

实验报告：忙等待问题

课程名称：操作系统实践

年级：23 级

上机实践成绩：

指导教师：张民

姓名：叶晓良

上机实践名称：忙等待问题

学号：10235101427

上机实践日期：2024/11/18

上机实践编号：

组号：

上机实践时间：2024/11/18

一、目的

本实验旨在深入研究 Pintos 操作系统中的忙等待（busy waiting）问题，并通过改进线程调度机制来减少 CPU 资源的无效占用，提升系统的整体性能。具体目标包括：

1. 分析并展示现有 `timer_sleep()` 函数中的忙等待现象。
2. 实现线程的 `sleep` 功能，替代忙等待机制。
3. 确保线程在唤醒后能正确地根据优先级抢占 CPU，减少调度延迟。
4. 通过实验结果验证改进效果，并撰写详细的实验报告。

二、内容与设计思想

1. 忙等待现象的分析与展示

在测试函数 `test_alarm_priority` 中，我们创建了 10 个优先级各不相同的线程。使用 `timer_sleep()` 函数让线程休眠一段时间后再唤醒继续执行。当前的 `timer_sleep()` 实现方式是线程在休眠期间不断将自己的状态从就绪到运行，再回到就绪，这种方式导致了 CPU 资源的持续占用，形成了忙等待现象。

具体的忙等待过程如下：

- 当前线程调用 `timer_sleep()`，进入休眠状态。
- 在 ticks（时间片）内，系统不断将该线程重新加入到就绪队列中，然后再次调度执行。
- 这个过程循环进行，直到倒计时结束，线程被正式唤醒。

为了直观展示忙等待，我们在 `thread_yield()` 函数中添加了打印语句，记录每次线程状态转换的信息。通过运行指令 `pintos -v -- -q run alarm-multiple`，可以观察到系统在 ticks 时间内频繁的线程调度行为，这就是忙等待的具体表现。

2. 实现线程的 sleep 功能

为了消除忙等待，我们实现了线程的 `sleep` 功能。具体设计思路如下：

- 线程调用 `sleep()` 时，主动放弃 CPU 执行权，并将自己从运行状态变为非就绪状态。
- 线程设定一个倒计时时间，倒计时结束后，系统将其重新唤醒并加入就绪队列。

在 wake up 时，我们同样添加了打印语句，记录线程唤醒的信息。通过重新运行 `pintos -v -- -q run alarm-multiple`，可以观察到线程在休眠期间不再频繁地进行状态转换，CPU

占用率显著降低。

3. 实现苏醒后优先级抢占

在苏醒后的线程调度中，我们引入了优先级抢占机制：

- 当一个休眠中的线程被唤醒时，如果当前运行线程的优先级低于该苏醒线程，系统将立即进行优先级抢占，让更高优先级的线程立即执行。

这一设计的实现思路如下：

- 修改 `check_and_wakeup_sleep_thread` 函数算法，确保在每次线程唤醒时，系统都能根据最新的优先级进行调度。

- 通过在 `check_and_wakeup_sleep_thread` 函数中添加条件判断，确保高优先级线程能够优先执行。

具体的代码修改包括在 `check_and_wakeup_sleep_thread` 函数中添加对唤醒线程优先级的检查，并在满足条件时触发线程切换。

三、使用环境

本次实验使用 Visual Studio Code (VSCode) 作为开发环境。VSCode 是微软研发的一款免费 IDE，适用于 Windows 和 macOS 系统，尤其适合进行系统开发。相比于 IntelliJ 系列的 IDE，VSCode 不仅免费，而且在远程开发方面有较好的支持。

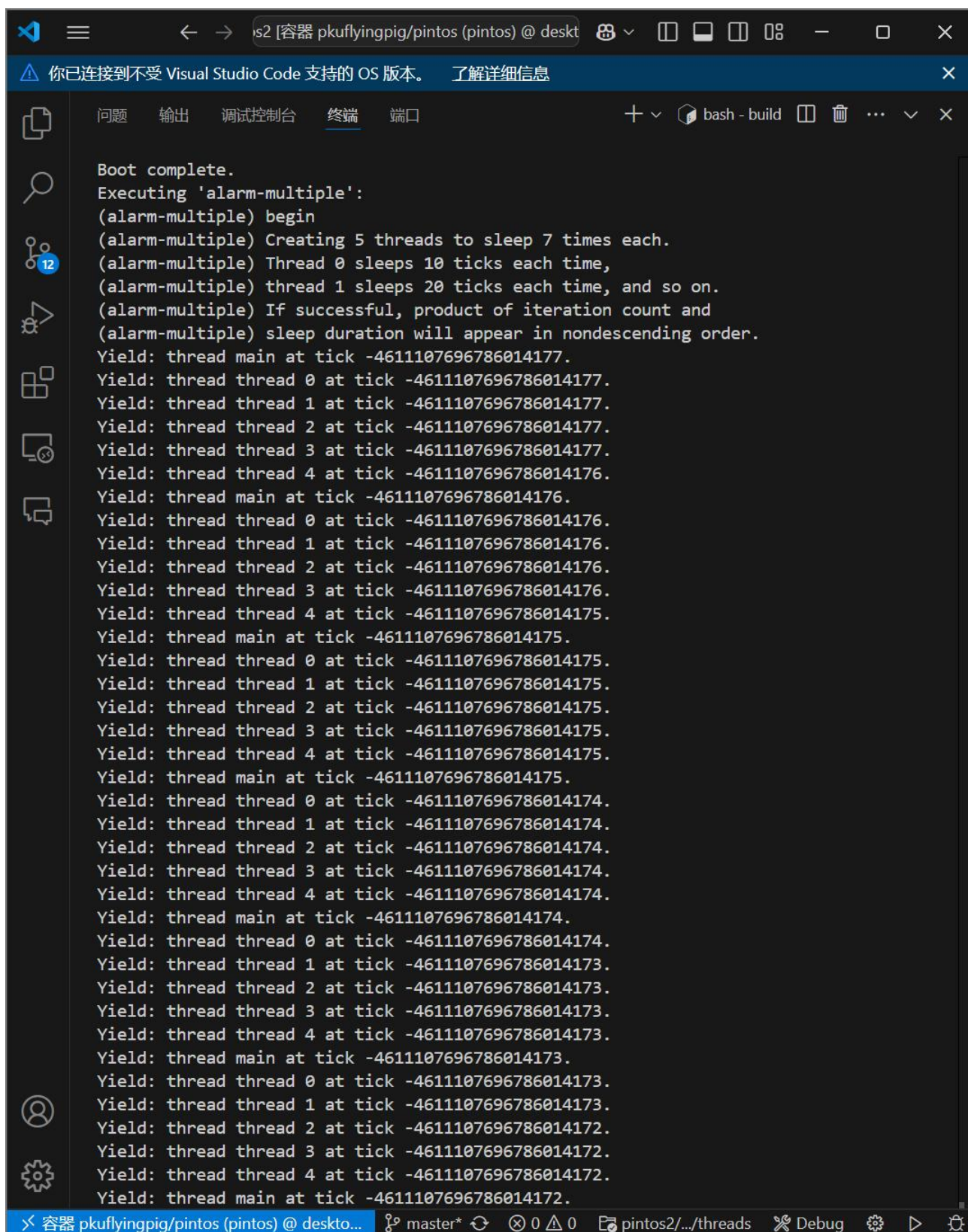
四、实验过程

1. 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`。

```
/** Yields the CPU. The current thread is not put to sleep and
    | may be scheduled again immediately at the scheduler's whim. */
void
thread_yield (void)
{
    struct thread *cur = thread_current ();
    enum intr_level old_level;

    ASSERT (!intr_context ());

    old_level = intr_disable ();
    if (cur != idle_thread)
        list_insert_ordered (&ready_list, &cur->elem, less_by_priority, NULL);
    printf("Yield: thread %s at tick %lld.\n", cur->name,
timer_ticks());
    cur->status = THREAD_READY;
    schedule ();
    intr_set_level (old_level);
}
```



The screenshot shows a Visual Studio Code window with a terminal open. The terminal title is "s2 [容器 pkuflyingpig/pintos (pintos) @ deskto". A blue notification bar at the top states "你已连接到不受 Visual Studio Code 支持的 OS 版本。 了解详细信息". The terminal tabs include "问题", "输出", "调试控制台", "终端", and "端口". The active tab is "终端", which shows the output of a "bash - build" session. The output text is as follows:

```
Boot complete.
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
(alarm-multiple) If successful, product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
Yield: thread main at tick -4611107696786014177.
Yield: thread thread 0 at tick -4611107696786014177.
Yield: thread thread 1 at tick -4611107696786014177.
Yield: thread thread 2 at tick -4611107696786014177.
Yield: thread thread 3 at tick -4611107696786014177.
Yield: thread thread 4 at tick -4611107696786014176.
Yield: thread main at tick -4611107696786014176.
Yield: thread thread 0 at tick -4611107696786014176.
Yield: thread thread 1 at tick -4611107696786014176.
Yield: thread thread 2 at tick -4611107696786014176.
Yield: thread thread 3 at tick -4611107696786014176.
Yield: thread thread 4 at tick -4611107696786014175.
Yield: thread main at tick -4611107696786014175.
Yield: thread thread 0 at tick -4611107696786014175.
Yield: thread thread 1 at tick -4611107696786014175.
Yield: thread thread 2 at tick -4611107696786014175.
Yield: thread thread 3 at tick -4611107696786014175.
Yield: thread thread 4 at tick -4611107696786014175.
Yield: thread main at tick -4611107696786014175.
Yield: thread thread 0 at tick -4611107696786014174.
Yield: thread thread 1 at tick -4611107696786014174.
Yield: thread thread 2 at tick -4611107696786014174.
Yield: thread thread 3 at tick -4611107696786014174.
Yield: thread thread 4 at tick -4611107696786014174.
Yield: thread main at tick -4611107696786014174.
Yield: thread thread 0 at tick -4611107696786014174.
Yield: thread thread 1 at tick -4611107696786014173.
Yield: thread thread 2 at tick -4611107696786014173.
Yield: thread thread 3 at tick -4611107696786014173.
Yield: thread thread 4 at tick -4611107696786014173.
Yield: thread main at tick -4611107696786014173.
Yield: thread thread 0 at tick -4611107696786014173.
Yield: thread thread 1 at tick -4611107696786014173.
Yield: thread thread 2 at tick -4611107696786014172.
Yield: thread thread 3 at tick -4611107696786014172.
Yield: thread thread 4 at tick -4611107696786014172.
Yield: thread main at tick -4611107696786014172.
```

The bottom status bar shows the container name "容器 pkuflyingpig/pintos (pintos) @ deskto...", the branch "master*", and the file path "pintos2/.../threads".

Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.

(alarm-multiple) If successful, product of iteration count and (alarm-multiple) sleep duration will appear in nondescending order.

测试用例开始执行，创建了 5 个线程，每个线程将休眠 7 次，每次休眠的时间递增（10 ticks, 20 ticks, 等）。如果测试成功，每个线程的迭代次数与休眠时间的乘积将按非降序排列。

忙等行为：

```
Yield: thread main at tick -4611107696786014173.
Yield: thread thread 0 at tick -4611107696786014173.
Yield: thread thread 1 at tick -4611107696786014173.
Yield: thread thread 2 at tick -4611107696786014173.
Yield: thread thread 3 at tick -4611107696786014173.
Yield: thread thread 4 at tick -4611107696786014173.
Yield: thread main at tick -4611107696786014173.
Yield: thread thread 0 at tick -4611107696786014172.
Yield: thread thread 1 at tick -4611107696786014172.
Yield: thread thread 2 at tick -4611107696786014172.
Yield: thread thread 3 at tick -4611107696786014172.
Yield: thread thread 4 at tick -4611107696786014172.
```

... ..

这里的日志显示了多个线程在不停地切换（Yield）。这种频繁的线程切换通常表明系统正处于“忙等”状态。忙等是指线程在没有实际工作的情况下不断检查某个条件是否满足，而不进入休眠状态。这种行为会消耗 CPU 资源，导致系统性能下降。

从日志中可以看出，系统确实表现出了忙等的行为。这是因为多个线程在没有实际工作的情况下不断切换，导致 CPU 资源被浪费在无意义的任务上。这种行为通常是由于线程调度器没有正确处理定时器中断或没有正确让线程进入休眠状态，从而导致线程不断检查条件而不释放 CPU 资源。

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`

2.1. 在 thread.h 中加入：线程状态：THREAD_SLEEP；线程结构体属性：wake_time

```
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     THREAD_SLEEP
16 };
```

```
102  /* Owned by thread.c. */
103  unsigned magic;          /**< Detects stack overflow.
104  */
104  int64_t wake_time;
105  };
106
```

2.2. `thread_yield` 不能再使用，为了让当前线程放弃 `cpu`，进入 `sleep` 状态（Sleep 概念：当前线程主动将 `CPU` 执行权释放，然后经过一段时间（ticks）后，倒计时到期，系统将该线程唤醒，重新加入到队列中等待调度），我们选择用 `thread_sleep()` 来替代 `thread_yield()`，做到将当前线程设置为 `sleep` 状态，并设置好线程应该醒来的时间。并在 `thread.h` 中加入对应信息。

```

87  /** Sleeps for approximately TICKS timer ticks.  Interrupts must
88      | be turned on. */
89  void
90  timer_sleep (int64_t ticks)
91  {
92      // int64_t start = timer_ticks ();
93
94      // ASSERT (intr_get_level () == INTR_ON);
95      // while (timer_elapsed (start) < ticks)
96      //     thread_yield ();
97      thread_sleep(ticks);
98  }
99

```

```

597  /* Replace the timer_sleep() in timer.c
598     Set the wake time, status of the cur_thread,
599     and make it fall asleep. zzw231113
600     */
601
602  void thread_sleep (int64_t ticks) {
603      if (ticks <= 0) return;
604      struct thread *cur=thread_current () ;
605
606      enum intr_level old_level=intr_disable();
607      // critical section
608      if (cur != idle_thread) {
609          cur->status=THREAD_SLEEP;
610          cur->wake_time=timer_ticks()+ticks;
611          schedule() ;
612      }
613      // critical section end
614      intr_set_level(old_level);
615  }
616

```

```

145  void thread_sleep (int64_t);
146  void check_and_wakeup_sleep_thread(void);
147  #endif /**< threads/thread.h */

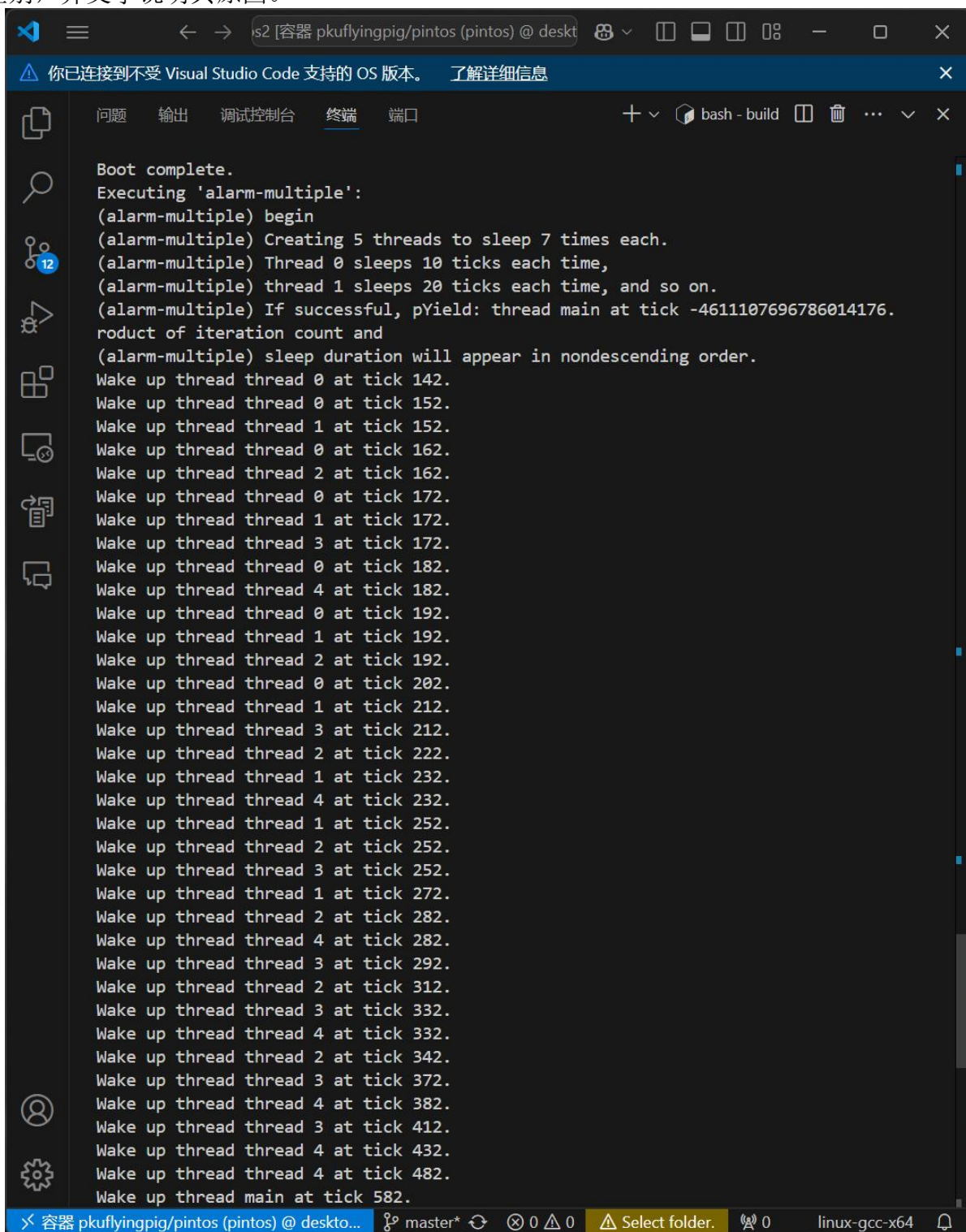
```

2.3. 设置了应唤醒时间，还需唤醒线程。timer_interrupt(), 现在的功能就是更新时间，在该处理程序里面加入对倒计时时间的更新，添加函数 check_and_wakeup_sleep_thread(): 遍历所有线程 (all_list)。如果处于 SLEEP 状态的线程可以被唤醒，那么把它加入 ready_list，状态改为 READY，并 print 信息。

```
170  /** Timer interrupt handler. */
171  static void
172  timer_interrupt (struct intr_frame *args UNUSED)
173  {
174      ticks++;
175      thread_tick ();
176      check_and_wakeup_sleep_thread();
177  }
```

```
618  /* Check all threads, if a sleep thread can wake up, then wake it up
619     and put it into ready list . zzw231113
620  */
621  void check_and_wakeup_sleep_thread(void) {
622      struct list_elem *e=list_begin(&all_list); // list for all threads
623      int64_t cur_ticks=timer_ticks();
624
625      while(e != list_end(&all_list)){
626          struct thread *t=list_entry(e, struct thread, allelem) ;
627          enum intr_level old_level=intr_disable();
628          if (t->status == THREAD_SLEEP && cur_ticks >= t->wake_time) {
629              // should wake
630              t->status=THREAD_READY;
631              list_insert_ordered (&ready_list, &t->elem, less_by_priority,
632                                  NULL); // priority insert by last lab
633              printf ("Wake up thread %s at tick %lld. \n", t->name,
634                      cur_ticks); // print info
635          }
636          e=list_next(e);
637          intr_set_level(old_level);
638      }
```


2.4. 运行指令 `pintos -v -- -q run alarm-multiple`，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。



```
Boot complete.
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
(alarm-multiple) If successful, pYield: thread main at tick -4611107696786014176.
product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
Wake up thread thread 0 at tick 142.
Wake up thread thread 0 at tick 152.
Wake up thread thread 1 at tick 152.
Wake up thread thread 0 at tick 162.
Wake up thread thread 2 at tick 162.
Wake up thread thread 0 at tick 172.
Wake up thread thread 1 at tick 172.
Wake up thread thread 3 at tick 172.
Wake up thread thread 0 at tick 182.
Wake up thread thread 4 at tick 182.
Wake up thread thread 0 at tick 192.
Wake up thread thread 1 at tick 192.
Wake up thread thread 2 at tick 192.
Wake up thread thread 0 at tick 202.
Wake up thread thread 1 at tick 212.
Wake up thread thread 3 at tick 212.
Wake up thread thread 2 at tick 222.
Wake up thread thread 1 at tick 232.
Wake up thread thread 4 at tick 232.
Wake up thread thread 1 at tick 252.
Wake up thread thread 2 at tick 252.
Wake up thread thread 3 at tick 252.
Wake up thread thread 1 at tick 272.
Wake up thread thread 2 at tick 282.
Wake up thread thread 4 at tick 282.
Wake up thread thread 3 at tick 292.
Wake up thread thread 2 at tick 312.
Wake up thread thread 3 at tick 332.
Wake up thread thread 4 at tick 332.
Wake up thread thread 2 at tick 342.
Wake up thread thread 3 at tick 372.
Wake up thread thread 4 at tick 382.
Wake up thread thread 3 at tick 412.
Wake up thread thread 4 at tick 432.
Wake up thread thread 4 at tick 482.
Wake up thread main at tick 582.
```

在实现休眠之前和之后，程序的执行结果有明显的不同。以下是对两个阶段的运行结果的详细分析及其原因说明：

- 实现休眠之前：

在实现休眠之前，所有线程的 Yield 操作记录的 tick 时间都是相同的，都是 -4611107696786014177、-4611107696786014176 等等。这表明在所有线程开始休眠之

前，系统的时间戳没有更新，所有线程都在同一时间点被调度或让出 CPU。

- 实现休眠之后：

在实现休眠之后，每个线程的 Wake up 操作记录的 tick 时间是不同的，并且按照线程休眠时间的不同逐渐增加。例如：

- 线程 0 每次唤醒的 tick 时间是 142, 152, 162, 172, 182, 192, 202，每次间隔 10 ticks。
- 线程 1 每次唤醒的 tick 时间是 152, 172, 192, 212, 232, 252, 272，每次间隔 20 ticks。
- 线程 2 每次唤醒的 tick 时间是 162, 192, 222, 252, 282, 312，每次间隔 30 ticks。

... ..

差别原因：

- 时间戳更新：在实现休眠之前，系统的时间戳没有更新，所有线程的时间都是相同的，这表明线程并没有真正进入休眠状态。而在实现休眠之后，每个线程的唤醒时间都按照预定的休眠时间递增，说明线程已经能够正确地进入休眠状态并按照预定的时间唤醒。

- 线程调度：在实现休眠之前，线程的调度可能没有按照预期的休眠时间进行，所有线程几乎同时被调度或让出 CPU。而在实现休眠之后，系统能够按照每个线程的休眠时间进行调度，确保线程在预定的时间被唤醒。

- 定时器功能：实现休眠之后，系统的定时器功能得到了正确的实现，能够为每个线程设置不同的时间间隔，并在到达指定时间后唤醒线程。

总结：

实现休眠之后，系统的定时器功能和线程调度功能得到了正确的实现，确保每个线程能够按照预定的休眠时间进行休眠并按时唤醒，这使得每个线程的唤醒时间按照预定的休眠时间递增。

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。

要实现苏醒后抢占功能，我们需要在唤醒线程时检查其优先级，并与当前正在运行的线程进行比较。如果唤醒的线程优先级更高，则应当触发抢占，切换到该高优先级线程执行。

实现思路：

1. 唤醒线程时检查优先级：在 `check_and_wakeup_sleep_thread` 函数中，当一个睡眠线程被唤醒并设置为 `THREAD_READY` 状态时，检查其优先级。
2. 比较当前运行线程的优先级：如果唤醒的线程优先级高于当前正在运行的线程优先级，则触发抢占。
3. 触发抢占：调用调度函数 `schedule()` 来切换到更高优先级的线程。

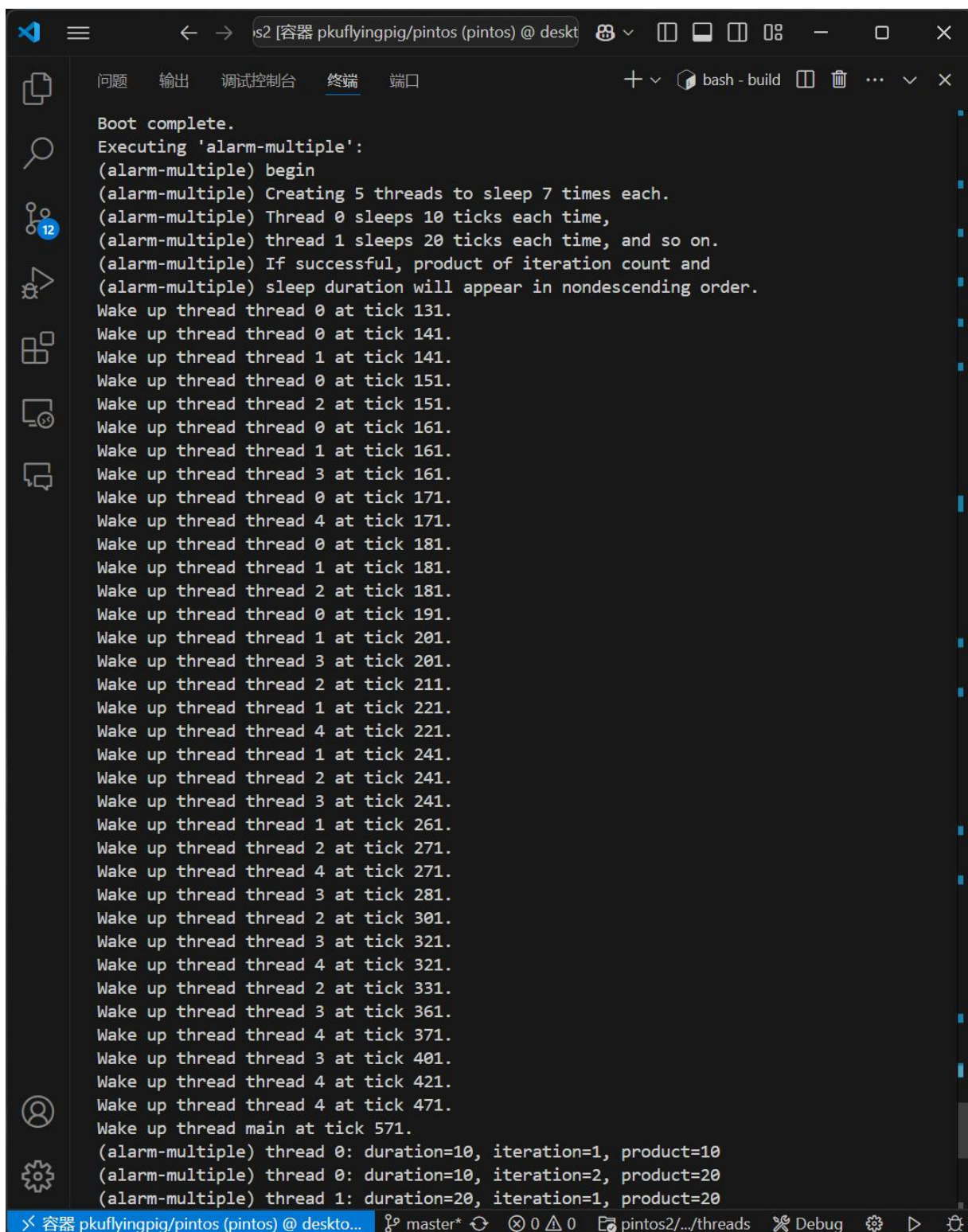
代码修改部分：

- 修改 `check_and_wakeup_sleep_thread` 函数：在唤醒线程后，检查其优先级并与当前运行线程的优先级进行比较。如果唤醒的线程优先级更高，则调用 `schedule()` 触发抢占。

总结：

通过在唤醒线程时检查其优先级，并在必要时调用 `schedule()` 函数，可以实现苏醒后抢占的功能。这样，系统能够动态调整运行线程，确保高优先级的线程能够及时获得执行机会，提升系统的响应性和效率。


```
618  /* Check all threads, if a sleep thread can wake up, then wake it up
    and put it into ready list . zzw231113
619
620  */
621  void check_and_wakeup_sleep_thread(void) {
622      struct list_elem *e = list_begin(&all_list); // list for all
    threads
623      int64_t cur_ticks = timer_ticks();
624
625      while (e != list_end(&all_list)) {
626          struct thread *t = list_entry(e, struct thread, allelem);
627          enum intr_level old_level = intr_disable();
628
629          if (t->status == THREAD_SLEEP && cur_ticks >= t->wake_time) {
630              // 唤醒线程
631              t->status = THREAD_READY;
632              list_insert_ordered(&ready_list, &t->elem, less_by_priority,
    NULL); // 按优先级插入
633              // list_push_back(&ready_list, &t->elem); // 将线程插入到
    ready_list的末尾
634
635              // 检查优先级并触发抢占
636              struct thread *cur_thread = thread_current();
637              if (cur_thread != idle_thread && t->priority >
    cur_thread->priority) {
638                  // 如果唤醒的线程优先级更高，触发抢占
639                  schedule();
640              }
641
642              printf("Wake up thread %s at tick %lld.\n", t->name,
    cur_ticks); // 打印信息
643          }
644          e = list_next(e);
645          intr_set_level(old_level);
646      }
647  }
```



```
Boot complete.
Executing 'alarm-multiple':
(alarm-multiple) begin
(alarm-multiple) Creating 5 threads to sleep 7 times each.
(alarm-multiple) Thread 0 sleeps 10 ticks each time,
(alarm-multiple) thread 1 sleeps 20 ticks each time, and so on.
(alarm-multiple) If successful, product of iteration count and
(alarm-multiple) sleep duration will appear in nondescending order.
Wake up thread thread 0 at tick 131.
Wake up thread thread 0 at tick 141.
Wake up thread thread 1 at tick 141.
Wake up thread thread 0 at tick 151.
Wake up thread thread 2 at tick 151.
Wake up thread thread 0 at tick 161.
Wake up thread thread 1 at tick 161.
Wake up thread thread 3 at tick 161.
Wake up thread thread 0 at tick 171.
Wake up thread thread 4 at tick 171.
Wake up thread thread 0 at tick 181.
Wake up thread thread 1 at tick 181.
Wake up thread thread 2 at tick 181.
Wake up thread thread 0 at tick 191.
Wake up thread thread 1 at tick 201.
Wake up thread thread 3 at tick 201.
Wake up thread thread 2 at tick 211.
Wake up thread thread 1 at tick 221.
Wake up thread thread 4 at tick 221.
Wake up thread thread 1 at tick 241.
Wake up thread thread 2 at tick 241.
Wake up thread thread 3 at tick 241.
Wake up thread thread 1 at tick 261.
Wake up thread thread 2 at tick 271.
Wake up thread thread 4 at tick 271.
Wake up thread thread 3 at tick 281.
Wake up thread thread 2 at tick 301.
Wake up thread thread 3 at tick 321.
Wake up thread thread 4 at tick 321.
Wake up thread thread 2 at tick 331.
Wake up thread thread 3 at tick 361.
Wake up thread thread 4 at tick 371.
Wake up thread thread 3 at tick 401.
Wake up thread thread 4 at tick 421.
Wake up thread thread 4 at tick 471.
Wake up thread main at tick 571.
(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
```

五、总结

本次实验深入研究了 Pintos 操作系统中的忙等待问题，并通过改进线程调度机制成功减少了 CPU 资源的无效占用，显著提升了系统的整体性能。以下是对实验过程和结果的总结：

1. 忙等待现象的分析与展示：

通过在 `timer_sleep()` 函数中观察到线程在休眠期间的频繁状态切换，我们明确了忙等待现象的存在。具体表现为线程在休眠期间不断从就绪状态到运行状态的反复切换，导致 CPU 资源的持续占用。通过在 `thread_yield()` 函数中添加打印语句，我们能够直观地展示这一现象，并通过运行 `pintos -v -- -q run alarm-multiple` 指令验证了忙等待的具体表现。

2. 实现线程的 `sleep` 功能：

为了消除忙等待，我们实现了线程的 `sleep` 功能，使线程在休眠期间主动放弃 CPU 执行权，并设定倒计时时间。具体实现包括修改线程状态、添加 `wake_time` 属性以及在 `timer_interrupt()` 中添加 `check_and_wakeup_sleep_thread()` 函数。通过这一改进，线程在休眠期间不再频繁切换状态，CPU 占用率显著降低，系统的整体性能得到提升。

3. 苏醒后优先级抢占的实现：

在实现线程 `sleep` 功能的基础上，我们进一步引入了苏醒后优先级抢占机制。通过在唤醒线程时检查其优先级，并与当前运行线程进行比较，如果唤醒线程的优先级高于当前运行线程，则触发抢占，确保高优先级线程能够及时获得执行机会。这一改进不仅减少了调度延迟，还提升了系统的响应性和效率。

4. 实验结果与总结：

通过对比实现 `sleep` 功能前后的运行结果，我们验证了改进的有效性。实现 `sleep` 功能后，线程的唤醒时间按照预定的时间递增，系统的时间戳更新准确，线程调度功能得到了正确实现。苏醒后优先级抢占机制的引入进一步确保了高优先级线程的及时执行，提升了系统的整体性能。

综上所述，本次实验通过深入分析忙等待问题，并采取针对性的改进措施，成功减少了 CPU 资源的无效占用，提升了 Pintos 操作系统的性能。实验结果表明，所采取的改进措施有效，达到了预期目标。

六、附录

GitHub 地址：[Redamancy-Xun/Operating-System-Practice](https://github.com/Redamancy-Xun/Operating-System-Practice): 操作系统实践 (github.com)

