Operating System Labs Project 2

Project 2a: Dynamic Memory Allocation

Part 1 概述：

1. 头部节点的定义

在实验的最开始要解决的问题就是如何定义每个块的头部节点,考虑到在 malloc 和 free 的时候会扩展和合并块区域，因此比较理想的方法是为每个头部节点内设置 size,flag,pre,next 四个部分，size 表示块的大小，flag 标识这个块是否已被分配，pre 与 next 分别连接上一个块与下一个块。但在实验要求中写了“The maximum size of such a header is 16 bytes.”通过 sizeof 检测了一下，一个 int 与一个指针结构的封装结构体正好为 16 个字节。查阅发现 64 位 Linux 下指针变量占 8 个字节，而操作系统本身又会以 8 位进行对齐（正如这次实验要求的一样），因此这样组合出的结构体已经占用了 16 个字节。

显然只通过 size 与 next 两个属性是不能够满足要求的,甚至没有变量来标识块是否空闲。PPT 中给出的一个参考方案是定义另一种头部节点，内部包含了 size 与 magic。并在 malloc 与 free 时对一个块的头部节点进行转换。然后通过 magic 地址处的值是否等于预先设定的值来检验这个区域是空闲的还是已经被分配了的。这是一个可行的方案，但在实践中发现有部分缺点:

① 转换过程使代码繁琐，尽管这对时间开销的影响很小。
② 在 malloc 时，一个 node 转换为 head 之后，这个 node 要从链表中移除，这意味着需要时刻关注每个节点的前驱节点。
③ 在 free 时，同样需要关注每个节点的前驱节点来将节点重新加入到空闲区域的链表当中，但由于占用的块不在链表中，寻找其前驱和后继节点不再是遍历链表那么简单了，大大增大了开销。

因此本次实验采用了另一种方案:只使用结构体 node，其定义如下:

```
typedef struct _node_t
{
    int size;
    struct _node_t *next;
}node;
```

实验中没有用到_header_t 的结构，为了区分一个块是否已经被分配，我们给 size 的含义进行附加:

① 当 size == 0 的时候，表示这个区域为已分配的区域，要获得其大小，可以通过其下一个节点的首地址减去当前节点的地址再减去 16 个 bytes。

② 当 size != 0 的时候，size 表示的就是这个区域的大小

可以发现，如果保证了每个块都在链表中，那么即使没有 size 属性，也能够表示每个块的大小。事实上，在这种模式下 size 已经是一个标志位，而非是属性，只是我们可以利用标志位的值等于块的大小来优化代码和增加可读性。这样的模式在 malloc 操作和 free 操作时表现的非常简约，我们将在后文详细阐述。

2. 整体流程

整个函数主要分为 4 个函数，主要的设计在前三个函数中。上文已经阐述了无论块是否被分配，所有的节点都采用统一的头部节点 node,因此具有统一性，在不考虑所有异常的情况下，我们可以以链表的角度来看待整个逻辑。

mem_init(): 初始化链表的头结点 head

mem_alloc():根据算法找到一个节点，在其前面插入一个节点

mem_free(): 搜索节点，将这个节点与相邻节点进行合并操作。

在这样的思想下加入对异常(内存不足，节点不存在)的处理，便可以完成整个代码的流程，下面我们对每个函数进行说明。

Part2 源码：

1. 全局变量

我们在 mem.c 中设置如下全局变量以供代码正常运行:

(1) const int nodeSize = sizeof(struct _node_t); //node 节点的大小

(2) int pageSize; //页的大小,通过 getpagesize()函数获得

(3) int m_error; //头文件中的 extern int m_error,表示错误类型

(4) int once = 0; //标志位，表示 mem_init()被成功调用的次数

(5) int sumSize; //整个块的大小

(6) node* _head_,*_tail_; //整个块的首地址与末尾地址

2. int mem_init(int size_of_region)函数说明

(1) 置 m_error = 0 来初始化

(2) 检测给定参数是否小于等于 0 或者在被成功调用之后再次被调用

(3) 若是,设置 m_error = E_BAD_ARGS; 返回-1

(4) 若否，首先对 size_of_region 作向上取整，使得其是 pageSize 的整

数倍。

```
pageSize = getpagesize();
if(size_of_region%pageSize != 0)
    size_of_region += pageSize-size_of_region%pageSize;
```

(5) 然后通过 mmap 分配指定大小的内存，若失败，返回异常。
(6) 若成功，初始化头结点，设置大小，初始化所有全局变量
(7) 设置 once 标志位为 1，返回 0

3. void * mem_alloc(int size, int style) 函数说明
(1) 置 m_error = 0 来初始化
(2) 检测_head_是否为 NULL，若是，代表 mem_init 尚未调用，设置 m_error = E_NO_SPACE，返回 NULL
(3) 同样的，我们要对分配内存的大小向上取整，使得其为 8 的倍数:

```
if(size%8 != 0)
    size += 8-(size%8);
```

(4) 然后，无论算法如何，按照以下步骤操作
(5) 首先找到第一个大于等于 size 的块，并设置一个参数 target 等于这个块的地址。
(6) 若没有找到，那么设置 m_error = E_NO_SPACE，返回 NULL
(7) 若找到了，再根据算法不同作不同处理:
① 若是 M_BESTFIT，那么从 target 开始不断寻找大于 size 但是小于 target->size 的块，然后更新 target
② 若是 M_WORSTFIT，那么从 target 开始不断寻找大于 target->size 的块，然后更新 target
③ 若是 M_FIRSTFIT，那么 target 就是目标
(8) 经过(7)之后，无论何种算法，target 都是目标块了，此时我们再统一进行插入操作。
(9) 在 target 和 target->next 之间新生成一个节点,然后连接三个节点，同时设置各个点的 size:

```
node *originalNext = target->next;                  //记录原生的后继节点
target->next = (void*)target+size+nodeSize;         //连接前两个点
target->next->size = target->size-(size+nodeSize); //设置新节点 size
target->next->next = originalNext;                  //连接后两个点
target->size = 0;                                   //标识为已经占用
```

(10) 返回头部节点的末尾地址: return (void*)target+nodeSize;

从上述流程中可以发现，在这样的模式下，因为被占用区的 size 被设置为 0，因此可以自动在遍历链表时被忽略，而被占用区本身又

在链表中，连接操作变得非常的简约。此外连接时并不需要前驱节点，操作进一步简化。

我们再讨论零头：一个块因为要包含头部节点，因此当我们对一个空闲块进行分割之后，如果剩余部分的大小小于等于 16 字节，那么显然是没有意义的，因此我们的处理方式是将其一同赋予前面的分割区域，因为对于分配区的 size 获取方式是地址相减，所以在 free 时不需要做额外操作也能够正确的释放了。

4. int mem_free(void * ptr)函数说明

(1) 置 m_error = 0 来初始化

(2) 如果 ptr == NULL，返回 0

(3) 通过 node *hptr = (void*)ptr-nodeSize 获取其头部节点的首地址
如果 _head_ == NULL || size != 0 那么设置 m_error = E_BAD_POINTER，返回-1

(4) 若这是一个合法的被分配的指针，那么搜索这个节点的前驱节点

(5) 如果前驱节点是空闲区，那么与 ptr 进行合并

(6) 如果 ptr->next 是空闲区，那么与 ptr 进行合并

(7) 返回 0

合并的操作也同样很简单，首先通过块的后继节点来重新获取 size,如果不存在后继节点，那么可以通过整个块的末尾地址来计算:

```
if(late->next != NULL)
    late->size = (void*)late->next-(void*)late-nodeSize;
else
    late->size = (void*)_head_+sumSize-(void*)late-nodeSize;
```

然后再进行合并，以前驱节点为例

```
pre->size += late->size+nodeSize; //合并大小
pre->next = late->next;           //和链表的删除操作非常类似
```

这样就完成了整个 free 的逻辑。

5. void mem_dump()函数说明

该函数打印每个块的大小，并以 available 和 alloced 来表面这个块的状态，对于块的大小计算，与 free 中节点 size 的恢复类似。

```
while(head != NULL)
{
    if(head->size != 0)printf("available:%d\n",head->size);
    else
    {
        if(head->next != NULL) printf("allocted:%ld\n",(void*)head->next-(void*)head-nodeSize);
        else printf("allocted:%ld\n",(void*)_head_+sumSize-(void*)head-nodeSize);
    }
    head = head->next;
}
```

Project 2b: xv6 Scheduling

Part 0 涉及文件:

1. xv6/include/pstat.h
2. xv6/kernel/sysfile.c
3. xv6/user/usys.S
4. xv6/include/syscall.h
5. xv6/kernel/syscall.c
6. xv6/kernel/sysfunc.h
7. xv6/user/user.h
8. xv6/kernel/proc.h
9. xv6/kernel/proc.c

其中 1-7 为添加系统调用，8-9 为抢占式多级反馈队列的实现。

Part 1 概述：

1. 首先总结题目要求:

① 整体设计一个四级队列反馈调度，优先级从高到低
② 前三级为循环队列，最后一级为 FIFO
③ 允许优先级抢占
④ 降低优先级的唯一条件:在某个队列中完成了一个队列的时间片
⑤ 升高优先级的唯一条件:在某个队列中等待时间 = 规定饥饿时间

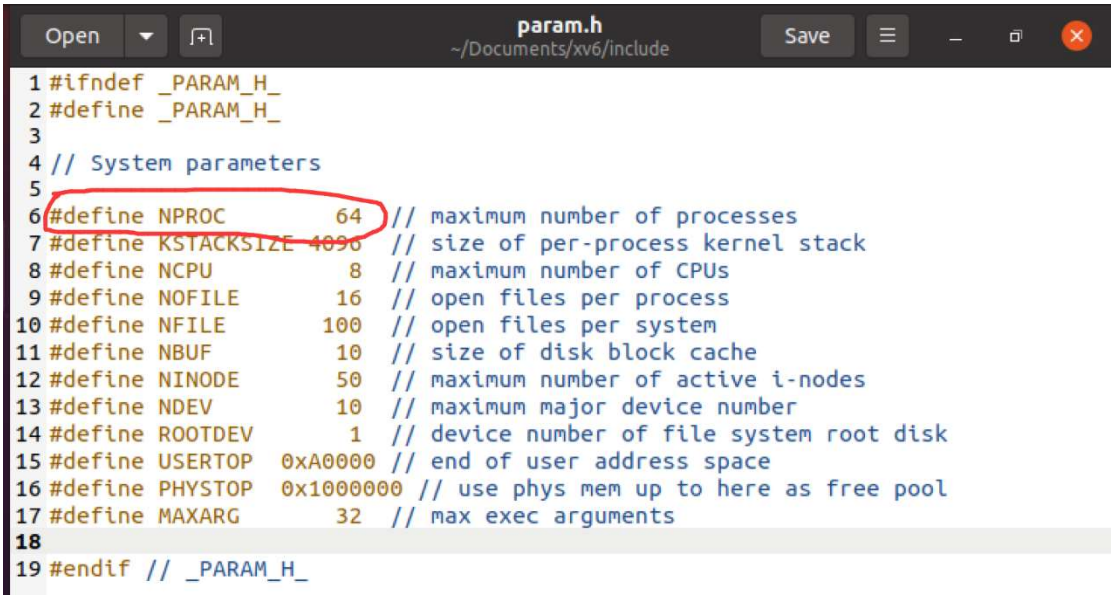
2. 在构建代码前，我们对于队列设计有两种思路:

(1) 显式的创建四个数组/队列，然后在操作的过程中将进程在各个队列之间转移。

每次处理的进程从 3 级至 0 级开始遍历,遇到一个非空队列就停止，然后执行这个队列的首个进程。对于优先级的升降，需要修改进程所在的队列编号。

(2) 不创建数组/队列，我们注意到，即使在队列中，我们所需要的信息只有两个：所在队列的优先级，在队列中的位置。至于等待时间，处理的时间片，无论是否创建队列，都需要参数来表达。

对比两个方法：对于优先级的表达，二者都需要一个参数 priority,因为队列本身不附带优先级属性，即使使用 4 个队列，仍需要为每个节点附件 priority 来查询(除非通过地址来判断，但是那样必须使用数组来保持连续，而不适用链表的队列维护起来相当繁琐，且判断需要大量的 if 分支，使得代码冗余)。对于在队列中的位置表达，显式的队列是确实有优势的，但是可以发现:等待时间同样可以表达数组的位置，等待时间最长的节点必然是在队列的首段，只不过每次需要进行枚举才能确定，查看页表的长度[NPROC]，在 xv6/include/param.h 中的 line 6 发现



也就是说，最大进程数为 64，因此通过等待时间查询队首的方法在数据规模上式可行的。并且不需要对 4 个队列进行维护，最终我们决定采用方法 2。我们将在 Part2 部分详细阐述实现方法。

Part 2 多级反馈队列实现：

1. 为 proc 结构体附加新属性

基于 Part1 中的阐述，我们需要对 proc 附加新属性来满足要求，
我们参照了给定头文件 pstat.h 中的 pstat 结构体，设计了三个属性
(1) int priority：即该进程的优先级,值域为{0,1,2,3},对应于 pstat 中的 priority[index]属性

(2) int wait[4]: wait[i]表示该进程到达 i 号队列开始的等待时间，注意是最后一次到达，也就是说，只要进程发生优先级的升降，等待时间都会被清 0，我们只关心进程在现在所在队列的等待时间。对应于 pstat 属性中的 wait_ticks[index][i]

(3)int done[4]: done[i]表示该进程在队列 i 中一共处理过多少时间片，与 wait 属性不同，这个值是一个累加值，不随优先级的改变而重置。

2. 算法流程:

(1) allocproc 函数部分: 为新属性增加初始化

```
p->priority = 3; //新进程最开始位于最高优先级队列
for(i = 0;i < 4; i++)
```

```
{
    p->wait[i] = 0;
    p->done[i] = 0;
}
```

简单分析一下 allocproc 函数，该函数的流程是枚举整个页表，然后找到第一个 state 为 UNUSED 的进程，对其进行初始化并返回。此时进程的状态时 EMBRYO，而 scheduler 函数则是一个无线循环，捕捉处于 RUNNABLE 的进程，因此猜想这两个函数是并发进行的，一个不断初始化更新进程的页表，另一个不断处理位于队列首段的进程。

(2) scheduler 函数部分:实现抢占式多级反馈队列

大体查看 scheduler 函数的结构，是一个无限循环的结构，而抢占式多级反馈队列使用的是 RR——时间片轮转算法，因此一次 for 循环应该就是对应于一个时间切片，这个函数本身的做法就是枚举页表，找到处于就绪态的进程然后用 switchuvm 函数来处理。因此要修改的就是这个部分。

① 首先修改的就是处理哪个进程，原函数不需要任何处理，找到处于 RUNNABLE 的进程进行处理。我们假设上一个时间片处理的进程为 proc，那么现在需要处理的进程在四种情况下可能需要改变 proc:

(a) 被抢占，更高优先级的进程加入了

```
for(p = ptable.proc;p < &ptable.proc[NPROC];p++)
{
    if(p->state == RUNNABLE && p->priority > proc->priority)
    {
        flag = 1;
        break;
    }
}
```

(b) proc 已全部完成

```
 if(proc->state != RUNNABLE)
    flag = 1;
```

(c) proc 已经完成所在队列的一个时间片

```
if(proc->done[proc->priority]%ticks[proc->priority] == 0 )
    flag = 1;
```

(d) proc 为空值

```
if(proc == NULL)
    flag = 1
```

针对上述四种条件，设置一个参数 flag，flag = 0 表示上述四种情况均未发生，即新一次的 for 循环所处理的进程等于上一个 for 循环的进程。而一旦上述四种情况发生任意一种，那么就要进行 proc 的替换。

替换的目标很明确：最高非空优先级的队列的首端。

我们设置一个参数 tar[4],分别表示 4 个队列的首端。然后枚举整个页表，对于每个优先级取其 wait 值最大的即可。

```
if(flag)
{
    for(i = 0;i < 4;i++)    tar[i] = NULL;
    for(p = ptable.proc;p < &ptable.proc[NPROC];p++)
    {
        if(p->state != RUNNABLE) continue;
        if(tar[p->priority] == NULL || p->wait[p->priority] > tar[p->priority]->wait[p->priority])
            tar[p->priority] = p;
    }
    flag = 0;
}
```

上述代码执行完后，若队列 i 为空，那么 tar[i] == NULL，自上向下的枚举四个值，可以获取非空最高优先级的元素，然后赋值给 proc，即为这个时间片要处理的进程。

② 然后是 wait 值，done 值的更新与优先级的升降。

(a) wait 值，对于除 proc 以外的所有就绪态的进程 p:

```
p->wait[p->priority]++;
```

(b) done 值，只对 proc 进程进行更新

```
proc->done[proc->priority]++;
```

(c)优先级的上升:

上文提到过，优先级仅在饥饿时间足够长时发生上升，因此我们还是枚举整个页表，如果饥饿时间已经达到了目标，那么优先级+1，等待时间清空:

```
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++)
{
    if(p->wait[p->priority] == 10*ticks[p->priority])
    {
        if(p->priority != 3)
        {
```

```
                p->wait[p->priority] = 0;
                p->priority++;
            }
        }
    }
```

(d) 优先级的下降:优先级下降仅在 proc 进程上发生，且其正好完成当前队列的一个时间片。done[i]记录了进程在这个队列处理的总时间，因为可能完成不只一个时间片，因此我们通过取余来判定

```
if(proc->done[proc->priority]%ticks[proc->priority] == 0)
{
    flag = 1;
    proc->wait[p->priority] = 0;
    proc->priority--;
}
```

其中 int ticks[4] = {50,32,16,8};

至此，我们就完成了抢占式多级反馈队列的设计，总结流程，我们需要考虑的只有:①何时替换 proc ②更新 proc 的 done 值(+1)并以此判断(处理完一个时间片)是否需要对其降级 ③更新除 proc 以外进程的 wait 值(+1)并以此判断(饥饿)是否需要对其进行上升优先级。

Part 3 添加一个系统调用：

1. 声明部分

(1) 在 xv6/include/下添加文件 pstat.h

(2) 在 xv6/kernel/sysfile.c 中
11 行添加#include "pstat.h"
406 行添加 sys_getpinfo(void)来调用 getpinfo

(3) 在 xv6/user/usys.S 中
33 行添加 SYSCALL(getpinfo)

(4) 在 xv6/include/syscall.h 中
27 行添加#define SYS_getpinfo 23

(5) 在 xv6/kernel/syscall.c 中
9 行添加#include "pstat.h"
107 行添加[SYS_getpinfo] sys_getpinfo

(6) 在 xv6/kernel/sysfunc.h 中

27 行添加 int sys_getpinfo(void);

(7) 在 xv6/user/user.h 中
5 行添加 struct pstat;
30 行添加 int getpinfo(struct pstat *);

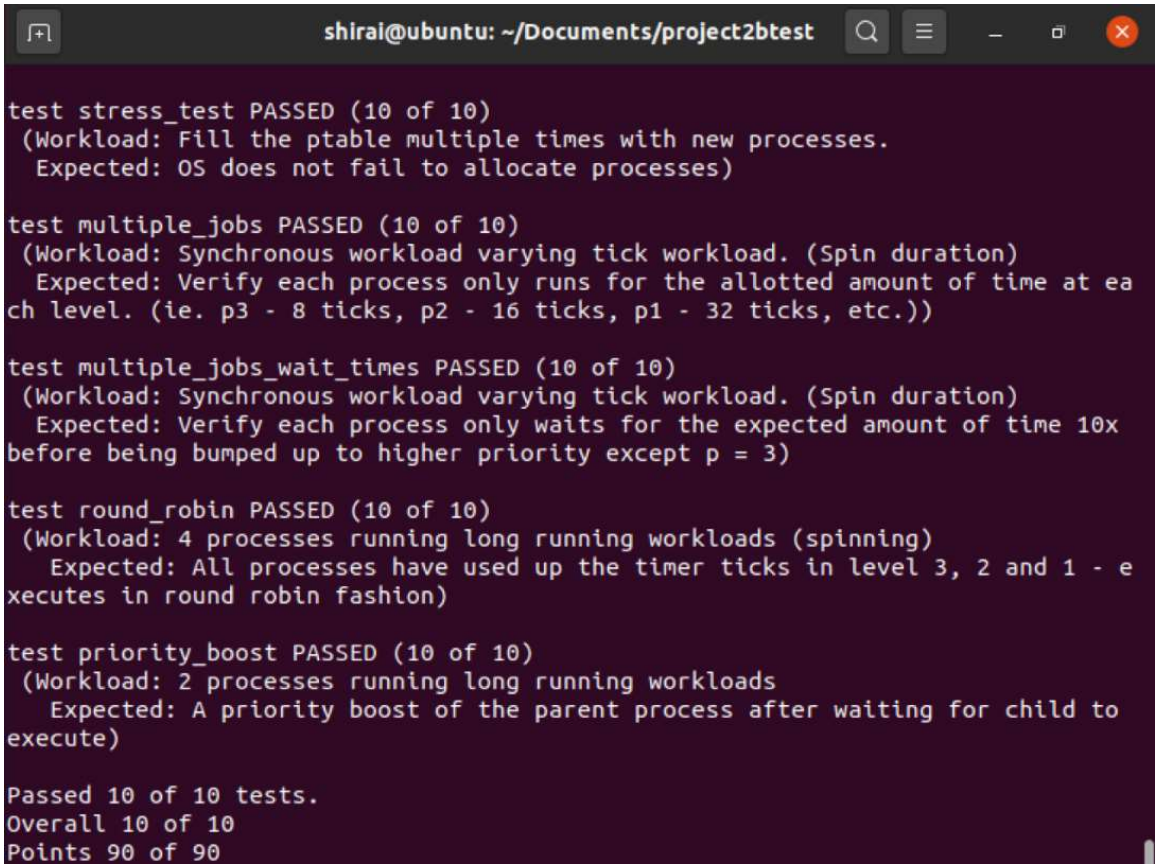
(8) 在 xv6/kernel/proc.c 中
末尾添加 getpinfo 的具体实现

2. getpinfo 的具体实现

函数原理很简单，就是把页表的每个进程信息赋予 pstat 且下标与页表中下标完全一致，源码如下

```
int getpinfo(struct pstat* pst)
{
    if(pst == NULL) return -1;
    struct proc*p;
    int i,j;
    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(j = 0;j < 4;j++)
        {
            pst->ticks[i][j] = p->done[j];
            pst->wait_ticks[i][j] = p->wait[j];
        }
    }
    return 0;
}
```

Part 4 测试结果：


shirai@ubuntu: ~/Documents/project2btest
test stress_test PASSED (10 of 10)
(Workload: Fill the ptable multiple times with new processes.
Expected: OS does not fail to allocate processes)
test multiple_jobs PASSED (10 of 10)
(Workload: Synchronous workload varying tick workload. (Spin duration)
Expected: Verify each process only runs for the allotted amount of time at ea
ch level. (ie. p3 - 8 ticks, p2 - 16 ticks, p1 - 32 ticks, etc.))
test multiple_jobs_wait_times PASSED (10 of 10)
(Workload: Synchronous workload varying tick workload. (Spin duration)
Expected: Verify each process only waits for the expected amount of time 10x
before being bumped up to higher priority except p = 3)
test round_robin PASSED (10 of 10)
(Workload: 4 processes running long running workloads (spinning)
Expected: All processes have used up the timer ticks in level 3, 2 and 1 - e
xecutes in round robin fashion)
test priority_boost PASSED (10 of 10)
(Workload: 2 processes running long running workloads
Expected: A priority boost of the parent process after waiting for child to
execute)
Passed 10 of 10 tests.
Overall 10 of 10
Points 90 of 90