Lab2

姓名: 陈锐林, 学号:21307130148

2023 年 10 月 13 日

实验一、procnum

一、实现思路:

在阅读了 proc.c 以及文档中的 hint 后，我感觉具体的实现并不复杂。进程存储在 proc[NPROC] 中，只要我们遍历该数组，状态不为 UNUSED 的就让计数器加 1 即可。

二、调用顺序：

上述的思路应该在 proc.c 中新增函数，因为在 sysproc.c 中是没法直接调用 proc 的。先在 proc.c 中增加 uint64 getnproc(void) 函数，所以在 def.h 中增加其声明，最后由 sysproc.c 中的函数 uint64 procnum(void) 调用；以及完成了 ppt 中的步骤后，之后用户程序 procnum.c 即可正常调用。

三、补充说明：

ppt 要求我们实现的 procnum 还要求对一个 int* 的 num 作出改变。这里需要额外处理这个地方。首先该地址的获取可以用已有 argint 完成；但是传出去好像在 syscall.c 中没找到类似的函数（也可能是没认真看），所以这里封装了另一个函数 fetchout(addr,num)，其调用 copyout（仿照 fetchaddr 的 copyin 调用）；完成了将一个 num 传给用户空间中地址为 addr 的内容改变。freemem 中也是类似处理，下面不再赘述。

四、实验代码：

```
uint64
sys_procnum(void){
  uint64 ptr;
  int x;
  argint(0,&x);
  uint64 nproc = getnproc();
  ptr = nproc;
  if (fetchout(x,&ptr) < 0)
    return -1;
  return nproc;
}
```

(a) sysproc.c 中

```
uint64 getnproc(void)
{
  struct proc *p;
  uint64 nproc = 0;
  for (p=proc; p< &proc[NPROC]; p++) {
    if (p->state != UNUSED)
      ++nproc;
  }
  return nproc;
}
```

(b) proc.c 中

五、运行结果：

```
hart 1 starting
hart 2 starting
init: starting sh
$ procnum
Number of process: 3
```

实验二、freemem

一、实现思路:

在阅读了 kalloc.c 以及文档中的 hint 后，我感觉具体的实现并不复杂。主要的空闲内存存在 freelist 中，我们先上锁，之后统计大小即可。而作为一个 list，可以很自然地用 next 遍历，而每次只要加上 xv6 自带参数 PGSIZE 即可。

二、调用顺序：

上述的思路应该在 kalloc.c 中新增函数，因为在 sysproc.c 中是没法直接调用 kmem 的。先在 kalloc.c 中增加 uint64 getfreemem(void) 函数，所以在 def.h 中增加其声明，最后由 sysproc.c 中的函数 uint64 freemem(void) 调用；以及完成了 ppt 中的步骤后，之后用户程序 freemem.c 即可正常调用。

三、实验代码：

```
uint64
sys_freemem(void){
  uint64 ptr;
  int x;
  argint(0,&x);
  uint64 freemem = getfreemem();
  ptr = freemem;
  if (fetchout(x,&ptr) < 0)
    return -1;
  return freemem;
}
```

(a) sysproc.c 中

```
uint64 getfreemem(void)
{
  uint64 freemem = 0;
  int nowcpu = cpuid();
  acquire(&kmem[nowcpu].lock);
  struct run *r = kmem[nowcpu].freelist;
  while (r) {
    freemem += PGSIZE;
    r = r->next;
  }
  release(&kmem[nowcpu].lock);
  return freemem;
}
```

(b) kalloc.c 中

五、运行结果：

```
hart 2 starting
hart 1 starting
init: starting sh
$ freemem
Number of bytes of free memory: 66744320
```

实验三、trace

一、实现思路:

首先，为了让系统调用被正确跟踪，需要从其掩码入手，给 struct proc 增加一个新的变量，int mask。之后在 syscall 时识别并及时打印就可以。

二、具体实现：

通过观察 syscall.c 中的 syscall 函数后，我们发现，syscall 会获取一个 num，而这个 num 就可以用来作掩码判断。以及还需要完善的点：sys_trace 负责获取参数，并向当前进程传递；需要考虑 fork 时 mask 也被传递，所以要在 proc.c 的 fork 函数中增添一行 np->mask = p->mask；在 syscall 函数中，如果直接打印 syscalls[num]，会出现乱码，为了解决该现象，新增一 char* 数组 syscall_name，如此打印就无误。

三、实验代码：

```
int mask = p->mask;
// trace syscall masked
if ((1 << num) & mask) {
  printf("%d: syscall %s -> %d\n", p->pid, syscall_name[num], p->trapframe->a0);
}
```

四、运行结果：

```
$ trace 32 grep hello README
4: syscall read -> 1023
4: syscall read -> 961
4: syscall read -> 321
4: syscall read -> 0
$
$ trace 2147483647 grep hello README
6: syscall trace -> 0
6: syscall exec -> 3
6: syscall open -> 3
6: syscall read -> 1023
6: syscall read -> 961
6: syscall read -> 321
6: syscall read -> 0
6: syscall close -> 0
```

实验四、流程概述

1. 系统调用过程：

用户使用系统调用的接口，通过 syscall 函数触发系统调用；处理器从用户模式变为内核模式；在内核模式内系统选择要执行的调用；执行系统调用，可能会会从用户空间拿取数据；返回系统调用结果，返回用户模式。

2.malloc 的底层实现原理：

查阅到的资料经概述后如下：(1) 内存池初始化：在启动时，xv6 会初始化一个内存池，用于存储所有可用的内存块。(2) 内存块分配：malloc 函数会调用内存分配器。内存分配器的任务是从内存池中找到足够大的空闲内存块，然后将其标

记为已分配。分配器返回指向此块的指针。(3) 内存块释放：完成使用分配的内存后，可以使用 free 函数来将内存块返回给内存分配器。内存分配器会将被释放的内存块标记为可用。(4) 碎片管理：内存分配器可能需要处理内存碎片的问题。碎片是未分配的小块内存，通常由已分配内存块之间的空间导致。一些内存分配器使用合并或分割策略来最小化碎片。在 xv6 中，内存分配器采用首次适配策略，即它会尽量使用第一个足够大的空闲块，而不会尝试合并或分割块。(5) 内存池管理：内存分配器还需要管理内存池，以跟踪哪些块是可用的，哪些是已分配的。