|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 学生姓名 | |  | 年级专业 |  | 学号 |  |
| 项目名称 | | **段页式虚拟存储管理** | | | | |
| 项  目  实  现  原  理  简  述 | 段页式系统的基本原理，是分段和分页原理的结合，即先将用户程序分成若干个段， 再把每个段分成若干个页，并为每一个段赋予一个段名。  每个进程有一个段表，记录各个段的基址和段长。  每个段有一个对应的页表，记录该段内各页在物理内存中的映射。  虚拟地址分为段号和段内偏移。首先通过段表找到对应段的基址，再通过该段的页表找到具体页的物理地址，最后加上页内偏移得到实际的物理地址。 | | | | | |
| 完  成  情  况 | #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #define TOTAL\_MEMORY 64  // 总内存大小（单位：KB）  #define PAGE\_SIZE 1  // 页大小（单位：KB）  #define MAX\_SEGMENTS 4  // 最大段数  #define MAX\_SEGMENT\_SIZE 16  // 最大段大小（单位：KB）  #define RESIDENT\_SET\_SIZE 8  // 驻留集大小（单位：页）  // 页表条目结构  typedef struct {      int page\_number;  // 页号      int frame\_number;  // 帧号      bool valid;  // 是否有效  } PageTableEntry;  // 段表条目结构  typedef struct {      int segment\_size;  // 段大小      PageTableEntry \*page\_table;  // 页表  } SegmentTableEntry;  // 进程结构  typedef struct {      SegmentTableEntry segments[MAX\_SEGMENTS];  // 段表      int num\_segments;  // 段数      int resident\_set[RESIDENT\_SET\_SIZE];  // 驻留集      int resident\_set\_count;  // 驻留集计数  } Process;  int memory[TOTAL\_MEMORY / PAGE\_SIZE];  // 内存数组  int free\_frames[TOTAL\_MEMORY / PAGE\_SIZE];  // 空闲帧数组  int num\_free\_frames = TOTAL\_MEMORY / PAGE\_SIZE;  // 空闲帧数  // 初始化内存，将所有帧标记为空闲  void init\_memory() {      for (int i = 0; i < TOTAL\_MEMORY / PAGE\_SIZE; i++) {          free\_frames[i] = 1;  // 所有帧初始为空闲      }  }  // 为进程分配内存  void allocate\_memory(Process \*proc, int segment\_sizes[], int num\_segments) {      proc->num\_segments = num\_segments;      for (int i = 0; i < num\_segments; i++) {          proc->segments[i].segment\_size = segment\_sizes[i];          int num\_pages = (segment\_sizes[i] + PAGE\_SIZE - 1) / PAGE\_SIZE;          proc->segments[i].page\_table = (PageTableEntry \*)malloc(num\_pages \* sizeof(PageTableEntry));          for (int j = 0; j < num\_pages; j++) {              proc->segments[i].page\_table[j].page\_number = j;              proc->segments[i].page\_table[j].frame\_number = -1;  // 尚未分配              proc->segments[i].page\_table[j].valid = false;          }      }      proc->resident\_set\_count = 0;  }  // 回收进程内存  void free\_memory(Process \*proc) {      for (int i = 0; i < proc->num\_segments; i++) {          free(proc->segments[i].page\_table);      }  }  // 分配一个空闲帧  int allocate\_frame() {      for (int i = 0; i < TOTAL\_MEMORY / PAGE\_SIZE; i++) {          if (free\_frames[i]) {              free\_frames[i] = 0;              num\_free\_frames--;              return i;          }      }      return -1;  // 无空闲帧  }  // 处理缺页中断  void handle\_page\_fault(Process \*proc, int segment, int page) {      if (proc->resident\_set\_count < RESIDENT\_SET\_SIZE) {          int frame = allocate\_frame();          if (frame != -1) {              proc->segments[segment].page\_table[page].frame\_number = frame;              proc->segments[segment].page\_table[page].valid = true;              proc->resident\_set[proc->resident\_set\_count++] = frame;          }      } else {          // 使用简单的FIFO策略替换一个页          int frame\_to\_replace = proc->resident\_set[0];          for (int i = 0; i < RESIDENT\_SET\_SIZE - 1; i++) {              proc->resident\_set[i] = proc->resident\_set[i + 1];          }          proc->resident\_set[RESIDENT\_SET\_SIZE - 1] = frame\_to\_replace;          for (int i = 0; i < proc->num\_segments; i++) {              int num\_pages = (proc->segments[i].segment\_size + PAGE\_SIZE - 1) / PAGE\_SIZE;              for (int j = 0; j < num\_pages; j++) {                  if (proc->segments[i].page\_table[j].frame\_number == frame\_to\_replace) {                      proc->segments[i].page\_table[j].valid = false;                      int frame = allocate\_frame();                      if (frame != -1) {                          proc->segments[i].page\_table[j].frame\_number = frame;                          proc->segments[i].page\_table[j].valid = true;                          proc->resident\_set[RESIDENT\_SET\_SIZE - 1] = frame;                      }                      return;                  }              }          }      }  }  // 地址转换  void translate\_address(Process \*proc, int segment, int page, int offset) {      if (segment >= proc->num\_segments) {          printf("Segment Fault\n");          return;      }      if (page >= (proc->segments[segment].segment\_size + PAGE\_SIZE - 1) / PAGE\_SIZE) {          printf("Page Fault\n");          return;      }      if (!proc->segments[segment].page\_table[page].valid) {          printf("Page Fault\n");          handle\_page\_fault(proc, segment, page);      }      int frame = proc->segments[segment].page\_table[page].frame\_number;      int physical\_address = frame \* PAGE\_SIZE + offset;      printf("Physical Address: %d\n", physical\_address);  }  // 打印内存状态  void print\_memory\_state() {      printf("Free frames: ");      for (int i = 0; i < TOTAL\_MEMORY / PAGE\_SIZE; i++) {          if (free\_frames[i]) {              printf("%d ", i);          }      }      printf("\n");  }  // 打印进程状态  void print\_process\_state(Process \*proc) {      for (int i = 0; i < proc->num\_segments; i++) {          printf("Segment %d:\n", i);          int num\_pages = (proc->segments[i].segment\_size + PAGE\_SIZE - 1) / PAGE\_SIZE;          for (int j = 0; j < num\_pages; j++) {              printf("Page %d -> Frame %d (Valid: %d)\n", j, proc->segments[i].page\_table[j].frame\_number, proc->segments[i].page\_table[j].valid);          }      }      printf("Resident Set: ");      for (int i = 0; i < proc->resident\_set\_count; i++) {          printf("%d ", proc->resident\_set[i]);      }      printf("\n");  }  int main() {      init\_memory();      Process proc;      int segment\_sizes[] = {8, 12, 4, 10};  // 示例段大小（单位：KB）      allocate\_memory(&proc, segment\_sizes, 4);      print\_memory\_state();      print\_process\_state(&proc);      // 示例地址转换      translate\_address(&proc, 0, 0, 0);      translate\_address(&proc, 1, 1, 512);      translate\_address(&proc, 2, 0, 256);      translate\_address(&proc, 3, 2, 128);      print\_memory\_state();      print\_process\_state(&proc);      free\_memory(&proc);      return 0;  } | | | | | |
| 主  要  特  点 | 设计特点：  可以更好地反映程序的逻辑结构，适应复杂的内存需求  页式管理减少了内存碎片问题，段式管理避免了过多的小段和大段  可以通过段来进行内存保护和共享，不同段可以有不同的权限设置  程序功能：  内存初始化和分配  地址转换和缺页中断处理  打印内存状态和进程段表、页表信息 | | | | | |
| 尚存问  题  及  改  进  思  路 | 当前程序中只实现了缺页处理，没有实现缺段处理  只在段和页存在的情况下进行了检查，没有对超过段或页最大限制的情况进行处理  程序中的错误处理不够全面，缺少对内存分配失败、帧分配失败等情况的处理  改进：输出内存和进程的详细状态，方便调试和理解内存分配情况，增加内存分配失败、帧分配失败等情况的错误处理和日志记录 | | | | | |