xv6: Adding a system call

Background

RISC-V CPU modes

User mode

- Least privilege
- User processes run in this mode
- Uses paging with pagetable corresponding to running user process

Supervisor mode (or Kernel mode)

- Medium privilege
- Kernel (including kernel modules and device drivers), hypervisor run in this mode
- Uses paging with pagetable reserved for kernel

Machine mode

- Highest privilege
- Bootloader, firmware run in this mode
- Does not use paging

Life cycle of a system call: sleep

Suppose the following code is executed by a user program test sleep.c:

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main()
{
   int status = sleep(100);
   if (status == -1)
       printf("Error\n");
   else
       printf("ok\n");
   return 0;
}
```

sleep is a system call. It will cause the following steps.

- 1. User function calls sleep (100)
 - It means sleep for 100 timer interrupts. Timer interrupts have an interval of 0.1 second (interval is defined in kernel/start.c/timerinit). So, this sleep takes about 10 seconds.
 - sleep system call takes an int as argument and returns an int, which is found from its declaration in user/user.h, int sleep(int)
- 2. The system call number of sleep is loaded into the register **a7** and a trap is generated.
 - The definition of the function int sleep (int) is provided in user/usys.S
 - sleep system call has system call number 13, which is defined as SYS_sleep in kernel/syscall.h
- 3. The trap generated in step 2, puts the processor into supervisor mode and calls the function kernel/trampoline.S/uservec. This function does some state management (saving state of registers and swapping pagetable) and finally jumps to kernel/trap.c/usertrap.
 - This function does not *call* usertrap. So, we will never get back to this function again while handling our sleep (100).
- 4. The function kernel/trap.c/usertrap handles the trap based on its cause. As, sleep is a system call, it calls kernel/syscall.c/syscall.
 - The cause of a trap is given by $r_scause()$ (supervisor trap cause). If it is 8, then the cause is system call. We will see other values in later assignments.

- 5. The function kernel/syscall.c/syscall is the single point of contact for handling system calls. It retrieves the system call number (13 in this case) from the saved (in step 3) value of register a7. Then it calls the handler function corresponding to the system call number. All system call handlers are saved in an array of function pointers, kernel/syscall.c/syscalls. In this array, the system call handler for sleep is sys_sleep.
- 6. The function <code>sys_sleep</code> is defined in <code>kernel/sysproc.c</code>. In this function, first the argument to **sleep** system call is retrieved. Then it does some work that actually makes the calling process sleep for the given amount of time. It returns -1, if there is any error, otherwise it returns 0.
- 7. Now, we are again in the kernel/syscall.c/syscall. The returned value of the system call handler (in this case, sys sleep) is saved in the register **a0**.
- 8. Then it returns to kernel/trap.c/usertrap and calls kernel/trap.c/usertrapret.
- 9. In kernel/trap.c/usertrapret, it does some state management and calls kernel/trampoline.S/userret.
- 10. kernel/trampoline.S/userret does some state management (restoring state of registers and swapping pagetable) and returns to user mode.

System call related codes explanation:

user/user.h

Contains declaration of system call related functions

kernel/syscall.h

Contains definition of system call number for each system call

user/usys.S

- Generated by user/usys.pl
- Contains the definition of system call related functions
- Generates a trap with corresponding system call number that puts the CPU into supervisor mode

kernel/trampoline.S

kernel/trampoline.S/uservec

- Saves the state of the registers
- Switches the pagetable of the user process with the pagetable for kernel
- Calls kernel/trap.c/usertrap

kernel/trampoline.S/userret

- Restores the state of the registers
- Switches the pagetable for kernel with the pagetable of the user process
- Switches mode from supervisor to user

kernel/trap.c

kernel/trap.c/usertrap

• Handles an interrupt, exception, or system call from user space

- Calls kernel/syscall.c/syscall if the trap is generated by system call (supervisor trap cause = 8)
- Calls kernel/trap.c/usertrapret after the trap is handled

kernel/trap.c/usertrapret

- Returns to user space
- Calls kernel/trampoline.S/uservec

kernel/syscall.c

• Contains declarations of the system call handlers

kernel/syscall.c/syscalls

- An array of function pointers that points to the actual code for handling each system call
- All the system call handlers must have the same type. For xv6, the type is uint64 (void). So, if a system call has arguments, it retrieves them from saved registers using kernel/syscall.c/argraw. The returned value must be a uint64, which means a system call can either return an integer or a pointer.

kernel/syscall.c/syscall

- The main point of contact for handling system calls
- Calls the corresponding handler for the system call that generated the trap. The system call that generated the trap is found in the system call number which was provided in user/usys.S.
- Stores the returned value from the system call handler

kernel/syscall.c/argraw

- Retrieves the arguments to the system call as uint 64
- Can retrieve at most 6 arguments, i.e., xv6 can only handle system call with at most 6 arguments

kernel/syscall.c/argint

 Retrieves the arguments to the system call and saves it at another variable of int type.

kernel/syscall.c/argaddr

• Retrieves the arguments to the system call and saves it at another variable which will be treated as an address.

kernel/sysproc.c

- Contains definition of the system call handlers related to process management kernel/sysfile.c
 - Contains definition of the system call handlers related to file management

Add a new system call

We will add a new system call to get an integer user id. We will call this system call **getuid**.

1. First we need to declare its signature in user/user.h.

```
// system calls
int fork(void);
...
int uptime(void);
int getuid(void); // add this line
```

2. Then we need to define the system call number for **getuid** in kernel/syscall.h.

```
// System call numbers
#define SYS_fork   1
...
#define SYS_close   21
#define SYS_getuid   22 // add this line
```

3. We need to have the definition of int getuid (void) in user/usys.S. This code is auto generated by user/usys.pl. So, we add an entry for getuid in user/usys.pl.

```
#!/usr/bin/perl -w
...
entry("uptime");
entry("getuid"); # add this line
```

4. We need to add a system call handler for **getuid**. For that we will add a function named sys_getuid. We need to add a record in the array kernel/syscall.c/syscalls. So, we make the following changes in kernel/syscall.c.

```
// Prototypes for the functions that handle system calls.
extern uint64 sys_fork(void);
...
extern uint64 sys_close(void);
extern uint64 sys_getuid(void); // add this line

// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
[SYS_fork] sys_fork,
...
[SYS_close] sys_close,
[SYS_getuid] sys_getuid, // add this line
};
```

5. We need to add the definition of the function sys_getuid. We can define it in both kernel/sysproc.c and kernel/sysproc.c this time. Add the following codes at the end of kernel/sysproc.c.

```
// return user id
uint64
sys_getuid(void)
{
   return getuid();
}
```

6. The function <code>sys_getuid</code> calls another function <code>getuid</code> that will be defined in <code>kernel/proc.c</code> (please note that this <code>getuid</code> function is different from that declared in <code>user/user.h</code>). Add the following codes at the end of <code>kernel/proc.c</code>. [This step would not be required if <code>sys_getuid</code> did not call a newly defined function.]

```
// define a global variable for user id
int uid = 123;

// getter for user id
int
getuid(void)
{
  return uid;
}
```

7. Now, to make <code>getuid</code> accessible from <code>kernel/sysproc.c</code>, add a line in <code>kernel/defs.h</code>. [This step would not be required if <code>sys_getuid</code> did not call a newly defined function.]

8. Now, create the following user program printuid.c following the instructions in **S** xv6: Adding a user program and run.

```
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int main()
{
   int uid = getuid();
   printf("%d\n", uid);
   return 0;
}
```

9. The output should look like this:

```
$ printuid
123
$
```

Practice

Add a system call to set the user id. Also, create a user program to test it.