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();
    }
  }
  ```
- ThreadTest0
  ```
  DEBUG("t", "Entering SimpleTest");
  Thread *t = new Thread("forked thread");
  t->Fork(SimpleThread, 1);
  SimpleThread0();
  ```

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
```

Thread Object

- In thread.h:
  ```
  enum ThreadStatus { NEW, READY, RUNNING, BLOCKED, FINISHED };
  class Thread {
  private:
    int* stackTop; // the current stack pointer
    int* machineStackBase; // all registers except for stackTop
    public:
    Thread(int msg); // initialize a thread
    void Fork(int FunctionPtr, int arg); // Make thread run (w/args)(arg)
    void Yield(); // relinquish the CPU if any
    void Sleep(); // Put the thread to sleep and
      // relinquish the processor
    void ResumeThread(Thread* st); // status = st;
    char* getName() { return name; }
    void Print() { printf("%s", name); }
  private:
    // name of the private data for this class in listed above
    int* stack; // bottom of the stack
    ThreadStatus status; // ready, running or blocked
    char* name; // ThreadAllocator(ThreadFunctionPtr func, int arg);
    // Allocate a stack for thread. Used internally by Fork() or
    // KERNO()
    void allocChar();
    void deleteThread();
  }
  ```

Thread Implementation

```
void Thread::Free(int FunctionPtr func, int arg)
{
    DEBUG("t", "Freeing thread \%s\%d with func = %p\%d arg = \%d\%d", func
    name, (int) func, arg);
    StackAllocator<TMP, arg>;
    stackStatus oldLevel = interrupt->SetLevel(MAINTASK);
    scheduler->ReadyToRun(now); // ready to run
    interrupt->SetLevel(oldLevel);
}
```
Scheduler

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

```c
class Scheduler {
public:

    void ReadyToRun(Thread* thread);  // Thread can be dispatched.
    Thread* FindRootThread();          // Dequeue first thread on the ready
    void Run(Thread* nextThread);      // Queue nextThread to start running
    void Print();
private:
    List* readyList;                  // Queue of threads that are ready to run.
                                        // but not running
};
```

// in scheduler.cc

```c
void
Scheduler::ReadyToRun(Thread* thread)
{
    DEBUG(1, "Putting thread %s on ready list.", thread->getName());
    thread->setStatus(READY);
    readyList->append(thread);
}
```

```c
Thread* Scheduler::FindRootThread()
{
    return (Thread *)readyList->remove();
}
```

- When you return from SWITCH, you will be back in "nextThread"

---

Scheduler::Run

```c
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"
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

```c
# from Makefile.common
CFLAGS = -g -Wall -Wshadow -fpermissive -std=gnu99
$(CINCFILE) $(DEFINES) $(HOST) -DHANGED
LDLFLAGS = 
# 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 = -DTRENDS
CINCFILE = -I../threads -I../machine
HFILES = $(THREAD_H)
CFILES = $(THREAD_C)
C_OFILES = $(THREAD_L)

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

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

Automatically Generated Dependencies

• 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 g++ with -g flag, add -g to the CFLAGS line in Makefile.common
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...

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 cc 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 http://www.gnu.ai.mit.edu/manual/make/index.html

Useful Commands

- list / list variable-name will list the code around the point where you currently are. If an argument is given, it will list the code around the value of the given variable
- display variable-name will print the value of the given variable in the current context
- next will run the next line, unless it is a single-step (when it will run the next line in the new procedure)
- 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
- run command list will run the commands in the specified list
- help gives a list of all commands (e.g., help breakpoints)
- run command list will run the commands in the specified list
- print variables points to the variables in that class
- heap gives help on heap command
- symbolic debug gives help on all symbolic debug commands
- compile your code with g++ to include the symbolic table with the executable
Program Stack

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

(gdb) break Scheduler::ReadyToRun
Breakpoint 1 at 0x1390: file ../threads/scheduler.c, line 56.
(gdb) run
Starting program: /u/users/1/ao30010/wet2/code/threads/nachos

Breakpoint 1, Scheduler::ReadyToRun [this=0x6360d0, thread=0xdeadbeef] at ../threads/scheduler.c:56
56 | BEEP() // "Putting thread 5e on ready list.\n
(gdb) where
#0 Scheduler::ReadyToRun [this=0x6360d0, thread=0xdeadbeef] at ../threads/scheduler.c:56
#1 0x1390c is Thread::Fork (child=0xdeadbeef, func=0x1390c4 <SimpleThread(int)>, args=1) at ../threads/thread.c:40
#2 0x14008 is Thread::Run () at ../threads/thread.c:40
#3 0x12000 is main (argc=1, argv[0]="fork.c") at ../threads/main.c:38

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

(gdb) up
#1 0x1390c is Thread::Fork (child=0xdeadbeef, func=0x1390c4 <SimpleThread(int)>, args=1) at ../threads/thread.c:40
#0 scheduler=0x6360d0, thread=0xdeadbeef

SUNY-Buffalo - CS150 Spring '09 Nachos Session 17