Assignment 1

Shreyansh Singh
2020CS10385

February 15, 2023

1 System calls

1.1 Trace

To keep a record of the trace on and off, I have declared a variable "toggle" in syscall.c which is 0 when the trace if off and one when the trace is on. The system call toggle sets the toggle variable one if it is zero else set it to zero and reset the count of all system calls.
Code:

```
int sys_toggle(void)
{
  int i;
  if(toggle==0){
    toggle = 1;
  }
  else{
    toggle = 0;
    for(i=0; i<21; i++){
      syscall_c[i] = 0;
    }
  }

  return 0;
}
```

1.2 Print Count

For print_count system call, I have declared two arrays in syscall.c, one of which stores the name of all the system calls and the other is used to store the count of all system calls. For keeping all the system calls in ascending order, I have an array that maps the system calls according to ascending order. If the toggle is set to one, then if any system call is done then the counter corresponding to that system call is incremented. In the system call function, all the system calls are printed whose count is more than zero. Code:

```
int sys_print_count(void)
{
  int i;
  for(i=0; i<21; i++){
    if(syscall_c[map_call[i]-1] != 0){
      cprintf("%s %d\n", syscall_n[i], syscall_c[map_call[i]-1]);
    }
  }
  return 0;
}
```

1.3 Add

This system call takes two integer arguments and returns their sum. Code:

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

1.4 Process List

This system call calls a function get_ps in proc.c, which iterates through the ptable and prints a list of all the currently running processes. ptable is locked during the execution of this function. Function code:

```
void get_ps(void)
{
  struct proc *p;

  acquire(&ptable.lock);
  for(p=ptable.proc; p < &ptable.proc[NPROC]; p++)
  {
    if(p->pid==0)
      break;
    cprintf("pid:%d name:%s\n", p->pid, p->name);
  }

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

2 IPC

For the Inter-Process Communications, I have used a global array in proc.c. The sender process adds the message to the buffer of the receiver process and the receiver process checks if there is any message in its buffer or not. The sender process wakes up the receiver process if it is sleeping. The receiver process reads the messages in the reverse order in which they were sent, i.e., if there are two unread messages then it will read the latest message first.
The multicast essentially calls the unicast multiple times, i.e., sending messages to receiver pids sequentially.
Sender function code:

```
// if the receiver buffer is full return -1
if(num[rec_pid] >= 64 && num[id]>-1){
  return -1;
}

for(int i=0; i<8; i++){
  msgs[rec_pid][num[rec_pid]][i] = msg[i];
}
num[rec_pid] += 1;

struct proc *p;
p = &ptable.proc[rec_pid];

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

Receiver code:

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

// when no message in buffer
if(num[id] == 0){
  return -1;
}

for(int i=0; i<8; i++){
  msg[i] = (msgs[id][num[id]-1][i]);
}

num[id] -= 1;
```

To make the receive call blocking, in the system call, receive function is called till 0 is returned. Code:

```
int stat=-1;
while(stat==-1){
  stat = receive_message(msg);
}
```

In the sys_send_multi function to send an integer array to the system call, I have created a new function "argptrint" (in syscall.c) which is similar to argptr (argptr is for char array and argptrint is for integer array). argptrint function:

```
int
argptrint(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;
}
```

Multi-send system call code:

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

  argint(0, &sender_pid);
  argptrint(1, &rec_pids, 8);
  argstr(2, &msg);

  for(int i=0; i<8; i++){
    if(rec_pids[i]==-1) break;
    send_message(sender_pid, rec_pids[i], msg);
  }

  return 1;
}
```

3 Distributed Algorithm

The initial thread becomes the coordinator thread, and it forks eight times creating eight child processes. For phase one, the array is equally divided into the child processes, and they send their sum to the parent process. The corresponding code is:

```
int sum=0;
for(int i=(tid-1); i<size; i+=THREAD_NUM)
    sum += arr[i];

send(getpid(),parent,&sum);
```

The coordinator thread receives the partial array sum from all the child processes and accumulates them to create the total sum of the array. The corresponding code is:

```
for(int i=0; i<THREAD_NUM; i++){
    int temp = 0;
    recv(&temp);
    tot_sum += temp;
}
```

Now, if the type is 0, all the processes are killed.
If the type is 1, all the child processes receive the mean from the coordinator thread and then calculate the variance of the array elements allocated to it. Each child process then sends the variance back to the coordinator process. The corresponding code is:

```
float mean = 0;
int q=0;
recv(&q);
mean = (1.0*q)/size;

float sum=0;
for(int i=(tid-1); i<size; i+=THREAD_NUM){
    sum += ((float)arr[i]-mean)*((float)arr[i]-mean);
}

int t=(sum*10);
send(getpid(),parent,&t);

exit();
```

If the type is 1, the coordinator thread broadcasts the mean to all the child processes, and then waits for all of them to exit. Then it receives partial variance from all the child processes to calculate the final variance. The corresponding code is:

```
float mean = (1.0*tot_sum)/size;
int t=(mean*1000);
send_multi(parent, thr_id, &t);

for (int i = 0; i < THREAD_NUM; i++)
    wait();

for(int i=0; i<THREAD_NUM; i++){
    int temp = 0;
    recv(&temp);
    variance += (1.0*temp)/10;
}
```

```
variance /= size;
print_variance(variance);
```

To send float values between the processes I have converted them to int by multiplying them, then diving them on the receiver side.