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

BÁO CÁO BÀI TẬP LỚN HỆ ĐIỀU HÀNH - CO2017

Đề tài:

Simple Operating System

Giảng viên hướng dẫn: Hoàng Lê Hải Thanh

Nhóm: A

Sinh viên: Nguyễn Văn A - 221xxxx
Nguyễn Văn A - 221xxxx
Nguyễn Văn A - 221xxxx
Nguyễn Văn A - 221xxxx

Thành phố Hồ Chí Minh, tháng 12 năm 2024

Phân chia công việc

STT	Họ và tên	MSSV	Phân công công việc	Tiến độ hoàn thành
1	Nguyễn Văn A	221xxxx	Memory management	100%
2	Nguyễn Văn A	221xxxx	Memory management	100%
3	Nguyễn Văn A	221xxxx	Scheduler	100%
4	Nguyễn Văn A	221xxxx	Scheduler	100%

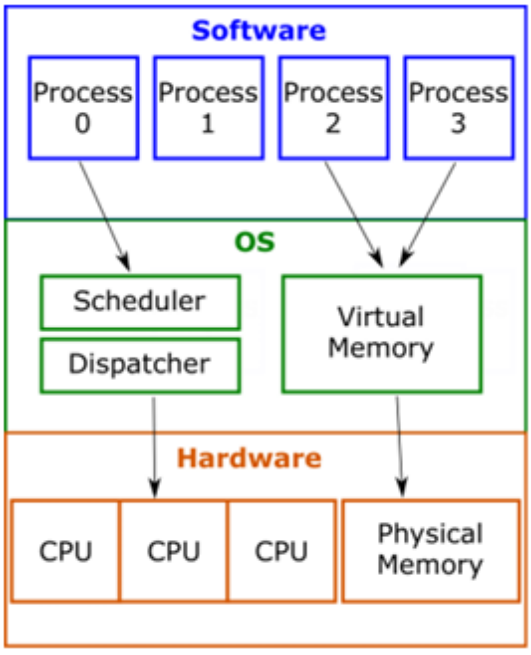
Mục lục

Danh sách hình vẽ

1 Tổng quan đề bài

Bài tập này nhằm mô phỏng một hệ điều hành đơn giản, giúp sinh viên hiểu các khái niệm cơ bản về lập lịch, đồng bộ hóa và quản lý bộ nhớ. Hình 1 minh họa kiến trúc tổng quan của hệ điều hành sẽ được triển khai. Hệ điều hành phải quản lý hai tài nguyên ảo: CPU và RAM thông qua hai thành phần cốt lõi:

- Bộ lập lịch (Scheduler) và Bộ phân phối (Dispatcher): Xác định tiến trình nào được phép chạy trên CPU nào.
- Bộ nhớ ảo (Virtual memory): Tách biệt không gian bộ nhớ của từng tiến trình để chúng không nhận biết sự tồn tại của nhau. RAM vật lý được chia sẻ giữa các tiến trình, nhưng mỗi tiến trình có không gian bộ nhớ ảo riêng. Bộ máy bộ nhớ ảo sẽ ánh xạ và chuyển đổi các địa chỉ ảo mà các tiến trình cung cấp sang địa chỉ vật lý tương ứng.



Hình 1: Tổng quan các modules trong assignmet

Thông qua các module này, hệ điều hành cho phép nhiều tiến trình do người dùng tạo ra chia sẻ và sử dụng các tài nguyên tính toán ảo. Vì vậy, trong bài tập này, chúng ta tập trung triển khai bộ lập lịch/bộ phân phối và bộ máy bộ nhớ ảo.

2 Scheduler

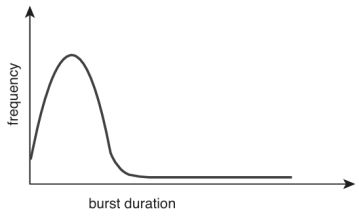
2.1 Scheduler concept

2.1.1 CPU Scheduler

CPU Scheduler là một thành phần của hệ điều hành, có nhiệm vụ chọn một process từ ready queue để cấp phát CPU khi CPU đang trong trạng thái nhàn rỗi. Process được chọn là một trong các processes đã sẵn sàng thực thi (đang ở bộ nhớ và chờ trong ready queue).

Ready queue không nhất thiết phải là hàng đợi kiểu FIFO (First-In, First-Out). Các cấu trúc dữ liệu có thể sử dụng cho ready queue bao gồm:

- FIFO queue (hàng đợi theo thứ tự vào trước ra trước).
- Priority queue (hàng đợi ưu tiên, dựa trên mức độ ưu tiên của quy trình).
- Cây (tree).
- Danh sách liên kết không theo thứ tự (unordered linked list).



Hình 2: Histogram of CPU-burst durations

2.1.2 Preemptive and Nonpreemptive Scheduling

Quyết định định thời CPU diễn ra theo một trong bốn điều kiện sau:

- Khi một quá trình chuyển từ trạng thái running sang waiting
- Khi một quá trình chuyển từ trạng thái running sang ready
- Khi một quá trình chuyển từ trạng thái waiting sang ready

- Khi một quá trình kết thúc

Định thời dưới điều kiện 1 và 4: không nhường (non-preemptive hoặc cooperative). Khi ở trạng thái running, process sẽ thực thi cho đến khi kết thúc hoặc bị blocked do yêu cầu I/O.

Định thời dưới điều kiện 2 và 3: process đang thực thi có thể bị ngắt quãng và chuyển về trạng thái ready, nhường CPU cho một quá trình khác (preemptive)

Preemptive Scheduling có thể gây ra race conditions, cần phải xem xét các vấn đề sau:

- Hai process cùng truy xuất shared data
- Preemptive trong kernel mode
- Vấn đề ngắt quãng trong các hoạt động của hệ điều hành

2.1.3 Dispatcher

- Dispatcher là module đưa quyền điều khiển CPU cho process được chọn bởi short-term scheduler, bao gồm:
  - Chuyển ngữ cảnh
  - Chuyển sang user mode
  - Nhảy đến địa chỉ người sử dụng tiếp tục chương trình
- Dispatch latency: thời gian dispatcher dừng một quá trình và bắt đầu chạy một quá trình khác

2.1.4 Scheduling Criteria

- Hiệu suất CPU (CPU utilization): giữ cho CPU luôn bận (luôn thực thi process) nếu có thể.
- Throughput: số lượng process hoàn thành việc thực thi trong một đơn vị thời gian.
- Turnaround time: Khoảng thời gian cần để việc thực thi một process hoàn tất.
- Waiting time: khoảng thời gian mà process phải chờ trong ready queue.

- Response time: khoảng thời gian process từ lúc nhận yêu cầu đến khi yêu cầu được đáp ứng lần đầu tiên
- Tối ưu hoá việc định thời là làm max CPU utilization, max Throughput, min Turnaround time, min Waiting time, min Response time
- Thông thường, cần tối ưu hoá giá trị đo trung bình của các tiêu chí này. Nhưng trong một số trường hợp, chúng ta làm max hoặc min của một giá trị đó nào đó.

2.2 Các giải thuật Scheduling

2.2.1 First-Come, First-Served Scheduling (FCFS)

FCFS là một trong những thuật toán lập lịch CPU đơn giản nhất và trực quan nhất. Quy trình thực hiện dựa trên nguyên tắc "Đến trước thì phục vụ trước" (First-Come, First-Served).

Tiến trình đầu tiên yêu cầu CPU sẽ được nhận phân bổ CPU đầu tiên trong phương thức này. Hàng đợi FIFO được sử dụng để quản lý chiến lược định thời này. PCB của tiến trình được liên kết với phần đuôi của hàng đợi khi nó đi vào. Do đó, bất cứ khi nào CPU trở nên khả dụng, nó sẽ được gán cho tiến trình ở đầu hàng đợi.

Điểm quan trọng của phương pháp này là nó cung cấp giải thuật định thời Non-Preemptive, khi một quy trình bắt đầu chạy trên CPU, nó sẽ chiếm CPU cho đến khi hoàn tất. Không có quy trình nào khác được phép gián đoạn quy trình hiện tại, ngay cả khi một quy trình có Burst Time ngắn hơn hoặc có độ ưu tiên cao hơn đến sau.

2.2.2 Shortest-Job-First Scheduling

Shortest-Job-First (SJF) là một thuật toán lập lịch CPU dựa trên thời gian Burst Time của các quy trình. Quy trình nào có thời gian thực thi ngắn nhất sẽ được ưu tiên thực hiện trước. Đây là một thuật toán rất hiệu quả trong việc tối ưu hóa thời gian chờ và thời gian hoàn tất.

Đây là giải thuật non-preemptive. Nó là một chính sách định thời ưu tiên cái tiến trình đang đợi có thời gian thực thi ngắn. Trong tất cả các giải thuật định thời SJF có lợi thế là có thời gian chờ đợi trung bình ngắn nhất. Đầu tiên, nó sắp xếp tất cả các tiến trình theo thời gian đến. Sau đó chọn phương thức có thời gian đến ngắn nhất và thời gian thực thi ngắn nhất. Sau khi

chọn, hãy tạo một nhóm các tiến trình sẽ chạy khi tiến trình trước đó hoàn tất, sau đó chọn tiến trình có thời gian thực thi ngắn nhất từ nhóm.

Điểm quan trọng của phương thức này là các tiến trình ngắn được xư lý rất nhanh chóng và Hệ thống cũng có chi phí thấp vì nó chỉ đưa ra quyết định khi một tiến trình kết thúc hoặc một tiến trình mới được thêm vào. Khi một tiến trình mới được thêm vào, giải thuật chỉ phải so sánh tiến trình hiện đang chạy với quy trình mới, bỏ qua bất kỳ tiến trình nào khác đang chờ chạy. Ngoài ra các tiến trình dài có thể bị hoãn vô thời hạn nếu các tiến trình ngắn được thêm vào một cách thường xuyên.

Nó còn được phân loại thành hai loại:

- Non-preemptive SJF: CPU không thể bị gián đoạn sau khi được cấp phát cho một quy trình.
- Preemptive SJF (Shortest Remaining Time First - SRTF): Quy trình đang chạy có thể bị gián đoạn nếu một quy trình mới có Burst Time ngắn hơn xuất hiện.

2.2.3 Round-Robin Scheduling (RR)

Thuật toán Round Robin (RR) tương tự như FCFS, nhưng được bổ sung khả năng gián đoạn (preemption) để hệ thống có thể chuyển đổi giữa các quy trình. Một đơn vị thời gian nhỏ, gọi là time quantum hoặc time slice, được định nghĩa. Time quantum thường có độ dài từ 10 đến 100 mili giây. Ready queue được xử lý như một circular queue. Bộ lập lịch CPU sẽ đi vòng quanh hàng đợi sẵn sàng, cấp CPU cho từng quy trình trong khoảng thời gian tối đa là 1 time quantum.

Nếu có n quá trình trong hàng đợi ready và khoảng quantum time là q thì không có quá trình nào phải chờ đợi quá (n - 1)* q đơn vị thời gian. Quantum time phải lớn hơn thời gian để xử lý clock interrupt (timer) và thời gian dispatching.

Hiệu suất:

+ Nếu q lớn: RR trở thành FCFS

+ Nếu q nhỏ: q phải đủ lớn, vì q nhỏ sẽ sinh ra chi phí chuyển ngữ cảnh lớn.

Thời gian chờ đợi trung bình của RR lớn hơn SJF nhưng có thời gian đáp ứng tốt hơn Trong thực tế, q từ 10ms đến 100ms, thời gian chuyển ngữ cảnh thường $<$ 10ms.

2.2.4 Priority Scheduling

Thuật toán SJF là một trường hợp đặc biệt của thuật toán lập lịch dựa trên độ ưu tiên (priority-scheduling). Mỗi quy trình được gán một độ ưu tiên, và CPU được cấp phát cho quy trình có độ ưu tiên cao nhất. Các quy trình có cùng độ ưu tiên sẽ được lập lịch theo thứ tự FCFS. Thuật toán SJF thực chất là một thuật toán ưu tiên, trong đó độ ưu tiên (p) là nghịch đảo của (thời gian CPU burst dự đoán). Thời gian CPU burst càng lớn thì độ ưu tiên càng thấp, và ngược lại.

Priority có thể là preemptive hoặc non-preemptive (equal-priority):

+ Preemptive Priority Scheduling: CPU có thể bị gián đoạn nếu một quy trình mới có độ ưu tiên cao hơn xuất hiện.

+ Non-Preemptive Priority Scheduling: Quy trình đang chạy trên CPU sẽ không bị gián đoạn, ngay cả khi có quy trình mới với độ ưu tiên cao hơn. Thích hợp cho hệ thống xử lý theo lô (Batch Systems).

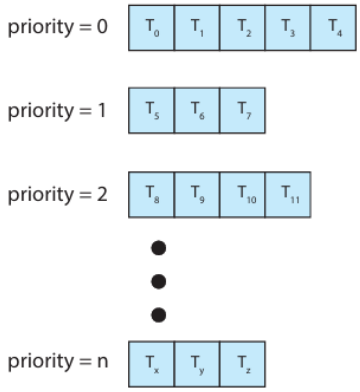
Giải thuật Priority có thể gây ra starvation, các quá trình với độ ưu tiên thấp có thể chờ vô thời hạn. Một trong những cách có thể giải quyết vấn đề này là aging, độ ưu tiên của các job sẽ tăng dần theo thời gian đợi.

2.2.5 Multilevel Queue Scheduling

Với cả lập lịch dựa trên độ ưu tiên (Priority Scheduling) và lập lịch vòng tròn (Round-Robin Scheduling), tất cả các quy trình có thể được đặt trong một hàng đợi duy nhất, và bộ lập lịch sẽ chọn quy trình có độ ưu tiên cao nhất để thực thi. Tùy thuộc vào cách quản lý hàng đợi, có thể cần thực hiện một phép tìm kiếm O(n) để xác định quy trình có độ ưu tiên cao nhất. Trong thực tế, việc sử dụng các hàng đợi riêng biệt cho mỗi mức độ ưu tiên thường dễ dàng hơn. Lập lịch ưu tiên chỉ cần thực thi các quy trình trong hàng đợi có độ ưu tiên cao nhất. Điều này được minh họa trong Hình 3.

Cách tiếp cận này – được gọi là multilevel queue (hàng đợi đa cấp) – cũng hoạt động hiệu quả khi kết hợp lập lịch ưu tiên với lập lịch vòng tròn (round-robin): nếu có nhiều quy trình trong hàng đợi có độ ưu tiên cao nhất, chúng sẽ được thực thi theo thứ tự vòng tròn (round-robin order). Trong dạng tổng quát nhất của cách tiếp cận này, mỗi quy trình được gán một mức độ

ưu tiên cố định (static priority), và quy trình sẽ giữ nguyên trong hàng đợi của nó trong suốt thời gian chạy.



Hình 3: Hàng đợi riêng biệt cho mỗi mức độ ưu tiên

Process không thể từ hàng đợi này sang hàng đợi khác, các quá trình này ở vĩnh viễn một hàng đợi cho đến khi thực hiện xong. Giải thuật này có thể gây ra starvation, các process ở độ hàng đợi có độ ưu tiên thấp có thể chờ vô thời hạn.

2.2.6 Multilevel Feedback Queue Scheduling

Gần giống như Multilevel Queue nhưng giải thuật này cho phép process di chuyển giữa các hàng đợi. Multilevel Feedback Queue được định nghĩa bởi các tham số sau:

+ Số lượng hàng đợi

+ Giải thuật định thời cho mỗi hàng đợi

+ Phương thức upgrade hoặc demote các quá trình từ hàng đợi này sang hàng đợi khác

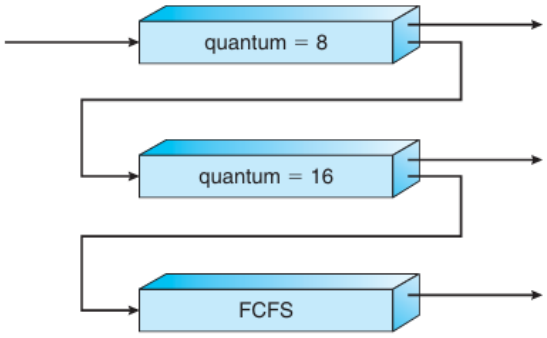
+ Phương thức quyết định hàng đợi nào được thực thi.

Cơ chế này giúp các process interactive và I/O bound ở hàng đợi có độ ưu tiên cao, một process chờ đợi lâu ở hàng đợi có độ ưu tiên thấp chuyển sang hàng đợi có độ ưu tiên cao hơn.Ví dụ với 3 hàng đợi:

+ Q0 : thực hiện giải thuật RR với q = 8ms

+ Q1: thực hiện giải thuật RR với q = 16ms

+ Q2: thực hiện giải thuật FCFS.



Hình 4: Multilevel feedback queues

2.3 Trả lời câu hỏi

Câu hỏi 1: What are the advantages of using the scheduling algorithm described in this assignment compared to other scheduling algorithms you have learned?

Dịch: Ưu điểm của giải thuật định thời được dùng trong bài tập lớn này là gì, so sánh nó với những giải thuật định thời mà bạn đã được học?

Giải thuật định thời được sử dụng trong bài tập lớn này là Multilevel Queue (MLQ). Giải thuật đã được trình bày chi tiết ở phần trước. Sau đây là ưu điểm của MLQ:

- Các tiến trình có đặc tính tương tự được xử lý bởi cùng một hàng đợi, dễ dàng quản lý hơn. Ví dụ: Các tiến trình tương tác yêu cầu thời gian phản hồi nhanh được ưu tiên, trong khi các tiến trình batch được thực hiện khi hệ thống rảnh rỗi.
- Tiến trình thuộc hàng đợi có mức ưu tiên cao hơn sẽ được xử lý trước, phù hợp với các ứng dụng đòi hỏi ưu tiên (ví dụ: tiến trình hệ thống hoặc thời gian thực).
- Các tiến trình trong cùng một hàng đợi được xử lý công bằng, nhờ áp dụng giải thuật như Round Robin hoặc FIFO.

2.4 Implement Scheduler

2.4.1 Code

enqueue(): Đưa một process mới vào cuối queue (hàng đợi). Dưới đây là code:

```
void enqueue(struct queue_t * q, struct pcb_t * proc) {
    /* TODO: put a new process to queue [q] */
    #ifdef MLQ_SCHED
    if (q != NULL && q->size < MAX_QUEUE_SIZE) {
        q->proc[q->size] = proc;
        q->size++;
    }
    #else
    for (int i = 0; i < q->size; i++) {
        if (q->proc[i]->priority <= proc->priority) {
            for (int j = q->size - 1; j >= i; j--)
            q->proc[j + 1] = q->proc[j];

            q->proc[i] = proc;
            q->size++;
            return;
        }
    }
    q->proc[q->size] = proc;
    q->size++;
    #endif
}
```

dequeue(): Lấy ra một process ở vị trí đầu tiên của queue (hàng đợi). Dưới đây là code:

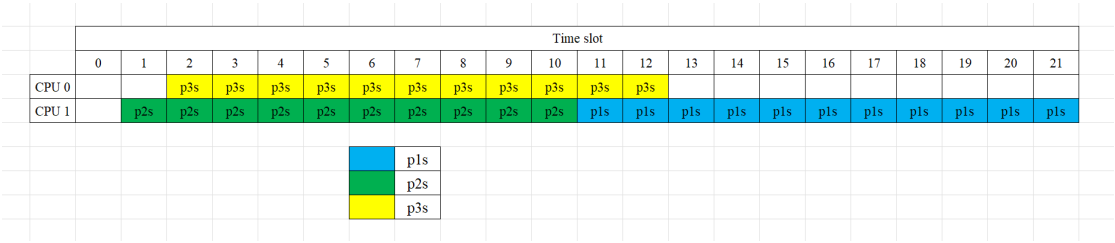
```
struct pcb_t * dequeue(struct queue_t * q) {
        /* TODO: return a pcb whose prioprity is the highest
         * in the queue [q] and remember to remove it from q
         * */
  struct pcb_t * proc = NULL;
  proc = q->proc[0];
        for(int j = 0; j < MAX_QUEUE_SIZE; j++){
               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

```
                        q->proc[j] = q->proc[j+1];
                else
                        q->proc[j] = NULL;
        }
        q->size--;
  return proc;
}
```

get_mlq_proc(): Lấy một process từ PRIORITY (ready_queue). Dưới đây là code:

```
static int slot[MAX_PRIO];
struct pcb_t * get_mlq_proc(void) {
  struct pcb_t * proc = NULL;
  /*TODO: get a process from PRIORITY [ready_queue].
   * Remember to use lock to protect the queue.
   * */
  static int i= 0;
  int check = 0;
  for (;; i = (i + 1)%MAX_PRIO) {
    if (slot[i] == 0) {
      slot[i] = MAX_PRIO - i;
      check = 0;
    continue;
    }
    if (empty(&mlq_ready_queue[i])) {
      check++;
      if (check == MAX_PRIO) break;
      continue;
    }
    check = 0;
    pthread_mutex_lock(&queue_lock);
    slot[i]--;
    proc = dequeue(&mlq_ready_queue[i]);
    pthread_mutex_unlock(&queue_lock);
    break;
  }
  return proc;
}
```

2.4.2 Biểu đồ Gantt



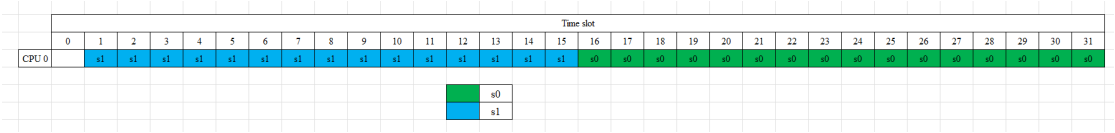
Hình 5: Biểu đồ Gantt của sched

Với giá trị đầu vào của sched là:

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

Với biểu đồ Gantt như hình trên ta thấy: CPU 1 sẽ chạy hết p2s xong sau đó chạy hết p1s, CPU 0 sẽ chỉ chạy p1s.

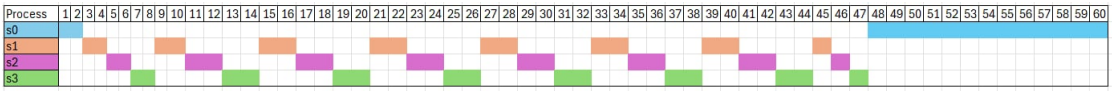
Với giá trị đầu vào của sched_0 là:



Hình 6: Biểu đồ Gantt của sched_0

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

Ta có biểu đồ Gantt như hình 6, ta thấy được: CPU 0 sẽ thực hiện tất cả s1 sau đó thực hiện s0. Với giá trị đầu vào của sched_1 là:



Hình 7: Biểu đồ Gantt của sched_1

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

Ta có biểu đồ Gantt của sched_1 như hình 7, ta thấy: Process s0 bắt đầu thực thi tại thời điểm 1 bắt đầu thời điểm 3, process s1 được chạy, sau đó 2 time slot, chạy process s2, sau đó 2 time slot, chạy process s3 và cứ thế chạy đến thời điểm 44, tại time slot 45, thực thi process s1, tại thời điểm 46 kết thúc s1 và thực thi s2, tại thời điểm 47, thực thi s3 và kết thúc s2, thời điểm 48 kết thúc s3 và chạy s0 đến khi kết thúc

3 Memory Management

Memory Management là một khái niệm cơ bản trong Hệ điều hành, liên quan đến cách hệ thống quản lý tài nguyên bộ nhớ để hỗ trợ thực thi chương trình và tối ưu hiệu suất. Nó đóng vai trò trung gian giữa phần cứng bộ nhớ (RAM, bộ nhớ phụ như SWAP) và phần mềm (các tiến trình, ứng dụng).

Nhiệm vụ của Memory Management:

+ Phân bổ bộ nhớ: Cấp phát bộ nhớ cho các tiến trình khi chúng yêu cầu.

+ Giải phóng bộ nhớ: Thu hồi lại bộ nhớ sau khi tiến trình kết thúc hoặc khi bộ nhớ không còn cần thiết.

+ Phân vùng bộ nhớ: Đảm bảo rằng các tiến trình không truy cập vùng nhớ của nhau.

+ Tối ưu hóa sử dụng bộ nhớ: Tăng hiệu quả sử dụng bộ nhớ thực (RAM) bằng cách sử dụng các cơ chế như phân trang, phân đoạn, hoặc bộ nhớ ảo.

3.1 The virtual memory mapping in each process and question

3.1.1 Virtual memory mapping

Trong mỗi quá trình (process), " Virtual memory mapping" là cách mà hệ điều hành tổ chức và quản lý không gian bộ nhớ mà quá trình đó có thể truy cập. Khái niệm này liên quan đến việc sử dụng không gian bộ nhớ ảo để mô phỏng bộ nhớ vật lý, giúp tăng hiệu suất và đơn giản hóa việc quản lý bộ nhớ. Các thành phần chính của " Virtual memory mapping" bao gồm:

- Bộ nhớ ảo (Virtual Memory): Là không gian bộ nhớ mà quá trình thấy như là bộ nhớ vật lý của nó. Bộ nhớ ảo được chia thành các phần tử gọi là trang (page), và mỗi trang có địa chỉ ảo của riêng nó.

- Bảng trang (Page Table): Là một bảng được sử dụng để ánh xạ địa chỉ ảo của mỗi trang trong bộ nhớ ảo thành địa chỉ vật lý tương ứng trong bộ nhớ vật lý. Mỗi quá trình có một bảng trang riêng, và hệ điều hành sử dụng bảng trang để điều hướng truy cập vào bộ nhớ.

- Trang (Page): Là một phần của bộ nhớ ảo, có kích thước cố định và được ánh xạ thành trang tương ứng trong bộ nhớ vật lý.

3.1.2 Trả lời câu hỏi

Câu hỏi 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?"

Dịch: Trong hệ điều hành đơn giản này, chúng ta hiện thực một thiết kế phân đoạn bộ nhớ hoặc vùng nhớ trong khai báo mã nguồn. Ưu điểm của đề xuất thiết kế nhiều phân đoạn là gì?

Trả lời: Multiple Segments trong hệ điều hành là khái niệm mô tả việc chia bộ nhớ thành nhiều phần khác nhau để phục vụ các mục đích riêng biệt. Điều này giúp tối ưu hóa quản lý tài nguyên bộ nhớ và tăng cường hiệu quả hoạt động của hệ thống.

Các phân đoạn chính trong hệ điều hành:

- Code Segment: Lưu trữ mã lệnh (instructions) của chương trình hoặc tiến trình. Đây là nơi chứa các lệnh mà CPU sẽ thực thi.
- Data Segment: Dành cho các biến toàn cục và biến tĩnh (static variables) được sử dụng trong chương trình.
- Stack Segment: Lưu trữ biến cục bộ, tham số truyền vào hàm, địa chỉ trả về, và các dữ liệu liên quan đến ngữ cảnh thực thi hoặc lời gọi hàm.
- Heap Segment: Dùng cho các biến được cấp phát động trong suốt quá trình thực thi chương trình.
- Kernel Segment: Phần dành riêng cho mã và dữ liệu của kernel. Phân đoạn này thường được bảo vệ nghiêm ngặt để đảm bảo bảo mật và tính ổn định của hệ thống.
- Shared Memory Segment: Khu vực bộ nhớ cho phép nhiều tiến trình cùng truy cập để trao đổi dữ liệu.
- File System Segment: Đại diện cho các phần khác nhau của hệ thống tệp, bao gồm các bảng inode, khối dữ liệu (data blocks), và các mục lục thư mục (directory entries).

Ưu điểm của việc sử dụng nhiều phân đoạn:

- Quản lý bộ nhớ hiệu quả: Phân chia bộ nhớ theo chức năng giúp tổ chức và quản lý tài nguyên dễ dàng hơn. Mỗi phân đoạn được tối ưu hóa để phục vụ đúng mục đích, giảm thiểu lãng phí bộ nhớ.
- Tăng cường bảo mật: Phân đoạn hóa giúp dễ dàng áp dụng các quyền truy cập như đọc, ghi, hoặc thực thi cho từng phần cụ thể. Bảo vệ dữ liệu nhạy cảm trong kernel hoặc các phần không nên bị sửa đổi.
- Tách biệt chức năng: Giúp hệ điều hành xử lý đồng thời nhiều phần của chương trình, cải thiện tính ổn định và hiệu năng của hệ thống. Hạn chế ảnh hưởng lẫn nhau giữa các phân đoạn khi có lỗi xảy ra
- Hỗ trợ quản lý bộ nhớ ảo: Mỗi phân đoạn có thể ánh xạ địa chỉ vật lý sang địa chỉ ảo theo cách riêng, giúp tối ưu hóa phân bổ bộ nhớ và dịch địa chỉ.
- Dễ dàng gỡ lỗi: Khi chương trình gặp lỗi, việc chia bộ nhớ thành các phân đoạn rõ ràng giúp xác định nhanh lỗi nằm ở phân đoạn nào, từ đó tăng tốc độ xử lý lỗi.
- Linh hoạt trong phân bổ tài nguyên: Các phân đoạn có thể tự điều chỉnh kích thước và vị trí tùy theo nhu cầu sử dụng của ứng dụng. Hỗ trợ hệ thống tối ưu hóa hiệu suất và khả năng đáp ứng khi cần xử lý nhiều tiến trình.

3.2 The system physical memory

3.2.1 Tóm tắt

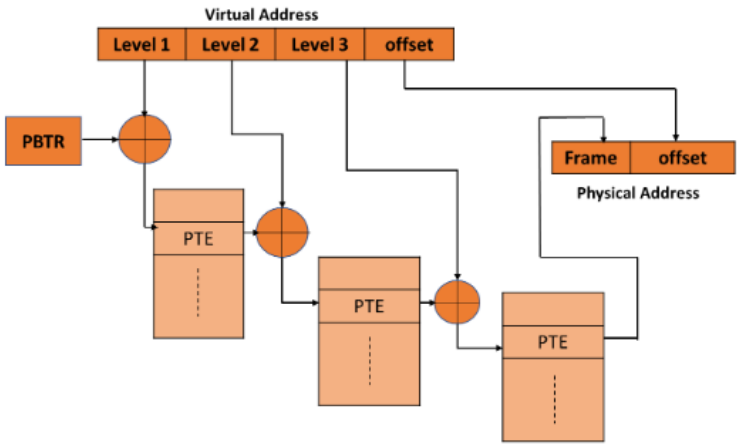
Hệ thống bộ nhớ vật lý là phần quan trọng của máy tính, dùng để lưu trữ dữ liệu và chương trình trong quá trình thực thi. Bộ nhớ vật lý bao gồm nhiều thành phần, chủ yếu là RAM (Random Access Memory), bộ nhớ cache và bộ nhớ ổ đĩa cứng.

- RAM là thành phần quan trọng nhất trong hệ thống bộ nhớ vật lý. Nó lưu trữ dữ liệu và các hướng dẫn mà CPU đang cần trong thời gian thực, giúp CPU có thể truy cập nhanh chóng khi thực thi các chương trình. RAM có tốc độ truy xuất rất nhanh, nhưng dung lượng giới hạn và dữ liệu trong RAM sẽ bị mất khi máy tính tắt nguồn.
- Bộ nhớ cache là một dạng bộ nhớ nhỏ, nhanh hơn so với RAM, được sử dụng để lưu trữ dữ liệu mà CPU truy cập thường xuyên. Cache giúp giảm thời gian truy xuất và tăng tốc độ xử lý của CPU, do CPU có thể lấy dữ liệu từ bộ nhớ cache thay vì phải truy cập RAM hoặc ổ đĩa cứng.
- Bộ nhớ ổ đĩa cứng (HDD/SSD) dùng để lưu trữ dữ liệu lâu dài và không bị mất khi máy tính tắt nguồn. Mặc dù tốc độ truy cập của bộ nhớ ổ đĩa cứng chậm hơn so với RAM và bộ nhớ cache, nhưng nó có dung lượng lớn hơn nhiều và là nơi lưu trữ hệ điều hành, chương trình và dữ liệu người dùng.

3.2.2 Trả lời câu hỏi

Câu hỏi 3: What will happen if we divide the address to more than 2-levels in the paging memory management system?

Dịch: Điều gì sẽ xảy ra nếu chúng ta chia địa chỉ thành hơn 2 cấp trong hệ thống quản lý bộ nhớ phân trang?



Hình 8: Three level paging system

Trả lời:

- **Giảm chi phí bộ nhớ:** Trong hệ thống phân trang đa cấp, các bảng trang được tổ chức thành một cấu trúc phân cấp, với mỗi cấp độ giữ ít mục hơn so với các cấp độ cao hơn. Điều này giúp giảm bộ nhớ sử dụng cho việc lưu trữ bảng trang. Các bảng trang cấp đầu tiên (thường là cấp độ đầu tiên) nằm trong bộ nhớ chính, và các bảng trang ở các cấp độ tiếp theo chỉ được đưa vào bộ nhớ khi cần thiết, điều này giúp giảm lãng phí bộ nhớ. Một trong những lợi ích chính của phương pháp phân trang đa cấp là không cần phải lưu trữ tất cả các bảng trang trong bộ nhớ cùng một lúc, giúp tối ưu hóa tài nguyên bộ nhớ của hệ thống.

- **Tăng hiệu suất tra cứu bảng trang:** Trong hệ thống phân trang đa cấp, mỗi cấp độ bảng trang sẽ chứa ít mục hơn, vì vậy thời gian để thực hiện tra cứu bảng trang sẽ giảm. Các bảng trang (ngoại trừ cấp độ cuối cùng) chứa các địa chỉ cơ sở của bảng trang cấp độ tiếp theo. Khi dịch một địa chỉ logic thành địa chỉ vật lý, hệ thống sẽ thực hiện một chuỗi tra cứu qua các cấp độ của bảng trang. Quá trình này giúp việc tra cứu trở nên nhanh chóng hơn vì mỗi cấp độ của bảng trang giữ một lượng nhỏ các mục nhập, và nhờ đó giảm được độ sâu của các tìm kiếm. Điều này không chỉ giúp tiết kiệm thời gian mà còn giảm thiểu khả năng gây tắc nghẽn trong hệ thống khi thực hiện các tra cứu bảng trang.

- **Dịch địa chỉ theo từng giai đoạn:** Phân trang đa cấp cho phép hệ thống dịch địa chỉ hiệu quả bằng cách thực hiện việc dịch theo từng giai đoạn (hoặc theo các bước dần dần). Thay vì phải tìm kiếm một bảng trang lớn và thực hiện việc tra cứu toàn bộ trong một lần, hệ thống sẽ dịch địa chỉ qua nhiều cấp độ khác nhau. Mỗi bước dịch địa chỉ này sẽ làm giảm độ phức tạp và thời gian xử lý của hệ thống. Do đó, quá trình phân giải địa chỉ logic thành địa chỉ vật lý sẽ trở nên nhanh chóng và hiệu quả hơn. Điều này cực kỳ quan trọng trong môi trường yêu cầu truy cập bộ nhớ nhanh, đặc biệt khi có số lượng lớn các trang bộ nhớ cần quản lý và truy xuất.

- **Khả năng mở rộng và thích ứng linh hoạt:** Khi kích thước của bảng trang vượt quá kích thước mong muốn hoặc khi hệ thống cần mở rộng để chứa không gian địa chỉ lớn hơn, chúng ta có thể dễ dàng thêm các cấp độ bảng trang mới. Tính linh hoạt này giúp hệ thống có thể xử lý một lượng lớn bộ nhớ mà không cần thay đổi cấu trúc cơ bản của hệ thống. Các cấp độ bổ sung có thể được tạo ra để cung cấp không gian bộ nhớ lớn hơn, cho phép hệ thống hỗ trợ các yêu cầu bộ nhớ phức tạp và thay đổi từ các chương trình ứng dụng. Điều này làm cho hệ thống phân trang đa cấp không chỉ hiệu quả mà còn cực kỳ thích ứng với các nhu cầu bộ nhớ thay đổi trong thời gian thực.

- **Khả năng xử lý yêu cầu bộ nhớ khác nhau:** Một lợi ích khác của việc sử dụng phân trang đa cấp là khả năng hỗ trợ nhiều loại yêu cầu bộ nhớ khác nhau. Các ứng dụng có thể yêu cầu bộ nhớ với kích thước và mức độ phân bổ khác nhau. Hệ thống phân trang đa cấp có thể xử lý các yêu cầu này bằng cách tự động điều chỉnh số lượng cấp độ bảng trang để đáp ứng các yêu cầu cụ thể của từng ứng dụng. Điều này giúp tối ưu hóa việc phân bổ bộ nhớ, giúp hệ thống hoạt động hiệu quả và giảm thiểu các vấn đề như phân mảnh bộ nhớ, qua đó cải thiện hiệu suất tổng thể của hệ thống.

3.3 Paging-based address translation scheme

3.3.1 Segmentation with Paging

Paging	Segmentation
Chương trình được chia thành các trang có kích thước cố định.	Chương trình được chia thành các phân đoạn có kích thước tùy biến.
Hệ điều hành chịu trách nhiệm cho việc phân trang.	Người dùng/trình biên dịch chịu trách nhiệm cho việc phân đoạn.
Tốc độ truy cập bộ nhớ của chiến lược phân trang nhanh hơn phân đoạn.	Tốc độ truy cập bộ nhớ của chiến lược phân đoạn chậm hơn phân trang.
Xuất hiện tình trạng phân mảnh trong.	Xuất hiện tình trạng phân mảnh ngoài.
Không gian địa chỉ logic được chia thành page number và page offset.	Không gian địa chỉ logic được chia thành segment number và segment offset.
Để lưu trữ trang ảo, paging yêu cầu page table.	Segmentation có mục nhập cho mỗi trạng thái ảo.
Hệ điều hành duy trì danh sách khung trang trống.	Hệ điều hành phải giữ danh sách các holes trong bộ nhớ chính.
Paging không thể nhìn thấy bởi người dùng.	Segmentation có thể nhìn thấy bởi người dùng.
Page table có số khung và các bits bảo vệ cho các trang.	Segmentation table có limit, base, và có thể chứa một vài bits để bảo vệ cho phân đoạn.

Bảng 1: Sơ lược Paging và Segmentation

Việc sử dụng tối ưu dung lượng của bộ nhớ là chủ đề nghiên cứu sâu rộng trong lĩnh vực máy tính. Trong bất kỳ máy nào, chúng ta cần chia sẻ bộ nhớ giữa nhiều chương trình. Tuy nhiên, nó có thể khiến quá trình xử lý chậm vì CPU cần tải dữ liệu vào RAM. Để giảm kích thước của bảng trang trong RAM, sử dụng chiến lược kết hợp cả phân đoạn và phân trang. Vì thế, chiến lược segmentation with paging là cần thiết.

3.3.2 Trả lời câu hỏi

Câu hỏi 4: "What is the advantage and disadvantage of segmentation with paging?"

Dịch: Ưu điểm và nhược điểm của phân đoạn với phân trang là gì?

Trả lời:

Về ưu điểm:

- Giảm thiểu việc sử dụng bộ nhớ: Phân đoạn paging giúp tối ưu hóa việc sử dụng bộ nhớ bằng cách cho phép phân bổ không liên tục, từ đó giảm thiểu lãng phí bộ nhớ.
- Kích thước bảng trang bị giới hạn bởi kích thước phân đoạn: Kích thước của bảng trang sẽ phụ thuộc vào kích thước của từng phân đoạn, giúp quản lý bộ nhớ hiệu quả hơn.
- Bảng phân đoạn chỉ có một mục tương ứng với một phân đoạn thực tế: Điều này giúp đơn giản hóa việc tra cứu và quản lý các phân đoạn trong bộ nhớ.
- Không có phân mảnh bên ngoài: Phân đoạn paging giúp loại bỏ vấn đề phân mảnh bên ngoài, vì các phân đoạn có thể được lưu trữ không liên tục mà không làm giảm hiệu quả sử dụng bộ nhớ.
- Đơn giản hóa việc phân bổ bộ nhớ: Phương pháp này giúp việc phân bổ bộ nhớ trở nên dễ dàng hơn, vì nó cho phép quản lý các phân đoạn và trang một cách hiệu quả.

Về nhược điểm:

- Sẽ có phân mảnh bên trong: Mặc dù phân đoạn paging giúp giảm thiểu phân mảnh bên ngoài, nhưng vẫn có thể xảy ra phân mảnh bên trong do kích thước trang không khớp với kích thước dữ liệu thực tế.
- Mức độ phức tạp cao hơn so với phân trang: Việc kết hợp cả phân đoạn và phân trang làm cho hệ thống trở nên phức tạp hơn, đòi hỏi nhiều quy trình và cấu trúc dữ liệu hơn để quản lý.
- Bảng trang cần được lưu trữ liên tục trong bộ nhớ: Điều này có thể gây khó khăn trong việc quản lý bộ nhớ, vì bảng trang cần phải được lưu trữ ở một vị trí liên tục, điều này có thể dẫn đến vấn đề phân mảnh.

3.4 Multiple memory segments

3.4.1 Tóm tắt

Phân đoạn bộ nhớ (Memory segmentation) là một kỹ thuật quản lý bộ nhớ của hệ điều hành, trong đó bộ nhớ chính của máy tính được chia thành các phân đoạn hoặc phần riêng biệt. Trong một hệ thống máy tính sử dụng phân đoạn, một tham chiếu đến vị trí bộ nhớ bao gồm một giá trị xác định phân đoạn và một giá trị dịch biểu thị vị trí bộ nhớ trong phân đoạn đó. Các phân đoạn hoặc phần này cũng được sử dụng trong các tệp đối tượng của các chương trình đã biên dịch, khi chúng được liên kết thành một hình ảnh chương trình và khi hình ảnh này được tải vào bộ nhớ.

Thông thường, các phân đoạn tương ứng với các phần tự nhiên của một chương trình, chẳng hạn như các đoạn mã lệnh hoặc bảng dữ liệu riêng lẻ. Do đó, phân đoạn thường hiển thị rõ ràng hơn với lập trình viên so với việc sử dụng phân trang đơn thuần. Các phân đoạn có thể được tạo cho các module chương trình hoặc cho các loại sử dụng bộ nhớ khác nhau, chẳng hạn như các phân đoạn mã và các phân đoạn dữ liệu. Một số phân đoạn nhất định có thể được chia sẻ giữa các chương trình.

Phân đoạn ban đầu được phát minh như một phương pháp để phần mềm hệ thống có thể cô lập các tiến trình phần mềm (tasks) và dữ liệu mà chúng sử dụng. Mục đích là tăng độ tin cậy của các hệ thống khi chạy đồng thời nhiều tiến trình.

3.4.2 Trả lời câu hỏi

Câu hỏi 5: What is the reason for data being separated into two different segments: data and heap?

Dịch: Lý do dữ liệu được tách thành hai phân đoạn khác nhau: data và heap là gì?

Trả lời:

Dữ liệu thường được chia thành các phân đoạn như ***data segment*** và ***heap segment*** vì nhiều lý do quan trọng dưới đây:

- Việc tổ chức và quản lý bộ nhớ trở nên dễ dàng hơn. Phân đoạn dữ liệu chứa các biến toàn cục và tĩnh được khởi tạo trước khi chương trình chạy, có kích thước và vòng đời cố định. Trong khi đó, phân đoạn heap được sử dụng để cấp phát bộ nhớ động, cho phép linh hoạt thay đổi bộ nhớ trong quá trình chương trình hoạt động.
- Việc phân chia này giúp tăng hiệu quả sử dụng bộ nhớ. Phân đoạn dữ liệu có kích thước cố định và được biết trước tại thời điểm biên dịch, giúp giảm thiểu phân mảnh bộ nhớ. Ngược lại, heap hỗ trợ các nhu cầu bộ nhớ động, phù hợp với các chương trình có yêu cầu thay đổi, dù điều này có thể gây phân mảnh nếu không được quản lý tốt.
- Việc tách biệt phân đoạn dữ liệu giúp bảo vệ các biến toàn cục và tĩnh khỏi bị ghi đè bởi bộ nhớ động. Đồng thời, phân đoạn heap riêng biệt giúp ngăn ngừa các lỗi tràn bộ đệm hoặc lỗi bộ nhớ khác ảnh hưởng đến các biến quan trọng.
- Truy cập vào phân đoạn dữ liệu thường nhanh hơn vì địa chỉ bộ nhớ của các biến đã cố định. Mặc dù truy cập heap có thể chậm hơn do yêu cầu cấp phát và giải phóng bộ nhớ, nhưng nó hỗ trợ sử dụng tài nguyên linh hoạt và hiệu quả hơn trong các trường hợp phức tạp.
- Các biến toàn cục và tĩnh nằm trong một phân đoạn riêng biệt, giúp dễ dàng xác định lỗi. Bộ nhớ động trong heap cũng có thể được theo dõi riêng, hỗ trợ việc phát hiện và sửa các lỗi như rò rỉ bộ nhớ.

4 The status of RAM

Ta chạy thử với input có bộ nhớ RAM lớn như sau:

```
6 2 4
1048576 16777216 0 0 0 3145728
0 p0s 0
2 p1s 15
```

- Time slot: 6
- Số lượng CPU: 2
- Số lượng tiến trình: 4
- Bộ nhớ RAM: 1048576
- 4 bộ nhớ SWAP lần lượt là: 16777216, 0, 0, 0
- Bộ nhớ ảo: 3145728
- Tiến trình 1: ArrivalTime = 0, Name = p0s, prio = 0;
- Tiến trình 2: ArrivalTime = 2, Name = p1s, prio = 15;

Với process p0s và p1s như bên dưới:

```
// process p0s
1 14
calc
alloc 300 0
alloc 300 4
free 0
alloc 100 1
write 100 1 20
read 1 20 20
write 102 2 20
read 2 20 20
write 103 3 20
read 3 20 20
```

```
14 calc
15 free 4
16 calc
17
18 //process p1s
19 1 10
20 calc
21 calc
22 calc
23 calc
24 calc
25 calc
26 calc
27 calc
28 calc
29 calc
30 calc
```

Kết quả thu được:

```
1  Time slot   0
2  ld_routine
3          Loaded a process at input/proc/p0s, PID: 1 PRIO: 0
4  Time slot   1
5          CPU 1: Dispatched process  1
6  Time slot   2
7          Loaded a process at input/proc/p1s, PID: 2 PRIO: 15
8  -------------------------
9  Status of memory allocated - process 1
10 ----------------------------
11 Virtual Area:
12 print_list_vma:
13 va[0->512]
14
15 ----------------------------
16 Allocated region:
17 Region id: 0, range [0 -> 300]
18 ----------------------------
```

```
Free Region List:
print_list_rg:
rg[300->512]
null
-------------------------
List Page:
print_list_pgn:
va[1]->va[0]->null
----------------------------
Used Frames List:
print_list_fp:
fp[1]->fp[0]->null
----------------------------
        CPU 0: Dispatched process  2
Time slot   3
-------------------------
Status of memory allocated - process 1
----------------------------
Virtual Area:
print_list_vma:
va[0->1024]

----------------------------
Allocated region:
Region id: 0, range [0 -> 300]
Region id: 4, range [512 -> 812]
----------------------------
Free Region List:
print_list_rg:
rg[812->1024]
rg[300->512]
null
-------------------------
List Page:
print_list_pgn:
va[3]->va[2]->va[1]->va[0]->null
----------------------------
```

```
Used Frames List:
print_list_fp:
fp[3]->fp[2]->fp[1]->fp[0]->null
----------------------------
Time slot   4
----------------------
The regions are decompressed:
Free Region id: 0 range [0 -> 300] of process 1
----------------------
Time slot   5
-------------------------
Status of memory allocated - process 1
----------------------------
Virtual Area:
print_list_vma:
va[0->1024]

----------------------------
Allocated region:
Region id: 1, range [0 -> 100]
Region id: 4, range [512 -> 812]
----------------------------
Free Region List:
print_list_rg:
rg[100->300]
rg[812->1024]
rg[300->512]
null
-------------------------
List Page:
print_list_pgn:
va[3]->va[2]->va[1]->va[0]->null
----------------------------
Used Frames List:
print_list_fp:
fp[3]->fp[2]->fp[1]->fp[0]->null
----------------------------
```

```
Time slot   6
write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
Time slot   7
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000014: 64000000
Time slot   8
        CPU 0: Put process  2 to run queue
        CPU 0: Dispatched process  2
write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000014: 64000000
Time slot   9
read region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000014: 66000000
Time slot  10
write region=3 offset=20 value=103
```

```
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000014: 66000000
Time slot  11
read region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000014: 67000000
Time slot  12
        CPU 0: Processed  2 has finished
        CPU 0 stopped
Time slot  13
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  1
----------------------
The regions are decompressed:
Free Region id: 4 range [512 -> 812] of process 1
----------------------
Time slot  14
Time slot  15
        CPU 1: Processed  1 has finished
        CPU 1 stopped
```

Giải thích kết quả:

- Các lệnh Calc: Chương trình không làm gì.
- Lệnh Alloc 300 0 của tiến trình p0s: Nghĩa là cấp phát một vùng nhớ 300 đơn vị có id là 0. Kết quả hình ảnh trong bộ nhớ ảo sẽ là vùng vm_area[0->512] và được chia ra thành 2 trang va[1]->va[0], các frames trong bộ nhớ vật lý đã sử dụng là fp[1]->fp[0]. Vùng nhớ đã cấp phát là [0->300] có id là 0 và vùng còn trống là [300->512].

```
1  ----------------------------
2  Status of memory allocated - process 1
3  ----------------------------
4  Virtual Area:
5  print_list_vma:
6  va[0->512]
7
8  ----------------------------
9  Allocated region:
10 Region id: 0, range [0 -> 300]
11 ----------------------------
12 Free Region List:
13 print_list_rg:
14 rg[300->512]
15 null
16 -------------------------
17 List Page:
18 print_list_pgn:
19 va[1]->va[0]->null
20 ----------------------------
21 Used Frames List:
22 print_list_fp:
23 fp[1]->fp[0]->null
24 ----------------------------
25
```

- Lệnh Alloc 300 4 của tiến trình p0s: Nghĩa là cấp phát một vùng nhớ 300 đơn vị có id là 4. Lúc này vùng trống không đủ cho cấp phát nên sẽ được mở rộng lên thành [0->1024]. Vùng nhớ được cấp phát là [512->812] có id 4 như bên dưới và danh sách các vùng còn trống, các frames đã sử dụng.

```
1  -------------------------
2  Status of memory allocated - process 1
3  ----------------------------
4  Virtual Area:
5  print_list_vma:
```

```
va[0->1024]

----------------------------
Allocated region:
Region id: 0, range [0 -> 300]
Region id: 4, range [512 -> 812]
----------------------------
Free Region List:
print_list_rg:
rg[812->1024]
rg[300->512]
null
--------------------------
List Page:
print_list_pgn:
va[3]->va[2]->va[1]->va[0]->null
----------------------------
Used Frames List:
print_list_fp:
fp[3]->fp[2]->fp[1]->fp[0]->null
----------------------------

```

- Lệnh free 0 của tiến trình p0s: Nghĩa là giải phóng vùng nhớ có id là 0. Vùng nhớ [0->300] đã được giải phóng.

```
----------------------
The regions are decompressed:
Free Region id: 0 range [0 -> 300] of process 1
----------------------

```

- Lệnh Alloc 100 0 của tiến trình p0s: Tương tự ở trên. Lần này vùng nhớ [0->300] đã được giải phóng nên vùng này sẽ được cấp phát lại cho câu lệnh trên. Kết quả là vùng nhớ [0->100] đã được sử dụng và danh sách các vùng trống như bên dưới.

```
-------------------------
```

```
Status of memory allocated - process 1
----------------------------
Virtual Area:
print_list_vma:
va[0->1024]

----------------------------
Allocated region:
Region id: 1, range [0 -> 100]
Region id: 4, range [512 -> 812]
----------------------------
Free Region List:
print_list_rg:
rg[100->300]
rg[812->1024]
rg[300->512]
null
-------------------------
List Page:
print_list_pgn:
va[3]->va[2]->va[1]->va[0]->null
----------------------------
Used Frames List:
print_list_fp:
fp[3]->fp[2]->fp[1]->fp[0]->null
----------------------------

```

- Lệnh write 100 1 20: Nghĩa là ghi giá trị 100 vào region 1, offset 20. 100 ở hệ hex là 64. Tương tự với các lệnh write bên dưới sẽ ghi đè lên vị trí này.

```
write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
```

```
Time slot   7
         CPU 1: Put process  1 to run queue
         CPU 1: Dispatched process  1
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000014: 64000000

```

Ta thử chạy với bộ nhớ RAM nhỏ:

```
2 4 8
2048 16777216 0 0 0 3145728
1 p0s  130
2 s3  39
4 m1s  15
6 s2  120
7 m0s  120
9 p1s  15
11 s0 38
16 s1 0

```

Kết quả thu được:

```
Time slot   0
ld_routine
Time slot   1
         Loaded a process at input/proc/p0s, PID: 1 PRIO: 130
         CPU 3: Dispatched process  1
Time slot   2
--------------------------
Status of memory allocated - process 1
---------------------------
Virtual Area:
```

```
print_list_vma:
va[0->512]

----------------------------
Allocated region:
Region id: 0, range [0 -> 300]
----------------------------
Free Region List:
print_list_rg:
rg[300->512]
null
-------------------------
List Page:
print_list_pgn:
va[1]->va[0]->null
----------------------------
Used Frames List:
print_list_fp:
fp[1]->fp[0]->null
----------------------------
        Loaded a process at input/proc/s3, PID: 2 PRIO: 39
Time slot   3
        CPU 3: Put process  1 to run queue
        CPU 3: Dispatched process  2
        CPU 2: Dispatched process  1
-------------------------
Status of memory allocated - process 1
----------------------------
Virtual Area:
print_list_vma:
va[0->1024]

----------------------------
Allocated region:
Region id: 0, range [0 -> 300]
Region id: 4, range [512 -> 812]
----------------------------
```

```
Free Region List:
print_list_rg:
rg[812->1024]
rg[300->512]
null
--------------------------
List Page:
print_list_pgn:
va[3]->va[2]->va[1]->va[0]->null
----------------------------
Used Frames List:
print_list_fp:
fp[3]->fp[2]->fp[1]->fp[0]->null
----------------------------
Time slot   4
----------------------
The regions are decompressed:
Free Region id: 0 range [0 -> 300] of process 1
----------------------
        Loaded a process at input/proc/m1s, PID: 3 PRIO: 15
Time slot   5
        CPU 3: Put process  2 to run queue
        CPU 3: Dispatched process  3
--------------------------
Status of memory allocated - process 3
----------------------------
Virtual Area:
print_list_vma:
va[0->512]

----------------------------
Allocated region:
Region id: 3, range [0 -> 300]
----------------------------
Free Region List:
print_list_rg:
rg[300->512]
```

```
null
-------------------------
List Page:
print_list_pgn:
va[1]->va[0]->null
----------------------------
Used Frames List:
print_list_fp:
fp[5]->fp[4]->fp[3]->fp[2]->fp[1]->fp[0]->null
----------------------------
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  2
        CPU 1: Dispatched process  1
-------------------------
Status of memory allocated - process 1
----------------------------
Virtual Area:
print_list_vma:
va[0->1024]

----------------------------
Allocated region:
Region id: 1, range [0 -> 100]
Region id: 4, range [512 -> 812]
----------------------------
Free Region List:
print_list_rg:
rg[100->300]
rg[812->1024]
rg[300->512]
null
-------------------------
List Page:
print_list_pgn:
va[3]->va[2]->va[1]->va[0]->null
----------------------------
Used Frames List:
```

```
print_list_fp:
fp[5]->fp[4]->fp[3]->fp[2]->fp[1]->fp[0]->null
----------------------------
Time slot   6
--------------------------
Status of memory allocated - process 3
----------------------------
Virtual Area:
print_list_vma:
va[0->512]

----------------------------
Allocated region:
Region id: 3, range [0 -> 300]
Region id: 4, range [300 -> 400]
----------------------------
Free Region List:
print_list_rg:
rg[400->512]
null
--------------------------
List Page:
print_list_pgn:
va[1]->va[0]->null
----------------------------
Used Frames List:
print_list_fp:
fp[5]->fp[4]->fp[3]->fp[2]->fp[1]->fp[0]->null
----------------------------
write region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
        Loaded a process at input/proc/s2, PID: 4 PRIO: 120
Time slot   7
```

```
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  3
----------------------
The regions are decompressed:
Free Region id: 0 range [0 -> 0] of process 3
----------------------
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
        CPU 1: Put process  1 to run queue
        CPU 1: Dispatched process  4
        CPU 0: Dispatched process  1
read region=1 offset=20 value=100
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000014: 64000000
        Loaded a process at input/proc/m0s, PID: 5 PRIO: 120
Time slot   8
--------------------------
Status of memory allocated - process 3
---------------------------
Virtual Area:
print_list_vma:
va[0->512]

---------------------------
Allocated region:
Region id: 3, range [0 -> 300]
Region id: 4, range [300 -> 400]
Region id: 5, range [400 -> 500]
---------------------------
Free Region List:
print_list_rg:
rg[500->512]
null
```

```
-------------------------
List Page:
print_list_pgn:
va[1]->va[0]->null
----------------------------
Used Frames List:
print_list_fp:
fp[5]->fp[4]->fp[3]->fp[2]->fp[1]->fp[0]->null
----------------------------
write region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000014: 64000000
Time slot   9
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  3
----------------------
The regions are decompressed:
Free Region id: 2 range [0 -> 0] of process 3
----------------------
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  2
        CPU 1: Put process  4 to run queue
        CPU 1: Dispatched process  5
-------------------------
Status of memory allocated - process 5
----------------------------
Virtual Area:
print_list_vma:
va[0->512]

----------------------------
Allocated region:
Region id: 6, range [0 -> 300]
```

```
---------------------------
Free Region List:
print_list_rg:
rg[300->512]
null
-------------------------
List Page:
print_list_pgn:
va[1]->va[0]->null
---------------------------
Used Frames List:
print_list_fp:
fp[7]->fp[6]->fp[5]->fp[4]->fp[3]->fp[2]->fp[1]->fp[0]->null
---------------------------
        CPU 0: Put process  1 to run queue
        CPU 0: Dispatched process  4
        Loaded a process at input/proc/p1s, PID: 6 PRIO: 15
Time slot  10
----------------------
The regions are decompressed:
Free Region id: 1 range [0 -> 0] of process 3
----------------------
-------------------------
Status of memory allocated - process 5
---------------------------
Virtual Area:
print_list_vma:
va[0->512]

---------------------------
Allocated region:
Region id: 6, range [0 -> 300]
Region id: 7, range [300 -> 400]
---------------------------
Free Region List:
print_list_rg:
rg[400->512]
```

```
null
-------------------------
List Page:
print_list_pgn:
va[1]->va[0]->null
----------------------------
Used Frames List:
print_list_fp:
fp[7]->fp[6]->fp[5]->fp[4]->fp[3]->fp[2]->fp[1]->fp[0]->null
----------------------------
Time slot  11
        CPU 3: Put process  3 to run queue
        CPU 3: Dispatched process  6
        CPU 2: Put process  2 to run queue
        CPU 2: Dispatched process  3
----------------------
The regions are decompressed:
Free Region id: 0 range [0 -> 0] of process 3
----------------------
        CPU 1: Put process  5 to run queue
        CPU 1: Dispatched process  2
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  5
----------------------
The regions are decompressed:
Free Region id: 0 range [0 -> 0] of process 5
----------------------
        Loaded a process at input/proc/s0, PID: 7 PRIO: 38
Time slot  12
----------------------
The regions are decompressed:
Free Region id: 0 range [0 -> 0] of process 3
----------------------
-------------------------
Status of memory allocated - process 5
----------------------------
Virtual Area:
```

```
print_list_vma:
va[0->512]

----------------------------
Allocated region:
Region id: 6, range [0 -> 300]
Region id: 7, range [300 -> 400]
Region id: 8, range [400 -> 500]
----------------------------
Free Region List:
print_list_rg:
rg[500->512]
null
-------------------------
List Page:
print_list_pgn:
va[1]->va[0]->null
----------------------------
Used Frames List:
print_list_fp:
fp[7]->fp[6]->fp[5]->fp[4]->fp[3]->fp[2]->fp[1]->fp[0]->null
----------------------------
Time slot  13
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
        CPU 2: Processed  3 has finished
        CPU 2: Dispatched process  7
        CPU 1: Put process  2 to run queue
        CPU 1: Dispatched process  2
        CPU 0: Put process  5 to run queue
        CPU 0: Dispatched process  4
Time slot  14
        CPU 1: Processed  2 has finished
        CPU 1: Dispatched process  5
write region=1 offset=20 value=102
print_pgtbl: 0 - 512
00000000: 80000007
```

```
00000004: 80000006
00000014: 66000000
Time slot  15
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  6
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  7
write region=2 offset=1000 value=1
print_pgtbl: 0 - 512
00000000: 80000007
00000004: 80000006
00000014: 66000000
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  16
        CPU 1: Put process  5 to run queue
        CPU 1: Dispatched process  5
write region=0 offset=0 value=0
print_pgtbl: 0 - 512
00000000: c0000000
00000004: 80000006
00000014: 66000000
000000e8: 01000000
        Loaded a process at input/proc/s1, PID: 8 PRIO: 0
Time slot  17
        CPU 3: Put process  6 to run queue
        CPU 3: Dispatched process  8
Ram is full, swapping page!
Couldn't find the local victim - Out of memory
Process 8: hasn't allocated
        CPU 2: Put process  7 to run queue
        CPU 2: Dispatched process  6
        CPU 1: Processed  5 has finished
        CPU 1: Dispatched process  7
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  18
```

```
Time slot  19
        CPU 3: Put process  8 to run queue
        CPU 3: Dispatched process  8
        CPU 2: Put process  6 to run queue
        CPU 2: Dispatched process  6
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 0: Put process  4 to run queue
        CPU 0: Dispatched process  4
Time slot  20
Time slot  21
        CPU 3: Put process  8 to run queue
        CPU 3: Dispatched process  8
        CPU 2: Processed  6 has finished
        CPU 2: Dispatched process  1
read region=2 offset=20 value=102
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000014: 66000000
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
        CPU 0: Processed  4 has finished
        CPU 0 stopped
Time slot  22
write region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000014: 66000000
Time slot  23
        CPU 3: Put process  8 to run queue
        CPU 3: Dispatched process  8
```

```
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  1
read region=3 offset=20 value=103
print_pgtbl: 0 - 1024
00000000: 80000001
00000004: 80000000
00000008: 80000003
00000012: 80000002
00000014: 67000000
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  24
        CPU 3: Processed  8 has finished
        CPU 3 stopped
Time slot  25
        CPU 2: Put process  1 to run queue
        CPU 2: Dispatched process  1
----------------------
The regions are decompressed:
Free Region id: 4 range [512 -> 812] of process 1
----------------------
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  26
Time slot  27
        CPU 2: Processed  1 has finished
        CPU 2 stopped
        CPU 1: Put process  7 to run queue
        CPU 1: Dispatched process  7
Time slot  28
        CPU 1: Processed  7 has finished
        CPU 1 stopped

```

Giải thích kết quả:

- Với bộ nhớ RAM = 2048, thì số lượng frames tối đa là 8 frames (mỗi frame 256). Nên ở thời điểm bộ nhớ RAM đã đầy. Thay trang sẽ xảy ra theo chế độ FIFO cục bộ, nghĩa là sẽ

tìm các trang đã cấp phát cho chính tiến trình cần trang để thay ra theo FIFO chứ không được thay các trang của tiến trình khác. Ở đây, vì process 8 được nạp vào lần đầu và bộ nhớ đã đầy, nên sẽ không tìm được nạn nhân để thay ra, process 8 không được cấp phát vùng nhớ. Các lệnh tương tự đã giải thích ở trên.

```
1 Ram is full, swapping page!
2 Couldn't find the local victim - Out of memory
3 Process 8: hasn't allocated
4
```

5 Put It All Together

Câu hỏi 6: What will happen 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.

Dịch: Điều gì sẽ xảy ra nếu việc đồng bộ hóa không được xử lý trong hệ điều hành đơn giản của bạn? Hãy minh họa vấn đề của hệ điều hành đơn giản của bạn bằng ví dụ nếu có.

Trả lời:

Nếu một hệ điều hành đơn giản không xử lý tốt vấn đề đồng bộ hóa, nó có thể dẫn đến nhiều hậu quả nghiêm trọng như điều kiện tranh chấp (race condition), dữ liệu không nhất quán, và thậm chí là tình trạng khóa chết (deadlock).

Giả sử hệ điều hành cho phép hai tiến trình đồng thời truy cập và thay đổi một biến chia sẻ mà không có cơ chế đồng bộ hóa. Hãy xét một tình huống trong đó:

- Process A thực hiện tăng biến đếm (count) với: count = count + 1.
- Process B thực hiện giảm biến đếm với mã lệnh: count = count - 1.

Cả hai tiến trình làm việc với biến count, ban đầu được khởi tạo giá trị bằng 0.

Do thiếu đồng bộ hóa, trình tự thực hiện sau có thể xảy ra:

- Process A đọc giá trị hiện tại của count = 0 vào thanh ghi cục bộ.
- Process B cũng đọc giá trị hiện tại của count = 0 vào thanh ghi cục bộ.
- Process A tăng giá trị trong thanh ghi cục bộ của mình lên 1, tức là 0 + 1 = 1.
- Process B giảm giá trị trong thanh ghi cục bộ của mình xuống -1, tức là 0 - 1 = -1.
- Process A ghi lại giá trị cập nhật của nó là 1 vào biến đếm.
- Process B ghi lại giá trị cập nhật của nó -1 vào biến đếm.

Kết quả cuối cùng là giá trị của count là -1, trong khi đúng ra phải là 0 nếu các thao tác được thực hiện theo cách tuần tự. Đây là một ví dụ điển hình của tình trạng tranh chấp, nơi kết quả của chương trình phụ thuộc vào thứ tự và sự xen kẽ giữa các thao tác đồng thời.

Nguy cơ khóa chết:

Bên cạnh vấn đề tranh chấp, việc thiếu cơ chế đồng bộ hóa cũng có thể dẫn đến khóa chết (deadlock). Khóa chết xảy ra khi hai hoặc nhiều tiến trình chờ đợi nhau để giải phóng tài nguyên, tạo ra một vòng tròn phụ thuộc vô tận. Nếu hệ điều hành không cung cấp cách kiểm soát quyền truy cập độc quyền vào tài nguyên chia sẻ, các tiến trình có thể rơi vào tình huống tất cả đều chờ đợi mà không tiến triển, dẫn đến hệ thống bị đình trệ.

Đồng bộ hóa là một phần thiết yếu trong quản lý hệ thống. Nó không chỉ đảm bảo tính nhất quán của dữ liệu mà còn giúp tránh được các tình trạng nguy hiểm như khóa chết hoặc lỗi do tranh chấp trong các ứng dụng đa nhiệm.

6 Mã nguồn bài báo cáo

Source code: `https://github.com/leduccuonghcmut/OS-Assignment`

7 Kết luận

Trong bài tập lớn môn Hệ điều hành này, nhóm chúng em đã hoàn thành việc mô phỏng một hệ điều hành với các hoạt động quan trọng như định thời, quản lý bộ nhớ đồng bộ hóa. Đây đều là những thành phần cốt lõi, đóng vai trò quan trọng trong việc vận hành và quản lý tài nguyên của một hệ điều hành thực tế.

Quá trình thực hiện bài tập lớn đã giúp chúng em hiểu rõ hơn về các nguyên lý và nguyên tắc cơ bản trong từng hoạt động của hệ điều hành. Đặc biệt, việc xây dựng mô phỏng này không chỉ giúp chúng em củng cố kiến thức lý thuyết đã học trên lớp mà còn mang lại cơ hội thực hành, áp dụng các khái niệm đó vào thực tiễn. Qua từng bước thiết kế và thực hiện, chúng em đã nắm bắt được cách hoạt động của các cơ chế như lập lịch CPU, quản lý phân bổ bộ nhớ, và cách đảm bảo đồng bộ hóa giữa các tiến trình, từ đó có cái nhìn toàn diện hơn về cách hệ điều hành quản lý tài nguyên và xử lý các yêu cầu từ người dùng.

Tuy nhiên, trong quá trình thực hiện bài tập lớn, nhóm cũng gặp không ít khó khăn và thách thức, đặc biệt là khi xử lý các thuật toán phức tạp hoặc mô phỏng các tình huống thực tế. Dù đã cố gắng tìm hiểu, phân tích và thực hiện một cách cẩn thận, nhưng chắc chắn không thể tránh khỏi những sai sót hoặc thiếu sót trong bài làm. Vì vậy, nhóm rất mong nhận được sự góp ý, nhận xét từ thầy và các bạn để bài báo cáo được hoàn thiện hơn, đồng thời giúp nhóm học hỏi thêm nhiều kinh nghiệm và kiến thức.

Tài liệu tham khảo

[1] Abraham Silberschatz, Greg Gagne, Peter B. Galvin. *Operating System Concepts*, 10th Edition. Wiley, 2018.

[2] Documentation of Linux Kernel Scheduling. [Online]. Available: `https://www.kernel.org/doc/`

[3] Multilevel Paging in Operating System. [Online]. Available: `https://www.naukri.com/code360/library/multilevel-paging-in-operating-system`

[4] Memory segmentation. [Online]. Available: `https://en.wikipedia.org/wiki/Memory_segmentation`