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 switched to on state, reset the array. This is done to initialize array with 0s as well as for removing value for values for last TRACE_ON state.

```
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

Print the previously stored counts of system calls on the console. Don’t print calls with 0 number of calls.

System call names are stored in syscall char array.

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

```
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",
};
```

Add

sys_add takes 2 ints and return the addition of the ints

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

ps

Iterate over process table. Print the process Id and Name of all process except whose state is unused.

```
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;
}
```

IPC Unicast

Queue send and receive snippet to enqueue and dequeue.

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;
```

```
// 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 enqueues message to previously slept processes' 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

Fork total_children number of children.

Each child compute sum of part of array.

Size of array for each child is 1/7th of total array size

Parent is the coordinator. It adds all partial sums to calculate total sum.

```
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;
            send(getpid(), parent_pid, msg);
            free(msg);
            break;
        }
        child_pid_arr[children_spawned-1] = cid;
    }
```

Sum of elements and calculations for variance of subarray

Both functions take pointer of starting position and ending location of subarry.

Iterate over all elements of subarray and do the required calculations.

```
int array_sum(short* start_pos_array,
short* last_pos_array) {
    int local_sum = 0;
    for (short* i=start_pos_array;
i<last_pos_array; i++) {
        local_sum += (int) *i;
    }
    return local_sum;
}
```

```
float array_sqr_distance_from_mean(float mean, short*
start_pos_array, short* last_pos_array) {
    float local_sum = 0.0;
    for (short* i=start_pos_array; i<last_pos_array; i++) {
        float fi = (float) *i;
        float diff = mean-fi;
        local_sum += diff*diff;
    }
    return local_sum;
}
```

Sending and receiving messages for parent

Communication between is done using unicast to calculate total sum and variance.

1. Wait for sum of subarray. By using sys_recv(msg). Gets blocked if no message present.
2. Calculate mean
3. Send mean to children (Done by unicast here).
4. Wait to receive sum of square of difference of element and mean.
5. Wait for all children to exit.

```
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();
      }
```

Sending and receiving messages for child process

All children communicate with coordinator process (parent).

1. Receive mean from coordinator
2. Send sum of square of difference of element and mean.
3. Exit.

```
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();
    }
```