = addr + pgnum * PAGE_SIZE;
    ret_rg->vmaid = 0;

    struct framephy_struct *fpit = frames;
    for (pgit = 0; pgit < pgnum && fpit != NULL; pgit++)
    {
        uint32_t *pte = &caller->mm->pgd[pgn + pgit];
        *pte = (fpit->fpn << 0) |
               PAGING_PTE_PRESENT_MASK;
        fpit = fpit->fp_next;
    }

    for (pgit = 0; pgit < pgnum; pgit++)
    {
        enlist_pgn_node(&caller->mm->fifo_pgn, pgn + pgit);
    }

    return 0;
}

#define ERROR_INSUFFICIENT_FRAMES -3000

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;
    struct framephy_struct *current = NULL;
    *frm_lst = NULL;

    for (pgit = 0; pgit < req_pgnum; pgit++)
    {
        newfp_str = (struct framephy_struct *)malloc(sizeof(struct framephy_struct));
        if (newfp_str == NULL)
        {
            while (*frm_lst != NULL)
            {
                struct framephy_struct *temp = *frm_lst;
                *frm_lst = (*frm_lst)->fp_next;
                free(temp);
            }
            return ERROR_INSUFFICIENT_FRAMES;
        }

        if (MEMPHY_get_freefp(caller->mram, &fpn) == 0)
        {
            newfp_str->fpn = fpn;
            newfp_str->fp_next = NULL;
        }
        else
        {
            free(newfp_str);
            while (*frm_lst != NULL)
            {
                struct framephy_struct *temp = *frm_lst;
                *frm_lst = (*frm_lst)->fp_next;
                free(temp);
            }
            return ERROR_INSUFFICIENT_FRAMES;
        }

        if (*frm_lst == NULL)
        {
            *frm_lst = newfp_str;
        }
        else
```

```
        {
            current->fp_next = newfp_str;
        }
        current = newfp_str;
    }

    return 0;
}

int vm_map_ram(struct pcb_t *caller, int astart, int aend, int mapstart, int incpgnum,
    struct vm_rg_struct *ret_rg)
{
    struct framephy_struct *frm_lst = NULL;
    int ret_alloc;

    ret_alloc = alloc_pages_range(caller, incpgnum, &frm_lst);

    if (ret_alloc < 0 && ret_alloc != -3000)
        return -1;

    if (ret_alloc == -3000)
    {
        return -1;
    }

    vmap_page_range(caller, mapstart, incpgnum, frm_lst, ret_rg);

    return 0;
}

int __swap_cp_page(struct memphy_struct *mpsrc, int srcfpn,
                   struct memphy_struct *mpdst, int dstfpn)
{
    int cellidx;
    int addrsrc, addrdst;
    for (cellidx = 0; cellidx < PAGING_PAGESZ; cellidx++)
    {
        addrsrc = srcfpn * PAGING_PAGESZ + cellidx;
        addrdst = dstfpn * PAGING_PAGESZ + cellidx;

        BYTE data;
        MEMPHY_read(mpsrc, addrsrc, &data);
        MEMPHY_write(mpdst, addrdst, data);
    }

    return 0;
}

int init_mm(struct mm_struct *mm, struct pcb_t *caller)
{
    struct vm_area_struct *vma0 = malloc(sizeof(struct vm_area_struct));

    mm->pgd = malloc(PAGING_MAX_PGN * sizeof(uint32_t));

    vma0->vm_id = 0;
    vma0->vm_start = 0;
    vma0->vm_end = vma0->vm_start;
    vma0->sbrk = vma0->vm_start;
    vma0->vm_mm = mm;

    vma0->vm_next = NULL;
    vma0->vm_freerg_list = NULL;

    mm->mmap = vma0;

    struct vm_rg_struct *first_rg = init_vm_rg(vma0->vm_start, vma0->vm_end);
    enlist_vm_rg_node(&vma0->vm_freerg_list, first_rg);

    return 0;
}

struct vm_rg_struct *init_vm_rg(int rg_start, int rg_end)
```

```
206 {
207     struct vm_rg_struct *rgnode = malloc(sizeof(struct vm_rg_struct));
208     if (!rgnode)
209         return NULL;
210     rgnode->rg_start = rg_start;
211     rgnode->rg_end = rg_end;
212     rgnode->rg_next = NULL;
213     return rgnode;
214 }
215 
216 
217 int enlist_vm_rg_node(struct vm_rg_struct **rglist, struct vm_rg_struct *rgnode)
218 {
219     rgnode->rg_next = *rglist;
220     *rglist = rgnode;
221 
222     return 0;
223 }
224 
225 int enlist_pgn_node(struct pgn_t **plist, int pgn)
226 {
227     struct pgn_t *pnode = malloc(sizeof(struct pgn_t));
228 
229     pnode->pgn = pgn;
230     pnode->pg_next = *plist;
231     *plist = pnode;
232 
233     return 0;
234 }
235 
236 int print_list_fp(struct framephy_struct *ifp)
237 {
238     struct framephy_struct *fp = ifp;
239 
240     printf("print_list_fp: ");
241     if (fp == NULL)
242     {
243         printf("NULL list\n");
244         return -1;
245     }
246     printf("\n");
247     while (fp != NULL)
248     {
249         printf("fp[%d]\n", fp->fpn);
250         fp = fp->fp_next;
251     }
252     printf("\n");
253     return 0;
254 }
255 
256 int print_list_rg(struct vm_rg_struct *irg)
257 {
258     struct vm_rg_struct *rg = irg;
259 
260     printf("print_list_rg: ");
261     if (rg == NULL)
262     {
263         printf("NULL list\n");
264         return -1;
265     }
266     printf("\n");
267     while (rg != NULL)
268     {
269         printf("rg[%ld->%ld]\n", rg->rg_start, rg->rg_end);
270         rg = rg->rg_next;
271     }
272     printf("\n");
273     return 0;
274 }
275 
276 int print_list_vma(struct vm_area_struct *ivma)
277 {
```

```
278     struct vm_area_struct *vma = ivma;
279
280     printf("print_list_vma: ");
281     if (vma == NULL)
282     {
283         printf("NULL list\n");
284         return -1;
285     }
286     printf("\n");
287     while (vma != NULL)
288     {
289         printf("va[%ld->%ld]\n", vma->vm_start, vma->vm_end);
290         vma = vma->vm_next;
291     }
292     printf("\n");
293     return 0;
294 }
295
296 int print_list_pgn(struct pgn_t *ip)
297 {
298     printf("print_list_pgn: ");
299     if (ip == NULL)
300     {
301         printf("NULL list\n");
302         return -1;
303     }
304     printf("\n");
305     while (ip != NULL)
306     {
307         printf("va[%d]-\n", ip->pgn);
308         ip = ip->pg_next;
309     }
310     printf("n");
311     return 0;
312 }
313
314 int print_pgtbl(struct pcb_t *caller, uint32_t start, uint32_t end)
315 {
316     if (end == -1)
317     {
318         struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, 0);
319         if (!cur_vma)
320         {
321             printf("ERROR: VMA not found while printing page table\n");
322             return -1;
323         }
324         end = cur_vma->vm_end;
325     }
326
327     printf("print_pgtbl: %d - %d\n", start, end);
328     return 0;
329 }
```

Giải thích các hàm

`init_pte`

- **Chức năng:** Khởi tạo một entry trong bảng trang (Page Table Entry - PTE).
- **Tham số:**
  - `pte`: Con trỏ đến PTE cần khởi tạo.
  - `pre`: Bit present (0 hoặc khác 0). Xác định xem trang có hiện diện trong RAM hay không.
  - `fpn`: Frame number (số frame vật lý) nếu trang hiện diện trong RAM.
  - `drt`: Bit dirty (0 hoặc khác 0). Cho biết trang có bị sửa đổi kể từ khi được đưa vào RAM hay không.
  - `swp`: Bit swap (0 hoặc khác 0). Cho biết trang đã được swap ra đĩa hay chưa.

  - `swptyp`: Loại swap (ví dụ: loại thiết bị lưu trữ swap).
  - `swpoff`: Offset trong khu vực swap trên đĩa.

- **Hoạt động:**
  1. Kiểm tra xem bit `pre` có khác 0 hay không (trang present).
  2. Nếu trang present và không swapped (`swp` = 0):
     (a) Kiểm tra `fpn` khác 0 (frame hợp lệ). Nếu không, trả về lỗi.
     (b) Đặt bit PRESENT trong PTE.
     (c) Xóa bit SWAPPED trong PTE.
     (d) Xóa bit DIRTY trong PTE.
     (e) Đặt giá trị `fpn` vào PTE bằng cách sử dụng macro `SETVAL`.
  3. Nếu trang present và swapped (`swp` khác 0):
     (a) Đặt bit PRESENT trong PTE.
     (b) Đặt bit SWAPPED trong PTE.
     (c) Xóa bit DIRTY trong PTE.
     (d) Đặt giá trị `swptyp` và `swpoff` vào PTE bằng cách sử dụng macro `SETVAL`.
- **Mục đích:** Khởi tạo PTE với các thông tin cần thiết để quản lý trang, bao gồm vị trí của nó trong RAM hoặc trên đĩa swap.

`pte_set_swap`

- **Chức năng:** Thiết lập PTE cho một trang đã được swap ra đĩa.
- **Tham số:**
  - `pte`: Con trỏ đến PTE cần cập nhật.
  - `swptyp`: Loại swap.
  - `swpoff`: Offset trong khu vực swap.
- **Hoạt động:**
  1. Đặt bit PRESENT trong PTE.
  2. Đặt bit SWAPPED trong PTE.
  3. Đặt giá trị `swptyp` và `swpoff` vào PTE bằng macro `SETVAL`.
- **Mục đích:** Cập nhật PTE khi một trang bị swap ra đĩa, lưu trữ thông tin cần thiết để swap trang trở lại RAM sau này.

`pte_set_fpn`

- **Chức năng:** Thiết lập PTE cho một trang đang hiện diện trong RAM.
- **Tham số:**
  - `pte`: Con trỏ đến PTE cần cập nhật.
  - `fpn`: Frame number.
- **Hoạt động:**
  1. Xóa hết nội dung hiện tại của PTE.
  2. Đặt bit PRESENT trong PTE.
  3. Xóa bit SWAPPED trong PTE.
  4. Đặt giá trị `fpn` vào PTE bằng macro `SETVAL`.
- **Mục đích:** Cập nhật PTE khi một trang được đưa vào RAM, lưu trữ số frame vật lý nơi trang được lưu trữ.

vmap_page_range

- **Chức năng:** Ánh xạ một dải các trang ảo tới các frame vật lý đã được cấp phát.
- **Tham số:**
  - `caller`: PCB (Process Control Block) của tiến trình.
  - `addr`: Địa chỉ ảo bắt đầu của dải cần ánh xạ.
  - `pgnum`: Số lượng trang cần ánh xạ.
  - `frames`: Danh sách các `framephy_struct` đại diện cho các frame vật lý đã được cấp phát.
  - `ret_rg`: Con trỏ đến một `vm_rg_struct` để lưu trữ thông tin về vùng bộ nhớ ảo (virtual memory region) đã ánh xạ (start address, end address).
- **Hoạt động:**
  1. Tính toán số trang (PGN) bắt đầu từ địa chỉ ảo `addr`.
  2. Thiết lập thông tin vùng bộ nhớ ảo (`ret_rg`) với địa chỉ bắt đầu, địa chỉ kết thúc và ID vùng ảo (VMA ID) là 0.
  3. Duyệt qua danh sách các frame vật lý (`frames`) và số lượng trang (`pgnum`).
  4. Với mỗi trang, lấy PTE tương ứng trong bảng trang của tiến trình (`caller->mm->pgd`).
  5. Đặt FPN (Frame Physical Number) và bit PRESENT vào PTE.
  6. Duyệt qua các PGN đã ánh xạ và thêm chúng vào danh sách FIFO của tiến trình để theo dõi thứ tự sử dụng trang.
- **Mục đích:** Tạo ánh xạ giữa bộ nhớ ảo và bộ nhớ vật lý, cho phép tiến trình truy cập bộ nhớ một cách liền mạch.

alloc_pages_range

- **Chức năng:** Cấp phát một dải các frame vật lý từ RAM.
- **Tham số:**
  - `caller`: PCB của tiến trình.
  - `req_pgnum`: Số lượng trang cần cấp phát.
  - `frm_lst`: Con trỏ đến con trỏ của một `framephy_struct`. Hàm sẽ tạo một danh sách liên kết các `framephy_struct` đại diện cho các frame đã cấp phát và gán danh sách này cho `*frm_lst`.
- **Hoạt động:**
  1. Duyệt qua số lượng trang cần cấp phát (`req_pgnum`).
  2. Với mỗi trang:
     (a) Cấp phát một `framephy_struct`. Nếu cấp phát thất bại, giải phóng mọi frame đã cấp phát trước đó và trả về lỗi.
     (b) Lấy một frame vật lý tự do từ RAM bằng `MEMPHY_get_freefp`. Nếu không có frame tự do, giải phóng `framephy_struct` vừa cấp phát, giải phóng mọi frame đã cấp phát trước đó và trả về lỗi.
     (c) Gán FPN của frame đã cấp phát cho `framephy_struct`.
     (d) Thêm `framephy_struct` vào danh sách liên kết `*frm_lst`.
- **Mục đích:** Cấp phát các frame vật lý từ RAM để sử dụng cho việc ánh xạ bộ nhớ ảo.

vm_map_ram

- **Chức năng:** Cấp phát các frame vật lý và ánh xạ chúng vào một vùng bộ nhớ ảo.
- **Tham số:**
  - `caller`: PCB của tiến trình.
  - `astart`: Địa chỉ ảo bắt đầu (không sử dụng).
  - `aend`: Địa chỉ ảo kết thúc (không sử dụng).
  - `mapstart`: Địa chỉ ảo bắt đầu của vùng cần ánh xạ.
  - `incpgnum`: Số lượng trang cần ánh xạ.
  - `ret_rg`: Con trỏ đến một `vm_rg_struct` để lưu trữ thông tin về vùng bộ nhớ ảo đã ánh xạ.
- **Hoạt động:**
  1. Cấp phát một dải các frame vật lý bằng cách gọi `alloc_pages_range`.
  2. Nếu cấp phát thành công, ánh xạ dải các trang ảo tới các frame vật lý đã cấp phát bằng cách gọi `vmap_page_range`.
- **Mục đích:** Kết hợp việc cấp phát frame và ánh xạ bộ nhớ ảo để tạo ra một vùng bộ nhớ ảo có thể sử dụng được.

__swap_cp_page

- **Chức năng:** Sao chép nội dung của một trang từ một vùng nhớ vật lý sang một vùng nhớ vật lý khác.
- **Tham số:**
  - `mpsrc`: Cấu trúc `memphy_struct` đại diện cho vùng nhớ nguồn.
  - `srcfpn`: Frame number của trang nguồn.
  - `mpdst`: Cấu trúc `memphy_struct` đại diện cho vùng nhớ đích.
  - `dstfpn`: Frame number của trang đích.
- **Hoạt động:**
  1. Duyệt qua từng byte trong trang (kích thước `PAGING_PAGESZ`).
  2. Tính địa chỉ vật lý của byte hiện tại trong trang nguồn và trang đích.
  3. Đọc byte từ trang nguồn.
  4. Ghi byte vào trang đích.
- **Mục đích:** Sao chép dữ liệu giữa các trang trong RAM hoặc giữa RAM và swap.

init_mm

- **Chức năng:** Khởi tạo cấu trúc `mm_struct` (memory management structure) cho một tiến trình.
- **Tham số:**
  - `mm`: Con trỏ đến `mm_struct` cần khởi tạo.
  - `caller`: PCB của tiến trình.
- **Hoạt động:**
  1. Cấp phát một `vm_area_struct` (VMA) cho vùng nhớ ảo 0.
  2. Cấp phát bảng trang (`pgd`) cho `mm_struct`.
  3. Khởi tạo các trường của VMA (ID, địa chỉ bắt đầu, địa chỉ kết thúc, sbrk).
  4. Khởi tạo danh sách các vùng tự do (free region list) của VMA.
- **Mục đích:** Thiết lập cấu trúc dữ liệu cần thiết để quản lý bộ nhớ ảo cho một tiến trình.

init_vm_rg

- **Chức năng:** Khởi tạo một `vm_rg_struct` (virtual memory region structure).
- **Tham số:**
  - `rg_start`: Địa chỉ bắt đầu của vùng.
  - `rg_end`: Địa chỉ kết thúc của vùng.
- **Hoạt động:**
  1. Cấp phát một `vm_rg_struct`.
  2. Gán địa chỉ bắt đầu và địa chỉ kết thúc cho `vm_rg_struct`.
- **Mục đích:** Tạo một cấu trúc đại diện cho một vùng bộ nhớ ảo.

enlist_vm_rg_node

- **Chức năng:** Thêm một `vm_rg_struct` vào danh sách liên kết các `vm_rg_struct`.
- **Tham số:**
  - `rglist`: Con trỏ đến con trỏ của phần tử đầu tiên trong danh sách liên kết.
  - `rgnode`: Con trỏ đến `vm_rg_struct` cần thêm vào danh sách.
- **Hoạt động:** Thêm `rgnode` vào đầu danh sách `rglist`.
- **Mục đích:** Quản lý danh sách các vùng bộ nhớ ảo.

enlist_pgn_node

- **Chức năng:** Thêm một số trang (page number) vào danh sách liên kết FIFO (First-In, First-Out).
- **Tham số:**
  - `plist`: Con trỏ đến con trỏ của phần tử đầu tiên trong danh sách liên kết.
  - `pgn`: Số trang cần thêm vào danh sách.
- **Hoạt động:** Thêm `pgn` vào đầu danh sách `plist`.
- **Mục đích:** Quản lý danh sách các trang theo thứ tự chúng được truy cập, để sử dụng cho thuật toán thay thế trang (page replacement).

print_pgtbl

- **Chức năng:** In ra nội dung của bảng trang (page table).
- **Tham số:**
  - `caller`: PCB của tiến trình.
  - `start`: Địa chỉ ảo bắt đầu (không sử dụng).
  - `end`: Địa chỉ ảo kết thúc.
- **Hoạt động:**
  1. Lấy VMA từ `caller->mm` dựa trên số 0.
  2. In ra địa chỉ bắt đầu và kết thúc của vùng nhớ.
- **Mục đích:** Gỡ lỗi (debugging) và kiểm tra nội dung của bảng trang.

2.6.3. File mm-vm.c

`mm-vm.c` chịu trách nhiệm quản lý bộ nhớ ảo (Virtual Memory - VM) cho tiến trình. Các chức năng chính bao gồm tổ chức danh sách các vùng nhớ ảo (Virtual Memory Area - VMA), cấp phát vùng nhớ mới, mở rộng vùng nhớ động (tương ứng với lời gọi hệ thống `brk()`), và kiểm tra sự chồng lấp vùng nhớ. Ngoài ra, module này còn hỗ trợ cơ chế hoán đổi trang (page swapping) giữa bộ nhớ chính và bộ nhớ phụ để tối ưu sử dụng tài nguyên.

Listing 5: mm-vm.c

```c
#include "string.h"
#include "mm.h"
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>

struct vm_area_struct *get_vma_by_num(struct mm_struct *mm, int vmaid)
{
    struct vm_area_struct *pvma = mm->mmap;
    while (pvma != NULL) {
        if (pvma->vm_id == vmaid)
            return pvma;
        pvma = pvma->vm_next;
    }
    return NULL;
}

int __mm_swap_page(struct pcb_t *caller, int vicfpn, int swpfpn)
{
    __swap_cp_page(caller->mram, vicfpn, caller->active_mswp, swpfpn);
    return 0;
}

struct vm_rg_struct *get_vm_area_node_at_brk(struct pcb_t *caller, int vmaid, int size,
    int alignedsz)
{
    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
    if (!cur_vma) {
        return NULL;
    }

    struct vm_rg_struct *newrg = malloc(sizeof(struct vm_rg_struct));
    if (newrg == NULL) {
        return NULL;
    }

    newrg->rg_start = cur_vma->sbrk;
    newrg->rg_end = cur_vma->sbrk + alignedsz;

    return newrg;
}

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

    while (vma != NULL) {
        if (vma->vm_id == vmaid) {
            vma = vma->vm_next;
            continue;
        }

        if (!(vmaend <= vma->vm_start || vmastart >= vma->vm_end)) {
            return -1;
        }

        vma = vma->vm_next;
    }

    return 0;
}

```

```c
int inc_vma_limit(struct pcb_t *caller, int vmaid, int inc_sz) {
    struct vm_rg_struct *newrg = malloc(sizeof(struct vm_rg_struct));
    if (newrg == NULL)
        return -1;

    int inc_amt = PAGING_PAGE_ALIGNSZ(inc_sz);
    int incnumpage = inc_amt / PAGING_PAGESZ;

    struct vm_rg_struct *area = get_vm_area_node_at_brk(caller, vmaid, inc_sz, inc_amt);
    if (area == NULL) {
        free(newrg);
        return -1;
    }

    struct vm_area_struct *cur_vma = get_vma_by_num(caller->mm, vmaid);
    if (cur_vma == NULL) {
        free(newrg);
        return -1;
    }

    int old_end = cur_vma->vm_end;

    if (validate_overlap_vm_area(caller, vmaid, area->rg_start, area->rg_end) < 0) {
        free(newrg);
        return -1;
    }

    cur_vma->vm_end = old_end + inc_amt;

    if (vm_map_ram(caller, area->rg_start, area->rg_end, old_end, incnumpage, newrg) < 0)
        {
        cur_vma->vm_end = old_end;
        free(newrg);
        return -1;
    }

    return 0;
}
```

Giải thích các hàm

`get_vma_by_num`

- **Chức năng:** Lấy một VMA (Virtual Memory Area) từ danh sách các VMA của một tiến trình dựa trên ID của VMA.
- **Tham số:**
  - `mm`: Cấu trúc `mm_struct` đại diện cho memory management của tiến trình.
  - `vmaid`: ID của VMA cần tìm.
- **Hoạt động:**
  1. Duyệt qua danh sách liên kết các VMA của `mm` (bắt đầu từ `mm->mmap`).
  2. Với mỗi VMA, so sánh `vmaid` với `vm_id` của VMA.
  3. Nếu tìm thấy VMA có ID khớp, trả về con trỏ đến VMA đó.
  4. Nếu duyệt hết danh sách mà không tìm thấy, trả về `NULL`.
- **Mục đích:** Tìm kiếm một VMA cụ thể trong danh sách các VMA của một tiến trình, cho phép truy cập và thao tác với VMA đó.

`__mm_swap_page`

- **Chức năng:** Thực hiện việc swap (hoán đổi) một trang nhớ giữa RAM và swap space.

- **Tham số:**
  - `caller`: PCB (Process Control Block) của tiến trình gọi.
  - `vicfpn`: Frame number của trang trong RAM (victim page frame number).
  - `swpfpn`: Frame number của trang trong swap space (swap page frame number).
- **Hoạt động:**
  1. Gọi hàm `__swap_cp_page` để sao chép nội dung của trang từ RAM (frame `vicfpn` trong `caller->mram`) sang swap space (frame `swpfpn` trong `caller->active_mswp`).
- **Mục đích:** Di chuyển một trang từ RAM sang swap space để giải phóng RAM hoặc đưa một trang từ swap space vào RAM.

`get_vm_area_node_at_brk`

- **Chức năng:** Tạo một `vm_rg_struct` đại diện cho một vùng nhớ mới, bắt đầu từ vị trí break hiện tại (`sbrk`) của một VMA.
- **Tham số:**
  - `caller`: PCB của tiến trình gọi.
  - `vmaid`: ID của VMA.
  - `size`: Kích thước yêu cầu của vùng nhớ mới.
  - `alignedsz`: Kích thước đã được căn chỉnh theo yêu cầu phân trang.
- **Hoạt động:**
  1. Tìm VMA tương ứng với `vmaid` bằng cách gọi `get_vma_by_num`.
  2. Nếu không tìm thấy VMA, trả về `NULL`.
  3. Cấp phát một `vm_rg_struct`. Nếu cấp phát thất bại, trả về `NULL`.
  4. Thiết lập `rg_start` của `vm_rg_struct` bằng giá trị `sbrk` của VMA.
  5. Thiết lập `rg_end` của `vm_rg_struct` bằng `sbrk` + `alignedsz`.
- **Mục đích:** Tạo một vùng nhớ ảo mới để cấp phát, sử dụng vị trí break (sbrk) hiện tại làm điểm bắt đầu và kích thước đã căn chỉnh để xác định điểm kết thúc.

`validate_overlap_vm_area`

- **Chức năng:** Kiểm tra xem một vùng nhớ ảo mới có bị overlap (chồng lấn) với các VMA (Virtual Memory Area) khác của tiến trình hay không.
- **Tham số:**
  - `caller`: PCB của tiến trình gọi.
  - `vmaid`: ID của VMA sẽ được mở rộng.
  - `vmastart`: Địa chỉ bắt đầu của vùng nhớ ảo mới.
  - `vmaend`: Địa chỉ kết thúc của vùng nhớ ảo mới.
- **Hoạt động:**
  1. Duyệt qua danh sách liên kết các VMA của tiến trình.
  2. Bỏ qua VMA đang được mở rộng (có `vm_id` trùng với `vmaid`).
  3. Với mỗi VMA còn lại, kiểm tra xem vùng nhớ mới có bị overlap với VMA đó hay không.
  4. Nếu phát hiện overlap, trả về -1.
  5. Nếu duyệt hết danh sách mà không phát hiện overlap, trả về 0.
- **Mục đích:** Đảm bảo rằng việc cấp phát hoặc mở rộng một vùng nhớ ảo không gây ra xung đột với các vùng nhớ khác, duy trì tính toàn vẹn của không gian địa chỉ ảo.

`inc_vma_limit`

- **Chức năng:** Tăng giới hạn (limit) của một VMA để cấp phát thêm bộ nhớ.
- **Tham số:**
  - `caller`: PCB của tiến trình gọi.
  - `vmaid`: ID của VMA cần tăng giới hạn.
  - `inc_sz`: Kích thước cần tăng thêm (increment size).
- **Hoạt động:**
  1. Tính kích thước đã căn chỉnh (`inc_amt`) và số trang cần thiết (`incnumpage`) dựa trên `inc_sz`.
  2. Lấy vùng nhớ mới dự kiến (`area`) bằng cách gọi `get_vm_area_node_at_brk`.
  3. Kiểm tra xem vùng nhớ mới có bị overlap với các VMA khác bằng cách gọi `validate_overlap_vm_area`.
  4. Nếu không có overlap:
     (a) Cập nhật giới hạn của VMA (`cur_vma->vm_end`).
     (b) Ánh xạ vùng nhớ mới vào RAM bằng cách gọi `vm_map_ram`.
     (c) Nếu việc ánh xạ thành công, trả về 0.
  5. Nếu có overlap hoặc việc ánh xạ thất bại, khôi phục lại giới hạn ban đầu của VMA và trả về -1.
- **Mục đích:** Mở rộng vùng nhớ ảo được quản lý bởi một VMA, cho phép tiến trình cấp phát thêm bộ nhớ trong vùng đó.

2.6.4. File mm-memphy.c

`mm-memphy.c` mô phỏng bộ nhớ vật lý (Physical Memory). Nó quản lý việc đọc/ghi dữ liệu từ RAM, khởi tạo các frame vật lý, và cung cấp các thao tác như lấy/trả frame trống. Ngoài ra, module còn hỗ trợ hai chế độ truy cập bộ nhớ: ngẫu nhiên (random access) và tuần tự (sequential access), đồng thời cung cấp hàm dump bộ nhớ để kiểm tra giá trị tại các địa chỉ cụ thể.

Listing 6: mm-memphy.c

```c
#include "mm.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

int MEMPHY_mv_csr(struct memphy_struct *mp, int offset)
{
   int numstep = 0;

   mp->cursor = 0;
   while (numstep < offset && numstep < mp->maxsz)
   {
      mp->cursor = (mp->cursor + 1) % mp->maxsz;
      numstep++;
   }

   return 0;
}

int MEMPHY_seq_read(struct memphy_struct *mp, int addr, BYTE *value)
{
   if (mp == NULL)
      return -1;

   if (!mp->rdmflg)
      return -1;

   MEMPHY_mv_csr(mp, addr);
   *value = (BYTE)mp->storage[addr];

```

```
    return 0;
}

int MEMPHY_read(struct memphy_struct *mp, int addr, BYTE *value)
{
    if (mp == NULL)
       return -1;

    if (mp->rdmflg)
       *value = mp->storage[addr];
    else
       return MEMPHY_seq_read(mp, addr, value);

    return 0;
}

int MEMPHY_seq_write(struct memphy_struct *mp, int addr, BYTE value)
{

    if (mp == NULL)
       return -1;

    if (!mp->rdmflg)
       return -1;

    MEMPHY_mv_csr(mp, addr);
    mp->storage[addr] = value;

    return 0;
}

int MEMPHY_write(struct memphy_struct *mp, int addr, BYTE data)
{
    if (mp == NULL) return -1;

    if (mp->rdmflg)
    {
        mp->storage[addr] = data;
    }
    else
    {
        return MEMPHY_seq_write(mp, addr, data);
    }

    return 0;
}


int MEMPHY_format(struct memphy_struct *mp, int pagesz)
{
    int numfp = mp->maxsz / pagesz;
    struct framephy_struct *newfst, *fst;
    int iter = 0;

    if (numfp <= 0)
       return -1;

    fst = malloc(sizeof(struct framephy_struct));
    fst->fpn = iter;
    mp->free_fp_list = fst;

    for (iter = 1; iter < numfp; iter++)
    {
       newfst = malloc(sizeof(struct framephy_struct));
       newfst->fpn = iter;
       newfst->fp_next = NULL;
       fst->fp_next = newfst;
       fst = newfst;
    }

    return 0;
}
```

```
104
105
106  int MEMPHY_get_freefp(struct memphy_struct *mp, int *retfpn)
107  {
108     struct framephy_struct *fp = mp->free_fp_list;
109
110     if (fp == NULL)
111        return -1;
112
113     *retfpn = fp->fpn;
114     mp->free_fp_list = fp->fp_next;
115
116     free(fp);
117     return 0;
118  }
119
120 int MEMPHY_dump(struct memphy_struct *mp) {
121   if (mp == NULL || mp->storage == NULL)
122     return -1;
123
124   for (int i = 0; i <= mp->maxsz - sizeof(uint32_t); i += sizeof(uint32_t)) {
125     uint32_t val;
126     memcpy(&val, &mp->storage[i], sizeof(uint32_t));
127     if (val != 0) {
128       printf("%08X: %u\n", i, val);
129     }
130   }
131
132   return 0;
133 }
134
135  int MEMPHY_put_freefp(struct memphy_struct *mp, int fpn)
136  {
137     struct framephy_struct *fp = mp->free_fp_list;
138     struct framephy_struct *newnode = malloc(sizeof(struct framephy_struct));
139
140     newnode->fpn = fpn;
141     newnode->fp_next = fp;
142     mp->free_fp_list = newnode;
143
144     return 0;
145  }
146
147  int init_memphy(struct memphy_struct *mp, int max_size, int randomflg)
148  {
149      if (!mp || max_size <= 0) return -1;
150
151      mp->storage = (BYTE *)malloc(max_size * sizeof(BYTE));
152      if (!mp->storage) {
153          fprintf(stderr, "Error: Cannot allocate physical memory storage.\n");
154          return -1;
155      }
156
157      mp->maxsz = max_size;
158      memset(mp->storage, 0, max_size * sizeof(BYTE));
159
160      if (MEMPHY_format(mp, PAGING_PAGESZ) < 0) {
161          fprintf(stderr, "Error: Failed to format physical memory.\n");
162          free(mp->storage);
163          return -1;
164      }
165
166      mp->rdmflg = (randomflg != 0) ? 1 : 0;
167
168      if (!mp->rdmflg) {
169          mp->cursor = 0;
170      }
171
172      return 0;
173  }
```

Giải thích về các hàm

`MEMPHY_mv_csr`

- **Chức năng:** Di chuyển con trỏ (cursor) trong bộ nhớ vật lý tuần tự.
- **Tham số:**
  - `mp`: Con trỏ đến cấu trúc `memphy_struct` đại diện cho bộ nhớ vật lý.
  - `offset`: Độ lệch (offset) cần di chuyển con trỏ.
- **Hoạt động:**
  1. Đặt `mp->cursor` về 0.
  2. Lặp lại quá trình tăng `mp->cursor` và đếm số bước cho đến khi đạt đến `offset` hoặc đến cuối bộ nhớ (`mp->maxsz`).
  3. `mp->cursor` được tăng theo modulo `mp->maxsz` để đảm bảo con trỏ luôn nằm trong phạm vi bộ nhớ.
- **Mục đích:** Cho phép truy cập tuần tự vào bộ nhớ vật lý. Hàm này thường được sử dụng khi bộ nhớ vật lý được coi là một thiết bị truy cập tuần tự.

`MEMPHY_seq_read`

- **Chức năng:** Đọc một byte từ bộ nhớ vật lý theo kiểu tuần tự.
- **Tham số:**
  - `mp`: Con trỏ đến cấu trúc `memphy_struct`.
  - `addr`: Địa chỉ cần đọc.
  - `value`: Con trỏ đến biến BYTE để lưu giá trị đọc được.
- **Hoạt động:**
  1. Kiểm tra xem `mp` có phải là `NULL` hay không. Nếu có, trả về lỗi.
  2. Kiểm tra xem bộ nhớ vật lý có hỗ trợ truy cập tuần tự (`mp->rdmflg == 0`) hay không. Nếu không, trả về lỗi.
  3. Di chuyển con trỏ đến địa chỉ cần đọc bằng cách gọi `MEMPHY_mv_csr`.
  4. Đọc byte từ bộ nhớ tại địa chỉ `addr` và lưu vào `*value`.
- **Mục đích:** Cung cấp cơ chế đọc tuần tự từ bộ nhớ vật lý.

`MEMPHY_read`

- **Chức năng:** Đọc một byte từ bộ nhớ vật lý (hỗ trợ cả truy cập ngẫu nhiên và tuần tự).
- **Tham số:**
  - `mp`: Con trỏ đến cấu trúc `memphy_struct`.
  - `addr`: Địa chỉ cần đọc.
  - `value`: Con trỏ đến biến BYTE để lưu giá trị đọc được.
- **Hoạt động:**
  1. Kiểm tra xem `mp` có phải là `NULL` hay không. Nếu có, trả về lỗi.
  2. Nếu bộ nhớ vật lý hỗ trợ truy cập ngẫu nhiên (`mp->rdmflg != 0`), đọc byte từ bộ nhớ tại địa chỉ `addr` và lưu vào `*value`.
  3. Nếu không hỗ trợ truy cập ngẫu nhiên, gọi `MEMPHY_seq_read` để đọc tuần tự.
- **Mục đích:** Cung cấp cơ chế đọc byte từ bộ nhớ vật lý, tự động chọn giữa truy cập ngẫu nhiên và tuần tự tùy thuộc vào cấu hình của bộ nhớ.

MEMPHY_seq_write

- **Chức năng:** Ghi một byte vào bộ nhớ vật lý theo kiểu tuần tự.
- **Tham số:**
  - `mp`: Con trỏ đến cấu trúc `memphy_struct`.
  - `addr`: Địa chỉ cần ghi.
  - `value`: Giá trị BYTE cần ghi.
- **Hoạt động:**
  1. Kiểm tra xem `mp` có phải là `NULL` hay không. Nếu có, trả về lỗi.
  2. Kiểm tra xem bộ nhớ vật lý có hỗ trợ truy cập tuần tự (`mp->rdmflg == 0`) hay không. Nếu không, trả về lỗi.
  3. Di chuyển con trỏ đến địa chỉ cần ghi bằng cách gọi `MEMPHY_mv_csr`.
  4. Ghi giá trị `value` vào bộ nhớ tại địa chỉ `addr`.
- **Mục đích:** Cung cấp cơ chế ghi tuần tự vào bộ nhớ vật lý.

MEMPHY_write

- **Chức năng:** Ghi một byte vào bộ nhớ vật lý (hỗ trợ cả truy cập ngẫu nhiên và tuần tự).
- **Tham số:**
  - `mp`: Con trỏ đến cấu trúc `memphy_struct`.
  - `addr`: Địa chỉ cần ghi.
  - `data`: Giá trị BYTE cần ghi.
- **Hoạt động:**
  1. Kiểm tra xem `mp` có phải là `NULL` hay không. Nếu có, trả về lỗi.
  2. Nếu bộ nhớ vật lý hỗ trợ truy cập ngẫu nhiên (`mp->rdmflg != 0`), ghi giá trị `data` vào bộ nhớ tại địa chỉ `addr`.
  3. Nếu không hỗ trợ truy cập ngẫu nhiên, gọi `MEMPHY_seq_write` để ghi tuần tự.
- **Mục đích:** Cung cấp cơ chế ghi byte vào bộ nhớ vật lý, tự động chọn giữa truy cập ngẫu nhiên và tuần tự tùy thuộc vào cấu hình của bộ nhớ.

MEMPHY_format

- **Chức năng:** Định dạng (format) bộ nhớ vật lý, tạo một danh sách các frame (khung) bộ nhớ tự do.
- **Tham số:**
  - `mp`: Con trỏ đến cấu trúc `memphy_struct`.
  - `pagesz`: Kích thước của một trang (page) bộ nhớ.
- **Hoạt động:**
  1. Tính số lượng frame có thể chứa trong bộ nhớ vật lý (`numfp`).
  2. Nếu số lượng frame nhỏ hơn hoặc bằng 0, trả về lỗi.
  3. Cấp phát một `framephy_struct` cho frame đầu tiên (frame 0).
  4. Thiết lập `mp->free_fp_list` (danh sách các frame tự do) trỏ đến `framephy_struct` vừa cấp phát.
  5. Duyệt qua các frame còn lại (từ 1 đến `numfp - 1`).
  6. Với mỗi frame, cấp phát một `framephy_struct`, thiết lập `fpn` (frame number) và thêm `framephy_struct` vào cuối danh sách `mp->free_fp_list`.
- **Mục đích:** Chuẩn bị bộ nhớ vật lý để sử dụng bằng cách chia nó thành các frame và tạo một danh sách các frame tự do để quản lý.

MEMPHY_get_freefp

- **Chức năng:** Lấy một frame tự do từ danh sách các frame tự do.
- **Tham số:**
  - `mp`: Con trỏ đến cấu trúc `memphy_struct`.
  - `retfpn`: Con trỏ đến biến integer để lưu FPN (Frame Physical Number) của frame được lấy ra.
- **Hoạt động:**
  1. Lấy phần tử đầu tiên (`fp`) từ danh sách `mp->free_fp_list`.
  2. Nếu danh sách trống (`fp == NULL`), trả về lỗi.
  3. Lưu FPN của frame (`fp->fpn`) vào `*retfpn`.
  4. Cập nhật `mp->free_fp_list` để trỏ đến phần tử tiếp theo trong danh sách.
  5. Giải phóng bộ nhớ của `fp`.
- **Mục đích:** Cấp phát một frame vật lý từ danh sách các frame tự do để sử dụng cho một trang (page) bộ nhớ.

MEMPHY_put_freefp

- **Chức năng:** Trả lại một frame đã sử dụng vào danh sách các frame tự do.
- **Tham số:**
  - `mp`: Con trỏ đến cấu trúc `memphy_struct`.
  - `fpn`: FPN (Frame Physical Number) của frame cần trả lại.
- **Hoạt động:**
  1. Cấp phát một `framephy_struct` mới (`newnode`).
  2. Thiết lập `newnode->fpn` bằng `fpn`.
  3. Thêm `newnode` vào đầu danh sách `mp->free_fp_list`.
- **Mục đích:** Giải phóng một frame vật lý đã sử dụng để có thể sử dụng lại sau này.

init_memphy

- **Chức năng:** Khởi tạo cấu trúc `memphy_struct` đại diện cho bộ nhớ vật lý.
- **Tham số:**
  - `mp`: Con trỏ đến cấu trúc `memphy_struct` cần khởi tạo.
  - `max_size`: Kích thước tối đa của bộ nhớ vật lý.
  - `randomflg`: Cờ để xác định xem bộ nhớ có hỗ trợ truy cập ngẫu nhiên (random access) hay không.
- **Hoạt động:**
  1. Kiểm tra tính hợp lệ của tham số `mp` và `max_size`.
  2. Cấp phát bộ nhớ cho `mp->storage` (mảng byte đại diện cho bộ nhớ vật lý). Nếu cấp phát thất bại, trả về lỗi.
  3. Thiết lập `mp->maxsz` bằng `max_size`.
  4. Đặt tất cả các byte trong `mp->storage` về 0.
  5. Định dạng bộ nhớ bằng cách gọi `MEMPHY_format`. Nếu định dạng thất bại, giải phóng `mp->storage` và trả về lỗi.
  6. Thiết lập `mp->rdmflg` dựa trên `randomflg`.
  7. Nếu không hỗ trợ truy cập ngẫu nhiên, đặt `mp->cursor` về 0.
- **Mục đích:** Thiết lập cấu trúc dữ liệu và bộ nhớ cần thiết để mô phỏng bộ nhớ vật lý.

`MEMPHY_dump`

- **Chức năng:** In ra nội dung của bộ nhớ vật lý.
- **Tham số:**
  - `mp`: Con trỏ đến cấu trúc `memphy_struct`.
- **Hoạt động:**
  1. Kiểm tra `mp` và `mp->storage` có phải là `NULL` hay không. Nếu có, trả về lỗi.
  2. Duyệt qua bộ nhớ theo từng đơn vị 4 byte.
  3. Đọc giá trị 4 byte từ bộ nhớ.
  4. Nếu giá trị khác 0, in ra địa chỉ và giá trị.
- **Mục đích:** Gỡ lỗi (debugging) và kiểm tra nội dung của bộ nhớ.

2.7. Tổng kết

Việc quản lý cấp phát và giải phóng vùng nhớ trong hệ điều hành mô phỏng dựa vào cấu trúc `vm_area`, `sbrk` và danh sách vùng trống. Các thao tác đều đơn giản hoá, nhưng vẫn mô phỏng đúng hành vi của hệ điều hành thực tế như Linux. Các vùng nhớ đã cấp phát không được thu hồi vật lý ngay, mà đưa về danh sách tái sử dụng — điều này giúp hạn chế phân mảnh và tăng hiệu suất sử dụng bộ nhớ.

3. System Call và tương tác giữa các Module

3.1. Khái niệm system call trong hệ điều hành mô phỏng

Lời gọi hệ thống là một thủ tục đóng vai trò làm cầu nối giữa tiến trình (process) và hệ điều hành. Đây là cách mà một chương trình máy tính gửi yêu cầu đến nhân (kernel) của hệ điều hành để thực hiện một dịch vụ nào đó (chẳng hạn như đọc/ghi tệp, tạo tiến trình, giao tiếp mạng...).

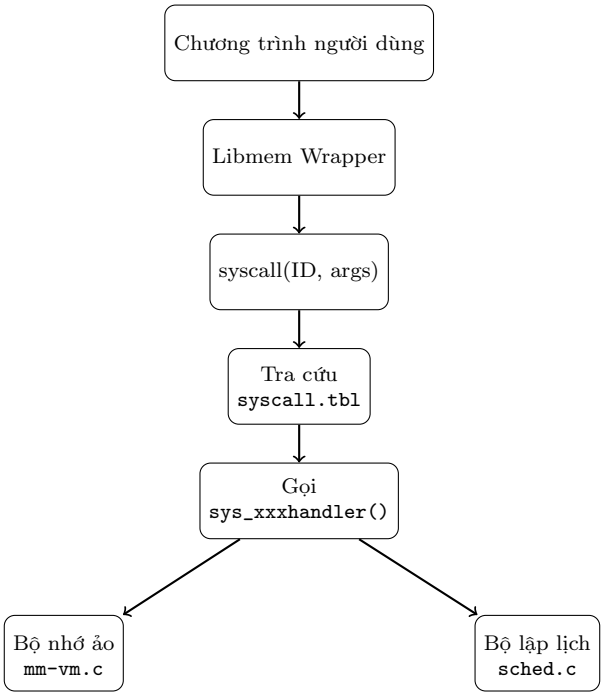
Mỗi hệ điều hành có thể sử dụng tập hợp lời gọi hệ thống khác nhau, tùy vào cách thiết kế và tổ chức của nó.

3.2. Cơ chế hoạt động từng bước

Khi một tiến trình gọi system call, luồng xử lý diễn ra như sau:

1. Chương trình người dùng gọi hàm `syscall(ID, ...)` thông qua thư viện `libmem.h`.
2. Các tham số được truyền qua thanh ghi ảo (`struct sregs`).
3. Hệ thống tra cứu ID trong bảng `syscall.tbl` để tìm handler tương ứng.
4. Gọi hàm xử lý `sys_xxxhandler(struct pcb_t *caller, struct sregs *regs)` trong kernel.
5. Handler này sẽ:
   - Truy cập/ghi dữ liệu bộ nhớ (nếu là `alloc`, `read`, `write`, v.v.).
   - Tương tác với bộ lập lịch (nếu là `killall` hoặc dừng tiến trình).
   - Thay đổi thông tin trong `PCB` nếu cần.

3.3. Sơ đồ luồng xử lý system call



3.4. Truyền tham số qua thanh ghi

Do tiến trình không thể truyền trực tiếp biến vào kernel, các tham số được lưu vào các thanh ghi mô phỏng (a0, a1, a2, ...). Ví dụ:

- ‘a0‘: chứa ID system call.
- ‘a1‘, ‘a2‘, ...: chứa các tham số truyền vào (ví dụ địa chỉ, kích thước, ID vùng nhớ...).

3.5. Ví dụ: system call `killall`

Chức năng: kết thúc tất cả tiến trình có cùng tên chương trình như tiến trình hiện tại.

- Người dùng gọi: 'syscall 101 REGIONID'
- `REGIONID` là ID vùng nhớ chứa tên chương trình (dạng chuỗi).
- Hệ thống đọc chuỗi tên từ bộ nhớ thông qua REGIONID.
- Duyệt toàn bộ danh sách tiến trình:
  - So sánh tên chương trình (trong vùng code của PCB).
  - Nếu khớp, tiến trình đó bị xóa khỏi hàng đợi lập lịch (Scheduler Queue).

Lưu ý: hệ thống phải đảm bảo thao tác này là thread-safe, do có thể đồng thời nhiều CPU thao tác trên scheduler queue.

3.6. Code minh họa cho System Call

3.6.1. File sys_killall.c

`__sys_killall` được thiết kế để hủy tất cả các tiến trình có tên trùng với tên tiến trình được cung cấp. Hàm này duyệt qua các hàng đợi tiến trình đang chạy và tiến trình trong các hàng đợi chuẩn (MLQ nếu có), so sánh tên tiến trình và xóa các tiến trình khớp với tên đã cho.

Listing 7: sys_killall.c

```c
#include "common.h"
#include "syscall.h"
#include "stdio.h"
#include "libmem.h"
#include "queue.h"
#include <string.h>

int __sys_killall(struct pcb_t *caller, struct sc_regs* regs)
{
    char proc_name[100];
    uint32_t data;

    uint32_t memrg = regs->a1;

    int i = 0;
    data = 0;
    while(data != (uint32_t)(-1) && i < 99){
        libread(caller, memrg, i, &data);
        proc_name[i]= (char)data;
        if(data == -1) proc_name[i]='\0';
        i++;
    }
    proc_name[i]='\0';
    printf("The procname retrieved from memregionid %d is \"%s\"\n", memrg, proc_name);

    for (int level = 0; level < MAX_PRIO; ++level) {
        struct queue_t* q = &caller->mlq_ready_queue[level];
        if (q == NULL) {
            break;
        }
        int new_size = 0;
        for (int j = 0; j < q->size; ++j) {
            struct pcb_t* proc = q->proc[j];
            if (strcmp(proc->path, proc_name) == 0) {
                printf("Terminating process \"%s\" in priority level %d\n", proc->path,
                    level);
            } else {
                q->proc[new_size++] = proc;
            }
        }
        q->size = new_size;
```

```
41      }
42
43      return 0;
44  }
```

Giải thích các hàm

`__sys_killall`

- **Chức năng:** Kết thúc tất cả các tiến trình có một tên nhất định.
- **Tham số:**
  - `caller`: Con trỏ đến cấu trúc `pcb_t` (Process Control Block) của tiến trình gọi system call.
  - `regs`: Con trỏ đến cấu trúc `sc_regs` (System Call Registers) chứa các thanh ghi (registers) được sử dụng để truyền tham số cho system call.
- **Hoạt động:**
  1. **Lấy tên tiến trình đích:**
     (a) Lấy ID của vùng nhớ (memory region ID) chứa tên tiến trình đích từ thanh ghi `a1` của `regs` (`memrg = regs->a1`).
     (b) Đọc tên tiến trình từ vùng nhớ được chỉ định bởi `memrg` bằng cách sử dụng hàm `libread`. Tên được đọc từng ký tự cho đến khi gặp ký tự kết thúc chuỗi (`-1` được ép thành `char '0'`) hoặc đạt đến giới hạn 99 ký tự.
     (c) Lưu tên tiến trình vào mảng `proc_name`.
     (d) In tên tiến trình đã đọc ra console để gỡ lỗi.
  2. **Duyệt và kết thúc tiến trình:**
     (a) Duyệt qua các hàng đợi ưu tiên (priority queues) `caller->mlq_ready_queue` (Multi-Level Queue Ready Queue) từ mức ưu tiên cao nhất đến thấp nhất.
     (b) Với mỗi hàng đợi, duyệt qua từng tiến trình trong hàng đợi.
     (c) So sánh tên của tiến trình (`proc->path`) với tên tiến trình đích (`proc_name`) bằng hàm `strcmp`.
     (d) Nếu tên tiến trình khớp:
        i. In thông báo cho biết tiến trình sẽ bị kết thúc.
        ii. **(Trong code hiện tại) Không thực hiện kết thúc tiến trình thực sự.** Code chỉ in ra thông báo, loại bỏ tiến trình khỏi hàng đợi bằng cách nén danh sách, tức chỉ cập nhật lại size của queue.
     (e) Nếu tên tiến trình không khớp, giữ tiến trình trong hàng đợi.
     (f) Cập nhật kích thước của hàng đợi `q->size` để loại bỏ các tiến trình đã bị "kết thúc".
- **Mục đích:** Cung cấp một system call để kết thúc tất cả các tiến trình có một tên nhất định, thường được sử dụng để gỡ lỗi hoặc để loại bỏ các tiến trình không mong muốn khỏi hệ thống.

3.7. Tổng kết

Phần System Call trong hệ điều hành mô phỏng đã xây dựng được một luồng truyền lệnh hoàn chỉnh từ không gian người dùng đến kernel. Qua đó, sinh viên có thể hiểu sâu hơn về kiến trúc OS thực tế, bao gồm: cơ chế trap, xử lý tham số, phân tách quyền truy cập và tương tác module.

4.Trả lời câu hỏi

Question 1

What are the advantages of using the scheduling algorithm described in this assignment compared to other scheduling algorithms you have learned?

Thuật toán lập lịch sử dụng trong bài tập này là Multi-Level Queue Scheduling (MLQ) kết hợp với Round Robin trong từng mức độ ưu tiên. Phương pháp này mang lại nhiều lợi thế so với các thuật toán cơ bản như First-Come-First-Served (FCFS) hay Round Robin đơn thuần:

- **Cân bằng giữa tính ưu tiên và công bằng:** Các tiến trình quan trọng (như tiến trình hệ thống, tiến trình tương tác) được ưu tiên phục vụ trước, trong khi tiến trình ít quan trọng hơn vẫn được đảm bảo không bị đói tài nguyên.
- **Độ trễ thấp cho tiến trình ưu tiên cao:** Hệ thống phản hồi nhanh đối với các tác vụ cần thời gian đáp ứng ngắn.
- **Chi phí vận hành thấp:** Vì mỗi tiến trình được cố định trong một hàng đợi theo ưu tiên, bộ lập lịch chỉ cần duyệt lần lượt qua các hàng đợi, không phải di chuyển tiến trình liên tục như một số thuật toán khác.
- **Khả năng phân loại tiến trình:** Dễ dàng áp dụng các chính sách khác nhau cho từng nhóm tiến trình (ví dụ: Round Robin cho tiến trình người dùng, FCFS cho tiến trình batch).

Tóm lại, thuật toán MLQ trong bài tập kết hợp được những ưu điểm của lập lịch theo ưu tiên và tính công bằng của Round Robin, điều mà các thuật toán đơn giản hơn khó đạt được đồng thời.

Question 2

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?

Việc thiết kế nhiều phân đoạn bộ nhớ (segment) mang lại các lợi ích nổi bật sau:

- **Tổ chức bộ nhớ rõ ràng:** Bộ nhớ được chia thành các khu vực riêng biệt như code, dữ liệu, heap và stack, giúp việc quản lý trở nên có cấu trúc hơn.
- **Quản lý linh hoạt và độc lập:** Mỗi phân đoạn có thể được cấp phát, mở rộng hoặc bảo vệ độc lập, phù hợp với nhu cầu của từng loại dữ liệu.
- **Giảm phân mảnh và tối ưu ánh xạ địa chỉ:** Một số phân đoạn có thể được ánh xạ theo cơ chế paging, trong khi các phân đoạn khác có thể được ánh xạ liên tục hoặc bỏ qua paging.
- **Tăng cường bảo vệ:** Có thể áp dụng chính sách bảo vệ khác nhau cho từng phân đoạn (ví dụ: chỉ đọc cho phân đoạn code, đọc/ghi cho phân đoạn dữ liệu).

Nhờ đó, thiết kế đa phân đoạn giúp hệ điều hành đơn giản quản lý bộ nhớ một cách modul hóa, hiệu quả và an toàn hơn.

Question 3

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

Việc chia địa chỉ thành nhiều hơn hai cấp độ trong hệ thống quản lý bộ nhớ phân trang dẫn đến:

- **Tiết kiệm bộ nhớ cho bảng trang:** Thay vì cần một bảng trang lớn liên tục, các bảng trang đa cấp nhỏ hơn chỉ được tạo khi cần thiết.
- **Khả năng mở rộng cao:** Các hệ thống có không gian địa chỉ ảo lớn (32-bit, 64-bit) được quản lý hiệu quả hơn mà không cần cấp phát thừa bộ nhớ.

- **Tăng độ trễ truy cập bộ nhớ:** Mỗi lần truy cập bộ nhớ cần qua nhiều mức bảng trang, làm tăng số lượng tra cứu (lookup) và thời gian truy cập.
- **Hỗ trợ tốt cho không gian địa chỉ thưa (sparse memory):** Các tiến trình có không gian địa chỉ lớn nhưng sử dụng thực tế ít (như heap lớn nhưng rỗng) không gây lãng phí bộ nhớ.

Như vậy, phân trang đa cấp là giải pháp hiệu quả cho quản lý bộ nhớ quy mô lớn, dù phải đánh đổi một phần hiệu năng do tăng độ trễ truy cập.

Question 4

What are the advantages and disadvantages of segmentation with paging?

Ưu điểm:

- **Tổ chức bộ nhớ logic:** Cho phép chia chương trình thành các đơn vị có ý nghĩa như hàm, cấu trúc dữ liệu, giúp dễ quản lý và bảo vệ.
- **Giảm phân mảnh bên ngoài:** Phân trang trong từng phân đoạn loại bỏ hiện tượng phân mảnh ngoài.
- **Chia sẻ hiệu quả:** Các phân đoạn (ví dụ như thư viện dùng chung) có thể được chia sẻ giữa nhiều tiến trình.
- **Hỗ trợ mở rộng linh hoạt:** Mỗi phân đoạn có thể mở rộng độc lập mà không ảnh hưởng đến các phân đoạn khác.

Nhược điểm:

- **Tăng độ phức tạp:** Quản lý cần cả bảng phân đoạn và bảng trang, làm tăng chi phí lưu trữ và xử lý.
- **Độ trễ truy cập cao hơn:** Mỗi lần truy cập bộ nhớ cần tra cứu bảng phân đoạn trước, sau đó tra cứu bảng trang.
- **Phân mảnh nội bộ:** Vẫn tồn tại phân mảnh nội bộ trong các trang (nếu dữ liệu không lấp đầy trang).

Question 5

What is the mechanism to pass a complex argument to a system call using the limited registers?

Cơ chế được sử dụng để truyền tham số phức tạp cho system call trong hệ điều hành đơn giản này là:

- **Truyền tham chiếu qua thanh ghi:** Thay vì truyền toàn bộ dữ liệu phức tạp qua thanh ghi (vốn số lượng hạn chế), chỉ truyền địa chỉ hoặc ID của vùng bộ nhớ chứa dữ liệu (ví dụ: sử dụng thanh ghi `a1` để truyền `memrg` - Memory Region ID).
- **Gián tiếp truy cập bộ nhớ:** Handler của system call sẽ đọc nội dung thực sự từ vùng bộ nhớ được chỉ định bằng cách sử dụng các hàm như `libread`.
- **Dựa trên ngữ cảnh tiến trình:** Thông qua PCB của tiến trình gọi (ví dụ struct `pcb_t *caller`), hệ thống có thể kiểm soát truy cập bộ nhớ một cách an toàn.

Cơ chế này vừa tận dụng tốt không gian thanh ghi hạn chế, vừa đảm bảo truyền dữ liệu phức tạp một cách an toàn và hiệu quả.

Question 6

What happens if the syscall job implementation takes too long execution time?

Khi thực hiện system call kéo dài quá lâu:

- **Nếu không có preemption trong kernel mode:** CPU sẽ bị chiếm dụng bởi tiến trình gọi system call, gây đình trệ cho toàn bộ hệ thống.
- **Nếu có preemption hoặc timer interrupt:** Hệ thống có thể gián đoạn system call, lưu trạng thái và chuyển CPU cho tiến trình khác, đảm bảo tính công bằng.
- **Trên hệ thống đa lõi:** Các CPU khác vẫn có thể tiếp tục thực thi tiến trình khác, tuy nhiên CPU đang xử lý system call dài sẽ bị khóa.

Hậu quả thực tế là hệ điều hành có thể bị giảm hiệu suất nghiêm trọng hoặc tệ hơn là bị treo nếu không có cơ chế kiểm soát thời gian thực thi system call thích hợp.

Question 7

What happens if the synchronization is not handled in your Simple OS? Illustrate the problem of your simple OS (assignment outputs) by example if you have any.

Nếu không xử lý đồng bộ trong hệ điều hành đơn giản, các lỗi nghiêm trọng sẽ xảy ra do:

- **Race conditions:** Nhiều CPU hoặc tiến trình có thể đồng thời truy cập và thay đổi các tài nguyên chung như hàng đợi tiến trình sẵn sàng (ready queue) hoặc vùng cấp phát bộ nhớ.
- **Mất dữ liệu hoặc hỏng cấu trúc:** Ví dụ, hai tiến trình cùng thêm vào ready queue nhưng không khóa sẽ gây ra mất nút hoặc đứt liên kết trong danh sách liên kết.
- **Hành vi không xác định:** Kết quả thực thi không ổn định, đôi khi hệ điều hành có thể bỏ sót tiến trình, phân bổ sai bộ nhớ hoặc gây lỗi hệ thống.

5.Tổng kết

Báo cáo này đã trình bày quá trình thiết kế và cài đặt một hệ điều hành mô phỏng đơn giản, tập trung vào ba thành phần chính:

- **Lập lịch tiến trình:**
  - Phần lập lịch tiến trình sử dụng thuật toán Multi-Level Queue (MLQ) kết hợp với cơ chế Round-Robin trong từng hàng đợi, nhằm quản lý thứ tự thực thi tiến trình dựa trên mức độ ưu tiên và thời lượng thời gian (time slice). Việc triển khai MLQ trong hệ thống mô phỏng đã đảm bảo phân chia tài nguyên CPU hợp lý cho các nhóm tiến trình có đặc tính khác nhau.

- **Quản lý bộ nhớ:**
  - Phần quản lý bộ nhớ xây dựng dựa trên mô hình bộ nhớ ảo phân trang, sử dụng các cấu trúc như bảng trang (Page Table), vùng ánh xạ bộ nhớ (VMA) và khu vực bộ nhớ (Region). Các cơ chế cấp phát và giải phóng bộ nhớ động được mô phỏng chi tiết, đồng thời xử lý lỗi trang (Page Fault) thông qua kỹ thuật hoán trang (Swapping) với chính sách FIFO. Ngoài ra, báo cáo cũng đã đề cập đến việc thực hiện các thao tác đọc/ghi bộ nhớ và các yêu cầu đồng bộ hóa dữ liệu.

- **System Call:**
  - Phần System Call mô tả cách cài đặt system call `killall`, cho phép kết thúc tất cả các tiến trình có cùng tên. Quá trình thực hiện system call này yêu cầu tương tác với module quản lý bộ nhớ để đọc tham số, module quản lý tiến trình để xác định và tiêu diệt tiến trình, và module lập lịch để điều chỉnh các hàng đợi tiến trình, trong khi vẫn đảm bảo cơ chế đồng bộ hóa hợp lý.

Bên cạnh việc triển khai các module, báo cáo còn giải quyết một số câu hỏi lý thuyết liên quan nhằm củng cố kiến thức nền tảng về hệ điều hành. Các kết quả kiểm thử từ những trường hợp mẫu đã chứng minh sự đúng đắn và hiệu quả của các chức năng được cài đặt.

Thông qua dự án này, nhóm đã có cơ hội hiểu sâu hơn về các cơ chế cơ bản trong hệ điều hành, từ quản lý tiến trình, quản lý bộ nhớ, đến xử lý lời gọi hệ thống, đồng thời rèn luyện kỹ năng lập trình hệ thống và tư duy thiết kế phần mềm quy mô nhỏ.