```
23      Pthread_create(&tid2, NULL, thread, &niters);
24      Pthread_join(tid1, NULL);
25      Pthread_join(tid2, NULL);
26
27      /* Check result */
28      if (cnt != (2 * niters))
29          printf("BOOM! cnt=%ld\n", cnt);
30      else
31          printf("OK cnt=%ld\n", cnt);
32      exit(0);
33  }
34
35  /* Thread routine */
36  void *thread(void *vargp)
37  {
38      long i, niters = *((long *)vargp);
39
40      for (i = 0; i < niters; i++)
41          cnt++;
42
43      return NULL;
44  }
```

code/conc/badcnt.c

图 12-16　（续）

因为每个线程都对计数器增加了 niters 次，我们预计它的最终值是 2×niters。这看上去简单而直接。然而，当在 Linux 系统上运行 badcnt.c 时，我们不仅得到错误的答案，而且每次得到的答案都还不相同！

```
linux>  ./badcnt 1000000
BOOM! cnt=1445085

linux>  ./badcnt 1000000
BOOM! cnt=1915220

linux>  ./badcnt 1000000
BOOM! cnt=1404746
```

那么哪里出错了呢？为了清晰地理解这个问题，我们需要研究计数器循环（第 40～41 行）的汇编代码，如图 12-17 所示。我们发现，将线程 i 的循环代码分解成五个部分是很有帮助的：

- H_i：在循环头部的指令块。
- L_i：加载共享变量 cnt 到累加寄存器%rdx$_i$的指令，这里%rdx$_i$表示线程 i 中的寄存器%rdx 的值。
- U_i：更新（增加）%rdx$_i$的指令。
- S_i：将%rdx$_i$的更新值存回到共享变量 cnt 的指令。
- T_i：循环尾部的指令块。

注意头和尾只操作本地栈变量，而 L_i、U_i和 S_i操作共享计数器变量的内容。

当 badcnt.c 中的两个对等线程在一个单处理器上并发运行时，机器指令以某种顺序一个接一个地完成。因此，每个并发执行定义了两个线程中的指令的某种全序（或者交叉）。不幸的是，这些顺序中的一些将会产生正确结果，但是其他的则不会。