浙江财经大学

实 验（实训）报 告

项 目 名 称 进程管理

所属课程名称 操作系统

项 目 类 型 验证/设计型

实验(实训)日期 2024.10.16

班 级 22 软件 2 班

学 号 220110900732

姓 名 谢柔吟

指导教师 陈伟锋

浙江财经大学教务处制

一、实验（实训）概述：

【目的及要求】

安装 gcc 并完成 hello.c、fork.c、proceses.c、communication.c 这四个程序。
用 C 语言进行进程调度模拟，实现对 N 个进程采用动态优先权算法的调度。

【基本原理】

1.编译和运行程序：
在命令行中，导航到每个.c 文件所在的目录，并使用 gcc 命令编译它们，例如 gcc hello.c -o hello。
运行编译生成的可执行文件，例如./hello。

2.进程调度模拟：
动态优先权算法根据进程的等待时间和服务时间来调整其优先级。
定义一个进程控制块（PCB）来存储每个进程的信息，实现用调度器来管理 PCB。

【实施环境】

VMware Workstation 17 Pro
Ubuntu 24.04.1

二、实验（实训）内容：

【实验（实训）过程】（步骤、记录、数据、程序等）

1. 安装 GCC

更新包列表 sudo apt update
安装 GCC sudo apt install build-essential
（build-essential 包包含了 GCC 以及编译 C 程序所需的其他工具。）

2.完成 xxx.c

创建：nano xxx.c
编译：gcc xxx.c -o a(可执行文件的名称)
运行：./a

（1）完成 hello.c

```
soft@soft:~/桌面/C$ gcc hello.c -o hello
soft@soft:~/桌面/C$ ./hello
Hello, World!
```

（2）完成 fork.c

```
soft@soft:~/桌面/C$ gcc fork.c -o fork
fork.c: In function ‘main’:
fork.c:18:9: warning: implicit declaration of function ‘wait’ [-Wimplicit-function-declaration]
   18 |         wait(NULL);
      |         ^~~~
soft@soft:~/桌面/C$ ./fork
fork  fork.c  hello  hello.c
Child completed
```

（3）完成 proceses.c

```
soft@soft:~/桌面/C$ nano processes.c
soft@soft:~/桌面/C$ gcc processes.c -o processes
soft@soft:~/桌面/C$ ./processes
In parent process, value is 0
In child process, value is 0
In child process, value's address is 0x5a663c61d02c
In parent process, value's address is 0x5a663c61d02c
In child process, value is 1
In parent process, value is 0
In parent process, value's address is 0x5a663c61d02c
In child process, value's address is 0x5a663c61d02c
In parent process, value is 0
In parent process, value's address is 0x5a663c61d02c
In child process, value is 2
In child process, value's address is 0x5a663c61d02c
In child process, value is 3
In child process, value's address is 0x5a663c61d02c
In parent process, value is 0
```

（4）完成 communication.c

```
soft@soft:~/桌面/C$ nano communication.c
soft@soft:~/桌面/C$ gcc communication.c -o communication
soft@soft:~/桌面/C$ ./communication
Process Parent pid 6755
child1=6756
child1=0
Process pid 6756
Process2 pid 6757
PID: 6756, I have sent.
PID: 6757, I received: I send you 1 times
PID: 6756, I have sent.
PID: 6757, I received: I send you 2 times
PID: 6756, I have sent.
PID: 6757, I received: I send you 3 times
PID: 6756, I have sent.
PID: 6757, I received: I send you 4 times
PID: 6756, I have sent.
PID: 6757, I received: I send you 5 times
PID: 6756, I have sent.
```

3. 用 C 语言实现对 N 个进程采用动态优先权算法的调度

部分代码：（完整代码附在附录）

```
void run_process()
{
    // 每一次循环代表一次时间片
    for (int time_slice = 1; ready_process_queue_head->NEXT != NULL ||
block_process_queue_head->NEXT != NULL; time_slice++)
    {
        printf("第%d 个 time_slice:\n", time_slice);
        // 找到当前优先数最大的就绪进程
        PCB *ready_to_run = find_max_priority_process();

        if (ready_to_run != NULL)
        {
            // 当前运行进程的优先数-3
            if (ready_to_run->PRIORITY - 3 > 0)
                ready_to_run->PRIORITY -= 3;
            else
                ready_to_run->PRIORITY = 0;

            // 就绪队列进程的优先数+1
            change_ready_process_priority(ready_to_run->ID);

            // 当前进程占用时间片+1
            ready_to_run->CPUTIME++;
            // 当前进程还需要的时间片-1
            ready_to_run->ALLTIME = ready_to_run->ALLTIME > 0 ? ready_to_run->ALLTIME
- 1 : ready_to_run->ALLTIME;
            // 当前进程开始阻塞倒计时-1
            if (ready_to_run->STARTBLOCK > 0)
                ready_to_run->STARTBLOCK--;
            printf("RUNNING_PROG: %d\n", ready_to_run->ID);

            // 到达阻塞时刻
            if (ready_to_run->STARTBLOCK == 0)
            {
                printf("开始阻塞\n");
                // 进入阻塞队列
                push_to_block_process(ready_to_run->ID);
            }

            // 进程完成
            if (ready_to_run->ALLTIME == 0)
                process_finish(ready_to_run->ID);
```

```c
            printf("READY_QUEUE: ");
            print_ready_queue(ready_to_run->ID);

            printf("BLOCK_QUEUE:");
            print_wait_queue();

            printf("-------------------------------------------------------\n");
        }
        // 阻塞队列检查
        check_block_process();
        print();
    }
}
```

终端输入：

```
soft@soft:~/桌面/C$ gcc main.c -o main
soft@soft:~/桌面/C$ ./main
输入需要初始化进程的个数：
5
输入进程的初始化信息：
9 0 3 2 3
38 0 3 1 4
30 0 6 4 2
29 0 3 1 1
0 0 4 1 2
1
```

输出的运行结果：

```
soft@soft: ~/桌面/C
第1个 time_slice:
RUNNING_PROG: 1
READY_QUEUE: -->id:0 -->id:2 -->id:3 -->id:4
BLOCK_QUEUE:
----------------------------------------------------
ID              0       1       2       3       4
PRIORITY        10      35      31      30      1
CPUTIME         0       1       0       0       0
ALLTIME         3       2       6       3       4
STARTBLOCK      2       -1      -1      -1      -1
BLOCKTIME       3       0       0       0       0
STATE           READY   READY   READY   READY   READY


第2个 time_slice:
RUNNING_PROG: 1
READY_QUEUE: -->id:0 -->id:2 -->id:3 -->id:4
BLOCK_QUEUE:
----------------------------------------------------
ID              0       1       2       3       4
PRIORITY        11      32      32      31      2
CPUTIME         0       2       0       0       0
ALLTIME         3       1       6       3       4
```

……

```
soft@soft: ~/桌面/C
PRIORITY        12      30      18      26      6
CPUTIME         3       3       6       3       3
ALLTIME         0       0       0       0       1
STARTBLOCK      -1      -1      -1      -1      -1
BLOCKTIME       0       0       0       0       0
STATE           END     END     END     END     READY


第19个 time_slice:
RUNNING_PROG: 4
READY_QUEUE:
BLOCK_QUEUE:
----------------------------------------------------
ID              0       1       2       3       4
PRIORITY        12      30      18      26      3
CPUTIME         3       3       6       3       4
ALLTIME         0       0       0       0       0
STARTBLOCK      -1      -1      -1      -1      -1
BLOCKTIME       0       0       0       0       0
STATE           END     END     END     END     END
```

【结论与讨论】（结果、分析）

在完成上述实验后，我对进程调度和进程间通信有了更深入的理解。

1. 进程调度算法的影响：不同的调度算法会导致不同的进程执行顺序，这可能影响到程序的响应时间和资源利用效率。

2. 优先权调整：动态优先权调度算法允许系统根据进程的行为（如等待时间和已运行时间）调整其优先级，这有助于防止饥饿并提高系统公平性。

3. 进程间通信的必要性：进程间通信是多任务操作系统中的一个关键方面，它允许进程协作完成任务。

三、指导教师评语及成绩：

评语：

成绩： **指导教师签名：**

批阅日期：

附录：
对 N 个进程采用动态优先权算法的调度的完整代码如下。
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 最大进程数
#define MAX_PROCESS_NUM 100
// 初始进程数
int process_num = 0;

// 进程结构体
typedef struct PCB {
    int ID;
    int PRIORITY;
    int CPUTIME;
    int ALLTIME;
    int STARTBLOCK;
    int BLOCKTIME;
    char STATE[10];
    struct PCB* NEXT;
} PCB;

// 全局进程队列
PCB* ALL_PROCESS[MAX_PROCESS_NUM];
// 阻塞队列头节点，空数据
PCB* block_process_queue_head = NULL;
// 就绪队列头节点，空数据
PCB* ready_process_queue_head = NULL;

// 初始化进程
void init_process() {
    printf("输入进程的初始化信息:\n");
    PCB* before = ready_process_queue_head;
    for (int i = 0; i < process_num; i++) {
        PCB* p = (PCB*)malloc(sizeof(PCB));
        // 加入全局进程队列
        ALL_PROCESS[i] = p;
        before->NEXT = p;
        scanf("%d %d %d %d %d", &p->PRIORITY, &p->CPUTIME, &p->ALLTIME,
&p->STARTBLOCK, &p->BLOCKTIME);
        p->ID = i;
        strcpy(p->STATE, "READY");
        p->NEXT = NULL;
```

```c
        before = p;
    }
}

void print_addr(PCB* head) {
    PCB* curr = head;
    while (curr != NULL) {
        printf("%p --> ", curr);
        curr = curr->NEXT;
    }
    printf("\n");
}

void print_ready_queue(int curr_pid) {
    PCB* curr = ready_process_queue_head->NEXT;
    while (curr != NULL) {
        if (curr->ID != curr_pid) {
            printf("-->id:%d ", curr->ID);
        }
        curr = curr->NEXT;
    }
    printf("\n");
}

void print() {
    printf("ID\t\t");
    for (int i = 0; i < process_num; i++) {
        printf("%d\t", ALL_PROCESS[i]->ID);
    }
    printf("\n");

    printf("PRIORITY\t");
    for (int i = 0; i < process_num; i++) {
        printf("%d\t", ALL_PROCESS[i]->PRIORITY);
    }
    printf("\n");

    printf("CPUTIME\t\t");
    for (int i = 0; i < process_num; i++) {
        printf("%d\t", ALL_PROCESS[i]->CPUTIME);
    }
    printf("\n");

    printf("ALLTIME\t\t");
```

```c
    for (int i = 0; i < process_num; i++) {
        printf("%d\t", ALL_PROCESS[i]->ALLTIME);
    }
    printf("\n");

    printf("STARTBLOCK\t");
    for (int i = 0; i < process_num; i++) {
        printf("%d\t", ALL_PROCESS[i]->STARTBLOCK);
    }
    printf("\n");

    printf("BLOCKTIME\t");
    for (int i = 0; i < process_num; i++) {
        printf("%d\t", ALL_PROCESS[i]->BLOCKTIME);
    }
    printf("\n");

    printf("STATE\t\t");
    for (int i = 0; i < process_num; i++) {
        printf("%s\t", ALL_PROCESS[i]->STATE);
    }

    printf("\n\n\n");
}

void print_wait_queue() {
    PCB* curr = block_process_queue_head->NEXT;
    while (curr != NULL) {
        printf("-->id:%d ", curr->ID);
        curr = curr->NEXT;
    }
    printf("\n");
}

// 查找优先级最高的就绪进程
PCB* find_max_priority_process() {
    PCB* temp = ready_process_queue_head->NEXT;
    int max_priority = 0;
    PCB* max_pointer = NULL;
    while (temp != NULL) {
        if (max_priority <= temp->PRIORITY) {
            max_priority = temp->PRIORITY;
            max_pointer = temp;
        }
```

```c
        temp = temp->NEXT;
    }
    return max_pointer;
}

// 将阻塞的进程加入等待队列，并从就绪队列中移除
void push_to_block_process(int be_block_pid) {
    PCB* before_curr = ready_process_queue_head;
    PCB* curr = ready_process_queue_head->NEXT;
    while (curr != NULL) {
        if (curr->ID == be_block_pid) {
            strcpy(curr->STATE, "BLOCK");
            before_curr->NEXT = curr->NEXT;

            PCB* temp = block_process_queue_head->NEXT;
            block_process_queue_head->NEXT = curr;
            curr->NEXT = temp;
            break;
        }
        before_curr = before_curr->NEXT;
        curr = curr->NEXT;
    }
}

// 就绪队列优先数改变
void change_ready_process_priority(int curr_pid) {
    PCB* curr = ready_process_queue_head->NEXT;
    while (curr != NULL) {
        if (curr->ID != curr_pid) {
            curr->PRIORITY++;
        }
        curr = curr->NEXT;
    }
}

// 阻塞队列检查
void check_block_process() {
    PCB* before_curr = block_process_queue_head;
    PCB* curr = block_process_queue_head->NEXT;
    while (curr != NULL) {
        if (curr->BLOCKTIME == 0) {
            // 移出阻塞队列，加入就绪队列
            before_curr->NEXT = curr->NEXT;
            PCB* temp = ready_process_queue_head->NEXT;
```

```c
            ready_process_queue_head->NEXT = curr;
            curr->NEXT = temp;
            curr->STARTBLOCK = -1;
            strcpy(curr->STATE, "READY");
            curr = before_curr->NEXT;
        } else if (curr->BLOCKTIME > 0) {
            curr->BLOCKTIME--;
            curr = curr->NEXT;
            before_curr = before_curr->NEXT;
        }
    }
}

// 进行执行完成，移出就绪队列
void process_finish(int curr_pid) {
    PCB* before = ready_process_queue_head;
    PCB* curr = ready_process_queue_head->NEXT;
    while (curr != NULL) {
        if (curr->ID == curr_pid) {
            before->NEXT = curr->NEXT;
            strcpy(curr->STATE, "END");
            break;
        }
        before = before->NEXT;
        curr = curr->NEXT;
    }
}

// 运行某一进程，每运行一个时间片，都要从队列中重新检查优先数
void run_process() {
    // 每一次循环代表一次时间片
    for (int time_slice = 1; ready_process_queue_head->NEXT != NULL ||
block_process_queue_head->NEXT != NULL; time_slice++) {
        printf("第%d 个 time_slice:\n", time_slice);
        // 找到当前优先数最大的就绪进程
        PCB* ready_to_run = find_max_priority_process();

        if (ready_to_run != NULL) {
            // 当前运行进程的优先数-3
            if (ready_to_run->PRIORITY - 3 > 0) {
                ready_to_run->PRIORITY -= 3;
            } else {
                ready_to_run->PRIORITY = 0;
            }
```

```
            // 就绪队列进程的优先数+1
            change_ready_process_priority(ready_to_run->ID);

            // 当前进程占用时间片+1
            ready_to_run->CPUTIME++;
            // 当前进程还需要的时间片-1
            ready_to_run->ALLTIME = ready_to_run->ALLTIME > 0 ?
ready_to_run->ALLTIME - 1 : ready_to_run->ALLTIME;
            // 当前进程开始阻塞倒计时-1
            if (ready_to_run->STARTBLOCK > 0) {
                ready_to_run->STARTBLOCK--;
            }
            printf("RUNNING_PROG: %d\n", ready_to_run->ID);

            // 到达阻塞时刻
            if (ready_to_run->STARTBLOCK == 0) {
                printf("开始阻塞\n");
                // 进入阻塞队列
                push_to_block_process(ready_to_run->ID);
            }

            // 进程完成
            if (ready_to_run->ALLTIME == 0) {
                process_finish(ready_to_run->ID);
            }

            printf("READY_QUEUE: ");
            print_ready_queue(ready_to_run->ID);

            printf("BLOCK_QUEUE:");
            print_wait_queue();

            printf("----------------------------------------------------
\n");
        }
        // 阻塞队列检查
        check_block_process();
        print();
    }
}

int main() {
    // 动态分配内存给队列头节点
```

```c
    block_process_queue_head = (PCB*)malloc(sizeof(PCB));
    ready_process_queue_head = (PCB*)malloc(sizeof(PCB));

    // 初始化头节点的数据
    block_process_queue_head->NEXT = NULL;
    ready_process_queue_head->NEXT = NULL;

    printf("输入需要初始化进程的个数:\n");
    scanf("%d", &process_num);
    // 初始化进程信息
    init_process();

    // 读取最后一行的输入
    int debug_mode;
    scanf("%d", &debug_mode);

    // 运行进程
    run_process();

    // 清理内存
    free(block_process_queue_head);
    free(ready_process_queue_head);

    return 0;
}
```