OS Assignment 1

Rahul Bansal
2016CS10344

Distributed Algorithm

- This algorithm is for the calculation of sum of an array using 8 processes.
- Then each of the processes send the sum calculated of its part of the array to the parent process which then sums them individual sums and finds out the sum of the entire array
- For the implementation I have created 8 child processes fo the parent process by using the **fork() system call** -

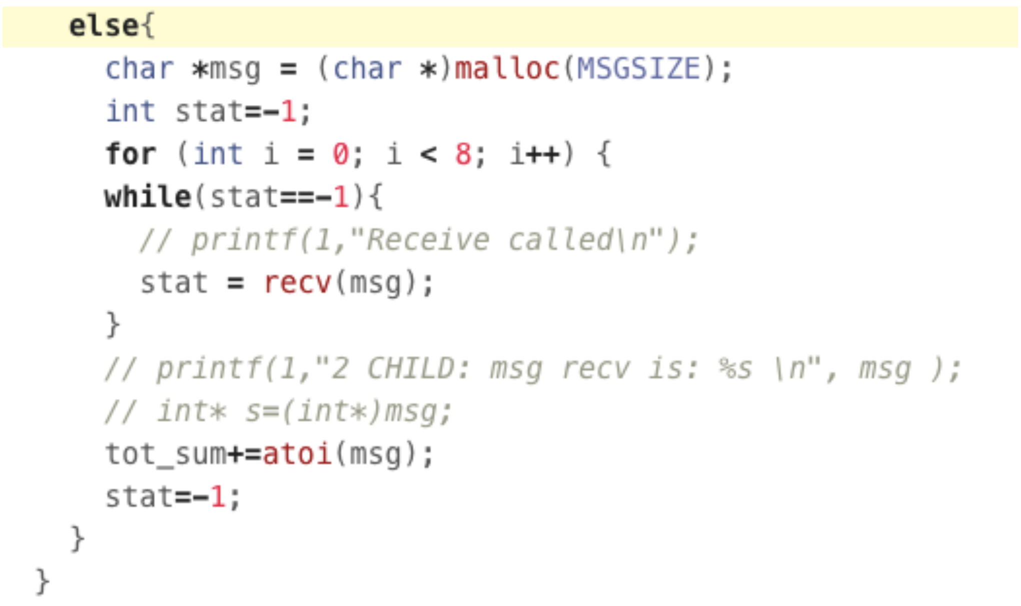
```
  for (i = 0; i < 8; i++) {
  cid = fork();
  if (cid==0) break;
}
```

In this loop the parent process creates 8 child processes which each of them calculate the sum of the sub-array.

- Now each of the child process calculates the sum of its sub array and send the value of the sum using the **send system call** to the parent process. Snippet of chid process code-

```
if(cid==0){
  //
  char* msg_child= (char *)malloc(MSGSIZE);
  int sum_1=0;
  for (int a = 125*i; a < (i+1)*125; a++) {
    sum_1+=arr[a];
  }
  strcpy(msg_child,itoa(sum_1,10));
  send(cid,pid,msg_child);

  free(msg_child);

  exit();
}
```

- Then now the parent process has to receive all the calculated partial sums and sum them. It receives the partial sums by using the **recv system call.**

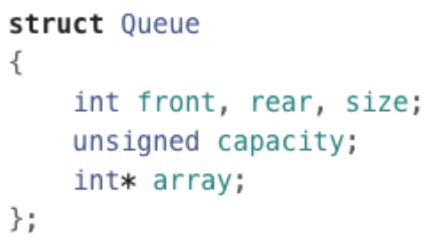


```
  else{
    char *msg = (char *)malloc(MSGSIZE);
    int stat=-1;
    for (int i = 0; i < 8; i++) {
    while(stat==-1){
      // printf(1,"Receive called\n");
      stat = recv(msg);
    }
    // printf(1,"2 CHILD: msg recv is: %s \n", msg );
    // int* s=(int*)msg;
    tot_sum+=atoi(msg);
    stat=-1;
  }
}
```

It receives all the 8 partial sums and then adds them to the tot_sum variable.

Inter Process Communication

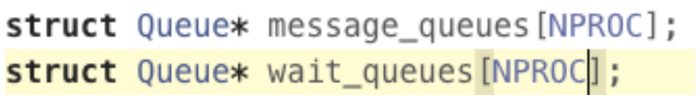
Defined Structs-
1. Struct Queue



```
struct Queue
{
    int front, rear, size;
    unsigned capacity;
    int* array;
};
```

For this struct I implemented all the required functions for the queues.

2. Wait and Message Queues



```
struct Queue* message_queues[NPROC];
struct Queue* wait_queues[NPROC];
```

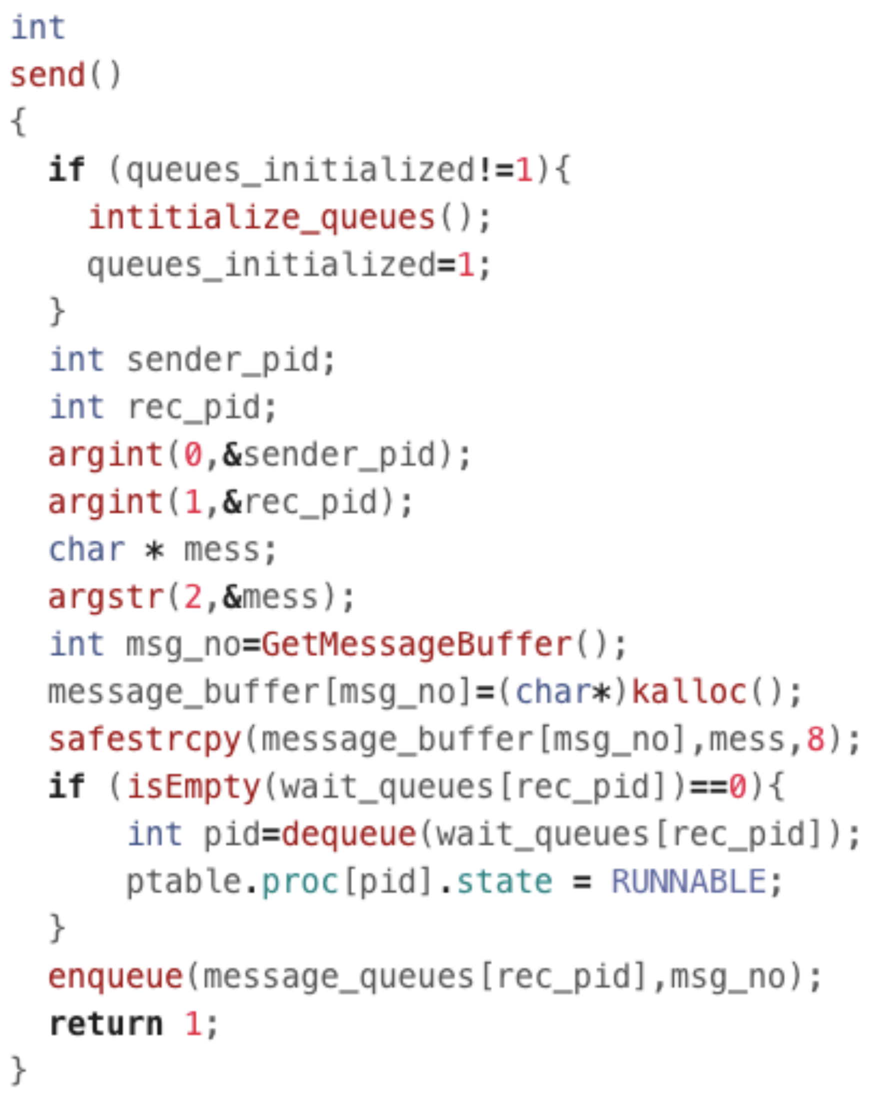
These are the respective wait and message queues which are used in the inter-process communication.

(a) Unicast-

In this we have to implement 2 system calls I.e.

sys_send(int sender_pid, int rec_pid, void *msg)
sys_recv(void *msg)

- In **send function** we insert the msg into the message_buffer and then check if the wait queue corresponding to the receiver pid is empty or not.
- If the queue is not empty then make the receiver process state to runnable and dequeue the pid from the queue
- If the queue is empty then we enqueue the msg_no corresponding to the message_buffer into the message_queue corresponding to the receiver process.



```
int
send()
{
  if (queues_initialized!=1){
    intitialize_queues();
    queues_initialized=1;
  }
  int sender_pid;
  int rec_pid;
  argint(0,&sender_pid);
  argint(1,&rec_pid);
  char * mess;
  argstr(2,&mess);
  int msg_no=GetMessageBuffer();
  message_buffer[msg_no]=(char*)kalloc();
  safestrcpy(message_buffer[msg_no],mess,8);
  if (isEmpty(wait_queues[rec_pid])==0){
      int pid=dequeue(wait_queues[rec_pid]);
      ptable.proc[pid].state = RUNNABLE;
  }
  enqueue(message_queues[rec_pid],msg_no);
  return 1;
}
```

- In **recv function** if the message queue is empty then it enqueues its pid into the wait queue and then the recv system process is blocked i.e. set the state to sleeping.

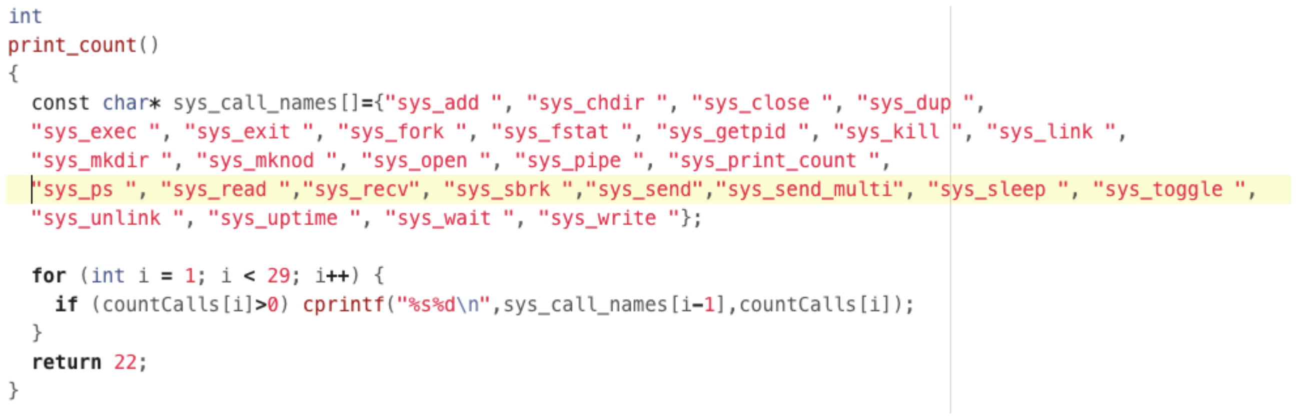
- If the message queue is not empty then we dequeue the msg_no from the message_queue and then fetch the msg from the message buffer.

```
int
sys_recv(void *msg)
{
  if (toggle_value==1) {
    countCalls[19]+=1;
  }
  char* mess;
  argstr(0,&mess);
  if (queues_initialized!=1){
    intitialize_queues();
    queues_initialized=1;
  }
  struct proc *curproc = myproc();

  if (isEmpty(message_queues[curproc->pid])==1){
    curproc->state=SLEEPING;
    enqueue(wait_queues[curproc->pid],curproc->pid);
    sched();
  }
  int msg_no=dequeue(message_queues[curproc->pid]);
  safestrcpy(mess,message_buffer[msg_no],8);
  return 0;
}
```

Add System Call

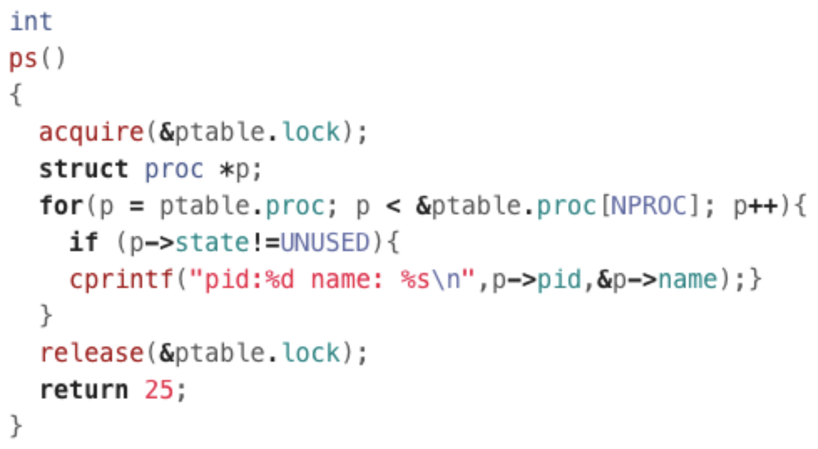
```
int
sys_add(void)
{
  int a;
  int b;
  argint(0,&a);
  argint(1,&b);
  // cprintf("%d\n",a+b);
  if (toggle_value==1) {
    countCalls[1]+=1;
  }
  return a+b;
}
```

Print Count System Call



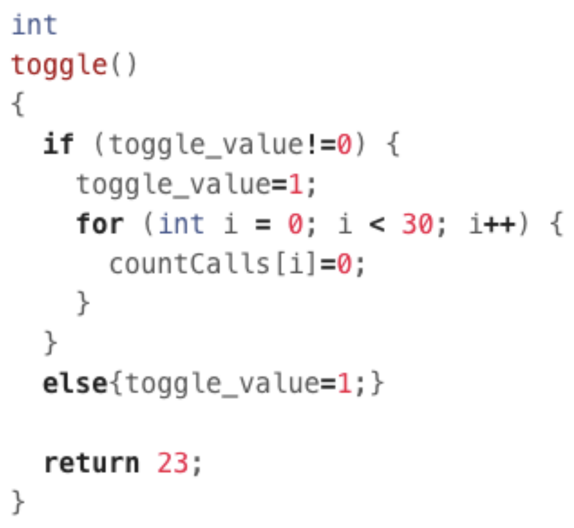
```
int
print_count()
{
  const char* sys_call_names[]={"sys_add ", "sys_chdir ", "sys_close ", "sys_dup ",
  "sys_exec ", "sys_exit ", "sys_fork ", "sys_fstat ", "sys_getpid ", "sys_kill ", "sys_link ",
  "sys_mkdir ", "sys_mknod ", "sys_open ", "sys_pipe ", "sys_print_count ",
  "sys_ps ", "sys_read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

  for (int i = 1; i < 29; i++) {
    if (countCalls[i]>0) cprintf("%s%d\n",sys_call_names[i-1],countCalls[i]);
  }
  return 22;
}
```

PS System Call



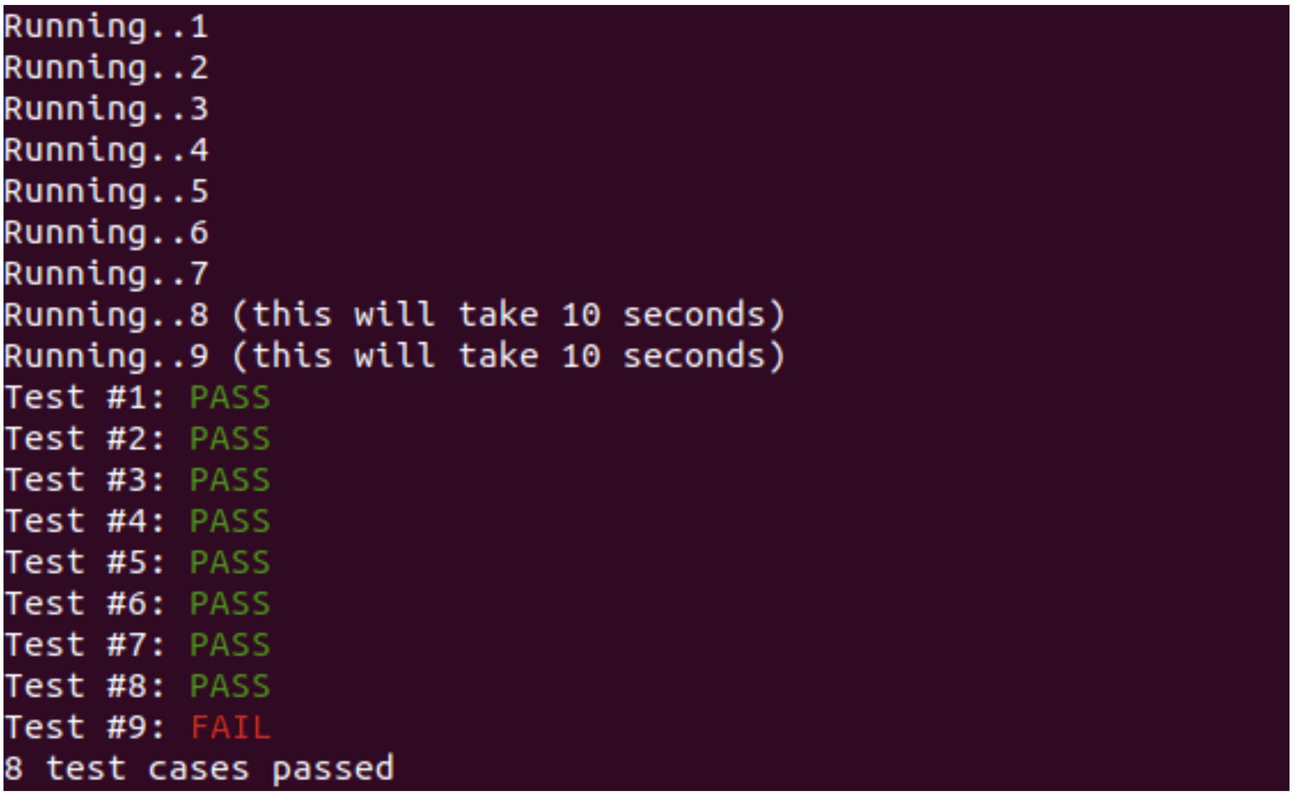
```
int
ps()
{
  acquire(&ptable.lock);
  struct proc *p;
  for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if (p->state!=UNUSED){
    cprintf("pid:%d name: %s\n",p->pid,&p->name);}
  }
  release(&ptable.lock);
  return 25;
}
```

Toggle System Call



```
int
toggle()
{
  if (toggle_value!=0) {
    toggle_value=1;
    for (int i = 0; i < 30; i++) {
      countCalls[i]=0;
    }
  }
  else{toggle_value=1;}

  return 23;
}
```

Test Cases



```
Running..1
Running..2
Running..3
Running..4
Running..5
Running..6
Running..7
Running..8 (this will take 10 seconds)
Running..9 (this will take 10 seconds)
Test #1: PASS
Test #2: PASS
Test #3: PASS
Test #4: PASS
Test #5: PASS
Test #6: PASS
Test #7: PASS
Test #8: PASS
Test #9: FAIL
8 test cases passed
```