ELL 405 Operating Systems

Assignment Report

Assignment 1

Anurag Gupta 2020EE10583

IIT DELHI, 10th February 2024

Contents

1 System Calls

In total 7 new syscalls were implemented: `toggle, print_count, add, ps, send, recv, send_multi`
Adding a new system call is generally a 5-step process.

- Adding SYSCALL No in syscall.h: This associates every syscall with a number. This number is used to call the associated function. Moreover, while using `call syscall`, the syscall no. is store in the eax register of the current process.

```
// Assignment 1.2
#define SYS_toggle      22
#define SYS_print_count 23
#define SYS_add         24
#define SYS_ps          25
// Assignment 1.3
#define SYS_send        26
#define SYS_recv        27
#define SYS_send_multi  28
```

- Adding function pointers in syscall.c: A pointer to the function associated with every syscall is added in this file.

```
// Assignment 1.2
extern int sys_toggle(void);
extern int sys_print_count(void);
extern int sys_add(void);
extern int sys_ps(void);
// Assignment 1.3
extern int sys_send(void);
extern int sys_recv(void);
extern int sys_send_multi(void);
 ...
static int (*syscalls[])(void) = {
   ...
    [SYS_toggle]      sys_toggle,
    [SYS_print_count] sys_print_count,
    [SYS_add]         sys_add,
    [SYS_ps]          sys_ps,
    [SYS_send]        sys_send,
    [SYS_recv]        sys_recv,
    [SYS_send_multi]  sys_send_multi
}
```

- Function definition in sysproc.c: The function declared above is usually defined in sysproc.c

- Adding helper code in usys.S: A macro defined in usys.S is used to map syscalls with the number in eax register.

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

- Adding definition in user.h: A function corresponding to each syscall is created for the user to call in a user program.

```
// Assignment 1.2
int toggle(void);
int print_count(void);
int add(int, int);
int ps(void);
// Assignment 1.3
int send(int, int, void*);
int recv(void*);
int send_multi(int, int*, void*);
```

1.1 Syscall Implementation

toggle: An int was created which when 0 indicated `TRACE_OFF`, when 1 indicated `TRACE_ON`.
print_count Prints the frequency array, by first sorting it.
add A function which takes in 2 int and returns the sum. A message is printed if the int cannot be retrieved from the stack.
ps Using the ptable all the current, not `UNUSED` processes are printed out. The ptable is locked while the function is being executed so that there are no changes in it.

2 IPC

For IPC, we need a message buffer to store all the messages for all the processes. I tried a few iterations of it. First, was to store the buffer in the proc struct and a bit to indicate if a message is received. The bit and the buffer are initialized (malloc) while `allocproc`. However, while doing the distributed algorithm, I realized we needed more than one buffer, so I created an array of them (still in the process struct), but that lead to problems while forking (messages were forked). So, finally, I created an external (common) array of messages for all the processes.

```
char messages[NPROC][MAX_MSG_PER_PROC][MSG_SIZE];
int msg_count[NPROC] = {0};
```

send The message buffer is copied to the common buffer array, (with checks for the limit of messages etc.). The message count is increased. The receiver process is made runnable if it is sleeping. This is all done while the ptable is locked.

```
int send_message(int s_pid, int r_pid, const char *msg){
  struct proc *rec_proc = get_proc_by_id(r_pid);
  acquire(&ptable.lock);

  if (msg_count[r_pid] >= MAX_MSG_PER_PROC){
    cprintf("[ERROR] Too many messages\n");
    release(&ptable.lock);
    return -1;
  }

  for (int i=0; i<MSG_SIZE; i++){
    messages[r_pid][msg_count[r_pid]][i] = *(msg + i);
  }
  msg_count[r_pid] += 1;
  if (rec_proc->state == SLEEPING){
    wakeup1(&rec_proc);
  }
  release(&ptable.lock);
  return 0;
}
```

recv Last message from the common buffer array is copied into the received message buffer. This is all done while the ptable is locked.

```c
int receive_message(char *msg){
  acquire(&ptable.lock);
  struct proc *cur_proc = myproc();
  if (msg_count[cur_proc->pid] <= 0){
    release(&ptable.lock);
    return -1;
  }
  for (int i=0; i<MSG_SIZE; i++){
    *(msg + i) = messages[cur_proc->pid][msg_count[cur_proc->pid] - 1][i];
  }
  msg_count[cur_proc->pid] -= 1;
  release(&ptable.lock);
  return 0;
}
```

send_multi A basic for loop to iterate over the `pids[]` and send individual messages to each

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

  if(argint(0, &sender_pid) < 0 || argptr(1, (void*)&rec_pids, 8 * sizeof(int)) < 0 || argptr(2, &msg, 8)<0) {
      return -1;
  }
  for (int i=0; i < 8; i++){
    int rec_pid = *(rec_pids + i);
    if (rec_pid > 0){
      int return_value = send_message(sender_pid, rec_pid, msg);
      if (return_value == -1){
        return -1;
      }
    }
  }
  return 1;
}
```

3 Distributed Algorithm

3.1 Unicast Sum

For the distributed algorithm, in total 9 processes are created (1 parent coordinator process, and 8 child worker process).

```c
for (int i = 1; i< MAX_PROCESSES; i++){
    child_id = i;
    pid = fork();
    if (pid == 0){
        // this is child;
        break;
    }else if (pid > 0){
        // this is parent
        final_child_pcount ++;
        child_id = 0;
    }
}
```

In this loop, 8 processes are created, each with a child_id. Whenever a child is created, it breaks out of the loop (to prevent forking of the child recursively), and it executes a worker process and exits after its completion. The chunk size is automatically decided by the number of running processes. Each worker sums all the numbers in the chuck.

```c
// this is common for all children
int start = (child_id - 1) * chunk_size;
int end = ( start + chunk_size - 1 ) > size - 1 ? size - 1 : ( start + chunk_size - 1 );
int partial_sum = 0;
for (int i = start; i <= end; i++){
    partial_sum += arr[i];
}
char* msg = (char*)malloc(MSG_SIZE);
itoa(partial_sum, msg);
send(getpid(), COORDINATOR_PID, msg);
exit();
```

The parent executes the coordinator process by receiving all the partial sums, adding them up and returning it. It also waits for all the child processes to exit.

```c
for (int i=1; i< MAX_PROCESSES; i++){
    char* partial_sum = (char*)malloc(MSG_SIZE);
    int stat = -1;
    while(stat == -1){
        stat = recv(partial_sum);
    }
    final_sum += atoi(partial_sum);
    free(partial_sum);
}

for (int i=1; i<MAX_PROCESSES; i++){
    wait();
}
return final_sum;
```

3.2 Multicast Variance

The process is similar to the unicast sum. The processes are created in an identical way and called similarly. The parent process, however, first sends a multicast message to all the children, the sum of the arr. The children receive it, calculates the variance for their chunk and sends it to the parent which receives it and calculates the final variance.

```c
char* mean_msg = (char *)malloc(MSG_SIZE);
int stat = -1;
while(stat == -1){
    stat = recv(mean_msg);
}
float mean = 1.0 * atoi(mean_msg);
int start = (child_id - 1) * chunk_size;
int end = ( start + chunk_size - 1 ) > size - 1 ? size - 1 : ( start + chunk_size - 1 );
float partial_sum = 0.0;
for (int i = start; i <= end; i++){
    partial_sum += (100.0*arr[i] - mean)*(100.0*arr[i] - mean);
}
int partial_sum_round = (int)(partial_sum);
char* msg = (char*)malloc(MSG_SIZE);
itoa(partial_sum_round, msg);
send(getpid(), COORDINATOR_PID, msg);
exit();
```

Figure 1. Child Process

```c
char* msg = (char*)malloc(MSG_SIZE);
float mean = 1.0 * sum/size;
int mean_round = (int)(mean*100.0);
itoa(mean_round, msg);
send_multi(COORDINATOR_PID, pids, msg);

for (int i=1; i< MAX_PROCESSES; i++){
    char* partial_sum_round = (char*)malloc(MSG_SIZE);
    int stat = -1;
    while(stat == -1){
        stat = recv(partial_sum_round);
    }
    final_variance += atoi(partial_sum_round)/10000.0;
    free(partial_sum_round);
}

for (int i=1; i<MAX_PROCESSES; i++){
    wait();
}
return final_variance / size;
```

Figure 2. Parent Process

Since, the itoa function I created for converting int to strings, only accepts int, floats are converting to int by shifting the decimal place to maintain some precision.

4 Extra

Proper user programs are created for add, which takes the arguments from the command line while the user program is running, check if it is a number and output it.