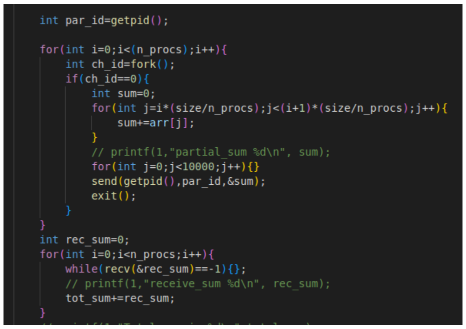
COL331

1 Distributed Algorithm: How exactly does your algorithm work?

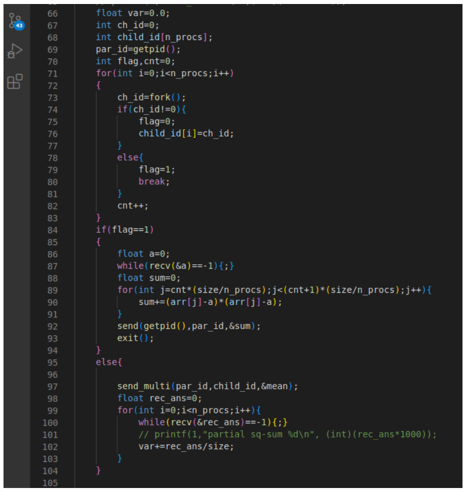
First, I have calculated the total sum and variance using two different for loops in a distributed algorithm, the first one is an explaination of total sum part.



```
int par_id=getpid();

for(int i=0;i<(n_procs);i++){
    int ch_id=fork();
    if(ch_id==0){
        int sum=0;
        for(int j=i*(size/n_procs);j<(i+1)*(size/n_procs);j++){
            sum+=arr[j];
        }
        // printf(1,"partial_sum %d\n", sum);
        for(int j=0;j<10000;j++){}
        send(getpid(),par_id,&sum);
        exit();
    }
}
int rec_sum=0;
for(int i=0;i<n_procs;i++){
    while(recv(&rec_sum)==-1){};
    // printf(1,"receive_sum %d\n", rec_sum);
    tot_sum+=rec_sum;
}
```

Figure 1: `Distributed algorithm:Total sum`

In , this parid is used to store the coordinator process which will accumulate the sum created by the child processes. The for loop is used in which fork is used which creates new child process. So, here if the chid is 0 then the particular process is a child process. So, we use this knowledge that this process should be used to calculate partial sum. In this we divide the 1000 size array into 125 size chunks and each child process works on itself given 125 size chunk.The for loop after that is used to provide a little wait. Then send uses the pid of the given process and send the sum to the specified parent whose id is stored in parid. After the loop completes all the send shall reach to parent process and then using the loop it sums all the partial sum received using sys receive system call to total sum.



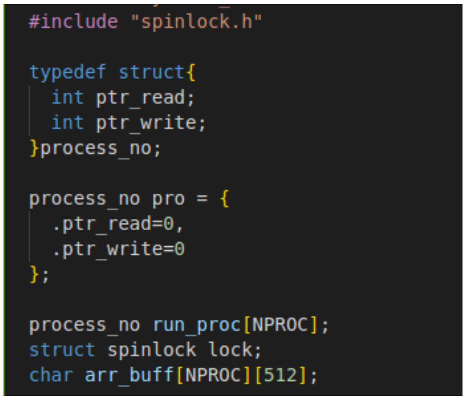
```c
float var=0.0;
int ch_id=0;
int child_id[n_procs];
par_id=getpid();
int flag,cnt=0;
for(int i=0;i<n_procs;i++)
{
    ch_id=fork();
    if(ch_id!=0){
        flag=0;
        child_id[i]=ch_id;
    }
    else{
        flag=1;
        break;
    }
    cnt++;
}
if(flag==1)
{
    float a=0;
    while(recv(&a)==-1){;}
    float sum=0;
    for(int j=cnt*(size/n_procs);j<(cnt+1)*(size/n_procs);j++){
        sum+=(arr[j]-a)*(arr[j]-a);
    }
    send(getpid(),par_id,&sum);
    exit();
}
else{

    send_multi(par_id,child_id,&mean);
    float rec_ans=0;
    for(int i=0;i<n_procs;i++){
        while(recv(&rec_ans)==-1){;}
        // printf(1,"partial sq-sum %d\n", (int)(rec_ans*1000));
        var+=rec_ans/size;
    }
}
```

Figure 2: `Distributed algorithm:Variance`

After calculating the total sum, this code calculates the variance. the parid again used to store the parent id.The for loop creates the different child processes of parent processes. In the loop if the process is a parent process then the array childid is used to store all the child process ids to this process. If it is a child process then loop that process comes out of loop as we have to compute the variance this child process is used to calculate the the diferent element in a chunk subtracted with the mean which is sent by the parent process to different child processes using send multi system call, then this calculated answer is sent to parent process then parent process receives this answer from all the child processes then calculates the variance by summing them and dividing them by the size of array.

2 IPC: Explain the implementation of the interrupt handler

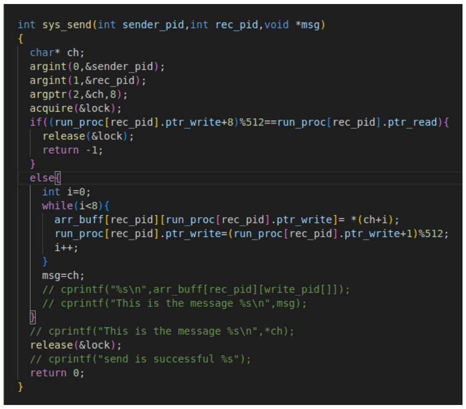
In this part three functions are made one for send, one for receive and one for send multi.



```c
#include "spinlock.h"

typedef struct{
  int ptr_read;
  int ptr_write;
}process_no;

process_no pro = {
  .ptr_read=0,
  .ptr_write=0
};

process_no run_proc[NPROC];
struct spinlock lock;
char arr_buff[NPROC][512];
```

Figure 3: `Struct used in send and receive`

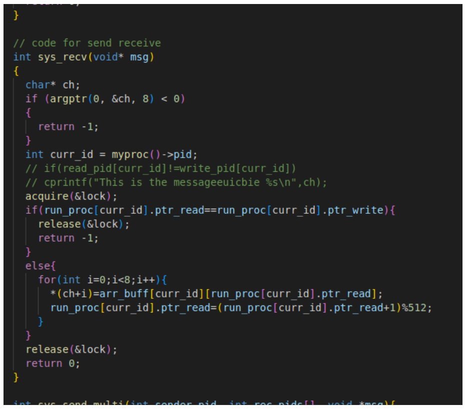
This new structure is created whose objects are created in the functions created.



```c
int sys_send(int sender_pid,int rec_pid,void *msg)
{
  char* ch;
  argint(0,&sender_pid);
  argint(1,&rec_pid);
  argptr(2,&ch,8);
  acquire(&lock);
  if((run_proc[rec_pid].ptr_write+8)%512==run_proc[rec_pid].ptr_read){
    release(&lock);
    return -1;
  }
  else{
    int i=0;
    while(i<8){
      arr_buff[rec_pid][run_proc[rec_pid].ptr_write]= *(ch+i);
      run_proc[rec_pid].ptr_write=(run_proc[rec_pid].ptr_write+1)%512;
      i++;
    }
    msg=ch;
    // cprintf("%s\n",arr_buff[rec_pid][write_pid[]]);
    // cprintf("This is the message %s\n",msg);
  }
  // cprintf("This is the message %s\n",*ch);
  release(&lock);
  // cprintf("send is successful %s");
  return 0;
}
```

Figure 4: `This function is created to implement send`

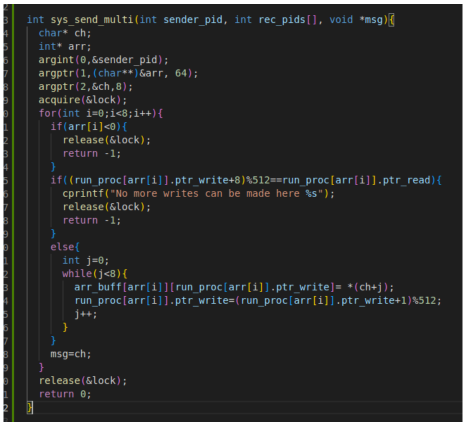
This function is created in which argint and agpptr are used to take input of sender pid , recive pid and the message. In this the basic algorithm used is a type of circular buffer in this for each pid a buffer is created whose size is 512 , this number is selected as it is multiple of 8 which is the size of message received. Here two pointers are used which are defined in the structure are read pointer and write pointer which are used to point to the point till which we have written in a respective pid buffer. If the message written and it gets overlap then we say that we cant write any more in the buffer and return -1 in that case. Here locks are used using spinlock which are used to lock a particular thread which are writing to the particular buffer.



```
}

// code for send receive
int sys_recv(void* msg)
{
  char* ch;
  if (argptr(0, &ch, 8) < 0)
  {
    return -1;
  }
  int curr_id = myproc()->pid;
  // if(read_pid[curr_id]!=write_pid[curr_id])
  // cprintf("This is the messageeuicbie %s\n",ch);
  acquire(&lock);
  if(run_proc[curr_id].ptr_read==run_proc[curr_id].ptr_write){
    release(&lock);
    return -1;
  }
  else{
    for(int i=0;i<8;i++){
      *(ch+i)=arr_buff[curr_id][run_proc[curr_id].ptr_read];
      run_proc[curr_id].ptr_read=(run_proc[curr_id].ptr_read+1)%512;
    }
  }
  release(&lock);
  return 0;
}

int sys_send_multi(int sender_pid, int rec_pids[], void *msg){
```

Figure 5: `This function is created to implement recieve`

This function input is same as earlier. Here we receive then read the particular message from the buffer.

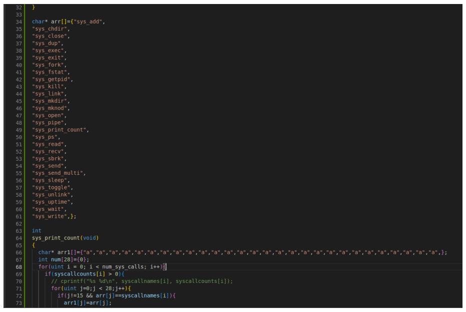


```
int sys_send_multi(int sender_pid, int rec_pids[], void *msg){
  char* ch;
  int* arr;
  argint(0,&sender_pid);
  argptr(1,(char**)&arr, 64);
  argptr(2,&ch,8);
  acquire(&lock);
  for(int i=0;i<8;i++){
    if(arr[i]<0){
      release(&lock);
      return -1;
    }
    if((run_proc[arr[i]].ptr_write+8)%512==run_proc[arr[i]].ptr_read){
      cprintf("No more writes can be made here %s");
      release(&lock);
      return -1;
    }
    else{
      int j=0;
      while(j<8){
        arr_buff[arr[i]][run_proc[arr[i]].ptr_write]= *(ch+j);
        run_proc[arr[i]].ptr_write=(run_proc[arr[i]].ptr_write+1)%512;
        j++;
      }
    }
    msg=ch;
  }
  release(&lock);
  return 0;
}
```

Figure 6: `This function is created to implement send multi`

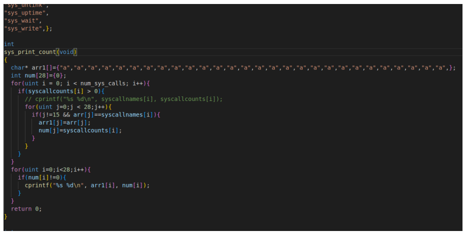
This function is same as send just here that an array of fixed size 8 is given, for loop is used and same type of code structure used in send function is used in this for loop.

3 Print Count



```c
}

char* arr[]={"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
sys_print_count(void)
{
  char* arr1[]={"a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a",};
  int num[28]={0};
  for(uint i = 0; i < num_sys_calls; i++){
    if(syscallcounts[i] > 0){
      // cprintf("%s %d\n", syscallnames[i], syscallcounts[i]);
      for(uint j=0;j < 28;j++){
        if(j!=15 && arr[j]==syscallnames[i]){
          arr1[j]=arr[j];
```

Figure 7: This function is created to implement print count

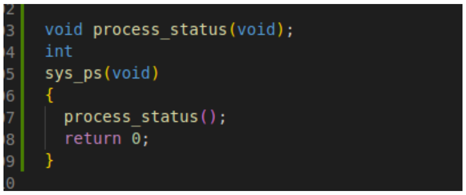


```c
"sys_unlink",
"sys_uptime",
"sys_wait",
"sys_write",};

int
sys_print_count(void)
{
  char* arr1[]={"a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a","a",};
  int num[28]={0};
  for(uint i = 0; i < num_sys_calls; i++){
    if(syscallcounts[i] > 0){
      // cprintf("%s %d\n", syscallnames[i], syscallcounts[i]);
      for(uint j=0;j < 28;j++){
        if(j!=15 && arr[j]==syscallnames[i]){
          arr1[j]=arr[j];
          num[j]=syscallcounts[i];
        }
      }
    }
  }
  for(uint i=0;i<28;i++){
    if(num[i]!=0){
      cprintf("%s %d\n", arr1[i], num[i]);
    }
  }
  return 0;
}
```

Figure 8: This function is created to implement print count

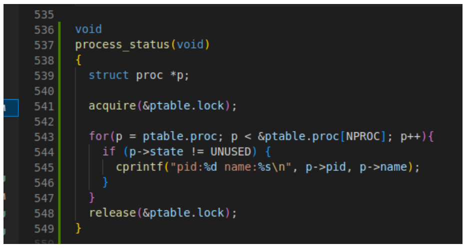
This system call is used to print the system call processes running in alphabetical order.The sorting is used in this is like an array is defined which contains all the system calls in a sorted order.Then all are compared to all the elements and then putted in a new array and then printed.

4 sysps



```c
void process_status(void);
int
sys_ps(void)
{
  process_status();
  return 0;
}
```

Figure 9: This function is created to implement ps system call

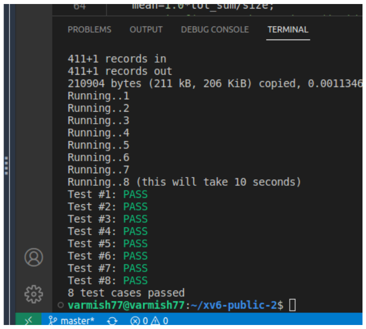


```c

void
process_status(void)
{
  struct proc *p;

  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);
}
```

Figure 10: `This function is created to implement ps system call`

In this the process status keeps the status of the of all the processes which is used by the ps system call.

This is the screenshot of the output of check.sh bash.



```
411+1 records in
411+1 records out
210904 bytes (211 kB, 206 KiB) copied, 0.0011346
Running..1
Running..2
Running..3
Running..4
Running..5
Running..6
Running..7
Running..8 (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
8 test cases passed
varmish77@varmish77:~/xv6-public-2$
```

Figure 11: `Test cases Result`