实验一

实验报告

一、实验目的

1. **深入理解进程调度算法的原理和实现过程**：通过实际编程练习，加深对进程调度算法的理解，特别是优先级调度算法的工作方式。
2. **学习使用C语言实现进程的创建、阻塞、就绪和运行状态的管理**：掌握如何使用C语言编写代码来管理和调度进程。
3. **掌握优先级调度算法和进程同步机制**：理解优先级调度算法如何影响进程的调度顺序，以及如何通过同步机制确保进程之间的协调。

二、实验内容

1. **编写一个模拟进程调度算法的程序**，实现进程的创建、就绪、运行和阻塞状态的管理。
2. **采用优先级调度算法**，实现进程的就绪队列和阻塞队列。
3. **每个进程具有不同的优先级、运行时间和阻塞时间**，程序需根据这些属性进行调度。
4. **模拟进程在CPU上的运行过程**，并输出每个时间片后的进程状态。

三、实验环境

- **计算机**：用于编写和运行实验代码。
- **编译器**：使用GCC或其他C语言编译器编译和运行代码。
- **文本编辑器**：用于编写代码，如Visual Studio Code、Sublime Text等。
- **终端/命令行工具**：用于编译和运行程序。

四、实验步骤

1. **定义进程控制块（PCB）结构体**，包含以下属性：

```
#define N 10
enum State { Ready, Run, Block, Finish };
struct PCB {
    int id;
    int priority;
    int cputime;
    int alltime;
    int startblock;
    int blocktime;
    int waittime;
    State state;
    PCB* next;
    PCB* pre;
};
```

    - 进程ID：唯一标识每个进程。
    - 优先级：决定进程被调度运行的顺序。
    - 运行时间：记录进程在CPU上已经运行的时间。
    - 总时间：进程需要运行的总时间。
    - 开始阻塞时间：进程在CPU上运行一段时间后需要阻塞的时间点。

- 阻塞时间：进程在阻塞队列中需要等待的时间。
- 等待时间：记录进程在阻塞队列中的等待时间。
- 状态：表示进程当前的状态（就绪、运行、阻塞、完成）。

2. **初始化系统资源**：

```
PCB* ready_pro = NULL, *block_pro = NULL, *ready_tail = NULL, *block_tail = NULL;
int ready_num = 0, block_num = 0;
```

- 就绪队列：用于存放处于就绪状态的进程。
- 阻塞队列：用于存放处于阻塞状态的进程。
- CPU上的进程数：初始设置为0，表示CPU空闲。

3. **创建进程并初始化**：

```
int id[] = { 0,1,2,3,4,5, 6, 7, 8, 9 };
int priority[] = { 9,38,30,29,0,20,40,31,23,45};
int cputime[] = { 0,0,0,0,0,0,0,0,0,0 };
int alltime[] = { 3,3,6,3,4,2,5,4,6,3};
int startblock[] = { 2,-1,-1,-1,-1,2,2,2,-1,-1 };
int blocktime[] = { 3,0,0,0,0,3,0,0,0,0};
State state[] = { Ready,Ready,Ready,Ready,Ready,Ready,Ready,Ready,Ready,Ready };

void init() {
    for (int i = 0; i < N; i++) {
        PCB* pro = (PCB*)malloc(sizeof(PCB));
        pro->id = id[i];
        pro->priority = priority[i];
        pro->cputime = cputime[i];
        pro->alltime = alltime[i];
        pro->startblock = startblock[i];
        pro->blocktime = blocktime[i];
        pro->waittime = 0;
        pro->pre = NULL;
        pro->next = NULL;
        ready_push(pro);
    }
}
```

- 创建10个进程，为每个进程分配一个PCB。
- 初始化进程的ID、优先级、运行时间、总时间、开始阻塞时间、阻塞时间和等待时间等属性。
- 将所有创建的进程按照优先级顺序加入就绪队列。

4. **进程调度模拟**：

```
void simulate() {
    PCB* cpu_pro = NULL;
    int times = 0;
    while (ready_num || block_num) {
        printf("第%d个时间片后:\n", times);
        if (ready_num > 0 && (cpu_pro == NULL || cpu_pro->alltime == 0)) {
            cpu_pro = ready_pop();
            cpu_work(cpu_pro);
        }
        if (cpu_pro != NULL && cpu_pro->alltime == 0) {
            free(cpu_pro);
            cpu_pro = NULL;
        }
        if (cpu_pro != NULL && cpu_pro->cputime == cpu_pro->startblock) {
            block_push(cpu_pro);
            cpu_pro = NULL;
        }
        block_updata();
        ready_updata();
        ready_work();
        block_work();
        printf("RUNNINGPROG:");
        if (cpu_pro != NULL) {
            printf("%d\n", cpu_pro->id);
        } else {
            printf("NULL\n");
        }
        times++;
    }
}
```

- 每个时间片执行以下操作：
  - 检查CPU是否空闲，如果空闲，则从就绪队列中取出优先级最高的进程进行运行。
  - 更新运行进程的优先级（通常优先级会随着时间片的流逝而降低），运行时间和总时间。
  - 如果运行进程的总时间减为0，表示进程执行完毕，将其状态设置为完成，并释放该进程所占用的内存资源。
  - 如果运行进程的运行时间达到开始阻塞时间，将其从运行状态转为阻塞状态，并加入阻塞队列。
  - 遍历阻塞队列，更新每个阻塞进程的等待时间。如果某个进程的等待时间超过其阻塞时间，将其从阻塞队列中移除，并重新加入就绪队列。
  - 遍历就绪队列，更新每个就绪进程的优先级，以反映它们在就绪队列中的等待时间。

5. **输出进程状态信息**：

```
void ready_work() {
    printf("READY_QUEUE:");
    PCB* pro = ready_pro;
    if (pro == NULL) printf("NULL");
    while (pro != NULL) {
        printf("->%d", pro->id);
        pro = pro->next;
    }
    printf("\n");
}

void block_work() {
    printf("BLOCK_QUEUE:");
    PCB* pro = block_pro;
    if (pro == NULL) printf("NULL");
    while (pro != NULL) {
        printf("->%d", pro->id);
        pro = pro->next;
    }
    printf("\n");
}

void print_table() {
    printf("============================================================
\n");
    printf("ID \t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
           record[0].id, record[1].id, record[2].id, record[3].id, record
[4].id, record[5].id, record[6].id, record[7].id, record[8].id, record[9].
id);
    printf("PRIORITY(优先) \t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
           record[0].priority, record[1].priority, record[2].priority, rec
ord[3].priority, record[4].priority, record[5].priority, record[6].priorit
y, record[7].priority, record[8].priority, record[9].priority);
    printf("CPUTIME(使用时间)  \t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
           record[0].cputime, record[1].cputime, record[2].cputime, record
[3].cputime, record[4].cputime, record[5].cputime, record[6].cputime, reco
rd[7].cputime, record[8].cputime, record[9].cputime);
    printf("ALLTIME \t%d\t%d \t%d \t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
           record[0].alltime, record[1].alltime, record[2].alltime, record
[3].alltime, record[4].alltime, record[5].alltime, record[6].alltime, reco
rd[7].alltime, record[8].alltime, record[9].alltime);
    printf("STARTBLOCK(在cpu中能运行的时间)\t%d\t%d\t%d \t%d\t%d\t%d\t%d\t%d
\t%d\t%d\n",
           record[0].startblock, record[1].startblock, record[2].startbloc
k, record[3].startblock, record[4].startblock, record[5].startblock, recor
```

```
d[6].startblock, record[7].startblock, record[8].startblock, record[9].startblock);
    printf("BLOCKTIME(阻塞了多久后，进入就绪队列)\t%d \t%d \t%d\t%d \t%d\t%d \t%d\t%d\t%d\t%d \n",
           record[0].blocktime, record[1].blocktime, record[2].blocktime, record[3].blocktime, record[4].blocktime, record[5].blocktime, record[6].blocktime, record[7].blocktime, record[8].blocktime, record[9].blocktime);
    printf("STATE \t");
    for (int i = 0; i < N; i++) {
        if (record[i].state == Run) printf("RUN \t");
        else if (record[i].state == Ready) printf("READY \t");
        else if (record[i].state == Finish) printf("FINISH\t");
        else if (record[i].state == Block) printf("BLOCK \t");
    }
    printf("\n");
```

- 在每个时间片结束后，输出当前时间片下的进程状态信息，包括：
  - 就绪队列中的进程列表及其属性。
  - 阻塞队列中的进程列表及其属性。
  - 当前正在CPU上运行的进程及其属性。

五、程序清单及描述

以下是代码中的程序清单及描述：

- *ready_push(PCB pro)*:
  - 功能：将指定的进程按优先级插入就绪队列。
  - 描述：使用插入排序算法，确保就绪队列中的进程按优先级从高到低排序。
- ready_pop():
  - 功能：从就绪队列中取出优先级最高的进程。
  - 描述：返回就绪队列的队首元素，并更新就绪队列的头指针。
- ready_updata():
  - 功能：遍历就绪队列，增加所有进程的优先级。
  - 描述：在每个时间片结束时调用，以调整进程的优先级。
- ready_work():
  - 功能：打印就绪队列中所有进程的信息。
  - 描述：输出就绪队列的状态，包括每个进程的ID。
- *block_push(PCB pro)*:

    - 功能：将指定的进程插入阻塞队列。
    - 描述：按照进程需要阻塞的时间长短进行插入排序，时间越短越靠前。
- **block_pop()**:
    - 功能：从阻塞队列中取出队首进程。
    - 描述：返回阻塞队列的队首元素，并更新阻塞队列的头指针。
- **block_updata()**:
    - 功能：更新阻塞队列中进程的等待时间，并将等待时间超过阻塞时间的进程移至就绪队列。
    - 描述：在每个时间片结束时调用，处理阻塞队列中的进程。
- **block_work()**:
    - 功能：打印阻塞队列中所有进程的信息。
    - 描述：输出阻塞队列的状态，包括每个进程的ID。
- *cpu_work(PCB cpu_pro)*:
    - 功能：模拟CPU执行指定进程。
    - 描述：减少进程的剩余运行时间，并更新其状态。
- **init()**:
    - 功能：初始化进程控制块和队列。
    - 描述：为每个进程分配内存，并初始化其属性，然后将它们加入就绪队列。
- **print_table()**:
    - 功能：打印所有进程的详细状态信息。
    - 描述：输出一个表格，显示每个进程的ID、优先级、CPU时间、总时间、开始阻塞时间、阻塞时间以及状态。

六、实验总结

实验总结

通过本次实验，我们深入理解了进程调度算法的原理和实现过程，并掌握了优先级调度算法和进程同步机制的具体应用。以下是主要收获和体会：

1. **深入理解进程调度算法**：
    - 学会了使用优先级调度算法管理进程，确保优先级最高的进程优先运行。
    - 掌握了如何使用链表管理就绪队列和阻塞队列，并保持队列中的进程按优先级或阻塞时间排序。

2. **掌握进程控制块（PCB）的设计**：
    - 定义了包含进程ID、优先级、运行时间、总时间、开始阻塞时间、阻塞时间、等待时间和状态等属性的PCB结构体。
    - 通过初始化和管理这些属性，能够准确跟踪每个进程的状态变化。
3. **实现进程调度模拟**：
    - 编写了模拟进程调度的程序，每个时间片执行一系列操作，包括检查CPU状态、调度进程、更新进程属性、处理阻塞队列等。
    - 清晰地观察到进程在不同状态之间的转换，以及调度算法对进程优先级的影响。
4. **解决实际问题的能力提升**：
    - 遇到了多个挑战，如PCB结构设计、队列管理、时间片轮转中的优先级更新等。
    - 通过查阅资料、讨论和调试，逐步解决了这些问题，提升了编程能力和问题解决能力。
5. **对操作系统内部机制的深入理解**：
    - 认识到好的调度算法能够提高系统资源利用率，减少进程等待时间，提升系统效率。
    - 学会了通过编程模拟真实系统行为，有助于理解和优化实际操作系统。

通过本次实验，我们不仅巩固了理论知识，还通过实践加深了对进程调度算法的理解。这些经验和技能将为我们今后的学习和职业生涯打下坚实基础。我们期待在后续的学习中，继续将理论与实际应用结合，为开发高效、稳定的系统软件做出贡献。

详细

一、实验目的

1. **深入理解进程调度算法的原理和实现过程**
    - **理论基础**：进程调度是操作系统的一个关键功能，负责决定哪个进程应该在什么时候使用CPU资源。常见的调度算法包括先来先服务（FCFS）、最短作业优先（SJF）、时间片轮转（RR）和优先级调度等。
    - **实际应用**：通过编程练习，可以更好地理解这些算法的内部机制和实现细节，特别是优先级调度算法如何根据进程的优先级来决定其执行顺序。
2. **学习使用C语言实现进程的创建、阻塞、就绪和运行状态的管理**

- **C语言基础**：C语言是一种底层编程语言，适合进行系统级编程。通过C语言，可以精细地控制内存管理和进程状态。
- **具体实现**：学会使用C语言的数据结构（如链表）和函数来管理进程的状态，包括创建新的进程、将进程加入就绪队列、从就绪队列中取出进程、将进程放入阻塞队列等。

3. **掌握优先级调度算法和进程同步机制**
   - **优先级调度**：优先级调度算法根据进程的优先级来决定其执行顺序。优先级可以是静态的（固定不变）或动态的（随时间变化）。
   - **进程同步**：进程同步机制确保多个进程之间能够协调工作，避免竞争条件和死锁。常见的同步机制包括互斥锁（Mutex）、信号量（Semaphore）和条件变量（Condition Variable）。

二、实验内容

1. **编写一个模拟进程调度算法的程序**
   - **程序结构**：程序主要包括进程控制块（PCB）的定义、队列管理函数、进程调度逻辑和状态输出函数。
   - **关键功能**：实现进程的创建、就绪、运行和阻塞状态的管理，并在每个时间片后输出进程状态。
2. **采用优先级调度算法**
   - **优先级队列**：使用优先级队列来管理就绪队列和阻塞队列，确保优先级最高的进程优先被调度。
   - **队列操作**：实现队列的插入（`ready_push` 和 `block_push`）、删除（`ready_pop` 和 `block_pop`）和更新（`ready_updata` 和 `block_updata`）操作。
3. **每个进程具有不同的优先级、运行时间和阻塞时间**
   - **进程属性**：每个进程都有唯一的ID、优先级、运行时间、总时间、开始阻塞时间、阻塞时间、等待时间和当前状态。
   - **属性管理**：在程序运行过程中，根据进程的行为动态更新这些属性。
4. **模拟进程在CPU上的运行过程，并输出每个时间片后的进程状态**
   - **时间片模拟**：每个时间片代表一个时间单位，在每个时间片结束时，更新进程的状态和属性。
   - **状态输出**：在每个时间片后，输出当前就绪队列、阻塞队列和CPU上的进程状态。

三、实验环境

- **计算机**：用于编写和运行实验代码。
- **编译器**：使用GCC或其他C语言编译器编译和运行代码。

- **文本编辑器**：如Visual Studio Code、Sublime Text等，用于编写代码。
- **终端/命令行工具**：用于编译和运行程序。

四、实验步骤

1. 定义进程控制块（PCB）结构体

```
#define N 10
enum State { Ready, Run, Block, Finish };
struct PCB {
    int id;
    int priority;
    int cputime;
    int alltime;
    int startblock;
    int blocktime;
    int waittime;
    State state;
    struct PCB* next;
    struct PCB* pre;
};
```

2. 初始化系统资源

```
struct PCB* ready_pro = NULL, *block_pro = NULL, *ready_tail = NULL, *block_tail = NULL;
int ready_num = 0, block_num = 0;
```

3. 创建进程并初始化

```
int id[] = { 0,1,2,3,4,5, 6, 7, 8, 9 };
int priority[] = { 9,38,30,29,0,20,40,31,23,45};
int cputime[] = { 0,0,0,0,0,0,0,0,0,0 };
int alltime[] = { 3,3,6,3,4,2,5,4,6,3};
int startblock[] = { 2,-1,-1,-1,-1,2,2,2,-1,-1 };
int blocktime[] = { 3,0,0,0,0,3,0,0,0,0};
enum State state[] = { Ready,Ready,Ready,Ready,Ready,Ready,Ready,Ready,Ready,Ready };

void init() {
    for (int i = 0; i < N; i++) {
        struct PCB* pro = (struct PCB*)malloc(sizeof(struct PCB));
        pro->id = id[i];
        pro->priority = priority[i];
        pro->cputime = cputime[i];
        pro->alltime = alltime[i];
        pro->startblock = startblock[i];
        pro->blocktime = blocktime[i];
        pro->waittime = 0;
        pro->pre = NULL;
        pro->next = NULL;
        ready_push(pro);
    }
}
```

4. 进程调度模拟

```c
void simulate() {
    struct PCB* cpu_pro = NULL;
    int times = 0;
    while (ready_num || block_num) {
        printf("第%d个时间片后:\n", times);
        if (ready_num > 0 && (cpu_pro == NULL || cpu_pro->alltime == 0)) {
            cpu_pro = ready_pop();
            cpu_work(cpu_pro);
        }
        if (cpu_pro != NULL && cpu_pro->alltime == 0) {
            free(cpu_pro);
            cpu_pro = NULL;
        }
        if (cpu_pro != NULL && cpu_pro->cputime == cpu_pro->startblock) {
            block_push(cpu_pro);
            cpu_pro = NULL;
        }
        block_updata();
        ready_updata();
        ready_work();
        block_work();
        printf("RUNNINGPROG:");
        if (cpu_pro != NULL) {
            printf("%d\n", cpu_pro->id);
        } else {
            printf("NULL\n");
        }
        times++;
    }
}
```

5. 输出进程状态信息

```c
void ready_work() {
    printf("READY_QUEUE:");
    struct PCB* pro = ready_pro;
    if (pro == NULL) printf("NULL");
    while (pro != NULL) {
        printf("->%d", pro->id);
        pro = pro->next;
    }
    printf("\n");
}

void block_work() {
    printf("BLOCK_QUEUE:");
    struct PCB* pro = block_pro;
    if (pro == NULL) printf("NULL");
    while (pro != NULL) {
        printf("->%d", pro->id);
        pro = pro->next;
    }
    printf("\n");
}

void print_table() {
    printf("============================================================\n");
    printf("ID \t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
           record[0].id, record[1].id, record[2].id, record[3].id, record[4].id, record[5].id, record[6].id, record[7].id, record[8].id, record[9].id);
    printf("PRIORITY(优先) \t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
           record[0].priority, record[1].priority, record[2].priority, record[3].priority, record[4].priority, record[5].priority, record[6].priority, record[7].priority, record[8].priority, record[9].priority);
    printf("CPUTIME(使用时间)  \t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
           record[0].cputime, record[1].cputime, record[2].cputime, record[3].cputime, record[4].cputime, record[5].cputime, record[6].cputime, record[7].cputime, record[8].cputime, record[9].cputime);
    printf("ALLTIME \t%d\t%d \t%d \t%d\t%d\t%d\t%d\t%d\t%d\t%d\n",
           record[0].alltime, record[1].alltime, record[2].alltime, record[3].alltime, record[4].alltime, record[5].alltime, record[6].alltime, record[7].alltime, record[8].alltime, record[9].alltime);
    printf("STARTBLOCK(在cpu中能运行的时间)\t%d\t%d\t%d \t%d\t%d\t%d\t%d\t%d\t%d\n",
           record[0].startblock, record[1].startblock, record[2].startblock, record[3].startblock, record[4].startblock, record[5].startblock, recor
```

```
d[6].startblock, record[7].startblock, record[8].startblock, record[9].startblock);
    printf("BLOCKTIME(阻塞了多久后，进入就绪队列)\t%d \t%d \t%d\t%d \t%d\t%d\t%d\t%d\t%d\t%d \n",
           record[0].blocktime, record[1].blocktime, record[2].blocktime, record[3].blocktime, record[4].blocktime, record[5].blocktime, record[6].blocktime, record[7].blocktime, record[8].blocktime, record[9].blocktime);
    printf("STATE \t");
    for (int i = 0; i < N; i++) {
        if (record[i].state == Run) printf("RUN \t");
        else if (record[i].state == Ready) printf("READY \t");
        else if (record[i].state == Finish) printf("FINISH\t");
        else if (record[i].state == Block) printf("BLOCK \t");
    }
    printf("\n");
```

五、程序清单及描述

1. ready_push(PCB pro)
   - **功能**：将指定的进程按优先级插入就绪队列。
   - **描述**：使用插入排序算法，确保就绪队列中的进程按优先级从高到低排序。
2. ready_pop()
   - **功能**：从就绪队列中取出优先级最高的进程。
   - **描述**：返回就绪队列的队首元素，并更新就绪队列的头指针。
3. ready_updata()
   - **功能**：遍历就绪队列，增加所有进程的优先级。
   - **描述**：在每个时间片结束时调用，以调整进程的优先级。
4. ready_work()
   - **功能**：打印就绪队列中所有进程的信息。
   - **描述**：输出就绪队列的状态，包括每个进程的ID。
5. block_push(PCB pro)
   - **功能**：将指定的进程插入阻塞队列。
   - **描述**：按照进程需要阻塞的时间长短进行插入排序，时间越短越靠前。
6. block_pop()
   - **功能**：从阻塞队列中取出队首进程。
   - **描述**：返回阻塞队列的队首元素，并更新阻塞队列的头指针。
7. block_updata()

- **功能**：更新阻塞队列中进程的等待时间，并将等待时间超过阻塞时间的进程移至就绪队列。
- **描述**：在每个时间片结束时调用，处理阻塞队列中的进程。

8. block_work()
   - **功能**：打印阻塞队列中所有进程的信息。
   - **描述**：输出阻塞队列的状态，包括每个进程的ID。
9. cpu_work(PCB cpu_pro)
   - **功能**：模拟CPU执行指定进程。
   - **描述**：减少进程的剩余运行时间，并更新其状态。
10. init()
    - **功能**：初始化进程控制块和队列。
    - **描述**：为每个进程分配内存，并初始化其属性，然后将它们加入就绪队列。
11. print_table()
    - **功能**：打印所有进程的详细状态信息。
    - **描述**：输出一个表格，显示每个进程的ID、优先级、CPU时间、总时间、开始阻塞时间、阻塞时间以及状态。

六、实验总结

1. **深入理解进程调度算法**
   - **理论与实践**：通过实验，不仅加深了对优先级调度算法的理解，还学会了如何在实际编程中实现这些算法。
   - **队列管理**：掌握了如何使用链表管理就绪队列和阻塞队列，并保持队列中的进程按优先级或阻塞时间排序。
2. **掌握进程控制块（PCB）的设计**
   - **数据结构**：定义了一个包含进程ID、优先级、运行时间、总时间、开始阻塞时间、阻塞时间、等待时间和状态等属性的PCB结构体。
   - **属性管理**：通过初始化和管理这些属性，能够准确跟踪每个进程的状态变化。
3. **实现进程调度模拟**
   - **时间片模拟**：编写了一个模拟进程调度的程序，每个时间片执行一系列操作，包括检查CPU状态、调度进程、更新进程属性、处理阻塞队列等。
   - **状态转换**：清晰地观察到进程在不同状态之间的转换，以及调度算法对进程优先级的影响。
4. **解决问题的能力提升**

- **挑战与解决**：遇到了多个挑战，如PCB结构设计、队列管理、时间片轮转中的优先级更新等。通过查阅资料、讨论和调试，逐步解决了这些问题，提升了编程能力和问题解决能力。

5. **对操作系统内部机制的深入理解**
   - **系统性能**：认识到好的调度算法能够提高系统资源利用率，减少进程等待时间，提升系统效率。
   - **实际应用**：学会了通过编程模拟真实系统行为，有助于理解和优化实际操作系统。

建议

1. **进一步探索其他调度算法**：尝试实现FCFS、SJF、RR等调度算法，并比较它们的性能。
2. **考虑更复杂的场景**：添加I/O操作、内存管理等复杂性，使模拟更加贴近真实情况。
3. **性能评估**：对不同调度算法进行性能评估，如平均等待时间、吞吐量等，以选择最优的调度策略。