Operating Systems
Assignment 1

Rishi Jain
2020CS10373

February 15, 2023

Contents

1 Building and Testing XV6

XV6 is built using the following command:
git clone https://github.com/mit-pdos/xv6-public.git
cd xv6
make
make qemu-nox

For testing of the xv6, I created a user program which forks a child process and the child process prints the string "Hello World" and exits. The parent process waits for the child process to exit and then exits. The code for the same is as follows:

```
int main(int argc, char *argv[])
{
  int pid;
  pid = fork();
  if(pid == 0)
  {
    printf(1, "Hello World\n");
    exit();
  }
  else
  {
    wait();
    exit();
  }
}
```

2 System Calls

Each system call is implemented in the following manner:

1. The system call is defined in the file user.h. The system call is defined as a function with the same name as the system call. The function takes in the arguments that are required by the system call and returns the value returned by the system call.

2. The system call is implemented in the file proc.c. The system call is implemented as a function with the same name as the system call. The function takes in the arguments that are required by the system call and returns the value returned by the system call.

3. System call is mapped with a number in the file syscall.h. Also the system call is added to the array of system calls in the file syscall.c.

4. Further the sys version of the system call is defined in the file sysproc.c. The sys version of the system call is defined as a function with the same name as the system call. The function takes in the arguments that are required by the system call and returns the value returned by the system call.

5. Finally the system call is added in usys.S and defs.h.

2.1 toggle_trace()

The toggle_trace() system call is implemented in the following manner:

1. The kernel_state variable is defined in the file proc.h. The kernel_state variable is used to store the state of the kernel. The kernel_state variable is set to 1 when the kernel is in tracing mode and is set to 0 when the kernel is not in tracing mode.

2. syscall_count array is defined in the file proc.h. The syscall_count array is used to store the number of times each system call is called. The syscall_count array is indexed by the system call number.

3. syscall_names array stores the name of each system call. The syscall_names array is indexed by the system call number.

4. When a system call is called, the syscall(void) is called from syscall.c file. I've modified the syscall(void) function such that when the kernel is in tracing mode, the count of the system call is incremented in the syscall_count array.

5. Also when the kernel_state toggles to non tracing mode, the syscall_count array is reset to 0 inside the toggle function.

Basic snippet of the toggle_trace system call is as follows:

```
void toggle(void){
  if (kernel_state== TRACE_OFF){
    kernel_state = TRACE_ON;
  }
  else{
    kernel_state = TRACE_OFF;
    for (int i = 0; i < 29; i++){
      syscalls_count[i] = 0;
    }
  }
}
```

2.2 print_count()

The print_count() system call is implemented in the following manner:

1. The print_count() system call is implemented in the file proc.c. print_count() iterates over the syscall_count array and stores the name and count of each system call in two arrays. The name of the system call is stored in the array sysnames and the count of the system call is stored in the array syscallcalls for non zero counts.

2. Using bubble sort, the arrays are sorted in ascending order of their names.

Basic snippet of the print_count system call is as follows:

```
    void
    print_count(void){

      int count = 0;
      for (int i = 0; i < 29; i++){
        if (syscalls_count[i] != 0){
          count++;
        }
      }
      // list of non zero system calls
      int syscalls[count];
      char* sysnames[count];
      int j = 0;
      for (int i = 0; i < 29; i++){
        if (syscalls_count[i] != 0){
          syscalls[j] = i;
          sysnames[j] = syscalls_names[i];
          j++;
        }
      }

      // sort the array on names
      for (int i = 0; i < count; i++){
        for (int j = i+1; j < count; j++){
          if (strncmp(sysnames[i], sysnames[j],1000) > 0){
            int temp = syscalls[i];
            syscalls[i] = syscalls[j];
            syscalls[j] = temp;
            char* temp_name = sysnames[i];
            sysnames[i] = sysnames[j];
            sysnames[j] = temp_name;
          }
        }
      }
      // print the sorted array
      for (int i = 0; i < count; i++){
        cprintf("%s %d \n", sysnames[i], syscalls_count[syscalls[i]]);
      }
    }
```

2.3 add()

The add() system call is implemented in a similar manner. It takes two arguments and simply returns the sum of the two arguments.

2.4 ps()

The ps() system call is implemented in the following manner: Iterate over the ptable and print the pid and name of each process which is not in the UNUSED state.

```
void ps(void){
  struct proc *p;
  acquire(&ptable.lock);
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != UNUSED){
      cprintf("pid:%d name:%s \n", p->pid, p->name);
    }
  }
  release(&ptable.lock);
}
```

3 Inter_Process_Communication

3.1 Implementation of Buffer

The buffer is implemented as a circular queue. The buffer is implemented in the file proc.c. Since max number of process structures is NPROC, the size of the buffer is set to NPROC. The index of the process in ptable works as the index in buffer. When a process ends its buffer is cleared and can be allotted to another process. The circular queue has two pointers, one for the head and one for the tail. The head pointer points to the first element in the queue and the tail pointer points to the last element in the queue. The head pointer is incremented when an element is removed from the queue and the tail pointer is incremented when an element is added to the queue. Also the buffer is protected by a lock.The buffer is implemented as follows.

```
struct queue {
    char data[QUEUE_SIZE][MSG_SIZE];
    int front;
    int rear;
    int size;
    struct spinlock lock;
    };
```

3.2 Implementation of send()

The send() system call works as follows:

1. If the buffer is not initialized, the buffer is initialized.
2. The send() uses the pid to find the index of the process in the ptable. The index is used to find the queue of the process in the buffer.
3. The message is then enqued in the queue of the process after locking the queue(implemented in enque).
4. If buffer is full, -1 is returned.
5. The sys_send function calls the send() system call until the message is sent thus implementing the blocking send.

The code for the send() system call is as follows:

```
    int send(int sender_pid,int receiver_pid,void *msg)
    {

      int buffer_ind = find_proc(receiver_pid);
      if(msg_sent==0) {
        for(int i=0;i<NPROC;i++)
        {
```

```
            init_queue(&buffer[i]);
            msg_sent=1;
          }
        }

        char *msg1 = msg;

        enqueue(&buffer[buffer_ind], msg1);

        return 0;
    }
```

The msg_sent varaible is used to check if the buffer is initialized or not. If the buffer is not initialized, the buffer is initialized.

3.3 Implementation of receive()

The receive() system call works as follows:

1. If the size of the queue is 0, or the buffer is not initialized, -1 is returned.
2. The receive() uses the pid to find the index of the process in the ptable. The index is used to find the queue of the process in the buffer.
3. From the buffer the message is dequeued and copied to the user space.

The code for the receive() system call is as follows:

```
    int recv(void* msg)
    {
      int pid = myproc()->pid;

      int buffer_ind = find_proc(pid);
      if(msg_sent==0 || buffer_ind==-1 || buffer[buffer_ind].size==0) {
        return -1;
      }
      int front_val = buffer[buffer_ind].front;

      for(int i=0;i<MSG_SIZE;i++)
      {
        *((char*)msg +i) = buffer[buffer_ind].data[front_val][i];
      }


      dequeue(&buffer[buffer_ind]);
      return 0;

    }
```

3.4 Implementation of send_multi()

The send_multi() system call repetatively calls the send() for each pid in the pid array. The code for the send_multi() system call is as follows:

```
    int send_multi(int sender_pid,int *rec_pids,void *msg)
    {
      int sizeList = 8;

      for(int i=0;i<sizeList;i++)
      {
        if(rec_pids[i] > 0)
        {
          send(sender_pid,rec_pids[i],msg);
        }
      }
      return 0;
    }
```

4 Distributed algorithm

4.1 Implementation of the unicast version

In unicast version we have to find the sum of the elements in an array.
The unicast version of the distributed algorithm is implemented as follows:

1. The parent process reads the input from the user and creates an array of size 1000.
2. It then forks 7 times and creates 8 processes.
3. Parent process also stores the pid of the child processes in an array.
4. The array is distributed to the child processes in interleaving manner. For example, first child will add elements with index 0,7,14...., second child will add elements with index 1,8,15.... and so on.
5. After each child process has added its elements, it sends the sum to the parent process using the send() system call.
6. Parent process receives the sum from the child processes and adds them to get the final sum.

The code for the work of the child process and the parent process is as follows:

```
if(curr_pid != par_pid){

  int sum = 0;
  for(int i=ind-1;i<size;i+=NO_OF_PROCS-1){
    sum += arr[i];
  }

  send(curr_pid,par_pid,(void*)&sum);
  }
else{
  int sum = 0;
  for(int i=0;i<NO_OF_PROCS-1;i++){

    void* msg = malloc(8);
    while(recv(msg)==-1);
    int *ps = (int*)msg;
    sum += *ps;
          free(msg);
  }
  tot_sum = sum;
  }
```

Note that curr_pid is the pid of the current process and par_pid is the pid of the parent process.

4.2 Implementation of the multicast version

In multicast version we have to find the variance of the elements in an array.

$$Variance\ of\ elements\ of\ array\ (\sigma^2) = \frac{\sum(x_i - \mu)^2}{Number\ of\ elements\ in\ the\ array}$$

We can find the variance by finding the mean of the elements in the array and then finding the sum of the squares of the difference of the elements from the mean.
The multicast version of the distributed algorithm is implemented as follows:

1. The mean of the elements in the array is calculated by the parent process by adding all the elements and dividing by the size of the array. The sum is calculated in the same way as in the unicast version.
2. The parent process sends mean to all the child processes using the send_multi() system call. The array of pids is passed as an argument to the send_multi() system call.
3. After receiving the mean, each child process calculates the sum of the squares of the difference of the elements from the mean in the same way as in the unicast version.

4. After each child process has calculated the sum of the squares of the difference of the elements from the mean, it sends the sum to the parent process using the send() system call.

5. Parent process receives the sum from the child processes and adds them and divides by the size of the array to get the final variance.

The code for the work of the child process and the parent process is as follows:

```
if(curr_pid != par_pid){

  int sum = 0;
  for(int i=ind-1;i<size;i+=NO_OF_PROCS-1){
    sum += arr[i];
  }
  send(curr_pid,par_pid,(void*)&sum);
      float varsum = 0.0;
      float mean = 0.0;
      void* msg = malloc(8);
  int k =-1;
      while(k==-1){
    k = recv(msg);
  }
      mean = *(float*)msg;
      free(msg);

      for(int i=ind-1;i<size;i+=NO_OF_PROCS-1){
          varsum += ((float)arr[i]-mean)*((float)arr[i]-mean);
      }
      send(curr_pid,par_pid,(void*)&varsum);
      exit();
}
else{
  int sum = 0;
  for(int i=0;i<NO_OF_PROCS-1;i++){

    void* msg = malloc(8);
    while(recv(msg)==-1);
    int *ps = (int*)msg;
    sum += *ps;
          free(msg);
  }
  tot_sum = sum;
      float mean = (float)sum/size;
      pid_arr[0] = -1;
      send_multi(par_pid,pid_arr,(void*)&mean);

      float variance = 0.0;
      for(int i=0;i<NO_OF_PROCS-1;i++){
          void* msg = malloc(8);
          while(recv(msg)==-1);
          float *ps = (float*)msg;
          variance += *ps;
          free(msg);
      }
      variance = variance/size;
  tot_var = variance;
}
```

pid_arr is the array of pids of all processes. The first element of the array is set to -1 so that the parent process does not receive the mean from itself.