

**实 验（实训）报 告**

**项 目 名 称**  （实验名称）

**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**  2024-10-23

**班 级**  22软件2

**学 号**  220110900420

**姓 名**  赖涵

**指导教师**  陈伟锋

浙江财经大学教务处制

|  |
| --- |
| **一、实验（实训）概述：（简单复述实验要求，1页以内，红字部分应删除）** |
| **【目的及要求】**  一、实验5：  （1）安装gcc  （2）完成hello.c  （3）完成fork.c  （4）完成proceses.c  （5）完成communication.c  二、实验6：进程调度模拟  用C语言实现对N个进程采用动态优先权算法的调度  默认输出进程调度序列的id，以空格分隔  当输入的最后一行为1时，输出每一步的系统状态，示例如下  1730267356329  **【基本原理】**  **【实施环境】**  **Linux Ubuntu,** |
| **二、实验（实训）内容：（记录实验内容，要求逻辑清晰、页面整洁、必要的图文并茂，核心的数据及程序代码，不能完全复制所有代码，完整代码若有必要，可以增加附录，尽量控制在6页以内，红字部分应删除）** |
| **【实验（实训）过程】（步骤、记录、数据、程序等）**  一、实验5：  （1）安装gcc  安装gcc,这里我们实际上安装的是"build-essential"，它包含了 GNU 编辑器集合，GNU 调试器，和其他编译软件所必需的开发库和工具。下面这个命令将会安装一系列软件包，包括gcc，g++，和make。  执行  1729058587188  结果1729058670948  检查验证 GCC 编译器是否被成功地安装：  1729058712893   1. 完成hello.c   第一步：vi创建 hello.c  1729061065450  第二步：为hello.c输入以下内容  1729061102990  第三步：编译程序。执行命令：  **1729061160808**  第四步：**执行程序，也就是运行程序。输入命令：**  **1729061205623**   1. 完成fork.c   在终端中输入vim fork.c  同样输入  1729662328682  然后·esc 输入：wq    然后1729662381957  再输入  1729662405922  得到结果  1729662438602   1. 完成proceses.c   终端中输入 vim processes.c  输入1729662528994  退出保存esc :wq  输入+得到1729662654958  可以用ctrl+c中断  1729662700264  （5）完成communication.c  输入vim communication.c  输入1729662825632  1729662846841  退出保存esc ：wq  1729662883007  输入gcc communication.c -o communication  ./communication  1729662992717  二、实验6：进程调度模拟  用C语言实现对N个进程采用动态优先权算法的调度  默认输出进程调度序列的id，以空格分隔  1.首先设计定义变量  进程标识号：PID  进程名：PNAME  进程优先数SUPER，并规定优先数越大的进程，其优先权越高；假定在调度过程中，进程每运行一个时间片，其优先数减2；进程每在就绪队列中待一个时间片，则其优先数加1。(可自行设定动态优先数的变化规律)  进程已占用的CPU时间RUNTIME（rtime）。  进程最大需占用的CPU时间NEEDTIME(ntime)。当RUNTIME等于NEEDTIME时，进程运行完毕。  进程状态STATE。假设实验中的进程只有三种状态：就绪（Wait）、运行（Running）和完成（Finished）。  ————————————————  2.设计所需结构体和函数  定义进程控制块PCB  对进程进行优先级排列函数  输入进程控制块函数  获取就绪状态的进程数  展示进程各个变量的具体内容  查看进程信息，显示当前处于运行态的进程和处于就绪队列的进程  建立进程就绪函数(进程运行时间到，置就绪状态)  3.具体操作  （1）建立chap4.c并打开  1730268589772   1. 编写chap4.c   /\*define the structure of process\*/  #include <stdio.h>  #define N 10  #define P proc  struct process{      int  id;      int  py;        /\*priority\*/      int  ct;        /\*cputime\*/      int  at;        /\*alltime\*/      int  sb;        /\*startblock\*/      int  bt;        /\*blocktime\*/      int  st;        /\*state:1 ready;-1 block;0 finish\*/  };  struct process proc[N];    int main(){      int  i,j,n;      int  max\_pt,priority;      int  timeslice=0;      int  ready\_q[N],block\_q[N];      char ch;      FILE \*fp;      char \*fname="c:\\a.txt";      fp=fopen(fname,"w+");      for(i=0;i<=N;i++){          ready\_q[i]=0;      block\_q[i]=0;      }      n=5;   /\* 定义进程数量为5 \*/      /\* 初始化进程数组 \*/      P[1].id=0; P[1].py=9;  P[1].ct=0; P[1].at=3; P[1].sb=2;  P[1].bt=3;      P[2].id=1; P[2].py=38; P[2].ct=0; P[2].at=3; P[2].sb=-1; P[2].bt=0;      P[3].id=2; P[3].py=30; P[3].ct=0; P[3].at=6; P[3].sb=-1; P[3].bt=0;      P[4].id=3; P[4].py=29; P[4].ct=0; P[4].at=3; P[4].sb=-1; P[4].bt=0;      P[5].id=4; P[5].py=0;  P[5].ct=0; P[5].at=4; P[5].sb=-1; P[5].bt=0;      /\* 输出初始化的进程数组内容\*/      fprintf(fp,"\n\n RUNNING RPOC:%d\n",-1);      fprintf(fp," READY\_QUEUE :");      for(i=1;i<=n;i++)          fprintf(fp,"->id%d",i-1);      fprintf(fp,"\n");      fprintf(fp," BLOCK\_QUEUE :\n ");      fprintf(fp,"===================================================== ===");      fprintf(fp,"\n %-12s","ID");      for(i=1;i<=n;i++)          fprintf(fp,"%8d",i-1);      fprintf(fp,"\n %-12s","PRIORITY");      for(i=1;i<=n;i++)          fprintf(fp,"%8d",P[i].py);      fprintf(fp,"\n %-12s","CPUTIME");      for(i=1;i<=n;i++)          fprintf(fp,"%8d",P[i].ct);      fprintf(fp,"\n %-12s","ALLTIME");      for(i=1;i<=n;i++)          fprintf(fp,"%8d",P[i].at);      fprintf(fp,"\n %-12s","STARTBLOCK");      for(i=1;i<=n;i++)          fprintf(fp,"%8d",P[i].sb);      fprintf(fp,"\n %-12s","BLOCKTIME");      for(i=1;i<=n;i++)          fprintf(fp,"%8d",P[i].bt);      fprintf(fp,"\n %-12s","STATE");      for(i=1;i<=n;i++){          P[i].st=1;          fprintf(fp,"%8s","READY");      }      for(i=1;i<=n;i++){          P[i].st=1;          ready\_q[i]=i;      }      ready\_q[0]=n;      /\* ready\_q[0]表示就绪队列中进程个数，block\_q[0]表示阻塞队列中进程个数\*/      /\* 进行运算\*/      do{          timeslice++; /\* 时间片加1 \*/          /\*          max\_pt记录优先级最高进程的pt,ready\_q数组记录就绪队列的顺序          priority记录优先级最高进程的ID          此程序段使ready\_q排序          \*/          for(i=1,max\_pt=-1;i<=ready\_q[0];i++)              if (P[ready\_q[i]].py>max\_pt){                  max\_pt=P[ready\_q[i]].py;                  j=i;                  priority=ready\_q[i];              }              for(i=j;i<ready\_q[0];i++)                  ready\_q[i]=ready\_q[i+1];              ready\_q[0]--;              /\* 按原则操作各进程，实质是按原则改变数组各值\*/                  P[priority].py-=3;              if (P[priority].py<0) P[priority].py=0;                  P[priority].ct++;              P[priority].at--;          /\* 除执行进程外，其他进程的操作\*/          for(i=1;i<=ready\_q[0];i++)              P[ready\_q[i]].py+=1;          /\* 对阻塞队列和就绪队列赋值\*/          /\*? 进程被阻塞的时间BLOCKTIME，表示已阻塞的进程再等待BLOCKTIME个时间片后，进程将转换成就绪状态；\*/          for(i=1;i<=block\_q[0];i++)          {              P[block\_q[i]].bt-=1; /\* 阻塞队列中的进程的blocktime－1 \*/               if (P[block\_q[i]].bt==0) /\* 如果该进程的blocktime＝0，说明该进程需要进入就绪队列\*/              { P[block\_q[i]].st=1; /\* 将该进程的st状态置1，将其从阻塞队列移入就绪队列\*/                  block\_q[0]--;                  ready\_q[0]++;                  ready\_q[ready\_q[0]]=i;              }          }          if(P[priority].at==0) /\* 对当前执行进程，如果at＝0说明执行完毕\*/          {              P[priority].st=0;              P[priority].py=-1;          }          else if (P[priority].sb==P[priority].ct) /\* 进程的阻塞时间STARTBLOCK，\*/          /\* 表示当进程再运行STARTBLOCK个时间片后，\*/          /\* 进程将进入阻塞状态；\*/          {              P[priority].st=-1;              block\_q[0]++;              block\_q[block\_q[0]]=priority;          }          else /\* 以上两个条件都不符合，直接进入就绪队列\*/          {              ready\_q[0]++;              ready\_q[ready\_q[0]]=priority;          }          /\* 运行一次后输出运行结果\*/          fprintf(fp,"\n\n RUNNING RPOC:%d TIMELICE:%d\n",priority-1,timeslice);          fprintf(fp," READY\_QUEUE :");          for(i=1;i<=ready\_q[0];i++)              fprintf(fp,"->id%d",ready\_q[i]-1);          fprintf(fp,"\n");          fprintf(fp," BLOCK\_QUEUE :");          for(i=1;i<=block\_q[0];i++)              fprintf(fp,"->id%d",block\_q[i]-1);          fprintf(fp,"\n");          fprintf(fp,"===================================================== ===");          fprintf(fp,"\n %-12s","ID");          for(i=1;i<=n;i++)              fprintf(fp,"%8d",i-1);          fprintf(fp,"\n %-12s","PRIORITY");          for(i=1;i<=n;i++)              fprintf(fp,"%8d",P[i].py);          fprintf(fp,"\n %-12s","CPUTIME");          for(i=1;i<=n;i++)              fprintf(fp,"%8d",P[i].ct);          fprintf(fp,"\n %-12s","ALLTIME");          for(i=1;i<=n;i++)              fprintf(fp,"%8d",P[i].at);          fprintf(fp,"\n %-12s","STARTBLOCK");          for(i=1;i<=n;i++)              fprintf(fp,"%8d",P[i].sb);          fprintf(fp,"\n %-12s","BLOCKTIME");          for(i=1;i<=n;i++)              fprintf(fp,"%8d",P[i].bt);          fprintf(fp,"\n %-12s","STATE");          for(i=1;i<=n;i++)          {              if(P[i].st==1)                  fprintf(fp,"%8s","READY");              else if (P[i].st==0)                  fprintf(fp,"%8s","FINISH");              else                  fprintf(fp,"%8s","BLOCK");          }          fprintf(fp,"\n");      }while((ready\_q[0]!=0)||(block\_q[0]!=0)); /\* 循环结束条件：就绪队列和阻塞队列全为空\*/  }  （3）运行代码  1730268626866  (4)在C:/a.txt中查看结果  在文件中找到该文件  1730268746597  输出的结果  1730268773128  **【结论与讨论】（结果、分析）**  通过完成 hello.c、fork.c、processes.c 和 communication.c 四个程序，成功地实现了进程的创建、执行、通信等操作，并加深了对进程概念的理解。  本次实验成功地实现了动态优先权算法的进程调度模拟。通过观察 C:/a.txt 文件中的输出结果，我们可以清晰地看到进程的状态变化、优先级调整以及调度顺序。实验结果表明，动态优先权算法能够有效地根据进程的优先级和运行时间进行调度，从而提高系统资源的利用率。 |
| **三、指导教师评语及成绩：** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |