OS-lab2-Report

21302010042
侯斌洋

实验一：procnum

（1）实验思路

先观察 proc.c 文件的结构，了解 xv6 中与 process 相关的信息。在 proc.c 文件中，发现有个函数 procdump()可以打印出当前进程的状态，可以用来 debug。然后参考 procdump()的函数结构写出统计进程数的代码，即//count process 所在的代码块。此时要解决的问题是如何把统计到的值从 kernel 传到 user。因为 kernel 与 user 的内存地址不同，所以无法通过直接获取指针并修改的方法传递数据。观察 syscall.c 文件，可以发现有与系统调用传参相关的代码，这里通过 argaddr()函数获取用户内存空间的地址，再通过 copyout()函数将值 src 写入该用户内存空间的地址来实现传递数据，即//get int pointer 和 //copy data from kernel to user 所在的代码块。

（2）实验过程

①实验代码：

[kernel/sysproc.c]

```
extern struct proc proc[NPROC];
uint64
sys_procnum()
{
  //for debug
  //procdump();

  // get int pointer
  uint64 proc_addr;
  argaddr(0, &proc_addr);

  // count processs
  int count = 0;
  uint64 src = 0;
  struct proc *p;
  for (p = proc; p < &proc[NPROC]; p++)
  {
    if (p->state == UNUSED)
      count++;
  }
  src = NPROC - count;

  // copy data from kernel to user
  struct proc *cur_proc = myproc();
  copyout(cur_proc->pagetable, proc_addr, (char *)(&src), sizeof(proc_addr));
  return src;
}
```

②系统调用流程：

当在 shell 中输入 procnum 时，shell 在其子进程中运行 user/procnum.c 中的代码，并在 procnum.c 中启动系统调用 procnum(&num)，(注：在 user.h 和 usys.pl 中标记了该系统调用及其入口)。 之后执行 usys.S（注：由 usys.pl 生成）中对应汇编代码，即：

[user/usys.S]

```
.global procnum
procnum:
 li a7, SYS_procnum
 ecall
 ret
```

这里 li 指令将 SYS_pronum (注: 记录在 kernel/syscall.h 中) 的系统调用号保存在 a7 寄存器中。 然后调用 ecall 陷入内核，运行 uservec 保存用户寄存器状态到 trapframe 中（详细操作在 kernel/trampoline.S 中），运行 usertrap 判断 trap 类型并处理（详细操作在 kernel/trap.c 中）。如果是系统调用的话，就将 pc 指向 ecall 的下一条指令，然后交给 syscall 函数处理，即：

[kernel/trap.c]

```
if(r_scause() == 8){
  // system call

  if(killed(p))
    exit(-1);

  // sepc points to the ecall instruction,
  // but we want to return to the next instruction.
  p->trapframe->epc += 4;

  // an interrupt will change sepc, scause, and sstatus,
  // so enable only now that we're done with those registers.
  intr_on();

  syscall();
```

syscall 获取 a7 寄存器中的系统调用号并执行相对应的系统调用，然后就到了 sysproc.c 中由我们编写的 sys_procnum 函数，完成该调用后将返回值保存在 trapframe->a0 中。最后调用 usertrapret()返回到用户空间，trap 结束，至此系统调用的过程结束。

在上面的过程中系统调用的参数是通过 uservec 保存的 trapframe 来传递的，例如使用 argraw()获取参数：

[kernel/syscall.c]

```
static uint64
argraw(int n)
{
  struct proc *p = myproc();
  switch (n)
  {
  case 0:
    return p->trapframe->a0;
  case 1:
    return p->trapframe->a1;
  case 2:
    return p->trapframe->a2;
  case 3:
    return p->trapframe->a3;
  case 4:
    return p->trapframe->a4;
  case 5:
    return p->trapframe->a5;
  }
  panic("argraw");
  return -1;
}
```

如果要在内核空间中读写用户空间某个地址的数据，则要使用专用的 copyin 和 copyout 函数（在 kernel/vm.c 中）通过当前进程对应的页表来进行操作。

（3）实验结果

【注：这里为了体现程序的正确性使用了 procdump()函数来打印当前正在运行的进程。】

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ procnum

1 sleep  init
2 sleep  sh
3 run    procnum
Number of process: 3
$ > usertest.out usertests &
$ procnum

1 sleep  init
2 sleep  sh
6 run    usertests
5 run    usertests
7 run    procnum
Number of process: 5
$ QEMU: Terminated
hby@hby-ubuntu:~/Desktop/xv6-labs-2022$
```

实验二：freemem

（1）实验思路

先观察 kalloc.c 文件的结构，发现内存是用 kmem 结构管理的，其中 freelist 记录了空闲的内存块，一个节点代表一页，每页的大小为 4096 Bytes。这样只需获取链表的长度即可计算出空闲字节数。即//my function 所在的代码块。

要把统计到的值从 kernel 传到 user，与实验一中的解决方法相同，使用 argaddr()和 copyout()函数，不再赘述。即//get int pointer 和 //copy data from kernel to user 所在的代码块。

（2）实验过程

①实验代码：

[kernel/sysproc.c]

```
extern int get_free_bytes_num();
uint64
sys_freemem()
{
  // get int pointer
  uint64 proc_addr;
  argaddr(0, &proc_addr);

  // get free bytes
  // the function below is in kalloc.c
  int num = get_free_bytes_num();
  uint64 src = num;

  // copy data from kernel to user
  struct proc *cur_proc = myproc();
  copyout(cur_proc->pagetable, proc_addr, (char *)(&src), sizeof(proc_addr));
  return num;
}
```

[kernel/kalloc.c]

```
//my funtion
int get_free_bytes_num(){
  acquire(&kmem.lock);
  int count = 0;
  struct run *temp = kmem.freelist;
  while (temp)
  {
    count++;
    temp=temp->next;
  }
  release(&kmem.lock);
  return count*4096;
}
```

②系统调用流程：(细节在实验一中已说明，下面将尽量简洁地阐述)

当在 shell 中输入 freemem 时，shell 在其子进程中运行 user/freemem.c 中的代码，并在 freemem.c 中启动系统调用 freemem(&num)。 之后执行汇编代码：

```
.global freemem
freemem:
 li a7, SYS_freemem
 ecall
 ret
```

li 指令将系统调用号保存在 a7 寄存器中。 然后调用 ecall 陷入内核，运行 uservec 保存用户寄存器状态到 trapframe 中，运行 usertrap 判断 trap 类型并处理。

如果是系统调用的话，就将 pc 指向 ecall 的下一条指令，然后交给 syscall 函数处理。syscall 获取 a7 寄存器中的系统调用号并执行相对应的系统调用，即 sys_freemem 函数。

当 syscall 执行完成后，调用 usertrapret()返回到用户空间，trap 结束，至此系统调用的过程结束。

在上面的过程中系统调用的参数是通过 uservec 保存的 trapframe 来传递的。且如果要在内核空间中读写用户空间某个地址的数据，则要使用专用的 copyin 和 copyout 函数通过当前进程对应的页表来进行操作。

（3）实验结果

由于程序逻辑较为简单，可直接从逻辑上看出对错，故此处只作简单的验证。

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ freemem
Number of bytes of free memory: 133378048
$ > usertest.out usertests &
$ freemem
Number of bytes of free memory: 67063808
$ QEMU: Terminated
hby@hby-ubuntu:~/Desktop/xv6-labs-2022$
```

可以看出在后台执行 usertests 后，空闲内存减少。

实验三：trace

（1）实验思路

由于 exec 不改变进程 id，故可通过在当前 proc 结构中保存 trace_mask 来使得之后 exec 的进程的 proc 中也有 trace_mask 的值，即//add trace_mask 对应的代码块。其中 trace_mask 的值需要通过//get trace_mask 的代码块获得。这样就完成了 proc 中 trace_mask 的标记。

之后在 syscall 中通过该标志来确定是否需要打印出 trace 信息。由于 trace_mask 是通过对某个或者某些位进行标志来确定跟踪的系统调用的，故可以直接用 & 运算符来确认 trace_mask 与（1 << num）的某些位是否有交集，如有的话则值不为 0，打印 trace 信息。即//After syscall,trace it if num can match mask 所在的代码块。

其中进程 ID 可直接通过 p 获得，系统调用名称通过定义的 syscall_str 数组查询系统调用号获得。返回值通过 trapframe->a0 获得。

若要同时跟踪其子进程的系统调用，只需在 fork 函数中将 trace_mask 同时复制给子进程的 proc 结构。即//copy trace_mask to children 所在的代码块。

（2）实验过程

①实验代码：

[kernel/sysproc.c]

```
uint64
sys_trace()
{
  // get trace_mask
  int trace_mask;
  argint(0, &trace_mask);

  // add trace_mask to the process
  struct proc *cur_proc = myproc();
  acquire(&(cur_proc->lock));
  cur_proc->trace_mask = trace_mask;
  release(&(cur_proc->lock));
  return 0;
}
```

[kernel/syscall.c]

```c
void
syscall(void)
{
  int num;
  struct proc *p = myproc();

  // syscall num
  num = p->trapframe->a7;

  if (num > 0 && num < NELEM(syscalls) && syscalls[num])
  {
    // Use num to lookup the system call function for num, call it,
    // and store its return value in p->trapframe->a0
    p->trapframe->a0 = syscalls[num]();

    // After syscall, trace it if num can match mask
    if (p->trace_mask & (1 << num))
    {
      printf("%d: syscall %s -> %d\n", p->pid, syscall_str[num], p->trapframe->a0);
    }
  }
  else
  {
    printf("%d %s: unknown sys call %d\n",
           p->pid, p->name, num);
    p->trapframe->a0 = -1;
  }
}
```

```c
#define MAXSYSCALLLEN 20
char syscall_str[][MAXSYSCALLLEN] = {
    "error",
    "fork",
    "exit",
    "wait",
    "pipe",
    "read",
    "kill",
    "exec",
    "fstat",
    "chdir",
    "dup",
    "getpid",
    "sbrk",
    "sleep",
    "uptime",
    "open",
    "write",
    "mknod",
    "unlink",
    "link",
    "mkdir",
    "close",
    "procnum",
    "freemem",
    "trace",
};
```

[kernel/proc.c]

```
int
fork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;

  //Copy trace_mask to children
  np->trace_mask = p->trace_mask;
```

②系统调用流程：

当在 shell 中输入 trace 命令时，shell 在其子进程中运行 user/trace.c 中的代码，并在 trace.c 中启动系统调用 trace(trace_mask)。 之后执行汇编代码：

```
.global trace
trace:
 li a7, SYS_trace
 ecall
 ret
```

li 指令将系统调用号保存在 a7 寄存器中。 然后调用 ecall 陷入内核，运行 uservec 保存用户寄存器状态到 trapframe 中，运行 usertrap 判断为系统调用，将 pc 指向 ecall 的下一条指令，然后交给 syscall 函数处理。syscall 获取 a7 寄存器中的系统调用号并执行相对应的系统调用，即 sys_trace 函数，该函数设置当前进程的 trace_mask。

当 syscall 执行完成后，调用 usertrapret()返回到用户空间，trap 结束，至此 trace 系统调用的过程结束。

之后返回用户空间继续执行 exec(nargv[0], nargv); 语句替换当前进程，但 trace_mask 的状态依然保留，这样在当前进程中每次执行系统调用时若 trace_mask 与系统调用号匹配就会打印 trace 信息。子进程由于复制了 trace_mask，故 trace 也会同时跟踪子进程。

（3）实验结果

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$  trace 32 grep hello README
3: syscall read -> 1023
3: syscall read -> 961
3: syscall read -> 321
3: syscall read -> 0
$  trace 2147483647 grep hello README
4: syscall trace -> 0
4: syscall exec -> 3
4: syscall open -> 3
4: syscall read -> 1023
4: syscall read -> 961
4: syscall read -> 321
4: syscall read -> 0
4: syscall close -> 0
$ trace 2 usertests forkforkfork
usertests starting
5: syscall fork -> 6
test forkforkfork: 5: syscall fork -> 7
7: syscall fork -> 8
8: syscall fork -> 9
8: syscall fork -> 10
9: syscall fork -> 11
8: syscall fork -> 12
8: syscall fork -> 13
9: syscall fork -> 14
10: syscall fork -> 15
9: syscall fork -> 16
10: syscall fork -> 17
```

```
10: syscall fork -> 50
11: syscall fork -> 51
10: syscall fork -> 52
11: syscall fork -> 53
24: syscall fork -> 55
17: syscall fork -> 54
11: syscall fork -> 56
24: syscall fork -> 57
39: syscall fork -> 58
11: syscall fork -> 59
39: syscall fork -> 60
40: syscall fork -> 63
11: syscall fork -> 62
24: syscall fork -> 61
40: syscall fork -> 64
25: syscall fork -> 65
42: syscall fork -> 66
42: syscall fork -> 67
38: syscall fork -> -1
OK
5: syscall fork -> 68
ALL TESTS PASSED
$ QEMU: Terminated
hby@hby-ubuntu:~/Desktop/xv6-labs-2022$
```

实验四：流程概述

（1）根据以上 xv6 的系统调用经验，总结过程如下：
在用户空间内保存了系统调用的声明以及入口，当启动系统调用时，首先保存系统调用号到一个特定的寄存器中（xv6 中为 a7），之后陷入内核（xv6 中通过执行 ecall 实现，在 ecall 时中保存了当前 pc 以从 trap 中返回），进入内核后首先保存用户寄存器的状态到一个特定的结构中（xv6 中为 trapframe），注意到在这里保存了系统调用的参数。然后识别 trap 类型并进行处理，如为系统调用，则交由系统调用处理程序执行，系统调用处理程序通过之前保存的系统调用号（xv6 中为 trapframe->a7）调用对应的系统调用，并将返回值保存在某处（xv6 中为 trapframe->a0）。最后设置 trapframe，恢复页表，寄存器，并设置 pc 以从 trap 返回到用户空间（xv6 中通过 usertrapret()实现）。

【注：以下资料对理解 trap 有很大帮助。】
https://www.cnblogs.com/weijunji/p/14338450.html
https://www.cnblogs.com/KatyuMarisaBlog/p/13934537.html
https://zhuanlan.zhihu.com/p/462538325

（2）
一些机制：
1：每个进程有独立的虚拟地址空间，进程访问的是虚拟地址。
2：内存和磁盘中的空间以页为单位进行组织。
3：虚拟地址可通过每个进程上的页表与物理地址进行映射，获得真正物理地址。
4：当访问的虚拟地址对应的物理地址不在物理内存中，则产生缺页中断，分配物理内存。
5：malloc 分配的只是虚拟内存，并未分配物理内存。

在 malloc 执行的过程中，当 malloc 的空间较小时（如小于 128K），malloc 函数调用 brk 系统调用，将_edata 指针（即堆指针）往高地址推对应的空间。之后进程访问该内存时，若发生缺页中断则分配物理内存，否则不会额外分配物理内存。在 free 时，brk 分配的内存需要等到高地址内存释放以后才能释放，故如果高地址的内存未被 free，该 free 内存块会形成一个内存碎片，_edata 指针也不会下移。而 free 时若发现 free 后最高地址空间的空闲内存超过一定空间（如 128K），则执行内存紧缩操作，_edata 指针下移。

当 malloc 的空间较大时（如大于等于 128K），malloc 函数调用 mmap（）系统调用来在堆和栈中间找一块空闲的虚拟内存。之后进程访问该内存时，若发生缺页中断则分配物理内存，否则不会额外分配物理内存。在 free 时，mmap（）系统分配的内存可以直接释放。

参考：https://www.cnblogs.com/ssezhangpeng/p/10808969.html
https://www.cnblogs.com/dongzhiquan/p/5621906.html