Assignment 1 (EASY) REPORT

Rajat Bhardwaj(2020CS50436)

0. Description

The main function of some system calls are in syscall.c and some are in sysproc.c. I have mentioned the name of the file in which each functions are located and also pasted snippets of functions

I have made relevent changes in the header files and usys.S file which I have not mentioned.

1. System calls

1.1 sys_toggle() (in syscall.c)

I have declared a struct enum "trace" which can have values in {TRACE_OFF, TRACE_ON} in the syscall.c file

Now the function sys_toggle(void) is also implemented in syscall.c which which called, changes the state of trace using simple if else command.

When trace is set to TRACE_OFF from TRACE_ON, it also clears the system calls that have been recorded

```
enum trace trace;

int sys_toggle(void)
{
  if (trace == TRACE_OFF)
  {
    trace = TRACE_ON;
  }
  else if (trace == TRACE_ON)
  {
    trace = TRACE_OFF;
    for (int i = 0; i < 28; i++)
    {
      all_system_calls[i] = 0;
    }
  }
  return 22;
}
```

In the syscall() function of the syscall.c I had made the following changes

```
if (num > 0 && num < NELEM(syscalls) && syscalls[num])
{
  if (trace == TRACE_ON)
  {
    if (num != 22 && num != 23)
    {
      all_system_calls[num - 1]++;
    }
  }
  curproc->tf->eax = syscalls[num]();
```

`num` is the number of the system call, an array of integers `int all_system_calls[28];` stores the count for each system call

1.2 sys_print_count() (in syscall.c)

I have mapped the system call number to the name of the system calls in sorted order.

I iterate in the array `order[28]` which returns me the number of system call number according to lexicographic order(system call name)

Now I check the number of times it has been called by any process
If the frequency is greater than 0 I simply print it on terminal.

```
int sys_print_count(void)
{

  int order[28] = {24,9,21,10,7,2,1,8,11,6,19,20,17,15,4,23,25,5,27,12
  char *syscall[28] = {"sys_add","sys_chdir","sys_close","sys_dup","sy
  for (int i = 0; i < 28; i++)
  {
    if (all_system_calls[order[i] - 1] != 0)
    {
      cprintf("%s %d\n", syscall[i], all_system_calls[order[i] - 1]);
    }
  }

  return 20;
}
```

1.3 sys_add() (in sysproc.c)

```
int sys_add(void)
{
  int a , b;
  argint(0 , &a);
  argint(1 , &b);
  return a+b;
}
```

It just reads the two input integers using argint and adds them and returns the result

1.4 sys_ps() (in sysproc.c)

It calls another function `print_all_processes` which is written in `proc.c`

```
int print_all_process()
{
  struct proc *p;
  acquire(&ptable.lock);
  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if (p->pid != 0)
    {

      cprintf("pid:%d name:%s\n", p->pid, p->name);
    }
  }
  release(&ptable.lock);
  return 20;
}      You, 23 hours ago • dist left
```

This proc[] in ptable struct contain all the processes that are running or can be allocated. The total number of processes are NPROC.
We iterate through proc (of the ptable) and check if the pid of the proc is not equal to zero (because pid = 0 means the process has not been assigned) and print the process ID and the name of the process

2 Inter-Process Communication

I have made a struct of queue in proc.c

```
struct queue
{
  char bufq[20][9];

  int front;
  int rear;

} queue;
```

I have declared an array of queues `struct queue unicast_msg_buffer1[NPROC + 1];` in the proc.c file.
The process with pid= i with be associated with the ith element of this array. The queue at that index will store the pending messages to be received by the process.

I have initialized the array queue for each processes in the function allocproc using the reset() function

```
void reset()
{
  for (int i = 0; i < NPROC + 1; i++)
  {
    unicast_msg_buffer1[i].front = 0;
    unicast_msg_buffer1[i].rear = -1;
  }
}
```

2.1 Unicast

2.1.1 sys_send() (in sysproc.c)

It just takes the input using argint and argstr and calls another function named `send_msg`

```
int sys_send(void)      You, 23 hours ag
{
  char *msg;
  int sender_pid, recvr_pid;
  argint(0, &sender_pid);
  argint(1, &recvr_pid);
  argstr(2, &msg);
  send_msg(sender_pid, recvr_pid, msg);
  return 0;
}
```

`send_msg()` is written in proc.c

```
int send_msg(int sender_pid, int recvr_pid, char *msg)
{
  unicast_msg_buffer1[recvr_pid].rear++;
  strncpy(unicast_msg_buffer1[recvr_pid].bufq[unicast_msg_buffer1[recvr_pid].rear], msg, 9);
  return 0;
}      You, 23 hours ago • dist left
```

The queue.rear int points to the index where the last message was written, therefore first we have to increment rear and we simply write the message to the queue.bufq[rear]

2.1.2 sys_recv() (in sysproc.c)

It just returns another function `recv_msg(msg)` which is written in proc.c.

```
int recv_msg(char *msg)
{

  int ret = -1;

  if (unicast_msg_buffer1[myproc()->pid].rear - unicast_msg_buffer1[myproc()->pid].front != -1)
  {
    strncpy(msg, unicast_msg_buffer1[myproc()->pid].bufq[unicast_msg_buffer1[myproc()->pid].front], 8);
    unicast_msg_buffer1[myproc()->pid].front++;
    ret = 0;
  }
      You, 24 hours ago • dist left …
  return ret;
}
```

This function first checks if the buffer is empty by taking the difference of rear and front. If the queue is empty, then return -1. Else, we just copy the data in the buffer in the string `msg` and increment the front by 1.

2.2 Multicast

2.2.1 sys_send_multi() (in sysproc.c)

```
int sys_send_multi(void)
{
  int sender_pid;
  char *receiver_pid;
  char *msg;

  argint(0, &sender_pid);
  argptr(1, &receiver_pid, 8 * 4);
  argstr(2, &msg);
  send_multiple_msgs(sender_pid, receiver_pid, msg);
      You, 22 minutes ago • Uncommitted changes
  return 0;
}
```

The input pid and msg was taken simpy via argint and argstr
For the input of int array having the pids of the processes the input is taken using argptr
Basically it returns the starting address of the array of maximum size 8. These parameters are passed in the function `send_multiple_msgs()` which is implemented in `proc.c`

```
int send_multiple_msgs(int sender_pid, char *receiver_pid, char *msg)
{
  for (int i = 0; i < 8; i++)
  {
    int cur_pid = *(receiver_pid + i * 4);
    if (cur_pid == -1)
      break;
    send_msg(sender_pid, cur_pid, msg);
  }
  return 0;
}
```

This function first iterates using the pointer receiver_pid
we increment by 4 at each iteration to get the address of the next pid and we return the integer written at that address.
Then we simply use `send_msg()` to send the message to the pid at the current iteration.

3. Distributed Algorithm

3.0 Some helper functions

`void itoa1(int j, char *c, int sz)`
This function takes integer j as an input and a char pointer and converts integer j into a string and copies to c

`int stoi(char *c, int sz)`
This function takes a string c as an input and converts it to integer and returns that integer

`void fts(float f, char *c, int sz)`
This convert float to string using the following algorithm
First the integer part and the float part are separated
A temporary string out1 is declared.
The integer part is first writte on the `out1` string in reverse manner as the ones digit is written first an so on.
After the integer part is written on the string then the integer part of the string is reversed to make it in the right order.
Then we simply extract each digit from the float part by multiplying it by 10 and type casting it to int for all the remaining digits and these are written on the `out1` string
the `out1` string is then copied to `c`

`float stof(char *c)`
This function converts string to float.
First we extract the int part from the string (untill we encounter the char '.')
Then we extract the float part digit by digit and keep dividing it by 10^i where i digits after decimal are extracted.
We add the int part to the float part and return it

3.1 Unicast

We run a `for loop` for the number of helper processes

At each iteration of this for loop we first fork the parent process P to make a child process C. at the next command we simply use goto() to send the child process to the branch `child`

For the parent process (at each iteration)

We store the pid of the current child created in `int child_pid[]` array
We wait for the message to be received in the `char sums[][9]` array
When the messages are received, we simply take the sum by running a loop over the `char sums[][]` array first converting the string to int using the helper function `stoi()`. After collecting and adding all the sums, we just print it and send it to branch unimulti.

For the child process

A variable `int curr` created before forking and set to the number process created.
For each child, we iterated through 1000/(NUM_HELPER) elements of the array. At each iteration, we increase the index by NUM_HELPER. We add the element of the array to our sum.
After getting the sum of the elements relevant to each process, we convert the sum to string via `itoa1()` `helper function and then simply send it to the parent using` send()` function. Then we go to the branch unimulti using goto() function

3.2 Multicast

It is implemented int the BRANCH `unimulti`
We first check whether the type is equal to 1 or not, if not then we simply exit the function.
If the type is 1 then we check whether the process is parent or child by comparing it to the parent_pid

If the process is parent

We calculate the mean by dividing it by the number of elements in the array.
We then convert this float(mean) to a string using the helper function `fts()`
Then we simply send it to all the processes using the send_multi() function.
Then we wait for the message from each process by running a `for loop` for NUM_HELPER number of times.
In each iteration, we use `recv()` function to receive a message from all children.
We then convert the received message to float using `stof()` helper function and add it to the `variance` variable.
After all the messages from all processes are received, we simply divide `variance` by number of elements in the string to get the variance and convert it to string print the correct variance

If the process is child

We wait for the parent process to send the message and receive the mean using `recv()` function.
We convert the received mean to float using `stof()` function

Then we add the square of the difference of mean and each element (1000 / NUMBER OF PROCESSES number of elements) relevant to the current process.
Then we convert it to string and send it to the parent process using the `send()` function.