操作系统原理实习三

**【实习题目1】：**

模拟单处理器系统的作业调度：分别采用时间片轮转法（时间片为 3）、先来先服务法、 短作业优先法和最高响应比优先法对下述一组程序进行调度，模拟中间的调度过程，输出最终每个进程的结束时间、等待时间、周转时间和带权周转时间。

进程 P1 P2 P3 P4 P5 P6 P7 P8 P9 P10

运行时间 16 11 14 13 15 21 18 10 7 14

已等待时间 6 5 4 3 2 1 2 3 4 5

**【题目分析、相关原理及设计思路】：**

**题目分析：**题目要求使用给出的进程信息通过四种不同的调度算法，计算每个进程的结束时间、等待时间、周转时间和带权周转时间。

**相关原理：**使用不同的调度算法对进程进行调度会产生不同的进程周转顺序，使用的时间也会不一样，在不同情况下使用不同是的算法对进程进行调度会有很大的差别；

单处理系统：即每次内存中只有一道程序，以此只能处理一个进程。

时间片轮转法：这种算法通过将 CPU 时间分割成较小的时间段，称为 "时间片"，并将这些时间片分配给不同的进程来工作。每个进程在完成一个时间片后，就会被暂停，让其他进程有机会使用 CPU。当一个进程的时间片用完后，它会排队等候下一个时间片；

先来先服务法：按照进程到达调度器的顺序进行处理，即先到达的任务先被执行，后到达的任务后被执行；

短作业优先法：按照任务的执行时间对任务进行排序，将执行时间最短的任务优先分配CPU时间；

最高响应比优先发：根据进程的响应比来调度进程的算法。响应比定义为（等待时间+服务时间）/服务时间，其数值越大，优先级越高。

**设计思路：**先构建一个存储进程各项数据的结构体，包括进程ID，到达时间，运行时间，结束时间，等待时间，周转时间，带权周转时间；然后通过四个函数以此实现四种调度算法，在主函数中将题目所给数据作为参数调用四种算法获取答案。

【详细步骤和相应截图】：

**1、定义结构体存储进程信息；**

struct Process {

string id; // 进程ID

int burstTime; // 运行时间

int waitTime; // 等待时间

int turnaroundTime; // 周转时间

float weightedTurnaroundTime; // 带权周转时间

int arriveTime; // 到达时间

int finishTime; //结束时间

Process(string i, int b,int w, int t, float wt, int at, int ft)

: id(i), burstTime(b), waitTime(w), turnaroundTime(t), weightedTurnaroundTime(wt), arriveTime(at), finishTime(ft){}

};

**2、实现时间片轮转法：**

// 时间片轮转算法

void roundRobin(vector<Process> processes, int quantum)

{

queue<Process> readyQueue;

for (Process& p : processes) {

readyQueue.push(p);

}

int currentTime = 0;

int timeSlice = 3;

while (!readyQueue.empty()) {

Process currentProcess = readyQueue.front();

readyQueue.pop();

if (currentProcess.burstTime > timeSlice) {

currentTime += timeSlice;

currentProcess.burstTime -= timeSlice;

readyQueue.push(currentProcess);

}

else {

currentTime += currentProcess.burstTime;

currentProcess.finishTime = currentTime;

currentProcess.turnaroundTime = currentProcess.finishTime - currentProcess.arriveTime;

currentProcess.waitTime = currentProcess.turnaroundTime - currentProcess.burstTime;

currentProcess.weightedTurnaroundTime = (float)currentProcess.turnaroundTime / currentProcess.burstTime;

processes[stoi(currentProcess.id.substr(1)) - 1] = currentProcess;

}

}

// 输出结果

cout << "进程ID\t结束时间\t等待时间\t周转时间\t带权周转时间" << endl;

for (int i = 0; i < processes.size(); i++)

{

cout << processes[i].id << "\t" << processes[i].finishTime<< "\t\t" << processes[i].waitTime << "\t\t" << processes[i].turnaroundTime << "\t\t" << processes[i].weightedTurnaroundTime << endl;

}

}

**3、实现先来先服务法：**

bool cmp(Process a, Process b) { // 按到达时间排序

return a.arriveTime < b.arriveTime;

}

void fcfs(vector<Process> processes)//先来先服务

{

int n = processes.size();

sort(processes.begin(), processes.end(), cmp); // 按到达时间排序

int currentTime = processes[0].arriveTime; // 当前时间为第一个进程到达时间

for (int i = 0; i < n; i++) {

if (processes[i].arriveTime > currentTime) { // 当前无进程运行

currentTime = processes[i].arriveTime;

}

processes[i].waitTime = currentTime - processes[i].arriveTime; // 计算等待时间

processes[i].turnaroundTime = processes[i].burstTime + processes[i].waitTime; // 计算周转时间

processes[i].weightedTurnaroundTime = (float)processes[i].turnaroundTime / processes[i].burstTime; // 计算带权周转时间

currentTime += processes[i].burstTime; // 更新当前时间

processes[i].finishTime = currentTime; // 记录结束时间

}

// 输出结果

cout << "进程ID\t结束时间\t等待时间\t周转时间\t带权周转时间" << endl;

for (int i = 0; i < processes.size(); i++)

{

cout << processes[i].id << "\t" << processes[i].finishTime << "\t\t" << processes[i].waitTime << "\t\t" << processes[i].turnaroundTime << "\t\t" << processes[i].weightedTurnaroundTime << endl;

}

}

**4、实现最短时间优先法：**

bool compare1(Process a, Process b) {

return a.burstTime < b.burstTime;

}

void sjf(vector<Process> processes)//短时间优先

{

sort(processes.begin(), processes.end(), compare1);

int currentTime = 0;

for (int i = 0; i < processes.size(); i++) {

Process& p = processes[i];

p.waitTime = currentTime - p.arriveTime;

if (p.waitTime < 0)

{

p.waitTime = 0;

}

currentTime += p.burstTime;

p.finishTime = currentTime;

p.turnaroundTime = p.finishTime - p.arriveTime;

p.weightedTurnaroundTime = (float)p.turnaroundTime / (float)p.burstTime;

}

cout << "进程ID\t结束时间\t等待时间\t周转时间\t带权周转时间" << endl;

for (int i = 0; i < processes.size(); i++)

{

cout << processes[i].id << "\t" << processes[i].finishTime << "\t\t" << processes[i].waitTime << "\t\t" << processes[i].turnaroundTime << "\t\t" << processes[i].weightedTurnaroundTime << endl;

}

}

**5、最高响应比优先法：**

// 比较函数，用于排序

bool compare2(Process p1, Process p2)

{

float responseRatio1 = calcResponseRatio(p1, 0);

float responseRatio2 = calcResponseRatio(p2, 0);

return responseRatio1 > responseRatio2;

}

void schedule(vector<Process> processes) //响应优先级

{

sort(processes.begin(), processes.end(), compare2);

int currentTime = 0;

for (int i = 0; i < processes.size(); i++)

{

Process& p = processes[i];

p.waitTime = currentTime - p.arriveTime;

p.turnaroundTime = p.waitTime + p.burstTime;

p.weightedTurnaroundTime = (float)p.turnaroundTime / p.burstTime;

p.finishTime = currentTime + p.burstTime;

currentTime = p.finishTime;

}

cout << "进程ID\t结束时间\t等待时间\t周转时间\t带权周转时间" << endl;

for (int i = 0; i < processes.size(); i++)

{

cout << processes[i].id << "\t" << processes[i].finishTime << "\t\t" << processes[i].waitTime << "\t\t" << processes[i].turnaroundTime << "\t\t" << processes[i].weightedTurnaroundTime << endl;

}

}

**6、主函数（此时默认所有进程的到达时间均为0且输出的等待时间没有把已等待时间算在内，已等待时间只用来计算了响应比）：**

int main()

{

// 初始化进程

vector<Process> processes =

{

Process("p1", 16, 6, 0, 0,0,0),//ID,运行时间，等待时间，周转时间，带权周转时间，到达时间，结束时间

Process("p2", 11, 5, 0, 0,0,0),

Process("p3", 14, 4, 0, 0,0,0),

Process("p4", 13, 3, 0, 0,0,0),

Process("p5", 15, 2, 0, 0,0,0),

Process("p6", 21, 1, 0, 0,0,0),

Process("p7", 18, 2, 0, 0,0,0),

Process("p8", 10, 3, 0, 0,0,0),

Process("p9", 7, 4, 0, 0,0,0),

Process("p10", 14, 5, 0, 0,0,0),

};

cout << "时间片轮转调度（时间片为3）：" << endl;

roundRobin(processes,3);

cout << "先来先服务调度：" << endl;

fcfs(processes);

cout << "短时间优先调度：" << endl;

sjf(processes);

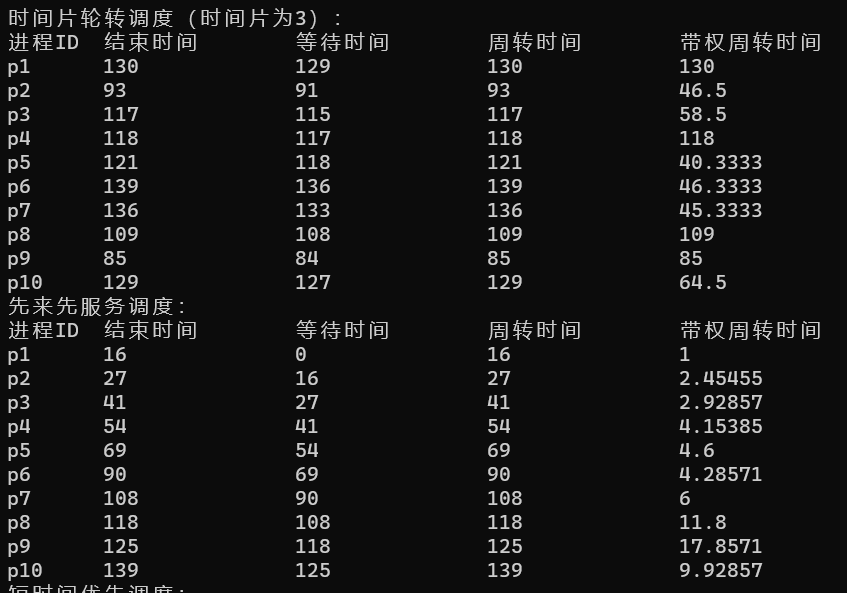
cout << "最高响应比优先调度：" << endl;

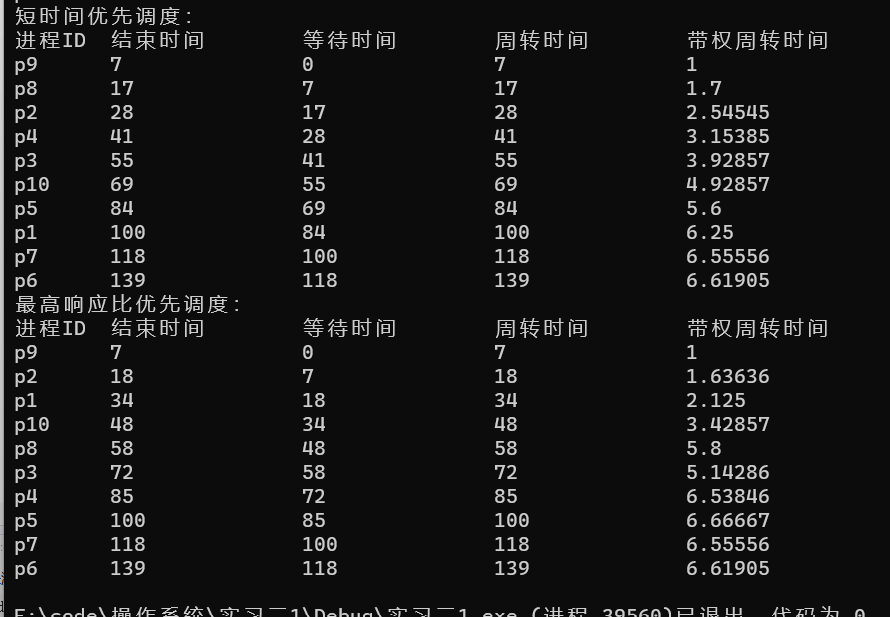
schedule(processes);

return 0;

}

**7、运行结果；**





【实习中遇到的困难及解决方法、实习心得或课程建议】：

**困难：**调度方法中有很多需要根据进程信息进行排序，如何更方便的实现。

**解决办法：**既然已经定义了结构体来存储进程的各项信息，在调度算法实现时参数定为一个进程结构体类型的vector数组，通过vector内置sort函数的第三个参数的控制，就能实现根据进程到达时间或运行时间以及响应比进行排序，从而实现四种调度方法。

**实习心得：**通过代码模拟了简单的单处理系统调度进程的四种方法，了解了他们之间的区别与联系，通过一个测试用例也直观的感受到他们各自的优缺点，更深入的理解了这四种调度方法的原理和实现思路，知道了怎样让进程最大效率的使用CPU，对四种调度的适用情景也更清晰。

【实习题目2】：

基于 mykernel 实现进程调度，默认支持时间片轮转，尝试支持其它调度策略。

【题目分析、相关原理及设计思路】：

**题目分析：**首先完成mykernel内核部署，对配置了mykernel的内核进行编译，再通过编写代码实现简单的时间片轮转调度法后再编译内核，实现简单的进程调度，同时通过QEMU进行可视化。

**相关原理：**mykernel是一个虚拟操作系统内核，它提供了一些基本的操作系统功能，包括进程管理、内存管理、文件系统等，通过使用mykernel提供的API来获取所有处于就绪状态的进程，并使用调度算法来选择下一个应该执行的进程。然后，可以使用mykernel提供的API来切换到选择的进程，并开始执行其代码。

QEMU有两种主要运作模式： User mode模拟模式，亦即是用户模式。QEMU能启动那些为不同中央处理器编译的Linux程序。而Wine及Dosemu是其主要目标。System mode模拟模式，亦即是系统模式。QEMU能模拟整个电脑系统，包括中央处理器及其他周边设备。它使得为跨平台编写的程序进行测试及除错工作变得容易。其亦能用来在一部主机上虚拟数部不同虚拟电脑。

**设计思路：**首先下载linux内核压缩包及其对应版本的mykernel，然后解压缩linux内核，解压后将mykernel配置给内核，内核文件中就会出现一个mykernel文件，再安装QEMU，然后编译内核，编译好后使用QEMU打开内核的bzImage文件，就会出现内核的运行情况，实现时间片调度则在mykernel目录下编写实现调度方法的代码，然后重新编译内核，再使用QEMU打开bzImage，获得进程调度状态。

【详细步骤和相应截图】：

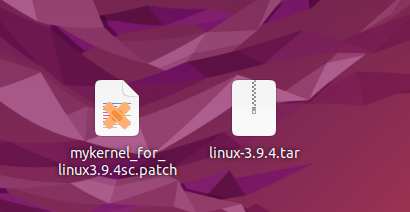
**1、下载linux内核压缩包及其对应的mykernel；**

输入指令

wget

https://www.kernel.org/pub/linux/kernel/v3.x/linux-3.9.4.tar.xz # download Linux Kernel 3.9.4 source code

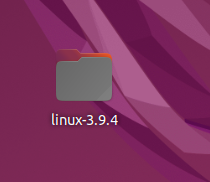
wget https:*//raw.github.com/mengning/mykernel/master/mykernel\_for\_linux3.9.4sc.patch # download mykernel\_for\_linux3.9.4sc.patch*



**2、解压缩linux内核；**

xz -d linux-3.9.4.tar.xz

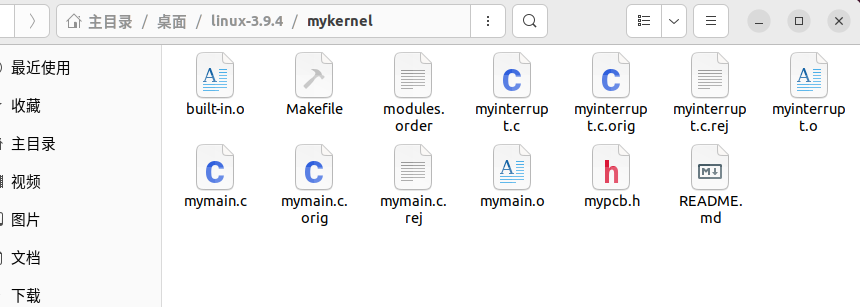
tar -xvf linux-3.9.4.tar



**3、为内核配置mykernel即打补丁；**

cd linux-3.9.4

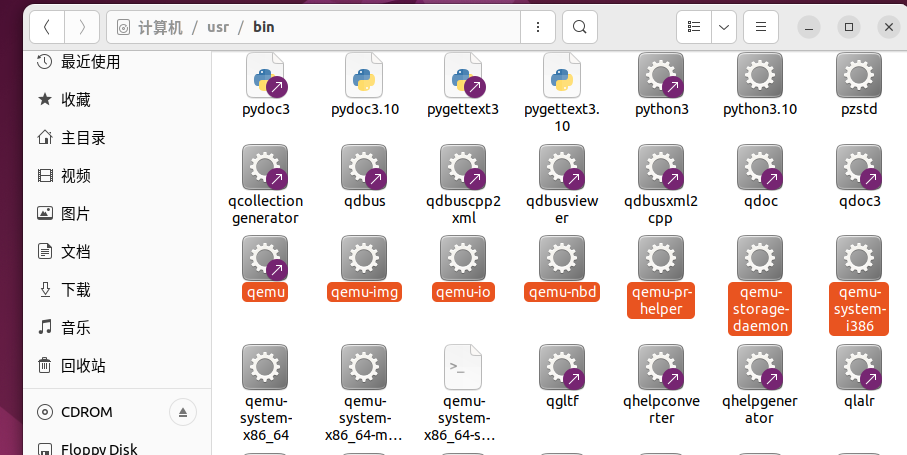
patch -p1 < ../mykernel\_for\_linux3.9.4sc.patch



**4、安装QEMU;**

sudo apt-get install qemu

sudo ln -s /usr/bin/qemu-system-i386 /usr/bin/qemu

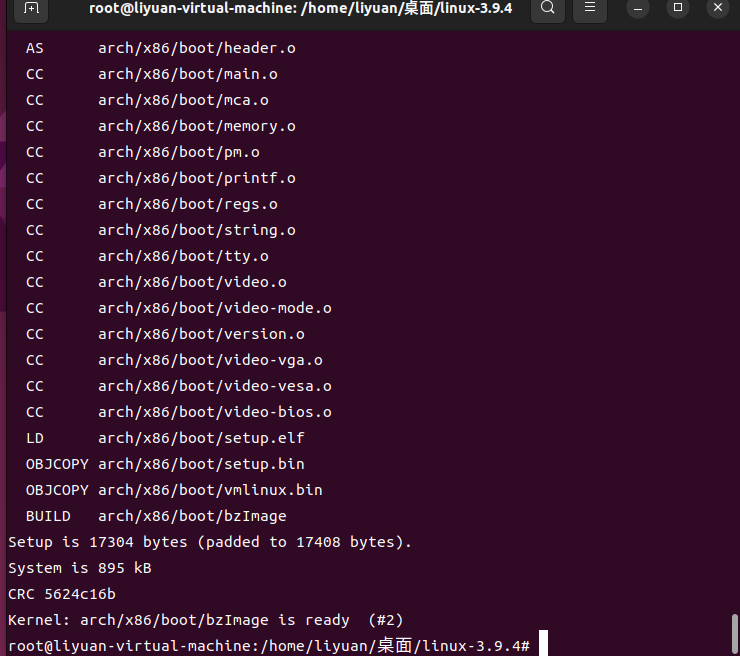


**5、下载依赖包然后编译内核；**

sudo apt install build-essential libncurses-dev bison flex libssl-dev libelf-dev

make allnoconfig

make

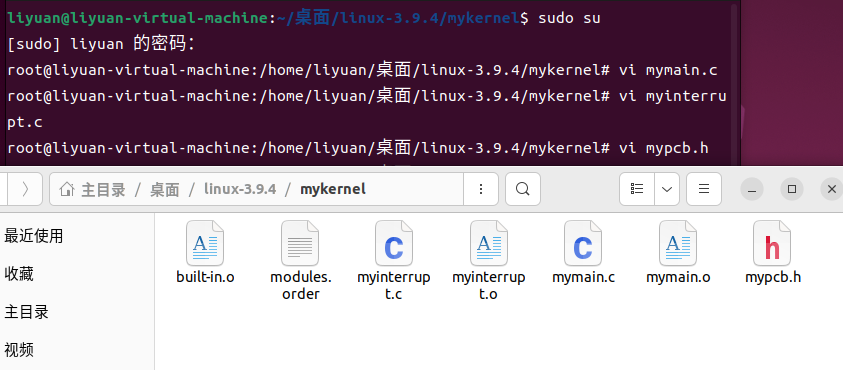


**6、****用QEMU打开bzImage；**

文本

描述已自动生成

**7、在mykernel目录下增添时间片轮转法的代码；**



**8、重新编译内核再次用QEMU打开bzImage；**

make clean

make allnoconfig

make

qemu -kernel arch/x86/boot/bzImage

文本

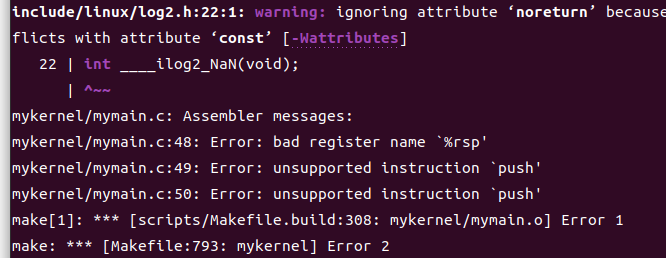
描述已自动生成

【实习中遇到的困难及解决方法、实习心得或课程建议】：

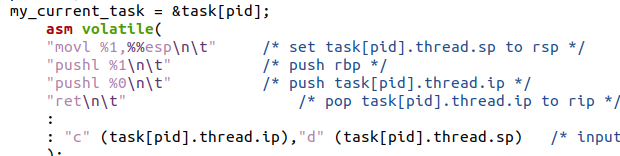
**困难1：**使用linux5.4.34版本时，内核编译总是出错，且编译后找不到bzImage文件；

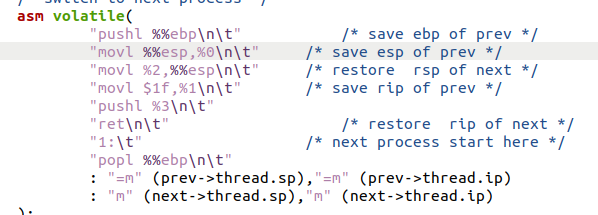
**解决方法：**查了各种资料，有可能是内核源代码有误的问题，实在是没有找到解决方法，只能换个暴力的方法就是换个版本的内核，使用3.9.4版本的内核，但是在编译时也出现了个小问题就是找不到compiler-gcc11.h文件，发现是因为版本过老只有compiler-gcc4.h，则把compiler-gcc4.h拷贝给compiler-gcc11.h：***cd include/linux/*** ***cp*** ***compiler-gcc4.h compiler-gcc5.h***，最后内核编译成功

**困难2：**在拷贝使用给出的时间片轮转代码后再编译内核发现报错：



**解决方法：**观察报错原因可知是mymain.c代码中的48-50行报错：不支持的指令，查找资料可知由于版本编译器较老不支持这种指令，则需要更改这一段的指令，改为老版本的：





即把所有的pushq和movq改为pushl和movl，所有的rsp和rbp改为esp和ebp，改完后重新编译，正确执行。

**实习心得：**通过这个题目，对linux内核编译的操作更加的熟练，知道了查看编译报错来寻找指定文件找到错误的方法，同时对mykernel的原理以及进程调度在内核中的具体实现方法有了实践经验，同时知道了使用qemu能够可视化内核的运行状态，在实验当中也遇到了很多困难，大部分能够花时间进行查资料解决，但是5.4.34内核版本的编译错误问题还是没能搞懂，将会在后续继续摸索寻找其中的问题和解决办法。