COL331 A1 (Easy) REPORT

Ananya Mathur - 2020CS50416

February 2023

1 Installing and Testing xv6

I successfully managed to install xv6 and qemu on my Linux system using the instructions provided in the assignment pdf.

2 System Calls

Trace Mode : I have created a global variable trace for trace mode (0 = trace off , 1 = trace on) in syscall.c which is initialised to 0. (Tracing is off initially)

```
int trace=0;
```

Count Array : In proc.c, I have defined 3 arrays : a, b and count.

```
char *a[29]={"start","sys_add","sys_chdir","sys_close","sys_dup","sys_exec","sys_exit","
    sys_fork","sys_fstat","sys_getpid","sys_kill","sys_link","sys_mkdir","sys_mknod","
    sys_open","sys_pipe","sys_print_count","sys_ps","sys_read","sys_recv","sys_sbrk","
    sys_send","sys_send_multi","sys_sleep","sys_toggle","sys_unlink","sys_uptime","sys_wait",
    "sys_write"};

int b[29]={0, 22, 9, 21, 10, 7, 2, 1, 8, 11, 6, 19, 20, 17, 15, 4, 23, 24, 5, 26, 12, 25,
    28, 13, 27, 18, 14, 3, 16};

int count[29]={0};
```

a contains all system call names in alphabetical order, b contains their respective system call number (as defined in syscall.h) and count maintains a count of the number of times each system call is invoked by any process from the time the last toggle to trace on took place. If toggle is off, count array contains 0 at all indices. For any index i of the arrays, a[i] is the name of a system call, b[i] is its number (syscall.h) and count[b[i]] is the number of times it has been called.

1. **System call sys_toggle()**
   This system call toggles the trace mode.
   If the current value of trace is 1, it makes it 0 (and vice versa). Its implemenatation is present in syscall.c's void syscall(void) function which is invoked on every system call. If the toggle system call is invoked, the value of trace is changed to 1-trace. (trace also defined in syscall.c)
   Also, if trace mode is toggled to trace off, the count array is set to 0 for all system calls as a fresh count of system calls should start whenever trace mode is toggled to trace on next.

```
  if (num == SYS_toggle)
  {
    trace = 1 - trace;
```

```
4
5     if (trace == 0)
6     {
7       for (int i = 1; i < 29; i++)
8         count[i] = 0;
9     }
10  }
```

2. **System call sys_print_count()**
   This system call prints the number of times (only if non zero) each system call was invoked from the time the trace mode was set to trace on last along with the names of the system calls (in alphabetical order). Its implementation is present in proc.c's print_count function, which is invoked on every sys_print_count() system call. This function iterates through the count array and prints all system calls (name and count) with non zero count.

```
1 int sys_print_count(void)
2 {
3   return print_count();
4 }
```

```
1 int print_count(void)
2 {
3   for (int i = 1; i < 29; i++)
4   {
5     if (count[b[i]] > 0)
6       cprintf("%s %d\n", a[i], count[b[i]]);
7   }
8   return 1;
9 }
```

3. **System call sys_add()**
   This system call takes two integers as input using argint function and returns their sum.

```
1 int sys_add(void)
2 {
3   int n1, n2;
4   argint(0, &n1);
5   argint(1, &n2);
6   return n1 + n2;
7 }
```

4. **System call sys_ps()**
   This system call invokes the ps() function (defined in proc.c as it has to access ptable).The ps() function iterates through ptable and prints pid and name of all current processes whose state is not UNUSED.

```
1 int sys_ps(void)
2 {
3   return ps();
4 }
```

```
1 int ps(void)
2 {
3   struct proc *p;
4
```

```
  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);
  return 0;
}
```

3 Inter-Process Communication

Sender, Receiver and Message Arrays : In sysproc.c, I have defined 3 global arrays to store sender, receiver and message for any communication between two processes. For any index i, senderarray[i] stores the pid of sender process, recvrarray[i] stores the pid of the receiver process and message[i] stores the message sent. A 0 in the sender/receiver arrays denotes an empty spot. Size of these arrays (maximum number of yet to be received messages by all processes together) is set to buffersize (kept at 50).

```
int senderarray[buffersize] = {0};
int recvrarray[buffersize] = {0};
char messagearray[buffersize][8];
```

Busy Flag : I have defined an int variable busy in sysproc.c whose value indicates whether a process is currently sending a message (manipulating the 3 arrays) or not. (1 if sending, 0 otherwise)

```
int busy = 0;
```

UNICAST

1. **System call sys_send()**
   This system call sends a message from a single sender process to a single receiver process. If busy flag is 1, it waits (infinite while loop) till it becomes 0 and then goes on to find an empty spot in the arrays and replace it with sender_pid, rec_pid, msg (taken as arguments to the system call) respectively. The system call finally sets busy to 0 and returns 0 for successful send.

```
int sys_send(void)
{
  while (busy)
  {
    ;
  }

  busy = 1;
  int n1, n2;
  void *n3;

  argint(0, &n1);
  argint(1, &n2);
  argptr(2, (void *)&n3, 8);

  char *message = (char *)n3;

```

```
  for (int i = 0; i < buffersize; i++)
  {
    if (senderarray[i] == 0)
    {
      senderarray[i] = n1;
      for (int j = 0; j < 8; j++)
        messagearray[i][j] = message[j];
      recvrarray[i] = n2;
      busy = 0;
      return 0;
    }
  }
  return 1;
}
```

2. **System call sys_recv()**
   This system call is invoked by a process to receive a message sent by any sender process. It takes a void pointer as input and makes it point to the received message. This system call is blocking, which means that no other system call can be made while the sys_recv function is running. To ensure this the for loop to search for receiver pid in recvrarray is nested inside an infinite while loop which is broken only if the receiver's pid is found in recvrarray. Thus, till a message is not received the while loop is not exited. Upon succesful receive, 0 is returned.

```
int sys_recv(void)
{
  void *n1;
  argptr(0, (void *)&n1, 8);

  int recvrpid = myproc()->pid;
  char *message = (char *)n1;

  while (1)
  {

    for (int i = 0; i < buffersize; i++)
    {
      if (recvrarray[i] == recvrpid)
      {
        for (int j = 0; j < 8; j++)
          message[j] = messagearray[i][j];

        recvrarray[i] = 0;
        senderarray[i] = 0;

        return 0;
      }
    }
  }
}
```

MULTICAST

1. **System call sys_send_multi()** This system call is used by a sender process to send the same message to multiple receiver processes. If busy flag is 1, it waits (infinte while loop) till it becomes 0 and then goes on to find (one at a time) the required number of empty spots (equal to the

number of receivers) in the arrays and replace each with sender_pid, rec_pid (traversing through the rec_pids array alongside), msg respectively. Once the message is sent to all receivers (all 3 arrays updated for each receiver), busy is set to 0 and 0 is returned denoting successful send.

```
int sys_send_multi(void)
{
  while (busy)
  {
    ;
  }

  busy = 1;
  int n1;
  void *n2, *n3;

  argint(0, &n1);
  argptr(1, (void *)&n2, 50);
  argptr(2, (void *)&n3, 8);

  int *recpids = (int *)n2;
  char *message = (char *)n3;
  int k = 0;

  for (int i = 0; i < buffersize; i++)
  {
    if (senderarray[i] == 0)
    {
      senderarray[i] = n1;
      for (int j = 0; j < 8; j++)
        messagearray[i][j] = message[j];
      recvrarray[i] = recpids[k];
      k++;
    }
    if (recpids[k] <= 0)
    {
      busy = 0;
      return 0;
    }
  }
  return 1;
}
```

4 Distributed Algorithm

I have implemented my distributed algorithm in **assign1_8.c** using **unicast** and **multicast** implemented before.

1. I have defined NUMBER_OF_PROCESSES as a macro (set to 8) and declared an array child of the same size to store pids of all child processes.

```
#define NUMBER_OF_PROCESSES 8
```

```
int child[NUMBER_OF_PROCESSES];
```

2. Also, pid of the parent process is found and stored in parentpid using getpid() function.

```
int parentpid = getpid();
```

3. tot_sum and variance variables are initialised to 0.

```
int tot_sum = 0;
float variance = 0;
```

UNICAST version (sum) :

The parent process calls fork() system call NUMBER_OF_PROCESSES times. Now, the pid of the child process created is returned to the parent process which is stored in child[i].

```
for (int i = 0; i < NUMBER_OF_PROCESSES; i++)
  {
    child[i] = fork();
```

0 is returned to the child process, in which case begin and end indices of the child's section of the array are assigned to it. Sum of this section of the array is computed by the child process and then sent to the parent process using send function developed before. Sender pid is found by the child using getpid() function, receiver pid is known globally as parentpid and msg is intiallised as char * using malloc. The conv_int_to_string function (defined by me in assig1_8.c) is used to make msg point to a string holding the partial sum. In this way each child process uses send function to send its computed partial sum.

If type is 0 (unicast), the job of the child processes is over and they are made to exit.

```
if (child[i] == 0)
    {
      int begin = (size / NUMBER_OF_PROCESSES) * i;
      int end = (size / NUMBER_OF_PROCESSES) * (i + 1);

      int sum = 0;
      for (int j = begin; j < end; j++)
        sum += arr[j];

      int len = num_of_digits(sum);

      char *msg = (char *)malloc(len);
      conv_int_to_string(msg, sum, len);

      send(getpid(), parentpid, msg);

      memset(msg, 0, sizeof(msg));

      if (type == 0)
        exit();
```

The parent process (out of the above for loop since child[i]!=0 in any iteration for the parent process) goes on to receive each of the partial sums using recv() function, converts them to integer using conv_string_to_int function (defined by me in assig1_8.c) and adds them to tot_sum (initialised to 0).

```
for (int i = 0; i < NUMBER_OF_PROCESSES; i++)
  {
    char *msg = (char *)malloc(8);
    recv(msg);
    int val = conv_string_to_int(msg);
```

```
    tot_sum += val;
    memset(msg, 0, sizeof(msg));
  }

```

Finally, since type is 0, total sum is printed by the parent which then waits for each of the child processes to exit, and then itself exits too.

```
 if (type == 0)
  { // unicast sum
    printf(1, "Sum of array for file %s is %d\n", filename, tot_sum);
  }
```

```
for (int i = 0; i < NUMBER_OF_PROCESSES; i++)
    wait();

  exit();

```

MULTICAST version (variance) :

This is an extension to the above unicast mechanism. So, if type is not 0 (1 in this case) the child processes do not exit right after sending their partial sums.

The parent process after computing tot_sum finds the average by dividing tot_sum by size of array. This average (float) is converted to string using conv_float_to_string function (defined by me in assig1_8.c) and sent to each child process using multicast send function send_multi, which takes parentpid as sender id, child array as rec_pids[], converted char * as message.

```
if (type == 1)
  {

    float average = tot_sum / (float)size;
    char *msg = (char *)malloc(8);
    conv_float_to_string(average, msg);

    send_multi(parentpid, child, msg);
    memset(msg, 0, sizeof(msg));
```

Each child process goes ahead in the for loop (did not exit as type=1) and calls recv to receive the average sent by the parent. It then converts the received string to float using conv_string_to_float function (defined by me in assig1_8.c).

```
msg = (char *)malloc(8);
recv(msg);
float avg = conv_string_to_float(msg);
memset(msg, 0, sizeof(msg));
```

Further on, each child computes the sum of squares of difference between array elements in its section and the average in x. This sum stored in x is then converted to string and sent using send function to parent process as before (partial sum case). The child process then calls exit().

```
float x = 0;

      for (int j = begin; j < end; j++)
        x += (arr[j] - avg) * (arr[j] - avg);
```

```
      msg = (char *)malloc(8);
      conv_float_to_string(x, msg);
      send(getpid(), parentpid, msg);

      memset(msg, 0, sizeof(msg));
      exit();
```

The parent invokes recv for each child process (NUMBER_OF_PROCESSES times), converts the received sum to float and adds it to variance.

```
for (int i = 0; i < NUMBER_OF_PROCESSES; i++)
    {

      char *msg = (char *)malloc(8);
      recv(msg);
      float val = conv_string_to_float(msg);
      variance += val;

      memset(msg, 0, sizeof(msg));
    }
```

Finally, variance is divided by size of the array to compute the final answer for variance of the array.

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

This is then printed by the parent process, again by converting to string using conv_float_to_string function as there is no provision to print floating point values as such in xv6.

```
{ // multicast variance
    char *var = (char *)malloc(8);
    conv_float_to_string(variance, var);
    printf(1, "Variance of array for file %s is %s\n", filename, var);
    memset(var, 0, sizeof(var));
  }
```

After this, the parent waits for each child process to exit and finally exits itself too.

```
for (int i = 0; i < NUMBER_OF_PROCESSES; i++)
    wait();

  exit();
}
```

Note that the child processes only execute the code inside the for loop and exit once their job is done. The parent on the other hand after creating all child processes comes out of the for loop and performs its tasks.

5 Extras

Since the send (both unicast and multicast) and receive functions send messages as pointers to strings, it was necessary to create functions which interconvert between int/float and char *.
Thus, I created 4 helper functions in assig1_8.c to facilitate writing my distributed algorithm.

```
int conv_string_to_int(char *string)
{
  int i = 0, j = 0;
```

```
  while (string[j] != '\0')
  {
    i *= 10;
    i += string[j] - '0';
    j++;
  }
  return i;
}
```

```
float conv_string_to_float(char *string)
{
  int intpart = 0, floatpart = 0, j = 0;
  while (string[j] != '.')
  {
    intpart *= 10;
    intpart += string[j] - '0';
    j++;
  }
  j++;
  int ct = 0;
  while (string[j] != '\0')
  {
    floatpart *= 10;
    floatpart += string[j] - '0';
    j++;
    ct++;
  }
  float scale = 1.0;
  for (j = 0; j < ct; j++)
    scale *= 10;

  return intpart + floatpart / scale;
}
```

```
void conv_int_to_string(char *p, int n, int len)
{

  for (int i = 0; i < len; i++)
  {
    p[len - 1 - i] = n % 10 + '0';
    n /= 10;
  }
}
```

```
void conv_float_to_string(float n, char *p)
{
  int ipart = (int)n;
  float fpart = n - (float)ipart;

  int len = num_of_digits(ipart);

  conv_int_to_string(p, ipart, len);

  p[len] = '.';
  fpart = fpart * 100;
  conv_int_to_string(p + len + 1, (int)fpart, 2);
}
```

I also made a helper function to find out the number of digits in an integer in assign1_8.c, which helped me in my implementation of the above 4 functions.

```
int num_of_digits(int n)
{
  int ct = 0;
  while (n > 0)
  {
    ct++;
    n /= 10;
  }
  return ct;
}
```

Additionally, for every system call implementation some basic changes (other than the ones mentioned in section 2) in a few files had to be made namely :

1. **syscall.h :** Added macros for newly defined system calls.

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

2. **usys.S :** Added newly created system call names.

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

3. **user.h :** Added function definitions with arguments for each system call.

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

4. **defs.h :** For any new functions defined in proc.c, their definition had to be given in header file defs.h.

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