# **操作系统课程设计报告**

## **模拟实现操作系统的部分功能**

### **一. 课程设计的性质和目的**

操作系统是计算机系统配置的基础软件，在整个计算机系统中扮演着核心角色。它负责统一调度和管理计算机资源，提供系统服务，为用户提供灵活便捷的使用环境。本次课程设计作为计算机及应用专业的专业主干课，旨在帮助学生掌握操作系统的基本概念、设计原理与实现技术，培养分析、设计和实现实际操作系统的能力。

### **二. 课程设计任务**

本次课程设计的主要任务是**模拟实现操作系统的部分核心功能**。

### **三. 设计要求**

本项目需实现以下关键功能模块，并确保它们之间能有机协同工作：

* **作业管理：** 模拟提交至少10个作业。系统将按照先来先服务（FCFS）原则，选择最多5个作业进入“内存”，准备执行。
* **内存管理：** 为每个进入内存的作业**创建进程**，并为其分配内存。用户内存空间设定为 **0—1024K**，采用**可变连续分配方式**（具体实现中采用首次适应算法及内存合并）。
* **进程调度：** 实现**时间片轮转调度算法**，公平地调度CPU资源给就绪进程。
* **进程阻塞与唤醒：** 模拟进程在运行过程中**随机进入阻塞状态**，并在模拟的**一段时间后自动唤醒**，重新进入就绪队列。
* **状态显示：** 程序需在每个模拟时间点清晰地**显示**当前系统状态，包括：
  + 后备作业队列信息。
  + 内存的当前分配情况（空闲与占用）。
  + 各个进程的详细信息（ID、状态、CPU使用、内存占用等）。
  + 已完成作业的统计情况。

### **四. 系统设计与主要功能实现**

本操作系统模拟器采用 C 语言编写，通过定义数据结构和一系列函数来模拟操作系统在作业、内存和进程管理方面的行为。

#### **4.1 总体架构**

模拟器采用事件驱动的方式，在每个模拟时间片（tick）内，依次执行内存加载、阻塞检查和进程调度等操作，并实时更新和显示系统状态。

graph TD  
 A[启动] --> B{初始化OS};  
 B --> C[生成初始作业];  
 C --> D{循环模拟时间片};  
 D -- 每时间片 --> E[加载作业到内存];  
 E --> F[检查并唤醒阻塞进程];  
 F --> G[执行进程调度];  
 G --> H[显示系统状态];  
 H --> D;  
 D -- 模拟结束 --> I[清理资源];  
 I --> J[程序退出];

#### **4.2 数据结构**

* **Process 结构体：** 模拟进程控制块 (PCB)，包含进程 ID、作业 ID、当前状态（如就绪、运行、阻塞、完成）、CPU 时间需求和已用时间、内存大小及起始地址、时间片使用情况以及阻塞相关信息。
* **MemoryBlock 结构体：** 模拟内存中的一个分区，包含起始地址、大小、空闲状态（是否被占用）和拥有者进程 ID。
* **队列：**
  + **后备作业队列：** 存放等待进入内存的作业，采用数组模拟**先来先服务**队列。
  + **就绪队列：** 存放等待 CPU 的进程，采用循环数组模拟**时间片轮转**队列。
  + **阻塞队列：** 存放因等待事件而暂停的进程，采用数组模拟。
  + **完成作业列表：** 存放已执行完毕的作业。

#### **4.3 内存管理 (可变连续分配)**

* **初始化：** 用户内存（0-1024K）初始化为一个大的空闲块。
* **分配：** 采用**首次适应（First-Fit）算法**。遍历内存块列表，找到第一个能满足作业内存需求的空闲块。如果空闲块过大，则将其**分裂**为两部分：一部分分配给新进程，另一部分仍保留为空闲。
* **释放与合并：** 当进程终止时，其占用的内存被释放。系统会检查该内存块与其**相邻**的内存块是否为空闲，如果是，则进行**合并**，以减少外部碎片。

#### **4.4 进程调度 (时间片轮转)**

* **调度核心：** scheduler() 函数负责进程调度。
* **时间片管理：** 运行中的进程每模拟一个时间单位，其已使用时间片计数器加一。
* **抢占：** 当运行进程的时间片用完 (time\_slice\_used >= TIME\_QUANTUM) 或进程随机阻塞时，当前进程被**抢占**。
* **队列操作：**
  + 被抢占的进程（未完成）或从阻塞状态唤醒的进程，重新进入**就绪队列**尾部。
  + 调度器从就绪队列头部选择下一个进程进入**运行状态**。

#### **4.5 进程阻塞与唤醒**

* **随机阻塞：** 在 scheduler() 中，运行中的进程有一定概率（例如 10%）会**随机进入阻塞状态**。
* **阻塞队列：** 阻塞的进程被移到阻塞队列，并记录其阻塞开始时间和随机生成的阻塞持续时间。
* **唤醒：** 在每个模拟时间片开始时，系统会检查阻塞队列。如果进程的阻塞时间已到 (current\_time >= block\_start\_time + block\_duration)，则将其从阻塞队列中移除，并放回**就绪队列**。

### **五. 总结与展望**

本项目成功模拟了操作系统在作业管理、内存分配（可变连续分配）、进程调度（时间片轮转）以及进程阻塞/唤醒方面的核心功能。通过本次实践，加深了对操作系统原理的理解。

**未来展望：**

* **更完善的内存管理：** 引入分页或分段机制。
* **更复杂的调度算法：** 如优先级调度、多级反馈队列。
* **同步与互斥：** 模拟信号量、管程等同步机制。
* **文件系统：** 简单模拟文件的创建、读写和目录结构。
* **中断机制：** 模拟时钟中断、I/O 中断等，使模拟器行为更接近真实操作系统。

## 附录：核心代码

#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <time.h>  
#include <unistd.h> // For sleep on Unix-like systems  
  
// --- 常量定义 ---  
#define USER\_MEMORY\_SIZE 1024 // 用户内存大小，单位：K  
#define TIME\_QUANTUM 3 // 时间片大小  
#define MAX\_JOBS 20 // 后备队列最大作业数  
#define MAX\_PROCESSES\_IN\_MEMORY 5 // 内存中最多允许的进程数  
  
// 进程状态  
typedef enum { JOB\_READY, READY, RUNNING, BLOCKED, TERMINATED } ProcessState;  
  
// --- 数据结构 ---  
  
// 进程结构体  
typedef struct {  
 int pid; // 进程ID  
 int job\_id; // 对应作业ID  
 ProcessState state; // 状态  
 int required\_cpu\_time; // 总CPU时间  
 int current\_cpu\_time; // 已执行CPU时间  
 int memory\_size; // 内存大小  
 int memory\_start; // 内存起始地址  
 int time\_slice\_used; // 当前时间片已用  
 long block\_start\_time; // 阻塞开始时间  
 int block\_duration; // 阻塞时长  
} Process;  
  
// 内存块结构体  
typedef struct {  
 int start\_address; // 起始地址  
 int size; // 大小  
 int is\_free; // 1空闲，0占用  
 int owner\_pid; // 拥有者PID  
} MemoryBlock;  
  
// --- 全局变量 ---  
long current\_time = 0;  
int next\_pid = 0;  
int next\_job\_id = 0;  
  
// 内存管理  
MemoryBlock memory\_map[MAX\_PROCESSES\_IN\_MEMORY \* 2 + 2]; // 足够存储分裂后的块  
int num\_memory\_blocks = 0;  
  
// 队列  
Process\* job\_queue[MAX\_JOBS];  
int job\_queue\_count = 0;  
Process\* ready\_queue[MAX\_PROCESSES\_IN\_MEMORY + 1]; // 循环队列  
int ready\_queue\_head = 0;  
int ready\_queue\_tail = 0;  
int ready\_queue\_count = 0;  
Process\* blocked\_queue[MAX\_PROCESSES\_IN\_MEMORY + 1];  
int blocked\_queue\_count = 0;  
  
Process\* running\_process = NULL;  
Process\* completed\_jobs[MAX\_JOBS]; // 假设最多完成MAX\_JOBS个作业  
int completed\_jobs\_count = 0;  
  
// --- 辅助函数声明 ---  
void enqueue\_job(Process\* p);  
Process\* dequeue\_job();  
void enqueue\_ready(Process\* p);  
Process\* dequeue\_ready();  
void enqueue\_blocked(Process\* p);  
void remove\_from\_blocked\_queue(int pid);  
  
// --- 内存管理 ---  
void init\_os() {  
 printf("--- 操作系统模拟器启动 ---\n");  
 memory\_map[0].start\_address = 0;  
 memory\_map[0].size = USER\_MEMORY\_SIZE;  
 memory\_map[0].is\_free = 1;  
 memory\_map[0].owner\_pid = -1;  
 num\_memory\_blocks = 1;  
 srand((unsigned int)time(NULL));  
}  
  
// 首次适应分配内存  
int allocate\_memory(int pid, int requested\_size) {  
 for (int i = 0; i < num\_memory\_blocks; i++) {  
 if (memory\_map[i].is\_free && memory\_map[i].size >= requested\_size) {  
 int allocated\_start = memory\_map[i].start\_address;  
 if (memory\_map[i].size == requested\_size) { // 刚好适配  
 memory\_map[i].is\_free = 0;  
 memory\_map[i].owner\_pid = pid;  
 } else { // 分裂空闲块  
 for (int j = num\_memory\_blocks; j > i + 1; j--) memory\_map[j] = memory\_map[j - 1];  
 memory\_map[i + 1].start\_address = allocated\_start + requested\_size;  
 memory\_map[i + 1].size = memory\_map[i].size - requested\_size;  
 memory\_map[i + 1].is\_free = 1;  
 memory\_map[i + 1].owner\_pid = -1;  
 num\_memory\_blocks++;  
 memory\_map[i].is\_free = 0;  
 memory\_map[i].owner\_pid = pid;  
 memory\_map[i].size = requested\_size;  
 }  
 return allocated\_start;  
 }  
 }  
 return -1; // 分配失败  
}  
  
// 释放内存并合并空闲块  
void free\_memory(int pid) {  
 int freed\_idx = -1;  
 for (int i = 0; i < num\_memory\_blocks; i++) {  
 if (!memory\_map[i].is\_free && memory\_map[i].owner\_pid == pid) {  
 memory\_map[i].is\_free = 1;  
 memory\_map[i].owner\_pid = -1;  
 freed\_idx = i;  
 break;  
 }  
 }  
 if (freed\_idx == -1) return;  
  
 // 合并前一个块  
 if (freed\_idx > 0 && memory\_map[freed\_idx - 1].is\_free) {  
 memory\_map[freed\_idx - 1].size += memory\_map[freed\_idx].size;  
 for (int i = freed\_idx; i < num\_memory\_blocks - 1; i++) memory\_map[i] = memory\_map[i + 1];  
 num\_memory\_blocks--;  
 freed\_idx--;  
 }  
 // 合并后一个块  
 if (freed\_idx < num\_memory\_blocks - 1 && memory\_map[freed\_idx + 1].is\_free) {  
 memory\_map[freed\_idx].size += memory\_map[freed\_idx + 1].size;  
 for (int i = freed\_idx + 1; i < num\_memory\_blocks - 1; i++) memory\_map[i] = memory\_map[i + 1];  
 num\_memory\_blocks--;  
 }  
 printf(" [内存管理]: 进程 %d 内存已释放。\n", pid);  
}  
  
// --- 队列管理 ---  
void enqueue\_job(Process\* p) {  
 if (job\_queue\_count < MAX\_JOBS) job\_queue[job\_queue\_count++] = p;  
 else { free(p); printf(" [作业调度]: 后备队列已满。\n"); }  
}  
  
Process\* dequeue\_job() {  
 if (job\_queue\_count == 0) return NULL;  
 Process\* p = job\_queue[0];  
 for (int i = 0; i < job\_queue\_count - 1; i++) job\_queue[i] = job\_queue[i + 1];  
 job\_queue\_count--;  
 return p;  
}  
  
void enqueue\_ready(Process\* p) {  
 if (ready\_queue\_count < MAX\_PROCESSES\_IN\_MEMORY) {  
 ready\_queue[ready\_queue\_tail] = p;  
 ready\_queue\_tail = (ready\_queue\_tail + 1) % (MAX\_PROCESSES\_IN\_MEMORY + 1);  
 ready\_queue\_count++;  
 p->state = READY;  
 } else printf(" [进程调度]: 就绪队列已满。\n");  
}  
  
Process\* dequeue\_ready() {  
 if (ready\_queue\_count == 0) return NULL;  
 Process\* p = ready\_queue[ready\_queue\_head];  
 ready\_queue\_head = (ready\_queue\_head + 1) % (MAX\_PROCESSES\_IN\_MEMORY + 1);  
 ready\_queue\_count--;  
 return p;  
}  
  
void enqueue\_blocked(Process\* p) {  
 if (blocked\_queue\_count < MAX\_PROCESSES\_IN\_MEMORY) {  
 blocked\_queue[blocked\_queue\_count++] = p;  
 p->state = BLOCKED;  
 p->block\_start\_time = current\_time;  
 p->block\_duration = 2 + (rand() % 4); // 随机阻塞2-5个时间单位  
 printf(" [进程调度]: 进程 %d 随机阻塞 %d 个时间单位。\n", p->pid, p->block\_duration);  
 } else printf(" [进程调度]: 阻塞队列已满。\n");  
}  
  
void remove\_from\_blocked\_queue(int pid) {  
 for (int i = 0; i < blocked\_queue\_count; i++) {  
 if (blocked\_queue[i]->pid == pid) {  
 for (int j = i; j < blocked\_queue\_count - 1; j++) blocked\_queue[j] = blocked\_queue[j + 1];  
 blocked\_queue\_count--;  
 return;  
 }  
 }  
}  
  
// --- 主要逻辑 ---  
void generate\_random\_job() {  
 if (job\_queue\_count < MAX\_JOBS) {  
 Process\* new\_job = (Process\*)malloc(sizeof(Process));  
 new\_job->job\_id = next\_job\_id++;  
 new\_job->pid = -1;  
 new\_job->state = JOB\_READY;  
 new\_job->required\_cpu\_time = 5 + (rand() % 10); // CPU时间5-14  
 new\_job->current\_cpu\_time = 0;  
 new\_job->memory\_size = 50 + (rand() % 200); // 内存50-249K  
 new\_job->memory\_start = -1;  
 new\_job->time\_slice\_used = 0;  
 new\_job->block\_start\_time = 0;  
 new\_job->block\_duration = 0;  
 enqueue\_job(new\_job);  
 printf(" [作业生成]: 作业 %d 进入后备队列。\n", new\_job->job\_id);  
 } else printf(" [作业生成]: 后备队列已满。\n");  
}  
  
// 将作业加载到内存并创建进程  
void load\_jobs\_to\_memory() {  
 while (job\_queue\_count > 0 && (ready\_queue\_count + (running\_process != NULL ? 1 : 0) + blocked\_queue\_count) < MAX\_PROCESSES\_IN\_MEMORY) {  
 Process\* job\_to\_load = dequeue\_job();  
 if (job\_to\_load == NULL) break;  
   
 int mem\_start = allocate\_memory(next\_pid, job\_to\_load->memory\_size);  
 if (mem\_start != -1) { // 内存分配成功  
 job\_to\_load->pid = next\_pid++;  
 job\_to\_load->memory\_start = mem\_start;  
 enqueue\_ready(job\_to\_load);  
 printf(" [内存管理]: 作业 %d (PID: %d) 进入内存，分配 %dK @ %d。\n",  
 job\_to\_load->job\_id, job\_to\_load->pid, job\_to\_load->memory\_size, job\_to\_load->memory\_start);  
 } else { // 内存不足  
 printf(" [内存管理]: 内存不足，作业 %d 无法进入内存。\n", job\_to\_load->job\_id);  
 free(job\_to\_load); // 简化处理，直接丢弃  
 break;  
 }  
 }  
}  
  
// 检查并唤醒阻塞进程  
void check\_blocked\_processes() {  
 for (int i = blocked\_queue\_count - 1; i >= 0; i--) {  
 Process\* p = blocked\_queue[i];  
 if (current\_time >= (p->block\_start\_time + p->block\_duration)) {  
 remove\_from\_blocked\_queue(p->pid);  
 enqueue\_ready(p);  
 printf(" [进程调度]: 进程 %d 阻塞结束，进入就绪队列。\n", p->pid);  
 }  
 }  
}  
  
// 进程调度器 (时间片轮转)  
void scheduler() {  
 if (running\_process != NULL) {  
 running\_process->current\_cpu\_time++;  
 running\_process->time\_slice\_used++;  
 printf(" [调度器]: 进程 %d 运行中 (CPU: %d/%d, 片已用: %d/%d)。\n",  
 running\_process->pid, running\_process->current\_cpu\_time,  
 running\_process->required\_cpu\_time, running\_process->time\_slice\_used, TIME\_QUANTUM);  
  
 if (rand() % 10 == 0) { // 约10%概率阻塞  
 printf(" [调度器]: 进程 %d 随机触发阻塞。\n", running\_process->pid);  
 enqueue\_blocked(running\_process);  
 running\_process = NULL; return;  
 }  
  
 if (running\_process->current\_cpu\_time >= running\_process->required\_cpu\_time) { // 进程完成  
 printf(" [调度器]: 进程 %d (作业 %d) 完成。\n", running\_process->pid, running\_process->job\_id);  
 running\_process->state = TERMINATED;  
 completed\_jobs[completed\_jobs\_count++] = running\_process;  
 free\_memory(running\_process->pid);  
 running\_process = NULL;  
 } else if (running\_process->time\_slice\_used >= TIME\_QUANTUM) { // 时间片用完  
 printf(" [调度器]: 进程 %d 时间片用完，被抢占。\n", running\_process->pid);  
 running\_process->time\_slice\_used = 0;  
 enqueue\_ready(running\_process);  
 running\_process = NULL;  
 }  
 }  
  
 if (running\_process == NULL && ready\_queue\_count > 0) { // 选择下一个运行进程  
 running\_process = dequeue\_ready();  
 if (running\_process != NULL) {  
 running\_process->state = RUNNING;  
 running\_process->time\_slice\_used = 0;  
 printf(" [调度器]: 进程 %d 开始运行。\n", running\_process->pid);  
 }  
 } else if (running\_process == NULL && ready\_queue\_count == 0) {  
 printf(" [调度器]: CPU 空闲。\n");  
 }  
}  
  
// --- 显示函数 ---  
void print\_memory\_map() {  
 printf("--- 内存分配情况 (0-%dK) ---\n", USER\_MEMORY\_SIZE);  
 printf("范围 大小(K) 状态 拥有者PID\n");  
 printf("-----------------------------------\n");  
 for (int i = 0; i < num\_memory\_blocks; i++) {  
 printf("%04dK-%04dK %-8d %-4s %-8d\n",  
 memory\_map[i].start\_address,  
 memory\_map[i].start\_address + memory\_map[i].size - 1,  
 memory\_map[i].size,  
 memory\_map[i].is\_free ? "空闲" : "占用",  
 memory\_map[i].owner\_pid);  
 }  
 printf("-----------------------------------\n");  
}  
  
void print\_process\_info(Process\* p) {  
 if (p == NULL) { printf(" (空)\n"); return; }  
 printf(" PID: %d (作业 %d) | 状态: %s | CPU: %d/%d | 内存: %dK@%d\n",  
 p->pid, p->job\_id,  
 p->state == JOB\_READY ? "作业就绪" : (p->state == READY ? "就绪" :  
 (p->state == RUNNING ? "运行" : (p->state == BLOCKED ? "阻塞" : "完成"))),  
 p->current\_cpu\_time, p->required\_cpu\_time, p->memory\_size, p->memory\_start);  
}  
  
void print\_queue\_info(const char\* name, Process\*\* queue, int count, int is\_circular) {  
 printf("--- %s (%d个) ---\n", name, count);  
 if (count == 0) { printf(" (队列为空)\n"); return; }  
 if (is\_circular) {  
 for (int i = 0; i < count; i++) print\_process\_info(queue[(ready\_queue\_head + i) % (MAX\_PROCESSES\_IN\_MEMORY + 1)]);  
 } else {  
 for (int i = 0; i < count; i++) print\_process\_info(queue[i]);  
 }  
}  
  
void display\_status() {  
 printf("\n====================================\n");  
 printf("模拟时间: %ld\n", current\_time);  
 printf("====================================\n");  
 print\_queue\_info("后备作业队列", job\_queue, job\_queue\_count, 0);  
 print\_memory\_map();  
 printf("--- 进程信息 ---\n");  
 printf("运行中进程: "); print\_process\_info(running\_process);  
 print\_queue\_info("就绪队列", ready\_queue, ready\_queue\_count, 1);  
 print\_queue\_info("阻塞队列", blocked\_queue, blocked\_queue\_count, 0);  
 printf("--- 完成作业情况 (%d个) ---\n", completed\_jobs\_count);  
 if (completed\_jobs\_count == 0) { printf(" (暂无完成作业)\n"); }  
 else { for (int i = 0; i < completed\_jobs\_count; i++) printf(" 作业 %d (PID %d) 完成，CPU: %d。\n", completed\_jobs[i]->job\_id, completed\_jobs[i]->pid, completed\_jobs[i]->current\_cpu\_time); }  
 printf("====================================\n");  
}  
  
// --- 主函数 ---  
int main() {  
 init\_os();  
 for (int i = 0; i < 10; i++) generate\_random\_job(); // 初始提交10个作业  
  
 for (int tick = 0; tick < 50; tick++) { // 模拟50个时间单位  
 current\_time++;  
 printf("\n--- 时间片 %ld ---\n", current\_time);  
 if (current\_time % 5 == 0 && job\_queue\_count < MAX\_JOBS) generate\_random\_job(); // 定期生成新作业  
   
 load\_jobs\_to\_memory();  
 check\_blocked\_processes();  
 scheduler();  
 display\_status();  
  
 if (job\_queue\_count == 0 && ready\_queue\_count == 0 && blocked\_queue\_count == 0 && running\_process == NULL && completed\_jobs\_count == next\_job\_id) {  
 printf("\n所有作业已完成，模拟器停止。\n");  
 break;  
 }  
 // sleep(1); // Unix/Linux 下用 sleep (秒), 网页环境不使用  
 }  
  
 printf("\n--- 模拟结束 ---\n");  
 // 释放内存  
 for (int i = 0; i < completed\_jobs\_count; i++) free(completed\_jobs[i]);  
 for (int i = 0; i < job\_queue\_count; i++) free(job\_queue[i]);  
 while (ready\_queue\_count > 0) free(dequeue\_ready());  
 for (int i = 0; i < blocked\_queue\_count; i++) free(blocked\_queue[i]);  
 if (running\_process != NULL) free(running\_process);  
   
 return 0;  
}

## 实现效果

