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 `sys_getSysCount()` function in kernel/`sysproc.c`, 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.

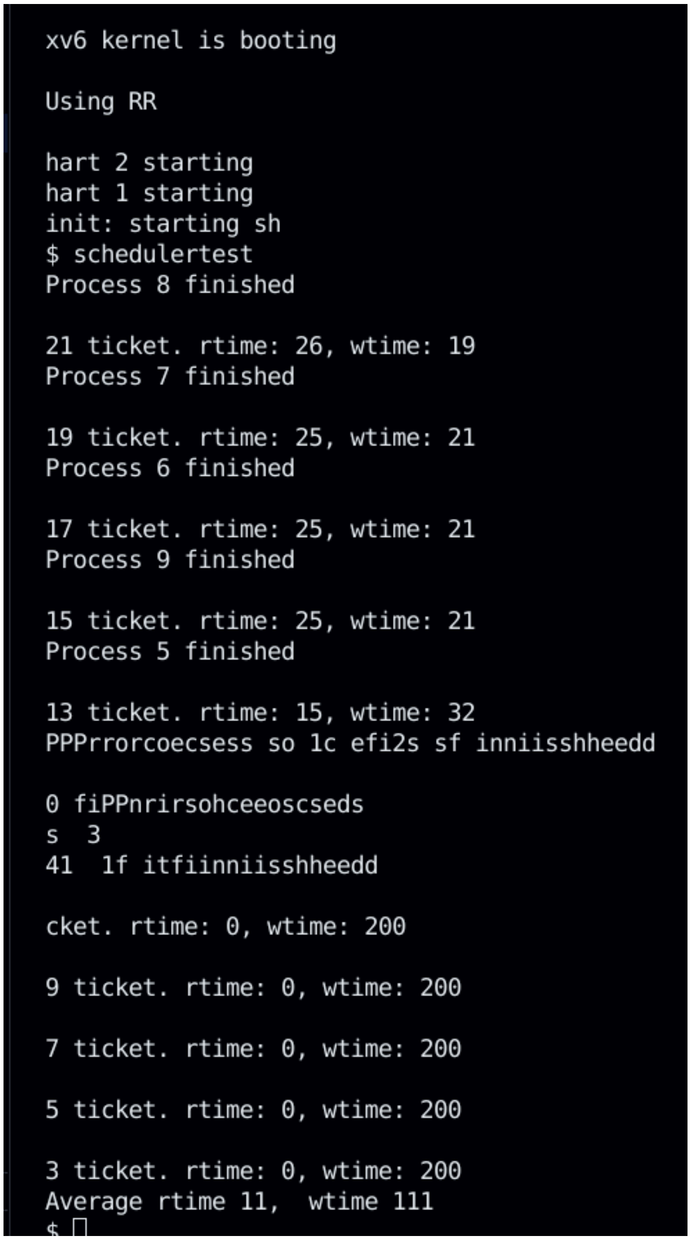
Scheduling

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
s  3
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.

Networking

1. XOXO

2. Fake it till you make it

Firstly created header.h:

- Define port for communication, size of each chunk and max chunks allowed
- Defined timeout interval, 100000 microseconds, or 1 second
- Defined packet struct containing it's sequence number, total chunks and the data itself
- Defined packet info struct which contains the packet, the time it was sent at and if it's been acked or not

Created server.c:

- Initialize socket, Packet to store incoming packet, ack, array for each chunk, total chunks and chunks received
- In an infinite loop, listen for messages
- If we received some data, print we received this chunk
- Send an acknowledgement for that chunk
- For first chunk, get total chunks and set NULL in the array to store upcoming chunks
- If value in array for any chunk seq num is currently NULL, allocate memory for that chunk's message and store it. [Data sequencing]
- If we've received all chunks, loop through array and print values and free memory. Now reset, total, chunks received for next message and start over

Created client.c:

- Initialize ack, message, packet info, socket
- Set the socket to non-blocking mode
- In an infinite loop, get message from user
- Calculate the total chunks needed to transmit the message based on the input
- Set up packet info by setting sequence num, total chunks and message value. Also define acked as 0 by default and get the current time to be the time it is sent
- While all values haven't been acked, loop through them and check if they've been acked, If any hasn't all acked becomes false. Now check the time that the non acked packet was sent at. If it's over our time definition, resend it and update the send time. We also add a timeout to decide when to next resend it

- In this ack loop we also check if anything has been received from the server in the form of an ACK
- If yes, update that packet's info with acked and print that we have received the ack

ChatGPT Commands

I'm trying to implement a lottery based scheduler in xv6. Can you help me out by providing a random function that generates a random number that I can use for the winning ticket

Certainly! To implement a lottery-based scheduler in xv6, you will need a function to generate random numbers. xv6 does not have a built-in random number generator, so you'll need to implement one. A simple pseudo-random number generator (PRNG) that you can use is the Linear Congruential Generator (LCG).

Here's a simple implementation of an LCG in C, which you can use to generate random numbers in xv6:

c Copy code

```c
#define RAND_MAX 0x7fffffff

static unsigned int seed = 1;

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

void srand(unsigned int s) {
    seed = s;
}
```

Here's how you can integrate this into your lottery scheduler:

1. **Add the random function to xv6:**

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 Copy code

```c
#include "rand.h"

#define RAND_MAX 0x7fffffff

static unsigned int seed = 1;

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

```
    return (seed & RAND_MAX);
}

void srand(unsigned int s) {
    seed = s;
}
```

rand.h:

```c
#ifndef RAND_H
#define RAND_H

void srand(unsigned int s);
int rand(void);

#endif
```

Integration in xv6:

1. Add the `rand.c` to your `Makefile`:
   - Find the line where the list of source files is defined and add `rand.c` to that list.

   ```makefile
   ...
   OBJS= \
       rand.o \
       ...  # other objects
   ```

2. Use the random number generator in you scheduler:

2. Use the random number generator in your scheduler:

c Copy code

```c
#include "rand.h"
```

c Copy code

```c
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 ge ate random numbers, and the scheduler uses c

Copy code

```c
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

1. **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 `mysyscall`.

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 Copy code

```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:

c Copy code

```
}
```

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:

```c
#define SYS_mysyscall 24 // Choose an available number
```

2. **Edit `syscall.c`:**
   - In `syscall.c`, add your syscall function to the syscall table, typically an array of function pointers:

```c
extern int sys_mysyscall(void); // Declare the syscall function

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

Step 4: Modify the User Interface

1. **Create a user-space wrapper:**
   - Add a user-level function to invoke your syscall. This is usually done in a header file such as `user.h`.

```c
```

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

Step 5: Update the Makefile

1. **Include your new syscall file**:
   - Ensure that the newly created `.c` file is included in the compilation process. Update the `Makefile` in the `kernel` 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 argc, char *argv[]) {
    mysyscall(); // Call your syscall
    exit(0);
}
```

2. **Compile and run the test**:

```
}
```

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.

how can i use non blocking sockets and udp to send data in C

Using non-blocking sockets with UDP in C involves several steps. Here's a complete example of how to set up a non-blocking UDP socket, send data, and handle sending and receiving without blocking. This example includes creating a socket, setting it to non-blocking mode, and sending data.

Step 1: Include Required Headers

c Copy code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <errno.h>
```

Step 2: Create a Non-Blocking UDP Socket

You will create a UDP socket and set it to non-blocking mode.

c Copy code

```c
int create_non_blocking_udp_socket() {
    int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        perror("socket creation failed");
```

```
#include <unistd.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <errno.h>
```

Step 2: Create a Non-Blocking UDP Socket

You will create a UDP socket and set it to non-blocking mode.

```c
int create_non_blocking_udp_socket() {
    int sockfd = socket(AF_INET, SOCK_DGRAM, 0);
    if (sockfd < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    // Set the socket to non-blocking mode
    int flags = fcntl(sockfd, F_GETFL, 0);
    if (flags < 0) {
        perror("fcntl get flags failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    if (fcntl(sockfd, F_SETFL, flags | O_NONBLOCK) < 0) {
        perror("fcntl set non-blocking failed");
        close(sockfd);
        exit(EXIT_FAILURE);
    }

    return sockfd;
}
```

```
    return sockfd;
}
```

Step 3: Send Data Using Non-Blocking Socket

Now you can implement a function to send data. If sending fails due to the socket being non-blocking, you can handle that error appropriately.

```c
void send_data(int sockfd, const char *message, const char *ip, int port) {
    struct sockaddr_in dest_addr;
    memset(&dest_addr, 0, sizeof(dest_addr));
    dest_addr.sin_family = AF_INET;
    dest_addr.sin_port = htons(port);
    inet_pton(AF_INET, ip, &dest_addr.sin_addr);

    ssize_t sent_bytes = sendto(sockfd, message, strlen(message), 0,
                                (struct sockaddr *)&dest_addr, sizeof(dest_addr));

    if (sent_bytes < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            // Would block, handle this case
            printf("Send would block, try again later\n");
        } else {
            perror("sendto failed");
        }
    } else {
        printf("Sent %ld bytes: %s\n", sent_bytes, message);
    }
}
```

Step 4: Example Usage

Now you can put everything together and create a simple program that sends data over UDP.

```c
int main() {
    int sockfd = create_non_blocking_udp_socket();
    const char *message = "Hello, UDP!";
    const char *ip = "127.0.0.1"; // Change this to the target IP
    int port = 8080; // Change this to the target port

    // Attempt to send data
    send_data(sockfd, message, ip, port);

    // Close the socket
    close(sockfd);
    return 0;
}
```

Explanation

1. **Socket Creation**: The `create_non_blocking_udp_socket` function creates a UDP socket and sets it to non-blocking mode using `fcntl`.
2. **Sending Data**: The `send_data` function prepares the destination address and sends the message using `sendto`. If sending would block (since the socket is non-blocking), it prints a message and continues.
3. **Error Handling**: If an error occurs (other than would-block), it prints the error.

Compilation and Running

```
    return 0;
}
```

Explanation

1. **Socket Creation**: The `create_non_blocking_udp_socket` function creates a UDP socket and sets it to non-blocking mode using `fcntl`.
2. **Sending Data**: The `send_data` function prepares the destination address and sends the message using `sendto`. If sending would block (since the socket is non-blocking), it prints a message and continues.
3. **Error Handling**: If an error occurs (other than would-block), it prints the error.

Compilation and Running

You can compile the code with `gcc`:

```bash
gcc -o udp_nonblocking udp_nonblocking.c
```

Then run it:

```bash
./udp_nonblocking
```

Note on Receiving Data

Receiving data with a non-blocking socket is similar; you would use `recvfrom` and check for `EAGAIN` or `EWOULDBLOCK` errors to handle the non-blocking nature.

Example of Receiving Data

```
./udp_nonblocking
```

Note on Receiving Data

Receiving data with a non-blocking socket is similar; you would use `recvfrom` and check for `EAGAIN` or `EWOULDBLOCK` errors to handle the non-blocking nature.

Example of Receiving Data

Here's a simple example of how to receive data with a non-blocking socket:

```c
void receive_data(int sockfd) {
    char buffer[1024];
    struct sockaddr_in src_addr;
    socklen_t addrlen = sizeof(src_addr);

    ssize_t received_bytes = recvfrom(sockfd, buffer, sizeof(buffer) - 1, 0,
                                      (struct sockaddr *)&src_addr, &addrlen);

    if (received_bytes < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            // Would block, handle this case
            printf("Receive would block, try again later\n");
        } else {
            perror("recvfrom failed");
        }
    } else {
        buffer[received_bytes] = '\0'; // Null-terminate the string
        printf("Received %ld bytes: %s\n", received_bytes, buffer);
    }
}
```

```c
void receive_data(int sockfd) {
    char buffer[1024];
    struct sockaddr_in src_addr;
    socklen_t addrlen = sizeof(src_addr);

    ssize_t received_bytes = recvfrom(sockfd, buffer, sizeof(buffer) - 1, 0,
                                      (struct sockaddr *)&src_addr, &addrlen);

    if (received_bytes < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
            // Would block, handle this case
            printf("Receive would block, try again later\n");
        } else {
            perror("recvfrom failed");
        }
    } else {
        buffer[received_bytes] = '\0'; // Null-terminate the string
        printf("Received %ld bytes: %s\n", received_bytes, buffer);
    }
}
```

You can call `receive_data(sockfd)` in a loop or after sending data to continuously check for incoming messages.

This code provides a solid foundation for working with non-blocking UDP sockets in C. You can expand upon this as needed for your application!