CE/CZ2005 and CPE205/CSC205: Operating Systems – Lab Experiment 1
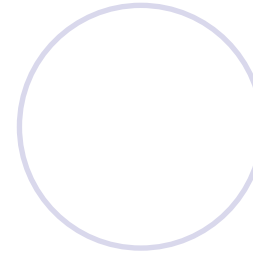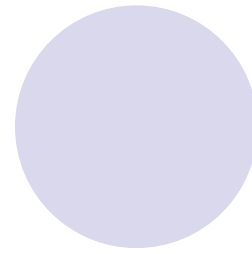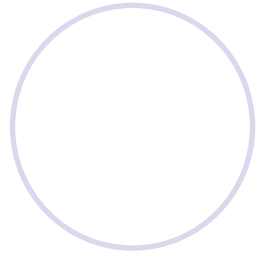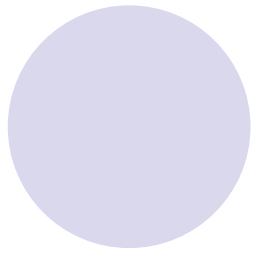
Contact: Bingsheng He (bshe@ntu.edu.sg)

Outline

- Overview of NachOS

- Thread Management and Scheduling in NachOS

- Discussion of Experiment 1
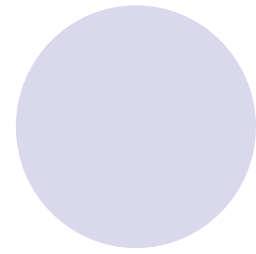
Overview of NachOS

NachOS – Background

- Real operating systems are very complex
  - Difficult to see how OS concepts are realized
  - Too time intensive to be part of an OS course
  - Therefore not so suitable for educational purposes

- NachOS is an educational operating system
  - Developed at UC Berkeley in C++
  - Freely available: http://www.cs.washington.edu/homes/tom/nachos
  - Idea: use the simplest possible implementation for each subsystem
  - Simple but complete

- Advantages of NachOS
  - Easier to read and understand the code
  - Easier to understand OS concepts
  - Easier to make changes and experiment
  - "Hands on" learning

NachOS – Architecture

- Approach in NachOS
  - Applications, kernel, and hardware simulator run in one UNIX process
  - NachOS uses a hardware simulation (virtual machine)

application	application

shell	user programs
MIPS simulation	
syscalls	virtual memory
RPC TCP	address spaces
	file system
thread management	portable OS kernel
machine-dependent OS layer	hardware simulation
I/O device simulation	

UNIX process
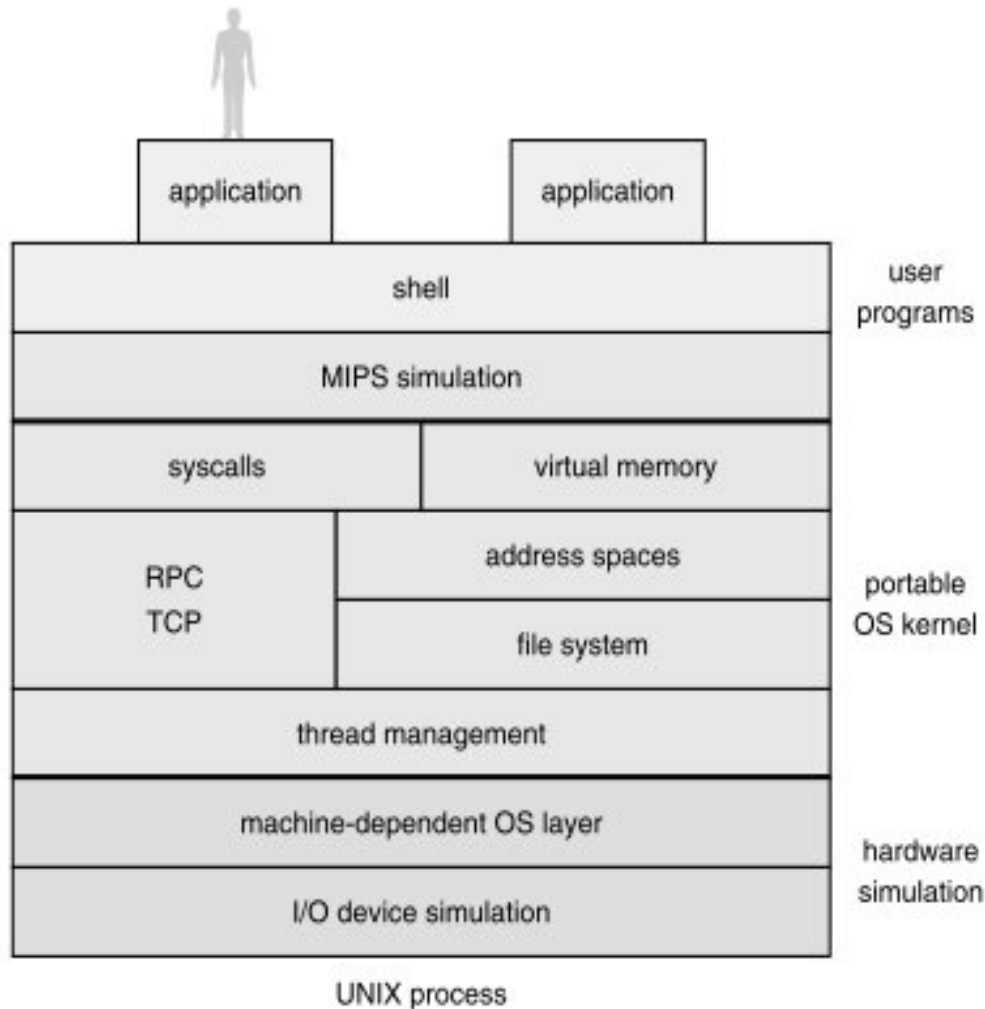
Advantages:
- System behavior is repeatable
- Easier to debug
- Shorter edit-compile-debug cycles

NachOS – Architecture

User Application

NachOS Process

application   application

shell   user
                               programs
MIPS simulation

syscalls   virtual memory

RPC   address spaces   portable
TCP   OS kernel
                 file system

thread management

machine-dependent OS layer   hardware
                                   simulation
I/O device simulation

UNIX process

Host Operating System (e.g., Linux)

Hardware
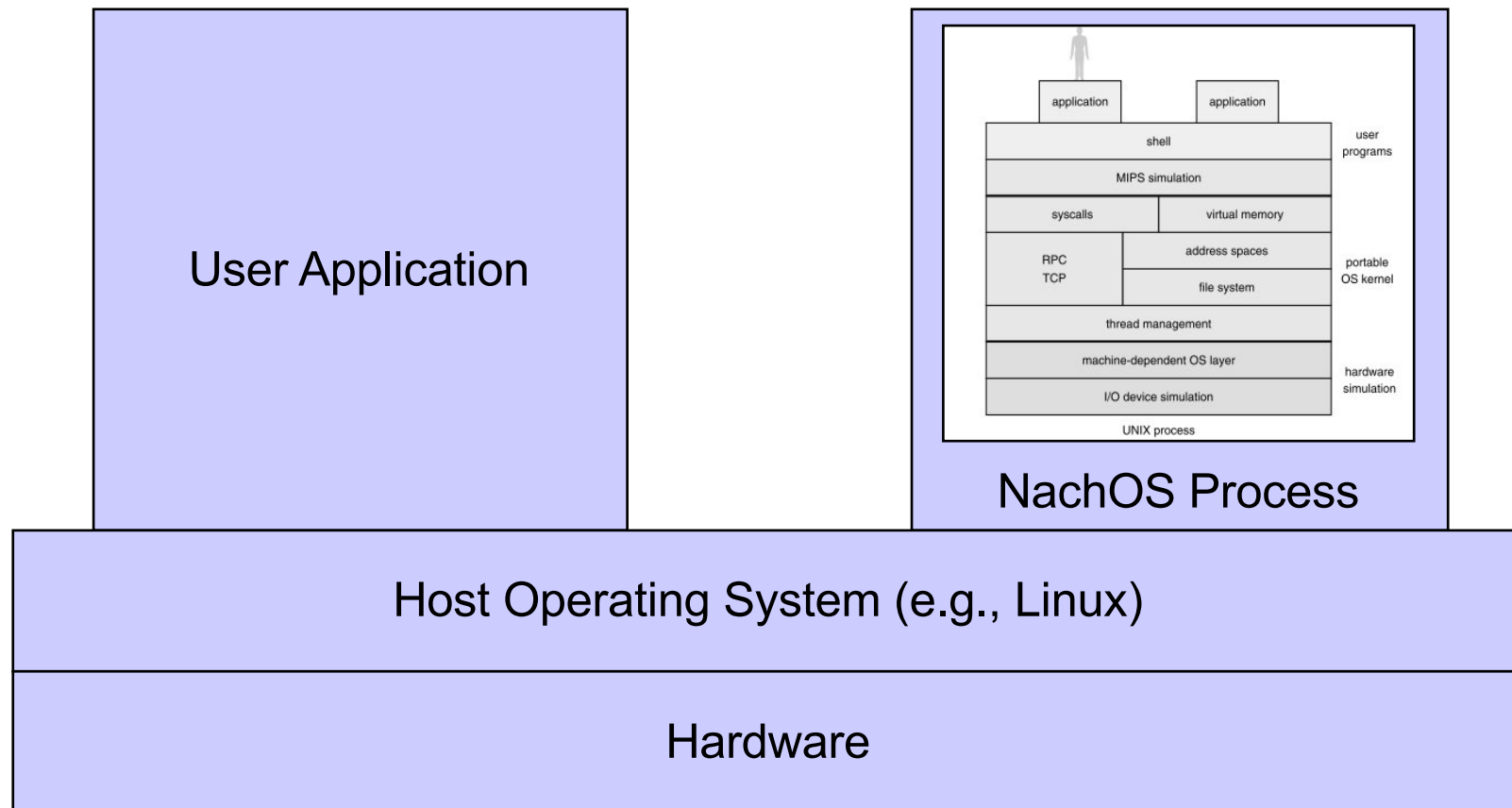
NachOS – How does it work?

Start NachOS binary
(./nachos)

Interrupt handling,
scheduler, file system,
networking, etc...

NachOS kernel

main() {
- Initialize()
- *Call Testproc()*
}

Run()

Machine

while(1) {
- Execute
}

Shutdown
NachOS

Instructions
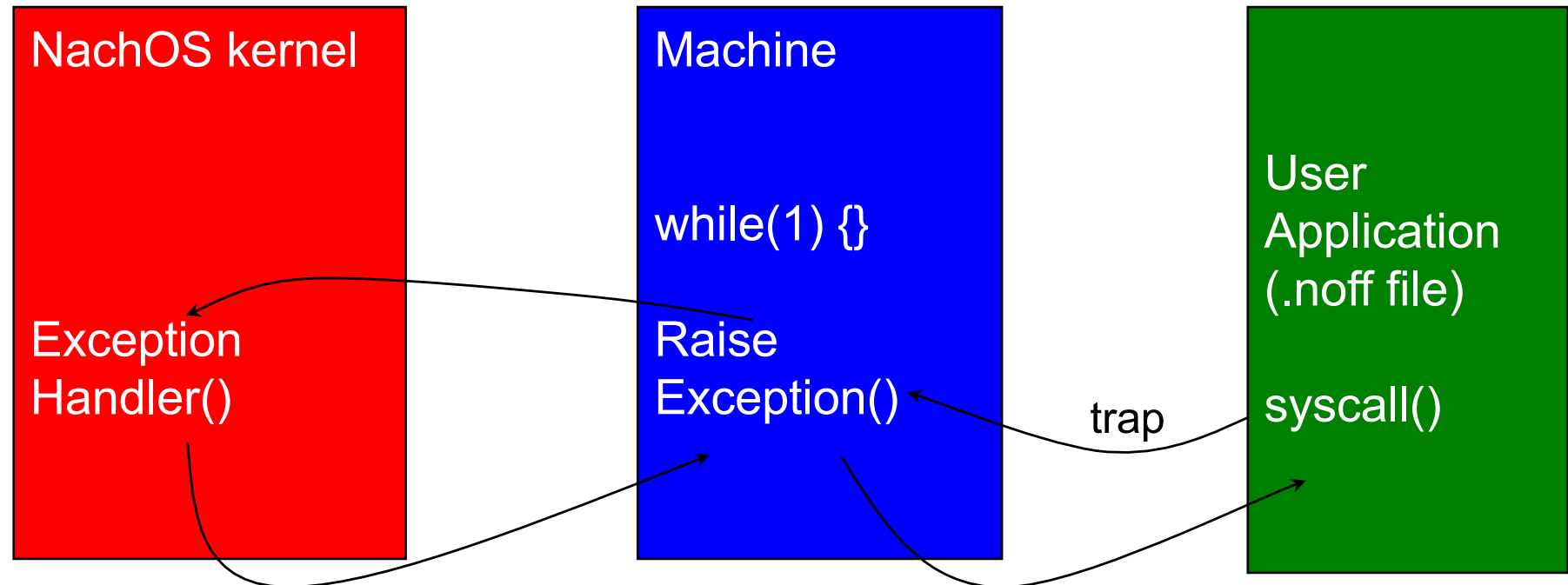
**User
Application
(.noff file)**

Halt()

print
Stats()

Return to host
OS shell

NachOS – How does it work?

NachOS kernel

Exception
Handler()

Machine

while(1) {}

Raise
Exception()

**User
Application
(.noff file)**

syscall()

trap
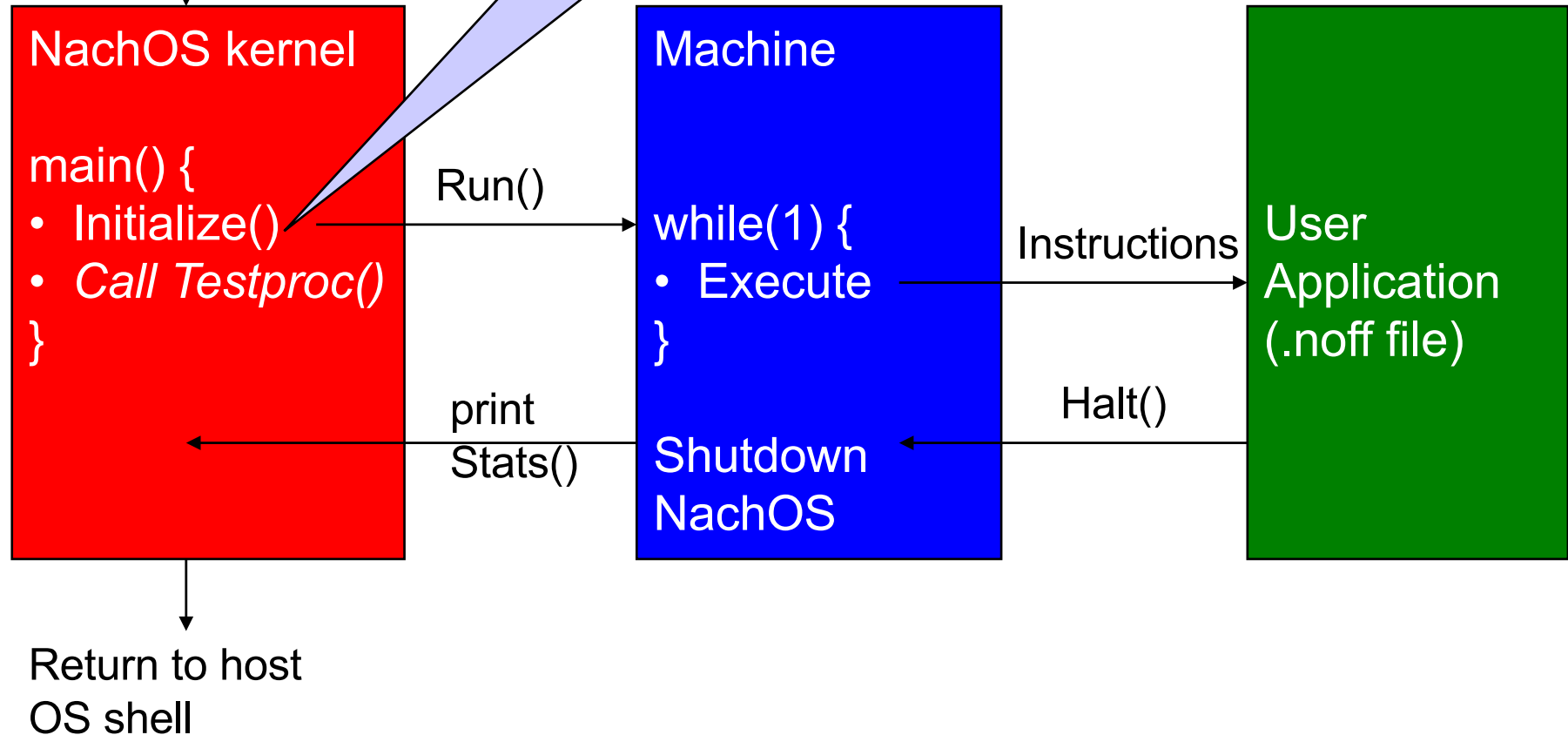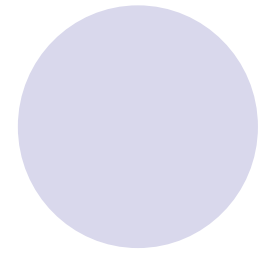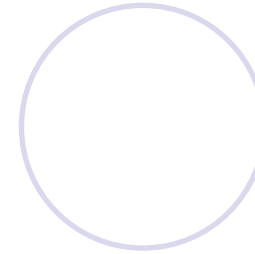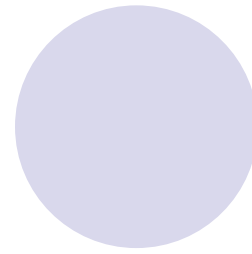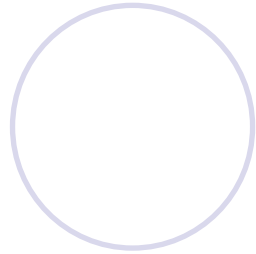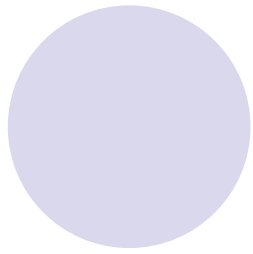
Thread Management and Scheduling in NachOS

NachOS – Thread Management

- Thread management is explicit
  - It is possible to trace statement for statement
  - See and understand what happens during a context switch

- Two things are important to understand how NachOS performs thread management
  - Scheduler
  - Threads

NachOS – Scheduler

- Maintains a list of threads that are ready to run: *ready list (ready queue)*

- The scheduler is invoked whenever the current thread gives up the CPU (non-preemptive)

- Simple scheduling policy is used
  - Assume equal priority for all threads
  - Select threads in FCFS fashion
  - Append at the end and remove from the front

- The scheduler code can be found in `threads/scheduler.cc`
  - ReadyToRun()
  - FindNextToRun()
  - Run()

NachOS – Scheduler

- ReadyToRun(thread)
  - ○ Makes a thread ready to run (set state to READY)
  - ○ Adds it to the ready list
  - ○ *Does not start the thread yet!*

- FindNextToRun() : thread
  - ○ Simply returns the thread at the front of the ready list

- Run(thread)
  - ○ Switches from one thread to another:
    1. Check whether the current thread overflowed its stack
    2. Change the state of the new thread to RUNNING
    3. Perform the actual context switch (by calling Switch())
    4. Terminate previous thread (if applicable)
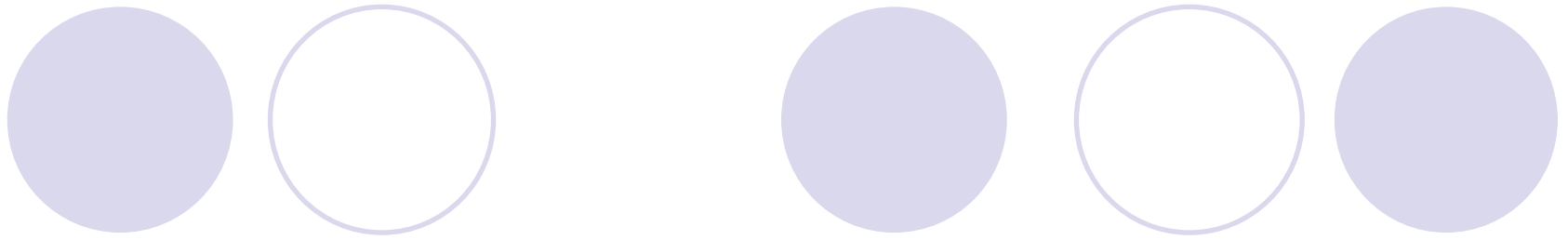
NachOS –Thread

- A thread can be in either of these states
  - READY: eligible to run
  - RUNNING: only one thread can be in this state
  - BLOCKED: waiting for some external event
  - JUST_CREATED: temporary state used during creation

- Implementation can be found in `threads/thread.cc`
  - Fork()
  - Yield()
  - Sleep()
  - Finish()

You will use them in Experiment 2

NachOS –Thread

- Fork(function, arg, flag)
  - Turns a thread into one that can be executed
  - Calls readyToRun()

- Yield()
  - Finds the next thread to run using findNextToRun()
  - If a another thread has been found
    - Call readyToRun() for the old thread
    - Run the new thread using Run()

- Sleep()
  - Set status to BLOCKED
  - Find another thread to run using findNextToRun()
  - If a another thread has been found
    - Run the new thread using Run()

- Finish()
  - Called at the end of execution
  - Marks a thread for termination

Discussion of Experiment 1

Experiment 1 – Overview

- Objective
  - Understand how context switches work
  - Trace the execution flow of a thread

- Tasks
  - Make a complete copy of NachOS
  - Build it & Run it
  - Analyze the output and the source code
  - Figure out what is happening and why

Directory Structure

bin	For generating NachOS format files, DO NOT CHANGE!
filesys	NachOS kernel related to file system, DO NOT CHANGE!
lab1	**Experiment 1**, no coding is required.
lab2	Experiment 2, process synchronization.
machine	MIPS H/W simulation, DO NOT CHANGE unless asked.
Makefile.common Makefile.dep	For compilation of NachOS, DO NOT CHANGE!
network	NachOS kernel related to network, DO NOT CHANGE!
port	Additional experiment for students registered with CPE205/CSC205
readme	Short description of OS labs and assessments
test	NachOS format files for testing virtual memory, DO NOT CHANGE!
threads	NachOS kernel related to thread management, DO NOT CHANGE!
userprog	NachOS kernel related to running user applications, DO NOT CHANGE!
vm	Experiment 3, coding virtual memory (TLB, page replacement)

Experiment 1 – User program

- User program for Experiment 1 can be found in `lab1/threadtest.cc`
  - SimpleThread()
  - ThreadTest() ← this is the test procedure called from within main()

Experiment 1 – ThreadTest()

```
void ThreadTest() {
    DEBUG('t', "Entering SimpleTest");
    Thread *t1 = new Thread("child1");
    t1->Fork(SimpleThread, 1, 0);
    Thread *t2 = new Thread("child2");
    t2->Fork(SimpleThread, 2, 0);
    SimpleThread(0);
}
```

Thread constructor does only minimal initialization

Turns a thread into one that can be scheduled and executed by the CPU

After starting two new threads execute the SimpleThread function

- Fork(function, arg, join)
  - "function": the procedure to be executed concurrently
    - Here: SimpleThread
  - "arg": a single parameter which is passed to the procedure
    - Here: the id of the thread (0,1,2)
  - "join": a flag which indicates whether the thread can be joined
    - Not important for the lab

Experiment 1 – SimpleThread()

```
void SimpleThread(_int which) {
    int num;
    for (num = 0; num < 3; num++) {
        printf("*** thread %d looped %d times\n", (int) which, num);
        currentThread->Yield();
    }
}
```

> Remember the only argument passed by fork? This is it. Here: 'which' tells you which thread

> Causes the current thread to give up the CPU

- Each thread is executing this function
- The function does nothing but prints a status line three times
  - Check your program output

Experiment 1 – Task description

- 8.6 Fill in the scheduling table whenever
  - the ready list is changed, or
  - the current thread is changed, or
  - a new message is printed in method SimpleTest()

ready list	current thread	printf message
Empty	main	
child1	main	
child1, child2	main	
child2, main	child1	thread 0 looped 0 times

- 8.7 List all context switches
  - For each switch: indicate from/to thread

- 8.10 Describe what and how a CPU scheduler is implemented in NachOS
  - Does the output reflect this behavior? Justify your answer

Experiment 1 – Summary

- **Objective:**
  - Understand how thread management and scheduling works
- **Assessment:**
  - Oral assessment in Lab 3
- **Where:**
  - Multimedia Lab 1 (N4-01a-02)
- **Documents:**
  - Can be found in edveNTUre

Acknowledgement

- The slides are revised from the previous versions created by Dr. Heiko Aydt.