CS1550 Assignment 5
Multiprogramming
Implementation notes
Matt Craven
Multiprogramming

- The goal of this project is to take the given Nachos code, which supports single tasks, and modify it so that multiple tasks can run simultaneously
- Use preemption
- Round-robin scheduling
Nachos Scheduler

- The Nachos scheduler already supports multiple kernel threads (Thread class)
- Threads can Yield to each other
- Scheduler is FIFO
- No preemption
Round robin scheduling

- Round robin scheduling runs tasks in the order they are submitted
- A fixed quantum is used for each process
- When the quantum expires, the current task yields the processor, and the next process is selected from the head of the ready queue
- The yielding task is put at the end of the ready queue
Round robin in Nachos

• As you may have noticed, round robin is simply FIFO scheduling with preemption
• You simply need a way to preempt a process based on a fixed timer interrupt
• Use the built-in Timer class
Nachos Timer

• Nachos uses a timer interrupt when you enable the -rs flag (remember project 2?)
• This turns on a random timer interrupt
• See the line in system.cc that looks like this:
  – Timer = new Timer(TimerInterruptHandler, 0, randomYield)
• You want a non-random timer interrupt
Nachos Timer

• Study the code from system.cc in the last slide, and pay attention to what it does, how you declare a timer, and what TimerInterruptHandler does

• Your timer can be created with one line of code similar to one in the last slide
Starting a Nachos Process

• When you start nachos with the -x argument and an executable file name, it will call the StartProcess() function in userprog/progtest.cc
• Pay close attention to what StartProcess does
• You will need to modify this code
StartProcess

- Remember that when you start Nachos, it automatically creates a “main” kernel thread that takes care of initializing Nachos and running the user program, turning on debugging, etc.
- StartProcess initializes a new AddrSpace for the executable, and then executes machine->Run()
StartProcess

• Take a look at the comments for the machine->Run() line
• Notice that when you call this method, it never returns
• Not good! If you can only call StartProcess once, you can only run one program at a time
StartProcess

- StartProcess is run within the “main” kernel thread
- StartProcess needs to return so that if there is another program to be run, we can call StartProcess again
- Solution: create a new thread for each user program so that the “main” thread can be used to call StartProcess more than once
More in StartProcess

• StartProcess performs a few tasks other than calling machine->Run()
• Calls AddrSpace() to set up an address space for the executable
• Sets up register values for your executable - be careful, how this is done will change when you start creating your own threads!
AddrSpace()

- Even though we’re not using virtual memory for this assignment, we still need a page table for each executable
- This saves us the trouble of worrying about PIDs and memory protection
- When threads are context-switched, the page table from the thread we switch to is used
AddrSpace

• Since we have essentially unlimited physical memory in this assignment, you need to worry about which physical pages are free
• You may use a bitmap for this
• You need to load the code and data from the executable like the last assignment
AddrSpace

- You also need to make sure that your addrspace has numPages of physical pages associated with it (ie, used in the pageTable) to hold all the code, data, uninitData and stack space
- You will set up your page table statically: each page in the pageTable will point to a valid physical page
AddrSpace

• **IMPORTANT**: When loading code and data from the executable into physical memory:
  – The last page of code will probably not be full, so that the last page of code will share space with the first page of initData
  – The last page of initData will also probably not be full, so it will be shared with the first page of uninitData
System call support

• You are also required to implement system call handlers (in exception.cc) for the Open, Read and Write system calls.

• For this, you simply need to
  – get the parameters for the system call from registers
  – convert any pointers (ie, char* pointers) from virtual to physical addresses
  – Pass the parameters to the appropriate UNIX system call (see manpages for open, read and write)
  – Place the return value in the appropriate register
Converting from virtual to physical addresses

• Since our user programs use virtual addresses for pointers, if a system call passes a pointer, the kernel must convert the pointer to a physical address (ie, an offset in machine->mainMemory)

• You cannot use the Translate() call to do this; you have to do it yourself
Converting from virtual to physical addresses

- Even though you can’t call Translate() yourself, you can translate addresses the same way Translate does, by looking up the virtual page in the pageTable, getting the physical page number, and then adding the offset.

- Look at the code in translate.cc for details.
System call support

• System call support is not necessary for multiprogramming, but you will probably want to get the system call part done first

• It’s very simple to get working
Project Summary

• Add the interrupt timer (system.cc)
• Change progtest.cc so that you create a new thread for each executable (progtest.cc)
• Create an address space for each executable, and load its code and data into memory, while setting up the page table (addrspace.cc)
• Add support for Open, Read and Write system calls (exception.cc)
Start Early

• For individuals, the project is due on April 23rd, at midnight
• For groups, the project is due on April 18th, at midnight
• Start now
• Come to lab sessions, recitations, and office hours