COL 331: Operating Systems

Assignment 1 -Easy

KARAN KUMAR 2019PH10633

1. **Adding system calls**- We need to modify these 5 files – syscall.c, syscall.h, sysproc.c, usys.S, user.h

In syscall.c file, we add the function pointer for all system call functions we want to define as well as the function prototypes.

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

In syscall.h, we define the number of system call that is to be used for indexing in the array of function pointers.

```
[SYS_getyear] sys_getyear,
[SYS_add] sys_add,
[SYS_toggle] sys_toggle,
[SYS_ps] sys_ps,
[SYS_print_count] sys_print_
[SYS_send] sys_send,
[SYS_recv] sys_recv,
```

In sysproc.c we implement our system calls.

```
int
sys_add(void)
{
  int a, b;

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

  return a + b;

}
```

In usys.S, we add the interface for use program to able to call this system call.

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

In user.h, we add the function prototype which user program will call.

```
int getyear(void);
int add(int, int);
int toggle(void);
int ps(void);
int print_count(void);
int send(int, int, void*);
int recv(void*);
```

And add the test program in the makefile in the user programs section.

For sys_toggle(), I keep a variable and toggles it value between 0 and 1. 0 indicates TRACE_OFF and 1 indicates TRACE_ON. When we transition from 1 to 0, I remove all the previous count of system calls.

For sys_print_count(), whenever syscall() function in syscall.c is called (indicating there has been a system call) and the TRACE_ON state is present, I store count of a particular system call in an integer array and display the count of all system calls when sys_print_count is called.

For sys_add(), I simply take the two argument values using argint() and then sum them and return the result.

For sys_ps(), I write a helper function named proclist() in proc.c where I access the processes through ptable.proc and return all those who are not “UNUSED”.

For all the system calls above, I have made user programs to test them namely user_toggle.c, print_count.c, add.c and ps.c respectively.

2. **Inter-Process Communication** –

UNICAST – For each process, I make a buffer, where the messages sent to it can be stored, MAX_MSG denotes the size of the buffer and MSG_SIZE denotes the size of the message. First and Last arrays denote the first unread and last unread + 1 positions in the buffer corresponding to each process. Size denotes the number of unread messages for each process.

```
#define MSG_SIZE 8
#define MAX_MSG 50

char buffer[NPROC][MAX_MSG][8];
int first[NPROC] = {0};
int last[NPROC] = {0};
int size[NPROC] = {0};
```

sys_send() –

```
int
sys_send(void)
{
  int s_id, r_id;
  char *message;

  if(argstr(2, &message) < 0) return -1;
  if(argint(0, &s_id) < 0) return -1;
  if(argint(1, &r_id) < 0) return -1;

  // cprintf("%d\n", message);

  send_message(s_id, r_id, message);
  return 0;
}
```

In the definition of sys_send(), I first take sender process id, receiver process id and the message from arguments using functions argint and argstr. And the function send_message (defined in proc.c) takes these values as arguments and stores the message in the buffer of the receiver process id.

Definition of send_message() -

```
void send_message(int s_id, int r_id, char *msg){

  if(size[r_id] >= MAX_MSG){
    panic("[!] Buffer full!");
    return;
  }

  int i;
  //acquire(&ptable.lock);
  for(i=0; i<MSG_SIZE; i++){
    buffer[r_id][last[r_id]][i] = *(msg+i);
  }
  //cprintf("%s\n", buffer[r_id][last[r_id]]);
  last[r_id]++;
  size[r_id]++;
  //release(&ptable.lock);

  struct proc *p;
  p = &ptable.proc[r_id];
  if(p->state == UNUSED){
    acquire(&ptable.lock);
    wakeup1(&p);
    release(&ptable.lock);
  }

}
```

sys_recv() –

```
int sys_recv(void)
{
  char *message;

  if(argstr(0, &message) < 0) return -1;

  return receive_message(message);
  //return 0;
}
```

receive_message (defined in proc.c) retrives the message from the buffer of the receving process and returns 0 on successful retrieval and -1 on some error.

Definition of receive_message() –

```
int receive_message(char *msg){

  int id = myproc()->pid;
  int b;

  //cprintf("messages received %d\n", size[id]);
  if(size[id] == 0) return -1;
  // If no messages in the buffer, put the process

  // if(size[id] == 0){
  //   struct proc *p;
  //   p = &ptable.proc[id];

  //   acquire(&ptable.lock);
  //   sleep(&p, &ptable.lock);
  //   release(&ptable.lock);
  // }

  for(b=0; b<MSG_SIZE; b++){
    *(msg + b) = (buffer[id][first[id]][b]);
  }

  size[id] -= 1;
  first[id] += 1;
  return 0;
```

For multi-casting – I use sys_send_multi() and sys_recv() systems calls. The idea is the same as sys_send() where sys_send_multi adds the message to the buffer of multiple receiver processes and the recevier processes reads the messages in their buffer using sys_recv system call.

sys_send_multi() – Initially, we take the value of the sender process id, multiple receiver process ids and message to be send and pass it on to the function send_to_multi() (defined in proc.c)

```
int sys_send_multi(void){
  int s_id;
  int* r_ids;
  char *r_ids_char;
  char *message;
  int len = 8;

  if(argint(0, &s_id) < 0) return -1;

  //if(argptr2(1, &r_ids, length_of_ptr*sizeof(int)) < 0) return -1;
  if(argptr(1, &r_ids_char, len*sizeof(int)) < 0) return -1;
  r_ids = (int *)r_ids_char;

  if(argstr(2, &message) < 0) return -1;

  send_to_multi(s_id, r_ids, message);
  return 1;
}
```

Definition of send_to_multi() – Stores the message in the buffer of the receiver processes.

```
void send_to_multi(int s_id, int* r_ids, char *msg){

  // put all into buffer
  int i;
  int l = 8;
  for(i=0; i<l; i++){
    int r_id = *(r_ids + i);

    if(r_id < 0) continue;

    if(size[r_id] >= MAX_MSG){
      panic("[!] Buffer full!");
      return;
    }

    int j;
    for(j=0; j<MSG_SIZE; j++){
      buffer[r_id][last[r_id]][j] = *(msg+j);
    }

    last[r_id] += 1;
    size[r_id] += 1;
  }


}
```

And the receiver processes read using the sys_recv system call from their buffers.

3. Distributed Algorithm –

How are 8 processes created –

```
int n1 = fork();
int n2 = fork();
int n3 = fork();
```

Using 3 forks back to back, I create 8 processes running in parallel. First fork makes 2 processes (a child and a parent), second fork creates a child for each of the parent and child. And hence, 3 forks gives us 8 processes in total.

These are 8 worker processes that are needed and I assign one of the worker process (the original process) as the coordinator whose task is to coordinate among the rest 7 to compute the sum and variance as well as do its computation part as well.

ALGORITHM WORKS LIKE THIS (Unicast and Multicast) –

1. Firstly all the 8 processes compute the sum of 1000/8 = 125 elements (partial_sum)
2. The worker processes send their partial sum to the coordinator process using sys_send system call (process id of the coordinator process is known to all)
3. The coordinator adds the messages received (using sys_recv system call) as well its own partial sum to compute the total sum as well as the average.

4. Next, the worker processes send their process ids using sys_send to the coordinator process (so that the coordinator can send messages back to them)
5. The coordinator sends the average of all the elements to the worker processes using sys_send_multi.
6. Next, all the processes calculate their partial sum of squares and send them to the coordinator.
7. The coordinator receives the messages and computes the variance.

Code snippet of the worker process –

```
int sum = 0;
for(int i=875; i<1000; i++){
    sum += arr[i];
}

char *msg = (char *)malloc(MSGSIZE);

int process_id = getpid();
int dig1 = tot_digits(sum);
int_to_string(msg, sum, dig1, 'a');
//printf(1,"1 PARENT: msg sent is: %s \n", msg);
send(process_id,leader_pid, msg);

int dig2 = tot_digits(process_id);
int_to_string(msg, process_id, dig2, 'b');
send(process_id, leader_pid, msg);

int stat=-1;
while(stat==-1){
    stat = recv(msg);
}
float avg_global = *((float *)msg);

int i;
float part_var = 0.0;
for(i = 875; i < 1000; i++){
    float d = (avg_global - (float)arr[i]);
    part_var +=  d * d;
}

for(i = 0; i < 4; i++) msg[i] = *((char*)&part_var + i);
send(process_id, leader_pid, msg);

free(msg);

//wait();
exit();
```

Code snippet of the coordinator –

```
for(int i=0; i<125; i++){
    sum += arr[i];
}
tot_sum += sum;
mean += sum;
//printf(1, "sum is %d\n", sum);

char *msg = (char *)malloc(MSGSIZE);
int *r_ind = (int *)malloc(8 * sizeof(int));

r_ind[7] = -10; // invalid pid deliberately
int ind = 0;

for(int i=0; i<14; i++){

    int stat=-1;
    while(stat==-1){
        //printf(1, "in loop");
        stat = recv(msg);
    }
    int p = 0;
    //printf(1, "%d\n", p);
    int val = string_to_int(msg, &p);
    //printf(1, "%d\n", p);


    if(p == 'a' - '0'){
        //printf(1, "value %d\n", val);
        int partial_sum = val;
        tot_sum += partial_sum;
        mean += partial_sum;
    }
    else if(p == 'b' - '0'){
        //printf(1, "pid value %d\n", val);
        r_ind[ind] = val;
        ind++;
    }
    //printf(1,"2 CHILD: msg marecv is: %d \n", partial_s

}
```

```
mean /= 1000;

int i;
for(i = 0; i < 4; i++) msg[i] = *((char *)&mean + i);

send_multi(leader_pid, r_ind, msg);


for(i = 0; i < 125; i++){
    float d = (mean - (float)arr[i]);
    variance +=  d * d;
}

for(int i=0; i<7; i++){
    int stat=-1;
    while(stat==-1){
        stat = recv(msg);
    }
    variance += *((float *)msg);
}

variance /= (float)1000;
//printf(1, "var is %d\n", (int)variance);
free(r_ind);
free(msg);
```