实验报告：Pintos Priority

课程名称：操作系统	年级：2023 级本科	上机实践成绩：
指导教师：张民	姓名：张梓卫	
上机实践名称：Pintos 忙等待问题	学号：10235101526	上机实践日期：2024/11/18
上机实践编号：（4）	组号：	上机实践时间：2 学时

目录

一 实验目的

本实验的目标是优化 Pintos 操作系统中的忙等待问题，通过引入更加高效的调度和同步机制，以减少 CPU 的空转浪费并提高系统资源利用率。传统的忙等待在资源不足或线程竞争时会导致 CPU 不断循环查询，既浪费了宝贵的处理资源，又可能阻塞其他线程的运行。

本次实验作出修改的代码如下所示：

同时上传到了 Github 之上，仓库地址为：https://github.com/Shichien/ECNU-23-SEI-Homework

请在上传的 PDF 文件中直接点击粉色链接即可。

二 内容与设计思想

三 使用环境

使用 Docker v27.1.1 进行 Pintos 的安装实验，基于 Windows 11 操作系统使用 WSL2。

实验报告使用 LaTeX 进行撰写，使用 VSCode + Vim 编辑器进行文本编辑。

四 实验过程与分析

1 进入实验背景

首先按照 PPT 的内容添加了 Printf 函数，显示

```
300 /** Yields the CPU.  The current thread is not put to sleep and
301    may be scheduled again immediately at the scheduler's whim. */
302 void
303 thread_yield (void)
304 {
305   struct thread *cur = thread_current ();
306   enum intr_level old_level;
307
308   ASSERT (!intr_context ());
309
310   old_level = intr_disable ();
311   if (cur != idle_thread)
312     // list_push_back (&ready_list, &cur->elem);
313     list_insert_ordered(&ready_list, &cur->elem, (list_less_func *) &priority_less_func, NULL);
314     printf("Yield: thread %s at tick %lld.\n", cur->name, timer_ticks());
315   cur->status = THREAD_READY;
316   schedule ();
317   intr_set_level (old_level);
318 }
```

图 1: 添加 Printf 函数

最初忘记使用 Make 编译，所以修改代码后输出仍未发生改变，使用 Make 命令后，可以看到有了课件上相似的输出：注意，这里的 Yield 刚开始打错了，后续在实验中进行了修改。

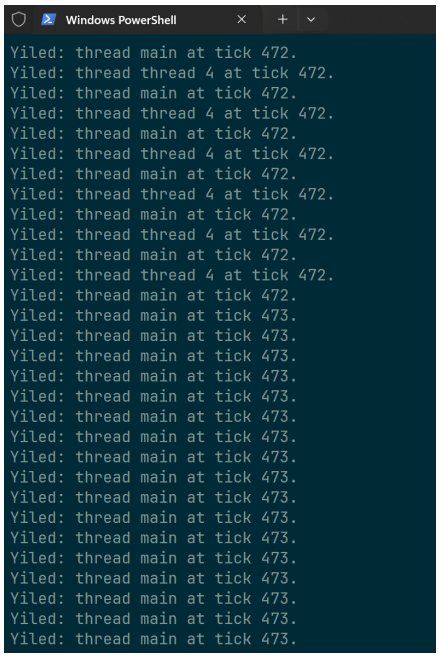

图 2: 编译后输出

忙等待是一种线程或进程等待资源或条件时的方式。在忙等待期间，线程不断地通过循环检查条件是否满足，而不主动让出 CPU。这种行为导致线程虽然在逻辑上是“等待”，但实际上仍然占用 CPU 资源进行无效计算。

我们对输出的某些片段进行分析：

```
1 Yield: thread main at tick 167.
```

```
Yield: thread thread 0 at tick 167.
Yield: thread thread 1 at tick 167.
Yield: thread thread 2 at tick 167.
Yield: thread thread 3 at tick 167.
Yield: thread thread 4 at tick 167.
...
Yield: thread main at tick 169.
Yield: thread thread 0 at tick 169.
Yield: thread thread 1 at tick 169.
Yield: thread thread 2 at tick 169.
Yield: thread thread 3 at tick 169.
Yield: thread thread 4 at tick 169.
```

主线程和多个子线程（thread 0 到 thread 4）在每个时钟周期（tick）内，都会多次输出 Yield。线程没有实际执行任务，而是在轮流进行 yield 操作。这说明线程是在无条件地反复轮询，而没有进行有效的睡眠或释放 CPU 的操作。

在最后的片段中，如下：主线程在多个连续的 tick 内，不断重复同样的 Yield 输出。同样，这表明主线程也在不停地轮询，等待某个条件满足。

```
Yield: thread main at tick 564.
Yield: thread main at tick 564.
Yield: thread main at tick 564.
Yield: thread main at tick 564.
...
Yield: thread main at tick 570.
```

线程使用了 yield 语句，而不是睡眠或等待机制。yield 的作用是将 CPU 时间片让给其他线程，但如果没有其他线程需要运行，则线程会立即重新获得 CPU 时间片并继续执行。这导致线程不断进入循环状态，占用 CPU。

2 实现休眠

```
102     /* Owned by thread.c. */
103     unsigned magic;                 /**< Detects stack overflow. */
104     int64_t wake_time               /** <Edit>: Time at which thread should wake up. */;
105   };
```

图 3: 修改 thread.h

```
/** States in a thread's life cycle. */
enum thread_status
  {
    THREAD_RUNNING,     /**< Running thread. */
    THREAD_READY,       /**< Not running but ready to run.
    THREAD_BLOCKED,     /**< Waiting for an event to trigge
    THREAD_DYING,       /**< About to be destroyed. */
    THREAD_SLEEP        /** <Edit>: Sleeping thread. */
  };
```

图 4: 修改 thread.h

在 thread.h 中，我们增加了 thread_sleep 函数，定义休眠接口，供其他模块调用，用来让线程自身状态由运行状态转为休眠状态。

```
  void thread_sleep(int64_t ticks);
```

在 thread.c 中，实现 thread_sleep 函数。

```
void thread_sleep(int64_t ticks) {
  if(ticks<=0) return;
    struct thread *cur = thread_current();

  enum intr_level old_level = intr_disable();
```

```
  if(cur!=idle_thread) {
    cur->status = THREAD_SLEEP;
    cur->wake_time = timer_ticks() + ticks;
    schedule();
  }
  intr_set_level(old_level);
}
```

- 检查休眠时间：如果休眠时间 ticks 小于或等于 0，直接返回，不进行休眠。
- 获取当前线程：调用 thread_current() 获取当前线程的控制块 struct thread。
- 禁用中断：禁用中断以保证线程状态和数据的修改是原子的，避免竞争条件。
- 线程状态更新：如果当前线程不是空闲线程（idle_thread），将其状态设置为 THREAD_SLEEP。
- 设置唤醒时间 wake_time 为当前时钟计数加上休眠的 ticks。
- 调度：调用 schedule() 将线程切换出去，当前线程进入睡眠状态。

在 timer.c 中的 timer_interrupt 函数中添加检查唤醒线程的逻辑。

定时器中断每次触发时，只检查需要唤醒的线程并唤醒它们，不进行不必要的轮询。唤醒的线程重新参与调度后执行自己的任务。

```
/** Timer interrupt handler. */
static void
timer_interrupt (struct intr_frame *args UNUSED)
{
  ticks++;
  thread_tick ();
  check_and_wakeup_sleep_thread();
}
```

图 5: 修改 timer.c

在 thread.c 中实现，同时与上同理，在 thread.h 中添加定义:

在调用 thread_sleep 后，线程被阻塞并加入 sleep_list，不再占用 CPU。阻塞线程不会消耗任何 CPU 时间，而是等待时钟中断唤醒。

唤醒逻辑如下：

- 获取当前时钟计数：调用 timer_ticks() 获取当前的时钟计数 cur_ticks。
- 遍历所有线程：遍历 all_list 中的所有线程，找到状态为 THREAD_SLEEP 且唤醒时间已到的线程。
- 更新线程状态：将满足条件的线程状态从 THREAD_SLEEP 更新为 THREAD_READY。
- 使用 list_insert_ordered 将线程加入 ready_list，确保就绪队列按照优先级排序。

```
void check_and_wakeup_sleep_thread(void) {
  struct list_elem *e = list_begin(&all_list);
  int64_t cur_ticks = timer_ticks();

  while (e != list_end(&all_list)) {
    struct thread *t = list_entry(e, struct thread, allelem);
    enum intr_level old_level = intr_disable();
    if (t->status == THREAD_SLEEP && cur_ticks >= t->wake_time) {
        t->status = THREAD_READY;
        list_insert_ordered(&ready_list, &t->elem, (list_less_func *) &priority_less_func, NULL);
        printf("Wake up thread %s at tick %lld.\n", t->name, cur_ticks);
      }
    e = list_next(e);
    intr_set_level(old_level);
  }
}
```

在定时器中断（timer_interrupt）中实现了对 sleep_list 的轮询检查，将满足唤醒条件的线程从睡眠队列中移出并唤醒。

3 解决忙等待

再次输入命令 `pintos -v -- -q run alarm-multiple`，此时的输出如下所示：
这里只截取了一部分可以看出忙等待已经获得解决的输出：

```
Wake up thread thread 1 at tick 203.
Wake up thread thread 3 at tick 203.
Wake up thread thread 2 at tick 213.
Wake up thread thread 1 at tick 223.
Wake up thread thread 4 at tick 223.
Wake up thread thread 1 at tick 243.
Wake up thread thread 2 at tick 243.
Wake up thread thread 3 at tick 243.
Wake up thread thread 1 at tick 263.
Wake up thread thread 2 at tick 273.
Wake up thread thread 4 at tick 273.
Wake up thread thread 3 at tick 283.
Wake up thread thread 2 at tick 303.
Wake up thread thread 3 at tick 323.
Wake up thread thread 4 at tick 323.
Wake up thread thread 2 at tick 333.
Wake up thread thread 3 at tick 363.
Wake up thread thread 4 at tick 373.
Wake up thread thread 3 at tick 403.
Wake up thread thread 4 at tick 423.
Wake up thread thread 4 at tick 473.
Wake up thread main at tick 573.
(alarm-multiple) thread 0: duration=10, iteration=1, product=10
(alarm-multiple) thread 0: duration=10, iteration=2, product=20
(alarm-multiple) thread 1: duration=20, iteration=1, product=20
(alarm-multiple) thread 0: duration=10, iteration=3, product=30
(alarm-multiple) thread 2: duration=30, iteration=1, product=30
(alarm-multiple) thread 0: duration=10, iteration=4, product=40
(alarm-multiple) thread 1: duration=20, iteration=2, product=40
(alarm-multiple) thread 3: duration=40, iteration=1, product=40
(alarm-multiple) thread 0: duration=10, iteration=5, product=50
(alarm-multiple) thread 4: duration=50, iteration=1, product=50
(alarm-multiple) thread 0: duration=10, iteration=6, product=60
(alarm-multiple) thread 1: duration=20, iteration=3, product=60
(alarm-multiple) thread 2: duration=30, iteration=2, product=60
(alarm-multiple) thread 0: duration=10, iteration=7, product=70
(alarm-multiple) thread 1: duration=20, iteration=4, product=80
(alarm-multiple) thread 3: duration=40, iteration=2, product=80
```

图 6: 修改后的输出

```
Wake up thread thread 0 at tick 133.
Wake up thread thread 1 at tick 143.
Wake up thread thread 2 at tick 153.
```

日志显示线程只在预期的时间点（wakeup_tick）被唤醒，每个线程只在需要执行任务时被唤醒，没有频繁轮询。

```
(alarm-multiple) thread 0: duration=10, iteration=1, product=10
...
(alarm-multiple) thread 4: duration=50, iteration=7, product=350
```

在 alarm-multiple 日志中，每个线程只在被唤醒后完成一次任务，并再次进入睡眠。日志显示的 product 值是单调递增的，表明线程按预期顺序执行，没有抢占或忙等待。

4 实现苏醒后抢占

其实，这部分只要在 check_and_wakeup_sleep_thread 函数中，稍作修改即可，实现逻辑如下的注释所示：

```

bool check_and_wakeup_sleep_thread(void) {
    struct list_elem *e = list_begin(&all_list);
    int64_t cur_ticks = timer_ticks();
    bool need_preempt = false;

    while (e != list_end(&all_list)) {
        struct thread *t = list_entry(e, struct thread, allelem);
        if (t->status == THREAD_SLEEP && cur_ticks >= t->wake_time) {
            t->status = THREAD_READY;
            list_insert_ordered(&ready_list, &t->elem, priority_less_func, NULL);
```

```
            printf("Wake up thread %s at tick %lld.\n", t->name, cur_ticks);
            if (t->priority > thread_current()->priority) {
              // 判断是否需要抢占：
              // 如果被唤醒线程后，函数将其优先级与当前运行线程 (thread_current()) 的优先级进行比较。
              // 如果被唤醒线程的优先级更高，设置 need_preempt 为 true，表示需要进行上下文切换，以便高优先级线程立即运行。
                need_preempt = true;
            }
        }
        e = list_next(e);
    }
    return need_preempt;
}
```

苏醒后抢占

在这个函数里，使用了 `list_insert_ordered(&ready_list, &t->elem, priority_less_func, NULL);`

确保线程按照优先级从高到低排列，因此 ready_list 的第一个线程始终是优先级最高的线程。

另外，我们需要调用这个函数，在 thread_tcik 中检查，如果返回 true，则调用 intr_yield_on_return() 来进行上下文切换

```
  void
thread_tick (void)
{
  struct thread *t = thread_current ();

  /* Update statistics. */
  if (t == idle_thread)
    idle_ticks++;
#ifdef USERPROG
  else if (t->pagedir != NULL)
    user_ticks++;
#endif
  else
    kernel_ticks++;

  /* Check if the current thread needs to be woken up. */
  bool need_preempt = check_and_wakeup_sleep_thread();

  /* Enforce preemption. */
  if (++thread_ticks >= TIME_SLICE || need_preempt) {
  printf("Tick: thread %s at tick %lld.\n", t->name, timer_ticks());
  // 为了能够看出是否实现了苏醒后抢占，我们可以添加一些提示信息
    intr_yield_on_return ();
  }
}
```

苏醒后抢占

这一次输出片段如下：

```
Tick: thread idle at tick 213.
Yield: thread idle at tick 213.
Wake up thread thread 1 at tick 223 (priority=31).
Thread thread 1 will preempt current thread idle.
Wake up thread thread 4 at tick 223 (priority=31).
Thread thread 4 will preempt current thread idle.
Tick: thread idle at tick 223.
Yield: thread idle at tick 223.
Wake up thread thread 1 at tick 243 (priority=31).
Thread thread 1 will preempt current thread idle.
Wake up thread thread 2 at tick 243 (priority=31).
Thread thread 2 will preempt current thread idle.
Wake up thread thread 3 at tick 243 (priority=31).
Thread thread 3 will preempt current thread idle.
```

可以看到，线程 1 到 4 都被唤醒，并且在被唤醒后，线程 1 开始运行，而线程 4 则被抢占，以便线程 1 运行。

5 实验结果

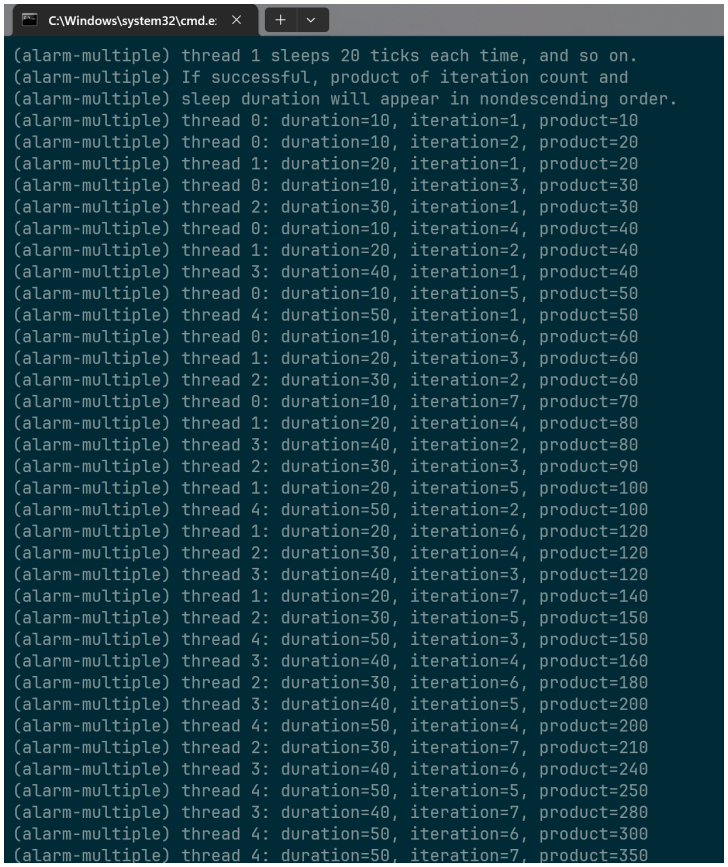

图 7: 实验结果

最后的”product” 消息显示线程完成其睡眠循环的顺序。乘积值的非降序顺序表明线程根据其睡眠持续时间和优先级按正确顺序被唤醒和执行。

五 总结

1 实验成果

1. 实现线程休眠

   - 在 `thread.c` 中实现了 `thread_sleep()` 函数，使线程在不需要执行任务时进入休眠状态，并在唤醒时间到达时由定时器中断恢复运行。
   - 休眠机制避免了线程频繁调用 `thread_yield()` 导致的资源浪费，使得线程在等待资源时不会占用 CPU，显著提高了资源利用效率。

2. 实现线程唤醒

   - 通过修改 `check_and_wakeup_sleep_thread()` 函数，定时器中断在适当的时刻检查所有线程，并将符合条件的线程从睡眠队列移动到就绪队列，同时更新其状态为 `THREAD_READY`。
   - 这一机制确保了线程仅在需要时被唤醒，进一步减少了不必要的 CPU 开销。

3. 实现苏醒后抢占

- 在线程唤醒的逻辑中，我们比较唤醒线程与当前运行线程的优先级。如果唤醒线程的优先级更高，则触发抢占，使高优先级线程得以优先运行。
- 这一机制提升了操作系统的实时性和响应性，使得优先级调度策略得以有效执行。

2 附录

参考资料：

- https://pkuflyingpig.gitbook.io/pintos/project-description/lab1-threads/faq