System Calls

1. GOTTA COUNT 'EM ALL

Added array of size 32 to process struct. In the syscall() function in kernel/syscall.c, whenever a syscall is called, add it to the count. In the syspecsec, extract the required syscall from the mask and return the value of it from the current process if it's valid. While forking, syscall count is copied from parent to child. The user program syscount forks and executes the command and the parent process waits and gets the appropriate count of the syscall and prints it.

2. WAKE ME UP WHEN MY TIMER ENDS

Added the following to proc struct in proc.h:

- int interval: duration after which handler called
- int current_ticks: track ticks passed so far
- int alarm_status: check if alarm currently set or not
- uint64 alarm handler: pointer to handler function
- **struct trapframe** ***alarm_tf**: copies trapframe for sigreturn

Initialized/freed these in allocproc()/freeproc() respectively. Added appropriate definitions/definitions in kernel/syscall.c, kernel/syscall.h, user/usys.pl, and user/user.h. Added the new syscalls to the list in user/syscount.c

In kernel/sysproc.c:

- Defined function uint64 sys_sigalarm(void) which updates the process with the alarm interval and handler and initializes current_ticks and alarm_status to 0.
- Defined function uint64 sys_sigreturn(void) which restores the original trapframe, frees the copied alarm trapframe, and resets alarm_status to 0. It also invokes usertrapret() which returns to userspace from the trap and continues execution of the process, now with the updated/restored trapframe.

In kernel/trap.c:

- Modified the usertrap() function
- Updated the section pertaining to timer interrupts
- If the interval is > 0, that means some interval is set, so now we check the status of the alarm. If it is 0, we can proceed as normal and update the current_ticks attribute of the process.
- Now, if the current_ticks reach the interval duration, we reset current_ticks to 0, and we
 now set alarm_status = 1. We use kalloc() and memmove() to store the state of the
 trapframe, and then update the program counter to the signal handler.

Scheduling

1. THE PROCESS POWERBALL

- Added scheduler maco to makefile.
- Added user program settickets that takes input and sets current process tickets accordingly
- Added settickets syscall to list of syscalls
- Modified proc struct in proc.h to have an additional field for tickets
- Initialize tickets to 0 in allocproc in proc.c
- In fork, set child process's tickets to same amount as it's parents tickets
- In the scheduler function, used preprocessor directives to switch between round robin and lottery based scheduling
- Now in the LBS scheduler, we iterate through all runnable processes to calculate total tickets held
- Defined a random generator and seed function in proc.c
- We then initialize the seed based on system tick count and accordingly generate random numbers
- The random number generated is deemed the winning ticket of the lottery
- Now again iterate through all processes, subtracting ticket values from the winning ticket till we get the process that holds the winning ticket
- We set this process to running and a context switch occurs to hand over control

```
xv6 kernel is booting
Using LBS
hart 2 starting
hart 1 starting
init: starting sh
$ schedulertest
Process 9 finished
Process 3 finished
Process 4 finished
Process 8 finished
Process 0 finished
Process 1 finished
Process 2 finished
21 ticket. rtime: 0, wtime: 41
19 ticket. rtime: 0, wtime: 41
17 ticket. rtime: 0, wtime: 41
15 ticket. rtime: 0, wtime: 40
13 ticket. rtime: 0, wtime: 40
11 ticket. rtime: 22, wtime: 19
9 ticket. rtime: 27, wtime: 13
Process 7 finished
7 ticket. rtime: 26, wtime: 17
Process 6 finished
5 ticket. rtime: 24, wtime: 22
Process 5 finished
3 ticket. rtime: 13, wtime: 38
Average rtime 11, wtime 31
```

The output for the lottery based scheduler can be seen above.

```
xv6 kernel is booting
Using RR
hart 2 starting
hart 1 starting
init: starting sh
$ schedulertest
Process 8 finished
21 ticket. rtime: 26, wtime: 19
Process 7 finished
19 ticket. rtime: 25, wtime: 21
Process 6 finished
17 ticket. rtime: 25, wtime: 21
Process 9 finished
15 ticket. rtime: 25, wtime: 21
Process 5 finished
13 ticket. rtime: 15, wtime: 32
PPPrrorcoecsess so 1c efi2s sf inniisshheedd
0 fiPPnrirsohceeoscseds
41 1f itfiinniisshheedd
cket. rtime: 0, wtime: 200
9 ticket. rtime: 0, wtime: 200
7 ticket. rtime: 0, wtime: 200
5 ticket. rtime: 0, wtime: 200
3 ticket. rtime: 0, wtime: 200
Average rtime 11, wtime 111
```

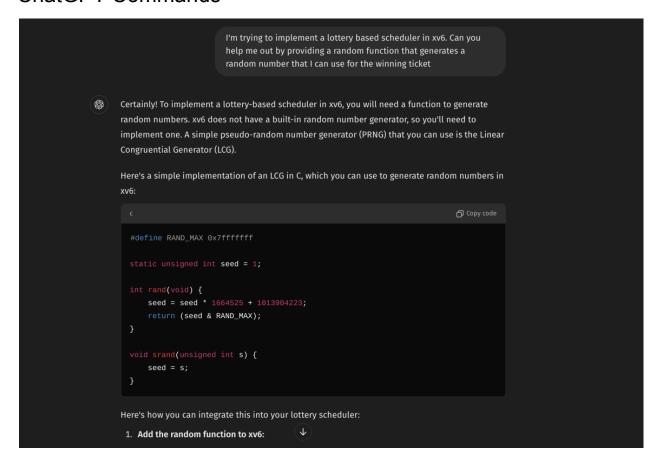
Meanwhile, above is the output for RR scheduling.

Answer to report question:

Adding arrival time ensures that if multiple processes have the same number of tickets, the older process will get higher priority. This helps avoid older processes from starvation. However, this approach also brings with it the additional overhead of checking and managing arrival times.

If all processes had the same number of tickets, the scheduler would essentially operate the same as a First Come First Serve scheduler, as it would then just continually choose the oldest process.

ChatGPT Commands



1. Add the random function to xv6:

 Add the rand and srand functions to a suitable file in the xv6 source code. You can create a new file called rand.c and add these functions there.

2. Include the random functions in your scheduler:

 In your scheduler code, you can include the rand.h header (which you'll need to create) to access the rand and srand functions.

3. Initialize the seed:

 Initialize the seed using srand at some point during the boot process or at the start of the scheduler.

4. Generate a random number to pick the winning ticket:

 Use the rand function to generate a random number each time you need to pick a winning ticket.

Here's how you can create the rand.c and rand.h files:

rand.c:

```
c
#include "rand.h"

#define RAND_MAX 0x7fffffff

static unsigned int seed = 1;

int rand(void) {
    seed = seed * 1664525 + 1013904223;
    return (seed & RAND_MAX);
}
```

This implementation uses a simple LCG to ge. \checkmark ate random numbers, and the scheduler uses

```
void scheduler(void) {
    ...
    srand(ticks); // Initialize the seed with the system tick count or any oth
    int total_tickets = get_total_tickets(); // Function to calculate total tic
    int winning_ticket = rand() % total_tickets;

struct proc *p;
    int current_ticket = 0;
    for(p = ptable.proc; p < &ptable.proc[NPROC]; p++) {
        if(p->state != RUNNABLE)
            continue;
        current_ticket += p->tickets;
        if(current_ticket > winning_ticket) {
            // p is the chosen process
            // Context switch to p
            ...
            break;
        }
    }
    ...
}
```

This implementation uses a simple LCG to generate random numbers, and the scheduler uses these random numbers to select a winning ticket for the lottery. This is a straightforward approach and should work well for basic lottery scheduling in xv6.



what are the general steps to add a syscall to xv6



Adding a system call (syscall) to the XV6 operating system involves several steps. Here's a general guide:

Step 1: Choose a Syscall Name and Function

- Decide on the functionality: Determine what the syscall will do (e.g., read a file, get system information).
- 2. Name the syscall: Choose a descriptive name for your syscall, such as <code>mysyscall</code>.

Step 2: Define the Syscall

- 1. Implement the function:
 - Create the implementation of your syscall in a .c file, usually in the kernel directory.
 For example, sys_mysyscall.c.

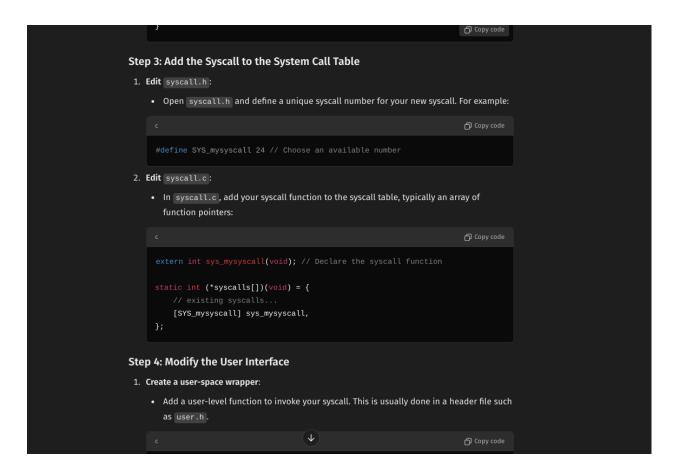
```
c
int sys_mysyscall(void) {
    // Implementation of the syscall
    return 0; // Return an integer as a status or result
}
```

Step 3: Add the Syscall to the System Call Table

- 1. Edit syscall.h:
 - Open syscall.h and define a unique syscall number for your new syscall. For example:



Copy code



```
c
int mysyscall(void) {
   return syscall(SYS_mysyscall);
}
```

Step 5: Update the Makefile

- 1. Include your new syscall file:
 - Ensure that the newly created <code>.c</code> file is included in the compilation process. Update the <code>Makefile</code> in the <code>kernel</code> directory:

```
makefile

OBJS = sys_mysyscall.o ... // Add your syscall object file here
```

Step 6: Test the Syscall

- 1. Create a user-level test program:
 - Write a simple user program that calls your new syscall to test its functionality.

```
c

#include "user.h"

int main(int arge, char *argv[]) {
   mysyscall(); // Call your syscall
   exit(0);
}
```

2. Compile and run the test:

Copy code

2. Compile and run the test:

• Compile your user program and run it to see if it works as expected.

Step 7: Debugging and Refinement

- 1. **Check for errors**: If the syscall does not work as intended, use print statements or a debugger to identify issues.
- 2. Refine the implementation: Make any necessary adjustments based on your testing.

Example Summary

This example briefly summarizes the process for adding a syscall named | mysyscall :

- Implement sys_mysyscall in sys_mysyscall.c.
- Define it in syscall.h.
- Add it to the syscall table in syscall.c.
- Create a user-level wrapper in user.h.
- Update the Makefile.
- Write a test program to validate your syscall.

Note

Make sure you consult the existing syscalls in XV6 as references to ensure consistency in your implementation.

ው ው ው ወ ፎ∨