COL 331 Assignment 1

Name-Sibasish Rout
Entry number-2020CS10386

Distributed Algorithm:How does your algorithm work?

My distributed algorithm makes use of the send , receive and send_multiple system calls made in Interprocess communication part to carry out the distribute algorithm.

```
        {
            status=recv(&psum);
        }
        // printf(1,"Parent ID is %d\n",getpid());
        tot_sum=tot_sum+psum;
    }
    if(type==1)
    {
        float mean=1.0*tot_sum/1000;
        int child1[]={child[0],child[1],child[2],child[3],child[4],child[5],child[6],child[7]};
        send_multi(getpid(),child1,&mean);
        for(int i=0;i<8;i++)
        {
            int status=-1;
            float pvar;
            while(status==-1)
            {
                status=recv(&pvar);
            }
            // printf(1,"pvar is %d\n",(int)pvar);
            svar=svar+pvar;
        }
    }
```

The steps for the algorithm are:

i)First we fork the parent using a loop into 8 children and store the values of the pid in an array using the property that fork return zero in child and child id in parent.

ii)We know that the resources of the parent can be read by child so the child can read the array created by reading the arr file.

iii)We then divide the array into contiguous blocks among the children. The children calculate the partial sum and send back the results using sys_send() system call.

iv)The main parent process acts as a coordinator process and adds up the various partial sums received to get the total sum of the array and then use it to calculate the mean.

v)The parent sens this mean to all the children using the sys_send_multi() system call . The children after receiving the mean calculate the partial sum of the deviations in their part of the array and send it to the parent.

vi)The parent uses the partial sums of the squares of the deviation to calculate the variance and uses the print variance snippet to print the variance.

```
for(int j=cnt*125;j<(cnt+1)*125;j++)
{
    psum=psum+arr[j];
}
int status=-1;
while(status==-1)
{
    status=send(getpid(),parentID,&psum);
}
if(type==1)
{
    float mean;
    status=-1;
    while(status==-1)
    {
        status=recv(&mean);
    }
    // printf(1,"Mean is %d\n",(int)mean);
    float pvar=0.0;
    for(int j=cnt*125;j<(cnt+1)*125;j++)
    {
        pvar=pvar+(mean-arr[j])*(mean-arr[j]);
```

The algorithm uses parallelism to some extent as the child processes can simultaneosly calculate the sums and send it back to the coordinator process. This is very useful in cases where

System Calls

Here to define a new system call we have to make changes in 5 files in the xv6 system:-

i)syscall.c

This has the function called syscall which is the interface through which the system calls are called so we need to specify the signature of the system call here.

ii)syscall.h

This defines the macro for the system calls index which are utilised while calling the system calls

iii)sysproc.c

The actual system call and the functionalities are defined here. The function definition is present here.

iv)user.H

The function signature for the user functions to use.

v)usys.S

This serves as an interface from which the system calls can be called by the user functions.

To run user programs we must also make changes in the Makefile and add it under the user section so that the process becomes visible on the qemu terminal

For add we use simple addition and perform changes to the necessary files. The user program to test the add system call is user_add.c

```
int sys_add(void)
{
  int x,y;
  argint(0,&x);
  argint(1,&y);
  return x+y;
}
```

For toggle system call we simply define a trace variable of enum type in syscall.c file. When we call the system call we can switch the state.

```
int sys_toggle(void)
{
  if(trace==TRACE_ON)
  {
    trace=TRACE_OFF;
    // cprintf("Tggle is now set to trace off \n");
  }
  else if(trace==TRACE_OFF)
    {
      trace=TRACE_ON;
      // cprintf("Tggle is now set to trace on \n");
      int i=0;
      while(i!=numsc)
      {
        sysCnts[i]=0;
        i++;
      }
    }
  return 0;
}
```

For ps system call we use the table attribute defined in proc.c to find all the processes which are running and print their respective pids.

```
#include "spinlock.h"

struct {
  struct spinlock lock;
  struct proc proc[NPROC];
} ptable;

static struct proc *initproc;
```

For print_count system call we define our sorting function and then define our own string array having the names of all the system calls and use it to print the system calls and frequencies.

Inter Process Communication

i)Here we define two system calls sys_send and sys_recv for unicast and sys_send_multi for multicast. To implement this we define a global buffer for the messages.

ii)This buffer has two pointers namely the read pointer and the write pointer to modify and see the contents of the buffer.

iii)The sender drops the message in the buffer of the receiver and the receiver then extracts the messages from this buffer.

```
struct spinlock lock;
int sys_send(int sender_pid, int rec_pid, void *msg)
{
  char* m;
  if(argint(0,&sender_pid))
  return -1;
  if(argint(1,&rec_pid))
  return -1;
  if(argptr(2,&m,8))
  return -1;
  // cprintf("Hello");
  acquire(&lock);
  for(int i=0;i<8;i++)
  {
    msgBuffer[rec_pid][pw[rec_pid]]=*(m+i);
    pw[rec_pid]=(pw[rec_pid]+1)%8000;
  }
  // cprintf("Hello1");
  release(&lock);
  // cprintf("Message sent%d %d %d\n",rec_pid,pw[rec_pid],pr[rec_pid]);
  return 0;
}
```

iv)In sys_send_multi we just use a loop to drop the messages in the respective locations for each receiving process.The messages can then duly be accessed by the receivers from the message.The status for a buffer can be inferred from the two pointers.

v)To ensure correct order of read and writes we use locks so that the receiver does not read before the writer completes editing the buffer,

Extra

I have also made testcases for unicast and multicast communication . This can be used to extensively test the reliability of the communication process.

The checkscpript results are as follows