Sprint 1 Report
Team StuckOverflow

Overview.

We implemented three system calls:

1. System Call 22 - getsyscallcount(): returns how many times a specific system call has been invoked since boot.
2. System Call 23 - getproclist(): display a list of the running processes, each process with its name, pid, memory size, status, and ppid.
3. System Call 24 - getppid(): returns the parent's pid.

System Call 22: getsyscallcount() system call

Target

- Create a system call that returns how many times a specific system call has been invoked since boot.

Pre Design Decisions

- We need to track the number of times each syscall is called.
- The syscall accepts an integer representing the syscall number (e.g., `SYS_getpid` = 11) and returns the count of how many times that syscall was called.
- Return `-1` if the syscall number is invalid.

Approach in Design

- Maintain a global array in the kernel to count invocations of each syscall.
- Instrument the syscall dispatch code to increment the count for each syscall invocation.
- Implement a new syscall `getsyscallcount` that receives a syscall number and returns the corresponding count.
- The kernel only provides raw counts; user programs decide how to display or use the data.

Steps

1. **Add a global array to count syscalls**
    - Declare an integer array `syscall_count[NUM_SYSCALLS]` in the kernel (e.g., `syscall.c` or a new file).
    - In the syscall dispatch function (e.g., `syscall.c`), increment the count for each syscall whenever it is invoked.
2. **Define the syscall number**

- Add a new syscall number in `syscall.h`:
  `#define SYS_getsyscallcount 22`

3. **Declare the kernel syscall function:**
   - In `syscall.c`:
     `extern int sys_getsyscallcount(void);`
4. **Add the syscall function pointer to the `syscalls` array:**
   - `[SYS_getsyscallcount] sys_getsyscallcount,`
5. **Implement the syscall function**
6. In `sysproc.c` or suitable file, implement:

```
int sys_getsyscallcount(void) {
    int num;
    if (argint(0, &num) < 0)
        return -1;
    if (num < 0 || num >= NUM_SYSCALLS)
        return -1;
    return syscall_count[num];
}
```

7. **Add a user-level wrapper**
   - In `user.h` add:
     `int getsyscallcount(int syscallnum);`
   - In `usys.S` add:

     `SYSCALL(getsyscallcount)`

8. **Implement user program `sccount.c`**
   - Calls `getsyscallcount` for specific syscalls (e.g., `SYS_getpid`), prints the result, and tests invalid syscall numbers.
9. **Update Makefile**
   - Add `_sccount` to the `UPROGS` list so it gets compiled and linked.
10. **Build and test**
    - Run `make clean && make` and then `make qemu` to launch xv6.
    - Execute `_sccount` inside xv6 shell to verify the syscall counts.

Testing

- Call the syscall multiple times from the user program.
- Verify the count returned matches the number of calls made.
- Test with invalid syscall numbers to check error handling.

```
$ _sccount
getpid was called 5 times
Invalid syscall count: -1
$
```

System Call 23: getproclist() system call

Target.

- Make a system call to display a list of the running processes with their info:
  - Size of process memory.
  - State of the process.
  - Process PID.
  - Parent Process.
  - Process Name.

Pre Design Decisions.

Deciding on the process's info.

- After inspecting the process's struct `proc` (that holds the per-process state) found in `proc.h`,

```c
// Per-process state
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
  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)
};
```

  - Decided to choose `sz` , `state` , `pid` , `parent` , `name` to represent each process in the list.

Approach in design.

- There are two approaches to design this system call:
  - Make a system call that prints these information directly to the kernel using `cprintf()` .
  - Define a struct of the desired info only, create it in the user space, pass it to the kernel, obtain information from kernel, and customize the output using `printf()` from the user side.
- I chose to build it using the second approach for consistency of design; the kernel only provides raw data, and user programs decide how to process it.
  - So, I created a struct called `pinfo` that holds the desired data.

- Then in the user space, I created an array of size `64` (maximum number of processes declared in `param.h`).

Steps.

1. Created a struct `pinfo.h` with the desired information to hold.
2. Added system call `SYS_getproclist` with number `23` in `syscall.h` .
3. Declared system call in the kernel in `syscall.c`
    1. `extern int sys_getproclist(void);`
    2. added it to the `syscalls` array.
4. Implemented `sys_getproclist(void)` :
    1. receives a pointer to the first struct in the array of structs of type `pinfo` . It fetches it using the `argptr()` system call implemented in `syscall.c` & then check if it is well-received or not.
        - Returns -1 if failed to fetch the pointer.
    2. It also receives an integer of the memory size for the array and fetches it using `argint()`
        - Returns -2 if failed to fetch the integer.
    3. initializes a counter of processes (initially = 0).
    4. acquires the lock on the `ptable` array (the table of processes in the system) to avoid returning incorrect results.
    5. iterates over all active (not UNUSED state) processes in the system.
    6. copies each process's desired info into a struct of type `pinfo` and add it to the array sent by the user.
        - here it uses `copyout` function implemented in `vm.c` to copy from kernel memory to user memory using the table page and returns if there is any errors.
    7. finally, it releases the lock and returns the number of processes.
5. Added the prototype `getproclist(struct pinfo *list, int max);` in `user.h` .
6. Added the system call wrapper `SYSCALL(getproclist)` in `usys.S` .
7. Implemented the user program `ps.c` where the user:
    1. creates a pointer to a list of type struct `pinfo` with size = 64.
    2. calls the `getproclist` system call.
    3. iterate over the received processes and display them.
    4. get notified if any errors happened during execution.
8. Added the `ps` program to `Makefile` .
9. Compiled using `make clean` & `make` .
10. Run through `make qemu-nox` .
11. Executed `ls` command and ensuring it is there.

12. Executing ps command and examining the output if it matches the expected output (list of processes with their names, PIDs, Parents, States, and Memory Sizes), which is the case.

```
$ ps
NAME    PID     PPID    MEMSIZE   STATE
init    1       0       12288     SLEEPING
sh      2       1       16384     SLEEPING
ps      3       2       12288     RUNNING
```

Testing.

To test if it produces the correct results or not:

1. Created more processes by running multiple shell commands (sh) and then executed ps command to test if they reflect in the ps or not.

```
$ sh
$ sh
$ ps
NAME    PID     PPID    MEMSIZE   STATE
init    1       0       12288     SLEEPING
sh      2       1       16384     SLEEPING
sh      4       2       16384     SLEEPING
sh      5       4       16384     SLEEPING
ps      6       5       12288     RUNNING
```

2. used the PID provided by the ps command to kill the executed sh shells above, and ran ps command again to check if they are killed or not.

```
$ kill 4
$ $ ps
NAME    PID     PPID    MEMSIZE   STATE
init    1       0       12288     SLEEPING
sh      2       1       16384     SLEEPING
ps      8       2       12288     RUNNING
sh      5       1       16384     RUNNABLE
$ kill 5
zombie!
$ ps
NAME    PID     PPID    MEMSIZE   STATE
init    1       0       12288     SLEEPING
sh      2       1       16384     SLEEPING
ps      10      2       12288     RUNNING
```

System Call 24: getppid() System call

1. Added it to the syscall file numbered 24
2. Then added the extern

```
extern int sys_getppid(void);
```

3. Added it to the array :

```
[SYS_getppid] sys_getppid,
```

4. Added the code logically in the file in the file “sysproc.c” that would implement the function for getting the parents process:

```
int sys_getppid(void) {   ///// for getting the parents proccess of a certain process
  struct proc *p = myproc();
  if (p->parent)
    return p->parent->pid;
  return -1;  // No parent (shouldn't happen)
}
```

5. Added a user level wrapper in the file user.h:

```
int getppid(void);
```

6. Added a syscall macro to `usys.S`:

```
SYSCALL(getproclist)
SYSCALL(getppid)
```

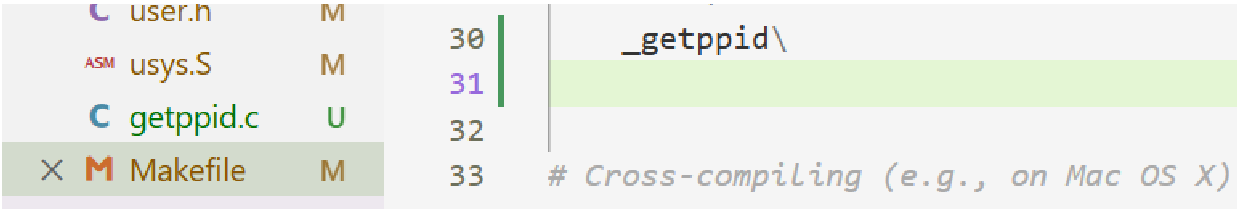
7. Created file getppid.c and saved the code needed in it:

getppid.c > ...

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

int main(void) {
    int pid = getpid();
    int ppid = getppid();
    printf(1, "My PID: %d\n", pid);
    printf(1, "My Parent PID: %d\n", ppid);
    exit();
}
```

8. Added in the makerfile:



9. Running it:

```
$ getppid
My PID: 4
My Parent PID: 2
```