COL331-Assignment 1

Darshan Rakhewar - 2020CS10340

1 Installing and Testing xv6:

Installation done.

2 System Calls:

For Adding custom syscalls, modifications were mainly made to the files 1)syscall.c , 2)syscall.h, 3)sysproc.c, 4)user.h, 5)usys.S.

System call trace:

```c
// syscall_trace.h > ...
#include "types.h"
#include "stat.h"

extern char* namesyscalls[];

extern enum trace_state {TRACE_OFF, TRACE_ON} trace;

extern int numsyscalls;

extern int countsyscalls[];

extern int indexing[];
```

The kernel state trace_state was added to the kernel. The above variables are used to store the number of system calls made for various system calls and the state of the kernel. Additionally, the array “indexing” was used to print the number of calls for different syscalls in the alphabetical order.

```c
if(trace == TRACE_ON && num!=22 && num!=23){
  countsyscalls[num-1]++;
}
```

This modification was made in syscall function to count the number of calls.

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

Toggling of the trace is handled by this function in sysproc.c. Here, we also reinitialise the array values to ‘0’ on toggling trace on.

```
int
sys_print_count(void)
{
  for(int i = 0; i < numsyscalls; i++){
    if(countsyscalls[indexing[i]] > 0 && indexing[i]+1!=22 && indexing[i]+1!=23)
      cprintf("%s %d\n", namesyscalls[indexing[i]], countsyscalls[indexing[i]]);
  }
  return 0;
}
```

This function takes care of printing the number of the various system calls that are made. It is worth to not that this function uses indexing array to print them insorted order.

sys_add():

```
int
sys_add(int a, int b)
{
  if(argint(0, &a)!=0 || argint(1, &b) !=0){
    return -1;
  }
  return a+b;
}
```

Process Status:

A function is defined in sysproc.c which in turn calls the function process_status() (defined in proc.c)

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

```c
void
process_status(void)
{
  struct proc *p;
  acquire(&ptable.lock);
  for(p=ptable.proc; p<&ptable.proc[NPROC]; p++){
    if(p->state != UNUSED)cprintf("pid:%d name:%s\n", p->pid, p->name);
  }
  release(&ptable.lock);
}
```

3 Inter-Process Communication:

Unicast:

```c
int
sys_send(int sender_pid, int rec_pid, void *msg)
{
  char* c;
  if(argint(0, &sender_pid) < 0)return -1;

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

  if(argptr(2, &c, message_size) < 0)
    return -1;

  acquire(&shared.lock);
  acquire(&lock);

  for(int i = 0; i < buffercount; i++){
    if(shared.destpid[i] < 1){

      shared.srcpid[i] = sender_pid;
      shared.destpid[i] = rec_pid;
      memmove(shared.buffers[i], c, message_size);
      if(shared.iswaiting[rec_pid] == 1){

        shared.iswaiting[rec_pid] = 0;
        // Wake up the reciever
        wakeup((void*)rec_pid);
      }
      release(&lock);
      release(&shared.lock);
      return 0;
    }
  }
  release(&shared.lock);
  release(&lock);
  return -1;
}
```

```c
int
sys_recv(void *msg)
{
  int id = myproc()->pid;
  int i = 0;
  char* c;
  if(argptr(0, &c, message_size)!=0)return -1;
  while(1){
    acquire(&shared.lock);
    acquire(&lock);
    for(i =0; i<buffercount; i=i+1){
      if(shared.destpid[i] == id){
        memmove(c, shared.buffers[i], message_size);
        shared.destpid[i] = -1; //this means that
        release(&lock);
        release(&shared.lock);
        return 0;//message was found so return
      }
    }
    release(&lock);
    // Wait and sleep if msg not found yet
    shared.iswaiting[id] = 1;
    sleep((void*)id, &shared.lock);
    release(&shared.lock);
  }
  return 0;
}
```

```c
#define buffercount 1000
#define message_size 8
struct spinlock lock;
typedef struct{
  struct spinlock lock;
  char buffers[buffercount][8];
  int srcpid[buffercount];
  int destpid[buffercount];
  int iswaiting[NPROC];
} buffer_share;

buffer_share shared = {
  .buffers = { "        " },
  .srcpid = { 0 },
  .destpid = { 0 },
  .iswaiting = { 0 }
};
```

The buffer_share shared is the globally shared data block among different processes.
Buffers is the array of buffer which is used for Communication.
Srcpid and destpid represent the the source and destination of the i'th buffer. iswaiting[i] is set to 1 if i'th process is waiting for a message. These two new syscalls were implemented to send

and receive data. Locks are used to avoid any data race andthe code is self explanatory with the comments.

Multicast:

Multicast is simply implemented by using the syscall send over all the pid's in the receiver pid list. It is worth noting that if a receiving process does not find a received message, it has went to sleep, thus we have to wake it up in that case.

```
int
sys_send_multi(int sender_pid, int rec_pids[], void *msg) {
  char* c;
  int* recv_pids;
  if(argint(0, &sender_pid)!=0){
    cprintf("sender_pid error\n");
  }
  if(argptr(1, (char**)&recv_pids, 8)!=0){
    for(int i=0;i<8; i++){
      cprintf("reciever_pid error\n");
    }
  }
  if(argptr(2, &c, message_size)!=0){
    cprintf("message error\n");
  }

  acquire(&shared.lock);
  acquire(&lock);
  //cprintf("locks acquired for sending...\n");
  for (int j = 0; j < 8; j++) {
    if(recv_pids[j]==-1)continue;
    for(int i = 0; i < buffercount; i++){
      if(shared.destpid[i] <= 0){
        memmove(shared.buffers[i], c, message_size);
        shared.srcpid[i] = sender_pid;
        shared.destpid[i] = recv_pids[j];
        if(shared.iswaiting[recv_pids[j]] == 1){
          //cprintf("recvid %d was waiting\n", recv_pids[j]);
          shared.iswaiting[recv_pids[j]] = 0;
          // Wake up the reciever
          wakeup((void*)recv_pids[j]);
          //cprintf("recvid %d was woken up\n", recv_pids[j]);
        }
        break;
      }
    }
    //cprintf("messsage sent to rec %d  \n", j);
  }
  release(&lock);
  release(&shared.lock);
  return 0;
}
```

Terminal (Ctrl+`)

```
user_multicast 2 28 15744
console        3 29 0
$ user_multicast
IPC Test case
pid:1 name:init
pid:2 name:sh
pid:4 name:user_multicast
pid:5 name:user_multicast
pid:6 name:user_multicast
pid:7 name:user_multicast
pid:8 name:user_multicast
pid:9 name:user_multicast
pid:10 name:user_multicast
pid:11 name:user_multicast
pid:12 name:user_multicast
PARENT: sent message: 3CHILD: received message: 3
CHILD: received CHILD: received message: 3
CHILD: received message: 3
CHILD: received message: 3
CHILD: received message: 3
CHILD: received message: 3
message: 3
CHILD: received message: 3
$
```

Working user_multicast.

In the IPC, however, the implementation is not made using the interrupt handler. Here, the processes know what to expect from other processes. Thus, the processes sleep and wait until the necessary message arrives.

4 Distributed Algorithm:

Unicast:

In this case, 8 children processes are made for the one coordinator process as follows:

```
//----FILL THE CODE HERE for unicast sum
int cid[NUMPROCS];
int cnum;
int coordinator=getpid();
int ischild=0;

for(int i=0; i<NUMPROCS; i++){
    cnum=i;
    cid[i]=fork();
    if(cid[i]!=0);//printf(1, "Init %d\n", cid[i]);
    else {
        ischild=1;
        break;
    }
}
```

The array is divided into 8 chunks and thus cnum'th chunk of the array is summed by the cnum'th child process. These process then send this partial sum to the coordinator process which then sums these partial sums to get the total sum.

```
if(ischild==1){
    //printf(1, "Child proc %d\n", getpid());
    int partial=0;
    for(int i=cnum*size/NUMPROCS; i<(cnum+1)*size/NUMPROCS; i=i+1)partial+=arr[i];
    //printf(1, "Child proc %d prtial sum is %d\n", getpid(), partial);
    send(getpid(), coordinator, &partial);
```

```
int sum=0;
int rec_partial;
for(int i=0;i<NUMPROCS; i++){
    recv(&rec_partial);
    sum+=rec_partial;
    //printf(1, "Received %dth partial sum %d, current sum = %d\n", i, rec_partial, sum);
    tot_sum=sum;
```

Other modifications:

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#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
```

```
int numsyscalls = NELEM(namesyscalls);

int countsyscalls[NELEM(syscalls)] = { 0 };

int indexing[NELEM(syscalls)] ={ 23, 8, 20, 9, 6, 1, 0, 7, 10, 5, 18, 19, 16, 14, 3, 21, 24, 4, 26, 11, 25, 27, 12, 22, 17, 13, 2, 15};
```

```
    "sys_mkdir",
    "sys_close",
    "sys_print_count",
    "sys_toggle",
    "sys_add",
    "sys_ps",
    "sys_send",
    "sys_recv",
    "sys_send_multi"
};
```

```
int sleep(int);
int uptime(void);
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*);
```

```
[SYS_close]   sys_close,
[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,


};
```