A

```
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。

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;
}
```

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;
}
```

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) == p
tr->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) == p
tr->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;
}
```

B

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,
```

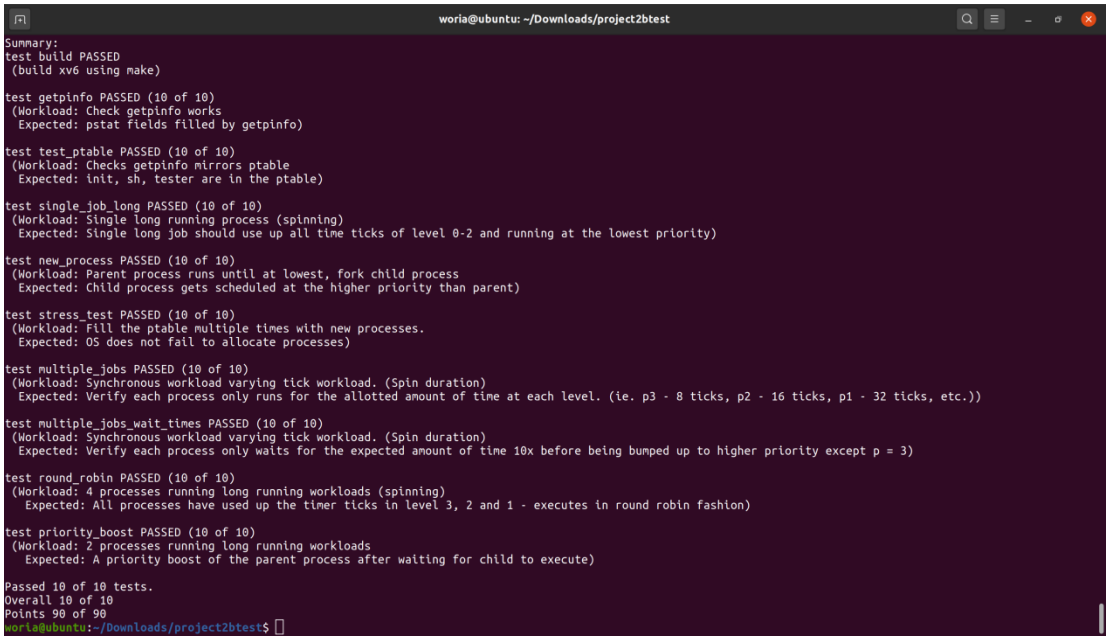
⑥添加了上述中段描述符之后，是要在 kernel/sysfunc.h 添加声明。

```
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*);
```

完成上面步骤，可以成功通过测试结果：


woria@ubuntu: ~/Downloads/project2btest
Summary:
test build PASSED
(build xv6 using make)
test getpinfo PASSED (10 of 10)
(Workload: Check getpinfo works
Expected: pstat fields filled by getpinfo)
test test_ptable PASSED (10 of 10)
(Workload: Checks getpinfo mirrors ptable
Expected: init, sh, tester are in the ptable)
test single_job_long PASSED (10 of 10)
(Workload: Single long running process (spinning)
Expected: Single long job should use up all time ticks of level 0-2 and running at the lowest priority)
test new_process PASSED (10 of 10)
(Workload: Parent process runs until at lowest, fork child process
Expected: Child process gets scheduled at the higher priority than parent)
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 each 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 - executes 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
woria@ubuntu:~/Downloads/project2btest$