NUCLEUS MESSAGE PASSING

Principles of Operating System Assignment 3

Sonal Patil [SUID: 997435672]
Chetali Mahore [SUID: 750500177]
Naina Bharadwaj [SUID: 810909672]

1. Concept –

Our assignment implements Nucleus Message Passing. In order to perform communication between threads and since original Nachos only allow one user program to run at a time, we have also implemented multiprogramming in Nachos. This allows multiple programs to run one by one (as per scheduler logic) and communicate between each other. For this we have referred the paper 'The Nucleus of a Multiprogramming System' by Per Brinch Hansen.

2. Design –

1. <u>Multiprogramming</u> –

   - In order to demonstrate Nucleus Message Passing, we need to run multiple user programs and to achieve that we need Nachos with Multiprogramming functionality.
   - We have implemented contiguous memory allocation multiprogramming. Since, the purpose of this assignment is to show the message passing communication between two/multiple threads, multiprogramming gets achieved by contiguous memory allocation.
   - For each user program a new thread gets forked and put in scheduler's ready list. This newly created thread gets appended into kernel's ListOfForkedThreads. The purpose of this list is to keep track of alive threads. At the end of each user program there is an Exit() system call which tells kernel that user program's job is done and can finish thread's execution. When that happens, we modify the ListOfForkedThreads by removing the entry of finishing thread.

2. <u>Message Passing</u> –

   - We have used a common pool of message buffers and a message queue for each thread instead of using lock and semaphores as described in the above paper.
   - In order to pass messages between threads i.e. between user programs, we have implemented the system calls which user program can use.
     1. SendMessage –
        Arguments – receiver (receiver thread name),
        message (actual message in our case a char string),
        buffer_id (buffer id of message buffer. When buffer is not allocated it's -1)

        Return – buffer_id (returns the buffer id allocated)

     2. WaitMessage –
        Arguments – sender (sender thread name),
        message (actual message in our case a char string),
        buffer_id (buffer id of message buffer. When buffer is not allocated it's -1)

        Return – buffer_id (returns the buffer id allocated)

3. SendAnswer –
   Arguments – result (dummy or answer, if -1 – dummy result, else – answer),
   answer (actual message in our case a char string),
   buffer_id (buffer id of message buffer. When buffer is not allocated it’s -1)

   Return – result (dummy or answer, if -1 – dummy result, else – answer)

4. WaitAnswer –
   Arguments – result (dummy or answer, if -1 – dummy result, else – answer),
   answer (char string),
   buffer_id (buffer id of message buffer. When buffer is not allocated it’s -1)

   Return – result (dummy or answer, if -1 – dummy result, else – answer)

- In order to avoid miscommunications or wasting of message sent to dead thread, as per suggested in above paper, we have set a limit on how many messages a thread can send to other threads. This message limit is defined on the kernel level as 10.
- In our assignment, kernel is our nucleus. It holds the threads created for user programs, buffers created during each communication, message limit. It also holds a bitmap to allocate the buffers. In our assignment, up to 100 buffers can be created.
- Kernel buffer pool holds map of integer i.e. buffer id and List of buffers created with same buffer id.
- Each thread then holds its own map of integer i.e. buffer id and buffer created with that buffer id.

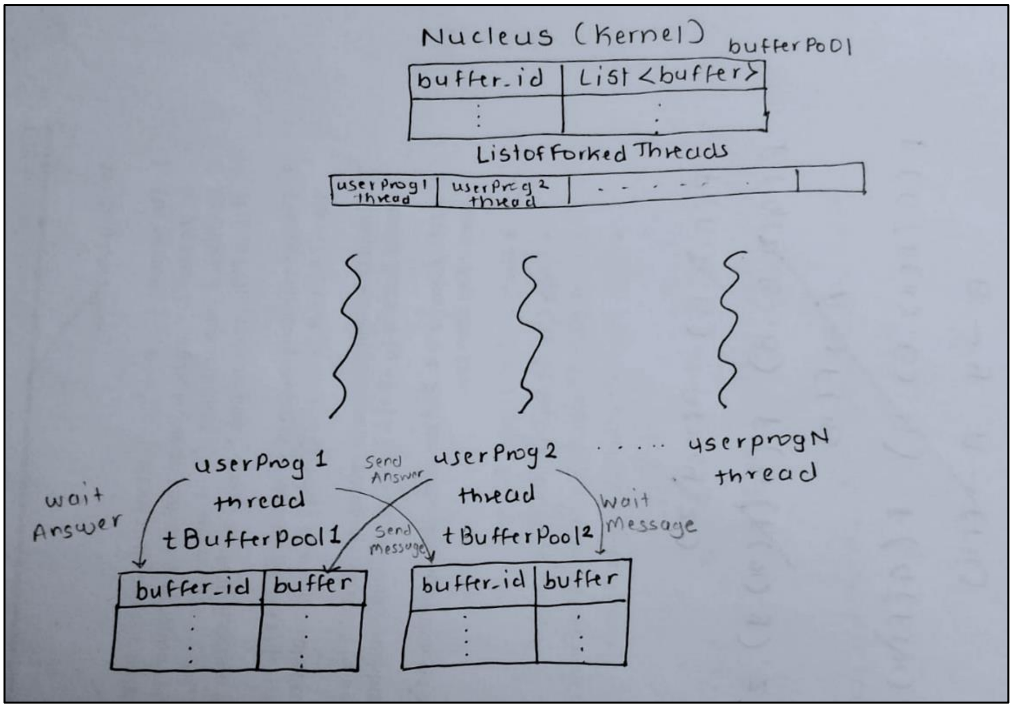


Fig. Nucleus Message Passing Design

- As per shown in above diagram, when a thread starts a communication with another thread, it gives a system call SendMessage. This system call then allocates a buffer depending upon the buffer availability and message limit of that thread. This buffer gets inserted into the receiver thread's (thread to whom current thread wants to send a message) tBufferpool (thread message queue).

- Vice versa, when a thread wants to send a reply back/answer to its message sent thread, it creates a buffer (if it's the first answer sending time) and inserts it into the sender thread's (thread who sent a message to current thread) tBufferpool (thread message queue).
- These two buffers have the same buffer id and they get inserted into the kernel's bufferPool (common pool of message buffers) with their buffer id as a single entry in map.
- When a thread finishes before sending a message and another thread is waiting for the message, a dummy response gets sent by the kernel in order to not break communication link and have a smooth message passing.
- It happens the same way when a thread finishes before sending the answer to sender thread.
- We have also implemented the security check feature for the message passing. If a thread tries to send a message using a wrong buffer, our Nucleus/kernel detects that and stops the communication.

3. **Development** –

1. Multiprogramming –

   To implement the multiprogramming feature, we have implemented the contiguous memory allocation for multiple programs to run simultaneously.

   We have implemented contiguous memory allocation as follows:

   - An int variable 'mark' keeps a track of the next free page in the page table for the next program to load.
   - The page table is initialized when AddrSpace::Load is called and the size of the program is calculated.
   - The files modified are nachos/code/userprog/addrspace.h and nachos/code/userprog/addrspace.cc

addrspace.h:

```
#ifndef ADDRSPACE_H
#define ADDRSPACE_H

#include "copyright.h"
#include "filesys.h"

#define UserStackSize         1024      // increase this as necessary!

class AddrSpace {
  public:
    AddrSpace();                        // Create an address space.
    ~AddrSpace();                       // De-allocate an address space

    bool Load(char *fileName);          // Load a program into addr space from
                    // a file
                                                  // return false if not found

    void Execute();               // Run a program
                                                  // assumes the program has already
                    // been loaded

    void SaveState();                   // Save/restore address space-specific
    void RestoreState();          // info on a context switch

    // Translate virtual address _vaddr_
    // to physical address _paddr_. _mode_
    // is 0 for Read, 1 for Write.
    ExceptionType Translate(unsigned int vaddr, unsigned int *paddr, int mode);

  private:
    TranslationEntry *pageTable;   // Assume linear page table translation
                                                  // for now!
    unsigned int numPages;                        // Number of pages in the virtual
                                                  // address space

    void InitRegisters();          // Initialize user-level CPU registers,
                                                  // before jumping to user code
   static int mark;

};

#endif // ADDRSPACE_H
```

<u>addrspace.cc:</u>

```
// addrspace.cc
//          Routines to manage address spaces (executing user programs).
//
//          In order to run a user program, you must:
//
//          1. link with the -n -T 0 option
//          2. run coff2noff to convert the object file to Nachos format
//                   (Nachos object code format is essentially just a simpler
//                   version of the UNIX executable object code format)
//          3. load the NOFF file into the Nachos file system
//                   (if you are using the "stub" file system, you
//                   don't need to do this last step)
//
// Copyright (c) 1992-1996 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.


#include "copyright.h"
#include "main.h"
#include "addrspace.h"
#include "machine.h"
#include "noff.h"


int AddrSpace::mark = 0;


//----------------------------------------------------------------------
```

```
// SwapHeader
//      Do little endian to big endian conversion on the bytes in the
//      object file header, in case the file was generated on a little
//      endian machine, and we're now running on a big endian machine.
//----------------------------------------------------------------------

static void
SwapHeader (NoffHeader *noffH)
{
   noffH->noffMagic = WordToHost(noffH->noffMagic);
   noffH->code.size = WordToHost(noffH->code.size);
   noffH->code.virtualAddr = WordToHost(noffH->code.virtualAddr);
   noffH->code.inFileAddr = WordToHost(noffH->code.inFileAddr);
#ifdef RDATA
   noffH->readonlyData.size = WordToHost(noffH->readonlyData.size);
   noffH->readonlyData.virtualAddr =
         WordToHost(noffH->readonlyData.virtualAddr);
   noffH->readonlyData.inFileAddr =
         WordToHost(noffH->readonlyData.inFileAddr);
#endif
   noffH->initData.size = WordToHost(noffH->initData.size);
   noffH->initData.virtualAddr = WordToHost(noffH->initData.virtualAddr);
   noffH->initData.inFileAddr = WordToHost(noffH->initData.inFileAddr);
   noffH->uninitData.size = WordToHost(noffH->uninitData.size);
   noffH->uninitData.virtualAddr = WordToHost(noffH->uninitData.virtualAddr);
   noffH->uninitData.inFileAddr = WordToHost(noffH->uninitData.inFileAddr);

#ifdef RDATA
   DEBUG(dbgAddr, "code = " << noffH->code.size <<
               " readonly = " << noffH->readonlyData.size <<
               " init = " << noffH->initData.size <<
               " uninit = " << noffH->uninitData.size << "\n");
#endif
}

//----------------------------------------------------------------------
// AddrSpace::AddrSpace
//          Create an address space to run a user program.
//          Set up the translation from program memory to physical
//          memory.  For now, this is really simple (1:1), since we are
//          only uniprogramming, and we have a single unsegmented page table
//----------------------------------------------------------------------

AddrSpace::AddrSpace()
{
}

//----------------------------------------------------------------------
// AddrSpace::~AddrSpace
//          Dealloate an address space.
//----------------------------------------------------------------------

AddrSpace::~AddrSpace()
{
  delete pageTable;
}



//----------------------------------------------------------------------
// AddrSpace::Load
//          Load a user program into memory from a file.
//
//          Assumes that the page table has been initialized, and that
//          the object code file is in NOFF format.
//
//          "fileName" is the file containing the object code to load into memory
//----------------------------------------------------------------------

bool
AddrSpace::Load(char *fileName)
{
   OpenFile *executable = kernel->fileSystem->Open(fileName);
   NoffHeader noffH;
   unsigned int size;
   if (executable == NULL) {
            cerr << "Unable to open file " << fileName << "\n";
            return FALSE;
   }

   executable->ReadAt((char *)&noffH, sizeof(noffH), 0);
   if ((noffH.noffMagic != NOFFMAGIC) &&
                        (WordToHost(noffH.noffMagic) == NOFFMAGIC))
            SwapHeader(&noffH);
   ASSERT(noffH.noffMagic == NOFFMAGIC);

#ifdef RDATA
// how big is address space?
   size = noffH.code.size + noffH.readonlyData.size + noffH.initData.size +
       noffH.uninitData.size + UserStackSize;
                             // we need to increase the size
                                                                     // to leave room for the stack
#else
// how big is address space?
   size = noffH.code.size + noffH.initData.size + noffH.uninitData.size
                                   + UserStackSize;       // we need to increase the size
                                                                     // to leave room for the stack
#endif
   numPages = divRoundUp(size, PageSize);
   size = numPages * PageSize;

   ASSERT(numPages <= NumPhysPages);                  // check we're not trying
                                                                     // to run anything too big --
                                                                     // at least until we have
                                                                     // virtual memory

   DEBUG(dbgAddr, "Initializing address space: " << numPages << ", " << size);

   pageTable = new TranslationEntry[numPages];
   for (int i = 0; i < numPages; i++) {
      pageTable[i].virtualPage = i;   // for now, virt page # = phys page #
```

```
    pageTable[i].physicalPage = i + mark;
    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[mark * PageSize], size);

// then, copy in the code and data segments into memory
// Note: this code assumes that virtual address = physical address
  if (noffH.code.size > 0) {
    DEBUG(dbgAddr, "Initializing code segment.");
          DEBUG(dbgAddr, noffH.code.virtualAddr << ", " << noffH.code.size);
    executable->ReadAt(
                &(kernel->machine->mainMemory[noffH.code.virtualAddr + mark * PageSize]),
                        noffH.code.size, noffH.code.inFileAddr);
  }
  if (noffH.initData.size > 0) {
    DEBUG(dbgAddr, "Initializing data segment.");
          DEBUG(dbgAddr, noffH.initData.virtualAddr << ", " << noffH.initData.size);
    executable->ReadAt(
                &(kernel->machine->mainMemory[noffH.initData.virtualAddr + mark * PageSize]),
                        noffH.initData.size, noffH.initData.inFileAddr);
  }

#ifdef RDATA
  if (noffH.readonlyData.size > 0) {
    DEBUG(dbgAddr, "Initializing read only data segment.");
          DEBUG(dbgAddr, noffH.readonlyData.virtualAddr << ", " << noffH.readonlyData.size);
    executable->ReadAt(
                &(kernel->machine->mainMemory[noffH.readonlyData.virtualAddr + mark * PageSize]),
                        noffH.readonlyData.size, noffH.readonlyData.inFileAddr);
  }
#endif
```

```
   mark += numPages;
   delete executable;                      // close file
   return TRUE;                            // success
}

//----------------------------------------------------------------------
// AddrSpace::Execute
//          Run a user program using the current thread
//
//    The program is assumed to have already been loaded into
//    the address space
//
//----------------------------------------------------------------------

void
AddrSpace::Execute()
{

   kernel->currentThread->space = this;

   this->InitRegisters();                  // set the initial register values
   this->RestoreState();                   // load page table register

   kernel->machine->Run();                 // jump to the user progam

   ASSERTNOTREACHED();                             // machine->Run never returns;
                                                   // the address space exits
                                                   // by doing the syscall "exit"
}


//----------------------------------------------------------------------
// AddrSpace::InitRegisters
//          Set the initial values for the user-level register set.
//
//          We write these directly into the "machine" registers, so
//          that we can immediately jump to user code.  Note that these
//              will be saved/restored into the currentThread->userRegisters
//              when this thread is context switched out.
//----------------------------------------------------------------------

void
AddrSpace::InitRegisters()
{
   Machine *machine = kernel->machine;
   int i;

   for (i = 0; i < NumTotalRegs; i++)
                machine->WriteRegister(i, 0);

   // Initial program counter -- must be location of "Start", which
   //  is assumed to be virtual address zero
   machine->WriteRegister(PCReg, 0);

   // Need to also tell MIPS where next instruction is, because
   // of branch delay possibility
   // Since instructions occupy four bytes each, the next instruction
   // after start will be at virtual address four.
   machine->WriteRegister(NextPCReg, 4);

  // Set the stack register to the end of the address space, where we
  // allocated the stack; but subtract off a bit, to make sure we don't
  // accidentally reference off the end!
   machine->WriteRegister(StackReg, numPages * PageSize - 16);
   DEBUG(dbgAddr, "Initializing stack pointer: " << numPages * PageSize - 16);
}

//----------------------------------------------------------------------
// AddrSpace::SaveState
//              On a context switch, save any machine state, specific
//              to this address space, that needs saving.
//
//              For now, don't need to save anything!
//----------------------------------------------------------------------
```

```
void AddrSpace::SaveState()
{}


//----------------------------------------------------------------------
// AddrSpace::RestoreState
//              On a context switch, restore the machine state so that
//              this address space can run.
//
//      For now, tell the machine where to find the page table.
//----------------------------------------------------------------------

void AddrSpace::RestoreState()
{
    kernel->machine->pageTable = pageTable;
    kernel->machine->pageTableSize = numPages;
}



//----------------------------------------------------------------------
// AddrSpace::Translate
//  Translate the virtual address in _vaddr_ to a physical address
//  and store the physical address in _paddr_.
//  The flag _isReadWrite_ is false (0) for read-only access; true (1)
//  for read-write access.
//  Return any exceptions caused by the address translation.
//----------------------------------------------------------------------
ExceptionType
AddrSpace::Translate(unsigned int vaddr, unsigned int *paddr, int isReadWrite)
{
    TranslationEntry *pte;
    int             pfn;
    unsigned int      vpn    = vaddr / PageSize;
    unsigned int      offset = vaddr % PageSize;


    if(vpn >= numPages) {
        return AddressErrorException;
    }


    pte = &pageTable[vpn];


    if(isReadWrite && pte->readOnly) {
        return ReadOnlyException;
    }


    pfn = pte->physicalPage;


    // if the pageFrame is too big, there is something really wrong!
    // An invalid translation was loaded into the page table or TLB.
    if (pfn >= NumPhysPages) {
        DEBUG(dbgAddr, "Illegal physical page " << pfn);
        return BusErrorException;
    }


    pte->use = TRUE;          // set the use, dirty bits


    if(isReadWrite)
        pte->dirty = TRUE;


    *paddr = pfn*PageSize + offset;


    ASSERT((*paddr < MemorySize));


    //cerr << " -- AddrSpace::Translate(): vaddr: " << vaddr <<
    //  ", paddr: " << *paddr << "\n";


    return NoException;
}
```

- We have also implemented the execution of multiple programs simultaneously by creating a list of the user programs given to the '-x' command of nachos and forking a new thread for each of the user program and calling RunUserProg on the forked threads to load them in the memory and add them to the scheduler's ready queue. Also, we add each forked thread to the kernel's pool i.e. listOfForkedThreads which keeps a track of the alive forked threads.

- For this, we have modified nachos/code/threads/main.cc.

main.cc:

```
// main.cc
#define MAIN
#include "copyright.h"
#undef MAIN

#include "main.h"
#include "filesys.h"
#include "openfile.h"
#include "sysdep.h"

// global variables
Kernel *kernel;
Debug *debug;

extern void ThreadTest(void);

List<char*> *listOfUserProgram = new List<char*>();
bool userProgramFlag = FALSE;

//----------------------------------------------------------------------
// Cleanup
//      Delete kernel data structures; called when user hits "ctl-C".
//----------------------------------------------------------------------

static void
Cleanup(int x)
{
    cerr << "\nCleaning up after signal " << x << "\n";
    delete kernel;
}

//-------------------------------------------------------------------
// Constant used by "Copy" and "Print"
//   It is the number of bytes read from the Unix file (for Copy)
//   or the Nachos file (for Print) by each read operation
//-------------------------------------------------------------------
static const int TransferSize = 128;


#ifndef FILESYS_STUB
//----------------------------------------------------------------------
// Copy
//      Copy the contents of the UNIX file "from" to the Nachos file "to"
//----------------------------------------------------------------------

static void
Copy(char *from, char *to)
{
    int fd;
    OpenFile* openFile;
    int amountRead, fileLength;
    char *buffer;

// Open UNIX file
    if ((fd = OpenForReadWrite(from,FALSE)) < 0) {
        printf("Copy: couldn't open input file %s\n", from);
        return;
    }

// Figure out length of UNIX file
    Lseek(fd, 0, 2);
    fileLength = Tell(fd);
    Lseek(fd, 0, 0);
```

```
// Create a Nachos file of the same length
    DEBUG('f', "Copying file " << from << " of size " << fileLength <<  " to file " << to);
    if (!kernel->fileSystem->Create(to, fileLength)) {   // Create Nachos file
        printf("Copy: couldn't create output file %s\n", to);
        Close(fd);
        return;
    }

    openFile = kernel->fileSystem->Open(to);
    ASSERT(openFile != NULL);

// Copy the data in TransferSize chunks
    buffer = new char[TransferSize];
    while ((amountRead=ReadPartial(fd, buffer, sizeof(char)*TransferSize)) > 0)
        openFile->Write(buffer, amountRead);
    delete [] buffer;

// Close the UNIX and the Nachos files
    delete openFile;
    Close(fd);
}

#endif // FILESYS_STUB

//----------------------------------------------------------------------
// Print
//      Print the contents of the Nachos file "name".
//----------------------------------------------------------------------

void
Print(char *name)
{
    OpenFile *openFile;
    int i, amountRead;
    char *buffer;

    if ((openFile = kernel->fileSystem->Open(name)) == NULL) {
        printf("Print: unable to open file %s\n", name);
        return;
    }

    buffer = new char[TransferSize];
    while ((amountRead = openFile->Read(buffer, TransferSize)) > 0)
        for (i = 0; i < amountRead; i++)
            printf("%c", buffer[i]);
    delete [] buffer;

    delete openFile;            // close the Nachos file
    return;
}

//----------------------------------------------------------------------
// RunUserProg
//      Run the user program in the given file.
//----------------------------------------------------------------------

void
RunUserProg(void *filename) {
    AddrSpace *space = new AddrSpace;
    ASSERT(space != (AddrSpace *)NULL);

  //  char *swapFile = kernel->swapFileName;

    if (space->Load((char*)filename)) {  // load the program into the space
    space->Execute();        // run the program
  }

  // if(space->Load((char*)swapFile))
  // {
  //   space->Execute();
  // }
  ASSERTNOTREACHED();
}

//----------------------------------------------------------------------
// main
//      Bootstrap the operating system kernel.
//
//      Initialize kernel data structures
//      Call some test routines
//      Call "Run" to start an initial user program running
//
//      "argc" is the number of command line arguments (including the name
//                      of the command) -- ex: "nachos -d +" -> argc = 3
//      "argv" is an array of strings, one for each command line argument
//                      ex: "nachos -d +" -> argv = {"nachos", "-d", "+"}
//----------------------------------------------------------------------

int
main(int argc, char **argv)
{
  int i;
  char *debugArg = "";
  char *userProgName = NULL;        // default is not to execute a user prog
  bool threadTestFlag = false;
  bool consoleTestFlag = false;
  bool networkTestFlag = false;
#ifndef FILESYS_STUB
  char *copyUnixFileName = NULL;    // UNIX file to be copied into Nachos
  char *copyNachosFileName = NULL;  // name of copied file in Nachos
  char *printFileName = NULL;
  char *removeFileName = NULL;
  bool dirListFlag = false;
  bool dumpFlag = false;

#endif //FILESYS_STUB

  // some command line arguments are handled here.
  // those that set kernel parameters are handled in
  // the Kernel constructor
  for (i = 1; i < argc; i++) {
   // if(argc > 3)
   // {
   //   userProgramFlag = TRUE;
   // }
    if (strcmp(argv[i], "-d") == 0) {
       ASSERT(i + 1 < argc);   // next argument is debug string
      debugArg = argv[i + 1];
       i++;
     }
     else if (strcmp(argv[i], "-z") == 0) {
      cout << copyright << "\n";
     }
     else if (strcmp(argv[i], "-x") == 0) {
       ASSERT(i + 1 < argc);
   userProgramFlag = TRUE;
       userProgName = argv[i + 1];
   listOfUserProgram->Append(userProgName);   // appending the filename into the list of filenames
       i++;
```

```
        }
        else if (strcmp(argv[i], "-K") == 0) {
            threadTestFlag = TRUE;
        }
        else if (strcmp(argv[i], "-C") == 0) {
            consoleTestFlag = TRUE;
        }
        else if (strcmp(argv[i], "-N") == 0) {
            networkTestFlag = TRUE;
        }
#ifndef FILESYS_STUB
        else if (strcmp(argv[i], "-cp") == 0) {
            ASSERT(i + 2 < argc);
            copyUnixFileName = argv[i + 1];
            copyNachosFileName = argv[i + 2];
            i += 2;
        }
        else if (strcmp(argv[i], "-p") == 0) {
            ASSERT(i + 1 < argc);
            printFileName = argv[i + 1];
            i++;
        }
        else if (strcmp(argv[i], "-r") == 0) {
            ASSERT(i + 1 < argc);
            removeFileName = argv[i + 1];
            i++;
        }
        else if (strcmp(argv[i], "-l") == 0) {
            dirListFlag = true;
        }
        else if (strcmp(argv[i], "-D") == 0) {
            dumpFlag = true;
        }
#endif //FILESYS_STUB
        else if (strcmp(argv[i], "-u") == 0) {
          cout << "Partial usage: nachos [-z -d debugFlags]\n";
          cout << "Partial usage: nachos [-x programName]\n";
            cout << "Partial usage: nachos [-K] [-C] [-N]\n";
#ifndef FILESYS_STUB
          cout << "Partial usage: nachos [-cp UnixFile NachosFile]\n";
          cout << "Partial usage: nachos [-p fileName] [-r fileName]\n";
          cout << "Partial usage: nachos [-l] [-D]\n";
#endif //FILESYS_STUB
        }

    }
    debug = new Debug(debugArg);

    DEBUG(dbgThread, "Entering main");

    kernel = new Kernel(argc, argv);

    kernel->Initialize();

    CallOnUserAbort(Cleanup);                           // if user hits ctl-C

    // at this point, the kernel is ready to do something
    // run some tests, if requested
    if (threadTestFlag) {
      //kernel->ThreadSelfTest();  // test threads and synchronization
      ThreadTest();
    }
    if (consoleTestFlag) {
      kernel->ConsoleTest();   // interactive test of the synchronized console
    }
    if (networkTestFlag) {
      kernel->NetworkTest();   // two-machine test of the network
    }

#ifndef FILESYS_STUB
    if (removeFileName != NULL) {
      kernel->fileSystem->Remove(removeFileName);
    }
    if (copyUnixFileName != NULL && copyNachosFileName != NULL) {
      Copy(copyUnixFileName,copyNachosFileName);
    }
    if (dumpFlag) {
      kernel->fileSystem->Print();
    }
    if (dirListFlag) {
      kernel->fileSystem->List();
    }
    if (printFileName != NULL) {
      Print(printFileName);
    }
#endif // FILESYS_STUB

    // finally, run an initial user program if requested to do so


    if(userProgramFlag)
    {
      if (!listOfUserProgram->IsEmpty()) {
     //  std::cout<<"Inside user program main.cc\n";
        ListIterator<char*> *iterator = new ListIterator<char*>(listOfUserProgram);
        for (; !iterator->IsDone(); iterator->Next()) {
          char  *threadName =  iterator->Item();
          Thread *thread = new Thread(threadName);
          std::cout<<"Forking thread : " << thread->getThreadId() << ", name:" <<thread->getName() << endl;

          kernel->listOfForkedThreads->Append(thread);   //adding the thread into list
          const void *voidArg = iterator->Item();
          thread->Fork((VoidFunctionPtr) RunUserProg, (void *)voidArg);
        }
     //  RunUserProg(userProgName);
      }
    }
    // NOTE: if the procedure "main" returns, then the program "nachos"
    // will exit (as any other normal program would).  But there may be
    // other threads on the ready list (started in SelfTest).
    // We switch to those threads by saying that the "main" thread
    // is finished, preventing it from returning.
    kernel->currentThread->Finish();

    ASSERTNOTREACHED();
}
```

- We also modified nachos/code/machine/machine.h and changed the page size to 256 and the number of physical pages as 256 to run multiple programs in the contiguous memory.

machine.h

```
// machine.h

//      Data structures for simulating the execution of user programs

//      running on top of Nachos.

//
```

```
//      User programs are loaded into "mainMemory"; to Nachos,
//      this looks just like an array of bytes.  Of course, the Nachos
//      kernel is in memory too -- but as in most machines these days,
//      the kernel is loaded into a separate memory region from user
//      programs, and accesses to kernel memory are not translated or paged.
//
//      In Nachos, user programs are executed one instruction at a time,
//      by the simulator.  Each memory reference is translated, checked
//      for errors, etc.
//
//  DO NOT CHANGE EXCEPT AS NOTED BELOW -- part of the machine emulation
//
// Copyright (c) 1992-1996 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#ifndef MACHINE_H
#define MACHINE_H

#include "copyright.h"
#include "utility.h"
#include "translate.h"

// Definitions related to the size, and format of user memory

const int PageSize = 256;               // set the page size equal to
                                        // the disk sector size, for simplicity

//
// You are allowed to change this value.
// Doing so will change the number of pages of physical memory
// available on the simulated machine.
//
const int NumPhysPages = 256;

const int MemorySize = (NumPhysPages * PageSize);
const int TLBSize = 4;                  // if there is a TLB, make it small
```

```
enum ExceptionType { NoException,           // Everything ok!
                     SyscallException,      // A program executed a system call.
                     PageFaultException,    // No valid translation found
                     ReadOnlyException,     // Write attempted to page marked
                                            // "read-only"
                     BusErrorException,     // Translation resulted in an
                                            // invalid physical address
                     AddressErrorException, // Unaligned reference or one that
                                            // was beyond the end of the
                                            // address space
                     OverflowException,     // Integer overflow in add or sub.
                     IllegalInstrException, // Unimplemented or reserved instr.

                     NumExceptionTypes
};

// User program CPU state.  The full set of MIPS registers, plus a few
// more because we need to be able to start/stop a user program between
// any two instructions (thus we need to keep track of things like load
// delay slots, etc.)

#define StackReg        29      // User's stack pointer
#define RetAddrReg      31      // Holds return address for procedure calls
#define NumGPRegs       32      // 32 general purpose registers on MIPS
#define HiReg           32      // Double register to hold multiply result
#define LoReg           33
#define PCReg           34      // Current program counter
#define NextPCReg       35      // Next program counter (for branch delay)
#define PrevPCReg       36      // Previous program counter (for debugging)
#define LoadReg         37      // The register target of a delayed load.
#define LoadValueReg    38      // The value to be loaded by a delayed load.
#define BadVAddrReg     39      // The failing virtual address on an exception

#define NumTotalRegs    40

// The following class defines the simulated host workstation hardware, as
// seen by user programs -- the CPU registers, main memory, etc.
// User programs shouldn't be able to tell that they are running on our
// simulator or on the real hardware, except
//      we don't support floating point instructions
//      the system call interface to Nachos is not the same as UNIX
//        (10 system calls in Nachos vs. 200 in UNIX!)
// If we were to implement more of the UNIX system calls, we ought to be
// able to run Nachos on top of Nachos!
//
// The procedures in this class are defined in machine.cc, mipssim.cc, and
// translate.cc.

class Instruction;
class Interrupt;

class Machine {
  public:
    Machine(bool debug);      // Initialize the simulation of the hardware
                              // for running user programs
    ~Machine();               // De-allocate the data structures

// Routines callable by the Nachos kernel
    void Run();               // Run a user program

    int ReadRegister(int num);      // read the contents of a CPU register

    void WriteRegister(int num, int value);
                              // store a value into a CPU register

// Data structures accessible to the Nachos kernel -- main memory and the
// page table/TLB.
//
// Note that *all* communication between the user program and the kernel
// are in terms of these data structures (plus the CPU registers).

    char *mainMemory;         // physical memory to store user program,
                              // code and data, while executing
// NOTE: the hardware translation of virtual addresses in the user program
// to physical addresses (relative to the beginning of "mainMemory")
// can be controlled by one of:
//      a traditional linear page table
//      a software-loaded translation lookaside buffer (tlb) -- a cache of
//        mappings of virtual page #'s to physical page #'s
//
// If "tlb" is NULL, the linear page table is used
// If "tlb" is non-NULL, the Nachos kernel is responsible for managing
//      the contents of the TLB.  But the kernel can use any data structure
//      it wants (eg, segmented paging) for handling TLB cache misses.
//
// For simplicity, both the page table pointer and the TLB pointer are
// public.  However, while there can be multiple page tables (one per address
// space, stored in memory), there is only one TLB (implemented in hardware).
// Thus the TLB pointer should be considered as *read-only*, although
// the contents of the TLB are free to be modified by the kernel software.

  TranslationEntry *tlb;                // this pointer should be considered
                                        //      "read-only" to Nachos kernel code

  TranslationEntry *pageTable;
  unsigned int pageTableSize;

  bool ReadMem(int addr, int size, int* value);
  bool WriteMem(int addr, int size, int value);
                                        // Read or write 1, 2, or 4 bytes of virtual
                                        // memory (at addr).  Return FALSE if a
                                        // correct translation couldn't be found.
 private:

// Routines internal to the machine simulation -- DO NOT call these directly
  void DelayedLoad(int nextReg, int nextVal);
                                        // Do a pending delayed load (modifying a reg)

  void OneInstruction(Instruction *instr);
```

```
                                        // Run one instruction of a user program.



    ExceptionType Translate(int virtAddr, int* physAddr, int size,bool writing);
                                        // Translate an address, and check for
                                        // alignment.  Set the use and dirty bits in
                                        // the translation entry appropriately,
                                        // and return an exception code if the
                                        // translation couldn't be completed.

    void RaiseException(ExceptionType which, int badVAddr);
                                        // Trap to the Nachos kernel, because of a
                                        // system call or other exception.

    void Debugger();                    // invoke the user program debugger
    void DumpState();                   // print the user CPU and memory state


  // Internal data structures

    int registers[NumTotalRegs]; // CPU registers, for executing user programs

    bool singleStep;                    // drop back into the debugger after each
                                        // simulated instruction
    int runUntilTime;                   // drop back into the debugger when simulated
                                        // time reaches this value

    friend class Interrupt;             // calls DelayedLoad()
  };

  extern void ExceptionHandler(ExceptionType which);
                                        // Entry point into Nachos for handling
                                        // user system calls and exceptions
                                        // Defined in exception.cc
```

```
// Routines for converting Words and Short Words to and from the
// simulated machine's format of little endian.  If the host machine
// is little endian (DEC and Intel), these end up being NOPs.
//
// What is stored in each format:
//      host byte ordering:
//         kernel data structures
//         user registers
//      simulated machine byte ordering:
//         contents of main memory

unsigned int WordToHost(unsigned int word);
unsigned short ShortToHost(unsigned short shortword);
unsigned int WordToMachine(unsigned int word);
unsigned short ShortToMachine(unsigned short shortword);

#endif // MACHINE_H
```

2. Message Passing –

- To implement message passing we have first implemented a buffer class to implement the message buffer between two threads. The members of this class are a unique buffer id, sender thread, receiver thread, status of the buffer and the list of messages with a limit of 10 messages per buffer.
- We have implemented a buffer pool at kernel level that keeps a track of all existing buffers as well as a bitmap for the allocation of unique buffer ids. The limit of kernel buffers that can be created is 100.
- We have implemented the getter setter methods for all the members of the Buffer class.
- Our buffer class has two parameterized constructors for its initialization, the first constructor takes a sender thread and a receiver thread and creates a new buffer with the next available id from the kernel bitmap.
- The second constructor takes a sender thread and a receiver thread as well as an existing buffer id to create new buffer instance with same buffer id.
- We have added files nachos/code/userprog/buffer.h and files nachos/code/userprog/buffer.cc

buffer.h:

```
#ifndef BUFFER_H
#define BUFFER_H

class Thread;

#include "list.h"
#include <string.h>

class Buffer{
private:
  int buffer_id;
  Thread *sender;
  Thread *receiver;
  bool isActiveStatus;
  List<std::string> *listOfMessages;   //list of messages specific to buffer


public:
  Buffer(Thread *Tsender, Thread *Treceiver);
  Buffer(Thread *Tsender, Thread *Treceiver, int buffer_id);
  ~Buffer();

  int getBufferId() {return buffer_id;}
  Thread* getSender() {return sender;}
  Thread* getReceiver() {return receiver;}
  bool getActiveStatus() {return isActiveStatus;}
  List<std::string>* getMessagesList() {return listOfMessages;}

  void setBufferId(int bufferId) {buffer_id = bufferId ;}
  void setSender(Thread *sender) {sender = sender;}
  void setReceiver(Thread *receiver) {receiver = receiver;}
  void setActiveStatus(bool activeFlag) {isActiveStatus = activeFlag;}

};

#endif
```

buffer.cc:

```
#include "main.h"
#include "buffer.h"

Buffer::Buffer(Thread *Tsender, Thread *Treceiver)
{
  buffer_id = kernel->bitmapForBuffer->FindAndSet();
  sender = Tsender;
  receiver = Treceiver;
  isActiveStatus = true;
  listOfMessages = new List<std::string>();
}

Buffer::Buffer(Thread *Tsender, Thread *Treceiver, int buffer_id)
{
  buffer_id = buffer_id;
  sender = Tsender;
  receiver = Treceiver;
  isActiveStatus = true;
  listOfMessages = new List<std::string>();
}

Buffer::~Buffer()
{
  delete listOfMessages;
}
```

- The kernel level bufferPool is a map of integer i.e the buffer ID and a list of buffers using this ID.
- To implement and initialize the bufferPool, message limit as well as the Bitmap i.e. bitmapForBuffer we have modified nachos/code/threads/kernel.h and nachos/code/threads/kernel.cc

kernel.h:

```
// kernel.h

//              Global variables for the Nachos kernel.

//

// Copyright (c) 1992-1996 The Regents of the University of California.

// All rights reserved.  See copyright.h for copyright notice and limitation

// of liability and disclaimer of warranty provisions.


#ifndef KERNEL_H

#define KERNEL_H

class Buffer;

#include "copyright.h"

#include "debug.h"

#include "utility.h"

#include "thread.h"

#include "scheduler.h"

#include "interrupt.h"

#include "stats.h"

#include "alarm.h"

#include "filesys.h"
```

```
#include "machine.h"

#include "bitmap.h"

#include <stdio.h>

#include <map>


class PostOfficeInput;

class PostOfficeOutput;

class SynchConsoleInput;

class SynchConsoleOutput;

class SynchDisk;


class Kernel {

 public:

   Kernel(int argc, char **argv);

                                         // Interpret command line arguments

   ~Kernel();                         // deallocate the kernel


   void Initialize();                 // initialize the kernel -- separated

                                         // from constructor because

                                         // refers to "kernel" as a global


   void ThreadSelfTest();             // self test of threads and synchronization


   void ConsoleTest();      // interactive console self test


   void NetworkTest();      // interactive 2-machine network test


// These are public for notational convenience; really,

// they're global variables used everywhere.


   Thread *currentThread;             // the thread holding the CPU

   Scheduler *scheduler;              // the ready list

   Interrupt *interrupt;              // interrupt status

   Statistics *stats;                 // performance metrics

   Alarm *alarm;                      // the software alarm clock

   Machine *machine;        // the simulated CPU

   SynchConsoleInput *synchConsoleIn;
   SynchConsoleOutput *synchConsoleOut;
   SynchDisk *synchDisk;
   FileSystem *fileSystem;
   PostOfficeInput *postOfficeIn;
   PostOfficeOutput *postOfficeOut;

   int hostName;            // machine identifier

   Bitmap *bitmapForBuffer;  //for buffer pool

   map<int, List<Buffer*>* > bufferPool;  //stores the mapping of buffer id to list of receving and sending buffers

   const int MessageLimit = 10;  //process can send only 10 messages

   const std::string DummyMessage = "Dummy Message";
   const std::string ErrorMessage = "Error Message";

   List<Thread*> *listOfForkedThreads; //to keep track of the threads created

   void deleteBufferEntryFromKernel(int buffer_id);

  private:
   bool randomSlice;                    // enable pseudo-random time slicing
   bool debugUserProg;        // single step user program
   double reliability;        // likelihood messages are dropped
   char *consoleIn;          // file to read console input from
   char *consoleOut;          // file to send console output to
#ifndef FILESYS_STUB
   bool formatFlag;         // format the disk if this is true
#endif
};



#endif // KERNEL_H
```

<u>kernel.cc:</u>

```
// kernel.cc
//              Initialization and cleanup routines for the Nachos kernel.
```

```
//
// Copyright (c) 1992-1996 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#include "copyright.h"
#include "debug.h"
#include "main.h"
#include "kernel.h"
#include "sysdep.h"
#include "synch.h"
#include "synchlist.h"
#include "libtest.h"
#include "string.h"
#include "synchconsole.h"
#include "synchdisk.h"
#include "post.h"

//----------------------------------------------------------------------
// Kernel::Kernel
//              Interpret command line arguments in order to determine flags
//              for the initialization (see also comments in main.cc)
//----------------------------------------------------------------------

Kernel::Kernel(int argc, char **argv)
{
    randomSlice = FALSE;
    debugUserProg = FALSE;
    consoleIn = NULL;          // default is stdin
    consoleOut = NULL;         // default is stdout
#ifndef FILESYS_STUB
    formatFlag = FALSE;
#endif
    reliability = 1;           // network reliability, default is 1.0
    hostName = 0;              // machine id, also UNIX socket name
                               // 0 is the default machine id
    for (int i = 1; i < argc; i++) {
```

```
    if (strcmp(argv[i], "-rs") == 0) {

            ASSERT(i + 1 < argc);

            RandomInit(atoi(argv[i + 1]));// initialize pseudo-random

                                                  // number generator

            randomSlice = TRUE;

            i++;

    } else if (strcmp(argv[i], "-s") == 0) {

      debugUserProg = TRUE;

          } else if (strcmp(argv[i], "-ci") == 0) {

            ASSERT(i + 1 < argc);

            consoleIn = argv[i + 1];

            i++;

          } else if (strcmp(argv[i], "-co") == 0) {

            ASSERT(i + 1 < argc);

            consoleOut = argv[i + 1];

            i++;

#ifndef FILESYS_STUB

          } else if (strcmp(argv[i], "-f") == 0) {

            formatFlag = TRUE;

#endif

    } else if (strcmp(argv[i], "-n") == 0) {

      ASSERT(i + 1 < argc);   // next argument is float

      reliability = atof(argv[i + 1]);

      i++;

    } else if (strcmp(argv[i], "-m") == 0) {

      ASSERT(i + 1 < argc);   // next argument is int

      hostName = atoi(argv[i + 1]);

      i++;

    } else if (strcmp(argv[i], "-u") == 0) {

      cout << "Partial usage: nachos [-rs randomSeed]\n";

            cout << "Partial usage: nachos [-s]\n";

      cout << "Partial usage: nachos [-ci consoleIn] [-co consoleOut]\n";

#ifndef FILESYS_STUB

            cout << "Partial usage: nachos [-nf]\n";

#endif

      cout << "Partial usage: nachos [-n #] [-m #]\n";

          }
```

```
  }
}

//----------------------------------------------------------------------
// Kernel::Initialize
//          Initialize Nachos global data structures.  Separate from the
//          constructor because some of these refer to earlier initialized
//          data via the "kernel" global variable.
//----------------------------------------------------------------------

void
Kernel::Initialize()
{
  // We didn't explicitly allocate the current thread we are running in.
  // But if it ever tries to give up the CPU, we better have a Thread
  // object to save its state.
  currentThread = new Thread("main");
  currentThread->setStatus(RUNNING);

  stats = new Statistics();                 // collect statistics
  interrupt = new Interrupt;                // start up interrupt handling
  scheduler = new Scheduler();   // initialize the ready queue
  alarm = new Alarm(randomSlice);           // start up time slicing
  machine = new Machine(debugUserProg);
  synchConsoleIn = new SynchConsoleInput(consoleIn); // input from stdin
  synchConsoleOut = new SynchConsoleOutput(consoleOut); // output to stdout
  synchDisk = new SynchDisk();   //
#ifdef FILESYS_STUB
  fileSystem = new FileSystem();
#else
  fileSystem = new FileSystem(formatFlag);
#endif // FILESYS_STUB
  postOfficeIn = new PostOfficeInput(10);
  postOfficeOut = new PostOfficeOutput(reliability);

  bitmapForBuffer = new Bitmap(128);
  listOfForkedThreads = new List<Thread*>();
```

```
    interrupt->Enable();
}

//----------------------------------------------------------------------
// Kernel::~Kernel
//              Nachos is halting.  De-allocate global data structures.
//----------------------------------------------------------------------

Kernel::~Kernel()
{
    delete stats;
    delete interrupt;
    delete scheduler;
    delete alarm;
    delete machine;
    delete synchConsoleIn;
    delete synchConsoleOut;
    delete synchDisk;
    delete fileSystem;
    delete postOfficeIn;
    delete postOfficeOut;

    Exit(0);
}

void Kernel::deleteBufferEntryFromKernel(int buffer_id)
{
  if(buffer_id != -1)
  {
    bitmapForBuffer->Clear(buffer_id);
    std::map<int, List<Buffer*>* >::iterator it;
    it = bufferPool.find(buffer_id);

    if(it != kernel->bufferPool.end())  //the entry is found
    {
      if(!it->second->IsEmpty())
      {
        bufferPool.erase(it);
      }
    }
  }

}



//----------------------------------------------------------------------
// Kernel::ThreadSelfTest
//      Test threads, semaphores, synchlists
//----------------------------------------------------------------------

void
Kernel::ThreadSelfTest() {
   Semaphore *semaphore;
   SynchList<int> *synchList;

   LibSelfTest();                // test library routines
   currentThread->SelfTest();    // test thread switching

                                 // test semaphore operation
   semaphore = new Semaphore("test", 0);
   semaphore->SelfTest();
   delete semaphore;

                                 // test locks, condition variables
                                 // using synchronized lists
   synchList = new SynchList<int>;
   synchList->SelfTest(9);
   delete synchList;

}

//----------------------------------------------------------------------
// Kernel::ConsoleTest
//      Test the synchconsole
//----------------------------------------------------------------------

void
Kernel::ConsoleTest() {
   char ch;

   cout << "Testing the console device.\n"
      << "Typed characters will be echoed, until ^D is typed.\n"
      << "Note newlines are needed to flush input through UNIX.\n";
   cout.flush();

   do {
      ch = synchConsoleIn->GetChar();
      if(ch != EOF) synchConsoleOut->PutChar(ch);   // echo it!
   } while (ch != EOF);

   cout << "\n";

}

//----------------------------------------------------------------------
// Kernel::NetworkTest
//      Test whether the post office is working. On machines #0 and #1, do:
//
//      1. send a message to the other machine at mail box #0
//      2. wait for the other machine's message to arrive (in our mailbox #0)
//      3. send an acknowledgment for the other machine's message
//      4. wait for an acknowledgement from the other machine to our
//         original message
//
//  This test works best if each Nachos machine has its own window
//----------------------------------------------------------------------

void
Kernel::NetworkTest() {
```

```
if (hostName == 0 || hostName == 1) {
  // if we're machine 1, send to 0 and vice versa
  int farHost = (hostName == 0 ? 1 : 0);
  PacketHeader outPktHdr, inPktHdr;
  MailHeader outMailHdr, inMailHdr;
  char *data = "Hello there!";
  char *ack = "Got it!";
  char buffer[MaxMailSize];

  // construct packet, mail header for original message
  // To: destination machine, mailbox 0
  // From: our machine, reply to: mailbox 1
  outPktHdr.to = farHost;
  outMailHdr.to = 0;
  outMailHdr.from = 1;
  outMailHdr.length = strlen(data) + 1;

  // Send the first message
  postOfficeOut->Send(outPktHdr, outMailHdr, data);

  // Wait for the first message from the other machine
  postOfficeIn->Receive(0, &inPktHdr, &inMailHdr, buffer);
  cout << "Got: " << buffer << " : from " << inPktHdr.from << ", box "
                      << inMailHdr.from << "\n";
  cout.flush();

  // Send acknowledgement to the other machine (using "reply to" mailbox
  // in the message that just arrived
  outPktHdr.to = inPktHdr.from;
  outMailHdr.to = inMailHdr.from;
  outMailHdr.length = strlen(ack) + 1;
  postOfficeOut->Send(outPktHdr, outMailHdr, ack);

  // Wait for the ack from the other machine to the first message we sent
        postOfficeIn->Receive(1, &inPktHdr, &inMailHdr, buffer);
  cout << "Got: " << buffer << " : from " << inPktHdr.from << ", box "
                      << inMailHdr.from << "\n";
        cout.flush();
    }

    // Then we're done!
}
```

- Every thread has its own buffer pool i.e. tBufferPool which is a map of an integer i.e the buffer ID and the buffer corresponding to it.
- Also, we have added a messageCount that keeps a track of how many messages a thread has sent.
- We have added a unique thread_id in the thread class and a global int counter to ensure that the id is unique.
- The files modified for this are nachos/code/threads/thread.h and nachos/code/threads/thread.cc

thread.h:

```
// thread.h
//	Data structures for managing threads.  A thread represents
//	sequential execution of code within a program.
//	So the state of a thread includes the program counter,
//	the processor registers, and the execution stack.
//
//	Note that because we allocate a fixed size stack for each
//	thread, it is possible to overflow the stack -- for instance,
//	by recursing to too deep a level.  The most common reason
//	for this occuring is allocating large data structures
//	on the stack.  For instance, this will cause problems:
//
//		void foo() { int buf[1000]; ...}
//
//	Instead, you should allocate all data structures dynamically:
//
//		void foo() { int *buf = new int[1000]; ...}
//
//
//	Bad things happen if you overflow the stack, and in the worst
//	case, the problem may not be caught explicitly.  Instead,
//	the only symptom may be bizarre segmentation faults.  (Of course,
//	other problems can cause seg faults, so that isn't a sure sign
//	that your thread stacks are too small.)
//
//	One thing to try if you find yourself with seg faults is to
```

```
//          increase the size of thread stack -- ThreadStackSize.
//
//          In this interface, forking a thread takes two steps.
//          We must first allocate a data structure for it: "t = new Thread".
//          Only then can we do the fork: "t->fork(f, arg)".
//
// Copyright (c) 1992-1996 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#ifndef THREAD_H
#define THREAD_H

class Buffer;

#include "copyright.h"
#include "utility.h"
#include "sysdep.h"

#include "machine.h"
#include "addrspace.h"
#include <map>


// CPU register state to be saved on context switch.
// The x86 needs to save only a few registers,
// SPARC and MIPS needs to save 10 registers,
// the Snake needs 18,
// and the RS6000 needs to save 75 (!)
// For simplicity, I just take the maximum over all architectures.

#define MachineStateSize 75


// Size of the thread's private execution stack.
// WATCH OUT IF THIS ISN'T BIG ENOUGH!!!!!
const int StackSize = (8 * 1024);   // in words
```

```
// Thread state
enum ThreadStatus { JUST_CREATED, RUNNING, READY, BLOCKED };


// The following class defines a "thread control block" -- which
// represents a single thread of execution.
//
//  Every thread has:
//     an execution stack for activation records ("stackTop" and "stack")
//     space to save CPU registers while not running ("machineState")
//     a "status" (running/ready/blocked)
//
//  Some threads also belong to a user address space; threads
//  that only run in the kernel have a NULL address space.

class Thread {
 private:
   // NOTE: DO NOT CHANGE the order of these first two members.
   // THEY MUST be in this position for SWITCH to work.
   int *stackTop;                          // the current stack pointer
   void *machineState[MachineStateSize];  // all registers except for stackTop

 public:
   Thread(char* debugName);               // initialize a Thread
   ~Thread();                                  // deallocate a Thread
                                               // NOTE -- thread being deleted
                                               // must not be running when delete
                                               // is called

   // basic thread operations

   void Fork(VoidFunctionPtr func, void *arg);
                                          // Make thread run (*func)(arg)
   void Yield();                    // Relinquish the CPU if any
                                          // other thread is runnable
```

```
  void Sleep(bool finishing); // Put the thread to sleep and
                                      // relinquish the processor
  void Begin();               // Startup code for the thread
  void Finish();              // The thread is done executing

  void CheckOverflow();       // Check if thread stack has overflowed
  void setStatus(ThreadStatus st) { status = st; }
  char* getName() { return (name); }
  void Print() { cout << name; }
  void SelfTest();            // test whether thread impl is working

  int getThreadId() {return thread_id;}
  void setThreadId(int thread_id) {thread_id = thread_id ;}

  static int threadCount;

  int getMessageCount() {return messageCount;}
  void incrementMessageCount() {messageCount++;}

  map<int, Buffer*> tBufferPool; //stores the list of buffers, the thread is in communication with


 private:
  // some of the private data for this class is listed above

  int *stack;                 // Bottom of the stack
                                      // NULL if this is the main thread
                                      // (If NULL, don't deallocate stack)
  ThreadStatus status;        // ready, running or blocked
  char* name;
  int thread_id;

  int messageCount = 0;

  void StackAllocate(VoidFunctionPtr func, void *arg);
                                      // Allocate a stack for thread.
                                      // Used internally by Fork()
```

```
// A thread running a user program actually has *two* sets of CPU registers --
// one for its state while executing user code, one for its state
// while executing kernel code.

    int userRegisters[NumTotalRegs];        // user-level CPU register state

  public:
    void SaveUserState();                   // save user-level register state
    void RestoreUserState();                // restore user-level register state

    AddrSpace *space;                       // User code this thread is running.
};

// external function, dummy routine whose sole job is to call Thread::Print
extern void ThreadPrint(Thread *thread);

// Magical machine-dependent routines, defined in switch.s

extern "C" {
// First frame on thread execution stack;
//          call ThreadBegin
//          call "func"
//          (when func returns, if ever) call ThreadFinish()
void ThreadRoot();

// Stop running oldThread and start running newThread
void SWITCH(Thread *oldThread, Thread *newThread);
}

#endif // THREAD_H
```

thread.cc:

```
// thread.cc
//          Routines to manage threads.  These are the main operations:
//
//          Fork -- create a thread to run a procedure concurrently
//                    with the caller (this is done in two steps -- first
//                  allocate the Thread object, then call Fork on it)
//          Begin -- called when the forked procedure starts up, to turn
//                  interrupts on and clean up after last thread
//          Finish -- called when the forked procedure finishes, to clean up
//          Yield -- relinquish control over the CPU to another ready thread
//          Sleep -- relinquish control over the CPU, but thread is now blocked.
//                  In other words, it will not run again, until explicitly
//                  put back on the ready queue.
//
// Copyright (c) 1992-1996 The Regents of the University of California.
// All rights reserved.  See copyright.h for copyright notice and limitation
// of liability and disclaimer of warranty provisions.

#include "copyright.h"
#include "thread.h"
#include "switch.h"
#include "synch.h"
#include "sysdep.h"

// this is put at the top of the execution stack, for detecting stack overflows
const int STACK_FENCEPOST = 0xdedbeef;

int Thread::threadCount = 0;

//----------------------------------------------------------------------
// Thread::Thread
//          Initialize a thread control block, so that we can then call
//          Thread::Fork.
//
//          "threadName" is an arbitrary string, useful for debugging.
//----------------------------------------------------------------------

Thread::Thread(char* threadName)
{
   name = threadName;
   stackTop = NULL;
   stack = NULL;
```

```
    status = JUST_CREATED;
    setThreadId(threadCount);
    threadCount++;
    for (int i = 0; i < MachineStateSize; i++) {
		machineState[i] = NULL;		// not strictly necessary, since
					// new thread ignores contents
					// of machine registers
    }
    space = NULL;
}


//----------------------------------------------------------------------
// Thread::~Thread
// 	De-allocate a thread.
//
// 	NOTE: the current thread *cannot* delete itself directly,
//	since it is still running on the stack that we need to delete.
//
//      NOTE: if this is the main thread, we can't delete the stack
//      because we didn't allocate it -- we got it automatically
//      as part of starting up Nachos.
//----------------------------------------------------------------------

Thread::~Thread()
{
    DEBUG(dbgThread, "Deleting thread: " << name);

    ASSERT(this != kernel->currentThread);
    if (stack != NULL)
	DeallocBoundedArray((char *) stack, StackSize * sizeof(int));
}

//----------------------------------------------------------------------
// Thread::Fork
// 	Invoke (*func)(arg), allowing caller and callee to execute
//	concurrently.
```

```
//
//          NOTE: although our definition allows only a single argument
//          to be passed to the procedure, it is possible to pass multiple
//          arguments by making them fields of a structure, and passing a pointer
//          to the structure as "arg".
//
//          Implemented as the following steps:
//                    1. Allocate a stack
//                    2. Initialize the stack so that a call to SWITCH will
//                    cause it to run the procedure
//                    3. Put the thread on the ready queue
//
//          "func" is the procedure to run concurrently.
//          "arg" is a single argument to be passed to the procedure.
//----------------------------------------------------------------------

void
Thread::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;

    DEBUG(dbgThread, "Forking thread: " << name << " f(a): " << (int) func << " " << arg);

    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRun(this);  // ReadyToRun assumes that interrupts
                                  // are disabled!
    (void) interrupt->SetLevel(oldLevel);
}

//----------------------------------------------------------------------
// Thread::CheckOverflow
//          Check a thread's stack to see if it has overrun the space
//          that has been allocated for it.  If we had a smarter compiler,
//		we wouldn't need to worry about this, but we don't.
//
//		NOTE: Nachos will not catch all stack overflow conditions.
//		In other words, your program may still crash because of an overflow.
//
//		If you get bizarre results (such as seg faults where there is no code)
//		then you *may* need to increase the stack size.  You can avoid stack
//		overflows by not putting large data structures on the stack.
//		Don't do this: void foo() { int bigArray[10000]; ... }
//----------------------------------------------------------------------

void
Thread::CheckOverflow()
{
   if (stack != NULL) {
#ifdef HPUX					// Stacks grow upward on the Snakes
		ASSERT(stack[StackSize - 1] == STACK_FENCEPOST);
#else
		ASSERT(*stack == STACK_FENCEPOST);
#endif
   }
}

//----------------------------------------------------------------------
// Thread::Begin
//		Called by ThreadRoot when a thread is about to begin
//		executing the forked procedure.
//
//		It's main responsibilities are:
//		1. deallocate the previously running thread if it finished
//				(see Thread::Finish())
//		2. enable interrupts (so we can get time-sliced)
//----------------------------------------------------------------------

void
Thread::Begin ()
{
    ASSERT(this == kernel->currentThread);
    DEBUG(dbgThread, "Beginning thread: " << name);

    kernel->scheduler->CheckToBeDestroyed();
    kernel->interrupt->Enable();
}

//----------------------------------------------------------------------
// Thread::Finish
//          Called by ThreadRoot when a thread is done executing the
//          forked procedure.
//
//          NOTE: we can't immediately de-allocate the thread data structure
//          or the execution stack, because we're still running in the thread
//          and we're still on the stack!  Instead, we tell the scheduler
//          to call the destructor, once it is running in the context of a different thread.
//
//          NOTE: we disable interrupts, because Sleep() assumes interrupts
//          are disabled.
//----------------------------------------------------------------------

//
void
Thread::Finish ()
{
    (void) kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentThread);

    DEBUG(dbgThread, "Finishing thread: " << name);

    Sleep(TRUE);                                        // invokes SWITCH
    // not reached
}

//----------------------------------------------------------------------
// Thread::Yield
//          Relinquish the CPU if any other thread is ready to run.
```

```
//		If so, put the thread on the end of the ready list, so that
//		it will eventually be re-scheduled.
//
//		NOTE: returns immediately if no other thread on the ready queue.
//		Otherwise returns when the thread eventually works its way
//		to the front of the ready list and gets re-scheduled.
//
//		NOTE: we disable interrupts, so that looking at the thread
//		on the front of the ready list, and switching to it, can be done
//		atomically.  On return, we re-set the interrupt level to its
//		original state, in case we are called with interrupts disabled.
//
//		Similar to Thread::Sleep(), but a little different.
//----------------------------------------------------------------------

void
Thread::Yield ()
{
   Thread *nextThread;
   IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

   ASSERT(this == kernel->currentThread);

   DEBUG(dbgThread, "Yielding thread: " << name);

   nextThread = kernel->scheduler->FindNextToRun();
   if (nextThread != NULL) {
		kernel->scheduler->ReadyToRun(this);
		kernel->scheduler->Run(nextThread, FALSE);
   }
   (void) kernel->interrupt->SetLevel(oldLevel);
}

//----------------------------------------------------------------------
// Thread::Sleep
//----------------------------------------------------------------------
void
```

```
Thread::Sleep (bool finishing)
{
    Thread *nextThread;

    ASSERT(this == kernel->currentThread);
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Sleeping thread: " << name);

    status = BLOCKED;
    while ((nextThread = kernel->scheduler->FindNextToRun()) == NULL)
            kernel->interrupt->Idle();          // no one to run, wait for an interrupt

    // returns when it's time for us to run
    kernel->scheduler->Run(nextThread, finishing);
}

//----------------------------------------------------------------------
// ThreadBegin, ThreadFinish,  ThreadPrint
//          Dummy functions because C++ does not (easily) allow pointers to member
//          functions.  So we create a dummy C function
//          (which we can pass a pointer to), that then simply calls the
//          member function.
//----------------------------------------------------------------------

static void ThreadFinish()    { kernel->currentThread->Finish(); }
static void ThreadBegin() { kernel->currentThread->Begin(); }
void ThreadPrint(Thread *t) { t->Print(); }

#ifdef PARISC

//----------------------------------------------------------------------
// PLabelToAddr
//          On HPUX, function pointers don't always directly point to code,
//          so we need to do the conversion.
//----------------------------------------------------------------------
```

```
static void *
PLabelToAddr(void *plabel)
{
   int funcPtr = (int) plabel;

   if (funcPtr & 0x02) {
      // L-Field is set.  This is a PLT pointer
      funcPtr -= 2;        // Get rid of the L bit
      return (*(void **)funcPtr);
   } else {
      // L-field not set.
      return plabel;
   }
}
#endif

//----------------------------------------------------------------------
// Thread::StackAllocate
//              Allocate and initialize an execution stack.  The stack is
//              initialized with an initial stack frame for ThreadRoot, which:
//                              enables interrupts
//                              calls (*func)(arg)
//                              calls Thread::Finish
//
//              "func" is the procedure to be forked
//              "arg" is the parameter to be passed to the procedure
//----------------------------------------------------------------------

void
Thread::StackAllocate (VoidFunctionPtr func, void *arg)
{
   stack = (int *) AllocBoundedArray(StackSize * sizeof(int));

#ifdef PARISC
   // HP stack works from low addresses to high addresses
   // everyone else works the other way: from high addresses to low addresses
   stackTop = stack + 16;              // HP requires 64-byte frame marker
   stack[StackSize - 1] = STACK_FENCEPOST;
#endif

#ifdef SPARC
   stackTop = stack + StackSize - 96;           // SPARC stack must contains at
                                                        // least 1 activation record
                                                        // to start with.
   *stack = STACK_FENCEPOST;
#endif

#ifdef PowerPC // RS6000
   stackTop = stack + StackSize - 16;           // RS6000 requires 64-byte frame marker
   *stack = STACK_FENCEPOST;
#endif

#ifdef DECMIPS
   stackTop = stack + StackSize - 4;            // -4 to be on the safe side!
   *stack = STACK_FENCEPOST;
#endif

#ifdef ALPHA
   stackTop = stack + StackSize - 8;            // -8 to be on the safe side!
   *stack = STACK_FENCEPOST;
#endif


#ifdef x86
   // the x86 passes the return address on the stack.  In order for SWITCH()
   // to go to ThreadRoot when we switch to this thread, the return addres
   // used in SWITCH() must be the starting address of ThreadRoot.
   stackTop = stack + StackSize - 4;            // -4 to be on the safe side!
   *(--stackTop) = (int) ThreadRoot;
   *stack = STACK_FENCEPOST;
#endif

#ifdef PARISC
   machineState[PCState] = PLabelToAddr(ThreadRoot);
   machineState[StartupPCState] = PLabelToAddr(ThreadBegin);
   machineState[InitialPCState] = PLabelToAddr(func);
   machineState[InitialArgState] = arg;
   machineState[WhenDonePCState] = PLabelToAddr(ThreadFinish);
#else
   machineState[PCState] = (void*)ThreadRoot;
   machineState[StartupPCState] = (void*)ThreadBegin;
   machineState[InitialPCState] = (void*)func;
   machineState[InitialArgState] = (void*)arg;
   machineState[WhenDonePCState] = (void*)ThreadFinish;
#endif
}

#include "machine.h"

//----------------------------------------------------------------------
// Thread::SaveUserState
//	Save the CPU state of a user program on a context switch.
//
//	Note that a user program thread has *two* sets of CPU registers -- 
//	one for its state while executing user code, one for its state 
//	while executing kernel code.  This routine saves the former.
//----------------------------------------------------------------------

void
Thread::SaveUserState()
{
   for (int i = 0; i < NumTotalRegs; i++)
	userRegisters[i] = kernel->machine->ReadRegister(i);
}

//----------------------------------------------------------------------
// Thread::RestoreUserState
//	Restore the CPU state of a user program on a context switch.
//
//	Note that a user program thread has *two* sets of CPU registers -- 
//	one for its state while executing user code, one for its state 
```

```
//		while executing kernel code.  This routine restores the former.
//----------------------------------------------------------------------

void
Thread::RestoreUserState()
{
   for (int i = 0; i < NumTotalRegs; i++)
		kernel->machine->WriteRegister(i, userRegisters[i]);
}



//----------------------------------------------------------------------
// SimpleThread
//		Loop 5 times, yielding the CPU to another ready thread
//		each iteration.
//
//		"which" is simply a number identifying the thread, for debugging
//		purposes.
//----------------------------------------------------------------------

static void
SimpleThread(int which)
{
   int num;

   for (num = 0; num < 5; num++) {
		cout << "*** thread " << which << " looped " << num << " times\n";
       kernel->currentThread->Yield();
   }
}

//----------------------------------------------------------------------
// Thread::SelfTest
//		Set up a ping-pong between two threads, by forking a thread
//		to call SimpleThread, and then calling SimpleThread ourselves.
//----------------------------------------------------------------------
```

```
void

Thread::SelfTest()

{

   DEBUG(dbgThread, "Entering Thread::SelfTest");



   Thread *t = new Thread("forked thread");



   t->Fork((VoidFunctionPtr) SimpleThread, (void *) 1);

   kernel->currentThread->Yield();

   SimpleThread(0);

}
```

- Next, we have implemented the five system calls i.e, SendMessage, WaitMessage, SendAnswer, WaitAnswer and Exit.

- To implement these calls we have modified nachos/code/test/start.S, nachos/code/userprog/syscall.h and nachos/code/userprog/exception.cc

start.S:

```
/* Start.s

 *              Assembly language assist for user programs running on top of Nachos.

 *

 *              Since we don't want to pull in the entire C library, we define

 *              what we need for a user program here, namely Start and the system

 *              calls.

 */


#define IN_ASM

#include "syscall.h"


      .text

      .align  2


/* -------------------------------------------------------------

 * __start

 *              Initialize running a C program, by calling "main".

 *

 *              NOTE: This has to be first, so that it gets loaded at location 0.

 *              The Nachos kernel always starts a program by jumping to location 0.

 * -------------------------------------------------------------
 */

        .globl __start
        .ent    __start
__start:
        jal     main
        move    $4,$0
        jal     Exit    /* if we return from main, exit(0) */
        .end __start

/* -------------------------------------------------------------
 * System call stubs:
 *      Assembly language assist to make system calls to the Nachos kernel.
 *      There is one stub per system call, that places the code for the
 *      system call into register r2, and leaves the arguments to the
 *      system call alone (in other words, arg1 is in r4, arg2 is
 *      in r5, arg3 is in r6, arg4 is in r7)
 *
 *      The return value is in r2. This follows the standard C calling
 *      convention on the MIPS.
 * -------------------------------------------------------------
 */

        .globl Halt
        .ent    Halt
Halt:
        addiu $2,$0,SC_Halt
        syscall
        j       $31
        .end Halt

        .globl Add
        .ent    Add
Add:
        addiu $2,$0,SC_Add
        syscall
        j       $31
        .end Add

        .globl Exit
        .ent    Exit
Exit:
        addiu $2,$0,SC_Exit
        syscall
        j       $31
        .end Exit

        .globl Exec
        .ent    Exec
Exec:
        addiu $2,$0,SC_Exec
        syscall
        j       $31
        .end Exec

        .globl ExecV
        .ent    ExecV
ExecV:
        addiu $2,$0,SC_ExecV
        syscall
        j       $31
        .end ExecV

        .globl Join
        .ent    Join
Join:
        addiu $2,$0,SC_Join
        syscall
        j       $31
        .end Join

        .globl Create
        .ent    Create
Create:
        addiu $2,$0,SC_Create
        syscall
        j       $31
        .end Create

        .globl Remove
        .ent    Remove
Remove:
        addiu $2,$0,SC_Remove
        syscall
        j       $31
        .end Remove

        .globl Open
        .ent    Open
Open:
        addiu $2,$0,SC_Open
        syscall
        j       $31
        .end Open

        .globl Read
        .ent    Read
Read:
        addiu $2,$0,SC_Read
        syscall
        j       $31
        .end Read

        .globl Write
        .ent    Write
Write:
        addiu $2,$0,SC_Write
        syscall
        j       $31
        .end Write
        .globl Close
        .ent        Close
Close:
        addiu $2,$0,SC_Close
        syscall
        j           $31
        .end Close

        .globl Seek
        .ent        Seek
Seek:
        addiu $2,$0,SC_Seek
        syscall
        j           $31
        .end Seek

    .globl ThreadFork
    .ent    ThreadFork
ThreadFork:
    addiu $2,$0,SC_ThreadFork
    syscall
    j      $31
    .end ThreadFork

    .globl ThreadYield
    .ent    ThreadYield
ThreadYield:
    addiu $2,$0,SC_ThreadYield
    syscall
    j      $31
    .end ThreadYield

        .globl ThreadExit
        .ent    ThreadExit
ThreadExit:
        addiu $2, $0, SC_ThreadExit
        syscall
```

```
          j          $31
          .end ThreadExit

          .globl ThreadJoin
          .ent    ThreadJoin
ThreadJoin:
          addiu $2, $0, SC_ThreadJoin
          syscall
          j          $31
          .end ThreadJoin

    .globl SendMessage
    .ent    SendMessage
SendMessage:
    addiu $2, $0, SC_SendMessage
    syscall
    j     $31
    .end SendMessage

    .globl WaitMessage
    .ent    WaitMessage
WaitMessage:
    addiu $2, $0, SC_WaitMessage
    syscall
    j     $31
    .end WaitMessage

          .globl WaitAnswer
          .ent    WaitAnswer
WaitAnswer:
    addiu $2, $0, SC_WaitAnswer
    syscall
    j     $31
    .end WaitAnswer

    .globl SendAnswer
    .ent    SendAnswer
```

```
SendAnswer:
        addiu $2, $0, SC_SendAnswer
        syscall
        j       $31
        .end SendAnswer

/* dummy function to keep gcc happy */
        .globl  __main
        .ent    __main
__main:
        j       $31
        .end    __main
```

<u>syscall.h:</u>

```
/* syscalls.h
 *              Nachos system call interface.  These are Nachos kernel operations
 *              that can be invoked from user programs, by trapping to the kernel
 *              via the "syscall" instruction.
 *
 *              This file is included by user programs and by the Nachos kernel.
 *
 * Copyright (c) 1992-1993 The Regents of the University of California.
 * All rights reserved.  See copyright.h for copyright notice and limitation
 * of liability and disclaimer of warranty provisions.
 */

#ifndef SYSCALLS_H
#define SYSCALLS_H

#include "copyright.h"
#include "errno.h"
/* system call codes -- used by the stubs to tell the kernel which system call
 * is being asked for
 */
#define SC_Halt                 0
#define SC_Exit                 1
#define SC_Exec                 2
```

```
#define SC_Join                 3

#define SC_Create      4

#define SC_Remove      5

#define SC_Open                 6

#define SC_Read                 7

#define SC_Write       8

#define SC_Seek       9

#define SC_Close       10

#define SC_ThreadFork           11

#define SC_ThreadYield          12

#define SC_ExecV       13

#define SC_ThreadExit   14

#define SC_ThreadJoin   15


#define SC_Add                  42


#define SC_SendMessage 50

#define SC_WaitMessage 51

#define SC_SendAnswer 52

#define SC_WaitAnswer 53


#ifndef IN_ASM


/* The system call interface.  These are the operations the Nachos

 * kernel needs to support, to be able to run user programs.

 *

 * Each of these is invoked by a user program by simply calling the

 * procedure; an assembly language stub stuffs the system call code

 * into a register, and traps to the kernel.  The kernel procedures

 * are then invoked in the Nachos kernel, after appropriate error checking,

 * from the system call entry point in exception.cc.

 */


/* Stop Nachos, and print out performance stats */

void Halt();


int SendMessage(char *receiver, char *message, int buffer_id);
int WaitMessage(char *sender, char *message, int buffer_id);


int SendAnswer(int result, char *answer, int buffer_id);


int WaitAnswer(int result, char *answer, int buffer_id);
/*
 * Add the two operants and return the result
 */


int Add(int op1, int op2);


/* Address space control operations: Exit, Exec, Execv, and Join */


/* This user program is done (status = 0 means exited normally). */
void Exit(int status);


/* A unique identifier for an executing user program (address space) */
typedef int SpaceId;


/* A unique identifier for a thread within a task */
typedef int ThreadId;


/* Run the specified executable, with no args */
/* This can be implemented as a call to ExecV.
 */
SpaceId Exec(char* exec_name);


/* Run the executable, stored in the Nachos file "argv[0]", with
 * parameters stored in argv[1..argc-1] and return the
 * address space identifier
 */
SpaceId ExecV(int argc, char* argv[]);


/* Only return once the user program "id" has finished.
 * Return the exit status.
 */
```

```
int Join(SpaceId id);


/* File system operations: Create, Remove, Open, Read, Write, Close
 * These functions are patterned after UNIX -- files represent
 * both files *and* hardware I/O devices.
 *
 * Note that the Nachos file system has a stub implementation, which
 * can be used to support these system calls if the regular Nachos
 * file system has not been implemented.
 */

/* A unique identifier for an open Nachos file. */
typedef int OpenFileId;

/* when an address space starts up, it has two open files, representing
 * keyboard input and display output (in UNIX terms, stdin and stdout).
 * Read and Write can be used directly on these, without first opening
 * the console device.
 */

#define ConsoleInput  0
#define ConsoleOutput        1

/* Create a Nachos file, with name "name" */
/* Note: Create does not open the file.   */
/* Return 1 on success, negative error code on failure */
int Create(char *name);

/* Remove a Nachos file, with name "name" */
int Remove(char *name);

/* Open the Nachos file "name", and return an "OpenFileId" that can
 * be used to read and write to the file.
 */
OpenFileId Open(char *name);
```

```
/* Write "size" bytes from "buffer" to the open file.
 * Return the number of bytes actually read on success.
 * On failure, a negative error code is returned.
 */
int Write(char *buffer, int size, OpenFileId id);

/* Read "size" bytes from the open file into "buffer".
 * Return the number of bytes actually read -- if the open file isn't
 * long enough, or if it is an I/O device, and there aren't enough
 * characters to read, return whatever is available (for I/O devices,
 * you should always wait until you can return at least one character).
 */
int Read(char *buffer, int size, OpenFileId id);

/* Set the seek position of the open file "id"
 * to the byte "position".
 */
int Seek(int position, OpenFileId id);

/* Close the file, we're done reading and writing to it.
 * Return 1 on success, negative error code on failure
 */
int Close(OpenFileId id);


/* User-level thread operations: Fork and Yield.  To allow multiple
 * threads to run within a user program.
 *
 * Could define other operations, such as LockAcquire, LockRelease, etc.
 */

/* Fork a thread to run a procedure ("func") in the *same* address space
 * as the current thread.
 * Return a positive ThreadId on success, negative error code on failure
 */
ThreadId ThreadFork(void (*func)());
```

```
/* Yield the CPU to another runnable thread, whether in this address space
 * or not.
 */
void ThreadYield();


/*
 * Blocks current thread until lokal thread ThreadID exits with ThreadExit.
 * Function returns the ExitCode of ThreadExit() of the exiting thread.
 */
int ThreadJoin(ThreadId id);


/*
 * Deletes current thread and returns ExitCode to every waiting lokal thread.
 */
void ThreadExit(int ExitCode);


#endif /* IN_ASM */


#endif /* SYSCALL_H */
```

System Calls:

We have implemented utility functions for the system calls.

a. convertToString: To convert char* to string:

- This function takes a char* as input and iterates till the \0 character and adds each char to a string returning a string as an output.

```
//utility function - convert char* to string
std::string convertToString(int startAddress)
{
  std::string resultString;
  int charIndex = startAddress;
  char character;

  while(true){
    bool flag = kernel->machine->ReadMem(startAddress, 1, &charIndex);
    character = (char)charIndex;
    if(flag && character != '\0'){
      resultString += character;
      startAddress++;
    }
    else{
      break;
    }
  }
  return resultString;
}
```

b. getThreadByName: To fetch a thread by its name:

- This method takes a string name as an input and iterates the kernel's list of forked threads using a ListIterator and compares each thread's name with the input, if there is a match it returns the corresponding thread, if no match is found it returns NULL.

```
//utility function to get the thread by its name
Thread* getThreadByName(std::string name)
{
  Thread *matchingThread = NULL;

  ListIterator<Thread*> *iterator = new ListIterator<Thread*>(kernel->listOfForkedThreads);
  for (; !iterator->IsDone(); iterator->Next()) {
    if(iterator->Item()->getName() == name){
       matchingThread = iterator->Item();
       break;
    }
  }
  return matchingThread;
}
```

c. <u>getExistingBuffer:</u> To get the buffer entry from the list of kernel buffer pool:

- This buffer takes a buffer id and receiver thread name and finds the buffer with the same id in the kernel's buffer pool, then we check if the buffer's receiver has the same name as the input receiver string, if they are equal we return the buffer, else we return NULL.

```
Buffer* getExistingBuffer(int buffer_id, std::string receiverThreadName)
{
  std::map<int, List<Buffer*>* >::iterator it;
  it = kernel->bufferPool.find(buffer_id);

  if(it != kernel->bufferPool.end())  //the entry is found
  {
    Buffer *newBuffer = it->second->Front();
    Thread *receiverThread = it->second->Front()->getReceiver();

    if(newBuffer != NULL && (newBuffer->getReceiver()->getName() == receiverThreadName)){
       return newBuffer;
    }
  }
  return NULL;

}
```

d. <u>printContentsOfMap</u>: To print contents of kernel buffer pool map:

- This method uses a map iterator to print the content of each pair on the console.

```
void printContentsOfMap()
{
  for(map<int, List<Buffer*>* >::const_iterator it = kernel->bufferPool.begin();
    it != kernel->bufferPool.end(); ++it)
  {
    std::cout << it->first << " " << it->second->Front()->getBufferId() << " " << "\n";
  }

}
```

e. <u>getThreadBuffer:</u> To get the thread buffer entry from current thread's queue by passing buffer id and the message sender:

- This method takes a buffer id and the sender name as input and iterates over the current threads's buffer pool to find the corresponding buffer, if the entry is found it returns the buffer. If the entry is not found it searches every buffer in the current threads' pool and returns the buffer where the sender's name is equal to the input sender. If no matching thread is found it returns NULL.

```
Buffer* getThreadBuffer(int buffer_id, std::string senderName)
{
  std::map<int, Buffer*>::iterator it;
```

```
 it = kernel->currentThread->tBufferPool.find(buffer_id);

 Buffer *newBuffer = NULL;
 if(it != kernel->currentThread->tBufferPool.end())  //the entry is found
 {
   newBuffer = it->second;
   return newBuffer;
 }
 else{
   for(map<int, Buffer* >::const_iterator it = kernel->currentThread->tBufferPool.begin();
     it != kernel->currentThread->tBufferPool.end(); ++it){
     if(it->second->getSender()->getName() == senderName){
       newBuffer = it->second;
       return newBuffer;
     }
   }
 }
 return newBuffer;
}
```

f. copyMessageToMem: To write the message into the memory:

- This method writes a string to the memory using the WriteMem method for writing each character in the string.

```
void copyMessageToMem(std::string message, int messagePtr)
{
   for(int i = messagePtr, j = 0; i < messagePtr + message.length(); i++, j++)
   {
      kernel->machine->WriteMem(i, 1, (int)message.at(j));
   }
}
```

g. updateProgramCounter: To update the program counter:

- This method updates the program counter using the PC registers.

```
void updateProgramCounter()
{
   /* set previous programm counter (debugging only)*/
   kernel->machine->WriteRegister(PrevPCReg, kernel->machine->ReadRegister(PCReg));

   /* set programm counter to next instruction (all Instructions are 4 byte wide)*/
   kernel->machine->WriteRegister(PCReg, kernel->machine->ReadRegister(PCReg) + 4);

   /* set next programm counter for brach execution */
   kernel->machine->WriteRegister(NextPCReg, kernel->machine->ReadRegister(PCReg)+4);
}
```

h. deleteBufferEntryFromKernel: To delete the buffer entry from kernel

- This method takes a buffer id and finds an entry with the same buffer id in the kernel buffer pool and deletes it from the buffer pool.

```
void Kernel::deleteBufferEntryFromKernel(int buffer_id)
{
 if(buffer_id != -1)
 {
   bitmapForBuffer->Clear(buffer_id);
   std::map<int, List<Buffer*>* >::iterator it;
   it = bufferPool.find(buffer_id);

   if(it != kernel->bufferPool.end())  //the entry is found
   {
     if(!it->second->IsEmpty())
     {
       bufferPool.erase(it);
     }
   }}}
```

1. <u>SendMessage(receiver, message, buffer_id):</u>

- For the implementation of this system call, we have first turned OFF the interrupts and then read the input of receiverIndex, messageIndex and the buffer_id.
- From the input of receiver's name we get the receiver thread using our utility function getThreadByName().
- First, we check if the receiver thread is not NULL, if it is we delete the buffer link between the sender and receiver.
- Next, we check if the currentThread has not exceeded its message limit, if it has then we send a dummy message to the receiver.
- Then, we check if the buffer_id provided by the user is -1, because this corresponds to the fact that a buffer does not exist between the sender and receiver and hence a new buffer is created, the message to be sent is added to the list of messages of the new buffer.The new buffer is then added to the kernel's buffer pool as well as the receiver thread's buffer bool.
- If the buffer_id is not -1, the user has passed a buffer_id and in this case we fetch the existing buffer using our utility function getExistingBuffer, if this function returns a valid buffer we send the message using this buffer, else it results in a security error as the buffer does not exist.
- At the end of this system call we turn ON the interrupts as well as update the program counter.

```
IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
    std::string receiver, message;

    int receiverIndex = (int)kernel->machine->ReadRegister(4);
    int messageIndex = (int) kernel->machine->ReadRegister(5);
    int buffer_id = (int)kernel->machine->ReadRegister(6);

    receiver = convertToString(receiverIndex);
    message = convertToString(messageIndex);

    Thread *receiverThread = getThreadByName(receiver);

    if(receiverThread != NULL)
    {
     if(kernel->currentThread->getMessageCount() < kernel->MessageLimit)
     {
       if(buffer_id == -1){
        Buffer *buffer = new Buffer(kernel->currentThread, receiverThread);

         if(buffer->getBufferId() == -1){
          kernel->machine->WriteRegister(2, (int)buffer->getBufferId());
          updateProgramCounter();
          (void) kernel->interrupt->SetLevel(oldLevel);
          break;
         }
        std::cout<<"-------------------------------------------\n";

        std::cout<<"Buffer ID: " << buffer->getBufferId() << " created between Threads: " << kernel->currentThread->getName() << " and " <<
receiverThread->getName() << endl;

         std::cout<<"-------------------------------------------\n";
        buffer->getMessagesList()->Append(message);
        List<Buffer*> *kernelBufferList = new List<Buffer*>();
        kernelBufferList->Append(buffer);

        kernel->bufferPool.insert(std::pair<int, List<Buffer*>* > (buffer->getBufferId(), kernelBufferList));
        receiverThread->tBufferPool.insert(std::pair<int, Buffer*> (buffer->getBufferId(), buffer));

        kernel->machine->WriteRegister(2, (int)buffer->getBufferId());

        std::cout<<"Message sent from " << kernel->currentThread->getName() << " to " << receiverThread->getName() << " , Message: " << message
<< endl;


        kernel->currentThread->incrementMessageCount();
       }
       else{  //user program has passed some buffer id
        Buffer *validateBuffer = getExistingBuffer(buffer_id, receiver);

        if(validateBuffer != NULL){
         validateBuffer->getMessagesList()->Append(message);
         receiverThread->tBufferPool.insert(std::pair<int, Buffer*> (buffer_id, validateBuffer));

         kernel->machine->WriteRegister(2, buffer_id);

         kernel->currentThread->incrementMessageCount();
         std::cout<<"Message sent from " << kernel->currentThread->getName() << " to " << receiverThread->getName() << " , Message: " << message
<< endl;
        }
        else{
         kernel->machine->WriteRegister(2, -1);
         std::cout<<"Security error while sending to " << receiverThread->getName() << endl;
        }
       }

     }
     else{  // since the message limit is reached, kernel is sending dummy messages
      Buffer *validateBuffer = getExistingBuffer(buffer_id, receiver);

      if(kernel->currentThread->getMessageCount() < kernel->MessageLimit)
      {
       if(validateBuffer != NULL){
        validateBuffer->getMessagesList()->Append(kernel->DummyMessage);
        receiverThread->tBufferPool.insert(std::pair<int, Buffer*> (buffer_id, validateBuffer));

        kernel->machine->WriteRegister(2, buffer_id);
       }

       else{
         std::cout<<"Dummy message sent to " << receiverThread->getName() << endl;
       }
      }
      else{
       std::cout<<"Maximum message limit reached for " << kernel->currentThread->getName();
       std::cout<<", so Dummy message sent to " << receiverThread->getName() << endl;
      }

     }
    }
    else{
     kernel->deleteBufferEntryFromKernel(buffer_id);
    }

    updateProgramCounter();
    (void)kernel-> interrupt->SetLevel(oldLevel);


   }
```

```
        return;
        //
        // ASSERTNOTREACHED();

          break;
```

2. <u>WaitMessage(sender, message, buffer_id):</u>

- For the implementation of this system call, we have first turned OFF the interrupts and then read the input of senderIndex, messageIndex and the buffer_id.
- From the input of sender 's name we get the receiver thread using our utility function getThreadByName().
- Next, we check if the sender thread is alive by checking if it exists in the kernel's list of forked threads.
- If it is alive, then we fetch the buffer using the getThreadBuffer() utility function. If this buffer exists, we read the message from its list of messages and copy it to memory and write it to the register. If the buffer does not exits the thread yields.
- If the sender thread is dead and the buffer is NULL, the buffer link between sender and receiver is deleted and a dummy message is received.
- At the end of this system call we turn ON the interrupts as well as update the program counter.

```
IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);
    std::string sender, message;

    int senderIndex = (int)kernel->machine->ReadRegister(4);
    int messageIndex = (int) kernel->machine->ReadRegister(5);
    int buffer_id = (int)kernel->machine->ReadRegister(6);

    sender = convertToString(senderIndex);
    message = convertToString(messageIndex);

    Thread *senderThread = getThreadByName(sender);

    while(TRUE)
    {
      if(kernel->listOfForkedThreads->IsInList(senderThread)) //sender thread is alive
      {
        Buffer *buffer = getThreadBuffer(buffer_id, sender);
        if(buffer != NULL && !buffer->getMessagesList()->IsEmpty())
        {
          std::string message = buffer->getMessagesList()->RemoveFront();
          copyMessageToMem(message,messageIndex);
          kernel->machine->WriteRegister(2, (int)buffer->getBufferId());
          cout << "Message received by " << kernel->currentThread->getName() << " sent from " << senderThread->getName() << " , Message : " << 
message << endl;
          break;
        }
        else{
         kernel->currentThread->Yield();
        }
      }
      else{ //sender thread dead
       Buffer *buffer = getThreadBuffer(buffer_id, sender);// get current thread pool
       if(buffer != NULL && !buffer->getMessagesList()->IsEmpty())
       {
         std::string message = buffer->getMessagesList()->RemoveFront();
         copyMessageToMem(message,messageIndex);
         kernel->machine->WriteRegister(2, (int)buffer->getBufferId());
         cout << "Message received by " << kernel->currentThread->getName() << " sent from " << sender << " , Message : " << message << endl;
       }
       else{
        kernel->deleteBufferEntryFromKernel(buffer_id);
        copyMessageToMem(kernel->DummyMessage,messageIndex);
        kernel->machine->WriteRegister(2,-1);
        cout << "Sender Thread finished without sending message to " << kernel->currentThread->getName();
        cout << ", so Dummy message sent to " << kernel->currentThread->getName() << endl;
       }
       break;
      }
    }
    updateProgramCounter();
    (void)kernel-> interrupt->SetLevel(oldLevel);
   }

   return;
   //
   // ASSERTNOTREACHED();

     break;
```

3. <u>SendAnswer:</u>

- For the implementation of this system call, we have first turned OFF the interrupts and then read the input of result, answerIndex and the buffer_id.
- In the buffer pool of the current thread i.e. the sender, we find the corresponding buffer. If the buffer exists for the buffer_id, the buffer's sender is assigned as the receiver thread.
- If this receiver thread exists in the kernel's list of forked thread, we search for the same buffer_id in the receiver thread's buffer pool and append the answer in the list of messages, if the buffer_id entry does not exist in the receiver thread's buffer pool we create a new buffer and append the answer to the list of messages.
- If this receiver thread does not exist in the kernel's list of forked thread we delete the buffer link between sender and receiver and send a Dummy Response.
- At the end of this system call we turn ON the interrupts as well as update the program counter.

```
IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    std::string answer;



    int result = (int)kernel->machine->ReadRegister(4);

    int answerIndex = (int) kernel->machine->ReadRegister(5);

    int buffer_id = (int)kernel->machine->ReadRegister(6);



    answer = convertToString(answerIndex);



    std::map<int, Buffer*>::iterator it1;

    it1 = kernel->currentThread->tBufferPool.find(buffer_id);

    Buffer *currentThreadBuffer = NULL;
```

```
 if(it1 != kernel->currentThread->tBufferPool.end()){  //the entry is found
   currentThreadBuffer = it1->second;
   Thread* receiverThread = currentThreadBuffer->getSender();
   //std::cout << "receiverThread " << receiverThread->getName() << endl;
   if(kernel->listOfForkedThreads->IsInList(receiverThread)){
     std::map<int, Buffer*>::iterator it2;
     it2 = receiverThread->tBufferPool.find(buffer_id);

     if(it2 != receiverThread->tBufferPool.end()){ //if receiver has the buffer entry append answer
      it2->second->getMessagesList()->Append(answer);
     }

     else{ //receiver doesn't have buffer entry create a new buffer instance in receiver bufferpool with same buffer id
      Buffer *newBuffer = new Buffer(kernel->currentThread, receiverThread, buffer_id);
      newBuffer->getMessagesList()->Append(answer);

      std::map<int, List<Buffer*>* >::iterator it3;
      it3 = kernel->bufferPool.find(buffer_id);
      if(it3 != kernel->bufferPool.end()){
       it3->second->Append(newBuffer);
      }

      receiverThread->tBufferPool.insert(std::pair<int, Buffer*> (newBuffer->getBufferId(), newBuffer));
     }
     kernel->machine->WriteRegister(2, (int)currentThreadBuffer->getBufferId());
     cout << "Answer sent from " << kernel->currentThread->getName() << " to " << receiverThread->getName() << ", Answer :" << answer << endl;
   }
   else{
    kernel->deleteBufferEntryFromKernel(buffer_id);
    kernel->machine->WriteRegister(2, -1);
    cout << "Dummy response sent in case of receiver being dead or null" << endl;
   }
 }
 updateProgramCounter();
 (void)kernel-> interrupt->SetLevel(oldLevel);
}
```

```
    return;

    //

    // ASSERTNOTREACHED();


    break;
```

4.WaitAnswer:

- For the implementation of this system call, we have first turned OFF the interrupts and then read the input of result, answerIndex and the buffer_id.
- In the buffer pool of the current thread i.e. the receiver, if the entry of the buffer_id exists, we assign the sender name as the sender from the buffer. If the entry does not exist, the buffer_id is found in the kernel's buffer pool and the receiver from the buffer is assigned as the sender.
- The sender thread is assigned by using the sender name found in the previous step.
- If the sender thread exists in the kernel's list of forked threads, we get the buffer using getThreadBuffer()for the sender and buffer_id, if the buffer is not NULL, we read the answer from the list of messages of the buffer. If the list of messages is empty or the buffer does not exist, the receiver thread yields.
- If the sender thread does not exist in the list of forked threads, we get the buffer from the current thread i.e sender thread pool and if it is not NULL, we receive the answer by removing the answer from list of messages. If it is NULL, we delete the buffer link between sender and receiver and receive a Dummy Response.
- At the end of this system call we turn ON the interrupts as well as update the program counter.

```
IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    int result = (int)kernel->machine->ReadRegister(4);

    int answerIndex = (int) kernel->machine->ReadRegister(5);

    int buffer_id = (int)kernel->machine->ReadRegister(6);

    std::string sender;


    std::map<int, Buffer*>::iterator it;

    it = kernel->currentThread->tBufferPool.find(buffer_id);

    if(it != kernel->currentThread->tBufferPool.end()) {// if entry exists

      sender = it->second->getSender()->getName();

    }

    else

    {

      std::map<int, List<Buffer*>* >::iterator it;

      it = kernel->bufferPool.find(buffer_id);

      if(it != kernel->bufferPool.end()){

        sender = it->second->Front()->getReceiver()->getName();
```

```
 }
}


Thread *senderThread = getThreadByName(sender);
while(TRUE) {
 if(kernel->listOfForkedThreads->IsInList(senderThread)){
  Buffer *buffer = getThreadBuffer(buffer_id, sender);
  if(buffer != NULL && !buffer->getMessagesList()->IsEmpty())
  {
   std::string message = buffer->getMessagesList()->RemoveFront();
   copyMessageToMem(message,answerIndex);
   kernel->machine->WriteRegister(2, (int)buffer->getBufferId());

   std::size_t found = message.find_last_of(" ");

            std::string senderExtracted = message.substr(found+1);

   cout << "Answer received by " << kernel->currentThread->getName() << " sent from " << senderExtracted << " , Answer : " << message << endl;
   break;
  }
  else
  {
   kernel->currentThread->Yield();
  }

 }
 else
 {
  //std::cout<<"BUffer id : " << buffer_id << " , sender : "<< sender << endl;
  Buffer *buffer = getThreadBuffer(buffer_id, sender);// get current thread pool

  //std::cout<<"BUffer entry : " << buffer << endl;

  if(buffer != NULL && !buffer->getMessagesList()->IsEmpty())
  {
   std::string message = buffer->getMessagesList()->RemoveFront();
   copyMessageToMem(message,answerIndex);
```

```
          kernel->machine->WriteRegister(2, (int)buffer->getBufferId());


          std::size_t found = message.find_last_of(" ");


                   std::string senderExtracted = message.substr(found+1);


          cout << "Answer received by " << kernel->currentThread->getName() << " sent from " << senderExtracted << " , Answer : " << message << endl;
         }
         else
         {
          kernel->deleteBufferEntryFromKernel(buffer_id);
          copyMessageToMem(kernel->DummyMessage,0);
          kernel->machine->WriteRegister(2,0);
          cout << "Dummy answer sent to " << kernel->currentThread->getName() << " due to security error by sending on wrong buffer id or sender thread
exited" << endl;
         }
         break;
         }
       }
       updateProgramCounter();
       (void)kernel-> interrupt->SetLevel(oldLevel);
      }


      return;
      //
      // ASSERTNOTREACHED();


      break;
```

5.Exit:

- In this system call, we remove the current thread from the kernel’s list of forked threads.
- Then we call Finish() on the current thread.
- At the end of this system call we turn ON the interrupts as well as update the program counter.

```
 kernel->listOfForkedThreads->Remove(kernel->currentThread);


    cout<<"Exit Called by :"<<kernel->currentThread->getName()<<endl;
```

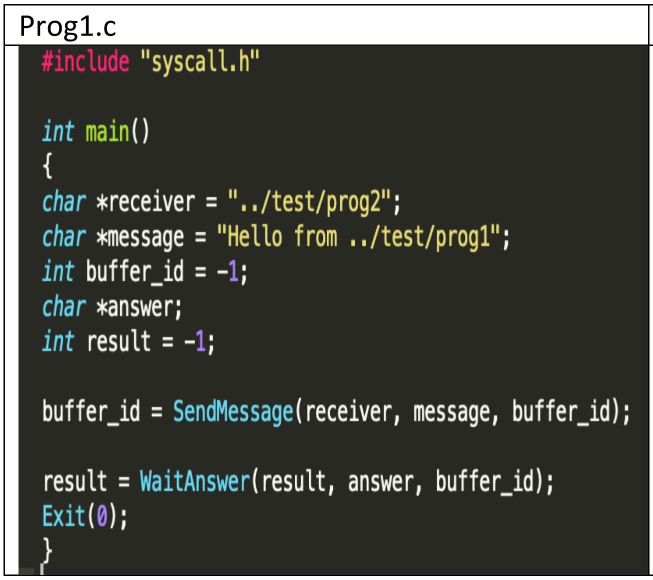
```
kernel->currentThread->Finish();

updateProgramCounter();
```

3. Test cases & output screenshots –

We have added 18 user program files for testing from prog1.c to prog18.c in the nachos/code/test directory.

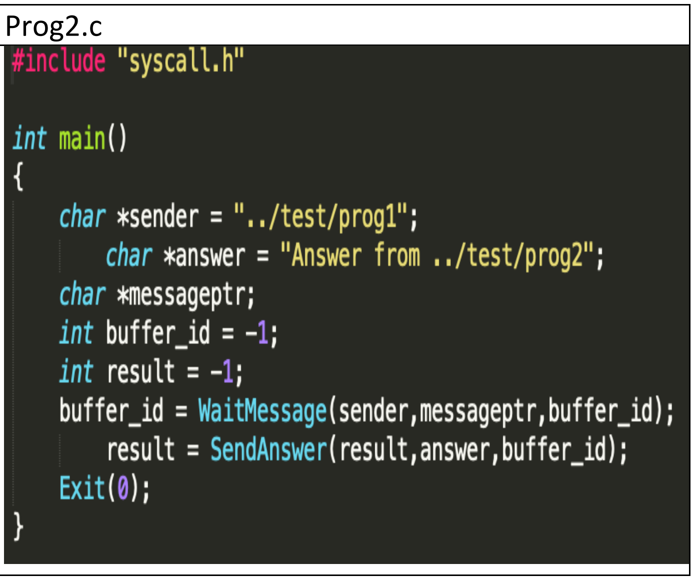
1. This test case shows simple communication between two user programs where prog1 sends a message and prog2 receives this message and sends an answer using the same buffer which prog1 receives.

Prog1.c



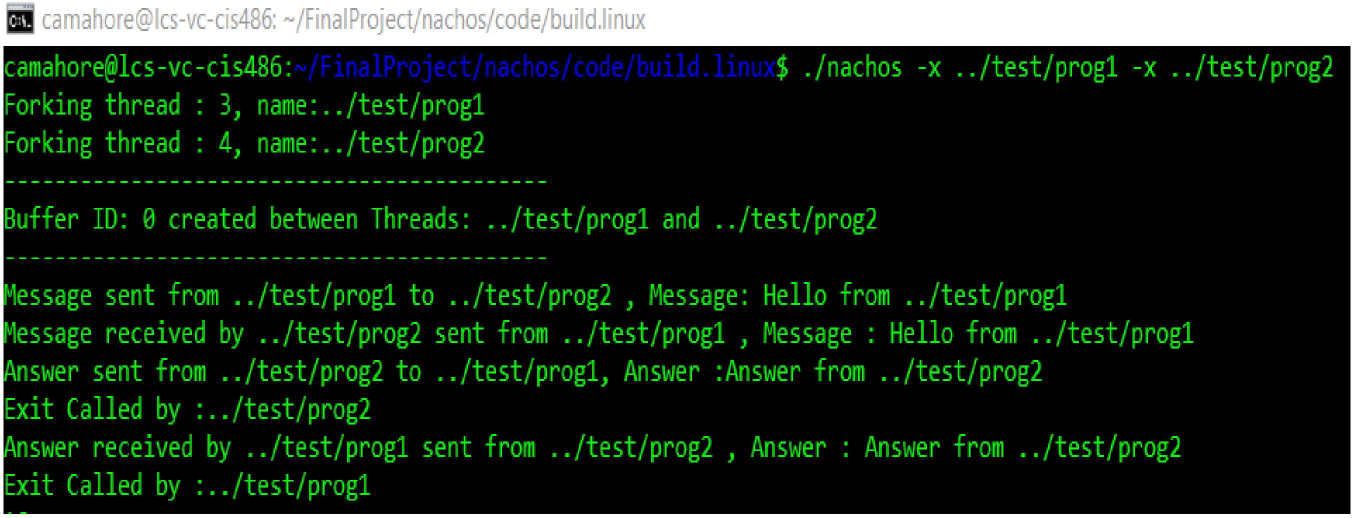
```
#include "syscall.h"

int main()
{
char *receiver = "../test/prog2";
char *message = "Hello from ../test/prog1";
int buffer_id = -1;
char *answer;
int result = -1;

buffer_id = SendMessage(receiver, message, buffer_id);

result = WaitAnswer(result, answer, buffer_id);
Exit(0);
}
```

Prog2.c



```
#include "syscall.h"

int main()
{
    char *sender = "../test/prog1";
        char *answer = "Answer from ../test/prog2";
    char *messageptr;
    int buffer_id = -1;
    int result = -1;
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
        result = SendAnswer(result,answer,buffer_id);
    Exit(0);
}
```

Output:

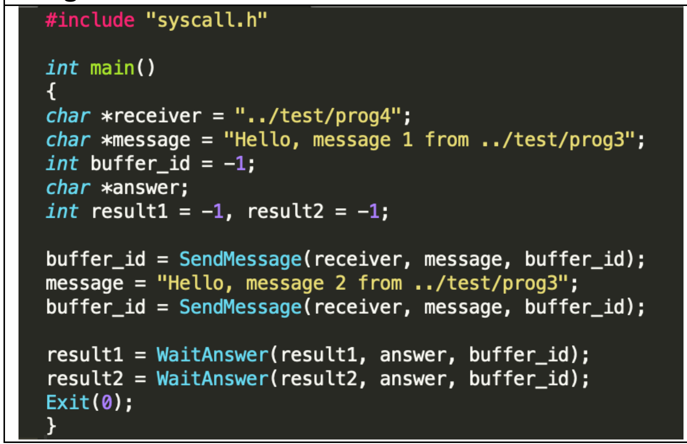


camahore@lcs-vc-cis486: ~/FinalProject/nachos/code/build.linux

```
camahore@lcs-vc-cis486:~/FinalProject/nachos/code/build.linux$ ./nachos -x ../test/prog1 -x ../test/prog2
Forking thread : 3, name:../test/prog1
Forking thread : 4, name:../test/prog2
-------------------------------------------
Buffer ID: 0 created between Threads: ../test/prog1 and ../test/prog2
-------------------------------------------
Message sent from ../test/prog1 to ../test/prog2 , Message: Hello from ../test/prog1
Message received by ../test/prog2 sent from ../test/prog1 , Message : Hello from ../test/prog1
Answer sent from ../test/prog2 to ../test/prog1, Answer :Answer from ../test/prog2
Exit Called by :../test/prog2
Answer received by ../test/prog1 sent from ../test/prog2 , Answer : Answer from ../test/prog2
Exit Called by :../test/prog1
```

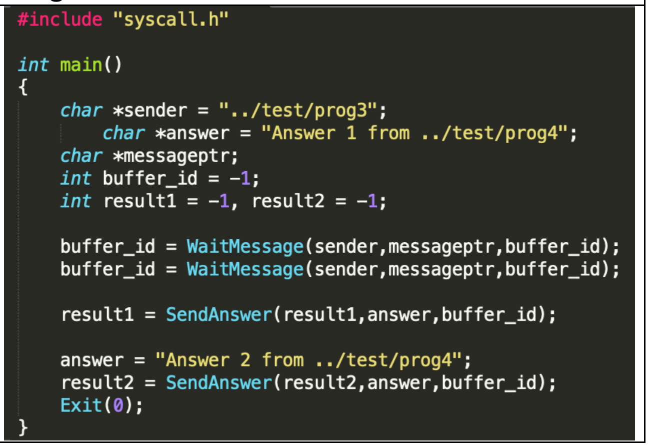
2. In this test case we are sending two messages from prog3 to prog4 and separate answers for each message. We can see that the same buffer is used to send both the messages. Also, prog3 sends both messages and then receives the answers whereas prog4 receives both messages and then sends both answers.

Prog3.c



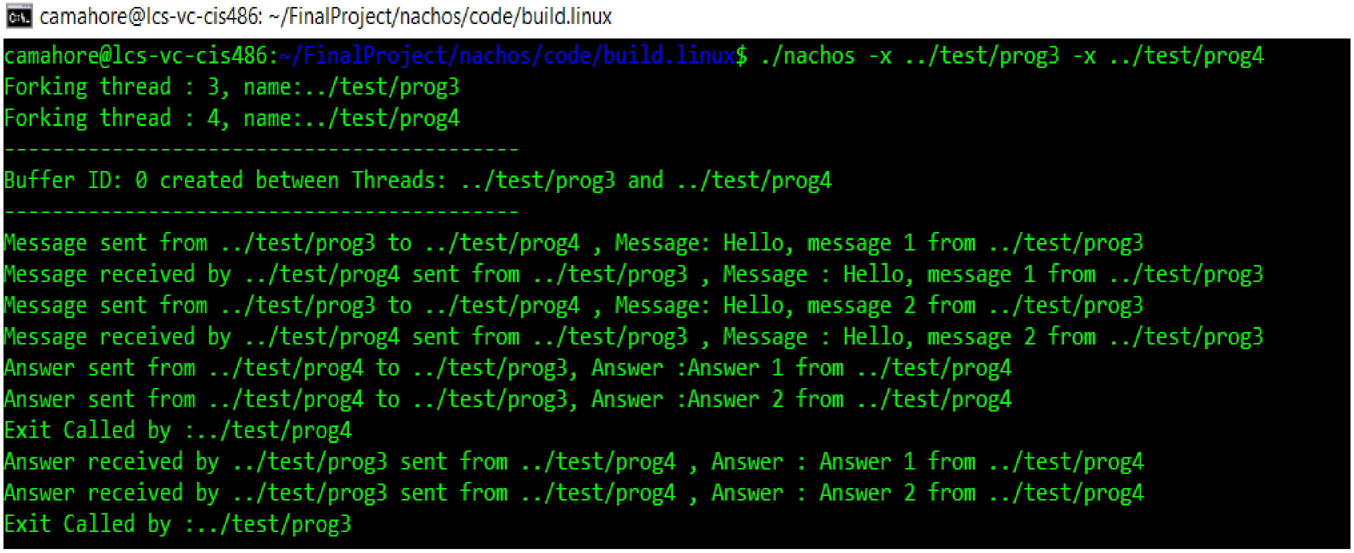
```
#include "syscall.h"

int main()
{
char *receiver = "../test/prog4";
char *message = "Hello, message 1 from ../test/prog3";
int buffer_id = -1;
char *answer;
int result1 = -1, result2 = -1;

buffer_id = SendMessage(receiver, message, buffer_id);
message = "Hello, message 2 from ../test/prog3";
buffer_id = SendMessage(receiver, message, buffer_id);

result1 = WaitAnswer(result1, answer, buffer_id);
result2 = WaitAnswer(result2, answer, buffer_id);
Exit(0);
}
```

Prog4.c



```
#include "syscall.h"

int main()
{
    char *sender = "../test/prog3";
        char *answer = "Answer 1 from ../test/prog4";
    char *messageptr;
    int buffer_id = -1;
    int result1 = -1, result2 = -1;

    buffer_id = WaitMessage(sender,messageptr,buffer_id);
    buffer_id = WaitMessage(sender,messageptr,buffer_id);

    result1 = SendAnswer(result1,answer,buffer_id);

    answer = "Answer 2 from ../test/prog4";
    result2 = SendAnswer(result2,answer,buffer_id);
    Exit(0);
}
```

Output:

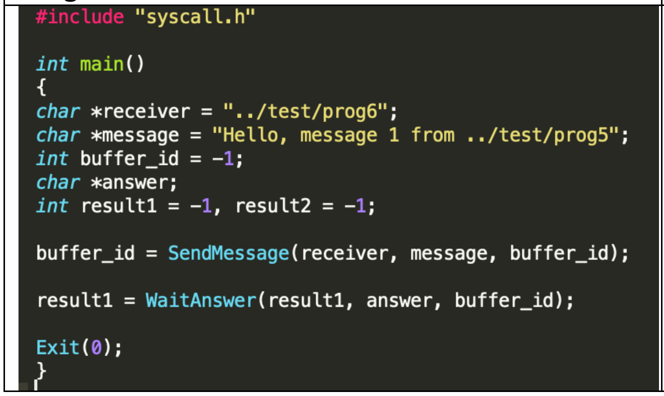


camahore@lcs-vc-cis486: ~/FinalProject/nachos/code/build.linux

```
camahore@lcs-vc-cis486:~/FinalProject/nachos/code/build.linux$ ./nachos -x ../test/prog3 -x ../test/prog4
Forking thread : 3, name:../test/prog3
Forking thread : 4, name:../test/prog4
--------------------------------------------
Buffer ID: 0 created between Threads: ../test/prog3 and ../test/prog4
--------------------------------------------
Message sent from ../test/prog3 to ../test/prog4 , Message: Hello, message 1 from ../test/prog3
Message received by ../test/prog4 sent from ../test/prog3 , Message : Hello, message 1 from ../test/prog3
Message sent from ../test/prog3 to ../test/prog4 , Message: Hello, message 2 from ../test/prog3
Message received by ../test/prog4 sent from ../test/prog3 , Message : Hello, message 2 from ../test/prog3
Answer sent from ../test/prog4 to ../test/prog3, Answer :Answer 1 from ../test/prog4
Answer sent from ../test/prog4 to ../test/prog3, Answer :Answer 2 from ../test/prog4
Exit Called by :../test/prog4
Answer received by ../test/prog3 sent from ../test/prog4 , Answer : Answer 1 from ../test/prog4
Answer received by ../test/prog3 sent from ../test/prog4 , Answer : Answer 2 from ../test/prog4
Exit Called by :../test/prog3
```

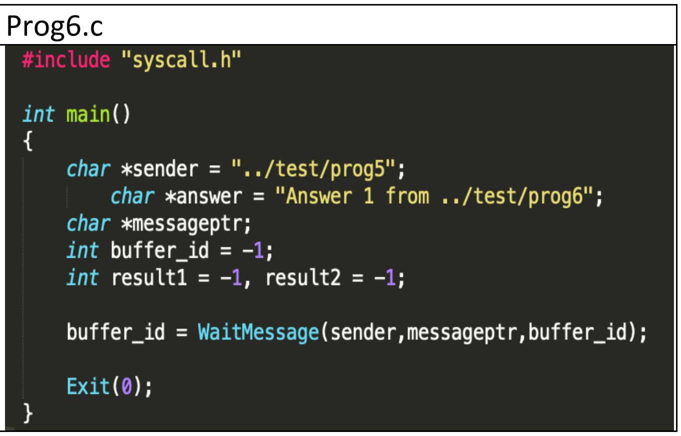
3. In this test case we can see that prog5 sends a message to prog6 which it receives and exits and thus a dummy answer is sent to prog5 as the sender thread is dead.

Prog5.c



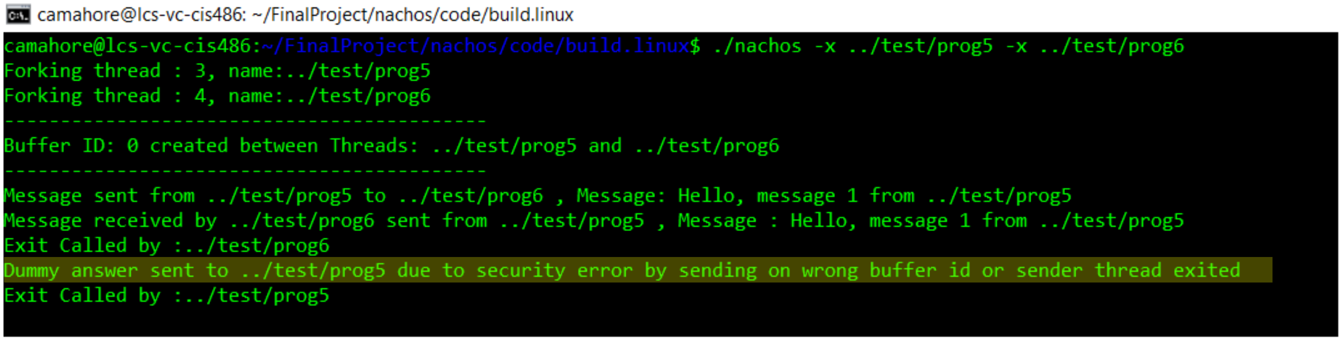
```
#include "syscall.h"

int main()
{
char *receiver = "../test/prog6";
char *message = "Hello, message 1 from ../test/prog5";
int buffer_id = -1;
char *answer;
int result1 = -1, result2 = -1;

buffer_id = SendMessage(receiver, message, buffer_id);

result1 = WaitAnswer(result1, answer, buffer_id);

Exit(0);
}
```

Prog6.c



```
#include "syscall.h"

int main()
{
    char *sender = "../test/prog5";
        char *answer = "Answer 1 from ../test/prog6";
    char *messageptr;
    int buffer_id = -1;
    int result1 = -1, result2 = -1;

    buffer_id = WaitMessage(sender,messageptr,buffer_id);

    Exit(0);
}
```

Output:

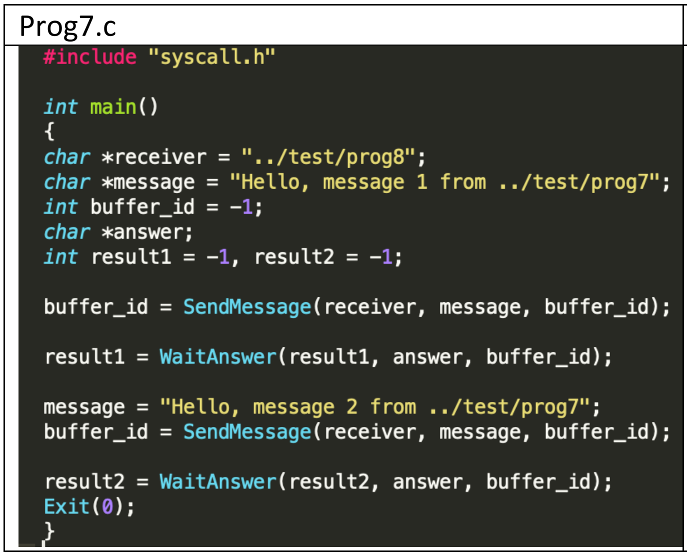


camahore@lcs-vc-cis486: ~/FinalProject/nachos/code/build.linux

```
camahore@lcs-vc-cis486:~/FinalProject/nachos/code/build.linux$ ./nachos -x ../test/prog5 -x ../test/prog6
Forking thread : 3, name:../test/prog5
Forking thread : 4, name:../test/prog6
-------------------------------------------
Buffer ID: 0 created between Threads: ../test/prog5 and ../test/prog6
-------------------------------------------
Message sent from ../test/prog5 to ../test/prog6 , Message: Hello, message 1 from ../test/prog5
Message received by ../test/prog6 sent from ../test/prog5 , Message : Hello, message 1 from ../test/prog5
Exit Called by :../test/prog6
Dummy answer sent to ../test/prog5 due to security error by sending on wrong buffer id or sender thread exited
Exit Called by :../test/prog5
```

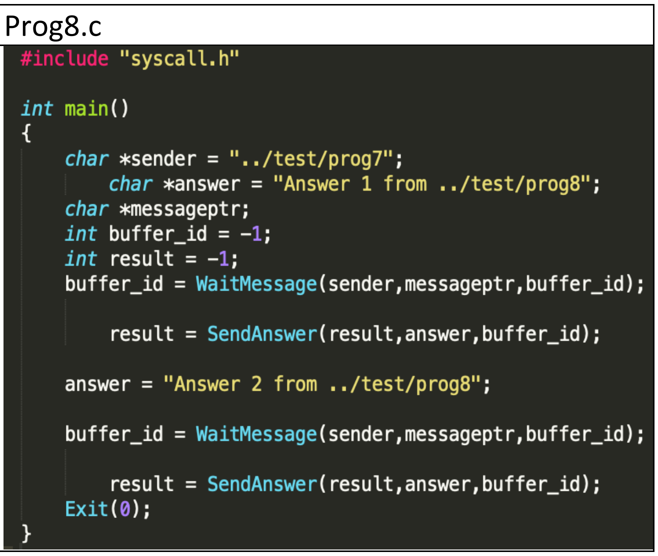
4. In this test case we are sending two messages from prog7 to prog8 and separate answers for each message. We can see that the same buffer is used to send both the messages. Also, prog7 sends one message, then waits for an answer and then sends the second message and waits for its answer whereas prog8 receives the first message then sends an answer, receives the next message and sends the next answer.

Prog7.c



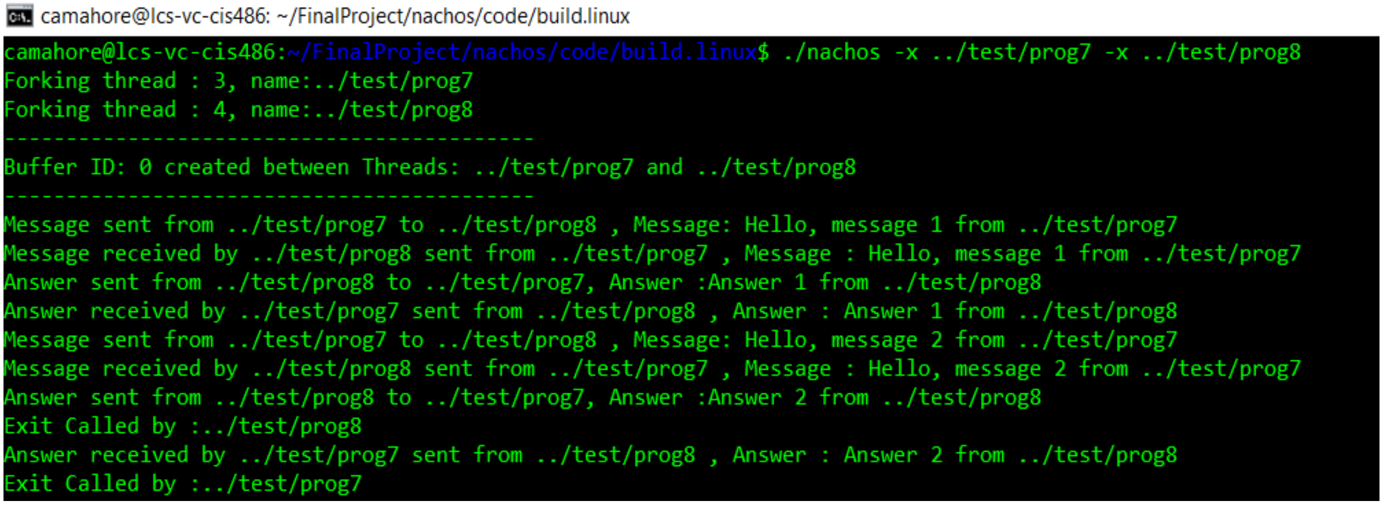
```
#include "syscall.h"

int main()
{
char *receiver = "../test/prog8";
char *message = "Hello, message 1 from ../test/prog7";
int buffer_id = -1;
char *answer;
int result1 = -1, result2 = -1;

buffer_id = SendMessage(receiver, message, buffer_id);

result1 = WaitAnswer(result1, answer, buffer_id);

message = "Hello, message 2 from ../test/prog7";
buffer_id = SendMessage(receiver, message, buffer_id);

result2 = WaitAnswer(result2, answer, buffer_id);
Exit(0);
}
```

Prog8.c



```
#include "syscall.h"

int main()
{
    char *sender = "../test/prog7";
        char *answer = "Answer 1 from ../test/prog8";
    char *messageptr;
    int buffer_id = -1;
    int result = -1;
    buffer_id = WaitMessage(sender,messageptr,buffer_id);

        result = SendAnswer(result,answer,buffer_id);

    answer = "Answer 2 from ../test/prog8";

    buffer_id = WaitMessage(sender,messageptr,buffer_id);

        result = SendAnswer(result,answer,buffer_id);
    Exit(0);
}
```

Output:

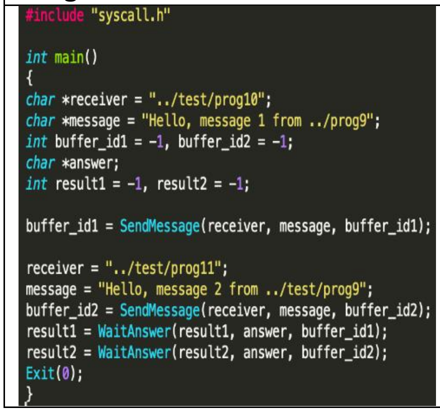


camahore@lcs-vc-cis486: ~/FinalProject/nachos/code/build.linux

```
camahore@lcs-vc-cis486:~/FinalProject/nachos/code/build.linux$ ./nachos -x ../test/prog7 -x ../test/prog8
Forking thread : 3, name:../test/prog7
Forking thread : 4, name:../test/prog8
-------------------------------------------
Buffer ID: 0 created between Threads: ../test/prog7 and ../test/prog8
-------------------------------------------
Message sent from ../test/prog7 to ../test/prog8 , Message: Hello, message 1 from ../test/prog7
Message received by ../test/prog8 sent from ../test/prog7 , Message : Hello, message 1 from ../test/prog7
Answer sent from ../test/prog8 to ../test/prog7, Answer :Answer 1 from ../test/prog8
Answer received by ../test/prog7 sent from ../test/prog8 , Answer : Answer 1 from ../test/prog8
Message sent from ../test/prog7 to ../test/prog8 , Message: Hello, message 2 from ../test/prog7
Message received by ../test/prog8 sent from ../test/prog7 , Message : Hello, message 2 from ../test/prog7
Answer sent from ../test/prog8 to ../test/prog7, Answer :Answer 2 from ../test/prog8
Exit Called by :../test/prog8
Answer received by ../test/prog7 sent from ../test/prog8 , Answer : Answer 2 from ../test/prog8
Exit Called by :../test/prog7
```

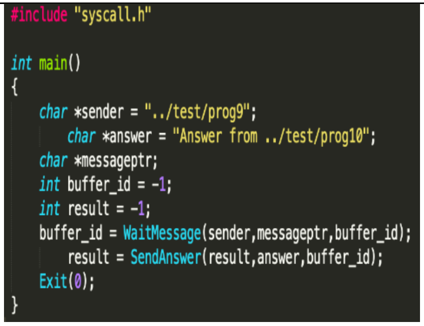
5. In this test case, prog9 sends a message to prog10 and a message to prog11, prog10 receives the message and sends an answer, prog11 receives the message and sends an answer. Communication between prog9 and prog10 and between prog9 and prog11 occurs using two different buffers.

Prog9.c



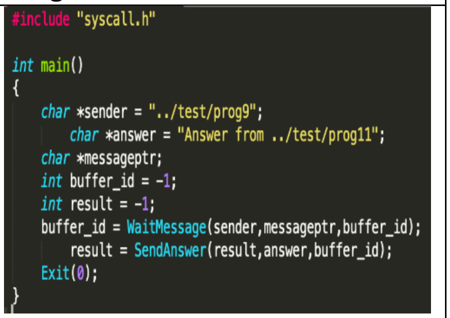
```
#include "syscall.h"

int main()
{
char *receiver = "../test/prog10";
char *message = "Hello, message 1 from ../prog9";
int buffer_id1 = -1, buffer_id2 = -1;
char *answer;
int result1 = -1, result2 = -1;

buffer_id1 = SendMessage(receiver, message, buffer_id1);

receiver = "../test/prog11";
message = "Hello, message 2 from ../test/prog9";
buffer_id2 = SendMessage(receiver, message, buffer_id2);
result1 = WaitAnswer(result1, answer, buffer_id1);
result2 = WaitAnswer(result2, answer, buffer_id2);
Exit(0);
}
```

Prog10.c



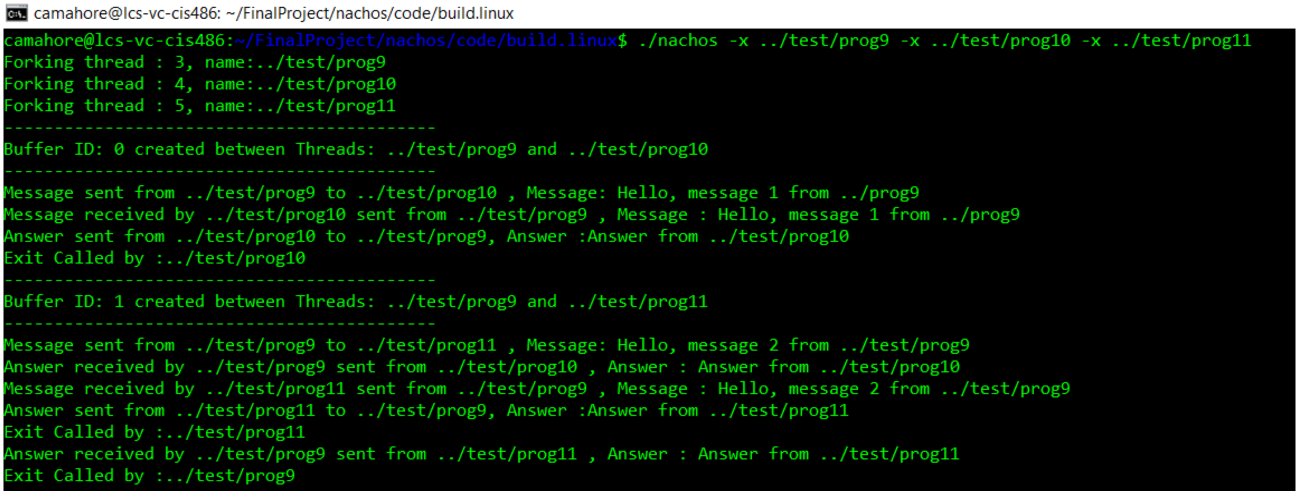
```
#include "syscall.h"

int main()
{
    char *sender = "../test/prog9";
        char *answer = "Answer from ../test/prog10";
    char *messageptr;
    int buffer_id = -1;
    int result = -1;
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
        result = SendAnswer(result,answer,buffer_id);
    Exit(0);
}
```

Prog11.c



```
#include "syscall.h"

int main()
{
    char *sender = "../test/prog9";
        char *answer = "Answer from ../test/prog11";
    char *messageptr;
    int buffer_id = -1;
    int result = -1;
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
        result = SendAnswer(result,answer,buffer_id);
    Exit(0);
}
```

Output:

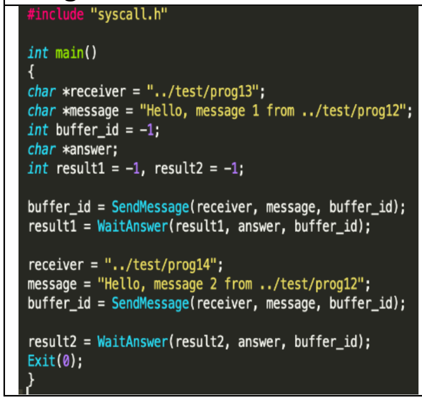


camahore@lcs-vc-cis486: ~/FinalProject/nachos/code/build.linux

```
camahore@lcs-vc-cis486:~/FinalProject/nachos/code/build.linux$ ./nachos -x ../test/prog9 -x ../test/prog10 -x ../test/prog11
Forking thread : 3, name:../test/prog9
Forking thread : 4, name:../test/prog10
Forking thread : 5, name:../test/prog11
-------------------------------------------
Buffer ID: 0 created between Threads: ../test/prog9 and ../test/prog10
-------------------------------------------
Message sent from ../test/prog9 to ../test/prog10 , Message: Hello, message 1 from ../prog9
Message received by ../test/prog10 sent from ../test/prog9 , Message : Hello, message 1 from ../prog9
Answer sent from ../test/prog10 to ../test/prog9, Answer :Answer from ../test/prog10
Exit Called by :../test/prog10
-------------------------------------------
Buffer ID: 1 created between Threads: ../test/prog9 and ../test/prog11
-------------------------------------------
Message sent from ../test/prog9 to ../test/prog11 , Message: Hello, message 2 from ../test/prog9
Answer received by ../test/prog9 sent from ../test/prog10 , Answer : Answer from ../test/prog10
Message received by ../test/prog11 sent from ../test/prog9 , Message : Hello, message 2 from ../test/prog9
Answer sent from ../test/prog11 to ../test/prog9, Answer :Answer from ../test/prog11
Exit Called by :../test/prog11
Answer received by ../test/prog9 sent from ../test/prog11 , Answer : Answer from ../test/prog11
Exit Called by :../test/prog9
```

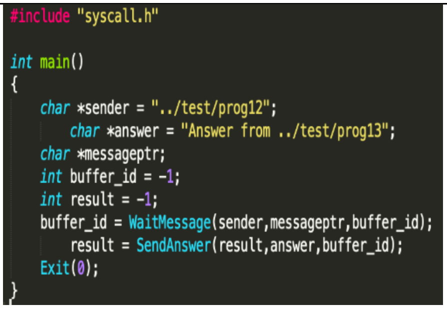
6. In this test case prog12 sends a message to prog13 and prog13 receives the message and sends an answer to prog12. As prog12 sends a message to prog14 on the buffer ID between prog12 and prog13, it causes a security error as the message is being sent on the wrong buffer ID so a dummy answer is sent to prog12 and it exits without sending a message to prog14 so a dummy message is sent to prog14.

Prog12.c



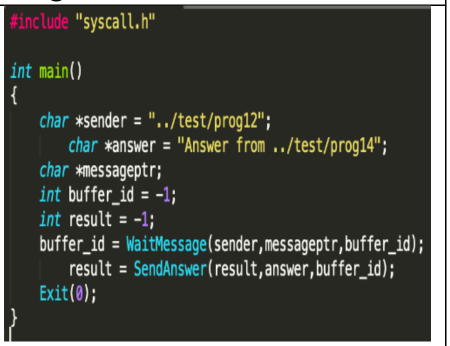
```
#include "syscall.h"

int main()
{
char *receiver = "../test/prog13";
char *message = "Hello, message 1 from ../test/prog12";
int buffer_id = -1;
char *answer;
int result1 = -1, result2 = -1;

buffer_id = SendMessage(receiver, message, buffer_id);
result1 = WaitAnswer(result1, answer, buffer_id);

receiver = "../test/prog14";
message = "Hello, message 2 from ../test/prog12";
buffer_id = SendMessage(receiver, message, buffer_id);

result2 = WaitAnswer(result2, answer, buffer_id);
Exit(0);
}
```

Prog13.c



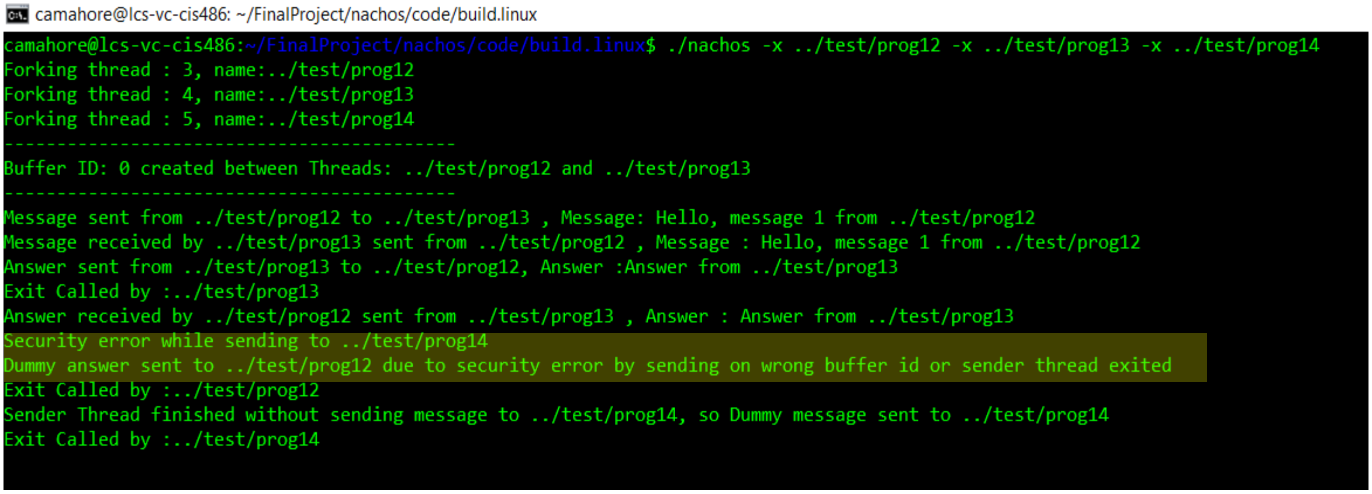
```
#include "syscall.h"

int main()
{
    char *sender = "../test/prog12";
        char *answer = "Answer from ../test/prog13";
    char *messageptr;
    int buffer_id = -1;
    int result = -1;
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
        result = SendAnswer(result,answer,buffer_id);
    Exit(0);
}
```

Prog14.c



```
#include "syscall.h"

int main()
{
    char *sender = "../test/prog12";
        char *answer = "Answer from ../test/prog14";
    char *messageptr;
    int buffer_id = -1;
    int result = -1;
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
        result = SendAnswer(result,answer,buffer_id);
    Exit(0);
}
```

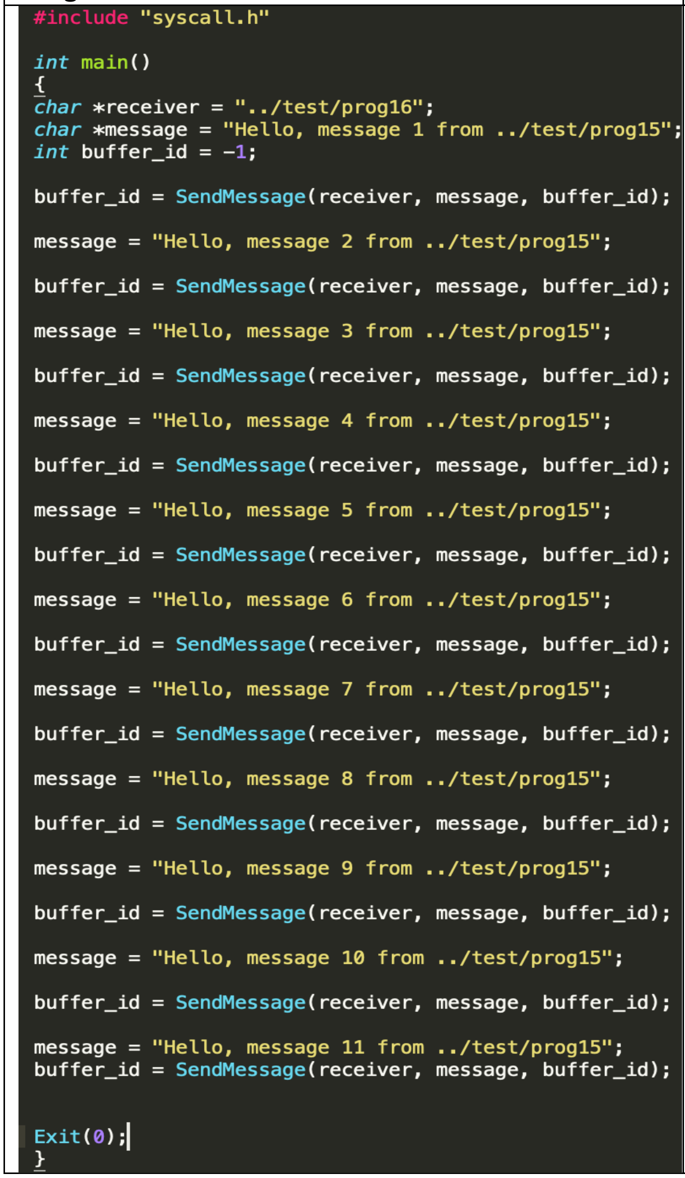
Output:



```
camahore@lcs-vc-cis486: ~/FinalProject/nachos/code/build.linux
camahore@lcs-vc-cis486:~/FinalProject/nachos/code/build.linux$ ./nachos -x ../test/prog12 -x ../test/prog13 -x ../test/prog14
Forking thread : 3, name:../test/prog12
Forking thread : 4, name:../test/prog13
Forking thread : 5, name:../test/prog14
-------------------------------------------
Buffer ID: 0 created between Threads: ../test/prog12 and ../test/prog13
-------------------------------------------
Message sent from ../test/prog12 to ../test/prog13 , Message: Hello, message 1 from ../test/prog12
Message received by ../test/prog13 sent from ../test/prog12 , Message : Hello, message 1 from ../test/prog12
Answer sent from ../test/prog13 to ../test/prog12, Answer :Answer from ../test/prog13
Exit Called by :../test/prog13
Answer received by ../test/prog12 sent from ../test/prog13 , Answer : Answer from ../test/prog13
Security error while sending to ../test/prog14
Dummy answer sent to ../test/prog12 due to security error by sending on wrong buffer id or sender thread exited
Exit Called by :../test/prog12
Sender Thread finished without sending message to ../test/prog14, so Dummy message sent to ../test/prog14
Exit Called by :../test/prog14
```

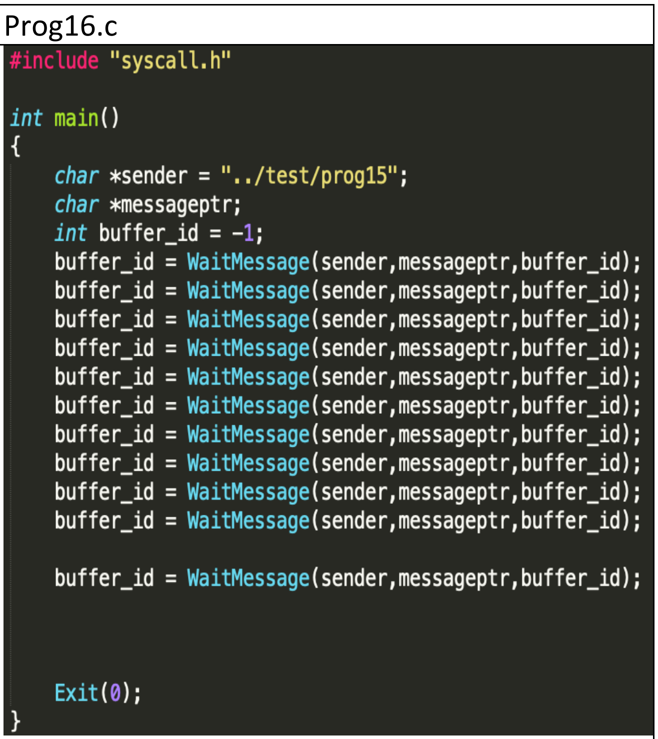
7. In this test case, prog15 tries to send 11 messages to prog16 and as we have defined the message limit as 10, prog16 receives 10 messages and then as the maximum limit is reached, a dummy message is sent to prog16.

Prog15.c



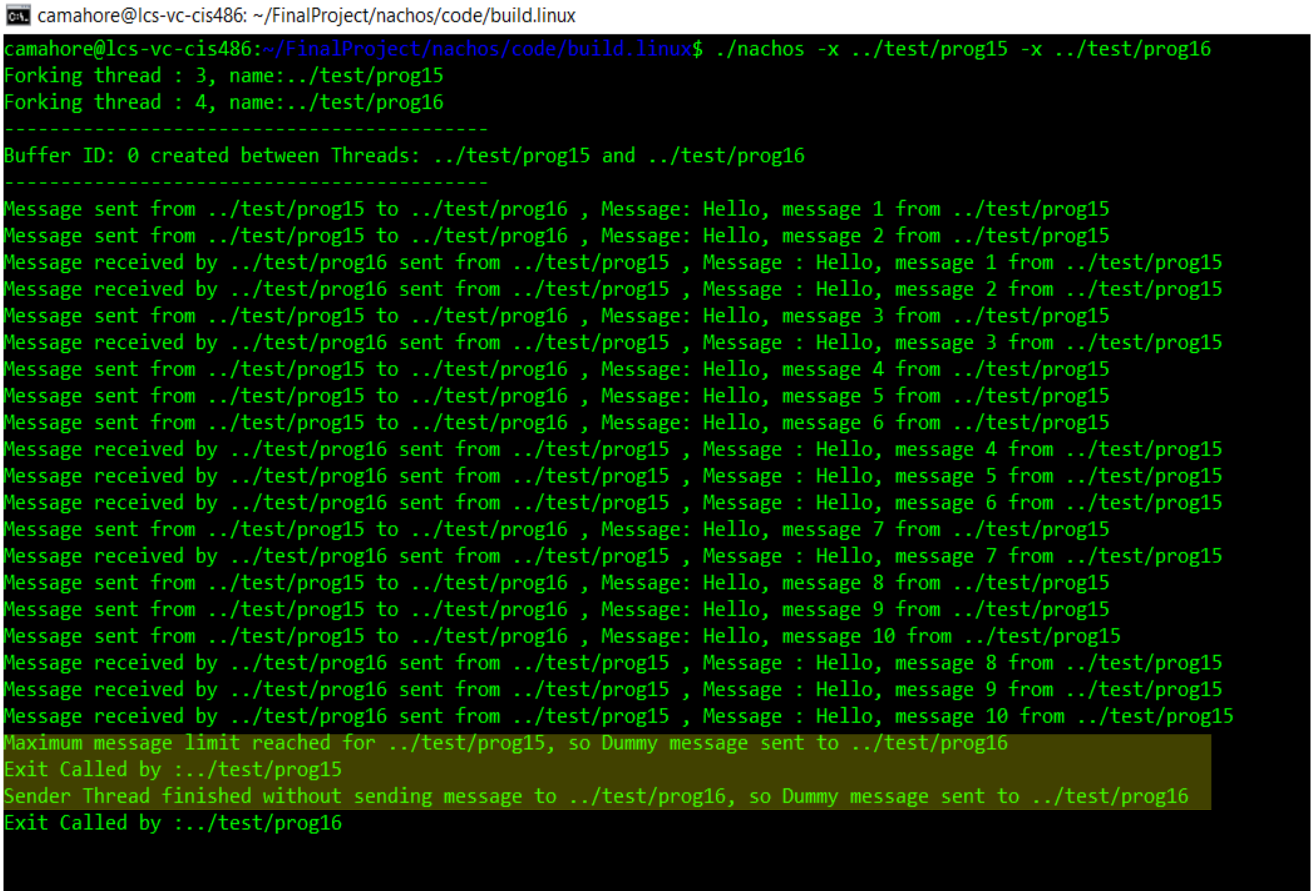
```
#include "syscall.h"

int main()
{
char *receiver = "../test/prog16";
char *message = "Hello, message 1 from ../test/prog15";
int buffer_id = -1;

buffer_id = SendMessage(receiver, message, buffer_id);

message = "Hello, message 2 from ../test/prog15";

buffer_id = SendMessage(receiver, message, buffer_id);

message = "Hello, message 3 from ../test/prog15";

buffer_id = SendMessage(receiver, message, buffer_id);

message = "Hello, message 4 from ../test/prog15";

buffer_id = SendMessage(receiver, message, buffer_id);

message = "Hello, message 5 from ../test/prog15";

buffer_id = SendMessage(receiver, message, buffer_id);

message = "Hello, message 6 from ../test/prog15";

buffer_id = SendMessage(receiver, message, buffer_id);

message = "Hello, message 7 from ../test/prog15";

buffer_id = SendMessage(receiver, message, buffer_id);

message = "Hello, message 8 from ../test/prog15";

buffer_id = SendMessage(receiver, message, buffer_id);

message = "Hello, message 9 from ../test/prog15";

buffer_id = SendMessage(receiver, message, buffer_id);

message = "Hello, message 10 from ../test/prog15";

buffer_id = SendMessage(receiver, message, buffer_id);

message = "Hello, message 11 from ../test/prog15";
buffer_id = SendMessage(receiver, message, buffer_id);


Exit(0);
}
```

Prog16.c



```
#include "syscall.h"

int main()
{
    char *sender = "../test/prog15";
    char *messageptr;
    int buffer_id = -1;
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
    buffer_id = WaitMessage(sender,messageptr,buffer_id);

    buffer_id = WaitMessage(sender,messageptr,buffer_id);



    Exit(0);
}
```

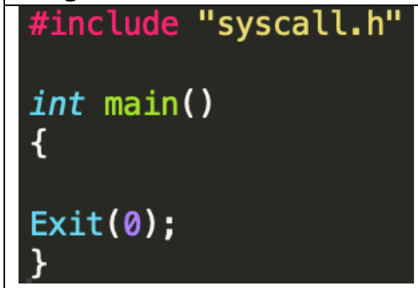
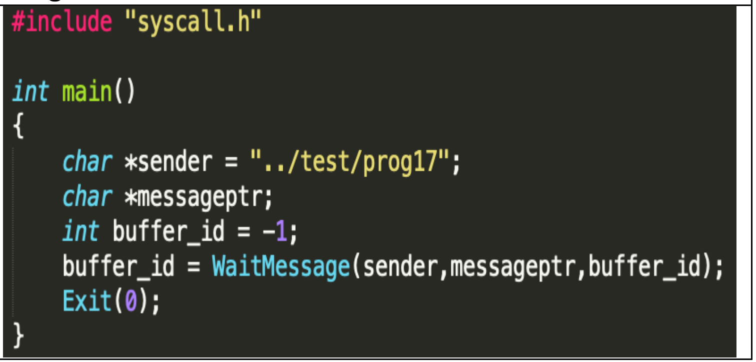
Output:



camahore@lcs-vc-cis486: ~/FinalProject/nachos/code/build.linux

```
camahore@lcs-vc-cis486:~/FinalProject/nachos/code/build.linux$ ./nachos -x ../test/prog15 -x ../test/prog16
Forking thread : 3, name:../test/prog15
Forking thread : 4, name:../test/prog16
-------------------------------------------
Buffer ID: 0 created between Threads: ../test/prog15 and ../test/prog16
-------------------------------------------
Message sent from ../test/prog15 to ../test/prog16 , Message: Hello, message 1 from ../test/prog15
Message sent from ../test/prog15 to ../test/prog16 , Message: Hello, message 2 from ../test/prog15
Message received by ../test/prog16 sent from ../test/prog15 , Message : Hello, message 1 from ../test/prog15
Message received by ../test/prog16 sent from ../test/prog15 , Message : Hello, message 2 from ../test/prog15
Message sent from ../test/prog15 to ../test/prog16 , Message: Hello, message 3 from ../test/prog15
Message received by ../test/prog16 sent from ../test/prog15 , Message : Hello, message 3 from ../test/prog15
Message sent from ../test/prog15 to ../test/prog16 , Message: Hello, message 4 from ../test/prog15
Message sent from ../test/prog15 to ../test/prog16 , Message: Hello, message 5 from ../test/prog15
Message sent from ../test/prog15 to ../test/prog16 , Message: Hello, message 6 from ../test/prog15
Message received by ../test/prog16 sent from ../test/prog15 , Message : Hello, message 4 from ../test/prog15
Message received by ../test/prog16 sent from ../test/prog15 , Message : Hello, message 5 from ../test/prog15
Message received by ../test/prog16 sent from ../test/prog15 , Message : Hello, message 6 from ../test/prog15
Message sent from ../test/prog15 to ../test/prog16 , Message: Hello, message 7 from ../test/prog15
Message received by ../test/prog16 sent from ../test/prog15 , Message : Hello, message 7 from ../test/prog15
Message sent from ../test/prog15 to ../test/prog16 , Message: Hello, message 8 from ../test/prog15
Message sent from ../test/prog15 to ../test/prog16 , Message: Hello, message 9 from ../test/prog15
Message sent from ../test/prog15 to ../test/prog16 , Message: Hello, message 10 from ../test/prog15
Message received by ../test/prog16 sent from ../test/prog15 , Message : Hello, message 8 from ../test/prog15
Message received by ../test/prog16 sent from ../test/prog15 , Message : Hello, message 9 from ../test/prog15
Message received by ../test/prog16 sent from ../test/prog15 , Message : Hello, message 10 from ../test/prog15
Maximum message limit reached for ../test/prog15, so Dummy message sent to ../test/prog16
Exit Called by :../test/prog15
Sender Thread finished without sending message to ../test/prog16, so Dummy message sent to ../test/prog16
Exit Called by :../test/prog16
```

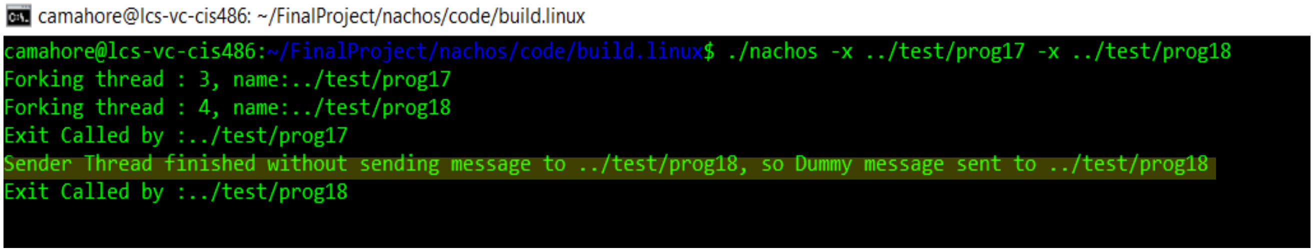
8. In this test case, prog17 exits without sending a message and hence prog18 receives a dummy message as the sender is dead.

Prog17.c	Prog18.c
	

```
#include "syscall.h"

int main()
{

Exit(0);
}
```

```
#include "syscall.h"

int main()
{
    char *sender = "../test/prog17";
    char *messageptr;
    int buffer_id = -1;
    buffer_id = WaitMessage(sender,messageptr,buffer_id);
    Exit(0);
}
```

Output:



```
camahore@lcs-vc-cis486: ~/FinalProject/nachos/code/build.linux
camahore@lcs-vc-cis486:~/FinalProject/nachos/code/build.linux$ ./nachos -x ../test/prog17 -x ../test/prog18
Forking thread : 3, name:../test/prog17
Forking thread : 4, name:../test/prog18
Exit Called by :../test/prog17
Sender Thread finished without sending message to ../test/prog18, so Dummy message sent to ../test/prog18
Exit Called by :../test/prog18
```

CIS657 Fall 2018

Assignment Disclosure Form

Assignment #: Assignment 3

Name: Sonal Patil [SUID: 997435672]

Chetali Mahore [SUID: 750500177]

Naina Bharadwaj [SUID: 810909672]

1. Did you consult with anyone other than instructor or TA/grader on parts of this assignment?
If Yes, please give the details.

No, we did not consult with any other group than instructor or TA

2. Did you consult an outside source such as an Internet forum or a book on parts of this assignment?
If Yes, please give the details.

No, we did not. We just used the nachos source code.

I assert that, to the best of my knowledge, the information on this sheet is true.

Signature: Sonal Patil, Chetali Mahore, Naina Bharadwaj Date : 12/05/2018