1.Objective:

To implement multi-threading capabilities to Nachos. From the Unix point of view, Nachos is a process with user space threads. From the Nachos point of view, Nachos is an operating system that has processes without multi-threading. This assignment requires that you add multi-threading in Nachos as follows:

1. Create a Process class
2. Integrate the Process class with the Nachos Threads and System
3. Change the scheduler

You need to make some extensions to the Nachos operating system.

2.What has been developed:

Process.h:

Declares the variables and function.

Status – holds the current status of process.

PId – contains ID of process

Priority – contain priority of process, name – name of the process, funcPtr – function pointer of the function, arg- argument for the function.

```
/* Process.h
Date structures for managing the Prcoess.
*/
#ifndef PROCESS_H
#define PROCESS_H

#include "list.h"
#include "thread.h"


class Scheduler;

class Process {

  public:
  Process(char* debugName, int pr);       // initialize a Process
  ~Process();                      // deallocate a Process

  Thread *currentThread;
  Scheduler *threadScheduler;

  void setStatus(ThreadStatus st);
      void setPriority(int p);
```

```
  void setPId(int i);
      void setName(char* c);
  void setFuncPtr(VoidFunctionPtr f);
  void setArg(void *a);

      ThreadStatus getStatus();
      int getPriority();
  int getPId();
      char* getName();
  VoidFunctionPtr getFUncPtr();
  void* getArg();

  void Print() { cout << name << "   " ; }
  void printProcess();
  char* printStatus();
  static int compare(Process* p1, Process* p2);
  Process* createChildProcess(char* n);
  void Fork(VoidFunctionPtr func, void *arg);
  void Yield();      // Relinquish the CPU if any other Process is
runnable
  void Terminate();
      void Sleep(bool finishing);

  private:

    ThreadStatus status;      // ready, running or blocked
    char* name;
    int priority;
    int pId;
    VoidFunctionPtr funcPtr;
    static int count;
    void* arg;
};
```

Process.cc: Defined all the functions required for process class

```
//process.cc
//Routine to manage the process

#include "process.h"
#include "switch.h"
#include "synch.h"

int Process::count = 0;
//Constructor for Process
Process::Process(char* processName, int priority_)
```

```
{
    name = processName;
    status = JUST_CREATED;
    priority = priority_;
    threadScheduler = new Scheduler(1);
    currentThread  = new Thread("Thread1");
    pId = count++;
}
//Destructor for process
Process::~Process()
{

}

//setter for status
void Process::setStatus(ThreadStatus st){
    status = st;
}

//setter for Priority
void Process::setPriority(int p){
    priority = p;
}

//setter for PId
void Process::setPId(int i){
    pId = i;
}

//setter for Name
void Process::setName(char* c){
    name = c;
}

 //setter for funcPtr
void Process::setFuncPtr(VoidFunctionPtr f){
    funcPtr = f;
}

//setter for arg
void Process::setArg(void *a){
    arg = a;
}

//getter for status
ThreadStatus Process::getStatus(){
```

```
    return status;
}

//getter for Priority
int Process::getPriority(){
    return priority;
}

//getter for PId
int Process::getPId(){
    return pId;
}

//getter for name
char* Process::getName(){
    return name;
}

//getter for funcPtr
VoidFunctionPtr Process::getFUncPtr(){
    return funcPtr;
}

//getter for arg
void* Process::getArg(){
    return arg;
}
//print the status of process
char* Process::printStatus(){
    switch(status){
        case 0: return "JUST_CREATED";
                break;
        case 1: return "RUNNING";
                break;
        case 2: return "READY";
                break;
        case 3: return "BLOCKED";
                break;
        default: return "NOT DEFINED";
    }
}
//This function is used while inserting the process in sorted list
//This function comapres Priorities of process.
int Process::compare(Process* p1, Process* p2){

    int p1Priority = p1->getPriority();
```

```
    int p2Priority = p2->getPriority();

    if(p1Priority > p2Priority)
        return -1;
    if(p1Priority == p2Priority)
        return 0;
    else
        return 1;
}

//This method forks the thread in given process
void Process::Fork(VoidFunctionPtr func, void *arg)
{
    Interrupt *interrupt = kernel->interrupt;
    Scheduler *scheduler = kernel->scheduler;
    IntStatus oldLevel;
    this->setFuncPtr(func);
    this->setArg(arg);

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

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

//This methos yields the process
void Process::Yield ()
{
    Process *nextProcess;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentProcess);

    DEBUG(dbgThread, "Yielding Process: " << name);
    cout <<  "Yielding Process: " << name<< endl;

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

```
}

//Print the Process details
void ProcessPrint(Process *t) { t->Print(); }

//Terminate the Process
void Process::Terminate ()
{
    Thread *nextThread;
    (void) kernel->interrupt->SetLevel(IntOff);
    ASSERT(this == kernel->currentProcess);
    bool finished = TRUE;

    Thread *cThread = kernel->currentProcess->currentThread;
    cout << "Finishing Thread: " << cThread->getName() << endl;
    if(finished){
        cout << "Terminating Process: " << name << endl;
        kernel->currentProcess->Sleep(TRUE);
    }

}
//Put the process on SLeep
void Process::Sleep (bool finishing)
{
    Process *nextProcess;

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

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

    status = BLOCKED;
    while ((nextProcess = 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(nextProcess, finishing);
}

//Create the child process
Process* Process::createChildProcess(char* cName){
     Process *p = new Process(cName,this->getPriority());
     p->Fork(this->getFUncPtr(),this->getArg());
     p->printProcess();
}

//Print the process details
```

```
void Process::printProcess(){
    cout << "----------------------------------\nProcess Details:\n";
    cout << "PId: "<< this->getPId() << ", Name: "<< this->getName() ;
    cout << ", Priority: " << this->getPriority();
    cout << ", Status: " << this->printStatus() << endl;
}
```

Thread.h

```
void printThread();
char* printStatusThread();
```

thread.cc : Modified functions to accommodate multithreading and process

changed `kernel->currentThread` to `kernel->currentProcess->currentThread`

changed `kernel->scheduler` to `kernel->currentProcess->threadScheduler`

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

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

    StackAllocate(func, arg);

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

//fork the thread in given process.

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

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

```
    StackAllocate(func, arg);

    oldLevel = interrupt->SetLevel(IntOff);
    scheduler->ReadyToRunT(this);   // ReadyToRun assumes that
interrupts
                    // are disabled!
    (void) interrupt->SetLevel(oldLevel);
}
void
Thread::Yield ()
{
    Thread *nextThread;
    IntStatus oldLevel = kernel->interrupt->SetLevel(IntOff);

    ASSERT(this == kernel->currentProcess->currentThread);

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

    nextThread = kernel->currentProcess->threadScheduler-
>FindNextToRunT();
    if (nextThread != NULL) {
         nextThread = kernel->currentProcess->threadScheduler-
>FindNextToRunT();
       if(nextThread != NULL){
              kernel->currentProcess->threadScheduler->RunT(nextThread,
FALSE);
            kernel->currentProcess->threadScheduler->ReadyToRunT(this);
        }
    }
    (void) kernel->interrupt->SetLevel(oldLevel);
}
void
Thread::Sleep (bool finishing)
{
    Thread *nextThread;

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

    cout<< "Sleeping thread: " << name << endl;

    status = BLOCKED;
    while ((nextThread = kernel->currentProcess->threadScheduler-
>FindNextToRunT()) == NULL)
```

```
      kernel->interrupt->Idle();    // no one to run, wait for an
interrupt
    cout<< "After while thread: " << name << endl;
    // returns when it's time for us to run
    kernel->currentProcess->threadScheduler->RunT(nextThread,
finishing);
}

//Return the status of Thread
char* Thread::printStatusThread(){
    switch(status){
        case 0: return "JUST_CREATED";
                break;
        case 1: return "RUNNING";
                break;
        case 2: return "READY";
                break;
        case 3: return "BLOCKED";
                break;
        default: return "NOT DEFINED";
    }
}

//Print the thread details
void Thread::printThread(){
    cout << "-----------------------------------\nThread Details:\n";
    cout << "Name: "<< name;
    cout << ", Status: " << printStatusThread() << endl;
}
```

threadTest.cc

```
#include "kernel.h"
#include "main.h"

//Test Function
void
SimpleThread(int which)
{
    int num;
    printf("In function SimpleThread\n");
    kernel->currentProcess->currentThread->Yield();
    kernel->currentProcess->Terminate();
}
//Test Function
void fun1(int which){
    cout << "\nIn function 1 \n";
```

```
    kernel->currentProcess->currentThread->Yield();
    kernel->currentProcess->Terminate();
}

//Runs the testcases
void
ThreadTest()
{
    cout<< "\nProcess1 Created \n";
    Process *p1 = new Process("Process1",1);
    p1->printProcess();
    cout << "Thread Forked for Process2\n";
    p1->Fork((VoidFunctionPtr) fun1, (void *) 1);
    p1->printProcess();
    cout<< "Create ChildProcess for Process1 (Process Fork)\n";
    p1->createChildProcess("Child Process1");
    cout<< "Process2 Created \n";
    Process *p2 = new Process("Process2",2);
    p2->printProcess();
    cout << "Thread Forked for Process2\n";
    p2->Fork((VoidFunctionPtr) SimpleThread, (void *) 1);
    p2->printProcess();
    cout << "Thread Forked for Process2\n";
    Thread *t = new Thread("Thread2");
    t->Fork((VoidFunctionPtr) SimpleThread, (void *) 1, p2);
    t->printThread();
    cout<< "Process2 Created \n";
    Process *p3 = new Process("Process3",3);
    p3->printProcess();
    cout << "Thread Forked for Process3\n";
    p3->Fork((VoidFunctionPtr) SimpleThread, (void *) 1);
    p3->printProcess();
    kernel->currentProcess->Yield();
}
```

Scheduler.h
Added functions for scheduling processes and modified functions for threads. Added sorted list for process.

```
void Print();           // Print contents of ready list
    void PrintThread();

    // SelfTest for scheduler is implemented in class Thread

    Scheduler(int i);
    void ReadyToRunT(Thread* thread);
                    // Thread can be dispatched.
```

```
    Thread* FindNextToRunT();   // Dequeue first thread on the ready
                // list, if any, and return thread.
    void RunT(Thread* nextThread, bool finishing);
SortedList<Process*> *readyList;
    Process *toBeDestroyProcess;
    List<Thread *> *readyListT;
```

Scheduler.cc: Modified functions to Accommodate new changes and defined new functions

```
Scheduler::Scheduler()
{
    //readyList = new List<Process *>;
    readyList=  new SortedList<Process*>(Process::compare);
    toBeDestroyProcess = NULL;
}

//Initialize the list of ready but not running threads.
Scheduler::Scheduler(int i)
{
    readyListT = new List<Thread *>;
    toBeDestroyed = NULL;
}
Scheduler::~Scheduler()
{
    delete readyListT;
    delete readyList;
}

void
Scheduler::ReadyToRunT (Thread *thread)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
    DEBUG(dbgThread, "Putting thread on ready list: " << thread-
>getName());

    thread->setStatus(READY);
    readyListT->Append(thread);
}

//ReadyToRun for process
void
Scheduler::ReadyToRun (Process *process)
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);
```

```
    DEBUG(dbgThread, "Putting Process on ready list: " << process-
>getName());

    process->setStatus(READY);
    process->currentThread->setStatus(READY);
    readyList->Insert(process);
}


Thread *
Scheduler::FindNextToRunT ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    if (readyListT->IsEmpty()) {
    return NULL;
    } else {
        return readyListT->RemoveFront();
    }
}

//Find the next prcoess from the readyList
Process *
Scheduler::FindNextToRun ()
{
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    if (readyList->IsEmpty()) {
    return NULL;
    } else {
        return readyList->RemoveFront();
    }
}



void
Scheduler::RunT (Thread *nextThread, bool finishing)
{
    Thread *oldThread = kernel->currentProcess->currentThread;

    cout << "\nCurrent Thread: "<< oldThread->getName()<< endl ;
    cout << "Next Thread: " << nextThread->getName() << endl;
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    if (finishing) {    // mark that we need to delete current thread
```

```
        ASSERT(toBeDestroyed == NULL);
     toBeDestroyed = oldThread;
    }

    if (oldThread->space != NULL) { // if this thread is a user
program,
        oldThread->SaveUserState();     // save the user's CPU
registers
    oldThread->space->SaveState();
    }

    oldThread->CheckOverflow();         // check if the old thread
                        // had an undetected stack overflow

    kernel->currentProcess->currentThread = nextThread;  // switch to
the next thread
    nextThread->setStatus(RUNNING);      // nextThread is now running

    DEBUG(dbgThread, "Switching from: " <<kernel->currentProcess << " "
<< oldThread->getName() << " to: " <<  kernel->currentProcess << " "<<
nextThread->getName());
    cout<<"Switching from: " <<kernel->currentProcess->getName() << " "
<< oldThread->getName() << " to: " <<  kernel->currentProcess->getName(
) << " "<< nextThread->getName()<< endl;
    nextThread->printThread();
    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside
world".

    SWITCH(oldThread, nextThread);


    // we're back, running oldThread

    // interrupts are off when we return from switch!
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

    CheckToBeDestroyed();       // check if thread we were running
                    // before this one has finished
                    // and needs to be cleaned up

    if (oldThread->space != NULL) {     // if there is an address space
```

```
        oldThread->RestoreUserState();     // to restore, do it.
    oldThread->space->RestoreState();
    }
}

//Switch to the next process
void
Scheduler::Run (Process *nextProcess, bool finishing)
{
    Thread *oldThread = kernel->currentProcess->currentThread;
    Process *oldProcess = kernel->currentProcess;
    cout<< "-----------------------------------\n";
    cout << "Current Process: "<<kernel->currentProcess->getName()<<
endl ;
    cout << "Next Process: " << nextProcess->getName() << endl;
    printf("-----------------------------------\n");
    kernel->scheduler->Print();
    printf("\n-----------------------------------\n");

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

    if (finishing) {    // mark that we need to delete current thread
        toBeDestroyProcess = NULL;
        ASSERT(toBeDestroyProcess == NULL);
        toBeDestroyProcess = oldProcess;
    }
    if (oldThread->space != NULL) {   // if this thread is a user
program,
        oldThread->SaveUserState();     // save the user's CPU
registers
    oldThread->space->SaveState();
    }

    //oldThread->CheckOverflow();           // check if the old thread
                        // had an undetected stack overflow

    kernel->currentProcess = nextProcess;
    Thread *nextThread = nextProcess->currentThread;

    kernel->currentProcess->currentThread = nextThread;  // switch to
the next thread
    nextProcess->currentThread->setStatus(RUNNING);      // nextThread
is now running
    nextProcess->setStatus(RUNNING);
```

```
    DEBUG(dbgThread, "Switching from: " << oldProcess->getName() <<"
"<< oldThread->getName() << " to: " << nextProcess->getName() << " " <<
nextThread->getName());
    cout <<"Process Switching from: " << oldProcess->getName() <<" "<<
oldThread->getName() << " to: " << nextProcess->getName() << " " <<
nextThread->getName() << endl;
    nextProcess->printProcess();
    nextProcess->currentThread->printThread();
    // This is a machine-dependent assembly language routine defined
    // in switch.s.  You may have to think
    // a bit to figure out what happens after this, both from the point
    // of view of the thread and from the perspective of the "outside
world".
     SWITCH(oldThread, nextThread);

    // we're back, running oldThread

    // interrupts are off when we return from switch!
    ASSERT(kernel->interrupt->getLevel() == IntOff);

    DEBUG(dbgThread, "Now in thread: " << oldThread->getName());

    CheckToBeDestroyedProcess();

    if (oldThread->space != NULL) {       // if there is an address
space
        oldThread->RestoreUserState();     // to restore, do it.
    oldThread->space->RestoreState();
    }
}

//Check the prcoess to be destroyed and delete it
void
Scheduler::CheckToBeDestroyedProcess()
{
    if (toBeDestroyProcess != NULL) {
        delete toBeDestroyProcess;
    toBeDestroyProcess = NULL;
    }
}

//----------------------------------------------------------------------
-
// Scheduler::Print
//  Print the scheduler state -- in other words, the contents of
//  the ready list.  For debugging.
```

```
//----------------------------------------------------------------------
-
void
Scheduler::Print()
{
    cout << "Ready list contents:\n";
    readyList->Apply(ProcessPrint);
}
```

Kernel.h :

```
Process *currentProcess;  // the process holding the CPU
```

Kernel.cc: Created new process main and code for handling the -quantum flag. Stored the argument from command line passed it to alarm class.

```
quantum=100;
else if (strcmp(argv[i], "-quantum") == 0){
        ASSERT(i + 1 < argc);
        RandomInit(atoi(argv[i + 1]));
        quantum = atoi(argv[i + 1]);
        i++;
        }

alarm = new Alarm(randomSlice,quantum);

currentProcess = new Process("main", 0);
currentProcess->setStatus(RUNNING);
```

main.cc:

```
kernel->currentProcess->Terminate();
```

synch.c and synch.cc:

changed `kernel->currentThread` to `kernel->currentProcess->currentThread`

changed `kernel->scheduler` to `kernel->currentProcess->threadScheduler`

timer.h and timer.cc: code to handle the quantum

```
int quantum;

Timer::Timer(bool doRandom, CallBackObj *toCall, int q)

{

    randomize = doRandom;
```

```
    callPeriodically = toCall;
    disable = FALSE;
    quantum = q;
    cout << "\nQuantum: " << quantum;
    SetInterrupt();
}
if (!disable) {
      int delay;
      delay = quantum;
       if (randomize) {
       delay = 1 + (RandomNumber() % (quantum * 2));
        }
       // schedule the next timer device interrupt
       kernel->interrupt->Schedule(this, delay, TimerInt);
    }
```

interrupt.cc, addrspace.cc and mipssim.cc:

changed `kernel->currentThread` to `kernel->currentProcess->currentThread`

changed `kernel->scheduler` to `kernel->currentProcess->threadScheduler`

alarm.h and alarm.cc: Pass the quantum to timer.

```
Alarm(bool doRandomYield,int quantum);
Alarm::Alarm(bool doRandom, int quantum)
{
    timer = new Timer(doRandom, this, quantum);
}
```

3.How to test your solution:

Run following commands to run the code:

cd nachos/code/build.linux
make
./nachos -K -quantum 1000

Output will be displayed on the terminal in the format given in section 5.

4.Files modified / Added:

Modified files:
nachos/code/build.linux/Makefile
nachos/code/threads/threadtest.cc
nachos/code/threads/thread.cc
nachos/code/threads/thread.h
nachos/code/threads/scheduler.cc
nachos/code/threads/scheduler.h
nachos/code/threads/kernel.cc
nachos/code/threads/kernel.h
nachos/code/threads/main.cc
nachos/code/threads/synch.h
nachos/code/threads/synch.cc
nachos/code/threads/alarm.h
nachos/code/threads/alarm.cc
nachos/code/machine/timer.h
nachos/code/machine/timer.cc
nachos/code/machine/interrupt.cc
nachos/code/machine/mipssim.cc
nachos/code/ userprog /addrspace.cc

Added files:
nachos/code/threads/process.cc
nachos/code/threads/process.h

5.Output:

Output is printed in following format:
Process1 – Priority 1, Created child process for process1
Process2 – Priority 2, Thread forked in same process.
Process3 – Priority 3

Processes will run according to the priority.

Processes created

```
gdamberk@lcs-vc-cis486:~/Ass1/nachos/code/build.linux$ ./nachos -K -quantum 1000

Quantum: 1000
Process1 Created
-----------------------------------
Process Details:
PId: 1, Name: Process1, Priority: 1, Status: JUST_CREATED
Thread Forked for Process2
-----------------------------------
Process Details:
PId: 1, Name: Process1, Priority: 1, Status: READY
Create ChildProcess for Process1 (Process Fork)
-----------------------------------
Process Details:
PId: 2, Name: Child Process1, Priority: 1, Status: READY
Process2 Created
-----------------------------------
Process Details:
PId: 3, Name: Process2, Priority: 2, Status: JUST_CREATED
Thread Forked for Process2
-----------------------------------
Process Details:
PId: 3, Name: Process2, Priority: 2, Status: READY
Thread Forked for Process2
-----------------------------------
Thread Details:
Name: Thread2, Status: READY
Process2 Created
-----------------------------------
Process Details:
PId: 4, Name: Process3, Priority: 3, Status: JUST_CREATED
Thread Forked for Process3
-----------------------------------
Process Details:
PId: 4, Name: Process3, Priority: 3, Status: READY
Yielding Process: main
```

Process 3, thread1 is runnning

```
-----------------------------------
Current Process: main
Next Process: Process3
-----------------------------------
Ready list contents:
Process2  Process1  Child Process1  main
-----------------------------------
Process Switching from: main Thread1 to: Process3 Thread1
-----------------------------------
Process Details:
PId: 4, Name: Process3, Priority: 3, Status: RUNNING
-----------------------------------
Thread Details:
Name: Thread1, Status: RUNNING
In function SimpleThread
Yielding thread: Thread1
Finishing Thread: Thread1
Terminating Process: Process3
-----------------------------------
```

Process2 , thread1 and thread2 are running

```
-----------------------------------
Current Process: Process3
Next Process: Process2
-----------------------------------
Ready list contents:
Process1  Child Process1  main
-----------------------------------
Process Switching from: Process3 Thread1 to: Process2 Thread1
-----------------------------------
Process Details:
PId: 3, Name: Process2, Priority: 2, Status: RUNNING
-----------------------------------
Thread Details:
Name: Thread1, Status: RUNNING
In function SimpleThread
Yielding thread: Thread1

Current Thread: Thread1
Next Thread: Thread2
Switching from: Process2 Thread1 to: Process2 Thread2
-----------------------------------
Thread Details:
Name: Thread2, Status: RUNNING
In function SimpleThread
Yielding thread: Thread2
Finishing Thread: Thread2
Terminating Process: Process2
-----------------------------------
```

Process1, thread1 and Child process1, thread 1 are running

```
Current Process: Process2
Next Process: Process1
-----------------------------------
Ready list contents:
Child Process1  main
-----------------------------------
Process Switching from: Process2 Thread2 to: Process1 Thread1
-----------------------------------
Process Details:
PId: 1, Name: Process1, Priority: 1, Status: RUNNING
-----------------------------------
Thread Details:
Name: Thread1, Status: RUNNING

In function 1
Yielding thread: Thread1
Finishing Thread: Thread1
Terminating Process: Process1
-----------------------------------
Current Process: Process1
Next Process: Child Process1
-----------------------------------
Ready list contents:
main
-----------------------------------
Process Switching from: Process1 Thread1 to: Child Process1 Thread1
-----------------------------------
Process Details:
PId: 2, Name: Child Process1, Priority: 1, Status: RUNNING
-----------------------------------
Thread Details:
Name: Thread1, Status: RUNNING

In function 1
Yielding thread: Thread1
Finishing Thread: Thread1
Terminating Process: Child Process1
-----------------------------------
```

Processes are completed terminating main.

```
-----------------------------------
Ready list contents:

-----------------------------------
Process Switching from: Child Process1 Thread1 to: main Thread1
-----------------------------------
Process Details:
PId: 0, Name: main, Priority: 0, Status: RUNNING
-----------------------------------
Thread Details:
Name: Thread1, Status: RUNNING
Finishing Thread: Thread1
Terminating Process: main
```

6.Signed disclosure form:

CIS657 Fall 2018

Assignment Disclosure Form

Assignment #: 1

Name: Gauri Amberkar

1. Did you consult with anyone other than instructor or TA/grader on parts of this assignment?

If Yes, please give the details.

- No

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.

- For switch case - https://syntaxdb.com/ref/cpp/switch

- For variable scope - https://www.tutorialspoint.com/cplusplus/cpp_variable_scope.htm

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

Signature: Gauri Amberkar Date : 11/04/2018