

**实 验（实训）报 告**

**项 目 名 称**  进程管理

**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 2024.11.20**

**班 级**  22软件工程1班

**学 号**  220110900734

**姓 名**  徐艺盟

**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. 完成hello.c 2. 完成fork.c 3. 完成process.c 4. 完成communication.c 5. 进程调度模拟   **【基本原理】**  在虚拟机上通过命令行完成c语言程序的运行  通过c语言代码模拟进程调度的过程  **【实施环境】**  **Ubuntu、dev c++** |
| **二、实验（实训）内容：** |
| **【实验（实训）过程】（步骤、记录、数据、程序等）**  1.        2.        3.        4.              5.  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  // 最大进程数  #define MAX\_PROCESS\_NUM 100  // 初始进程数  int process\_num = 0;  // 进程结构体  typedef struct PCB {  int ID;  int PRIORITY;  int CPUTIME;  int ALLTIME;  int STARTBLOCK;  int BLOCKTIME;  char STATE[10];  struct PCB\* NEXT;  } PCB;  // 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 = NULL;  // 就绪队列头节点，空数据  PCB\* ready\_process\_queue\_head = NULL;  // 函数声明  void init\_process();  void init\_queues();  void print\_addr(PCB\* head);  void print\_ready\_queue(int curr\_pid);  void print();  void print\_wait\_queue();  PCB\* find\_max\_priority\_process();  void push\_to\_block\_process(int be\_block\_pid);  void change\_ready\_process\_priority(int curr\_pid);  void check\_block\_process();  void process\_finish(int curr\_pid);  void run\_process();  // 初始化进程  void init\_process() {  printf("输入进程的初始化信息:\n");  PCB\* before = ready\_process\_queue\_head;  for (int i = 0; i < process\_num; i++) {  PCB\* p = (PCB\*)malloc(sizeof(PCB));  // 加入全局进程队列  ALL\_PROCESS[i] = p;  before->NEXT = p;  scanf("%d %d %d %d %d %d %s", &p->ID, &p->PRIORITY, &p->CPUTIME, &p->ALLTIME, &p->STARTBLOCK, &p->BLOCKTIME, p->STATE);  p->NEXT = NULL;  before = p;  }  }  void init\_queues() {  // 初始化阻塞队列和就绪队列的头节点  block\_process\_queue\_head = (PCB\*)malloc(sizeof(PCB));  ready\_process\_queue\_head = (PCB\*)malloc(sizeof(PCB));  block\_process\_queue\_head->NEXT = NULL;  ready\_process\_queue\_head->NEXT = NULL;  }  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\tPRIORITY\tCPUTIME\tALLTIME\tSTARTBLOCK\tBLOCKTIME\tSTATE\n");  for (int i = 0; i < process\_num; i++) {  printf("%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%d\t\t%s\n",  ALL\_PROCESS[i]->ID,  ALL\_PROCESS[i]->PRIORITY,  ALL\_PROCESS[i]->CPUTIME,  ALL\_PROCESS[i]->ALLTIME,  ALL\_PROCESS[i]->STARTBLOCK,  ALL\_PROCESS[i]->BLOCKTIME,  ALL\_PROCESS[i]->STATE);  }  }  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\_priority <= temp->PRIORITY) {  max\_priority = temp->PRIORITY;  max\_pointer = temp;  }  temp = temp->NEXT;  }  return max\_pointer;  }  void push\_to\_block\_process(int be\_block\_pid) {  PCB\* before\_curr = ready\_process\_queue\_head;  PCB\* curr = ready\_process\_queue\_head->NEXT;  while (curr != NULL) {  if (curr->ID == be\_block\_pid) {  strcpy(curr->STATE, "BLOCK");  before\_curr->NEXT = curr->NEXT;  curr->NEXT = block\_process\_queue\_head->NEXT;  block\_process\_queue\_head->NEXT = curr;  break;  }  before\_curr = before\_curr->NEXT;  curr = curr->NEXT;  }  }  void change\_ready\_process\_priority(int curr\_pid) {  PCB\* curr = ready\_process\_queue\_head->NEXT;  while (curr != NULL) {  if (curr->ID != curr\_pid) {  curr->PRIORITY++;  }  curr = curr->NEXT;  }  }  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;  curr->NEXT = ready\_process\_queue\_head->NEXT;  ready\_process\_queue\_head->NEXT = curr;  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 : 0;  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() {  printf("输入需要初始化进程的个数:\n");  scanf("%d", &process\_num);  init\_queues();  init\_process();  run\_process();  for (int i = 0; i < process\_num; i++) {  free(ALL\_PROCESS[i]);  }  free(block\_process\_queue\_head);  free(ready\_process\_queue\_head);  return 0;  }            **【结论与讨论】（结果、分析）**  通过本次实验，涉及了 C 语言中的基本进程操作、进程间通信以及进程调度算法的模拟。每个实验都成功地实现了目标功能，验证了进程控制和通信机制的基本概念。同时，进程调度模拟的实验让我们对不同调度算法的优缺点有了更直观的了解，这对于操作系统原理的学习具有重要意义。  在实际的操作系统中，进程调度是一个复杂且至关重要的部分，它直接影响到系统的响应时间、资源利用率和整体性能。通过这些实验，我们能够更好地理解操作系统在管理进程时的工作原理，以及如何根据具体的需求选择合适的调度策略。 |
| **三、指导教师评语及成绩：** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |