

**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**  2024.10.23

**班 级**  22软件1班

**学 号**  220110900422

**姓 名**  梁子妍

**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processes.c 5. 完成communication.c 6. 用C语言实现对N个进程采用动态优先权算法的调度   **【基本原理】**   1. 进程是进程实体的运行过程，是系统进行资源分配和调度的一个独立单位，分为三种状态，就绪状态、执行状态和阻塞状态，PCB是进程的一部分，是操作系统中最重要的记录性=型数据结构，是进程存在的唯一标志，通常常驻内存。 2. 进程通信是指进程之间的信息交换，分为低级进程通信和高级进程通信，低级进程通信指进程的同步和互斥，效率低且通信对用户不透明，高级进程通信使用方便，且能高效地传送大量数据。进程通信有以下类型：   共享存储器系统、管道通信、消息传递系统、客户机-服务器系统   1. 处理机的调度分为高级调度、低级调度和中级调度。其任务是保存处理机的现场信息，按某种算法选取进程和把处理器分配给进程。进程调度的方式有抢占方式和非抢占方式。 2. 抢占方式：允许调度程序根据某种原则，去暂停某个正在执行的进程，将已分配给该进程的处理机重新分配给另一进程。   非抢占方式：一旦把处理机分配给某进程后，便让该进程一直执行，直至该进程完成某时间而被阻塞时，才再把处理机分配给其他进程，决不允许某进程抢占已经分配出去的处理机。  5、进程调度算法有先来先服务调度算法，短作业优先调度算法，优先权调度算法，时间片轮转调度算法，多级队列调度算法，多级反馈队列调度算法及基于公平原则的调度算法。  **【实施环境】**  1、虚拟机软件：VMware Workstation Player，支持Windows和Linux。  2、Linux发行版：Ubuntu  3、操作系统：Windows11  4、计算机硬件：  CPU：现代的多核处理器，支持虚拟化技术（如Intel VT-x或AMD-V）。  内存：建议至少8GB RAM，推荐16GB以流畅运行虚拟机和主机应用。  硬盘：足够的存储空间以容纳虚拟机镜像和操作系统，建议至少100GB可用空间。  5、命令行工具：终端，在Ubuntu中，可以使用默认的终端来执行命令。 |
| **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      1. 完成hello.c          1. 完成fork.c          1. 完成processes.c      1. 完成communication.c      1. 用C语言实现对N个进程采用动态优先权算法的调度   #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define MAX\_PROCESS\_NUM 100 // 定义能同时管理的最大进程数为100  int process\_num = 0; // 存储当前初始化的进程数  typedef struct PCB  {  int id; // 进程id  int priority; // 进程priority  int cputime; // 已占用CPU时间  int alltime; // 还需占用CPU时间  int startblock; // 进程阻塞时间  int blocktime; // 进程被阻塞时间  char state[10]; // 基础状态  struct PCB \*next; // 指向下一个进程的指针  } PCB; // 定义进程结构体  // 全局进程队列  PCB \*ALL\_PROCESS[MAX\_PROCESS\_NUM];  // 阻塞队列头节点，空数据  PCB \*block\_process\_queue\_head = (PCB \*)malloc(sizeof(PCB));  // 就绪队列头节点，空数据  PCB \*ready\_process\_queue\_head = (PCB \*)malloc(sizeof(PCB));  // 初始化进程信息  void init\_process()  {  PCB \*before = ready\_process\_queue\_head;  for (int i = 0; i < process\_num; i++)  {  PCB \*p = (PCB \*)malloc(sizeof(PCB));  ALL\_PROCESS[i] = p;  before->next = p;  p->id = i;  scanf("%d %d %d %d %d %s", &p->priority, &p->cputime, &p->alltime, &p->startblock, &p->blocktime, p->state);  p->next = NULL;  before = p;  }  }  void print\_addr(PCB \*head)  {  PCB \*curr = head;  while (curr != NULL)  {  printf("%p --> ", curr);  curr = curr->next;  }  printf("\n");  }  void print\_ready\_queue(int curr\_pid)  {  PCB \*curr = ready\_process\_queue\_head->next;  while (curr != NULL)  {  if (curr->id != curr\_pid)  {  printf("-->id:%d ", curr->id);  }  curr = curr->next;  }  printf("\n");  }  void print()  {  printf("id\t\t");  for (int i = 0; i < 5; i++)  {  printf("%d\t", ALL\_PROCESS[i]->id);  }  printf("\n");  printf("priority\t\t");  for (int i = 0; i < 5; i++)  {  printf("%d\t", ALL\_PROCESS[i]->priority);  }  printf("\n");  printf("cputime\t\t");  for (int i = 0; i < 5; i++)  {  printf("%d\t", ALL\_PROCESS[i]->cputime);  }  printf("\n");  printf("alltime\t\t");  for (int i = 0; i < 5; i++)  {  printf("%d\t", ALL\_PROCESS[i]->alltime);  }  printf("\n");  printf("startblock\t\t");  for (int i = 0; i < 5; i++)  {  printf("%d\t", ALL\_PROCESS[i]->startblock);  }  printf("\n");  printf("blocktime\t\t");  for (int i = 0; i < 5; i++)  {  printf("%d\t", ALL\_PROCESS[i]->blocktime);  }  printf("\n");  printf("state\t\t");  for (int i = 0; i < 5; i++)  {  printf("%d\t", ALL\_PROCESS[i]->state);  }  printf("\n\n\n");  }  void print\_wait\_queue()  {  PCB \*curr = block\_process\_queue\_head->next;  while (curr != NULL)  {  printf("-->id:%d ", curr->id);  curr = curr->next;  }  printf("\n");  }  PCB \*find\_max\_priority\_process()  {  PCB \*temp = ready\_process\_queue\_head->next;  int max\_priority = 0;  PCB \*max\_pointer = NULL;  while (temp != NULL)  {  if (max\_pointer <= temp->priority)  {  max\_priority = temp->priority;  max\_pointer = temp;  }  temp = temp->next;  }  return max\_pointer;  }  void check\_block\_process()  {  PCB \*before\_curr = block\_process\_queue\_head;  PCB \*curr = block\_process\_queue\_head->next;  while (curr != NULL)  {  if (curr->blocktime == 0)  {  before\_curr->next = curr->next;  PCB \*temp = ready\_process\_queue\_head->next;  ready\_process\_queue\_head->next = curr;  curr->next = temp;  curr->startblock = -1;  strcpy(curr->state, "READY");  curr = before\_curr->next;  }  else if (curr->blocktime > 0)  {  curr->blocktime--;  curr = curr->next;  before\_curr = before\_curr->next;  }  }  }  void process\_finish(int curr\_pid)  {  PCB \*before = ready\_process\_queue\_head;  PCB \*curr = ready\_process\_queue\_head->next;  while (curr != NULL)  {  if (curr->id == curr\_pid)  {  before->next = curr->next;  strcpy(curr->state, "END");  break;  }  before = before->next;  curr = curr->next;  }  }  void run\_process()  {  for (int time\_slice = 1; ready\_process\_queue\_head->next != NULL || block\_process\_queue\_head->next != NULL; time\_slice++)  {  printf("第%d个 time\_slice:\n", time\_slice);    PCB \*ready\_to\_run = find\_max\_priority\_process();    if (ready\_to\_run != NULL)  {    if (ready\_to\_run->priority - 3 > 0)  {  ready\_to\_run->priority -= 3;  }  else  {  ready\_to\_run->priority = 0;  }    change\_ready\_process\_priority(ready\_to\_run->id);    ready\_to\_run->cputime++;    ready\_to\_run->alltime = ready\_to\_run->alltime > 0 ? ready\_to\_run->alltime - 1 : ready\_to\_run->alltime;    if (ready\_to\_run->startblock > 0)  {  ready\_to\_run->startblock--;  }  printf("RUNNING\_PROG -> %d\n", ready\_to\_run->id);    if (ready\_to\_run->startblock == 0)  {  printf("开始阻塞\n");  push\_to\_block\_process(ready\_to\_run->id);  }    if (ready\_to\_run->alltime == 0)  {  process\_finish(ready\_to\_run->id);  }    printf("READY\_QUEUE ");  print\_ready\_queue(ready\_to\_run->id);    printf("BLOCK\_QUEUE ");  print\_wait\_queue();    printf("=================================================\n");  }  check\_block\_process();  print();  }  }  int main()  {  scanf("%d", &process\_num);  // 初始化进程信息  init\_process();  // 运行进程  run\_process();  return 0;  }  【结论与讨论】（结果、分析）  通过本次进程通信与进程调度的实验，成功实现并深入理解了以下几个关键点：  1、环境设置与基础知识：  安装GCC后顺利编译和运行了多个C语言程序，验证了开发环境的正确性。hello.c程序的实现帮助我们确认了C语言的基本语法和编译过程。进程的创建和调度是操作系统中至关重要的部分。我们实验中完成的基本任务，如创建和协调多个进程，实际模拟了多任务操作系统的基本功能。  2、进程创建与管理：  使用fork()实现了多进程的创建，通过fork.c程序观察到父子进程之间的关系及其独立性。这增强了对进程生命周期、调度和上下文切换的理解。  3、进程间通信：  在communication.c中，我们成功实现了父子进程间的数据传递，熟悉了运用管道作为进程间通信的机制。这为我们在将来的多进程应用开发打下了基础。通信的成功执行表明进程间同步和数据一致性处理的重要性。尽管实验使用的管道实现了基本的同步和数据交换，但在复杂的应用场景中，可能需要考虑更高效的IPC机制。在进一步的学习中，我们应探讨共享内存和消息队列的应用，以应对高并发情况下的通信需求。  4、动态优先权调度算法的实现：  在处理N个进程调度时，我们设计并实现了动态优先权调度算法。这次实验使我们了解了调度算法的复杂性和可调性，以及如何通过动态调整优先级来优化系统性能。  5、调度算法对系统性能的影响：  不同的调度算法对系统的响应时间、吞吐量和资源占用等表现有显著影响。通过本次实验，我们能感觉到调度策略的调整对于整体系统性能的提升或降低具有直接作用。在未来的应用开发中，这将成为优化系统性能的重要考虑因素。  总结总体而言，本次实验不仅让我们巩固了对操作系统中的进程管理和调度基本概念的理解，还提升了编程能力和解决具体问题的能力。未来，希望进一步研究进程调度的其他算法以及进程优化策略，从而更好地应对更复杂的编程任务和系统设计挑战。 |
| **三、指导教师评语及成绩：** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |