Lab1 Nachos系统的安装与调试

TASKS:

---

1. **在你所生成的 Nachos 系统中，下述函数的地址是多少？并说明找到这些函数地址的过程及方法。**

   i. `InterruptEnable()`

   ii. `SimpleThread()`

   iii. `ThreadFinish()`

   iv. `ThreadRoot()`

- 为各函数设置断点，例如：`b InterruptEnable`，观察到它们的地址分别是：`0x3027`、`0x3275`、`0x2ffc`、`0x4e7c`。

---

2. **下述线程对象的地址是多少？并说明找到这些对象地址的过程及方法。**

   i. `the main thread of the Nachos`

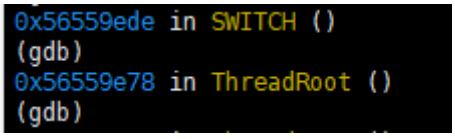
   ii. `the forked thread created by the main thread`

- 为 `Initialize` 函数设置断点，执行 `run` 运行程序到该函数入口；
- 依次执行 `step` 步进程序，当函数调用 `currentThread = new Thread("main")` 后，打印其地址：`print currentThread`，得到 `main thread` 的地址为 `0x56563ca0`；
- 为 `ThreadTest` 函数设置断点，继续执行程序：`continue`，进入该函数并执行 `Thread *t = new Thread("forked thread")` 后，执行语句 `print t`，得到 `forked thread` 的地址为 `0x5655df40`。

---

3. **当主线程第一次运行 SWITCH()函数，执行到函数 SWITCH()的最后一条指令 ret 时，CPU 返回的地址是多少？ 该地址对应程序的什么位置?**

- `Scheduler::Run()` 函数调用了 `SWITCH()` 函数，实现线程的上下文切换。执行 `b SWITCH` 设置断点，`run` 执行到该处后暂停，利用 `ni` 单步执行，执行完 `SWITH` 进入到 `ThreadRoot` 后，观察 `SWITCH` 的最后一行之后的第一行地址 `0x56559e78` 就是 `ret` 的返回地址。



- 在 `MIPS` 架构下，寄存器 `$ra` 的值由语句 `lw ra, PC(a1)` 装入进来，其中，`a1` 表示指向新线程的指针。
- `Fork` 方法分配一块固定大小的内存作为线程的堆栈，在栈顶放入 `ThreadRoot` 的地址。当新线程被调上 CPU 时，要用 `SWITCH` 函数切换线程图像，`SWITCH` 函数返回时，会从栈顶取出返回地址，于是将 `ThreadRoot` 放在栈顶，在 `SWITCH` 结束后就会立即执行 `ThreadRoot`函数。`ThreadRoot` 是所有线程的入口，它会调用 `Fork` 的两个参数，运行用户指定的函数；`Yield` 方法用于本线程放弃处理机。`Sleep` 方法可以使当前线程转入阻塞态，并放弃 CPU，直到被另一个线程唤醒，把它放回就绪线程队列。在没有就绪线程时，就把时钟前进到一个中断发生的时刻，让

中断发生并处理此中断，这是因为在没有线程占用 CPU 时，只有中断处理程序可能唤醒一个线程，并把它放入就绪线程队列。

- 线程要等到本线程被唤醒后，并且又被线程调度模块调上 CPU 时，才会从 `Sleep` 函数返回，新取出的就绪线程有可能就是这个要睡眠的线程。例如，如果系统中只有一个 A 线程，A 线程在读磁盘的时候会进入睡眠，等待磁盘操作完成。因为这时只有一个线程，所以 A 线程不会被调下 CPU，只是在循环语句中等待中断。当磁盘操作完成时，磁盘会发出一个磁盘读操作中断，此中断将唤醒 A 线程，把它放入就绪队列。这样，当 A 线程跳出循环时，取出的就绪线程就是自己。这就要求线程的正文切换程序可以将一个线程切换到自己，`Nachos` 的线程正文切换程序 `SWITCH` 可以做到这一点，于是 A 线程实际上并没有被调下CPU，而是继续运行下去了。

---

4. **当调用 `Fork()` 新建的线程首次运行 `SWITCH()` 函数时，当执行到函数 `SWITCH()` 的最后一条指令 `ret` 时，CPU 返回的地址是多少？ 该地址对应程序的什么 位置?**

- 设置断点 `b SWITCH`，第一次进入 `SWITCH` 后，执行 `c` 进入到下一次调用 `SWITCH` 的位置。如下图，函数 `Scheduler::Run` 的地址 `0x56556a26` 就是 `ret` 返回的地址。
- ```
  0x56559ede in SWITCH ()
  (gdb)
  0x56556a26 in Scheduler::Run (this=0x56563c80, nextThread=0x56563d00) at scheduler.cc:116
  116         SWITCH(oldThread, nextThread);
  ```
- `Scheduler::Run()` 函数调用了 `SWITCH()` 函数，实现线程的上下文切换。
- Nachos 的主线程 `main` 中的语句 `t->Fork(SimpleThread, 1)` 调用 `Thread::Fork()` 创建了一个子线程（命名为 `forked thread`），将子线程设为就绪状态并 进入就绪队列的尾部，子线程被调度时所执行的代码是 `SimpleThread(1)`；主线程 `main` 被调度时所执行的执行的代码为 `SimpleThread(0)`；
- `Thread::Yield ()` 中将子线程从就绪队里中取出：`nextThread = scheduler->FindNextToRun()`，将主线程的状态从执行转到就绪并放入就绪队列尾：`scheduler->ReadyToRun(this)`，将子线程设为执行状态：`currentThread = nextThread`，`currentThread->setStatus(RUNNING)`，然后第调用 `SWITCH()` 将主线的上下文切换到 子线程的上下文，子线程开始执行：`scheduler->Run(nextThread)`。这里的 `SWITCH()` 是第一次被调用。 这次 `SWITCH()` 的返回到 `ThreadRoot()` 的第一条指令处开始执行，由于子线程是 从头开始执行，因此 `ThreadRoot()` 是所有利用 `Thread::Fork()` 创建的线程的入口。 子线程开始执行后，后续与主函数发生的上下文切换都是从上次被中断的地方开 始执行，即 `Scheduler::Run()` 中语句 `SWITCH(oldThread, nextThread)` 之后。

Extended tasks

Nachos 广泛采用线程的概念，是多线程操作系统。线程是 Nachos 处理机调度的单位，在Nachos 中线程分成两类，一类是系统线程。所谓系统线程是只运行核心代码的线程，它运行在核心态下，并且占用宿主机的资源，系统线程共享Nachos操作系统本身的正文段和数据段；一个系统线程完成一件独立的任务，比如在 Nachos 网络部分，有一个独立的线程一直监测有无发给自己的数据报。

Nachos 的另一类线程同 Nachos 中的用户进程有关。Nachos 中用户进程由两部分组成，核心代码部分和用户程序部分。用户进程的进程控制块是线程控制块基础上的扩充。每当系统接收到生成用户进程的请求时，首先生成一个系统线程，进程控制块中有保存线程运行现场的空间，保证线程切换时现场不会丢失。该线程的作用是给用户程序分配虚拟机内存空间，并把用户程序的代码段和数据段装入用户地址空间，然后调用解释器解释执行用户程序；由于 Nachos 模拟的是一个单机环境，多个用户进程会竞争使用 Nachos唯一的处理机资源，所以在 Nachos 用户进程的进程控制块中增加有虚拟机运行现场空间以及进程的地址空间指针等内容，保证用户进程在虚拟机上的正常运行。
  ```

当线程运行终止时，由于当前线程仍然运行在自己的栈空间上，所以不能直接释放空间，只有借助其他的线程释放自己。

系统中，有两个与机器相关的函数，正文切换过程依赖于具体的机器，这是因为系统线程切换是借助于宿主机的正文切换，正文切换过程中的寄存器保护，建立初始调用框架等操作对不同的处理机结构是不一样的。其中一个函数是**ThreadRoot**，它是所有线程运行的入口；另一个函数是**SWITCH**，它负责线程之间的切换。**Scheduler** 类用于实现线程的调度。它维护一个就绪线程队列，当一个线程可以占用处理机时，就可以调用 ReadyToRun 方法把这个线程放入就绪线程队列，并把线程状态改成就绪态。FindNextToRun 方法根据调度策略，取出下一个应运行的线程，并把这个线程从就绪线程队列中删除。如果就绪线程队列为空，则此函数返回空(NULL)。现有的调度策略是先进先出策略(FIFO)，Thread 类的对象既用作线程的控制块，相当于进程管理中的 PCB，作用是保存线程状态、进行一些统计，又是用户调用线程系统的界面。

---

main.cc

初始化线程 `Initialize(argc, argv)` --> 线程执行函数 `ThreadTest()` --> 线程终止函数 `currentThread->Finish()`。

---

system.cc

`Initialize(int argc, char **argv)`:

- 初始化系统状态类 `stats`、中断处理类 `interrupt`、线程调度类 `scheduler`。
- 初始化主线程 `main`，并将其状态设置为 `RUNNING`

`Fork(VoidFunctionPtr func, _int arg)`:

- 调用函数 `StackAllocate` 为新线程分配栈空间，同时为该线程保存CPU寄存器值，便于调用 `SWITCH` 时能确保该线程得到运行。
- 将线程放入就绪队列中。

---

switch.s

线程启动时调用 `ThreadRoot` 函数，首先根据 `StartupPC` 调用线程启动时要调用的函数，然后将执行线程的参数列表 `InitialArg` 传入寄存器 `$a0` 中，再调用 `InitialPC` 定义的线程的程序计数器来执行程序，最后根据 `WhenDonePC` 来执行线程结束后要调用的例程。

在 `SWITCH` 函数中，程序先保存线程的栈指针 `$sp`，然后保存线程相关的寄存器 `$s0-$s7、$fp、$ra`，主要是 `$s0-$s3`，分别对应 `InitialPC、InitialArg、WhenDonePC、StartupPC`。

---

thread.h

定义了管理 `Thread` 的数据结构，即 Nachos 中线程的上下文环境，主要包括当前线程栈顶指针、CPU 寄存器的状态、栈底（栈是向下生长的，栈底是高地址，栈顶是低地址）、线程状态（`JUST_CREATED, RUNNING, READY or BLOCKED`）、线程名字。

必须先定义栈指针再定义寄存器状态数组，因为为这两个变量和线程的切换有密切的关系，在 SWITCH 函数中：

```
void SWITCH(Thread *oldThread, Thread *newThread);
```

`$a0` -- 指向 `oldThread` 寄存器状态的指针

`$a1` -- 指向 `newThread` 寄存器状态的指针

---

thread.cc

`thread.cc` 中主要是管理 `thread` 的一些事务。主要包括了四个主要的方法。`Fork()、Finish()、Yield()、Sleep()`。在 `thread.h` 中对它们进行了声明，在 `thread.cc` 中则负责具体的实现。注意到，这里实现的方法大多是都是原子操作，在方法的一开始保存中断层次关闭中断，并在最后恢复原状态。

`Fork(func, arg)` -- 允许调用者和被调用者并发执行 `func` 函数：

- `StackAllocate(func, arg)`：为新线程分配栈空间，由于 `MIPS` 架构的处理器中栈是由上往下增长的，栈顶地址 = 栈底地址 + 栈空间 - 4，其中 4 是安全边界，防止栈顶越界。`machineState` 保存该线程的各状态寄存器的值。之后，将CPU寄存器状态值保存到 `machineState` 中，以便线程未运行的时候也能知道CPU的各寄存器值。
- 初始化栈空间，以便于 SWITCH 函数调用；
- 将新线程放入就绪队列中。

`Yield ()` -- 转让CPU控制权给就绪队列中的线程执行：

- 获得就绪队列中位于队首的线程，将该线程从就绪队列中删除。
- 将当前线程的状态设置为 `READY`，加入到线程就绪队列中。
- 调用 `scheduler->Run(nextThread)` 运行第1步中获得的线程。

---

threadtest.cc

`ThreadTest()`：

- 创建一个新的线程 `t`，对 `t` 调用 `Fork` 函数，生成一个子线程，该子线程负责执行函数 `SimpleThread`，参数为 `1`。
- 主线程调用函数 `SimpleThread`，传入参数为 `0`，由主线程打印的语句，前半句均为 `thread 0`。

`SimpleThread(_int which)`：

- 主线程进入该函数后，打印 `*** thread 0 looped 0 times`。
- 主线程调用方法 `currentThread->Yield()`，将当前CPU转给子线程，子线程执行 `func == SimpleThread`，打印 `*** thread 1 looped 0 times`。
- 循环5次，每次 `num += 1`。

---

scheduler.cc

`Run (Thread *nextThread)`：

- 获取当前正在运行的线程 `currentThread`，检查 `nextThread` 是否有栈溢出，通过后将 `currentThread` 设置为 `nextThread`，同时更新线程状态。
- 调用 `SWITCH(oldThread, nextThread)` 函数，完成线程的切换。