

**实 验（实训）报 告**

**项 目 名 称**  进程通信与进程调度

**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**  2024/11/6

**班 级**  22软件1班

**学 号**  220110900439

**姓 名**  叶佳颖

**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】**  1.安装gcc  2.完成hello.c  3.完成fork.c  4.完成proceses.c  5.完成communication.c  6.进程调度模拟  **【基本原理】**  **【实施环境】**  硬件环境：一台能够支持虚拟化的物理机，至少有2GB以上的RAM和足够的磁盘空间。  软件环境：虚拟化软件（如VirtualBox、VMware Workstation）、Linux发行版镜像（如Ubuntu Server ISO文件）。 |
| **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    2.完成hello.c    3.完成fork.c    4.完成processes.c    5.完成communication.c    6.进程调度模拟  （1）. 定义一个进程控制块（PCB）结构体，包含进程的各种属性（id、优先级、CPU时间、总时间、阻塞时间等）。  （2）. 从 input.txt文件中读取进程数量和每个进程的初始状态。  （3）. 实现动态优先权调度算法：  - 就绪队列中的进程每经过一个时间片，优先级增加1。  - 阻塞队列中的进程优先级不变。  - 运行中的进程每经过一个时间片，优先级减少3。  （4）. 根据输入决定是否输出每一步的系统状态。  （5）. 输出最终的进程调度序列。  代码如下：  #include <stdio.h>  #include <stdlib.h>  typedef enum { READY, BLOCK, RUNNING } State;  typedef struct {      int id;      int priority;      int cputime;      int alltime;      int startblock;      int blocktime;      State state;  } PCB;  void print\_system\_state(PCB \*processes, int n, int running\_id) {      printf("Running\_program -> %d\n", running\_id);      printf("Ready\_queue -> ");      for (int i = 0; i < n; i++) {          if (processes[i].state == READY) {              printf("%d -> ", processes[i].id);          }      }      printf("\nBlock\_queue -> ");      for (int i = 0; i < n; i++) {          if (processes[i].state == BLOCK) {              printf("%d -> ", processes[i].id);          }      }      printf("\n\n==============================================\n\n");      printf("ID       ");      for (int i = 0; i < n; i++) printf("%d   ", processes[i].id);      printf("\nPRIORITY ");      for (int i = 0; i < n; i++) printf("%d   ", processes[i].priority);      printf("\nCPUTIME  ");      for (int i = 0; i < n; i++) printf("%d   ", processes[i].cputime);      printf("\nALLTIME  ");      for (int i = 0; i < n; i++) printf("%d   ", processes[i].alltime);      printf("\nSTARTBLOCK ");      for (int i = 0; i < n; i++) printf("%d   ", processes[i].startblock);      printf("\nBLOCKTIME ");      for (int i = 0; i < n; i++) printf("%d   ", processes[i].blocktime);      printf("\nSTATE    ");      for (int i = 0; i < n; i++) printf("%d   ", processes[i].state);      printf("\n");  }  int main() {      FILE \*file = fopen("input.txt", "r");      if (file == NULL) {          perror("Failed to open file");          return 1;      }      int n, verbose;      fscanf(file, "%d", &n);      PCB \*processes = (PCB \*)malloc(n \* sizeof(PCB));      for (int i = 0; i < n; i++) {          processes[i].id = i;          fscanf(file, "%d %d %d %d %d", &processes[i].priority, &processes[i].cputime, &processes[i].alltime, &processes[i].startblock, &processes[i].blocktime);          processes[i].state = READY;      }      fscanf(file, "%d", &verbose);      fclose(file);      int time = 0;      while (1) {          int running\_id = -1;          int max\_priority = -1;          for (int i = 0; i < n; i++) {              if (processes[i].state == READY && processes[i].priority > max\_priority) {                  max\_priority = processes[i].priority;                  running\_id = i;              }          }          if (running\_id == -1) break;          processes[running\_id].state = RUNNING;          processes[running\_id].cputime++;          processes[running\_id].alltime--;          processes[running\_id].priority -= 3;          if (verbose) {              print\_system\_state(processes, n, running\_id);          }          for (int i = 0; i < n; i++) {              if (i != running\_id) {                  if (processes[i].state == READY) {                      processes[i].priority++;                  } else if (processes[i].state == BLOCK) {                      processes[i].blocktime--;                      if (processes[i].blocktime == 0) {                          processes[i].state = READY;                      }                  }              }          }          if (processes[running\_id].alltime == 0) {              processes[running\_id].state = READY;          } else if (processes[running\_id].cputime % processes[running\_id].startblock == 0) {              processes[running\_id].state = BLOCK;              processes[running\_id].blocktime = processes[running\_id].blocktime;          } else {              processes[running\_id].state = READY;          }          time++;      }      for (int i = 0; i < n; i++) {          if (processes[i].state == READY) {              printf("%d ", processes[i].id);          }      }      printf("\n");      free(processes);      return 0;  }  输出如下：    **【结论与讨论】（结果、分析）**  实验成功实现了进程通信与调度的基本功能。通过编写和运行一系列C语言程序，验证了进程间通信的各种方法，并通过模拟的进程调度器展示了不同调度策略的效果。特别地，在进程调度部分，实现了基于优先级的动态调度算法，能够根据进程的运行状态自动调整其优先级，从而优化系统资源的分配和利用效率。 |
|  |
| **三、指导教师评语及成绩：** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |