COL331 - Operating Systems
Assignment 1

Nilaksh Agarwal, 2015PH10813

February 25, 2019

Objective

To modify the xv6 kernel to achieve the following functionalities:

1. System Calls
   (a) Print out a line for each system call invocation, and the number of times it has been called since trace has been on
   (b) Introduce a sys_toggle() system call to toggle the system trace on or off.
   (c) sys_add() System Call - To add a system call that takes two integer arguments and return their sum.
   (d) sys_ps() System Call - To add a system call that prints a list of all the current running processes.
2. Inter-Process Communication
   (a) Unicast communication - To send a message from one process to one other process.
   (b) Multicast communication - To send a message from one process to multiple other processes.
3. Distributed Algorithm
   (a) To compute the total of an array using unicast method and a coordinator process.
   (b) To compute the variance of the array using multicast method.

Procedure

Each part is implemented in the order mentioned above.

Part 1(a) - System Trace

In order to print out the system trace along with it's count, the following array is introduced in *syscall.c*

```
int calls_count[syscall_len]; //maintains a count of syscall
```

The calls_count array keeps the count of the number of times each system call has been called since the trace was toggled on

The syscall() function in *syscall.c* was modified as follows to keep track of the system call count. Below is the syscall() function from *syscall.c*, with the modifications mentioned as "added".

```
void
syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  num = curproc->tf->eax;
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    if(toggle_on == 1) //added
    {
          calls_count[num]++; //added
    }
    curproc->tf->eax = syscalls[num]();
  } else {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

(Note that toggle_on is a part of Part 1(b) and will be defined later).

The *sys_print_count* call has been added for the same, in the following manner

1. Add the following in *syscall.c*.

   ```
   extern int sys_print_count(void);
   ```

2. Add the following in *syscall.h*.

   ```
   #define SYS_print_count 22
   ```

3. Add the following to the array of functions in *syscall.c*.

   ```
   [SYS_print_count] sys_print_count,
   ```

4. Now let's define the sys_print_count system call. This definition will be given in *sysproc.c*. Add the following snippet to this file.

```
char *calls_name[] = {"sys_call0", "sys_fork", "sys_exit",
"sys_wait", "sys_pipe", "sys_read", "sys_kill", "sys_exec" ,
"sys_fstat", "sys_chdir", "sys_dup","sys_getpid","sys_sbrk",
"sys_sleep", "sys_uptime","sys_open""sys_write","sys_mknod",
"sys_unlink", "sys_link", "sys_mkdir", "sys_close",
"sys_print_count", "sys_toggle", "sys_add", "sys_ps",
"sys_send", "sys_recv", "sys_send_multi"};

extern int calls_count[syscall_len];

int
sys_print_count(void)
{
  for(int i=1; i<syscall_len; ++i)
  {
    if(calls_count[i]>0)
    {
      cprintf("%s %d\n", calls_name[i], calls_count[i]);
    }
  }
  return 0;
}
```

This function prints the system trace, being maintained by the array defined inside the *syscall.c* file.

5. Add the following in *usys.S*.

```
SYSCALL(print_count)
```

6. Finally add the following in *user.h*

```
int print_count(void);
```

Part 1(b) - Toggle Trace

For this, we have to declare and define the sys_toggle system call. toggle() will be the user function that will call the system call. This is done in a similar manner, so I will only show the *sysproc.c* changed henceforth. The proceedure to add a new system call remains unchanged.
Add the following in *sysproc.c*.

```
int toggle_on = 0;

int sys_toggle(void)
{
if(toggle_on == 0)
{
    for(int i=0; i<syscall_len;++i)
    {
```

```
            calls_count[i] = 0;
        }
        toggle_on = 1;
    }
    else
    {
        toggle_on = 0;
    }
    return 0;
    }
```

Part 1(c) - System Add

For this, we have to declare and define the sys_add system call. add() will be the user function that will call the system call. This is done as follows.
Add the following in *sysproc.c*.

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

Part 1(d) - Print Process List

For this, we have to declare and define the sys_ps system call. ps() will be the user function that will call the system call. This is done as follows.

1. Let's define the sys_ps system call. The definition will be given in *sysproc.c*. Add the following snippet to this file.

```
extern void print_pid(void);

int
sys_ps(void)
{
    print_pid();
    return 0;
}
```

2. The sys_ps system call used a function print_pid() which does the main job of printing the process name and id. The definition will be given in *proc.c*. Add the following snippet to this file.

```
void
print_pid(void)
{
  struct proc *x;
  for(x = ptable.proc; x < &ptable.proc[NPROC]; x++)
  {
    if(x->state == RUNNING || x->state == SLEEPING ||
```

```
    x->state == RUNNABLE)
    {
      cprintf("pid:%d name:%s\n",x->pid, x->name);
    }
  }
}
```

Part 2(a) - Unicast

In this section, we impliment the unicast method for Inter-Process Communication. For this we have two system calls *sys_send()* and *sys_recv()*

1. First, we define a few arrays to store the message and it's sender and receivers. These are defined in *sysproc.c*

```
    int send_arr[buf_len];
    int recv_arr[buf_len];
    char msg_arr[buf_len][MSGSIZE];
```

2. Next, we define the sys_send() function, which takes 2 integer and one char * argument, and stores the message in the buffer. Here the lock is used to ensure no other send/recv call can access the buffers during this time.

```
        int
sys_send(int sender_pid, int rec_pid, void* msg)
{
  while(lock > 0)
  {

  }
  lock = 1;
  char *message = (char *)msg;
  argint(0, &sender_pid);
  argint(1, &rec_pid);
  argptr(2, &message, MSGSIZE);
  int loc = -1;
  for(int i=0; i<buf_len; ++i)
  {
    if(send_arr[i] == 0)
    {
      loc = i;
      break;
    }
  }
  if(loc<0)
  {
    lock = 0;
    return -1;
  }
  for(int i=0; i<MSGSIZE; ++i) {
    if(message[i] == '\0') {
      msg_arr[loc][i] = '\0';
      break;
    }
```

```
    msg_arr[loc][i] = message[i];
  }
  send_arr[loc] = sender_pid;
  recv_arr[loc] = rec_pid;
  lock = 0;
  return 0;
}
```

3. Now, we implement the sys_recv() call, which takes one char * argument and writes tha message to this location. Locks are implimented in a similar fashion. This is done using a blocking system call, so the receiving process will keep waiting for the message until it recieves it.
   This is done by continuously searching the buffer until a message appears for the calling process.

```
        int
sys_recv(void *msg)
{
  char *message = (char *)msg;
  argptr(0, &message, MSGSIZE);
  int loc = -1;
  int my_id = sys_getpid();
  //cprintf("%s %d\n", "RECIVING PROCESS", my_id);
  while(loc < 0) {
    for(int i=0; i<buf_len; ++i)
    {
      if(recv_arr[i] == my_id) {
        loc = i;
        break;
      }
    }
  }
  if(loc<0) {
    lock = 0;
    return -1;
  }
    while(lock > 0)
  {

  }
  lock = 1;
  for(int i=0; i<MSGSIZE; ++i)
  {
    if(msg_arr[loc][i] == '\0')
    {
      message[i] = '\0';
      break;
    }
     message[i] = msg_arr[loc][i];
  }
  send_arr[loc] = 0;
  recv_arr[loc] = 0;
  lock = 0;
  return 0;
}
```

Part 2(b) - Multicast

Here, we define a new system call sys_send_multi() which takes one integer argument of the sender pid, one int array of the reciver(s) pid, the char * message, and the numbers of receivers. This is implemented using an interrupt method, which means the receiving process will not wait for the message, rather a process interrupt will be executed once the messages are sent, to notify the receiving processes of the same.

```
int
sys_send_multi(int sender_pid, void *rec_pid, void *msg,
int len)
{
  char *message = (char *)msg;
  argint(0, &sender_pid);
  argptr(2, &message, MSGSIZE);
  argint(3, &len);
  int* rec_multi = (int *)rec_pid;
  argintptr(1, &rec_multi,len);

  while(lock > 0) {
  }
  for(int i=0; i<len; ++i) {
    int loc = -1;
    for(int j=0; j<buf_len; ++j) {
      if(send_arr[j] == 0) {
        loc = j;
        break; } }

    if(loc<0) {
      lock = 0;
      return -1; }

    send_arr[loc] = sender_pid;
    recv_arr[loc] = rec_multi[i];
    for(int j=0; j<MSGSIZE; ++j) {
      if(*(message + j) == '\0') {
        msg_arr[loc][j] = '\0';
        break; }
      else {
        msg_arr[loc][j] = *(message + j);
      } } }
  lock = 0;
  return 0;
}
```

Here, the interrupt is done using a signal handler which is defined inside the process table in *proc.c*. This will save the state of the running process and execute the interrupt function called using the interrupt signal.

Part 3 - Distributed Algorithm

Here, the changes are made in *assig1_8.c* to implement the distributed algorithm to compute the sum of a 1000-element array. This program has two parts, one to compute the sum and another to compute the variance.

For the sum, each child process computes the sum of it's part of the array and unicasts this message back to the parent. The parent then adds these up to procure the final sum of the array.

For the variance, the parent uses the sum to compute the mean, and then multicasts this mean back to each child process. The child processes then compute the variance and unicast it back to the parent process. Who computes the final variance for this array.

```
//----FILL THE CODE HERE for unicast sum and multicast variance
  int parent_id = getpid();
  int child_id[8];

  for(int i=0; i<process_count; ++i){
    child_id[i] = fork();
    if(child_id[i] == 0){
      int sum = 0;
      int start_loc = (size/process_count) * i;
      int end_loc = (size/process_count) * (i+1);
      if(i == process_count - 1){
        end_loc = size;
      }
      for(int j = start_loc; j<end_loc; ++j) {
        sum = sum + arr[j];
      }
      char *msg = (char *)malloc(MSGSIZE);
      msg_prep(sum, msg, 0);
      send(getpid(), parent_id, msg);

      if(type == 0) {
        exit();
      }
      recv(msg);
      int avg = 0;
      avg = msg_prep(avg, msg, 1);
      int sigma = 0;
      for(int j = start_loc; j<end_loc; ++j) {
        sigma = sigma + (arr[j]-avg)*(arr[j]-avg);
      }
      msg_prep(sigma, msg, 0);
      send(getpid(), parent_id, msg);
      exit();
    }
  }
  for(int i=0; i<process_count; ++i) {
    char *msg = (char *)malloc(MSGSIZE);
    recv(msg);
    int sum = 0;
    sum = msg_prep(sum, msg, 1);
    tot_sum = tot_sum + sum;
  }
  if(type == 1) {
```

```
    int avg = tot_sum/size;
    char *msg1 = (char *)malloc(MSGSIZE);
    msg_prep(avg, msg1, 0);

    send_mult(parent_id, child_id, msg1, process_count);
    for(int i=0; i<process_count; ++i) {
      char *msg = (char *)malloc(MSGSIZE);
      recv(msg);
      int sigma = 0;
      sigma = msg_prep(sigma, msg, 1);
      variance = variance + sigma;
    }

    variance = variance/size;
  }
//------------------
```

This program also uses a helper function msg_prep which basically processes an integer input to a char * message, or vice versa, depending on the mode.

```
int
msg_prep(int value, char *msg, int type) {
//Function to process the int and convert it to a string
  char *message = (char *)msg;
  int return_val = -1;
  if(type == 0) {
    int temp = value;
    int index = 0;
    while(temp > 0) {
      *(message + index) = '0' + temp%10;
      temp = temp/10;
      index = index + 1;
    }
    *(message+ index) = '\0';
    return_val = 0;
  }
  else {
    value = 0;
    int index = 0;
    while(*(message + index) != '\0') {
      index = index + 1;
    }
    index = index - 1;
    while(index >= 0) {
      value = value * 10;
      value = value + (*(message + index) - '0');
      index = index - 1;
      }
    return_val = value;
    }
return return_val;
}
```

Appendix

This part contains some small additions to some files for defining constants, some small helper functions etc.

1. Added to *syscall.c* to pass int * pointers to system calls.

```
int
argintptr(int n, int **pp, int size)
{
  int i;
  struct proc *curproc = myproc();
  if(argint(n, &i) < 0)
    return -1;
  if(size < 0 || (uint)i >= curproc->sz || (uint)i+size > curproc->
      sz)
    return -1;
  *pp = (int*)i;
  return 0;
  }
```

2. All additions to *syscall.h*, defining the added system calls.

```
#define SYS_print_count 22
#define SYS_toggle 23
#define SYS_add 24
#define SYS_ps 25
#define SYS_send 26
#define SYS_recv 27
#define SYS_send_multi 28
```

3. Added to *defs.h* to include *argintptr()*

```
//line 154
int argintptr(int, int**, int);
```

4. All additions to *syscall.c*, for adding new system calls.

```
extern int sys_print_count(void);
extern int sys_toggle(void);
extern int sys_add(void);
extern int sys_ps(void);
extern int sys_send(void);
extern int sys_recv(void);
extern int sys_send_multi(void);

//inside the array
[SYS_print_count] sys_print_count,
[SYS_toggle] sys_toggle,
[SYS_add] sys_add,
[SYS_ps] sys_ps,
[SYS_send] sys_send,
[SYS_recv] sys_recv,
[SYS_send_multi] sys_send_multi,
};
```

5. Definitions in *types.h* defining some constants.

```
#define MSGSIZE 8
#define syscall_len 29
#define process_count 7
#define buf_len 20
```

6. Added a wait() to *assig1_7.c* to prevent interleaving of print statements, leading to errors with the outputs. Line 25

```
send(getpid(),cid,msg_child);
wait(); //added
printf(1,"1 PARENT: msg sent is: %s \n", msg_child );
```

7. Added the assig1_*.c files to the Makefile.

```
//line 184 (in UPROGS=\)
        _assig1_1\
        _assig1_2\
        _assig1_3\
        _assig1_4\
        _assig1_5\
        _assig1_6\
        _assig1_7\
        _assig1_8\
-----------------
\\line 193 (fs.img:)
fs.img: mkfs README arr $(UPROGS)
        ./mkfs fs.img README arr $(UPROGS)
-----------------
//line 258 (in EXTRA=\)
        _assig1_1.c _assig1_2.c _assig1_3.c _assig1_4.c _assig1_5.c
            _assig1_6.c _assig1_7.c _assig1_8.c\
```

8. All additions to *user.h*, for adding new system calls.

```
//under "system calls"
int print_count(void);
int toggle(void);
int add(int, int);
int ps(void);
int send(int, int, void *);
int recv(void *);
int send_multi(int, int *, void *, int);
```

9. All additions to *usys.S*, for adding new system calls.

```
SYSCALL(print_count)
SYSCALL(toggle)
SYSCALL(add)
SYSCALL(ps)
SYSCALL(send)
SYSCALL(recv)
SYSCALL(send_multi)
```

Outputs

Contains the obtained outputs for all parts of the check scripts.

1. Part 1

```
sys_fork 1
sys_print_count 1
sys_write 18
```

2. Part 2

```
sys_close 1
sys_open 1
sys_print_count 1
```

3. Part 3

```
sum of 2 and 3 is: 5
```

4. Part 4

```
sum of 10 and -6 is: 4
```

5. Part 5

```
pid:1 name:init
pid:2 name:sh
pid:3 name:assig1_5
```

6. Part 6

```
pid:1 name:init
pid:1 name:init
pid:2 name:sh
pid:2 name:sh
pid:3 name:assig1_6
pid:3 name:assig1_6
pid:4 name:assig1_6
pid:4 name:assig1_6
```

7. Part 7

```
1 PARENT: msg sent is: P
2 CHILD: msg recv is: P
```

8. Part 8

```
Sum of array for file arr is 4545
```

9. Part 9

```
Variance of array for file arr is 8
```

```
182376 bytes (182 kB, 178 KiB) copied, 0.00138058 s, 132 MB/s
Running..1
Running..2
Running..3
Running..4
Running..5
Running..6
Running..7
Running..8 (this will take 10 seconds)
Running..9 (this will take 10 seconds)
Test #1: PASS
Test #2: PASS
Test #3: PASS
Test #4: PASS
Test #5: PASS
Test #6: PASS
Test #7: PASS
Test #8: PASS
Test #9: PASS
9 test cases passed
```