|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **LINUX实验报告**  Project 2 | | | | |
|  | 课 程 名 称： | ： | Linux操作系统实践 |  |
| 年 级： | ： | 2018级 |
| 组 内 成 员： | ： | 朱张弛 10185102126 |
| ： | 汪子凡 10185102153 |
| ： | 白彬惠 10185102121 |
| 2020年11月 | | | | |

目录

**[（一） Project 2a:Malloc & Free 3](#_Toc7959)**

[一、 实验目的 3](#_Toc30189)

[二、 问题重述 3](#_Toc14251)

[三、 算法设计 4](#_Toc14814)

**[（二） Project 2b:xv6 Scheduler 8](#_Toc20078)**

[一、 实验目的 8](#_Toc11000)

[二、 问题重述 8](#_Toc17305)

[三、 算法设计 8](#_Toc14373)

[四、 测试结果 14](#_Toc933)

1. **Project 2a:Malloc & Free**
2. 实验目的
3. To understand the nuances of building a memory allocator.
4. To do so in a performance-efficient manner.
5. To create a shared library.
6. 问题重述

In this project, you will be implementing a memory allocator for the heap of a user-level process. Your functions will be to build your own malloc() and free().

Memory allocators have two distinct tasks. First, the memory allocator asks the operating system to expand the heap portion of the process's address space by calling either sbrk or mmap. Second, the memory allocator doles out this memory to the calling process. This involves managing a free list of memory and finding a contiguous chunk of memory that is large enough for the user's request; when the user later frees memory, it is added back to this list.

This memory allocator is usually provided as part of a standard library and is not part of the OS. To be clear, the memory allocator operates entirely within the virtual address space of a single process and knows nothing about which physical pages have been allocated to this process or the mapping from logical addresses to physical addresses; that part is handled by the operating system.

When implementing this basic functionality in your project, we have a few guidelines. First, when requesting memory from the OS, you must use **mmap()** (which is easier to use than sbrk()). Second, although a real memory allocator requests more memory from the OS whenever it can't satisfy a request from the user, your memory allocator must call mmap() only **one** time (when it is first initialized).

Classic malloc() and free() are defined as follows:

* void \*malloc(size\_t size), which allocates size bytes and returns a pointer to the allocated memory. The memory is not cleared.
* void free(void \*ptr), which frees the memory space pointed to by ptr, which must have been returned by a previous call to malloc() (or calloc() or realloc()). Otherwise, or if free(ptr) has already been called before, undefined behaviour occurs. If ptr is NULL, no operation is performed.

For simplicity, your implementations of mem\_alloc() and mem\_free() should basically follow what malloc() and free() do; see below for details.

You will also provide a supporting function, mem\_dump(), described below; this routine simply prints which regions are currently free and should be used by you for debugging purposes.

1. 算法设计
2. 声明、结构体及全局变量

int m\_error;   
int has\_inited = 0;   
   
typedef struct node   
{   
 int size;   
 struct node \*next;   
}node;   
   
node\* headfree;   
int tot\_size;

pg\_size记录一个页的大小，以便舍入存储区域大小为页的倍数。

m\_error记录错误信息。

has*inited保证mem*init只被调用一次。

node结构体为链表中存储的节点，指向一块连续的存储区域。node中size为用户可用的区域的大小。

headfree为链表头指针。

tot\_size为总共申请的存储区域大小，每次用户申请前需检查是否超出总大小。

1. 初始化

mem\_init中输入需要的存储大小，并申请内存，初始化内存，该函数仅调用一次。

int mem\_init(int size\_of\_region)   
{   
 m\_error = 0;   
 pg\_size = getpagesize();   
 if(size\_of\_region <= 0 || has\_inited){   
 m\_error = E\_BAD\_ARGS;   
 return -1;   
 }   
 size\_of\_region += (pg\_size - size\_of\_region % pg\_size) % pg\_size;   
 int fd = open("/dev/zero", O\_RDWR);   
 headfree = mmap(NULL, size\_of\_region, PROT\_READ | PROT\_WRITE, MAP\_PRIVATE, fd, 0);   
 if(headfree == MAP\_FAILED){ perror("mmap"); exit(1); }   
 close(fd);   
 tot\_size = size\_of\_region;   
 headfree->size = tot\_size - sizeof(node);   
 has\_inited = 1;   
 return 0;   
}

1. 分配

mem\_alloc申请size大小的连续存储区域，并选择查找方式。

以下代码根据不同查找方式找到符合条件的存储区域的头指针。

case M\_BESTFIT:   
{   
 int mn = tot\_size + 10;   
 while(p != NULL){   
 if(p->size >= size){   
 if(p->size < mn){   
 mn = p->size;   
 res = p;   
 }   
 }   
 p = p->next;   
 }   
 break;   
}   
case M\_WORSTFIT:   
{   
 int mx = -1;   
 while(p != NULL){   
 if(p->size > mx){   
 mx = p->size;   
 res = p;   
 }   
 p = p->next;   
 }   
 if(res != NULL && res->size < size)res = NULL;   
 break;   
}   
case M\_FIRSTFIT:   
{   
 while(p != NULL){   
 if(p->size >= size){   
 res = p;   
 break;   
 }   
 p = p->next;   
 }   
 break;   
}

得到指针后进行后续处理，由于以链表方式存储，因此需要修改链表头指针，前驱指针，后继指针，并判断是否加入新的指针，修改存储区域大小。

if(res->size > size + sizeof(node)){   
 node\* np = res + size + sizeof(node);   
 np->size = res->size - size - 2 \* sizeof(node);   
 np->next = res->next;   
 if(res == headfree){   
 headfree = np;   
 }   
 else{   
 p = headfree;   
 while(p != NULL){   
 if(p->next == res){   
 p->next = np;   
 break;   
 }   
 p = p->next;   
 }   
 }   
 }   
 else{   
 if(res == headfree){   
 headfree = res->next;   
 }   
 else{   
 p = headfree;   
 while(p != NULL){   
 if(p->next == res){   
 p->next = res->next;   
 break;   
 }   
 p = p->next;   
 }   
 }   
 }   
 res->size = size;   
 return (void\*)res;   
}

1. 释放

mem\_free函数释放存储空间，并合并连续的存储区域，判断如果两个连续指针指向连续的存储区域，则合并两个指针。

int mem\_free(void \*pptr)   
{   
 if(pptr == NULL)return 0;   
 node\* ptr = (node\*)pptr;   
 if(ptr < headfree){   
 if(headfree < ptr + ptr->size + sizeof(node)){   
 return -1;   
 }   
 node\* tmp = headfree;   
 headfree = ptr;   
 ptr->next = tmp;   
 if(ptr->next != NULL && ptr + ptr->size + sizeof(node) == ptr->next){   
 ptr->size += ptr->next->size + sizeof(node);   
 ptr->next = ptr->next->next;   
 }   
 }   
 else{   
 node\* p = headfree;   
 while(p != NULL){   
 if(p < ptr && (p->next > ptr))break;   
 p = p->next;   
 }   
 if(p == NULL)return -1;   
 if(p + p->size + sizeof(node) > ptr || (p->next != NULL && ptr + ptr->size + sizeof(node) > p->next))   
 return -1;   
 ptr->next = p->next;   
 p->next = ptr;   
 if(ptr->next != NULL && ptr + ptr->size + sizeof(node) == ptr->next){   
 ptr->size += ptr->next->size + sizeof(node);   
 ptr->next = ptr->next->next;   
 }   
 if(p + p->size + sizeof(node) == ptr){   
 p->size += ptr->size + sizeof(node);   
 p->next = ptr->next;   
 }   
 }   
 return 0;   
}

1. **Project 2b:xv6 Scheduler**
2. 实验目的
3. To understand code for performing context-switches in the xv6 kernel.
4. To implement a basic MLFQ scheduler.
5. To create system call that extract process states.
6. 问题重述

In this project, you'll be implementing a simplified **multi-level feedback queue (MLFQ)**scheduler in xv6.

The basic idea is simple. Build an MLFQ scheduler with four priority queues; the top queue (numbered 3) has the highest priority and the bottom queue (numbered 0) has the lowest priority. When a process uses up its time-slice (counted as a number of ticks), it should be downgraded to the next (lower) priority level. The time-slices for higher priorities will be shorter than lower priorities. The scheduling method in each of these queues will be round-robin, except the bottom queue which will be implemented as FIFO.

1. 算法设计

观察原来的xv6代码，发现原来的 schedule() 函数是进程调度函数，每个timer tick(即一次for循环)从 ptable 的 proc 数组里选取一个下标最靠前的处于 runnable 的进程进行调度。

对题目的要求进行总结：

①有4个队列，队列3优先级最高(每次新建立的进程进入队列3)，队列0优先级最低。前三个队列中采用时间片轮转法(时间片分别为8, 16, 32 time ticks)，最后一个队列采取FIFO。

如果在前三个队列时间片用完了，就降低一个优先级，进入序号低的队列。

②每次timer tick结束之后，进程不管有没有用完时间片都要 yield 一次放弃CPU，如果此时有优先级更高的进程，就会选取优先级高的进程，否则继续执行，直到时间片用完或者执行完毕/阻塞。所以我们要实现的是**抢占式多级反馈队列**。

③对于后3个队列的进程，没有被调度的时间分别大于160,320,500 timer ticks 时，便出现饥饿现象，会升一个优先级，进入序号高的队列。

为了实现以上功能，可以对 结构体proc扩充定义：

**struct** proc {

  uint sz;                     // Size of process memory (bytes)

  pde\_t\* pgdir;                // Page table

**char** \*kstack;                // Bottom of kernel stack for this process

**enum** procstate state;        // Process state

**volatile** **int** pid;            // Process ID

**struct** proc \*parent;         // Parent process

**struct** trapframe \*tf;        // Trap frame for current syscall

**struct** context \*context;     // swtch() here to run process

**void** \*chan;                  // If non-zero, sleeping on chan

**int** killed;                  // If non-zero, have been killed

**struct** file \*ofile[NOFILE];  // Open files

**struct** inode \*cwd;           // Current directory

**char** name[16];               // Process name (debugging)

**int** priority;

**int** waited\_ticks[NPriority];

**int** used\_ticks[NPriority];

**int** que0\_ticks;

};

由于进行了扩充，所以在allocproc() 中要加入相应的初始化:

p->priority = NPriority - 1;

  p->que0\_ticks = 0;

**for**(**int** i = 0; i < NPriority; i++){

      p->waited\_ticks[i] = 0;

      p->used\_ticks[i] = 0;

  }

其中，waited\_ticks 记录在每个队列里已经等待的 ticks(**waited\_ticks不会累加**，所以当某个进程被调用时，waited\_ticks会被置为0)，waited\_ticks 的记录有3点用处：

①对应于之后系统调用中 pstat 里的 wait\_ticks[NPROC][4]；

②来判断该进程在该队列中是否发生饥饿；

③前三个队列是用时间片轮转法实现调度的，当选取某个队列 q 中的进程进行调度时，对应于选取 wait\_ticks[q] 最大的进程，即**每次选取队列中等待时间最长的进程**。

因为对于一个队列来说，我们每次都选取队首的元素，而队首的元素一定是这个队列中等待时间最长的，示例如下：

|  |  |  |
| --- | --- | --- |
| P1 | P2 | P3 |

按照入队顺序为P1，P2，P3，那P1的等待时间一定是最长的，但**如果一个进程时间片没有用完被抢占了或者被阻塞了不能继续占有CPU，相当于从队首删去，加入队尾。**

比如P1执行了一半时间片，但是CPU被抢占了，当优先级更高的进程执行完毕之后，执行的顺序应该如下：

|  |  |  |
| --- | --- | --- |
| P2 | P3 | P1 |

但是对于第0个队列，是不能根据wait\_ticks 进行判断的，因为队首的 wait\_ticks 可能很低(不久前被执行)，队尾的 wait\_ticks也可能很低(刚刚入队)。

也不能拿 used\_ticks 进行判断。比如队列中有一个进程A执行了100 ticks, 这时候进入了一个进程B，按照FIFO应该先执行完A再执行B，但是若是B之前在第0个队列中执行了200 ticks，是发生饥饿了之后进入前一个队列，现在又回来了，那么 B 的 used\_ticks 就会比 A 大。

所以在结构体中新定义了一个que0\_ticks，记录这个进程进入第0个队列的总时间，即无论处于runnable 还是 running 都要计时，**当发生饥饿进入上一个队列时，计时会清空。当进入第0个队列，计时开始。**每次选取que0\_ticks最大的，即满足FIFO。

所以scheduler修改如下(详情见注释)：

**void**

scheduler(**void**)

{

**struct** proc \*p;

**struct** proc \*pr[NPriority];

**int** nd\_sched = 0;

  proc = 0;

**for**(;;){

    // Enable interrupts on this processor.

    sti();

    // Loop over process table looking for process to run.

acquire(&ptable.lock);

// nd\_sched=0表示当前进程时间片没有用完,也没有进程升队,只需要查看有没有优先级更高的进程加入

**if**(proc != NULL && !nd\_sched){

**for**(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

**if**(p->state != RUNNABLE)**continue**;

**if**(p->priority > proc->priority){

                  nd\_sched = 1;

**break**;

              }

        }

}

// nd\_sched = 1进程时间片用完，或者有进程升队，或者有优先级更高的进程加入

**if**(proc == NULL || nd\_sched){

**for**(**int** i = 0; i < NPriority; i++)pr[i] = NULL;

**for**(p = ptable.proc; p < &ptable.proc[NPROC]; p++){

**if**(p->state != RUNNABLE)**continue**;

**if**(p->priority == 0){ //第0个队列选择入队时间最长的

**if**(pr[0] == NULL || p->que0\_ticks > pr[0]->que0\_ticks){

                      pr[0] = p;

                  }

              }

**else**{ //1,2,3队列选择等待时间最长的

**if**(pr[p->priority] == NULL

                    || p->waited\_ticks[p->priority] > pr[p->priority]->waited\_ticks[p->priority]){

                      pr[p->priority] = p;

                  }

              }

        }

        nd\_sched = 0;

    }

**for**(**int** i = NPriority - 1; i >= 0; i--){ //选择优先级最高的队列

**if**(pr[i] != NULL){

            proc = pr[i];

**break**;

        }

    }

**if**(proc == NULL){

        release(&ptable.lock);

**continue**;

    }

**for** (p = ptable.proc; p < &ptable.proc[NPROC]; p++){ //第0个队列增加入队时间

**if**(p->priority == 0 && (p->state == RUNNABLE || p->state == RUNNING)){

            p->que0\_ticks++;

        }

**if**(p->pid == proc->pid){

**continue**;

    }

**if** (p->state == RUNNABLE) { //1,2,3队列非调度进程增加等待时间

        p->waited\_ticks[p->priority]++;

    }

**if** ((p->priority == 0 && p->waited\_ticks[p->priority] == 500) //判断饥饿

    || (p->priority != 0 && p->waited\_ticks[p->priority] == 10\*mxtick[p->priority])) {

**if**(p->priority != NPriority - 1){

            p->priority++;

            p->que0\_ticks = 0;

        }

    }

    }

**for**(**int** i = 0; i < NPriority; i++){ //被调度队列等待时间清空

        proc->waited\_ticks[i] = 0;

    }

    proc->used\_ticks[proc->priority]++;

**if**(proc->used\_ticks[proc->priority] % mxtick[proc->priority] == 0 && proc->priority != 0){ //判断是不是到达时间片

    nd\_sched = 1;

    proc->priority--;

    }

    switchuvm(proc); //开始调用

    proc->state = RUNNING;

    swtch(&cpu->scheduler, proc->context);

    switchkvm();

**if**(proc->state != RUNNABLE){ //判断调用完状态

    nd\_sched = 1;

    }

    release(&ptable.lock);

  }

}

补充完了 schedule() 函数，还有一个任务是增加系统调用getpinfo()，

①这个函数的 body 定义在 proc中，需要将传入的结构体填入进程的信息，成功返回0，失败返回-1，如下实现：

**int** getpinfo(**struct** pstat \* pst){

**if**(pst == NULL)**return** -1; //-1表示失败

**struct** proc \*p;

**int** i;

**for**(i = 0, p = ptable.proc; i < NPROC && p < &ptable.proc[NPROC]; i++, p++){

**if**(p->state == SLEEPING || p->state == RUNNABLE || p->state == RUNNING)

            pst->inuse[i] = 1;

**else** pst->inuse[i] = 0;

        pst->pid[i] = p->pid;

        pst->priority[i] = p->priority;

        pst->state[i] = p->state;

**for**(**int** j = 0; j < NPriority; j++){ //用新增的变量去初始化pst

            pst->ticks[i][j] = p->used\_ticks[j];

            pst->wait\_ticks[i][j] = p->waited\_ticks[j];

        }

    }

**return** 0;

}

由于使用了pst,需要在proc.c 加入头文件：

#include "pstat.h"

②在sysfile.c加入系统调用的函数 sys\_getpinfo，为了将参数从用户空间传入内核空间，使用了argptr函数：

**int**

sys\_getpinfo(**void**)

{

**struct** pstat \*p;

**if**(argptr(0, (**void**\*)(&p), **sizeof**(\*p))<0)  **return** -1;

**return** getpinfo(p);

}

③在usys.S 中增加 sys\_getpinfo的宏定义：

SYSCALL(getpinfo)

④上面要用到 sys\_getpinfo, 所以在 include/syscall.h 中添加定义。

#define SYS\_getpinfo 23

⑤在IDT表中添加相应的中断描述符，在kernel/syscall.c中添加。

[SYS\_getpinfo] sys\_getpinfo,

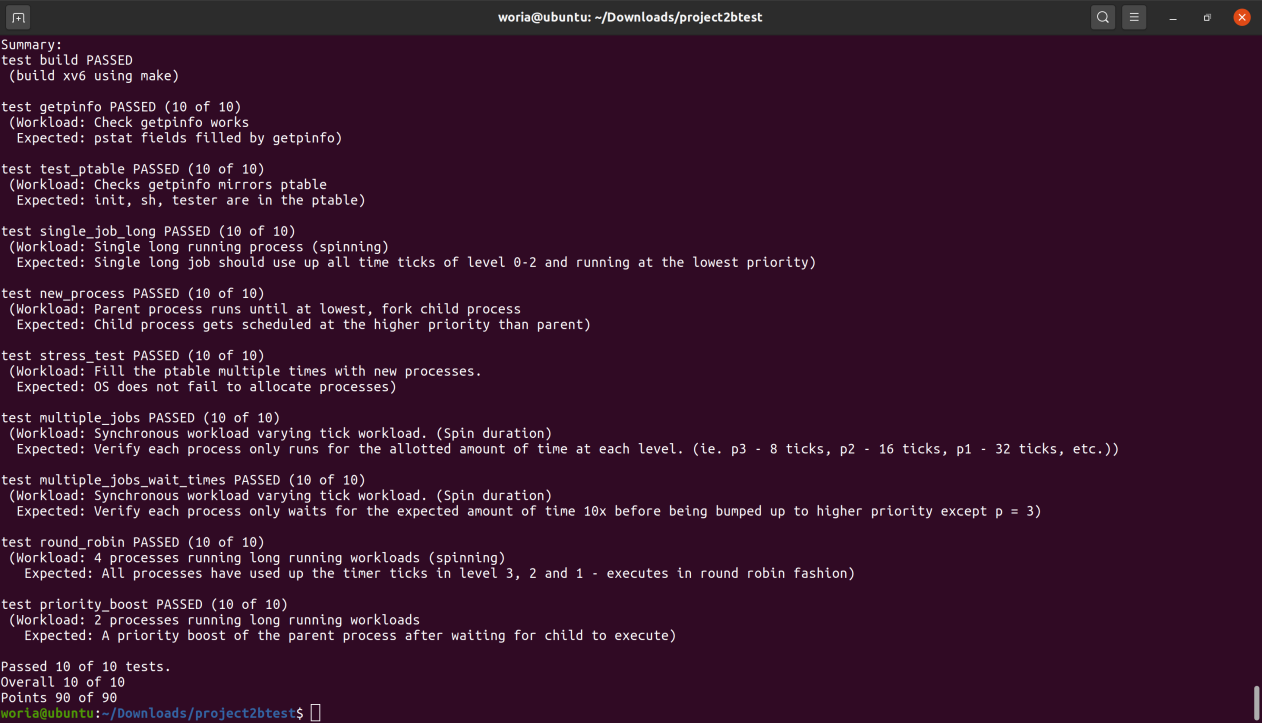
⑥添加了上述中段描述符之后，是要在 kernel/sysfunc.h 添加声明。

**int** sys\_getpinfo(**void**);

⑦要供用户使用的话，在user.h 加入 pstat和int getpinfo(struct pstat\*)

int getpinfo(struct pstat\*);

1. 测试结果

****