操作系统实验报告

学 生 姓 名　　苏靖博
学　　号　　20105050110
班　　级　　计实验20

实验名称	进程管理实验	实验序号	1
实验日期	Sep. 29 2022	实验人	苏靖博

一、实验目的和要求

1. 通过对进程调度算法的模拟，进一步理解进程的基本概念
2. 加深对进程运行状态和进程过程、调度算法的理解
3. 进行模拟的调度算法有:先来先服务调度算法 (First Come First Serve - FCFS,时间片轮转法调度算法 (Round Robin - RR)和动态优先级调度算法 (Preemptive Priority - PP)

二、相关背景知识

1. 了解相关OS作业调度算法 Scheduling:先来先服务调度算法 (FCFS)、时间片轮转法调度算法 (RR)和动态优先级调度算法;
2. 基础C language编程技巧 (pointer, etc.);
3. 熟悉Linux 环境及常用工具操作技巧 (gcc, gdb, vim, tmux, etc.).

三、实验内容

1. 用C语言实现对N个进程采用某种进程调度算法（如先来先服务调度、时间片轮转调度、动态优先级调度）的调度;
2. 为了清楚地观察每个进程的调度过程，程序应将每个进程的被调度情况显示出来;
3. 分析程序运行的结果，并谈一下自己的收获.

四、关键数据结构与函数的说明

Process:

```
typedef struct Process {
  int pid;
  int arrival_time;
  int execution_time;
  int wait_time;
  int priority;
} process_t;

void initialize(); // initialize the process array
```

Queue:

```
typedef struct queue {
  struct queue* next;
  struct queue* prev;
  process_t pcb;
} *queue_t;

queue_t create(); // create a queue based on double-end linkedlist, and return the head of the list
void clear(queue_t head); // clear the queue and release its memory
int is_empty(queue_t head); // return 1 if the queue is empty, and 0
```

otherwise
`void append(process_t x, queue_t head);` // append an element to the back of the queue
`process_t pop(queue_t head);` // pop the top element of the queue and return it
`void await(queue_t head);` // print the status of the queue now

(Min) Heapq:

`void push(process_t x, int t);` // push an element into heap at t
`void pull(int x);` // order the from x in heap
`process_t popq(int t);` // pop the top (minium) element at t and return it
`int update();` // update the priority of elements in the heap and return the top (minium) element at the same time

五、编译与执行过程截图

USE **`gcc -g main.c -Wall -Werror -std=c11 && ./a.out fcfs`** to test the First Come First Serve scheduling ONLY
OR **`gcc -g main.c -Wall -Werror -std=c11 && ./a.out rr`** to test the Round Robin scheduling ONLY
OR **`gcc -g main.c -Wall -Werror -std=c11 && ./a.out pp`** to test the Preemptive Priority scheduling ONLY
OR **`gcc -g main.c -Wall -Werror -std=c11 && ./a.out`** can test all three schedulings together with the same series of processes.

Experiments:
5 Processes, using the last compiling instruction:

```
PID AT    ET    PRI

1  0      6     6
2  6      3     9
3  8      1     7
4  5      7     8
5  4      5     6
---------------------------

[FCFS Running...]
Now PID
1   1
2   1
3   1
4   1
5   1
6   1
[Context Switching! Process 5 has waited 2 s]
7   5
8   5
9   5
10  5
11  5
[Context Switching! Process 4 has waited 6 s]
12  4
13  4
14  4
```

```
15  4
16  4
17  4
18  4
[Context Switching! Process 2 has waited 12 s]
19  2
20  2
21  2
[Context Switching! Process 3 has waited 13 s]
PID Waiting Time
1   0
2   12
3   13
4   6
5   2
=====================================
Total Waiting Time: 33s
Average Waiting Time: 6.60s


[Round Robin Running...]
Now: 0 s, PID: 1, [Queue]: No element in the queue...
Now: 1 s, PID: 1, [Queue]: No element in the queue...
Now: 2 s, PID: 1, [Queue]: No element in the queue...
Now: 3 s, PID: 1, [Queue]: No element in the queue...
Now: 4 s, PID: 5, [Queue]: 1
Now: 5 s, PID: 5, [Queue]: 1 4
Now: 6 s, PID: 1, [Queue]: 4 2 5
Now: 7 s, PID: 1, [Queue]: 4 2 5
Now: 8 s, PID: 4, [Queue]: 2 5 3
Now: 9 s, PID: 4, [Queue]: 2 5 3
Now: 10 s, PID: 2, [Queue]: 5 3 4
Now: 11 s, PID: 2, [Queue]: 5 3 4
Now: 12 s, PID: 5, [Queue]: 3 4 2
Now: 13 s, PID: 5, [Queue]: 3 4 2
Now: 14 s, PID: 3, [Queue]: 4 2 5
Now: 15 s, PID: 4, [Queue]: 2 5
Now: 16 s, PID: 4, [Queue]: 2 5
Now: 17 s, PID: 2, [Queue]: 5 4
Now: 18 s, PID: 5, [Queue]: 4
Now: 19 s, PID: 4, [Queue]: No element in the queue...
Now: 20 s, PID: 4, [Queue]: No element in the queue...
Now: 21 s, PID: 4, [Queue]: No element in the queue...
PID Waiting Time
1   22
2   6
3   10
4   9
5   2
=====================================
Total Waiting Time: 49s
Average Waiting Time: 9.80s


[Preemptive Priority Running...]
Now: 0 s, PID: 1
Now: 1 s, PID: 1
Now: 2 s, PID: 1
Now: 3 s, PID: 1
Now: 4 s, PID: 1
Now: 5 s, PID: 5
Now: 6 s, PID: 5
Now: 7 s, PID: 1
Now: 7 s, PID: 5
Now: 8 s, PID: 5
Now: 9 s, PID: 4
Now: 10 s, PID: 4
Now: 11 s, PID: 5
Now: 11 s, PID: 4
Now: 12 s, PID: 3
Now: 12 s, PID: 2
Now: 13 s, PID: 2
Now: 14 s, PID: 4
Now: 15 s, PID: 4
```

```
Now: 16 s, PID: 2
Now: 16 s, PID: 4
Now: 17 s, PID: 4
PID Waiting Time
1   0
2   4
3   9
4   9
5   3
====================================
Total Waiting Time: 25s
Average Waiting Time: 5.00s
```

六、实验结果与分析

1. First Come First Serve Scheduling拥有最简单的实现方式(FIFO)，然而其策略也导致了它不那么efficient，会有更长的waiting time。通常FCFS的waiting time并不稳定，会根据process的生成情况变化；
2. 一般地，Round Robin Scheduling的waiting time较长，在现实情况中多次的context switching(出入queue)会造成大量时间开销；抢占式(Preemptive)策略效率会受最大CPU time slice影响，因此可能会有更长的waiting time和responce time，还有较低的吞吐量（throughput）；
3. 通常情况下，Preemptive Priority Scheduling的average waiting time最短，通过堆进行priority ordering能够优化CPU对process的处理顺序及效率，使每个process都有机会执行，能够提高throughput并减少每个process的average waiting time，因此效率较高。特别地，如果两个随机生成的process的priority相等，那么该策略就会转换为FCFS Scheduling，使得平均效率降低。

七、调试时遇到的问题及解决方法

1. 连续执行后waiting time忘记清零

   解决方法: 每次执行任务前及时清零计数器

2. 死循环

   通过打断点+输出调试找到死循环的位置进行修正

3. malloc abort

   对malloc结构的格式进行检查

4. segmentation fault

   常见的问题，一样通过不断注释+输出调试找到出现段错误的位置加以修正

八、调试后的程序源代码

🔗 https://github.com/Sue217/NCUT/tree/main/OS/lab1

九、实验体会

1. malloc时要时刻注意内容及大小是否符合要求
2. 出现死循环或段错误要保持头脑冷静清晰，不断注释打断点找到出现问题的位置，一般情况下出现类似问题的原因较为固定，经过不断coding和debug会逐渐培养出对此类问题的判断和观察力
3. 将多个任务分开写便于分别调试，最后再将其合并在一起，不至于在整体编写时出现混乱的情况