**操作系统实习答辩考核**

班级：计算机224 学号：202205010419 姓名：徐丁

* 请选择各课设所参与的内容：（用⚪标记）

课设1：题目（进程状态转换）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 内容： | 调研 | 设计 | 编码 | 测试 | 撰写 |
| 参与： | ⚪ |  |  | ⚪ |  |

课设2：题目（多级反馈队列调度）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 内容： | 调研 | 设计 | 编码 | 测试 | 撰写 |
| 参与： | ⚪ | ⚪ | ⚪ | ⚪ | ⚪ |

课设3：题目（利用内核模块实现/proc文件系统）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 内容： | 调研 | 设计 | 编码 | 测试 | 撰写 |
| 参与： | ⚪ |  |  |  |  |

课设4：题目（动态分区分配方式的模拟）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 内容： | 调研 | 设计 | 编码 | 测试 | 撰写 |
| 参与： | ⚪ |  |  | ⚪ |  |

课设5：题目（编写一个简单的内核模块）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 内容： | 调研 | 设计 | 编码 | 测试 | 撰写 |
| 参与： | ⚪ |  |  |  |  |

* **自我评价：假设满分是100，你给自己的打分是： （100）**
* **如果你是组长，请给自己的组员评分（满分100）：（例：张三 80 ）**

|  |  |
| --- | --- |
| **姓名** | **分数** |
|  |  |
|  |  |
|  |  |
|  |  |

**请回答下列各题：**

1. **具体工作**：
   * 在整个实习过程中，你具体负责了哪些课设的哪些模块或功能？请逐一具体描述。

答：

在本次实习中，我主要参与了以下几个模块

1. 进程状态转换

我主要负责进程状态转换的调研、测试部分。我首先阅读了课本上进程状态转换模拟的相关内容。在同学编写完成代码之后一起调试，找出bug。

1. 多级反馈队列调度

我主要负责多级反馈队列调度的调研、设计、编码、测试和撰写部分，我首先是阅读课本上的相关内容，了解算法的大致原理后，在网络上搜查相关的参考代码后，我尝试根据自己的理解编写相关代码，但是最后并没有跑通。于是我再去阅读老师上课给出的参考文档，以及CSDN上的一些参考代码，对比后发现自己的代码有很多由于理解错误而写错的地方，通过这个过程我对整个算法有了比较好的掌握。接下来，我开始重新设计整个课设的模块，包括创建进程、状态转换、回收进程等功能需要用到的函数。在设计完后，我结合参考代码，重新实现了一份自己的代码，经过不断的测试后，最终成功跑通。最后将所有内容撰写成报告。

1. 利用内核模块实现/proc文件系统

我查阅资料，理解 /proc 文件系统的工作机制，动态文件的创建和访问流程。

1. 内存动态分区分配

我主要负责调研网络上关于内存动态分区分配的算法，与小组成员讨论，分析比较不同的实现方式，在最后的代码出现问题的时候，参与调试。

1. 编写一个简单的内核模块

我主要阅读有关操作系统内核的书籍，了解内核模块的基本概念和工作原理。

1. **代码贡献说明**：
   * 在完成具体的代码编写时，你能列举出你个人贡献的具体代码段或功能点吗？请提供一些相关的函数或者方法来支持你的陈述。

答：

进程控制块PCB类

class PCB {

private:

    string processName; // 进程名（标识符）

    int arriveTime; // 到达时间

    int totalTime; // 需要运行时间

    int usedTime; // 已用CPU时间

    char status; // 进程状态

public:

    PCB() {

        processName = "";

        arriveTime = 0;

        totalTime = 0;

        usedTime = 0;

        status = 'W';

    }

    PCB(string processName, int arriveTime, int totalTime) {

        this->processName = processName; // 进程名

        this->arriveTime = arriveTime; // 到达时间

        this->totalTime = totalTime; // 需要运行时间

        this->usedTime = 0; // 已用CPU时间初始化为0

        this->status = 'W'; // 进程状态初始化为就绪态

    }

    const string &getProcessName() const {

        return processName;

    }

    void setProcessName(const string &processName) {

        this->processName = processName;

    }

    int getArriveTime() const {

        return arriveTime;

    }

    void setArriveTime(int arriveTime) {

        this->arriveTime = arriveTime;

    }

    int getTotalTime() const {

        return totalTime;

    }

    void setTotalTime(int totalTime) {

        this->totalTime = totalTime;

    }

    int getUsedTime() const {

        return usedTime;

    }

    void setUsedTime(int usedTime) {

        this->usedTime = usedTime;

    }

    char getStatus() const {

        return status;

    }

    void setStatus(char status) {

        this->status = status;

    }

    // 重载运算符==

    bool operator==(const PCB &pcb) const {

        return processName == pcb.processName;

    }

    // 重载运算符!=

    bool operator!=(const PCB &pcb) const {

        return !(pcb == \*this);

    }

};

添加进程菜单

void addProcess() {

    string processName; // 进程名

    int arriveTime = nowTime, totalTime; // 到达时间、所需要总时间

    cout << "processName: ";

    cin >> processName;

    cout << "totalTime: ";

    cin >> totalTime;

    PCB pcb(processName, arriveTime, totalTime); // 创建PCB对象

    Process process(&pcb); // 创建进程对象

    processTable.insertProcess(process); // 将进程放入系统进程表中

    process.getPCB()->setStatus('W');

    firstQueue.push(process);

    processTable.display();

    menu();

}

采用多级反馈队列调度算法运行若干时间片

void Run(int time) {

    // 模拟时间流逝

    for (int i = 0; i < time;) {

//        vector<Process> processes = processTable.getProcesses(); // 获取当前系统进程表

        // 循环系统进程表

//        for (int j = 0; j < processTable.getProcesses().size(); j++) {

//            PCB \*pcb = processTable.getProcesses()[j].getPCB(); // 获取PCB

//            // 如果到达时间等于当前时间，说明该进程刚刚到达，将其放入第1级队列的队尾

//            if (pcb->getArriveTime() == nowTime) {

//                firstQueue.push(processTable.getProcesses()[j]);

//                processTable.getProcesses()[j].getPCB()->setStatus('R'); // 设置状态为运行态

//            }

//        }

        // 只要第1队队列不空，就持续对第1队队列中的进程调度并分配时间片

        if (!firstQueue.empty()) {

            // 此时已经进入第1队队列中，如果第2、3队队列不空，说明上次时间片分配给的是第2、3队队列的进程，此时需要将处于第2、3队队列中的进程的持续运行记录清0

            if (!secondQueue.empty()) {

//                for (int j = 0; j < secondQueue.size(); j++) {

//                    Process \*process = &(secondQueue.front());

//                    process->setContinuation(0);

//                    secondQueue.pop();

//                    secondQueue.push(\*process); // 这一行和上一行的顺序不能调换，他们操作的是同一个process对象

//                }

                if (secondQueue.front().getContinuation() != 0) {

                    Process \*process = &(secondQueue.front()); // 获取第2级队列队头进程

                    process->getPCB()->setStatus('W'); // 发生调度后变为等待态

                    process->setContinuation(0); // 连续运行时间清0

                    secondQueue.pop();

                    secondQueue.push(\*process); // 这一行和上一行的顺序不能调换，他们操作的是同一个process对象

                }

            }

            if (!thirdQueue.empty()) {

//                for (int j = 0; j < thirdQueue.size(); j++) {

//                    Process \*process = &(thirdQueue.front());

//                    process->setContinuation(0);

//                    thirdQueue.pop();

//                    thirdQueue.push(\*process); // 这一行和上一行的顺序不能调换，他们操作的是同一个process对象

//                }

                if (thirdQueue.front().getContinuation() != 0) {

                    Process \*process = &(thirdQueue.front()); // 获取第3级队列队头进程

                    process->getPCB()->setStatus('W'); // 发生调度后变为等待态

                    process->setContinuation(0); // 连续运行时间清0

                    thirdQueue.pop();

                    thirdQueue.push(\*process); // 这一行和上一行的顺序不能调换，他们操作的是同一个process对象

                }

            }

            nowTime++; // 当前时间增加

            i++; // 模拟分配了时间片=1的CPU时间

            Process &process = firstQueue.front(); // 获取第1队队列首进程

            PCB \*pcb = process.getPCB(); // 获取该进程的PCB

            pcb->setStatus('R'); // 进程上处理机运行转为运行态

            pcb->setUsedTime(pcb->getUsedTime() + 1); // 已用CPU时间+1

            process.setContinuation(process.getContinuation() + 1); // 连续运行时间+1

            // 如果运行完成

            if (pcb->getUsedTime() == pcb->getTotalTime()) {

                firstQueue.pop(); // 弹出第1队队列

                pcb->setStatus('F'); // 状态设置为完成态

            } else {

                // 由于第一队队列的时间片为1，因此不用考虑已占用CPU时间大于需要运行时间的情况，此时考虑该进程未完成的情况

                pcb->setStatus('W'); // 发生进程调度变为等待态

                process.setContinuation(0); // 发生调度，需要将进程的持续运行记录清0

                secondQueue.push(process); // 放入第2队队列的队尾

//                cout << process.getContinuation() << endl;

                firstQueue.pop(); // 弹出第1队队列

            }

        } else if (!secondQueue.empty()) {

            // 同上，此时已经进入第2队队列中，如果第3队队列不空，此时需要将处于第3队队列中的进程的持续运行记录清0

            if (!thirdQueue.empty()) {

//                for (int j = 0; j < thirdQueue.size(); j++) {

//                    Process \*process = &(thirdQueue.front());

//                    process->setContinuation(0);

//                    thirdQueue.pop();

//                    thirdQueue.push(\*process);

//                }

                if (thirdQueue.front().getContinuation() != 0) {

                    Process \*process = &(thirdQueue.front()); // 获取第3级队列队首进程

                    process->getPCB()->setStatus('W'); // 获取该进程的PCB

                    process->setContinuation(0); // 连续运行时间清0

                    thirdQueue.pop();

                    thirdQueue.push(\*process); // 这一行和上一行的顺序不能调换，他们操作的是同一个process对象

                }

            }

//            cout << "firstQueueEmpty!" << endl;

            Process \*process = &(secondQueue.front()); // 获取第2队队列的队首进程

//            secondQueue.front().getPCB()->setUsedTime(2);

            PCB \*pcb = (\*process).getPCB(); // 该进程的PCB

//            pcb->setUsedTime(2);

            // 只要第2队队列不为空且本次分配的CPU时间片未使用完，就不断对处于第2队队列中的进程进行调度和分配时间片

            while (!secondQueue.empty() && i != time) {

                i++; // 模拟分配了时间片=1的CPU时间

                nowTime++; // 当前时间增加

                pcb->setStatus('R'); // 将状态设置为运行态

                pcb->setUsedTime(pcb->getUsedTime() + 1); // 当前进程已占用CPU时间增加

                process->setContinuation(process->getContinuation() + 1); // 当前进程连续运行时间的记录增加

                // 若进程完成

                if (pcb->getUsedTime() == pcb->getTotalTime()) {

//                    cout << pcb->getProcessName() << " pop from secondQueue" << endl;

                    pcb->setStatus('F'); // 将状态设置为完成态

                    secondQueue.pop(); // 从第2队队列中弹出该进程

                    process = &(secondQueue.front()); // 切换下一个进程

                    pcb = process->getPCB(); // 获取该进程PCB

                    continue; // 跳过后续处理

                }

                // 若达到第2队队列所能分配的最大时间片

                if (process->getContinuation() == secondQueueTime) {

//                    cout << pcb->getProcessName() << " remove to thirdQueue from secondQueue" << endl;

                    pcb->setStatus('W'); // 发生进程调度状态变为等待态

                    process->setContinuation(0); // 发生进程调度，进程连续运行时间记录清0

                    thirdQueue.push(\*process); // 将该进程放入第3队队列

                    secondQueue.pop(); // 将该进程从第2队队列中弹出

                    process = &(secondQueue.front()); // 切换下一个进程

                    pcb = process->getPCB(); // 获取该进程的PCB

                }

            }

        } else {

//            cout << "secondQueueEmpty" << endl;

            // 当其他队列为空时便对第3队队列中的进程进行调度并分配时间片

            Process \*process = &(thirdQueue.front()); // 获取第3队队列队首的进程

            PCB \*pcb = (\*process).getPCB(); // 获得该进程的PCB

            // 当程序进入到处理第三队队列的进程调度时，这意味着前两个队列都是空的，由于第3队队列是最高级的队列，进程在这将采用时间片轮转的调度算法

            while (!thirdQueue.empty() && i != time) {

                i++;

                nowTime++;

                pcb->setStatus('R'); // 状态变为运行态

                pcb->setUsedTime(pcb->getUsedTime() + 1); // 当前进程已占用CPU时间增加

                process->setContinuation(process->getContinuation() + 1); // 当前进程连续运行时间的记录增加

                // 若进程完成

                if (pcb->getUsedTime() == pcb->getTotalTime()) {

//                    cout << pcb->getProcessName() << " pop from thirdQueue" << endl;

                    pcb->setStatus('F'); // 将状态设置为完成态

                    thirdQueue.pop(); // 从第3级队列中弹出该进程

                    process = &(thirdQueue.front()); // 切换下一个进程

                    pcb = process->getPCB();// 获取该进程的PCB

                    continue; // 跳过后续处理

                }

                // 若达到第3级队列所能分配的最大时间片

                if (process->getContinuation() == thirdQueueTime) {

//                    cout << pcb->getProcessName() << " remove to the end" << endl;

                    pcb->setStatus('W'); // 发生进程调度状态变为等待态

                    process->setContinuation(0); // 连续运行时间清0

                    thirdQueue.pop();

                    thirdQueue.push(\*process); // 这一行和上一行顺序不能调换，他们操作的是同一个process对象

                    process = &(thirdQueue.front()); // 切换下一个进程

                    pcb = process->getPCB(); // 获取该进程的PCB

                }

            }

            // 所有进程运行完成

            if (thirdQueue.empty()) {

                nowTime += time - i;

                i = time; // 直接结束

            }

        }

    }

    processTable.display(); // 打印进程信息

    menu(); // 返回菜单

}

1. **算法策略**：
   * 在整个实习课设中，你提出或者设计了哪些具体的算法或策略？请详细说明。并描述你在实现过程中遇到的挑战及解决方案。

答：

多级反馈队列调度算法中，我设计了动态调整优先级的策略，防止低优先级任务长期得不到调度。

挑战：

在时间片分配时出现高优先级任务过多导致低优先级任务饥饿问题。

解决方案：改变时间片大小，高优先级时间片小，低优先级时间片大，动态提升长时间未执行任务的优先级sss。

1. **团队合作体验**：
   * 在整个实习课设中，你与其他组员的合作如何？请提供具体例子说明你在团队中扮演的角色，以及你如何协助解决团队中遇到的问题。

答：

首先，我对多个课题进行了调研，这样可以帮助我更好地了解各个同学的工作内容，遇到问题时我能帮上忙。我主要担任开发角色。

在调试多级反馈队列调度模块时，我与团队成员合作解决了队列溢出问题。当同学在实现/proc文件系统时遇到内核挂载问题，我通过查阅文档并测试提供了解决方案。

1. **测试与调试**：
   * 针对你负责的模块，你进行了哪些测试来确保其功能正确？在调试过程中，你发现了哪些问题，并且是如何解决这些问题的？

答：

我主要进行了以下几个测试：

(1) 进程状态转换测试

模拟各种进程事件，如创建进程、进程调度、请求 I/O、I/O 完成、时间片到期、其他阻塞事件、进程挂起及完成等，检查进程状态是否按照预期在就绪、运行、阻塞、挂起和完成等状态之间正确转换。例如，创建多个进程，设置不同的到达时间和服务时间，观察在调度过程中进程是否按照FCFS或其他选定的调度算法依次进入运行状态，并且在遇到阻塞事件时是否准确地切换到阻塞状态，在相应事件完成后是否能正确地回到就绪状态继续执行

(2) 边界条件测试

测试系统在处理少量进程和大量进程时的行为。当只有少数进程时，检查系统是否能正常启动调度，各个队列的初始化和操作是否正确。对于大量进程，观察系统的性能表现，如是否会出现内存溢出、处理速度明显下降等问题。同时，验证在大量进程情况下，调度算法是否仍能正确地选择和执行进程，进程状态转换是否正常调试问题及解决：

问题：动态分区分配时偶尔出现分区越界错误。

解决：增加边界检查，避免非法访问。

1. **文档编写与维护**：
   * 在项目文档方面，你贡献了哪些内容？例如，你是否编写了部分设计文档、或测试报告？请描述你的文档编写工作及其对项目的意义。

答：

我主要负责编写PCB结构体设计、队列设计与操作说明、调度算法、进程状态转换的说明，同时也撰写了具体代码段和测试报告。

编写了设计文档，详细描述了多级反馈队列调度算法的实现原理和测试策略。让每个组员都能清楚了解我做的内容，同时，清晰的代码段也能方便后面的维护，保证项目质量。

1. **创新点与改进**：
   * 在实现课设要求的过程中，你提出了哪些创新点或对现有方法的改进？请详细描述这些创新或改进，并解释它们如何提升了项目的质量或性能。

答：

1. 在原有的进程状态显示功能基础上，我的想法是用颜色和图形来可视化进程运行的状态，可以帮助用户更直观的查看系统进程的运行情况。
2. 同时，我曾在多级反馈队列调度中尝试加入了“任务动态权重”机制，根据任务的资源占用动态调整其优先级。
3. **项目管理与时间分配**：
   1. 在整个实习课设期间，你是如何管理你的时间和任务的？请举例说明你是如何平衡个人任务与团队任务的。

答：

* 在课设开始时，我计划将项目划分为调研、设计、编码、测试和报告撰写等几个主要阶段，安排一周时间进行相关技术和算法的调研，两周时间完成系统设计，两周时间进行编码实现，一周时间用于全面测试，最后一周完成报告撰写和项目总结。
* 其中，白天专注个人模块的编码工作，晚上参与团队任务，帮助室友解决开发中遇到的问题。
* 举例：在多级反馈队列调度算法实现期间，我利用碎片时间回顾算法实现原理，晚上在室友代码出问题时一起调试。

1. **个人心得**：
   1. 通过整个实习课设，你觉得你的哪方面存在不足？你有改善它们的计划吗？请具体描述。

答：

不足之处：

1. 当程序出现错误时，调试过程不够高效。主要原因可能在于不善于利用调试工具来快速定位问题所在。在面对复杂的进程状态转换错误或算法执行错误，只能通过简单的打印输出来排查问题。

改进计划：

1. 加强对操作系统内核的学习，阅读更多相关源码和书籍。
2. 多参与实际开发项目，提升实践经验，期望在寒假能手写一个自己的操作系统。