CSL 301 - Operating System
Tutorial On XV6

Background: Understanding the Files Involved in Adding a System Call

When adding a new system call in **`xv6`**, several files must be changed, each bridging the user program to the kernel and playing a specific role:

- `proc.c` — Kernel-level process management logic.
  Manages process creation, scheduling, termination, and state transitions inside the kernel. Although it doesn't directly implement syscalls, it maintains process structures that many syscalls interact with (e.g., `fork()`, `exit()`, `wait()`). Provides the foundation for syscalls that modify or query process state.

- `sysproc.c` — Kernel syscall handler functions.
  Implements the logic executed when a syscall is invoked. For example, `sys_getyear()` returns 2025. These handlers validate input, perform kernel actions, and return results to user programs.

- `syscall.h` — Header file listing syscall numbers.
  Defines unique constant IDs (e.g., `#define SYS_getyear 22`) for each syscall to ensure consistent identification by the kernel.

- `syscall.c` — Syscall dispatcher mapping syscall numbers to handlers.
  Contains the syscall table mapping syscall IDs to their handler functions. When a syscall is invoked, the kernel uses this table to call the correct handler.

- `user.h` — Declares syscall prototypes for user programs.
  Provides function prototypes for syscalls so user programs can call them like regular functions.

- `usys.S` — Assembly syscall stubs for user space.
  Contains assembly code that acts as entry points for syscalls in user programs, setting up registers and triggering traps to kernel mode.

- User programs — Test and utility applications invoking syscalls.
  Run in user mode to test or use syscalls, verifying syscall functionality and serving as usage examples.

Let's discuss an example - "Adding the `getyear()` Syscall"

Step 1: Define syscall ID in `syscall.h`

Assign a unique ID for your syscall.

```
#define SYS_getyear 22
```

Why? Each syscall must have a unique numerical ID so the kernel can identify which syscall is being requested. Defining `SYS_getyear` in `syscall.h` ensures consistent use of this ID throughout the kernel and user space.

Step 2: Declare handler prototype and add mapping in `syscall.c`

Declare your handler and add to the syscall table:

```
extern int sys_getyear(void);

static int (*syscalls[])(void) = {
        // existing syscalls...
        [SYS_getyear] sys_getyear,
};
```

Why? The syscall dispatcher uses this table to map syscall numbers to their corresponding handler functions. Declaring the handler prototype and adding it to this table allows the kernel to call `sys_getyear()` when a `SYS_getyear` syscall is invoked.

Step 3: Implement handler logic in `sysproc.c`

For `getyear()`, simply return 2025:

```
int sys_getyear(void) {
        return 2025;
}
```

Why? This is the core implementation of the syscall. The handler runs in kernel mode and provides the service requested by the user program. In this case, it returns the fixed value 2025 as the current year.

Step 4: Declare prototype in `user.h`

```
int getyear(void);
```

Why? Declaring the syscall prototype here allows user programs to call `getyear()` like any other function. This header file bridges the user-level call with the kernel syscall mechanism.

Step 5: Add syscall stub in `usys.S`

```
SYSCALL(getyear)
```

Why? The assembly stub sets up the user-to-kernel transition, packaging the syscall number and arguments, then triggering a trap to enter kernel mode. This stub enables the user program to invoke the syscall correctly.

Step 6: Write a user test program `testgetyear.c`

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

int main(void) {
        printf(1, ‘‘Current year from kernel: %d\n’’, getyear());
        exit();
}
```

Why? This user program tests your syscall end-to-end, verifying that the user-space call correctly invokes the kernel handler and returns the expected result.

Step 7: Add program to `Makefile`

Add your test program to `UPROGS`:

```
UPROGS=\
...
_testgetyear\
```

Why? Adding your test program to `UPROGS` ensures it will be compiled and included in the kernel's user programs when you build the OS, allowing you to run and test it in the xv6 environment.

Summary: How These Files Work Together

1. User program calls syscall function.
2. The user-space assembly stub in `usys.S` triggers a trap to kernel mode.
3. `syscall.c` dispatches the syscall number to the correct kernel handler.
4. Handler in `sysproc.c` runs kernel logic (may use `proc.c` for process info).
5. Kernel returns the result to user space.
6. User program prints the result.

Each file has a clear role in the transition and execution flow between user and kernel, keeping the OS modular and maintainable.