

# CMPS 455

Andrew Shullaw

August 30, 2021

## Homework 1

1

- 1.1 Modify the threadtest.cc file such that three new threads are created and each thread executes a different function. Upload your updated threadtest.cc file along with a snapshot of your Nachos run (you can copy-paste these in the same document that you are answering the other questions).

```
student@nachos: ~/nachos/nachos-3.4/code/threads
student@nachos:~/nachos/nachos-3.4/code/threads$ nachos -tt 3
*** THIS IS FUNCTION: 134520760 --> SimpleThreadOG!
*** thread 1 looped 0 times
*** thread 1 looped 1 times
*** thread 1 looped 2 times
*** thread 1 looped 3 times
*** thread 1 looped 4 times
*** THIS IS FUNCTION: 134520852 --> SimpleThreadYield!
*** thread 42 looped 0 times
*** THIS IS FUNCTION: 134521021 --> SimpleThreadPrint!
*** The user initialized which_func: 3
*** THIS IS FUNCTION: 134520852 --> SimpleThreadYield!
*** thread 42 looped 1 times
*** thread 42 looped 2 times
*** thread 42 looped 3 times
*** thread 42 looped 4 times
*** FUNCTION: 134520852 --> SimpleThreadYield is complete!
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 120, idle 0, system 120, user 0
Disk I/O: reads 0, writes 0
Console I/O: reads 0, writes 0
Paging: faults 0
Network I/O: packets received 0, sent 0

Cleaning up...
student@nachos:~/nachos/nachos-3.4/code/threads$
```

```

// threadtest.cc
//      Simple test case for the threads assignment.
//
//      Create two threads, and have them context switch
//      back and forth between themselves by calling Thread::Yield,
//      to illustrate the inner workings of the thread system.
//
// Copyright (c) 1992–1993, 2021 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 "system.h"

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

// void SimpleThread(int which)
// {
//     int num;
//
//     for (num = 0; num < 5; num++)    // HOMEWORK 1
//     {
//         printf("*** thread %d looped %d times\n", which, num);
//         currentThread->Yield();
//     }
// }
// HOMEWORK 1
void SimpleThreadOG(int which)
{
    printf("***_THIS_IS_FUNCTION: %d—>%s\n", (int) SimpleThreadOG, "
SimpleThreadOG!");
    int start = 0;
    int num;
    int limit = 5;

    for (num = start; num < limit; num++)    // HOMEWORK 1
    {
        printf("***_thread %d looped %d times\n", which, num);
    }
}
// HOMEWORK 1
void SimpleThreadYield(int which)
{
    printf("***_THIS_IS_FUNCTION: %d—>%s\n", (int) SimpleThreadYield, "
SimpleThreadYield!");
    int start = 0;
    int num;
    int limit = 5;

    for (num = start; num < limit; num++)    // HOMEWORK 1
    {
        printf("***_thread %d looped %d times\n", which, num);
        currentThread->Yield();
        if (num==0) {

```

```

        printf("***_THIS_IS_FUNCTION: %d-->%s\n", (int) SimpleThreadYield
            , "SimpleThreadYield!");
    }
}
printf("***_FUNCTION: %d-->%s\n", (int) SimpleThreadYield , "
    SimpleThreadYield_is_complete!");
}
// HOMEWORK 1
void SimpleThreadPrint(int which)
{
    printf("***_THIS_IS_FUNCTION: %d-->%s\n", (int) SimpleThreadPrint , "
        SimpleThreadPrint!");
    printf("***_The_user_initialized_which_func: %d\n", which_func);
}

//-----
// ThreadTest
//     Invoke a test routine.
//-----

void ThreadTest()
{
    DEBUG('t', "Entering_ThreadTest");

    Thread *t1 = new Thread("forked_thread_T1");
    Thread *t2 = new Thread("forked_thread_T2"); // HOMEWORK 1
    Thread *t3 = new Thread("forked_thread_T3"); // HOMEWORK 1

    t1->Fork(SimpleThreadOG, 1);
    t2->Fork(SimpleThreadYield, 42); // HOMEWORK 1
    t3->Fork(SimpleThreadPrint, 3); // HOMEWORK 1
    // SimpleThread(0);
    // printf("which_func: %d\n", which_func); // HOMEWORK 1
}

```

## 2 Please provide a summary of the two videos

### 2.1

Codes contains 8 folders of which **Threads** is the most important for our upcoming project. Threads is responsible for kernel/startup, synchronization of threads, switching and the scheduler. Machine simulates a MIPS architecture. **Userprog** is for running user programs. Use `./nachos` in the Threads folder to run the looping program. This program starts in **main.cc** which contains **Initialize()** (which is defined in **system.cc**). **#ifdef THREADS** declares

if in the folder named THREADS, then run.

`main.cc` takes *args* and then **DebugInit**, **Statistics**, **Interrupt**, **Scheduler**, **randomYield** (which randomly yields threads), **ThreadTest** (which is exposed through *extern void ThreadTest()* and forks thread invoking **SimpleThread()**) are all declared in `system.cc` and eventually called through the processes of **Initialize()**. Once *currentThread* points to **Finish()** (which is a function of the Thread object), **Sleep()** is called (another Thread function) which checks for *currentThread* (active threads) and if not, then *interrupt* will point to **Idle()** (an **Interrupt** function created in the Machine dir) which means the threads are waiting on I/O. If nothing then **Halt()** and **Cleanup()** are called (this stops the process and destroys the thread).

## 2.2

We begin in *thread.h* which contains kernel threads, status of threads, stack, registers, and runs functions such as **Fork()**, **Yield()**, and **Sleep()**. *UserProg* directory contains user state and address space.

When creating a thread:

**Thread()** and **Fork()** are invoked (of which the latter takes a thread running function) or we can **Yield()**.

When taking user input:

we can already run flags like **-d** (debug) at this point. *system.cc* contains the code which declares the flags. **include system.h** is in *threadtest.cc* which all starts in *main.cc*. Editing **ThreadTest()** and **SimpleThread()** in order to generate our own Thread Test. Jason then goes on to explain an example process of running threads which includes parallel and sequential threading. He also mentions thread 2 is sensitive to its counterpart thread 1, which we attempt to address with **Yield()** which places *currentThread* (Thread 1) in the back of the queue. He also mentions that you should not access global variables inside of a running thread for this would create a race condition where more than one thread is attempting to access the same address of memory.

## 4

### 4.1 Please read and summarize the main concepts from chapter/section 3.1, pages 106-110 in a few paragraphs

Older computer systems such as batch systems executed **jobs**. **Time-shared** systems ran **user programs; tasks**. **Single-user systems could run several programs**, although the operating system may need to support its own internal programmed activities (memory management). All of these are **processes; jobs**.

A process is the status of the current activity represented by the value of the **PC** and contents of the processor's registers. Memory layout is as follows:

- **text section** - executable (static)
- **data section** - global vars (static)
- **heap section** - dynamically allocated memory during program run time.
- **stack section** - temporary data storage when invoking functions (params, return addresses, local vars) (dynamic)

Each function call pushes params, local vars, and return addresses onto the stack (**Activation Record**). When control is returned from a function the Activation Record is popped from the stack (this is true for heap memory as well). The stack and heap grow *towards* one another while the OS ensures they do not *overlap* one another. Considered an *active entity* in which the PC holds the next instruction to execute and set of associated resources. **State** refers to the current activity of processes which can be:

- New (create)
- Running (execute)
- Waiting (signal or I/O)
- Ready (idle)

- Terminated (finished)

Previously you could only run one process per core at one time. The **Process Control Block (PCB)** also known as the **Task Control Block** simply serves as the repository for all the data needed to start, or restart, a process, along with some accounting data. PCB contains information about specific processes such as:

- State
- PC
- CPU Registers (save state info for interrupts)
- CPU-scheduling info (priority & queues)
- Memory-management information (value of base/limit of registers & page tables/segment tables)
- Accounting info (CPU & real time used, time limits, account numbers, process numbers)
- I/O status information (list of I/O devices allocated and open files)

A program is a *passive* entity; i.e. instructions stored on disk (**executable file**). Programs become processes when loaded into memory; i.e. double-click or command-line .out. Two processes can be associated to the same program, but considered two separate execution sequences; i.e. an execution environment (Java JVM).

## 4.2 Please read and summarize the main concepts from chapter/section 4.1, pages 160-162 in a few paragraphs

Threads control processes and PCB is expanded to include info on each thread in multi-thread systems. Most kernels are actually multithreaded. Share ID, PC, register set, stack, code & data section, and files & signals with threads of the same process.

Multicore processing, often called parallel processing, is more efficient because creating processes is costly. Benefits of multithreading include:

- **Responsiveness** (continue running if blocked & useful for UI/UX)
- **Resource** sharing (shared memory & message passing with several threads of activity in the same place)
- **Economy** (as stated before, costly to create process)
- **Scalability** (single-threaded can only run on one processor no matter how many cores may be available)

## 5 Which concepts did you have difficulty understanding? What questions do you have for Dr. Kumar and TAs? [This question is also graded and you must ask a question or clarification]

I'd like to understand what I am simulating/emulating more. I know—an operating system, but really... what is that? Perhaps understanding why for instance when I looped the threads 5 times, but the threads were actually only "looped" 4 times. What is a "looping" thread? What happens when the thread is "forked"? That one right there for sure I have no idea what is happening.