华东师范大学软件工程学院实验报告

课程名称:	操作系统实践	指导教师:	张民
姓　　名:	王海生	学　　号:	10235101559
实验编号:	实验四	实验名称:	修改忙等

代码仓库：https://github.com/Hanson-Wang-chn/ECNU-Operating-System-WHS.git

1 实验目的

1. 展示忙等：在函数 `thread_yield()` 中添加 `print` 语句打印必要信息，运行指令 `pintos -v -- -q run alarm-multiple`，查看展示运行结果，并文字说明其是忙等；
2. 实现休眠：实现 `thread` 的 `sleep` 功能，在 wake up 时添加 `print` 语句打印必要信息，再次运行指令 `pintos -v -- -q run alarm-multiple`，查看展示新的运行结果，以及和之前结果的差别，并文字说明其原因；
3. 实现苏醒后抢占：`sleep` 的进程醒来时，如果当前 running 的进程优先级比它低，醒来的进程抢占执行。可以回忆上一次的实践课内容关于 priority 的内容。

2 实验内容与设计思想

在 `src/threads/`目录下运行 `make check`，可以看到有下面几个测试点：

```
1     pass tests/threads/alarm-single
2     pass tests/threads/helloworld
3     pass tests/threads/alarm-multiple
4     pass tests/threads/alarm-simultaneous
5     FAIL tests/threads/alarm-priority
6     pass tests/threads/alarm-zero
7     pass tests/threads/alarm-negative
```

1. 展示忙等：在函数 `thread_yield()` 中添加打印语句，用于输出当前线程名称及其被调度时的系统 tick 数。通过执行 `pintos -v -- -q run alarm-multiple` 命令，并观察输出结果，可以发现系统频繁地进行上下文切换，即使某些线程没有实际工作需要处理。这表明原始实现采用了效率较低的忙等待机制来处理休眠请求。
2. 实现休眠：为了优化休眠机制，我们需要重新实现 `timer_sleep()` 函数。新的实现方式是将请求休眠的线程放入一个专门的等待队列 `wait_list` 中，并为每个线程设置了一个预计唤醒时间点 `wake_tick`。每

当系统经过一个 tick 时，仅需检查 `wait_list` 中的第一个元素是否达到了预定的唤醒时刻。如果条件满足，则将该线程从 `wait_list` 移除并加入到就绪队列 `ready_list` 中准备执行。此外，在每次线程被唤醒时也加入了相应的打印信息，以便于后续分析线程的实际行为模式变化。通过再次运行 `pintos -v --q run alarm-multiple` 命令，可以看到新机制下的输出结果更加有序，减少了不必要的 CPU 资源消耗。

3. 实现苏醒后抢占：为了让系统能够根据优先级进行更灵活的调度，我们需要对 `thread_unwait()` 函数进行调整。当高优先级的线程从休眠状态恢复且发现当前正在执行的线程具有较低优先级时，新唤醒的线程会立即获得处理器控制权。为此，我们在检测到有线程被唤醒后设置了标志变量 `yield_flag`，并通过 `intr_yield_on_return()` 确保在中断返回时触发一次强制性的上下文切换尝试。这样，不仅增强了 Pintos 操作系统对于睡眠/唤醒操作的支持，同时也强化了其按照优先级进行调度的能力，使得整个系统更加高效和响应迅速。

3 使用环境

3.1 主机系统配置

本次实验的主机系统环境如下表所示：

项目名称	详细信息
操作系统	macOS Sequoia 15.0
系统类型	64 位操作系统，基于 ARM 的处理器
CPU	Apple M1 Pro
GPU	Apple M1 Pro
内存	16GB 统一内存
磁盘	512GB SSD

3.2 Docker 配置

在官网下载并安装后，Docker 容器正常运行，如下图所示：

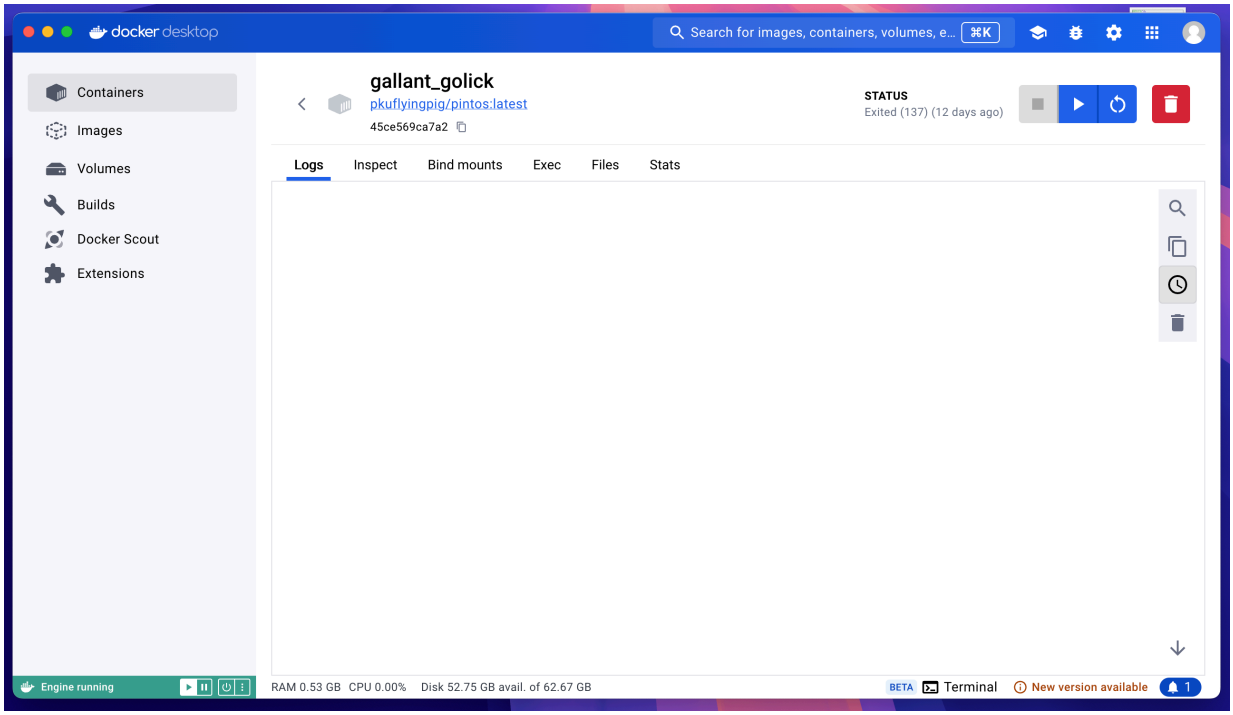

图 1: Docker 容器

接着使用下面的命令实现磁盘挂载，方便文件管理：

启动 Docker 容器并挂载文件

```
1    docker run -it --rm --name pintos --mount type=bind,\
2    source=/Users/wanghaisheng/Desktop/Coding/ECNU-Operating-System-WHS/pintos,\
3    target=/home/PKUOS/pintos pkuflyingpig/pintos bash
```

完成后如下图所示：

图 2: 完成环境配置

4 实验过程

4.1 展示忙等

在 thread_yield() 函数中加入打印语句，以便观察线程何时被调度。修改后的 thread_yield() 函数如下：

thread_yield

```
1     void
2     thread_yield (void)
3     {
4             struct thread *cur = thread_current ();
5             enum intr_level old_level;
6
7             ASSERT (!intr_context ());
8             old_level = intr_disable ();
9             if (cur != idle_thread)
10            list_insert_ordered(&ready_list, &cur->elem, thread_less_priority,
                  NULL);
```

```
11              printf("Yield: thread(%s) at tick %d.\n", cur->name, timer_ticks());
12              // ASSERT(false);
13              cur->status = THREAD_READY;
14              schedule ();
15              intr_set_level (old_level);
16      }
```

运行 `pintos -v -- -q run alarm-multiple` 的结果如下所示。可以看到，每个线程在每个时间片结束时都进行了调度，这表明系统正在使用忙等待的方式进行调度，所以效率很低。

```
1       (alarm-multiple) begin
2       (alarm-multiple) Creating 5 threads to sleep 7 times each.
3       (alarm-multiple) Thread 0 sleeps 10 ticks each time,
4       (alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
5       (alarm-multiple) If successful, product of iteration count and
6       (alarm-multiple) sleep duration will appear in nondescending order.
7       Yield: thread(main) at tick 26.
8       Yield: thread(thread 0) at tick 27.
9       Yield: thread(thread 1) at tick 27.
10      Yield: thread(thread 2) at tick 27.
11      Yield: thread(thread 3) at tick 27.
12      Yield: thread(thread 4) at tick 28.
13      Yield: thread(main) at tick 28.
14      Yield: thread(thread 0) at tick 28.
15      Yield: thread(thread 1) at tick 28.
16      Yield: thread(thread 2) at tick 28.
17      Yield: thread(thread 3) at tick 29.
18      Yield: thread(thread 4) at tick 29.
19      Yield: thread(main) at tick 29.
20      Yield: thread(thread 0) at tick 29.
21      Yield: thread(thread 1) at tick 30.
22      Yield: thread(thread 2) at tick 30.
23      Yield: thread(thread 3) at tick 30.
24      Yield: thread(thread 4) at tick 30.
25      Yield: thread(main) at tick 31.
26      Yield: thread(thread 0) at tick 31.
27      Yield: thread(thread 1) at tick 31.
28      Yield: thread(thread 2) at tick 31.
29      Yield: thread(thread 3) at tick 32.
30      Yield: thread(thread 4) at tick 32.
31      Yield: thread(main) at tick 32.
32      Yield: thread(thread 0) at tick 32.
```

```
33      Yield: thread(thread 1) at tick 33.
34      Yield: thread(thread 2) at tick 33.
35      Yield: thread(thread 3) at tick 33.
36      Yield: thread(thread 4) at tick 33.
37      Yield: thread(main) at tick 34.
```

4.2 实现休眠

通过对 alarm_wait.c 测试脚本的分析，我们了解到首先需要创建 thread_cnt 个进程，每个进程将执行 sleeper 函数。接着，主测试进程会进入休眠状态足够长的时间以确保所有子进程完成它们的任务。在深入研究 sleeper 函数时发现，它调用了 timer_sleep 函数使每个进程按照 iteration 指定的次数进行休眠，并记录下各个线程被唤醒的顺序。同时，从 alarm-zero.c 和 alarm-negative.c 中了解到还需处理定时器滴答数小于或等于零的情况。

随后，在验证阶段，根据各线程被唤醒的顺序来进行检查，该顺序必须与预期一致；此外，还需确认每个进程确实被唤醒了 iteration 次。只有当这些条件都满足时，才认为整个过程是正确的。

当前面临的主要挑战在于实现 timer_sleep 函数，特别是需要对其现有的忙等待机制进行改进。

timer_sleep

```
1       void timer_sleep(int64_t ticks) {
2               int64_t start = timer_ticks();
3               ASSERT(intr_get_level() == INTR_ON);
4               while (timer_elapsed(start) < ticks)
5               thread_yield();
6       }
```

一种实现思路是在每个时钟滴答（tick）检查所有处于等待状态的线程。为此，每个线程需要维护一个名为 wait_ticks 的变量，用于表示该线程还需要等待多少个时钟滴答。每当发生一次时钟滴答时，遍历整个等待队列，并将每个线程的 wait_ticks 变量减一。如果某个线程的 wait_ticks 减至 0，则将其从等待队列中移除并添加到就绪队列 ready_list 中。为了优化性能，可以单独设立一个 wait_list 专门存放等待中的线程，从而减少每次时钟滴答时需遍历的线程数量。这种方法的插入操作复杂度为 O(1)，而处理时钟滴答的操作复杂度为 O(n)。

另一种方法同样使用 wait_list,但每个进程记录的是其预期唤醒的具体时间点。在将新进程加入 wait_list 时，按照唤醒时间进行排序。这样，在每次时钟滴答时只需要检查 wait_list 链表头部的进程是否已达到唤醒时间即可。如果条件满足，则将该进程转移到 ready_list。这种方法的插入操作复杂度为 O(n)，而处理时钟滴答的操作复杂度降低到了 O(1)。

考虑到实际应用中，向 wait_list 插入元素的频率远低于时钟滴答事件发生的频率，因此选择第二种方法更为高效。

timer_sleep 函数的主要职责是将当前线程放入等待队列,并设置相应的 wait_ticks 值。可以参考 thread_unbloc 函数来了解如何将线程添加到 ready_list 的过程，同时借鉴 ready_list 初始化的方式来初始化 wait_list。醒来的时间应设定为当前时间加上所需的等待时间。

类似其他涉及进程状态转换的函数，在修改进程状态时必须确保操作的原子性。在 Pintos 操作系统环境中，这通常通过关闭中断来实现。因此，在 `timer_sleep` 或类似的 `thread_wait` 函数中，应当在关闭中断的状态下完成对进程状态的所有修改。

thread_wait

```
1     bool thread_less_wake_tick(const struct list_elem *a, const struct list_elem
          *b, void *aux) {
2             int64_t x = list_entry(a, struct thread, waitelem)->wake_tick;
3             int64_t y = list_entry(b, struct thread, waitelem)->wake_tick;
4             return x < y;
5     }
6
7     void thread_wait(int64_t wait_ticks) {
8             ASSERT(intr_get_level() == INTR_ON);
9             enum intr_level old_level = intr_disable();
10            int64_t cur_tick = timer_ticks();
11            struct thread *t = thread_current();
12            t->wake_tick = wait_ticks + cur_tick; // printf("wake_tick:%d\n", t
                  ->wake_tick);
13            list_insert_ordered(&wait_list, &t->waitelem, thread_less_wake_tick,
                   NULL);
14            thread_block();
15            intr_set_level(old_level);
16    }
17
18    void timer_sleep(int64_t ticks) {
19            if (ticks <= 0) return;
20            thread_wait(ticks);
21    }
```

接下来的任务是在每个时钟滴答（tick）中唤醒符合条件的线程。可以参考 `thread_foreach` 函数来遍历等待队列 `wait_list`。当检测到当前系统时间大于或等于某个进程设定的唤醒时间时，该进程应被唤醒，并通过调用 `thread_unblock` 函数将其加入到就绪队列 `ready_list` 中。同时，从 `wait_list` 中移除该进程对应的元素。

由于唤醒时间具有单调递增的特性，一旦遇到一个尚未达到唤醒时间的进程，即可停止继续检查剩余的进程，即在此处使用 `break` 语句终止循环。这样可以有效地减少不必要的遍历操作，提高效率。

thread_unwait

```
1     void thread_unwait(void) {
2             struct list_elem *e;
3             ASSERT(intr_get_level() == INTR_OFF);
4             int64_t cur_tick = timer_ticks();
5
```

```
6             for (e = list_begin(&wait_list); e != list_end(&wait_list); e =
                  list_next(e)) {
7                 struct thread *t = list_entry(e, struct thread, waitelem);
8                 if (t->wake_tick > cur_tick) break;
9                 // printf("wake:%s%d\n", t->name, t->wake_tick);
10                list_remove(e);
11                thread_unblock(t);
12            }
13    }
```

输出如下, 可以看到每个进程在该醒来的时候就醒来, 不必像忙等待一样时刻都在检查。

```
1     Executing 'alarm-multiple':
2     (alarm-multiple) begin
3     (alarm-multiple) Creating 5 threads to sleep 7 times each.
4     (alarm-multiple) Thread 0 sleeps 10 ticks each time,
5     (alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
6     (alarm-multiple) If successful, product of iteration count and
7     (alarm-multiple) sleep duration will appear in nondescending order.
8     wake: thread 0 135
9     wake: thread 1 145
10    wake: thread 0 145
11    wake: thread 2 155
12    wake: thread 0 155
13    wake: thread 3 165
14    wake: thread 1 165
15    wake: thread 0 165
16    wake: thread 4 175
17    wake: thread 0 175
18    wake: thread 2 185
19    wake: thread 1 185
20    wake: thread 0 185
21    wake: thread 0 195
22    wake: thread 3 205
23    wake: thread 1 205
24    wake: thread 2 215
25    wake: thread 4 225
26    wake: thread 1 225
27    wake: thread 3 245
28    wake: thread 2 245
29    wake: thread 1 245
30    wake: thread 1 265
31    wake: thread 4 275
32    wake: thread 2 275
33    wake: thread 3 285
```

```
wake: thread 2 305
wake: thread 4 325
wake: thread 3 325
wake: thread 2 335
wake: thread 3 365
wake: thread 4 375
wake: thread 3 405
wake: thread 4 425
wake: thread 4 475
wake: main 575
```

4.3 苏醒后抢占

当进程从阻塞状态恢复时，可以认为是 thread_unwait 通过调用 thread_unblock 实现了这一转变。为了确保优先级调整后的正确性，在所有涉及改变优先级的地方适当添加 thread_yield 调用。这样处理后，系统就能够顺利通过与优先级相关的测试点。

thread_create

```
tid_t
thread_create (const char *name, int priority,
thread_func *function, void *aux)
{
        //...
        /* Add to run queue. */
        thread_unblock (t);
        thread_yield();

        return tid;
}

void
thread_set_priority (int new_priority)
{
        thread_current ()->priority = new_priority;
        thread_yield();
}
```

然而，thread_unwait 函数是由中断处理程序调用的，在这种情况下当前运行的是 idle 进程，直接调用 thread_yield 来进行调度是不合适的。通过审查源代码发现，intr_return_yield 机制能够满足这一需求。因此，在 thread_unwait 函数中设置一个标志来指示是否有进程被唤醒，如果确实有进程被唤醒，则在中断返回时利用 intr_return_yield 来触发一次调度。

thread_unwait

```
bool thread_unwait()
{

        struct list_elem *e;
        ASSERT (intr_get_level () == INTR_OFF);
        int64_t cur_tick = timer_ticks();

        bool yield_flag = false;

        for (e = list_begin (&wait_list); e != list_end (&wait_list);
        e = list_next (e))
        {
                struct thread *t = list_entry (e, struct thread, waitelem);
                if (t->wake_tick > cur_tick) break;

                yield_flag = true;
                list_remove(e);
                thread_unblock(t);
        }

        return yield_flag;
}
```

基于之前的实验，我们添加了一个测试脚本，该脚本创建一个高优先级的子进程并使其进入休眠状态，而主进程则进入一个无限循环。当子进程被唤醒后，它应当能够根据其较高的优先级抢占主进程。随后，降低子进程的优先级，这时主进程的循环结束，并且由于优先级调整，尽管此时子进程的优先级更高，但控制权将先返回给主进程。随着子进程的完成，最终主进程也结束运行。以下是具体的测试脚本。

测试脚本

```
#include <stdio.h>
#include "tests/threads/tests.h"
#include "threads/init.h"
#include "threads/malloc.h"
#include "threads/synch.h"
#include "threads/thread.h"
#include "devices/timer.h"

static void high_priority_sleeper(void*);

void test_priority_alarm(void) {
        int high_priority = PRI_DEFAULT + 10;
        thread_create("high_priority_sleeper", high_priority,
            high_priority_sleeper, NULL);
```

```
14
15              msg("Main thread starts running.");
16              int64_t start_tick = timer_ticks();
17              while (timer_elapsed(start_tick) < 20);
18
19              msg("Main thread thread changed its priority to 21");
20              thread_set_priority(21);
21              msg("Main thread completed execution.");
22      }
23
24      static void high_priority_sleeper(void *aux UNUSED) {
25              msg("High-priority thread starts and goes to sleep.");
26              timer_sleep(10);
27              msg("High-priority thread woke up at and preempts the main thread.")
                    ;
28              msg("High-priority thread changed its priority to 26");
29              thread_set_priority(PRI_DEFAULT - 5);
30              msg("High-priority thread exit.");
31      }
32
33      void thread_tick(void) {
34              struct thread *t = thread_current();
35
36              /* Update statistics. */
37              if (t == idle_thread)
38                      idle_ticks++;
39              #ifdef USERPROG
40              else if (t->pagedir != NULL)
41                      user_ticks++;
42              #endif
43
44              else
45                      kernel_ticks++;
46
47              /* Enforce preemption. */
48              bool yield_flag = thread_unwait();
49              if (++thread_ticks >= TIME_SLICE || yield_flag)
50                      intr_yield_on_return();
51      }
```

预期输出如下：

```
1       # -*- perl -*-
2       use strict;
3       use warnings;
```

```
4     use tests::tests;
5     check_expected ([<<'EOF']);
6     (priority-alarm) begin
7     (priority-alarm) High-priority thread starts and goes to sleep.
8     (priority-alarm) Main thread starts running.
9     (priority-alarm) High-priority thread woke up at and preempts the main
          thread.
10    (priority-alarm) High-priority thread changed its priority to 26
11    (priority-alarm) Main thread thread changed its priority to 21
12    (priority-alarm) High-priority thread exit.
13    (priority-alarm) Main thread completed execution.
14    (priority-alarm) end
15    EOF
16    pass;
```

5 实验总结

至此，本实验相关的测试点全部通过，截图如下：

```
root@a33841aa4d9c:~/pintos/src/threads# make check
cd build && make check
make[1]: Entering directory '/home/PKUOS/pintos/src/threads/build'
pass tests/threads/alarm-single
pass tests/threads/helloworld
pass tests/threads/alarm-multiple
pass tests/threads/alarm-simultaneous
pass tests/threads/alarm-priority
pass tests/threads/alarm-zero
pass tests/threads/alarm-negative
pass tests/threads/priority-change
FAIL tests/threads/priority-donate-one
pass tests/threads/priority-alarm
FAIL tests/threads/priority-donate-multiple
FAIL tests/threads/priority-donate-multiple2
FAIL tests/threads/priority-donate-nest
FAIL tests/threads/priority-donate-sema
FAIL tests/threads/priority-donate-lower
pass tests/threads/priority-fifo
pass tests/threads/priority-preempt
FAIL tests/threads/priority-sema
FAIL tests/threads/priority-condvar
FAIL tests/threads/priority-donate-chain
FAIL tests/threads/mlfqs-load-1
FAIL tests/threads/mlfqs-load-60
FAIL tests/threads/mlfqs-load-avg
FAIL tests/threads/mlfqs-recent-1
pass tests/threads/mlfqs-fair-2
pass tests/threads/mlfqs-fair-20
FAIL tests/threads/mlfqs-nice-2
FAIL tests/threads/mlfqs-nice-10
FAIL tests/threads/mlfqs-block
16 of 29 tests failed.
../../tests/Make.tests:26: recipe for target 'check' failed
make[1]: *** [check] Error 1
make[1]: Leaving directory '/home/PKUOS/pintos/src/threads/build'
../Makefile.kernel:10: recipe for target 'check' failed
make: *** [check] Error 2
root@a33841aa4d9c:~/pintos/src/threads#
```

图 3: Docker 容器

其中，`priority-alarm` 为自定义测试点。

6 附录：修改后的代码

pintos/src/devices/timer.c

```
#include "devices/timer.h"
#include <debug.h>
#include <inttypes.h>
#include <round.h>
#include <stdio.h>
#include "devices/pit.h"
#include "threads/interrupt.h"
#include "threads/synch.h"
#include "threads/thread.h"

/** See [8254] for hardware details of the 8254 timer chip. */

#if TIMER_FREQ < 19
#error 8254 timer requires TIMER_FREQ >= 19
#endif
#if TIMER_FREQ > 1000
#error TIMER_FREQ <= 1000 recommended
#endif

/** Number of timer ticks since OS booted. */
static int64_t ticks;

/** Number of loops per timer tick.
Initialized by timer_calibrate(). */
static unsigned loops_per_tick;

static intr_handler_func timer_interrupt;
static bool too_many_loops (unsigned loops);
static void busy_wait (int64_t loops);
static void real_time_sleep (int64_t num, int32_t denom);
static void real_time_delay (int64_t num, int32_t denom);

/** Sets up the timer to interrupt TIMER_FREQ times per second,
and registers the corresponding interrupt. */
void
timer_init (void)
{
        pit_configure_channel (0, 2, TIMER_FREQ);
        intr_register_ext (0x20, timer_interrupt, "8254␣Timer");
}
```

```
41
42      /** Calibrates loops_per_tick, used to implement brief delays. */
43      void
44      timer_calibrate (void)
45      {
46              unsigned high_bit, test_bit;
47
48              ASSERT (intr_get_level () == INTR_ON);
49              printf ("Calibrating␣timer...␣␣");
50
51              /* Approximate loops_per_tick as the largest power-of-two
52              still less than one timer tick. */
53              loops_per_tick = 1u << 10;
54              while (!too_many_loops (loops_per_tick << 1))
55              {
56                      loops_per_tick <<= 1;
57                      ASSERT (loops_per_tick != 0);
58              }
59
60              /* Refine the next 8 bits of loops_per_tick. */
61              high_bit = loops_per_tick;
62              for (test_bit = high_bit >> 1; test_bit != high_bit >> 10; test_bit
                    >>= 1)
63              if (!too_many_loops (high_bit | test_bit))
64              loops_per_tick |= test_bit;
65
66              printf ("%'"PRIu64"␣loops/s.\n", (uint64_t) loops_per_tick *
                    TIMER_FREQ);
67      }
68
69      /** Returns the number of timer ticks since the OS booted. */
70      int64_t
71      timer_ticks (void)
72      {
73              enum intr_level old_level = intr_disable ();
74              int64_t t = ticks;
75              intr_set_level (old_level);
76              return t;
77      }
78
79      /** Returns the number of timer ticks elapsed since THEN, which
80      should be a value once returned by timer_ticks(). */
81      int64_t
82      timer_elapsed (int64_t then)
```

```
83      {
84              return timer_ticks () - then;
85      }
86
87      /** Sleeps for approximately TICKS timer ticks.  Interrupts must
88      be turned on. */
89      void
90      timer_sleep (int64_t wait_ticks)
91      {
92              if (wait_ticks <= 0) return;
93              thread_wait(wait_ticks);
94              // int64_t start = timer_ticks ();
95
96              // ASSERT (intr_get_level () == INTR_ON);
97              // while (timer_elapsed (start) < ticks)
98              //   thread_yield ();
99
100     }
101
102     /** Sleeps for approximately MS milliseconds.  Interrupts must be
103     turned on. */
104     void
105     timer_msleep (int64_t ms)
106     {
107             real_time_sleep (ms, 1000);
108     }
109
110     /** Sleeps for approximately US microseconds.  Interrupts must be
111     turned on. */
112     void
113     timer_usleep (int64_t us)
114     {
115             real_time_sleep (us, 1000 * 1000);
116     }
117
118     /** Sleeps for approximately NS nanoseconds.  Interrupts must be
119     turned on. */
120     void
121     timer_nsleep (int64_t ns)
122     {
123             real_time_sleep (ns, 1000 * 1000 * 1000);
124     }
125
126     /** Busy-waits for approximately MS milliseconds.  Interrupts need
```

```
127     not be turned on.
128
129     Busy waiting wastes CPU cycles, and busy waiting with
130     interrupts off for the interval between timer ticks or longer
131     will cause timer ticks to be lost.  Thus, use timer_msleep()
132     instead if interrupts are enabled. */
133     void
134     timer_mdelay (int64_t ms)
135     {
136             real_time_delay (ms, 1000);
137     }
138
139     /** Sleeps for approximately US microseconds.  Interrupts need not
140     be turned on.
141
142     Busy waiting wastes CPU cycles, and busy waiting with
143     interrupts off for the interval between timer ticks or longer
144     will cause timer ticks to be lost.  Thus, use timer_usleep()
145     instead if interrupts are enabled. */
146     void
147     timer_udelay (int64_t us)
148     {
149             real_time_delay (us, 1000 * 1000);
150     }
151
152     /** Sleeps execution for approximately NS nanoseconds.  Interrupts
153     need not be turned on.
154
155     Busy waiting wastes CPU cycles, and busy waiting with
156     interrupts off for the interval between timer ticks or longer
157     will cause timer ticks to be lost.  Thus, use timer_nsleep()
158     instead if interrupts are enabled.*/
159     void
160     timer_ndelay (int64_t ns)
161     {
162             real_time_delay (ns, 1000 * 1000 * 1000);
163     }
164
165     /** Prints timer statistics. */
166     void
167     timer_print_stats (void)
168     {
169             printf ("Timer:␣%"PRId64"␣ticks\n", timer_ticks ());
170     }
```

```
171
172     /** Timer interrupt handler. */
173     static void
174     timer_interrupt (struct intr_frame *args UNUSED)
175     {
176             ticks++;
177             thread_tick ();
178     }
179
180     /** Returns true if LOOPS iterations waits for more than one timer
181     tick, otherwise false. */
182     static bool
183     too_many_loops (unsigned loops)
184     {
185             /* Wait for a timer tick. */
186             int64_t start = ticks;
187             while (ticks == start)
188             barrier ();
189
190             /* Run LOOPS loops. */
191             start = ticks;
192             busy_wait (loops);
193
194             /* If the tick count changed, we iterated too long. */
195             barrier ();
196             return start != ticks;
197     }
198
199     /** Iterates through a simple loop LOOPS times, for implementing
200     brief delays.
201
202     Marked NO_INLINE because code alignment can significantly
203     affect timings, so that if this function was inlined
204     differently in different places the results would be difficult
205     to predict. */
206     static void NO_INLINE
207     busy_wait (int64_t loops)
208     {
209             while (loops-- > 0)
210             barrier ();
211     }
212
213     /** Sleep for approximately NUM/DENOM seconds. */
214     static void
```

```
215     real_time_sleep (int64_t num, int32_t denom)
216     {
217             /* Convert NUM/DENOM seconds into timer ticks, rounding down.
218
219             (NUM / DENOM) s
220             ---------------------- = NUM * TIMER_FREQ / DENOM ticks.
221             1 s / TIMER_FREQ ticks
222             */
223             int64_t ticks = num * TIMER_FREQ / denom;
224
225             ASSERT (intr_get_level () == INTR_ON);
226             if (ticks > 0)
227             {
228                     /* We're waiting for at least one full timer tick.  Use
229                     timer_sleep() because it will yield the CPU to other
230                     processes. */
231                     timer_sleep (ticks);
232             }
233             else
234             {
235                     /* Otherwise, use a busy-wait loop for more accurate
236                     sub-tick timing. */
237                     real_time_delay (num, denom);
238             }
239     }
240
241     /** Busy-wait for approximately NUM/DENOM seconds. */
242     static void
243     real_time_delay (int64_t num, int32_t denom)
244     {
245             /* Scale the numerator and denominator down by 1000 to avoid
246             the possibility of overflow. */
247             ASSERT (denom % 1000 == 0);
248             busy_wait (loops_per_tick * num / 1000 * TIMER_FREQ / (denom / 1000)
                    );
249     }
```

pintos/src/tests/threads/alarm-wait.c

```
1     /** Creates N threads, each of which sleeps a different, fixed
2     duration, M times.  Records the wake-up order and verifies
3     that it is valid. */
4
5     #include <stdio.h>
```

```
6   #include "tests/threads/tests.h"
7   #include "threads/init.h"
8   #include "threads/malloc.h"
9   #include "threads/synch.h"
10  #include "threads/thread.h"
11  #include "devices/timer.h"
12
13  static void test_sleep (int thread_cnt, int iterations);
14
15  void
16  test_alarm_single (void)
17  {
18          test_sleep (5, 1);
19  }
20
21  void
22  test_alarm_multiple (void)
23  {
24          test_sleep (5, 7);
25  }
26
27  /** Information about the test. */
28  struct sleep_test
29  {
30          int64_t start;              /**< Current time at start of test. */
31          int iterations;             /**< Number of iterations per thread. */
32
33          /* Output. */
34          struct lock output_lock;    /**< Lock protecting output buffer. */
35          int *output_pos;            /**< Current position in output buffer.
                */
36  };
37
38  /** Information about an individual thread in the test. */
39  struct sleep_thread
40  {
41          struct sleep_test *test;     /**< Info shared between all threads.
                */
42          int id;                     /**< Sleeper ID. */
43          int duration;               /**< Number of ticks to sleep. */
44          int iterations;             /**< Iterations counted so far. */
45  };
46
47  static void sleeper (void *);
```

```
48
49      /** Runs THREAD_CNT threads thread sleep ITERATIONS times each. */
50      static void
51      test_sleep (int thread_cnt, int iterations)
52      {
53              struct sleep_test test;
54              struct sleep_thread *threads;
55              int *output, *op;
56              int product;
57              int i;
58
59              /* This test does not work with the MLFQS. */
60              ASSERT (!thread_mlfqs);
61              msg ("Creating␣%d␣threads␣to␣sleep␣%d␣times␣each.", thread_cnt,
                    iterations);
62              msg ("Thread␣0␣sleeps␣10␣ticks␣each␣time,");
63              msg ("thread␣1␣sleeps␣20␣ticks␣each␣time,␣and␣so␣on.");
64              msg ("If␣successful,␣product␣of␣iteration␣count␣and");
65              msg ("sleep␣duration␣will␣appear␣in␣nondescending␣order.");
66              /* Allocate memory. */
67              threads = malloc (sizeof *threads * thread_cnt);
68              output = malloc (sizeof *output * iterations * thread_cnt * 2);
69              if (threads == NULL || output == NULL)
70              PANIC ("couldn't␣allocate␣memory␣for␣test");
71
72              /* Initialize test. */
73              test.start = timer_ticks () + 100;
74              test.iterations = iterations;
75              lock_init (&test.output_lock);
76              test.output_pos = output;
77
78              /* Start threads. */
79              ASSERT (output != NULL);
80              for (i = 0; i < thread_cnt; i++)
81              {
82                      struct sleep_thread *t = threads + i;
83                      char name[16];
84
85                      t->test = &test;
86                      t->id = i;
87                      t->duration = (i + 1) * 10;
88                      t->iterations = 0;
89                      snprintf (name, sizeof name, "thread␣%d", i);
90                      thread_create (name, PRI_DEFAULT, sleeper, t);
```

```
91              }
92
93              /* Wait long enough for all the threads to finish. */
94              timer_sleep (100 + thread_cnt * iterations * 10 + 100);
95              /* Acquire the output lock in case some rogue thread is still
96              running. */
97              lock_acquire (&test.output_lock);
98
99              /* Print completion order. */
100             product = 0;
101             for (op = output; op < test.output_pos; op++)
102             {
103                     struct sleep_thread *t;
104                     int new_prod;
105
106                     ASSERT (*op >= 0 && *op < thread_cnt);
107                     t = threads + *op;
108
109                     new_prod = ++t->iterations * t->duration;
110
111                     msg ("thread␣%d:␣duration=%d,␣iteration=%d,␣product=%d",
112                     t->id, t->duration, t->iterations, new_prod);
113
114                     if (new_prod >= product)
115                     product = new_prod;
116                     else
117                     fail ("thread␣%d␣woke␣up␣out␣of␣order␣(%d␣>␣%d)!",
118                     t->id, product, new_prod);
119             }
120
121             /* Verify that we had the proper number of wakeups. */
122             for (i = 0; i < thread_cnt; i++)
123             if (threads[i].iterations != iterations)
124             fail ("thread␣%d␣woke␣up␣%d␣times␣instead␣of␣%d",
125             i, threads[i].iterations, iterations);
126
127             lock_release (&test.output_lock);
128             free (output);
129             free (threads);
130     }
131
132     /** Sleeper thread. */
133     static void
134     sleeper (void *t_)
```

```
135     {
136             struct sleep_thread *t = t_;
137             struct sleep_test *test = t->test;
138             int i;
139
140             for (i = 1; i <= test->iterations; i++)
141             {
142                     int64_t sleep_until = test->start + i * t->duration;
143                     timer_sleep (sleep_until - timer_ticks ());
144                     lock_acquire (&test->output_lock);
145                     *test->output_pos++ = t->id;
146                     lock_release (&test->output_lock);
147             }
148     }
```

pintos/src/tests/threads/priority-alarm.c

```
1     #include <stdio.h>
2     #include "tests/threads/tests.h"
3     #include "threads/init.h"
4     #include "threads/malloc.h"
5     #include "threads/synch.h"
6     #include "threads/thread.h"
7     #include "devices/timer.h"
8
9     static void high_priority_sleeper(void *);
10
11    void test_priority_alarm(void) {
12            int high_priority = PRI_DEFAULT + 10;
13            thread_create("high_priority_sleeper", high_priority,
                  high_priority_sleeper, NULL);
14
15            msg("Main␣thread␣starts␣running.");
16
17            int64_t start_tick = timer_ticks();
18            while (timer_elapsed(start_tick) < 20);
19
20            msg("Main␣thread␣thread␣changed␣its␣priority␣to␣21");
21
22            thread_set_priority(21);
23
24            msg("Main␣thread␣completed␣execution.");
25
26    }
```

```
27
28      static void
29      high_priority_sleeper(void *aux UNUSED) {
30              msg("High-priority␣thread␣starts␣and␣goes␣to␣sleep.");
31              timer_sleep(10);
32              msg("High-priority␣thread␣woke␣up␣at␣and␣preempts␣the␣main␣thread.")
                    ;
33              msg("High-priority␣thread␣changed␣its␣priority␣to␣26");
34              thread_set_priority(PRI_DEFAULT - 5);
35              msg("High-priority␣thread␣exit.");
36      }
```

pintos/src/tests/threads/priority-alarm.ck

```
1       # -*- perl -*-
2       use strict;
3       use warnings;
4       use tests::tests;
5       check_expected ([<<'EOF']);
6       (priority-alarm) begin
7       (priority-alarm) High-priority thread starts and goes to sleep.
8       (priority-alarm) Main thread starts running.
9       (priority-alarm) High-priority thread woke up at and preempts the main
            thread.
10      (priority-alarm) High-priority thread changed its priority to 26
11      (priority-alarm) Main thread thread changed its priority to 21
12      (priority-alarm) High-priority thread exit.
13      (priority-alarm) Main thread completed execution.
14      (priority-alarm) end
15      EOF
16      pass;
```

pintos/src/tests/threads/tests.c

```
1       #include "tests/threads/tests.h"
2       #include <debug.h>
3       #include <string.h>
4       #include <stdio.h>
5
6       struct test
7       {
8               const char *name;
9               test_func *function;
10      };
11
```

```
12      static const struct test tests[] =
13      {
14              {"alarm-single", test_alarm_single},
15              {"alarm-multiple", test_alarm_multiple},
16              {"alarm-simultaneous", test_alarm_simultaneous},
17              {"alarm-priority", test_alarm_priority},
18              {"alarm-zero", test_alarm_zero},
19              {"alarm-negative", test_alarm_negative},
20              {"priority-change", test_priority_change},
21              {"priority-donate-one", test_priority_donate_one},
22              {"priority-donate-multiple", test_priority_donate_multiple},
23              {"priority-donate-multiple2", test_priority_donate_multiple2},
24              {"priority-donate-nest", test_priority_donate_nest},
25              {"priority-donate-sema", test_priority_donate_sema},
26              {"priority-donate-lower", test_priority_donate_lower},
27              {"priority-donate-chain", test_priority_donate_chain},
28              {"priority-fifo", test_priority_fifo},
29              {"priority-preempt", test_priority_preempt},
30              {"priority-sema", test_priority_sema},
31              {"priority-condvar", test_priority_condvar},
32              {"mlfqs-load-1", test_mlfqs_load_1},
33              {"mlfqs-load-60", test_mlfqs_load_60},
34              {"mlfqs-load-avg", test_mlfqs_load_avg},
35              {"mlfqs-recent-1", test_mlfqs_recent_1},
36              {"mlfqs-fair-2", test_mlfqs_fair_2},
37              {"mlfqs-fair-20", test_mlfqs_fair_20},
38              {"mlfqs-nice-2", test_mlfqs_nice_2},
39              {"mlfqs-nice-10", test_mlfqs_nice_10},
40              {"mlfqs-block", test_mlfqs_block},
41
42
43              {"helloworld", test_hello_world},
44              {"priority-alarm", test_priority_alarm},
45      };
46
47      static const char *test_name;
48
49      /** Runs the test named NAME. */
50      void
51      run_test (const char *name)
52      {
53              const struct test *t;
54
55              for (t = tests; t < tests + sizeof tests / sizeof *tests; t++)
```

```
56              if (!strcmp (name, t->name))
57              {
58                      test_name = name;
59                      msg ("begin");
60                      t->function ();
61                      msg ("end");
62                      return;
63              }
64              PANIC ("no test named \"%s\"", name);
65      }
66
67      /** Prints FORMAT as if with printf(),
68      prefixing the output by the name of the test
69      and following it with a new-line character. */
70      void
71      msg (const char *format, ...)
72      {
73              va_list args;
74
75              printf ("(%s) ", test_name);
76              va_start (args, format);
77              vprintf (format, args);
78              va_end (args);
79              putchar ('\n');
80      }
81
82      /** Prints failure message FORMAT as if with printf(),
83      prefixing the output by the name of the test and FAIL:
84      and following it with a new-line character,
85      and then panics the kernel. */
86      void
87      fail (const char *format, ...)
88      {
89              va_list args;
90
91              printf ("(%s) FAIL: ", test_name);
92              va_start (args, format);
93              vprintf (format, args);
94              va_end (args);
95              putchar ('\n');
96
97              PANIC ("test failed");
98      }
99
```

```
100     /** Prints a message indicating the current test passed. */
101     void
102     pass (void)
103     {
104             printf ("(%s) PASS\n", test_name);
105     }
```

pintos/src/tests/threads/tests.h

```
1     #ifndef TESTS_THREADS_TESTS_H
2     #define TESTS_THREADS_TESTS_H
3
4     void run_test (const char *);
5
6     typedef void test_func (void);
7
8     extern test_func test_alarm_single;
9     extern test_func test_alarm_multiple;
10    extern test_func test_alarm_simultaneous;
11    extern test_func test_alarm_priority;
12    extern test_func test_alarm_zero;
13    extern test_func test_alarm_negative;
14    extern test_func test_priority_change;
15    extern test_func test_priority_donate_one;
16    extern test_func test_priority_donate_multiple;
17    extern test_func test_priority_donate_multiple2;
18    extern test_func test_priority_donate_sema;
19    extern test_func test_priority_donate_nest;
20    extern test_func test_priority_donate_lower;
21    extern test_func test_priority_donate_chain;
22    extern test_func test_priority_fifo;
23    extern test_func test_priority_preempt;
24    extern test_func test_priority_sema;
25    extern test_func test_priority_condvar;
26    extern test_func test_mlfqs_load_1;
27    extern test_func test_mlfqs_load_60;
28    extern test_func test_mlfqs_load_avg;
29    extern test_func test_mlfqs_recent_1;
30    extern test_func test_mlfqs_fair_2;
31    extern test_func test_mlfqs_fair_20;
32    extern test_func test_mlfqs_nice_2;
33    extern test_func test_mlfqs_nice_10;
34    extern test_func test_mlfqs_block;
35    extern test_func test_priority_alarm;
```

```
36
37      extern test_func test_hello_world;
38
39      void msg (const char *, ...);
40      void fail (const char *, ...);
41      void pass (void);
42
43      #endif /**< tests/threads/tests.h */
```

pintos/src/threads/thread.c

```
1       #include "threads/thread.h"
2       #include <debug.h>
3       #include <stddef.h>
4       #include <random.h>
5       #include <stdio.h>
6       #include <string.h>
7       #include "threads/flags.h"
8       #include "threads/interrupt.h"
9       #include "threads/intr-stubs.h"
10      #include "threads/palloc.h"
11      #include "threads/switch.h"
12      #include "threads/synch.h"
13      #include "threads/vaddr.h"
14      #ifdef USERPROG
15      #include "userprog/process.h"
16      #endif
17
18      /** Random value for struct thread's `magic' member.
19      Used to detect stack overflow.  See the big comment at the top
20      of thread.h for details. */
21      #define THREAD_MAGIC 0xcd6abf4b
22
23      static struct list wait_list;
24      /** List of processes in THREAD_READY state, that is, processes
25      that are ready to run but not actually running. */
26      static struct list ready_list;
27
28      /** List of all processes.  Processes are added to this list
29      when they are first scheduled and removed when they exit. */
30      static struct list all_list;
31
32      /** Idle thread. */
33      static struct thread *idle_thread;
```

```
34
35      /** Initial thread, the thread running init.c:main(). */
36      static struct thread *initial_thread;
37
38      /** Lock used by allocate_tid(). */
39      static struct lock tid_lock;
40
41      /** Stack frame for kernel_thread(). */
42      struct kernel_thread_frame
43      {
44              void *eip;                  /**< Return address. */
45              thread_func *function;      /**< Function to call. */
46              void *aux;                  /**< Auxiliary data for function. */
47      };
48
49      /** Statistics. */
50      static long long idle_ticks;    /**< # of timer ticks spent idle. */
51      static long long kernel_ticks;  /**< # of timer ticks in kernel threads. */
52      static long long user_ticks;    /**< # of timer ticks in user programs. */
53
54      /** Scheduling. */
55      #define TIME_SLICE 4            /**< # of timer ticks to give each thread.
            */
56      static unsigned thread_ticks;   /**< # of timer ticks since last yield. */
57
58      /** If false (default), use round-robin scheduler.
59      If true, use multi-level feedback queue scheduler.
60      Controlled by kernel command-line option "-o mlfqs". */
61      bool thread_mlfqs;
62
63      static void kernel_thread (thread_func *, void *aux);
64
65      static void idle (void *aux UNUSED);
66      static struct thread *running_thread (void);
67      static struct thread *next_thread_to_run (void);
68      static void init_thread (struct thread *, const char *name, int priority);
69      static bool is_thread (struct thread *) UNUSED;
70      static void *alloc_frame (struct thread *, size_t size);
71      static void schedule (void);
72      void thread_schedule_tail (struct thread *prev);
73      static tid_t allocate_tid (void);
74      bool thread_more_priority (const struct list_elem *a,
75      const struct list_elem *b,
76      void *aux);
```

```
77
78      bool thread_less_wake_tick(const struct list_elem *a,
79      const struct list_elem *b,
80      void *aux);
81
82      /** Initializes the threading system by transforming the code
83      that's currently running into a thread.  This can't work in
84      general and it is possible in this case only because loader.S
85      was careful to put the bottom of the stack at a page boundary.
86
87      Also initializes the run queue and the tid lock.
88
89      After calling this function, be sure to initialize the page
90      allocator before trying to create any threads with
91      thread_create().
92
93      It is not safe to call thread_current() until this function
94      finishes. */
95      void
96      thread_init (void)
97      {
98              ASSERT (intr_get_level () == INTR_OFF);
99
100             lock_init (&tid_lock);
101             list_init (&ready_list);
102             list_init (&wait_list);
103             list_init (&all_list);
104
105             /* Set up a thread structure for the running thread. */
106             initial_thread = running_thread ();
107             init_thread (initial_thread, "main", PRI_DEFAULT);
108             initial_thread->status = THREAD_RUNNING;
109             initial_thread->tid = allocate_tid ();
110     }
111
112     /** Starts preemptive thread scheduling by enabling interrupts.
113     Also creates the idle thread. */
114     void
115     thread_start (void)
116     {
117             /* Create the idle thread. */
118             struct semaphore idle_started;
119             sema_init (&idle_started, 0);
120             thread_create ("idle", PRI_MIN, idle, &idle_started);
```

```
121             /* Start preemptive thread scheduling. */
122             intr_enable ();
123 
124             /* Wait for the idle thread to initialize idle_thread. */
125             sema_down (&idle_started);
126     }
127 
128     /** Called by the timer interrupt handler at each timer tick.
129     Thus, this function runs in an external interrupt context. */
130     void
131     thread_tick (void)
132     {
133             struct thread *t = thread_current ();
134 
135             /* Update statistics. */
136             if (t == idle_thread)
137             idle_ticks++;
138             #ifdef USERPROG
139             else if (t->pagedir != NULL)
140             user_ticks++;
141             #endif
142             else
143             kernel_ticks++;
144 
145             /* Enforce preemption. */
146             bool yield_flag = thread_unwait();
147             if (++thread_ticks >= TIME_SLICE || yield_flag)
148             intr_yield_on_return ();
149     }
150 
151     /** Prints thread statistics. */
152     void
153     thread_print_stats (void)
154     {
155             printf ("Thread:␣%lld␣idle␣ticks,␣%lld␣kernel␣ticks,␣%lld␣user␣ticks
                \n",
156             idle_ticks, kernel_ticks, user_ticks);
157     }
158 
159     /** Creates a new kernel thread named NAME with the given initial
160     PRIORITY, which executes FUNCTION passing AUX as the argument,
161     and adds it to the ready queue.  Returns the thread identifier
162     for the new thread, or TID_ERROR if creation fails.
163 
```

```
164     If thread_start() has been called, then the new thread may be
165     scheduled before thread_create() returns.  It could even exit
166     before thread_create() returns.  Contrariwise, the original
167     thread may run for any amount of time before the new thread is
168     scheduled.  Use a semaphore or some other form of
169     synchronization if you need to ensure ordering.
170
171     The code provided sets the new thread's `priority' member to
172     PRIORITY, but no actual priority scheduling is implemented.
173     Priority scheduling is the goal of Problem 1-3. */
174     tid_t
175     thread_create (const char *name, int priority,
176     thread_func *function, void *aux)
177     {
178             struct thread *t;
179             struct kernel_thread_frame *kf;
180             struct switch_entry_frame *ef;
181             struct switch_threads_frame *sf;
182             tid_t tid;
183
184             ASSERT (function != NULL);
185
186             /* Allocate thread. */
187             t = palloc_get_page (PAL_ZERO);
188             if (t == NULL)
189             return TID_ERROR;
190
191             /* Initialize thread. */
192             init_thread (t, name, priority);
193             tid = t->tid = allocate_tid ();
194
195             /* Stack frame for kernel_thread(). */
196             kf = alloc_frame (t, sizeof *kf);
197             kf->eip = NULL;
198             kf->function = function;
199             kf->aux = aux;
200
201             /* Stack frame for switch_entry(). */
202             ef = alloc_frame (t, sizeof *ef);
203             ef->eip = (void (*) (void)) kernel_thread;
204
205             /* Stack frame for switch_threads(). */
206             sf = alloc_frame (t, sizeof *sf);
207             sf->eip = switch_entry;
```

```
208             sf->ebp = 0;
209
210             /* Add to run queue. */
211             thread_unblock (t);
212             thread_yield();
213
214             return tid;
215     }
216
217     /** Puts the current thread to sleep.  It will not be scheduled
218     again until awoken by thread_unblock().
219
220     This function must be called with interrupts turned off.  It
221     is usually a better idea to use one of the synchronization
222     primitives in synch.h. */
223     void
224     thread_block (void)
225     {
226             ASSERT (!intr_context ());
227             ASSERT (intr_get_level () == INTR_OFF);
228
229             thread_current ()->status = THREAD_BLOCKED;
230             schedule ();
231     }
232
233     bool thread_unwait()
234     {
235
236             struct list_elem *e;
237             ASSERT (intr_get_level () == INTR_OFF);
238             int64_t cur_tick = timer_ticks();
239
240             bool yield_flag = false;
241
242             for (e = list_begin (&wait_list); e != list_end (&wait_list);
243             e = list_next (e))
244             {
245                     struct thread *t = list_entry (e, struct thread, waitelem);
246                     if (t->wake_tick > cur_tick) break;
247
248                     yield_flag = true;
249                     list_remove(e);
250                     thread_unblock(t);
251             }
```

```
252
253             return yield_flag;
254     }
255
256
257     /** Transitions a blocked thread T to the ready-to-run state.
258     This is an error if T is not blocked.  (Use thread_yield() to
259     make the running thread ready.)
260
261     This function does not preempt the running thread.  This can
262     be important: if the caller had disabled interrupts itself,
263     it may expect that it can atomically unblock a thread and
264     update other data. */
265     void
266     thread_unblock (struct thread *t)
267     {
268             enum intr_level old_level;
269
270             ASSERT (is_thread (t));
271
272             old_level = intr_disable ();
273             ASSERT (t->status == THREAD_BLOCKED);
274             list_insert_ordered(&ready_list, &t->elem, thread_more_priority,
                    NULL);
275             // list_push_back (&ready_list, &t->elem);
276             t->status = THREAD_READY;
277             intr_set_level (old_level);
278     }
279
280
281     void thread_wait(int64_t wait_ticks)
282     {
283
284             ASSERT(intr_get_level() == INTR_ON);
285             enum intr_level old_level = intr_disable();
286             int64_t cur_tick = timer_ticks();
287             struct thread* t = thread_current();
288             t->wake_tick = wait_ticks + cur_tick;
289             // printf("wake_tick: %d\n", t->wake_tick);
290
291             list_insert_ordered(&wait_list, &t->waitelem, thread_less_wake_tick,
                    NULL);
292             thread_block();
293
```

```
294             intr_set_level(old_level);
295     }
296
297     /** Returns the name of the running thread. */
298     const char *
299     thread_name (void)
300     {
301             return thread_current ()->name;
302     }
303
304     /** Returns the running thread.
305     This is running_thread() plus a couple of sanity checks.
306     See the big comment at the top of thread.h for details. */
307     struct thread *
308     thread_current (void)
309     {
310             struct thread *t = running_thread ();
311
312             /* Make sure T is really a thread.
313             If either of these assertions fire, then your thread may
314             have overflowed its stack.  Each thread has less than 4 kB
315             of stack, so a few big automatic arrays or moderate
316             recursion can cause stack overflow. */
317             ASSERT (is_thread (t));
318             ASSERT (t->status == THREAD_RUNNING);
319
320             return t;
321     }
322
323     /** Returns the running thread's tid. */
324     tid_t
325     thread_tid (void)
326     {
327             return thread_current ()->tid;
328     }
329
330     /** Deschedules the current thread and destroys it.  Never
331     returns to the caller. */
332     void
333     thread_exit (void)
334     {
335             ASSERT (!intr_context ());
336
337             #ifdef USERPROG
```

```
338             process_exit ();
339             #endif
340
341             /* Remove thread from all threads list, set our status to dying,
342             and schedule another process.  That process will destroy us
343             when it calls thread_schedule_tail(). */
344             intr_disable ();
345             list_remove (&thread_current()->allelem);
346             thread_current ()->status = THREAD_DYING;
347             schedule ();
348             NOT_REACHED ();
349     }
350
351     /** Yields the CPU.  The current thread is not put to sleep and
352     may be scheduled again immediately at the scheduler's whim. */
353     void
354     thread_yield (void)
355     {
356             struct thread *cur = thread_current ();
357             enum intr_level old_level;
358
359             ASSERT (!intr_context ());
360
361             old_level = intr_disable ();
362             if (cur != idle_thread)
363             list_insert_ordered(&ready_list, &cur->elem, thread_more_priority,
                    NULL);
364             // list_push_back (&ready_list, &cur->elem);
365             // ASSERT(false);
366             cur->status = THREAD_READY;
367             schedule ();
368             intr_set_level (old_level);
369     }
370
371     /** Invoke function 'func' on all threads, passing along 'aux'.
372     This function must be called with interrupts off. */
373     void
374     thread_foreach (thread_action_func *func, void *aux)
375     {
376             struct list_elem *e;
377
378             ASSERT (intr_get_level () == INTR_OFF);
379
380             for (e = list_begin (&all_list); e != list_end (&all_list);
```

```
381             e = list_next (e))
382           {
383                   struct thread *t = list_entry (e, struct thread, allelem);
384                   func (t, aux);
385           }
386   }
387
388   /** Sets the current thread's priority to NEW_PRIORITY. */
389   void
390   thread_set_priority (int new_priority)
391   {
392           thread_current ()->priority = new_priority;
393           thread_yield();
394   }
395
396   /** Returns the current thread's priority. */
397   int
398   thread_get_priority (void)
399   {
400           return thread_current ()->priority;
401   }
402
403   /** Sets the current thread's nice value to NICE. */
404   void
405   thread_set_nice (int nice UNUSED)
406   {
407           /* Not yet implemented. */
408   }
409
410   /** Returns the current thread's nice value. */
411   int
412   thread_get_nice (void)
413   {
414           /* Not yet implemented. */
415           return 0;
416   }
417
418   /** Returns 100 times the system load average. */
419   int
420   thread_get_load_avg (void)
421   {
422           /* Not yet implemented. */
423           return 0;
424   }
```

```
425
426     /** Returns 100 times the current thread's recent_cpu value. */
427     int
428     thread_get_recent_cpu (void)
429     {
430             /* Not yet implemented. */
431             return 0;
432     }
433
434     /** Idle thread.  Executes when no other thread is ready to run.
435
436     The idle thread is initially put on the ready list by
437     thread_start().  It will be scheduled once initially, at which
438     point it initializes idle_thread, "up"s the semaphore passed
439     to it to enable thread_start() to continue, and immediately
440     blocks.  After that, the idle thread never appears in the
441     ready list.  It is returned by next_thread_to_run() as a
442     special case when the ready list is empty. */
443     static void
444     idle (void *idle_started_ UNUSED)
445     {
446             struct semaphore *idle_started = idle_started_;
447             idle_thread = thread_current ();
448             sema_up (idle_started);
449
450             for (;;)
451             {
452                     /* Let someone else run. */
453                     intr_disable ();
454                     thread_block ();
455
456                     /* Re-enable interrupts and wait for the next one.
457
458                     The `sti' instruction disables interrupts until the
459                     completion of the next instruction, so these two
460                     instructions are executed atomically.  This atomicity is
461                     important; otherwise, an interrupt could be handled
462                     between re-enabling interrupts and waiting for the next
463                     one to occur, wasting as much as one clock tick worth of
464                     time.
465
466                     See [IA32-v2a] "HLT", [IA32-v2b] "STI", and [IA32-v3a]
467                     7.11.1 "HLT Instruction". */
468                     asm volatile ("sti; hlt" : : : "memory");
```

```
469             }
470     }
471
472     /** Function used as the basis for a kernel thread. */
473     static void
474     kernel_thread (thread_func *function, void *aux)
475     {
476             ASSERT (function != NULL);
477
478             intr_enable ();       /**< The scheduler runs with interrupts off.
                    */
479             function (aux);       /**< Execute the thread function. */
480             thread_exit ();       /**< If function() returns, kill the thread.
                    */
481     }
482
483     /** Returns the running thread. */
484     struct thread *
485     running_thread (void)
486     {
487             uint32_t *esp;
488
489             /* Copy the CPU's stack pointer into `esp', and then round that
490             down to the start of a page.  Because `struct thread' is
491             always at the beginning of a page and the stack pointer is
492             somewhere in the middle, this locates the curent thread. */
493             asm ("mov␣%%esp,␣%0" : "=g" (esp));
494             return pg_round_down (esp);
495     }
496
497     /** Returns true if T appears to point to a valid thread. */
498     static bool
499     is_thread (struct thread *t)
500     {
501             return t != NULL && t->magic == THREAD_MAGIC;
502     }
503
504     /** Does basic initialization of T as a blocked thread named
505     NAME. */
506     static void
507     init_thread (struct thread *t, const char *name, int priority)
508     {
509             enum intr_level old_level;
510
```

```
511             ASSERT (t != NULL);
512             ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
513             ASSERT (name != NULL);
514
515             memset (t, 0, sizeof *t);
516             t->status = THREAD_BLOCKED;
517             strlcpy (t->name, name, sizeof t->name);
518             t->stack = (uint8_t *) t + PGSIZE;
519             t->priority = priority;
520             t->magic = THREAD_MAGIC;
521             t->wake_tick = -1;
522
523             old_level = intr_disable ();
524
525             list_insert_ordered(&all_list, &t->allelem, thread_more_priority,
                    NULL);
526             // list_push_back (&all_list, &t->allelem);
527             intr_set_level (old_level);
528     }
529
530     /** Allocates a SIZE-byte frame at the top of thread T's stack and
531     returns a pointer to the frame's base. */
532     static void *
533     alloc_frame (struct thread *t, size_t size)
534     {
535             /* Stack data is always allocated in word-size units. */
536             ASSERT (is_thread (t));
537             ASSERT (size % sizeof (uint32_t) == 0);
538
539             t->stack -= size;
540             return t->stack;
541     }
542
543     /** Chooses and returns the next thread to be scheduled.  Should
544     return a thread from the run queue, unless the run queue is
545     empty.  (If the running thread can continue running, then it
546     will be in the run queue.)  If the run queue is empty, return
547     idle_thread. */
548     static struct thread *
549     next_thread_to_run (void)
550     {
551             if (list_empty (&ready_list))
552             return idle_thread;
553             else
```

```
554             return list_entry (list_pop_front (&ready_list), struct thread, elem
                    );
555     }
556
557     /** Completes a thread switch by activating the new thread's page
558     tables, and, if the previous thread is dying, destroying it.
559
560     At this function's invocation, we just switched from thread
561     PREV, the new thread is already running, and interrupts are
562     still disabled.  This function is normally invoked by
563     thread_schedule() as its final action before returning, but
564     the first time a thread is scheduled it is called by
565     switch_entry() (see switch.S).
566
567     It's not safe to call printf() until the thread switch is
568     complete.  In practice that means that printf()s should be
569     added at the end of the function.
570
571     After this function and its caller returns, the thread switch
572     is complete. */
573     void
574     thread_schedule_tail (struct thread *prev)
575     {
576             struct thread *cur = running_thread ();
577
578             ASSERT (intr_get_level () == INTR_OFF);
579
580             /* Mark us as running. */
581             cur->status = THREAD_RUNNING;
582
583             /* Start new time slice. */
584             thread_ticks = 0;
585
586             #ifdef USERPROG
587             /* Activate the new address space. */
588             process_activate ();
589             #endif
590
591             /* If the thread we switched from is dying, destroy its struct
592             thread.  This must happen late so that thread_exit() doesn't
593             pull out the rug under itself.  (We don't free
594             initial_thread because its memory was not obtained via
595             palloc().) */
596             if (prev != NULL && prev->status == THREAD_DYING && prev !=
```

```
                initial_thread)
597             {
598                     ASSERT (prev != cur);
599                     palloc_free_page (prev);
600             }
601     }
602
603     /** Schedules a new process.  At entry, interrupts must be off and
604     the running process's state must have been changed from
605     running to some other state.  This function finds another
606     thread to run and switches to it.
607
608     It's not safe to call printf() until thread_schedule_tail()
609     has completed. */
610     static void
611     schedule (void)
612     {
613             struct thread *cur = running_thread ();
614             struct thread *next = next_thread_to_run ();
615             struct thread *prev = NULL;
616
617             ASSERT (intr_get_level () == INTR_OFF);
618             ASSERT (cur->status != THREAD_RUNNING);
619             ASSERT (is_thread (next));
620
621             if (cur != next)
622             prev = switch_threads (cur, next);
623             thread_schedule_tail (prev);
624     }
625
626     /** Returns a tid to use for a new thread. */
627     static tid_t
628     allocate_tid (void)
629     {
630             static tid_t next_tid = 1;
631             tid_t tid;
632
633             lock_acquire (&tid_lock);
634             tid = next_tid++;
635             lock_release (&tid_lock);
636
637             return tid;
638     }
639
```

```
640     /** Offset of `stack' member within `struct thread'.
641     Used by switch.S, which can't figure it out on its own. */
642     uint32_t thread_stack_ofs = offsetof (struct thread, stack);
643
644     bool thread_more_priority (const struct list_elem *a,
645     const struct list_elem *b,
646     void *aux)
647     {
648             int x = list_entry(a, struct thread, elem)->priority;
649             int y = list_entry(b, struct thread, elem)->priority;
650             return x > y;
651     }
652
653     bool thread_less_wake_tick(const struct list_elem *a,
654     const struct list_elem *b,
655     void *aux)
656     {
657             int64_t x = list_entry(a, struct thread, waitelem)->wake_tick;
658             int64_t y = list_entry(b, struct thread, waitelem)->wake_tick;
659             return x < y;
660     }
```

pintos/src/threads/thread.h

```
1     #ifndef THREADS_THREAD_H
2     #define THREADS_THREAD_H
3
4     #include <debug.h>
5     #include <list.h>
6     #include <stdint.h>
7
8     /** States in a thread's life cycle. */
9     enum thread_status
10    {
11            THREAD_RUNNING,     /**< Running thread. */
12            THREAD_READY,       /**< Not running but ready to run. */
13            THREAD_BLOCKED,     /**< Waiting for an event to trigger. */
14            THREAD_DYING        /**< About to be destroyed. */
15    };
16
17    /** Thread identifier type.
18    You can redefine this to whatever type you like. */
19    typedef int tid_t;
20    #define TID_ERROR ((tid_t) -1)          /**< Error value for tid_t. */
```

```
21
22      /** Thread priorities. */
23      #define PRI_MIN 0                       /**< Lowest priority. */
24      #define PRI_DEFAULT 31                  /**< Default priority. */
25      #define PRI_MAX 63                      /**< Highest priority. */
26
27      /** A kernel thread or user process.
28
29      Each thread structure is stored in its own 4 kB page.  The
30      thread structure itself sits at the very bottom of the page
31      (at offset 0).  The rest of the page is reserved for the
32      thread's kernel stack, which grows downward from the top of
33      the page (at offset 4 kB).  Here's an illustration:
34
35      4 kB +---------------------------------+
36      |          kernel stack           |
37      |                |                |
38      |                |                |
39      |                V                |
40      |         grows downward          |
41      |                                 |
42      |                                 |
43      |                                 |
44      |                                 |
45      |                                 |
46      |                                 |
47      |                                 |
48      |                                 |
49      +---------------------------------+
50      |              magic              |
51      |                :                |
52      |                :                |
53      |               name              |
54      |              status             |
55      0 kB +---------------------------------+
56
57      The upshot of this is twofold:
58
59      1. First, `struct thread' must not be allowed to grow too
60      big.  If it does, then there will not be enough room for
61      the kernel stack.  Our base `struct thread' is only a
62      few bytes in size.  It probably should stay well under 1
63      kB.
64
```

```
65      2. Second, kernel stacks must not be allowed to grow too
66      large.  If a stack overflows, it will corrupt the thread
67      state.  Thus, kernel functions should not allocate large
68      structures or arrays as non-static local variables.  Use
69      dynamic allocation with malloc() or palloc_get_page()
70      instead.
71
72      The first symptom of either of these problems will probably be
73      an assertion failure in thread_current(), which checks that
74      the `magic' member of the running thread's `struct thread' is
75      set to THREAD_MAGIC.  Stack overflow will normally change this
76      value, triggering the assertion. */
77      /** The `elem' member has a dual purpose.  It can be an element in
78      the run queue (thread.c), or it can be an element in a
79      semaphore wait list (synch.c).  It can be used these two ways
80      only because they are mutually exclusive: only a thread in the
81      ready state is on the run queue, whereas only a thread in the
82      blocked state is on a semaphore wait list. */
83      struct thread
84      {
85              /* Owned by thread.c. */
86              tid_t tid;                          /**< Thread identifier. */
87              enum thread_status status;          /**< Thread state. */
88              char name[16];                      /**< Name (for debugging
                    purposes). */
89              uint8_t *stack;                     /**< Saved stack pointer. */
90              int priority;                       /**< Priority. */
91              struct list_elem allelem;           /**< List element for all
                    threads list. */
92
93              struct list_elem waitelem;
94              int64_t wake_tick;
95
96              /* Shared between thread.c and synch.c. */
97              struct list_elem elem;              /**< List element. */
98
99              #ifdef USERPROG
100             /* Owned by userprog/process.c. */
101             uint32_t *pagedir;                  /**< Page directory. */
102             #endif
103
104             /* Owned by thread.c. */
105             unsigned magic;                     /**< Detects stack overflow. */
106     };
```

```
107
108     /** If false (default), use round-robin scheduler.
109     If true, use multi-level feedback queue scheduler.
110     Controlled by kernel command-line option "-o mlfqs". */
111     extern bool thread_mlfqs;
112
113     void thread_init (void);
114     void thread_start (void);
115
116     void thread_tick (void);
117     void thread_print_stats (void);
118
119     typedef void thread_func (void *aux);
120     tid_t thread_create (const char *name, int priority, thread_func *, void *);
121
122     void thread_block (void);
123     void thread_unblock (struct thread *);
124     void thread_wait(int64_t ticks);
125     bool thread_unwait();
126
127     struct thread *thread_current (void);
128     tid_t thread_tid (void);
129     const char *thread_name (void);
130
131     void thread_exit (void) NO_RETURN;
132     void thread_yield (void);
133
134     /** Performs some operation on thread t, given auxiliary data AUX. */
135     typedef void thread_action_func (struct thread *t, void *aux);
136     void thread_foreach (thread_action_func *, void *);
137
138     int thread_get_priority (void);
139     void thread_set_priority (int);
140
141     int thread_get_nice (void);
142     void thread_set_nice (int);
143     int thread_get_recent_cpu (void);
144     int thread_get_load_avg (void);
145
146     #endif /**< threads/thread.h */
```