ELL783: Operating Systems

Assignment 1 Report

Ritik Choudhary 2017EE10483
Sunav Kumar Vidhyarthi 2017ME10618
Suryansh Agarwal 2017ME10620

1 Tracing the system call

We implemented this system call with the help of-

1. An Integer array *count_array*, to count how many times a system call has been called
   - It is of the size→ total number of system calls in our system + 1(to ignore the 0th index)
2. An integer variable *trace* denotes whether we want to print a trace or not
   - It has 2 states→ 0 & 1
   - It is initially set to 0, which means the default state is trace off
   - It can be changed to state 1, with the help of another system call(discussed in the next section), when we change its state to 1, we set all the entries in the array *count_array* to 0, so that we get a fresh array to count system calls again

The *syscall* function in *syscall.c* is the function that gets called every time a system call is called. So we implement the *sys_print_count* system call in this file itself. Whenever a system call gets called, we check if the *trace* is set to 1(this is done in the *syscall* function), if it is the case, we increment an element in the *count_array* array, else we do nothing.
When *sys_print_count* is called, we traverse the array *count_array* and go alphabetically according to the system call names, we created an array *ascending_order*, in this array all system calls are sorted in ascending order. Finally, if an element is not equal to 0 we print the system call name along with the count, the number of times the system call has been called.

```
int sys_print_count(void)
{
  int i = 0;
  while(i < NELEM(ascending_order) ) {
    int temp = ascending_order[i];
    if(count_array[temp] != 0){
      cprintf("%s %d\n",sys_call_names[temp],count_array[temp]);
    }
    i++;
  }
  return 0;
}
```

2 Toggling the tracing mode

As we discussed in the previous section, we implemented this system call with the help of an Integer variable *trace*, when *sys_toggle* is called,

- If *trace* is set to 0 which is its default state, then we change its value to 1
- It *trace* is set to 1, then we change its value to 0

Additionally to implement the previous system call we set the elements of array *count_array* to 0 if we change the toggle state from 0 → 1. This system call is also implemented in the *syscall.c* file so that we can use the *trace* flag along with the *sys_print_count* system call.

3 Add system call: *sys_add*

To implement this system call, we need to pass arguments from user-level functions to kernel-level functions, XV6 has its own built-in functions for passing arguments into a kernel function. To pass in an integer, the *argint()* function is called.
According to the xv6 book[1]-
System call implementations in the kernel need to find the arguments passed by user code. Because user code calls system call wrapper functions, the arguments are initially where the RISC-V C calling convention places them: in registers. The kernel trap code saves user registers to the current process's trap frame, where kernel code can find them. The function argint retrieves the n 'th system call argument from the trap frame as an integer, pointer, or a file descriptor.
We defined two integer variables *a*, *b* to hold the values of the system call i.e the numbers which are to be added.
We used the *argint* function to assign values to *a* and *b* and if the *argint* does not extract correctly we return -1. We define the position of the argument and pass the variables by address.

```
if(argint(0,&a)<0) return -1;
if(argint(1,&b)<0) return -1;
```

Finally we return *a+b* as the answer.

4 Process List: *sys_ps*

To implement this system call, we need to look at a couple of things-

1. *proc* struct - illustrates process structure in xv6
    - Each process has a PCB (process control block) defined by struct *proc* in xv6
    - Allows process to resume execution after a while
    - Keep track of resources used and the process state - *UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE*
    - We defined all the state other than the *UNUSED* state as running

[1] "xv6 book - PDOS-MIT." 31 Aug. 2020, http://pdos.csail.mit.edu/6.828/2020/xv6/book-riscv-rev1.pdf. Accessed 27 Mar. 2021.

```
// 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;     // switch() 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)
};
```

2. *ptable* struct - It is an array containing all PCBs
   It has the following structure→
   - A fixed size array of all processes
   - A semaphore lock to keep more than one thing from accessing it at once
   - NPROC is the maximum number of processes that can be present in the system
     Rule: don't change a process's state (RUNNING, etc.) without 'acquiring' lock

```
struct {
    struct spinlock lock;
    struct proc proc[NPROC]
}ptable;
```

Since we don't have access to *ptable* in *syscall.c* we added a function *print_proc* in *proc.c* to iterate over this *ptable* and do stuff. The *sys_ps* system call just calls *print_ps.*

Implementation-

1. Gain access to the *ptable*
2. Acquire the *ptable* lock as counting the processes without lock, opens up the possibility of the *ptable* changing while you're counting
3. Loop through it and print the processes which have a *state* which is not equal to *UNUSED*
4. Release the ptable lock after the loop is completed

```
int print_proc(void)
{
  struct proc *proc_head;

  acquire(&ptable.lock);

  proc_head = ptable.proc;
  while(proc_head < &ptable.proc[NPROC] ){
      if(proc_head->state != UNUSED || proc_head->state == RUNNING)
      {
        cprintf("pid:%d name:%s\n",proc_head->pid,proc_head->name);
      }
      proc_head++;
  }

  release(&ptable.lock);
  return 0;
}
```

5 IPC system calls for unicast communication

To implement this system call also, we need the *ptable,* hence we implemented the core logic in *proc.c* and the *sys_send* and *sys_recv* system calls in the *syscall.c* file checks the validity of the arguments and pass on the job to *send_msg* and *recv_msg* implemented in *proc.c.*
We defined four arrays-

1. *inbox* - This is a char array which acts like a global inbox for all processes
    - It is a 3D array of size [NPROC][MAX_MSG][MSG_SIZE], where NPROC=64,MAX_MSG = 20 and MSG_SIZE = 8
    - Any process can access messages that it has received from this inbox
2. *first_msg* - This acts as a pointer to the first message in the inbox for a particular process
    - It is of size NPROC and type int
3. *last_msg* - This acts as a pointer to the last message in the inbox for a particular process
    - It is of size NPROC and type int
4. *Inbox_size* - This array keeps track of the number of messages a particular process has received
    - It is of size NPROC and type int

Processes / Messages→ ↓	1	2	. . . . .	19	20
1					
2					
. . . .					
63					
64					

Fig: Visual representation of array *inbox*

Each cell in this 2-D array is of size = MSG_SIZE

```
char inbox[NPROC][MAX_MSG][8]; //acts as inbox for all processes
int first_msg[NPROC] = {0};
int last_msg[NPROC] = {0};
int inbox_size[NPROC] = {0};
```

First we will take a look at the sending process,

Implementation-

1. Get arguments from user-level functions to kernel-level functions
2. Check the validity of arguments
3. Call *send_msg* function in *proc.c*
4. Check if the receiver has exhausted the space allotted i.e if it has less than 20 messages stored, it not we send an error
5. If the receiver inbox has space, then we increment inbox_size[rec_pid]
6. Then we do a loop to copy the message in the global inbox
7. Then we increase the pointer of this processes inbox so that we can receive new messages at next location
8. Then we need to wake up the receiver process incase it is *UNUSED* or *SLEEPING*, so we acquire the ptable lock and change the state of the receiver process to RUNNABLE, and release the lock

```
int send_msg(int sender_pid, int rec_pid, char *msg){

  if(inbox_size[rec_pid] >= MAX_MSG){
    panic("Maximum messages limit crossed!");
    return -1;
  }
  else{
    int i = 0;
    inbox_size[rec_pid] += 1;
    while(i < MSG_SIZE){
      inbox[rec_pid][last_msg[rec_pid]][i] = *(msg+i);
      i++;
    }
    last_msg[rec_pid] += 1;
    struct proc *p;
    p = &ptable.proc[rec_pid];

    if(p->state == UNUSED || p->state == SLEEPING){
      acquire(&ptable.lock);
      wakeup1(&p);
      release(&ptable.lock);
    }
    return 0;
  }
}
```

Coming onto the receiver process,

Implementation-

1. We get the process id which is receiving via *myproc()→pid*
2. Let *myproc()→pid* => rec_pid
3. We check if the process has received any messages, if *inbox_size[rec_pid] == 0* then we just return a null message
4. Otherwise, we go into a loop to extract the message from the global inbox in the same way we sent the message
5. We decrement the *inbox_size[rec_pid]* so that we don't read the same message again
6. We increase the *first_msg* pointer for this process in the *first_msg* array, so that we don't read this message again

```
char* recv_msg(char *msg){

  if(inbox_size[myproc()->pid] == 0){
     return msg;
  }

  int b = 0;
  while(b<MSG_SIZE){
    *(msg + b) = (inbox[myproc()->pid][first_msg[myproc()->pid]][b]);
    b++;
  }
  inbox_size[myproc()->pid] -= 1;
  first_msg[myproc()->pid] += 1;
  return msg;
}
```

6 Distributed Algorithm to Calculate Sum of an array

We aim to calculate the sum of an array of size 1000 consisting of single digit positive integers using a distributed algorithm, using 8 processes. We approach the problem using a distributed algorithm by forking child processes in a *for* loop. As the parent process iterates over the main *for* loop, it forks a child process in each iteration. Each child process is assigned the job of calculating the sum of chunks of the array. Chunk, here is the contiguous subarray of the main array of size 125. We have following two strategies for calculating the sum of the main array

1. Calculate the sum of the whole array in chunks of size 125 in 8 child processes and then accumulate the sum in the main process
2. Again, calculate the sum of the whole array in chunks of size 125 but use 7 child processes and the parent process to calculate the sum. Then, the parent process accumulates the sum.

For sake of brevity and easy understandability of the user, we used the approach 1.
Child processes use the *add(int a, int b)* system call, as we implemented in part 2 of the assignment, to calculate the sum of their respective arrays. In order for parent process to accumulate the sum from child processes and child process to report their respective sums to their parent process,, child processes use *send(int myid, int recid, char* msg)* system call. Here, we saw the technique of message passing which forms fundamentals of distributed systems.

```
int ppid = getpid();
for(int i = 0 ; i < NUM_PROC; i++){
  int cid = fork();
```

```
   if(cid < 0){
        printf(1,"Forking %i process failed", i);
    }
     else if (cid == 0){
        int start = ((i*1000)/NUM_PROC);
        int end = start + 125;
        int sum = 0;
        for(int j = start; j < end; j++){
            sum = add(sum, arr[j]);
        }
           char *msg = itos(sum); // itos→ integer to string
        send(getpid(),ppid,msg);
        exit();
    }
}
```

After the main process completes its iterations over the main loop which forks the size processes, it *waits()* outside the main *for* loop for its child processes. After all the child processes report to their parent process, the child processes must have sent their respective calculated sums to the receive message buffer of the parent process simulating a mailbox.

```
for(int k = 0 ; k < NUM_PROC ; k++){
    wait();
}
```

Next task is to gather the sum from the receive message buffer. To do that, parent process iterates over another for loop which calls the *receive(char* msg)* system call to retrieve the sums from its receive message buffer. The loop must run as many times as there were child processes. We define the number of child processes using the *#define* preprocessor directive as ***NUM_PROC***. As the parent process finishes gathering sums from its child processes, it keeps accumulating the sum of each chunk in an int variable called *tot_sum* using again the *add(int a, int b)* system call. As the program finishes, we report the total sum with the name of the input array file and the parent process exits by calling *exit()* system call. Core Logic Snippet→

```
tot_sum = 0;
for(int i = 0 ; i < NUM_PROC; i++){
     char *msg_child = (char *)malloc(MSGSIZE);
     int stat=-1;
     while(stat==-1){
           stat = recv(msg_child);
     }
     tot_sum = tot_sum + stoi(msg_child);// stoi→ string to int
     free(msg_child);
}
```