

**实 验（实训）报 告**

**项 目 名 称**  linux进程实验

**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**  2024/11/3

**班 级**  22软工1班

**学 号**  220110900442

**姓 名**  张楠

**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.LAMP与应用  2.进程控制与通信  3.进程调度  4.内存分配  **【基本原理】**  **【实施环境】**  **Linux gcc / windows devc++** |
| **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  命令行输入sudo apt install gcc进行安装  安装完毕后输入 gcc --version进行检测  出现    为安装成功  2.使用gcc完成hello.c  输入touch hello.c创建hello.c文档  Vim hello.c打开hello.c文档  输入i进入插入模式    输入完毕后esc退出插入模式输入：wq保存并退出  输入gcc hello.c进行编译hello.c  输入ls发现出现a.out文件    输入./a.out显示输出    3.写Fork.c    编译运行后出现以下问题：    加入#include <sys/wait.h>头文件报错消失，输入./Fork运行后出现以下结果    3.写Processes.c      运行后死循环：    会一直1234...一直到int上限然后报错，只能关闭终端强行终止任务。  加个判断value范围的条件可以避免此情况。  3.写communication.c    因为代码太长就不放完了。  运行后迅速且循环的出现：    模拟动态优先权算法的进程调度  #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define MAX\_PROCESS\_NUM 100  int process\_num = 0;  typedef struct PCB {  int ID;  int PRIORITY;  int CPUTIME;  int ALLTIME;  int STARTBLOCK;  int BLOCKTIME;  char STATE[10];  struct PCB\* NEXT;  } PCB;  PCB\* ALL\_PROCESS[MAX\_PROCESS\_NUM];  PCB\* block\_process\_queue\_head = NULL;  PCB\* ready\_process\_queue\_head = NULL;  void init\_process() {  printf("ÇëÊäÈë³õÊ¼»¯½ø³ÌµÄ²ÎÊý£º\n");  ready\_process\_queue\_head = (PCB\*)malloc(sizeof(PCB));  if (!ready\_process\_queue\_head) {  exit(1); // Èç¹ûÄÚ´æ·ÖÅäÊ§°Ü£¬ÍË³ö³ÌÐò  }  ready\_process\_queue\_head->NEXT = NULL;  block\_process\_queue\_head = (PCB\*)malloc(sizeof(PCB));  if (!block\_process\_queue\_head) {  exit(1); // Èç¹ûÄÚ´æ·ÖÅäÊ§°Ü£¬ÍË³ö³ÌÐò  }  block\_process\_queue\_head->NEXT = NULL;  PCB\* before = ready\_process\_queue\_head;  for (int i = 0; i < process\_num; i++) {  PCB\* p = (PCB\*)malloc(sizeof(PCB));  if (!p) {  exit(1); // Èç¹ûÄÚ´æ·ÖÅäÊ§°Ü£¬ÍË³ö³ÌÐò  }  ALL\_PROCESS[i] = p;  before->NEXT = p;  if (scanf("%d %d %d %d %d %d %s", &p->ID, &p->PRIORITY, &p->CPUTIME, &p->ALLTIME, &p->STARTBLOCK, &p->BLOCKTIME, p->STATE) != 7) {  exit(1); // Èç¹û¶ÁÈ¡Ê§°Ü£¬ÍË³ö³ÌÐò  }  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 < process\_num; i++) {  printf("%d\t", ALL\_PROCESS[i]->ID);  }  printf("\n");  printf("PRIORITY\t");  for (int i = 0; i < process\_num; i++) {  printf("%d\t", ALL\_PROCESS[i]->PRIORITY);  }  printf("\n");  printf("CPUTIME\t\t");  for (int i = 0; i < process\_num; i++) {  printf("%d\t", ALL\_PROCESS[i]->CPUTIME);  }  printf("\n");  printf("ALLTIME\t\t");  for (int i = 0; i < process\_num; i++) {  printf("%d\t", ALL\_PROCESS[i]->ALLTIME);  }  printf("\n");  printf("STARTBLOCK\t");  for (int i = 0; i < process\_num; i++) {  printf("%d\t", ALL\_PROCESS[i]->STARTBLOCK);  }  printf("\n");  printf("BLOCKTIME\t");  for (int i = 0; i < process\_num; i++) {  printf("%d\t", ALL\_PROCESS[i]->BLOCKTIME);  }  printf("\n");  printf("STATE\t\t");  for (int i = 0; i < process\_num; i++) {  printf("%s\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\_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 = curr;  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--;  before\_curr = before\_curr->NEXT;  curr = 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");  free(curr); // ÊÍ·Å½áÊø½ø³ÌµÄÄÚ´æ  break;  }  before = curr;  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("µÚ%d¸ö 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("¿ªÊ¼×èÈû\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("ÇëÊäÈëÒÑ³õÊ¼»¯½ø³ÌµÄÊýÁ¿£º\n");  scanf("%d", &process\_num);  init\_process();  run\_process();  // ÊÍ·ÅÊ£ÓàÄÚ´æ  PCB\* temp;  while (ready\_process\_queue\_head != NULL) {  temp = ready\_process\_queue\_head;  ready\_process\_queue\_head = ready\_process\_queue\_head->NEXT;  free(temp);  }  while (block\_process\_queue\_head != NULL) {  temp = block\_process\_queue\_head;  block\_process\_queue\_head = block\_process\_queue\_head->NEXT;  free(temp);  }  for (int i = 0; i < process\_num; i++) {  free(ALL\_PROCESS[i]);  }  free(ready\_process\_queue\_head);  free(block\_process\_queue\_head);  return 0;  }  首先更新所有进程的优先级。  然后找到具有最高优先级的就绪进程。  将找到的进程设置为运行状态，并打印当前时间和运行的进程ID。  增加该进程的CPU时间，并更新模拟时间。  如果进程已完成（CPU时间等于总时间），则将其状态设置为就绪；否则，减少其优先级。  如果模拟时间达到1000，或者没有就绪进程，模拟结束。    639c2e2e3d2372c7f00f3d39f5dc1b0  6d1ec4f8cb15e57a035ee3f4fbcdacd  **【结论与讨论】（结果、分析）** |
| **三、指导教师评语及成绩：** |
| **评语：**  **成绩： 指导教师签名：**  **批阅日期：** |