NachOS – threads bootstrap

• In the code/thread directory, look at main.cc

    #ifdef THREADS //Defined for this assignment
        ThreadTest();
    #endif

• ThreadTest is in threadtest.cc

    void SimpleThread(int which) {
        int num;

        for (num = 0; num < 5; num++) {
            printf("*** thread %d looped %d times\n", which, num);
            currentThread->Yield();
        }
    }
    void ThreadTest() {
        DEBUG(’t’, "Entering SimpleTest");
        Thread *t = new Thread("forked thread");
        t->Fork(SimpleThread, 1);
        SimpleThread(0);
    }
NachOS (cont’d)

• In the directory code, type make
  – Help with makefiles tomorrow

• A nachos executable is created in the directory code/threads

• Run it (type ./nachos in the threads directory)

  *** thread 0 looped 0 times
  *** thread 1 looped 0 times
  *** thread 0 looped 1 times
  *** thread 1 looped 1 times
  *** thread 0 looped 2 times
  *** thread 1 looped 2 times
  *** thread 0 looped 3 times
  *** thread 1 looped 3 times
  *** thread 0 looped 4 times
  *** thread 1 looped 4 times

  No threads ready or runnable, and no pending interrupts.
  Assuming the program completed.
  Machine halting!
  Ticks: total 130, idle 0, system 130, 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
Thread Object

- In thread.h:

```c
enum ThreadStatus { JUST_CREATED, RUNNING, READY, BLOCKED };

class Thread {
private:
    int* stackTop; // the current stack pointer
    int machineState[MachineStateSize]; // all registers except for stackTop

public:
    Thread(char* debugName); // initialize a Thread
    void Fork(VoidFunctionPtr func, int arg); // Make thread run (*func)(arg)
    void Yield(); // Relinquish the CPU if any
    void Sleep(); // other thread is runnable
    void setStatus(ThreadStatus st) { status = st; }
    char* getName() { return (name); }
    void Print() { printf("%s, ", name); }

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

    int* stack; // Bottom of the stack
    ThreadStatus status; // ready, running or blocked
    char* name;
    void StackAllocate(VoidFunctionPtr func, int arg);
    // Allocate a stack for thread. Used internally by Fork()
```

SUNY-Binghamton – CS350 Spring ’01 NachOS Session
void Thread::Fork(VoidFunctionPtr func, int arg)
{
    DEBUG('t', "Forking thread \"%s\" with func = 0x%x, arg = %d\n",
            name, (int) func, arg);

    StackAllocate(func, arg);

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

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

    ASSERT(this == currentThread);

    DEBUG('t', "Yielding thread \"%s\"\n", getName());

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

• Scheduler maintains a list of ready threads; in scheduler.h

```cpp
class Scheduler {
public: ...  

    void ReadyToRun(Thread* thread); // Thread can be dispatched.
    Thread* FindNextToRun(); // Dequeue first thread on the ready
    // list, if any, and return thread.
    void Run(Thread* nextThread); // Cause nextThread to start running
    void Print(); // Print contents of ready list

private:
    List *readyList; // queue of threads that are ready to run,
    // but not running
};
```

//in scheduler.cc
void
Scheduler::ReadyToRun (Thread *thread)
{
    DEBUG('t', "Putting thread %s on ready list.\n", thread->getName());
    thread->setStatus(READY);
    readyList->Append((void *)thread);
}

Thread *
Scheduler::FindNextToRun ()
{
    return (Thread *)readyList->Remove();
}
```
void Scheduler::Run (Thread *nextThread) 
{
    Thread *oldThread = currentThread;
    currentThread = nextThread;
    currentThread->setStatus(RUNNING);

    // 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);

    if (threadToBeDestroyed != NULL) {
        delete threadToBeDestroyed;
        threadToBeDestroyed = NULL;
    }
}

• When you return from SWITCH, you will be back
in “nextThread”
Synchronization Primitives

- synch.h has the definition for Semaphores, Locks and Conditional Locks (three types of locks we will discuss in class)

- synch.cc has the implementation for Semaphores only – you will implement Locks and Conditional Locks

- You will use these mechanisms to implement the synchronization problems
Programming Tools in Unix

- Overview how programs are compiled and linked
- Makefiles
- Symbolic debugging – gdb
Compiling and Linking

• How to compile programs that are in many files?
  – How does a procedure in one file make a call to a procedure in another?

• A procedure prototype promises that a non-local procedure will eventually be supplied
  – That is why you include .h files

• Each file is compiled into an object code, that includes some procedure calls that are not resolved yet
Makefile by Example

MAKE = make
LPR = lpr

all:
    cd threads; $(MAKE) depend
cd threads; $(MAKE) nachos
...

clean:
    /bin/csh -c "rm -f */{core,nachos,DISK,*.o,swtch.s}
test/{*.coff} bin/{coff2flat,coff2noff,disassemble,out
Makefile by Example

# from Makefile.common
CFLAGS = -g -Wall -Wshadow -fwrtable-strings
        $(INCPATH) $(DEFINES) $(HOST) -DCHANGED
LDFLAGS =
# These definitions may change as the software is updated.
# Some of them are also system dependent
CPP=/lib/cpp
CC = g++
LD = g++
AS = as
# in threads/Makefile
DEFINES = -DTHREADS
INCPATH = -I../threads -I../machine
HFILES = $(THREAD_H)
CFILES = $(THREAD_C)
C_OFILES = $(THREAD_0)

include ../Makefile.common
include ../Makefile.dep

# from Makefile.common
$(C_OFILES): $(CFILES)
    $(CC) $(CFLAGS) -c $<

SUNY-Binghamton – CS350 Spring ’01 NachOS Session   11
Automatically Generated Dependencies

```
#Dependencies generated automatically

main.o: ../threads/main.cc ../threads/copyright.h ../threads/utility.h \\
       ../machine/sysdep.h /usr/include/stdio.h \\
       /usr/include/sys/feature_tests.h /usr/include/sys/va_list.h \\
       /usr/include/string.h ../threads/system.h ../threads/thread.h \\
       ../threads/scheduler.h ../threads/list.h ../machine/interrupt.h \\
       ../machine/stats.h ../machine/timer.h
```

- To add another file to be compiled by the Makefile in threads, add the name of the file to Makefile.common for the THREAD_C variable

- To be able to use gdb, add -g to the CFLAGS line in Makefile.common
Makefiles (contd)

- Do not forget the TAB at the beginning of every rule
- use make depend to generate dependencies automatically
- Make sure that you use gnu make (/opt/local/GNU/bin/make) and not the system make (/usr/ccs/bin/make). To find out which is in your default path type (which make); add /opt/local/GNU/bin/ to your path first if the ccs make comes up
- Make is a powerful multifeatured tool. You can get more help/documentation from the gnu archives (or one of the many mirrors) at:
  http://www.gnu.ai.mit.edu/ and for make specifically at
GNU Symbolic Debugger (gdb)

• Compile your code with -g to include the symbol table with the executable

• Starting gdb:

  `gdb program_name`

• Use gdb built-in help

  – `help` gives a list of classes of commands (e.g., breakpoints)
  – `help classname` gives the commands in that class
  – `help command` gives help on the specific command

• Running your program. After you start gdb, type

  `run <argument list>`
Setting breakpoints

• break [file:] line (e.g., break main.cc:37) stops the program at the specified line in the specified file

• break [file:] function (e.g., break main.cc:main) stops the program when it enters main() in main.cc

• You can break if a specific condition applies (e.g., a certain value for a variable)

• continue resumes execution

• After you break, you can examine variables, procedure call stack, step through instructions, etc...
Useful Commands

• step executes a single instruction then stops, if the instruction is a procedure call, it will stop at the first line in this new procedure

• next like step, but treats the full procedure call as one line

• print variable-name
  – If a pointer, can examine its contents (print *pointer)

• display variable-name will print the value after every command

• list will list the code around the point where you currently are
  – list number will list the code around the give line number
  – list file.cc:number also possible
Program Stack

- where will display your program stack (procedure call stack)

    (gdb) break Scheduler::ReadyToRun
    Breakpoint 1 at 0x12840: file ../threads/scheduler.cc, line 56.
    (gdb) run
    Starting program: /u0/users/1/cs350-1/test/code/threads/nachos

    Breakpoint 1, Scheduler::ReadyToRun (this=0x2dcc8, thread=0x2e118)
    at ../threads/scheduler.cc:56
    56 DEBUG('t', "Putting thread %s on ready list.

    thread->getName());
    (gdb) where
    #0 Scheduler::ReadyToRun (this=0x2dcc8, thread=0x2e118)
    at ../threads/scheduler.cc:56
    #1 0x1392c in Thread::Fork (this=0x2e118, func=0x13f4c <SimpleThread(int)>,
        arg=1) at ../threads/thread.cc:96
    #2 0x14058 in ThreadTest () at ../threads/threadtest.cc:48
    #3 0x12030 in main (argc=1, argv=0xffbefb1c) at ../threads/main.cc:88

- You can move up the stack or back down it, examining variables at every level

    gdb) up
    #1 0x1392c in Thread::Fork (this=0x2e118, func=0x13f4c <SimpleThread(int)>,
        arg=1) at ../threads/thread.cc:96
    96 scheduler->ReadyToRun(this);