实验报告：pintos 解决忙等待

课程名称：操作系统　　**年级：**2023 级　　**上机实践成绩：**

指导教师：张民　　**姓名：**张建夫

上机实践名称：pintos 解决忙等待　　**学号：**10235101477　　**上机实践日期：**11.4.2024

上机实践编号：　　**组号：**　　**上机实践时间：**14：50~16：30

一、目的

明白忙等待的产生原因以及解决办法

二、内容与设计思想

操作系统中的忙等待指的是线程在应该休眠的时候并不休眠（例如该线程试图获取一个锁，而锁已经被获取了时，这里的例子与实验中的实际忙等待有一定区别，实验的忙等待是指定了睡眠时间，尽管意思一样），而是仅仅将 cpu 让出，但是，由于该操作只是让出 cpu，该线程在后面仍有被调度的可能，一旦该线程又被调度，如果此时锁还未释放，该线程只会浪费 cpu 时间以及切换线程的时间，这就是忙等待，该线程浪费 cpu 时间却不做事（只会让出 cpu）。

要解决忙等待问题，需要在该线程要休眠时把该线程拿出就绪队列，这样操作系统就不会调用该线程，因此，对于目前的 pintos 系统，thread 状态还没有 thread_sleep 的状态，我们需要加入这个状态并添加一个睡眠队列（这样操作系统知道什么时候唤醒队列中的人，以及唤醒谁），之后在 timer_sleep 函数和 timer_interrupt 函数中加入相关代码（这里睡眠的具体实现还要封装相应函数）即可。ps：这里之所以要改变 timer_interrupt 是因为 timer_sleep 是指定了睡眠时间的，每一次硬件的的中断都代表时间过了 10ms，因此每次中断都要判断是否 sleep 足够时间了，而判断的代码就是在 timer_interrupt 函数中（实际上是在封装的函数里）。

三、使用环境

1. 主机 os：Windows-11
2. 虚拟机 os:Linux
3. 虚拟环境：docker 的 linux 镜像
4. 代码编辑器:vscode

四、实验过程

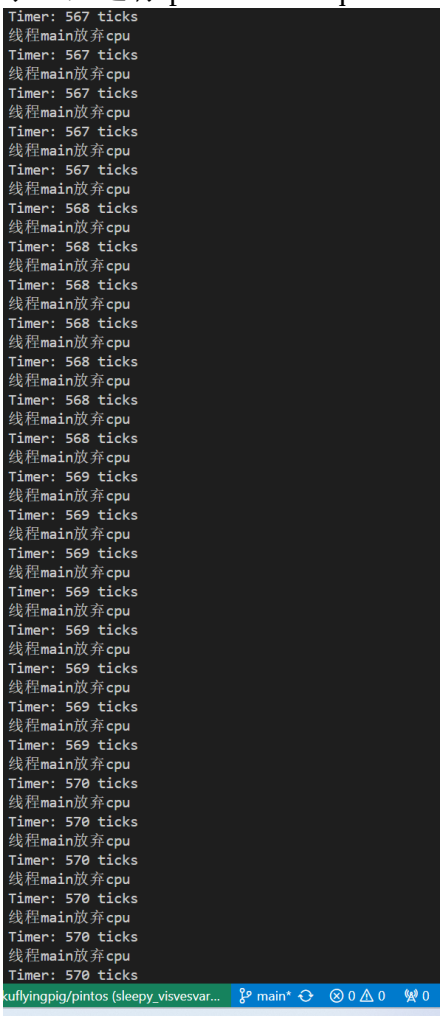
1.展示忙等：

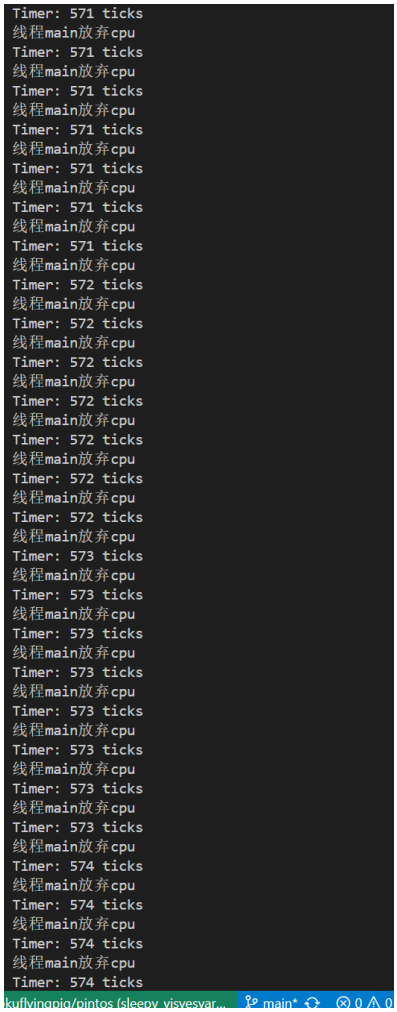
先展示一下忙等待的情况，我们在 thread_yield 函数里加了如下代码以展示忙等待：

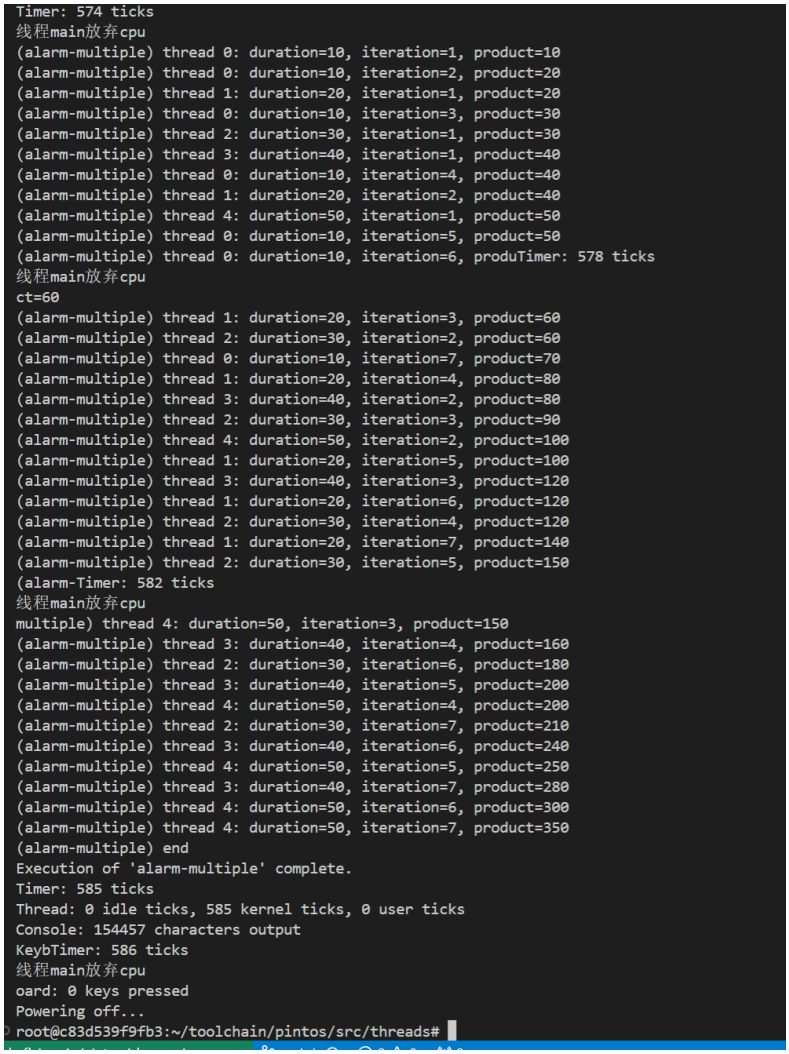
```c
/** Yields the CPU.  The current thread is not put to sleep and
   may be scheduled again immediately at the scheduler's whim. */
void
thread_yield (void)
{
  struct thread *cur = thread_current ();
  enum intr_level old_level;

  ASSERT (!intr_context ());
  old_level = intr_disable ();
  timer_print_stats ();
  printf("线程%s放弃cpu\n",cur->name);
  if (cur != idle_thread)   /*如果该线程不是idle空转线程*/
  if(!thread_mlfqs)
  priority_insert_threads(cur,&ready_list);/*调度线程时将线程加入就绪队列*/
  cur->status = THREAD_READY;
  schedule ();
  intr_set_level (old_level);
}
```

此处的 printf 语句和 timer_print_stats 函数（打印当前时间戳，是 timer.c 里面内置的函数，不是我自己写的）是为了满足实验和调试的要求（我 github 库里的则将这些 printf 加上注释了），运行 pintos -v -- -q run alarm-multiple：







可以从前面两张图看出来，该代码忙等待问题严重（上面直接将控制台的输出缓冲区占满了，pintos 初始化的部分都看不到了），线程 main 不断释放 cpu，又拿回 cpu，无效占用 cpu 时间，同时线程切换也是要时间的，这段时间也浪费了。

2.问题解决：

这个实验和 cs162 的解决办法一样，先修改 timer_sleep 函数：

```
87  /** Sleeps for approximately TICKS timer ticks.  Interrupts must
88     be turned on. */
89  void timer_sleep(int64_t ticks) {
90    thread_sleep(ticks);
91  }
92
```

以下是 thread_sleep 函数的具体实现：

```
521 /*让当前进程休眠ticks时间*/
522 void thread_sleep(int64_t ticks)
523 {
524   if(ticks<=0)return;
525   struct thread*cur=thread_current();
526 
527   enum intr_level old_level=intr_disable(); //关闭中断
528   if(cur!=idle_thread)
529   {
530     cur->status=THREAD_SLEEP;
531     cur->wake_time=timer_ticks()+ticks;
532     schedule();
533   }
534   intr_set_level(old_level);  //恢复中断
535 }
```

注意此处要关闭中断，因为此处要读取硬件中断的总次数，如果此时中断了，读取的中断次数就不一样了。

接着，改变 timer_interuppt 函数，实现唤醒进程的功能：

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

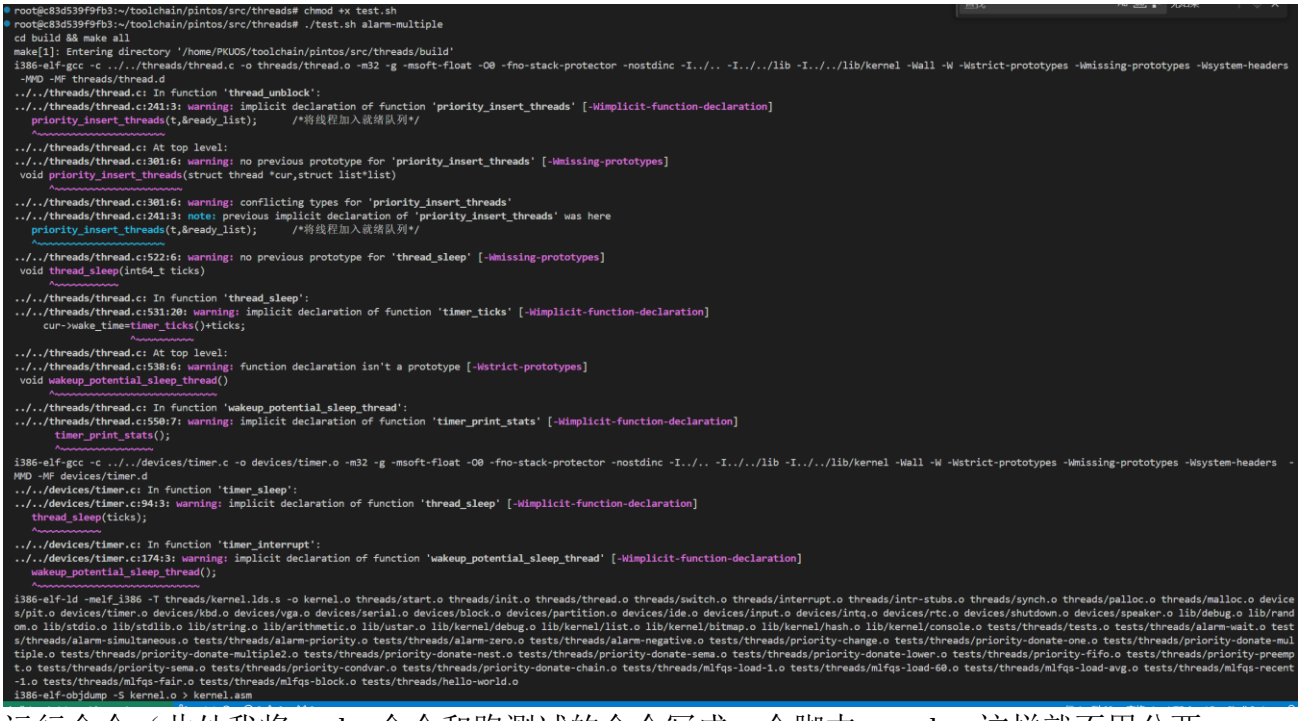
以下是 wakeup_potential_sleep_thread 函数的具体实现：

```
537 /*检查是否有休眠进程*/
538 void wakeup_potential_sleep_thread()
539 {
540   struct list_elem*e;
541   int64_t cur_time=timer_ticks();
542   for(e=list_begin(&all_list);e!=list_end(&all_list);e=list_next(e))
543   {
544     struct thread*tmp=list_entry(e,struct thread,allelem);
545     enum intr_level old_level=intr_disable(); //关闭中断
546     if(tmp->status==THREAD_SLEEP&&tmp->wake_time<=cur_time)
547     {
548       /*唤醒进程*/
549       tmp->status=THREAD_READY;
550       timer_print_stats();
551       printf("线程%s醒来\n",tmp->name);
552       if(!thread_mlfqs)
553       priority_insert_threads(tmp,&ready_list);
554     }
555     intr_set_level(old_level);//关闭中断
556   }
557 }
```

在执行唤醒线程的相关操作中，需要关闭中断是因为此处有线程加入到就绪队列的操作，该操作必须是原子的，否则在其他线程遍历就绪队列时会出现访问问题。

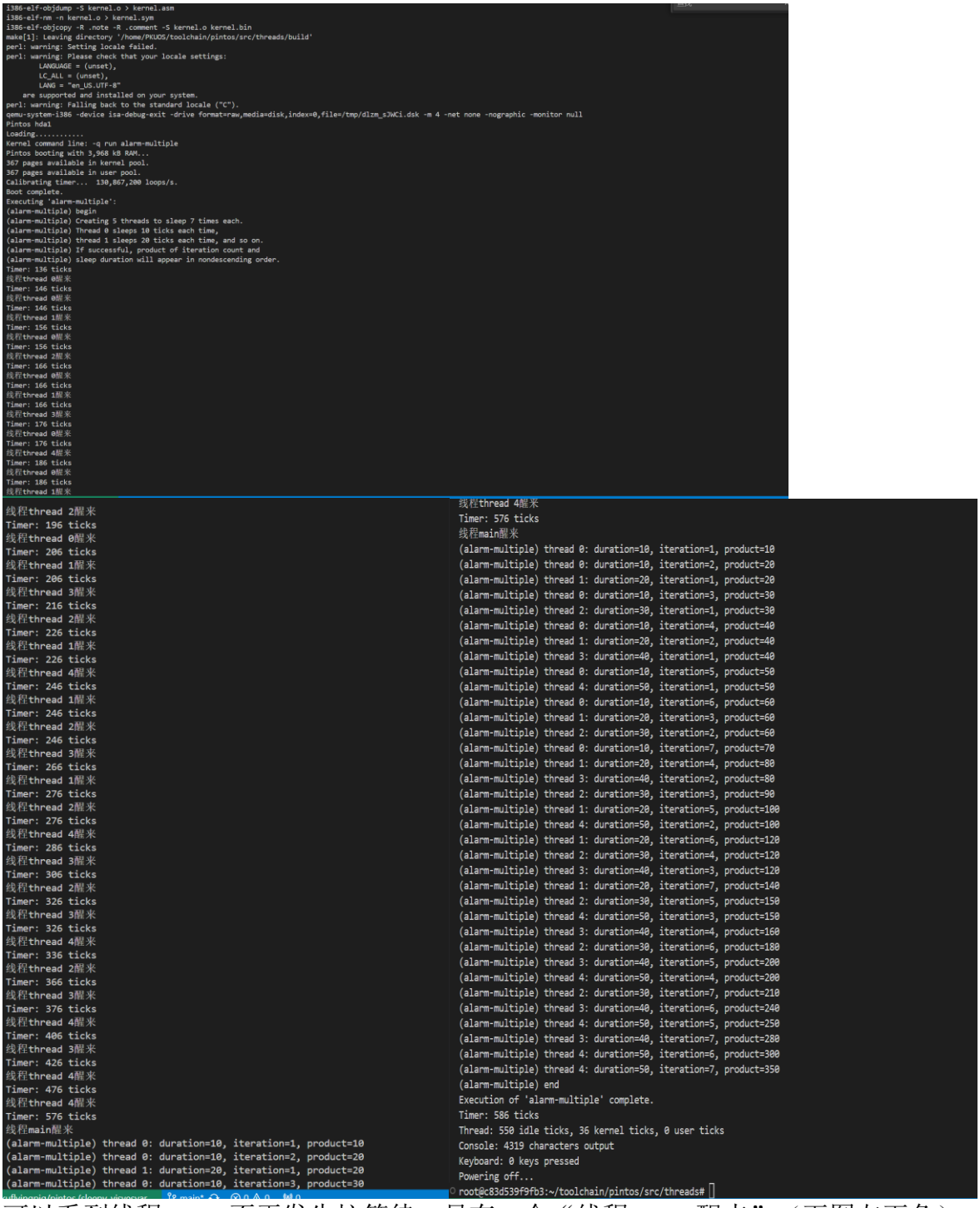
此处的 printf 语句和 timer_print_stats 函数作用和上面一样，是为了实验的调试要求，说明实现了忙等。此处有一个 if(!thread_mlfqs)，这个是方便后面实验多级队列实现的，暂且不谈。

实验结果：



运行命令（此处我将 make 命令和跑测试的命令写成一个脚本 test.sh，这样就不用分两

次敲了）



可以看到线程 main 不再发生忙等待，只有一个“线程 main 醒来”（下图左下角），而没有线程 main 放弃进程的提示了。

个人 github 的代码库：https://github.com/saydontgo/school_OS_course

五、总结

本次实验很好的展现了忙等待的危害（未修改代码前，main 线程不断地释放和持有进程，cpu 时间白白浪费），同时让我明白了线程休眠状态的具体实现（虽然之前做过 cs162 的 project，但实际上对于一些实现细节，如关闭中断，并没有完全明白，这次再做一次的过程中，对于所有的实现细节就都明白了）。