Operating System Hw3 Report

1. Implementation details:

a. Make parameter –x to support multiple user programs, and then call kernel->multiProgramTest(), which will call Thread::multiProgram(). [threads/main.cc]

```
else if (strcmp(argv[i], "-x") == 0) {
    ASSERT(i + 1 < argc);
    //userProgName = argv[i + 1];
        while (i + 1 < argc && argv[i + 1][0] != '-'){
           userProgName.push_back(std::string(argv[i + 1]));
           i++;
        }
        i--;
}
```

```
if (userProgName.size() != 0){
  kernel->multiProgramTest(userProgName);
  while (true){
     if (kernel->scheduler->getOrderList()->IsEmpty()) break;
     // kernel->currentThread->Yield();
  }
}
```

b. Implement OSPrint() just as what we have done in Hw1. We used PutString(), instead of PutChar() to implement it. [userprog/exception.cc]

We changed syscall.h, start.S, exception.cc and ksyscall.h.

```
int SysOSPrint(char* op1, int op2)
{
   int cnt = 0, flag = 0, d;
   char con[1024];
   bool err = false;

   while (true){
      if (!kernel->machine->ReadMem((int)op1 + flag, 1, &d))  err = true;
      con[cnt] = (char)d;
      cnt++;   flag++;
      if (cnt > 1 && con[cnt - 2] == '%' && con[cnt - 1] == 'd')  break;
   }
   cnt += sprintf(con + cnt - 2, "%d", op2) - 2;
   while (true){
      if (!kernel->machine->ReadMem((int)op1 + flag, 1, &d))  err = true;
      con[cnt] = (char)d;
      cnt++;   flag++;
      if (con[cnt - 1] == '\n'){
         con[cnt] = 0;
         break;
      }
   }
   kernel->synchConsoleOut->PutString(con);
   return cnt;
}
```

c. To implement automatically context switch, we turned off ConsoleWriteInt and prevent it from sleeping. [machine/console.cc]

```
void
SynchConsoleOutput::PutString(char* str)
{
    lock->Acquire();
    consoleOutput->PutString(str);
    //waitFor->P();
    lock->Release();
}
```

```
void
ConsoleOutput::PutString(char* str)
{
    //ASSERT(putBusy == FALSE);
    WriteFile(writeFileNo, str, strlen(str));
    putBusy = TRUE;
    //kernel->interrupt->Schedule(this, 1, ConsoleWriteInt);
}
```

d. Use an integer former to record how many physical pages other program have used. We changed both AddrSpace::AddrSpace(), and AddrSpace::Load() to fulfill this requirement. [userprog/addrspace.cc]

```
AddrSpace::AddrSpace(int former)
{
    pageTable = new TranslationEntry[NumPhysPages];
    for (int i = 0; i < NumPhysPages; i++) {
    pageTable[i].virtualPage = i;   // for now, virt
    pageTable[i].physicalPage = i + former;
    pageTable[i].valid = TRUE;
    pageTable[i].use = FALSE;
    pageTable[i].dirty = FALSE;
    pageTable[i].readOnly = FALSE;
    }

    // zero out the entire address space
    //bzero(kernel->machine->mainMemory, MemorySize);
}
```

e. For every user program, we first new an AddrSpace for it. Second, loaded the program into certain address of main memory, and, in the meanwhile, print out the required information. At the end, call thread::Fork(). After all the above-said were completed, let the currentThread call Yield(). [threads/thread.cc]

```
void static func(int r){
   kernel->currentThread->space->Execute();
}

void Thread::multiProgram(std::vector<std::string> &userProgName){
   int physicalMemBeing = 0;
   bzero(kernel->machine->mainMemory, MemorySize);

   for (std::vector<std::string>::iterator it=userProgName.begin(); it!=userProgName.end(); it++){
      printf("%s\n", it->c_str());
      AddrSpace *space = new AddrSpace(physicalMemBeing);
      ASSERT(space != (AddrSpace *)NULL);

      char *loadString = new char[it->length() + 1];
      strcpy(loadString, it->c_str());

      int inc = space->Load(loadString, physicalMemBeing);

      if (inc != 0){
         Thread *t = new Thread(loadString);
         t->space = space;

         t -> setPriority(1);
         t -> Fork((VoidFunctionPtr)func, (void *) 0);// (void *) (it-userProgName.begin()));
      }

      physicalMemBeing += inc;
   }
   kernel->currentThread->Yield();
}
```

f. When each thread was about to run, printed context switch information and also the certain thread name. [In threads/scheduler.cc]

```
void
Scheduler::Run (Thread *nextThread, bool finishing)
{
    fprintf(stdout, "###Context Switch###\n%s\n", nextThread->getName());
    fflush(stdout);
```

```
case SC_Exit:{
   kernel->interrupt->SetLevel(IntOff);
   kernel->currentThread->Sleep(true);
   return;
}
```

g. When a program reached its end, we turned off the interrupt handler for it and put the certain thread into sleep forever.

2. Problems that we have met:

a. First problem we met was that we did not know how to let one thread perform AddrSpace::Execute() as one of the parameter of Thread::Fork(). We checked a lot of websites for the solution, but we failed in vain. At the end, we set up the Thread::space to AddrSpace which is corresponding to the certain user program and then called another function to execute kernel->currentThread->space->Execute() indirectly.

b. The second problem was much more complicated and time-consuming for debugging. We initially found that we could not print a single-line-output via kernel->SynchConsoleOutput::PutChar(). So we added two similar functions, kernel->SynchConsoleOutput::PutString() and ConsoleOutput::PutString() respectively to fulfill this requirement.

   However, after that, the thread could only print out one line and then it disappeared forever. This is because of the following two reasons.

   The first reason was that we used the former homework code as basis and tried to accomplish Hw3 work with them. So the code we used now have marked "alarm = new Alarm(randomSlice)" in code/threads/kernel.cc and "OneTick()" in code/machine/interrupt.cc. That is to say, when one thread called waitFor->P(), it would be put to sleep, but with no one to execute CheckIsDue() in OneTick() to wake it up. What's more, there wasn't any kernel alarm, so even one hundred ticks passed, no interrupt of TimerInt would occur.

   Second reason was that when called PutString(), it would schedule an interrupt of ConsoleWriteInt into pending, and take the NextThread in OrderList. If the nextThread happened to be "main", then the nachos system would shut itself down and leave all the remaining threads alone.

3. Reference:

The majority concept of this homework came up with our mind, and was confirmed in the nachos code. We got quite a lot of help from my classmates. To just named a few, such as, B00902001, B00902061, and B00902107. A lot of resources were searched from Google.