OS Assignment 1

Report

Yash Malviya
2016CS50403

Toggle

A flag variable to maintain whether TRACE is ON or OFF. An array to save number time system call called

```
int timesSysCallsUsed[totalSysCallNum];
int traceSyscalls = 0;
```

If trace is on then start then update count when syscall is actually called.

```
void
syscall(void)
{
 int num;
 struct proc *curproc = myproc();

 num = curproc->tf->eax;
 if(num > 0 && num < NELEM(syscalls) &&
syscalls[num]) {
   if (traceSyscalls==1) {
     timesSysCallsUsed[num-1] += 1;
   }
   curproc->tf->eax = syscalls[num]();
 }
```

Toggle switch. When trace is on reset array

```
int sys_toggle(void) {
 if (traceSyscalls == 1) {
   traceSyscalls = 0;
 } else {
       traceSyscalls = 1;
       for (int i=0; i<totalSysCallNum; i++) {
           timesSysCallsUsed[i] = 0;
   }
 }
 cprintf("trace set to %d\n", traceSyscalls);
 return 22;
}
```

Print Count

Just print the previously store counts. Don’t print calls with 0 number of calls.

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

```
for (int i=0; i<totalSysCallNum; i++) {
        if (timesSysCallsUsed[i] != 0) {
            cprintf("%s %d\n",syscalls[i],
timesSysCallsUsed[i]);
        }
    }
    return 23;
}
```

Add and ps

Simply addition of ints

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

```
int ps(void) {
  for (int i=0; i<NPROC; i++) {
   struct proc* p = &ptable.proc[i];
   if (p->state!=UNUSED) {
            cprintf("pid:%d name:%s\n", p->pid, p->name);
   }
  }
 return 25;
}
```

Iterate over process table. If state is unused. Print the process Id and Name

IPC Unicast

Array of queue Data Structure
Each queue has its locks to prevent concurrent writes

```
// msg queue array
struct {
 char
message_store[NPROC][200*MSGSIZE];
 int info[NPROC][3];
 int waiting_for_recv[NPROC];
 struct spinlock letter_box_locks[NPROC];
 int chan_start;
 int max_q_size;
} msg_q_arr;

int is_msg_q_arr_init = 0;
```

Queue send and receive snippet to enqueue and dequeue.

```
// enqueue
char* char_msg = (char*) msg;
 int tail = msg_q_arr.info[pt_index][1];
 for (int i=0; i<MSGSIZE; i++) {
   msg_q_arr.message_store[pt_index][tail+i] =
*(char_msg+i);
 }
 msg_q_arr.info[pt_index][1] =
(msg_q_arr.info[pt_index][1]+MSGSIZE)%ms
g_q_arr.max_q_size;
 msg_q_arr.info[pt_index][2] += MSGSIZE;
```

```
// dequeue
char* char_msg = (char*) msg;
 int head = msg_q_arr.info[pt_index][0];
 for (int i=0; i<MSGSIZE; i++) {
   *(char_msg+i) =
msg_q_arr.message_store[pt_index][head+i];
 }

 msg_q_arr.info[pt_index][0] =
(msg_q_arr.info[pt_index][0]+MSGSIZE)%ms
g_q_arr.max_q_size;
 msg_q_arr.info[pt_index][2] -= MSGSIZE;
```

Blocking and waking for unicast

If queue empty sleep the process

```
if (msg_q_arr.info[pt_index][2]<=0) {
   msg_q_arr.waiting_for_recv[pt_index] = 1;
   struct proc* chan;
   chan = &ptable.proc[pt_index];
   sleep(chan,
&msg_q_arr.letter_box_locks[pt_index]);
 }
```

Wake when any sender adds to queue

```
if (msg_q_arr.waiting_for_recv[pt_index] == 1) {
   struct proc* chan;
   chan = &ptable.proc[pt_index];
   wakeup(chan);
   msg_q_arr.waiting_for_recv[pt_index] = 0;
 }
```

Distributed Algorithm

```
int total_children = 7;
    int child_pid_arr[total_children];
    int children_spawned = 0;
    int parent_pid = getpid();
    short* start_pos;
    short* last_pos;

    int cid;
    while(children_spawned<total_children) {
        cid = fork();
        children_spawned++;
        if (cid==0) {
            start_pos = (short*) &arr;
            last_pos = (short*) &arr;
            start_pos += (children_spawned-1)*(size/total_children);
            if (children_spawned==total_children) {
                last_pos += size;
} else {
                last_pos +=
(children_spawned)*(size/total_children);
            }
            int partial_sum = array_sum(start_pos, last_pos);
            int* msg = (int*) malloc(8);
            *msg = partial_sum;
            // printf(1, "%d\n", *((int*) msg));
            send(getpid(), parent_pid, msg);
            free(msg);
            break;
        }
        child_pid_arr[children_spawned-1] = cid;
    }
```

```
float mean;
    if (cid!=0) {
        float* msg = (float*) malloc(8);
        for (int i=0; i<total_children; i++)
            recv(msg);
            tot_sum += *((int*)msg);
        }
        if (type==1) {
            mean = (float) tot_sum;
            mean /= (float) size;
            *msg = mean;
            for (int i=0; i<total_children; i++) {
                send(parent_pid, child_pid_arr[i], msg);
            }
```

```
for (int i=0; i<total_children; i++) {
                recv(msg);
                variance += *msg;
            }
        variance /= (float) size;
        }
        free(msg);
        for (int i=0; i<children_spawned; i++) {
            wait();
        }
    } else {
        if (type==1) {
            float* msg = (float*) malloc(8);
            recv(msg);
            mean = *msg;
            float sqr_sum =
array_sqr_distance_from_mean(mean, start_pos, last_pos);
```

```
*msg = sqr_sum;
            send(getpid(),
parent_pid, msg);
            free(msg);
        }
        exit();
```