Project 1 - Operating Systems - New getcnt() syscall

DCC605 - Sistemas Operacionais - Universidade Federal de Minas Gerais

Group: Juan Marcos Braga Faria, Lucas Almeida Santos de Souza

Tasks

1. Modify xv6 so the kernel keeps track of how many times each syscall has been called.
   - The getcnt syscall receives one integer parameter.
   - It returns the number of times the syscall with the given number has been called.
   - The user space declaration of getcnt should be `int getcnt(int)`
2. Implement a user-space program to call the syscall. The program should be named "getcnt" and receive a single integer as parameter corresponding to the target syscall.

Modified files:

- kernel/syscall.c
- kernel/syscall.h
- kernel/sysfile.c
- kernel/proc.c
- kernel/proc.h
- user/user.h
- user/usys.pl
- user/getcnt.c
- Makefile

Implementation Choice

We may use different existing kernel functions as reference for `getcnt`. However that would depend on what processes the new functionality is meant to keep track of: the amount of times any system call has been called by a certain process, or by the whole system. On the first case, the `getpid` would be a good candidate as it looks up a variable local to the process. For the second case, maybe `uptime` would be better, as it has to acquire an external value. The function that has the job of finding and invoking the call, `syscall`, could be the primary candidate to do the incrementation task, albeit with a cost.

As the integer variable have limits, and the students have no data on how many times system calls are made during an expected system uptime - not to mention the performance overhead or the much longer uptime of servers-, the implementation chosen for `getcnt` will track how many times the system calls were called by each process.

It makes sense also as registering global values usually have limits imposed, such as the default number of registers for executed commands collected by `history` is `1000`.

xv6 modifications

Data structure in `proc.h`

kernel\proc.h

```diff
@@ -84,24 +84,26 @@
 // Per-process state
 struct proc {
   struct spinlock lock;

   // p->lock must be held when using these:
   enum procstate state;        // Process state
   void *chan;                  // If non-zero, sleeping on chan
   int killed;                  // If non-zero, have been killed
   int xstate;                  // Exit status to be returned to parent's wait
   int pid;                     // Process ID

   // wait_lock must be held when using this:
   struct proc *parent;         // Parent process

   // these are private to the process, so p->lock need not be held.
   uint64 kstack;               // Virtual address of kernel stack
   uint64 sz;                   // Size of process memory (bytes)
   pagetable_t pagetable;       // User page table
   struct trapframe *trapframe; // data page for trampoline.S
   struct context context;      // swtch() here to run process
   struct file *ofile[NOFILE];  // Open files
   struct inode *cwd;           // Current directory
   char name[16];               // Process name (debugging)
+
+  uint64 syscall_count[22];    // Array que contabiliza o número de vezes que cada syscall foi chamada
 };
```

The chosen data structure was a simple integer array, `uint syscall_count`, to prioritise performance and access. It increases the length of a data structure that is created in every single process: `struct proc`.

Why here? As the chosen implementation keeps track of calls from each process, the data structure that contains each process's "metadata". It is initialized at the beginning of a process, and dies with it.

Keeping track of calls in `syscall.c`

kernel\syscall.c

```diff
   [SYS_link]    sys_link,
   [SYS_mkdir]   sys_mkdir,
   [SYS_close]   sys_close,
+  [SYS_getcnt]  sys_getcnt,
   };

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

     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
+
+          // Incrementa o número de vezes que a syscall foi chamada
+          p->syscall_count[num-1]++;
+
       p->trapframe->a0 = syscalls[num]();
     } else {
       printf("%d %s: unknown sys call %d\n",
               p->pid, p->name, num);
       p->trapframe->a0 = -1;
     }
   }

```

The data structure needs to be updated every time a system call is invoked. As `syscll_count` items matches the number of system calls, a simple jump using the call number as index should suffice: `p->syscall_count[num-1]++`.

Why here? The function responsible for validating and calling the call seems to be the best place to insert code to do these updates.

Initializing data structure in `proc.c`

kernel\proc.c

```
    // Look in the process table for an UNUSED proc.
    // If found, initialize state required to run in the kernel,
    // and return with p->lock held.
    // If there are no free procs, or a memory allocation fails, return 0.
    static struct proc*
    allocproc(void)
    {
      struct proc *p;

      for(p = proc; p < &proc[NPROC]; p++) {
        acquire(&p->lock);
        if(p->state == UNUSED) {
          goto found;
        } else {
          release(&p->lock);
        }
      }
+
+     // Inicializa syscall_count em zero para todas as syscalls.
+     for (int i = 0; i < NELEM(p->syscall_count); i++)
+       p->syscall_count[i] = 0;
+
      return 0;

    found:
      p->pid = allocpid();
      p->state = USED;
```

According to the xv6 manual, `allocproc` is the function responsible for allocating a process's resources when it is initialized. It is the clear candidate for a code that is responsible for initializing data for `syscall_count`.

Actual system function code in `sysfile.c`

kernel\sysfile.c

```diff
@@ -503,3 +503,18 @@ sys_pipe(void)
   }
   return 0;
 }
+
+uint64
+sys_getcnt(void)
+{
+  int syscallID;
+  argint(0, &syscallID);
+
+  struct proc *p = myproc();
+
+  if(syscallID < 1 || syscallID > NELEM(p->syscall_count))
+        return -1;
+
+  int cnt = p->syscall_count[syscallID-1];
+  return cnt;
+}
```

`argint()` places the provided call number to the local syscall variable. It then acquires a pointer to the local `struct proc` through the `myproc()` interface, checking everything later. Grabbing the correct value is then a simple matter of reaching for the count value using the call number as index.

Why here? Naturally this file is where other similar system calls are located.

Testing

user\getcnt.c

```
@@ -0,0 +1,22 @@
+ #include "kernel/types.h"
+ #include "kernel/stat.h"
+ #include "user/user.h"
+
+ int main(int argc, char **argv) {
+   if(argc != 2){
+         fprintf(2, "Usage: getcnt <syscall id>\n");
+         exit(1);
+   }
+
+   int syscall_num = atoi(argv[1]);
+
+   int cnt;
+
+   if((cnt = getcnt(syscall_num)) < 0){
+         fprintf(2, "getcnt: failed to get count\n");
+         exit(1);
+   }
+
+   fprintf(1, "syscall %d has been called %d times\n", syscall_num, cnt);
+   exit(0);
+ }
```

The students used a series of simple tests. The user interface from Task 2, implemented in `user/getcnt.c` was used to check for the expected behaviors.

The students tried to verify the call doing the following:

- Call getcnt 22 (the code of the getcnt system call itself) several consecutive times to check if the counter incremented with each call.
- Call getcnt 2 on each execution to observe the behavior of the exit syscall (this syscall is called at the end of each command).
- Create a directory with mkdir and then call getcnt(20) (20 is the system call code for mkdir).

Sources

https://github.com/palladian1/xv6-annotated?tab=readme-ov-file

https://github.com/remzi-arpacidusseau/ostep-projects/tree/master/initial-xv6

https://moss.cs.iit.edu/cs450/mp1-xv6.html

https://pdos.csail.mit.edu/6.828/2023/xv6/book-riscv-rev3.pdf