|  |  |  |  |
| --- | --- | --- | --- |
| **实验名称** | **综合实验** | | |
| **学号** | **1120141826** | **姓名** | **林理露** |
| **一、实验目的**  1. 练习内核堆管理的实现  2. 理解内核线程的创建与切换  **二、实验内容**  **三、实验环境及配置方法**  **四、实验方法和实验步骤（程序设计与实现）**  <林理露>  **4.内核堆管理的实现**  本内核堆管理采用了链表式管理方法，内存在申请前需要在堆内存储一个描述当前内存块的结构体，其定义如下：  **(1)内存结构体定义**   1. #ifndef INCLUDE\_HEAP\_H\_ 2. #define INCLUDE\_HEAP\_H\_ 4. #include "types.h" 6. // 堆起始地址 7. #define HEAP\_START 0xE0000000 9. // 内存块管理结构 10. **typedef** 11. **struct** header { 12. **struct** header \*prev;    // 前后内存块管理结构指针 13. **struct** header \*next; 14. uint32\_t allocated : 1; // 该内存块是否已经被申请 15. uint32\_t length : 31;   // 当前内存块的长度 16. } header\_t; 18. // 初始化堆 19. **void** init\_heap(); 21. // 内存申请 22. **void** \*kmalloc(uint32\_t len); 24. // 内存释放 25. **void** kfree(**void** \*p); 27. // 测试内核堆申请释放 28. **void** test\_heap(); 30. #endif  // INCLUDE\_HEAP\_H\_   **(2)堆内存操作函数具体实现**  为了减少内存碎片，内存的分配使用堆来处理更加方便而高效。使用堆可以使申请的内存连续，且回收方便，不会造成内存泄漏等问题。  1)堆内存申请  由外部程序在堆上申请内存时调用   1. **void** \*kmalloc(uint32\_t len) 2. { 3. // 把申请的内存长度加上头结构的长度，以便存储内存管理头结构 4. len += **sizeof**(header\_t); 6. header\_t \*cur\_header = heap\_first; 7. header\_t \*prev\_header = 0; 9. **while** (cur\_header) { 10. // 如果当前内存块没有被申请过而且长度大于待申请的块 11. **if** (cur\_header->allocated == 0 && cur\_header->length >= len) { 12. // 按照当前长度切割内存 13. split\_chunk(cur\_header, len); 14. cur\_header->allocated = 1; 15. // 返回的时候必须将指针挪到管理头结构之后 16. **return** (**void** \*)((uint32\_t)cur\_header + **sizeof**(header\_t)); 17. } 18. // 逐次移动指针 19. prev\_header = cur\_header; 20. cur\_header = cur\_header->next; 21. } 23. uint32\_t chunk\_start; 25. // 第一次执行该函数则初始化内存块起始位置 26. // 后面每次根据当前指针加上申请的长度 27. **if** (prev\_header) { 28. chunk\_start = (uint32\_t)prev\_header + prev\_header->length; 29. } **else** { 30. chunk\_start = HEAP\_START; 31. heap\_first = (header\_t \*)chunk\_start; 32. } 34. // 检测是否需要申请内存页 35. alloc\_chunk(chunk\_start, len); 36. cur\_header = (header\_t \*)chunk\_start; 37. cur\_header->prev = prev\_header; 38. cur\_header->next = 0; 39. cur\_header->allocated = 1; 40. cur\_header->length = len; 42. **if** (prev\_header) { 43. prev\_header->next = cur\_header; 44. } 46. **return** (**void**\*)(chunk\_start + **sizeof**(header\_t)); 47. }   2)堆内存释放  由外部程序在堆上释放内存时调用   1. **void** kfree(**void** \*p) 2. { 3. // 指针回退到管理头结构，并将已使用标记置 0 4. header\_t \*header = (header\_t\*)((uint32\_t)p - **sizeof**(header\_t)); 5. header->allocated = 0; 7. // 合并内存块 8. glue\_chunk(header); 9. }   3)内存块申请  堆内部自己使用的内存申请   1. **void** alloc\_chunk(uint32\_t start, uint32\_t len) 2. { 3. // 如果当前堆的位置已经到达界限则申请内存页 4. // 必须要循环申请内存页，直到有到足够的可用内存为止 5. **while** (start + len > heap\_max) { 6. uint32\_t page = pmm\_alloc\_page(); 7. map(pgd\_kern, heap\_max, page, PAGE\_PRESENT | PAGE\_WRITE); 8. heap\_max += PAGE\_SIZE; 9. } 10. }   4)内存块释放  堆内部自己使用的内存释放   1. **void** free\_chunk(header\_t \*chunk) 2. { 3. **if** (chunk->prev == 0) { 4. heap\_first = 0; 5. } **else** { 6. chunk->prev->next = 0; 7. } 9. // 空闲的内存超过 1 页则释放掉 10. **while** ((heap\_max - PAGE\_SIZE) >= (uint32\_t)chunk) { 11. heap\_max -= PAGE\_SIZE; 12. uint32\_t page; 13. get\_mapping(pgd\_kern, heap\_max, &page); 14. unmap(pgd\_kern, heap\_max); 15. pmm\_free\_page(page); 16. } 17. }   5)内存块切分  堆内部自己使用的对内存块的切分操作   1. **void** split\_chunk(header\_t \*chunk, uint32\_t len) 2. { 3. // 切分内存块时需保证切分之后的剩余内存至少容纳一个内存管理头结构 4. **if** (chunk->length - len > **sizeof** (header\_t)) { 5. header\_t \*newchunk = (header\_t \*)((uint32\_t)chunk + len); 6. newchunk->prev = chunk; 7. newchunk->next = chunk->next; 8. newchunk->allocated = 0; 9. newchunk->length = chunk->length - len; 11. chunk->next = newchunk; 12. chunk->length = len; 13. } 14. }   6)内存块合并  堆内部自己使用的对内存块的合并操作   1. **void** glue\_chunk(header\_t \*chunk) 2. { 3. // 如果该内存块后面链接有内存块且未被使用则合并 4. **if** (chunk->next && chunk->next->allocated == 0) { 5. chunk->length = chunk->length + chunk->next->length; 6. **if** (chunk->next->next) { 7. chunk->next->next->prev = chunk; 8. } 9. chunk->next = chunk->next->next; 10. } 12. // 如果该内存块前面链接有内存块且未被使用则合并 13. **if** (chunk->prev && chunk->prev->allocated == 0) { 14. chunk->prev->length = chunk->prev->length + chunk->length; 15. chunk->prev->next = chunk->next; 16. **if** (chunk->next) { 17. chunk->next->prev = chunk->prev; 18. } 19. chunk = chunk->prev; 20. } 22. // 假如该内存后面没有内存块则直接释放掉 23. **if** (chunk->next == 0) { 24. free\_chunk(chunk); 25. } 26. }   **(3)测试堆功能**  1)代码实现   1. **void** test\_heap() 2. { 3. printk\_color(rc\_black, rc\_magenta, "kmalloc() and kfree() tests:\n\n"); 5. **void** \*address1 = kmalloc(60); 6. printk("kmalloc 60 byte at 0x%X\n", address1); 7. **void** \*address2 = kmalloc(600); 8. printk("kmalloc 600 byte at 0x%X\n", address2); 9. **void** \*address3 = kmalloc(6000); 10. printk("kmalloc 6000 byte at 0x%X\n", address3); 11. **void** \*address4 = kmalloc(60000); 12. printk("kmalloc 60000 byte at 0x%X\n\n", address4); 14. printk("kfree mem at 0x%X\n", address1); 15. kfree(address1); 16. printk("kfree mem at 0x%X\n", address2); 17. kfree(address2); 18. printk("kfree mem at 0x%X\n", address3); 19. kfree(address3); 20. printk("kfree mem at 0x%X\n\n", address4); 21. kfree(address4); 22. }   2)效果截图    **5.内核线程的创建与切换**  本内核的线程切换较为基础，仅仅涉及内核态内部的线程切换，不涉及用户态内核态间的切换，因此没有特权级转换过程。  **(1)PCB结构以及上下文信息结构定义**   1. #ifndef INCLUDE\_TASK\_H\_ 2. #define INCLUDE\_TASK\_H\_ 4. #include "types.h" 5. #include "pmm.h" 6. #include "vmm.h" 8. // 进程状态描述 9. **typedef** 10. **enum** task\_state { 11. TASK\_UNINIT = 0,    // 未初始化 12. TASK\_SLEEPING = 1,  // 睡眠中 13. TASK\_RUNNABLE = 2,  // 可运行 14. TASK\_ZOMBIE = 3,    // 僵尸状态 15. } task\_state; 17. // 内核线程的上下文切换保存的信息 18. **struct** context { 19. uint32\_t esp; 20. uint32\_t ebp; 21. uint32\_t ebx; 22. uint32\_t esi; 23. uint32\_t edi; 24. uint32\_t eflags; 25. }; 27. // 进程内存地址结构 28. **struct** mm\_struct { 29. pgd\_t \*pgd\_dir;     // 进程页表 30. }; 32. // 进程控制块 PCB 33. **struct** task\_struct { 34. **volatile** task\_state state;  // 进程当前状态 35. pid\_t    pid;               // 进程标识符 36. **void**     \*stack;            // 进程的内核栈地址 37. **struct**   mm\_struct \*mm;     // 当前进程的内存地址映像 38. **struct**   context context;   // 进程切换需要的上下文信息 39. **struct**   task\_struct \*next; // 链表指针 40. }; 42. // 全局 pid 值 43. **extern** pid\_t now\_pid; 45. // 内核线程创建 46. int32\_t kernel\_thread(**int** (\*fn)(**void** \*), **void** \*arg); 48. // 线程退出函数 49. **void** kthread\_exit(); 51. #endif  // INCLUDE\_TASK\_H\_   **(2)内核线程的创建与退出**  1)创建   1. // 内核线程创建 2. int32\_t kernel\_thread(**int** (\*fn)(**void** \*), **void** \*arg) 3. { 4. **struct** task\_struct \*new\_task = (**struct** task\_struct \*)kmalloc(STACK\_SIZE); 5. assert(new\_task != NULL, "kern\_thread: kmalloc error"); 7. // 将栈低端结构信息初始化为 0 8. bzero(new\_task, **sizeof**(**struct** task\_struct)); 10. new\_task->state = TASK\_RUNNABLE; 11. new\_task->stack = current; 12. new\_task->pid = now\_pid++; 13. new\_task->mm = NULL; 15. uint32\_t \*stack\_top = (uint32\_t \*)((uint32\_t)new\_task + STACK\_SIZE); 17. \*(--stack\_top) = (uint32\_t)arg; 18. \*(--stack\_top) = (uint32\_t)kthread\_exit; 19. \*(--stack\_top) = (uint32\_t)fn; 21. new\_task->context.esp = (uint32\_t)new\_task + STACK\_SIZE - **sizeof**(uint32\_t) \* 3; 23. // 设置新任务的标志寄存器未屏蔽中断 24. new\_task->context.eflags = 0x200; 25. new\_task->next = running\_proc\_head; 27. // 找到当前进任务队列，插入到末尾 28. **struct** task\_struct \*tail = running\_proc\_head; 29. assert(tail != NULL, "Must init sched!"); 31. **while** (tail->next != running\_proc\_head) { 32. tail = tail->next; 33. } 34. tail->next = new\_task; 36. **return** new\_task->pid; 37. }   2)退出   1. **void** kthread\_exit() 2. { 3. **register** uint32\_t val asm ("eax"); 5. printk("Thread exited with value %d\n", val); 7. **while** (1); 8. }   **(3)内核线程的切换**  此处采用很基础的切换至下一个任务的调度算法，未来也可更换至其他更加优秀的任务切换调度算法。  1)初始化   1. **void** init\_sched() 2. { 3. // 为当前执行流创建信息结构体 该结构位于当前执行流的栈最低端 4. current = (**struct** task\_struct \*)(kern\_stack\_top - STACK\_SIZE); 6. current->state = TASK\_RUNNABLE; 7. current->pid = now\_pid++; 8. current->stack = current;    // 该成员指向栈低地址 9. current->mm = NULL;      // 内核线程不需要该成员 11. // 单向循环链表 12. current->next = current; 14. running\_proc\_head = current; 15. }   2)调度   1. **void** schedule() 2. { 3. **if** (current) { 4. change\_task\_to(current->next); 5. } 6. }   3)切换至特定任务   1. **void** change\_task\_to(**struct** task\_struct \*next) 2. { 3. **if** (current != next) { 4. **struct** task\_struct \*prev = current; 5. current = next; 6. switch\_to(&(prev->context), &(current->context)); 7. } 8. }   4)具体的切换操作   1. [global switch\_to] 3. ; 具体的线程切换操作，寄存器的保存与恢复 4. switch\_to: 5. mov eax, [esp+4] 7. mov [eax+0],  esp 8. mov [eax+4],  ebp 9. mov [eax+8],  ebx 10. mov [eax+12], esi 11. mov [eax+16], edi 12. pushf 13. pop ecx 14. mov [eax+20], ecx 16. mov eax, [esp+8] 18. mov esp, [eax+0] 19. mov ebp, [eax+4] 20. mov ebx, [eax+8] 21. mov esi, [eax+12] 22. mov edi, [eax+16] 23. mov eax, [eax+20] 24. push eax 25. popf 27. ret 29. ;   **(4)实际效果测试**  1)代码实现   1. **int** **thread**(**void** \*arg) 2. { 3. **while** (1) { 4. **if** (flag == 1) { 5. printk\_color(rc\_black, rc\_green, "S"); 6. flag = 0; 7. } 8. } 10. **return** 0; 11. } 13. **void** kern\_init() 14. { 15. ... 17. kernel\_thread(**thread**, NULL); 19. **while** (1) { 20. **if** (flag == 0) { 21. printk\_color(rc\_black, rc\_red, "O"); 22. flag = 1; 23. } 24. } 26. ... 27. }   2)效果截图    </林理露>  **五、实验结果和分析**  **六、讨论、心得** | | | |