Last commit : 2b70f41a35f0ce4b0c377172c3664b77d644ede7

1) علت غیرفعال کردن وقفه چیست؟ توابع ()pushcli و ()popcli به چه منظور استفاده شده و چه تفاوتی با cli و sti دارند؟

وقفه‌ها به این عات غیرفعال می‌شوند تا بتوانیم برخی کدهایی که می‌خواهیم (که دارای critical area هستند) را به صورت پیوسته اجرا کنیم تا مطمئن شویم با توجه ارتباط برخی متغیرها بین پردازه‌ها مشکلی در اجرای این نوع کدها به وجود نمی‌آید و به صورت atomicاجرا می‌شوند. برای غیرفعال کردن وقفه‌ها در ابتدا به کمک تابع pushcli، وقفه را غیرفعال می‌کنیم و پس از عبور از critical area و اجرای این بخش و همچنین باز کردن قفل (release)، تابع popcli فراخوانی می‌شود تا وقفه‌ها مجددا فعال شوند. definition توابع popcli و pushcli در فایل spinlock.c وجود دارد که در ادامه آورده شده است همچنین توابع cli و sti نیز در x86.h پیاده‌سازی شده‌اند که تصویر آن نیز در زیر قابل مشاهده است.

```
void
pushcli(void)
{
  int eflags;

  eflags = readeflags();
  cli();
  if(mycpu()->ncli == 0)
    mycpu()->intena = eflags & FL_IF;
  mycpu()->ncli += 1;
```

```
void
popcli(void)
{
  if(readeflags()&FL_IF)
    panic("popcli - interruptible");
  if(--mycpu()->ncli < 0)
    panic("popcli");
  if(mycpu()->ncli == 0 && mycpu()->intena)
    sti();
}
```

```
static inline void
sti(void)
{
  asm volatile("sti");
}
```

```
static inline void
cli(void)
{
  asm volatile("cli");
}
```

همانطور که می‌بینیم در توابع pushcli و popcli به ترتیب cli و sti فراخوانی شده است ولی تفاوت این توابع با cli و sti این است که در این توابع، قابلیت شمارش وجود دارد و مشخص است که هر برنامه چقدر اجرا شده است و در مدیریت آن کمک‌کننده است. در واقع می‌توان با استفده از pushcli به یک تعداد مشخص interrup ها را غیرفعال کنیم و (گویا به آن تعداد عدد در استک فرضی پوش شده است و باید برای فعال کردن interrup ها به تعداد همان مقدار عدد از استک فرضی پاپ کنیم.) اگر عدد

ncli از 0 بزرگتر باشد، interrupt ها غیر فعال و اگر برابر با 0 باشد، interrupt ها فعال است؛ در موقع استفاده از spinlock از pushcli استفاده می‌کنیم.

2) چرا قفل مذکور در سیستم‌های تک‌هسته‌ای مناسب نیست؟ روی کد توضیح دهید.

```
void
acquire(struct spinlock *lk)
{
  pushcli(); // disable interrupts to avoid deadlock.
  if(holding(lk))
    panic("acquire");

  // The xchg is atomic.
  while(xchg(&lk->locked, 1) != 0)
    ;

  // Tell the C compiler and the processor to not move loads or stores
  // past this point, to ensure that the critical section's memory
  // references happen after the lock is acquired.
  __sync_synchronize();

  // Record info about lock acquisition for debugging.
  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

تکه کد بالا definition تابع acquire را نشان می‌دهد که در آن تابع holding فراخوانی می‌شود در این تابع چک می‌شود که آیا cpu قفل را نگه داشته است یا خیر؛ یعنی در واقع پیداسازی قفل‌ها در سیستم‌عامل xv6 به صورت busy waiting است و اگر پردازنده تک هسته‌ای باشد، می‌توان یا مشغول انجام عملیات و یا نگهداری قفل باشد و این دو کار باهم ممکن نیست.

```
int
holding(struct spinlock *lock)
{
  int r;
  pushcli();
  r = lock->locked && lock->cpu == mycpu();
  popcli();
  return r;
}
```

3) مختصری راجع به تعامل میان پردازه‌ها توسط دو تابع مذکور توضیح دهید. چرا در مثال تولیدکننده/مصرف کننده استفاده از قفل‌های چرخشی ممکن نیست؟

```
void
acquiresleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  while (lk->locked) {
    sleep(lk, &lk->lk);
  }
  lk->locked = 1;
  lk->pid = myproc()->pid;
  release(&lk->lk);
}
```

```
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

همانطور که می‌بینیم آدرس قفل به تابع acquiresleep پاس داده می‌شود و پردازه تا زمانی که فرصت برای در دست گرفتن قفل به آن داده نشده است sleep می‌کند. در تابع releasesleep، پردازه‌ای که قفل را نگه داشته بود، تابع wakeup را فراخوانی می‌کند که در آن wakeup1 فراخوانی می‌شود و در این تابع، تمام پردازه‌هایی که روی آن قفل خاص، sleep کرده‌اند را بیدار می‌کند و در واقع وضعیت آنها را از SLEEPING به RUNNABLE تغییر می‌دهد. در مثال تولیدکننده/ مصرف کننده، اگر صرفا از spinlock استفاده کنیم، چون در صورت آزاد شدن قفل ،ممکن است وضعیت چندین پردازه به RUNNABLE تبدیل شود، نمی‌توان تضمین کرد که در صورت آزاد شدن قفل توسط مصرف کننده و خالی شدن بافر، قفل بلافاصله به تولیدکننده برسد.

```
//PAGEBREAK!
// Wake up all processes sleeping on chan.
// The ptable lock must be held.
static void
wakeup1(void *chan)
{
  struct proc *p;

  for (p = ptable.proc; p < &ptable.proc[NPROC]; p++)
    if (p->state == SLEEPING && p->chan == chan)
      p->state = RUNNABLE;
}

// Wake up all processes sleeping on chan.
void wakeup(void *chan)
{
  acquire(&ptable.lock);
  wakeup1(chan);
  release(&ptable.lock);
}
```

4) حالات مختلف پردازه‌ها در xv6 را توضیح دهید. تابع ()sched چه وظیفه‌ای دارد؟

تابع ()sched زمانی فراخوانی می‌شود که کار یک پردازه به پایان رسیده است و کار آن، ذخیره کاتکست فعلی و بازیابی کاتکست‌ها است که توسط فراخوانی تابع swch، عمل کانتکست سوییچ انجام می‌شود. تابع sched به در فراخوانی scheduler swch آن پردازه را نیز پاس می‌دهد. این تابع باید فقط قفل ptable را نگه دارد و intena را ذخیره و بازیابی کند زیرا آن ،یک property ریسه کرنل است و نه cpu.

حالات مختلف پردازه‌ها و توضیح آنها در ادامه آورده شده است.

```
// Enter scheduler.  Must hold only ptable.lock
// and have changed proc->state. Saves and restores
// intena because intena is a property of this
// kernel thread, not this CPU. It should
// be proc->intena and proc->ncli, but that would
// break in the few places where a lock is held but
// there's no process.
void sched(void)
{
  int intena;
  struct proc *p = myproc();

  if (!holding(&ptable.lock))
    panic("sched ptable.lock");
  if (mycpu()->ncli != 1)
    panic("sched locks");
  if (p->state == RUNNING)
    panic("sched running");
  if (readeflags() & FL_IF)
    panic("sched interruptible");
  intena = mycpu()->intena;
  swtch(&p->context, mycpu()->scheduler);
  mycpu()->intena = intena;
}
```

UNUSED: در این حالت از پردازه استفاده‌ای نمی‌شود.

EMBRYO: زمانی که پردازه‌ای از وضعیت UNUSED خارج می‌شود به وضعیت EMBRYO می‌رود.

SLEEPING: زمانی که پردازه به منبعی نیاز دارد که آماده نیست (مثلا عملیات I/O)، پردازه در حالت SLEEPING قرار می‌گیرد و دیگر در cpu نیست.

RUNNABLE: وقتی پردازه‌ای در این حالت قرار می‌گیرد یعنی آماده است و منتظر است تا scheduler، cpu را در اختیار آن قرار دهد.

RUNNNING: وقتی پردازه‌ای در این حالت قرار دارد یعنی در حال اجرا است و cpu در حال حاضر به آن اختصاص داده شده است.

ZOMBIE: زمانی که کار پردازه‌ای تمام می‌شود و پایان می‌یابد، به حالت ZOMBIE در می‌آید و تا زمانی که والدش wait را فراخوانی نکرده است در این حالت می‌ماند زیرا با وجود پایان این پردازه، اطلاعات آن هنوز در ptable وجود دارد.

5) تغییری در توابع دسته دوم داده تا تنها پردازه صاحب قفل، قادر به آزادسازی آن باشد. قفل معادل در هسته لینوکس را به طور مختصر معرفی نمایید.

کد مربوط به mutex در لینوکس در لینک زیر موجود است:

https://github.com/torvalds/linux/blob/master/include/linux/mutex.h

در mutex لینوکس، شرط اینکه چه پردازه‌ای درحال فراخوانی تابع است بررسی می‌شود و در آن یک owner برای چک کردن، تعریف شده است. با توضیح داده شده تنها پردازه صاحب قفل، قادر به آزادسازی آن است و busy waiting نیز وجود ندارد.

برای اینکه تنها پردازه صاحب قفا قادر به آزادسازی آن باشد، شرط زیر را در تابع releasesleep اضافه کردیم تا اگر شماره پردازه فعلی با شماره پردازه صاحب قفل یکی بود، بتواند آن را آزاد کند.

```
void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);
  if (lk->pid == myproc()->pid)
  {
    lk->locked = 0;
    lk->pid = 0;
    wakeup(lk);
  }
  release(&lk->lk);
}
```

6) یکی از روش‌های افزایش کارایی در بارهای کاری چندریسه‌ای استفاده از حافظه تراکنشی بوده که در کتاب نیز به آن اشاره شده است. به عنوان مثال این فناوری در پردازه‌های جدیدتر اینتل تحت عنوان افزونه‌های همگام‌سازی تراکنشی (TSX) پشتیبانی می‌شود. آن را مختصرا شرح داده و نقش حذف قفل را در آن بیان کنید.

حافظه تراکنشی دنباله‌ای از عملیات خواندن و نوشتن حافظه است که atomic هستند؛ اگر تمامی عملیات در یک تراکنش کامل شده باشند، حافظه تراکنشی انجام شده است. فواید حافظه تراکنشی توسط فیچرهایی که به زبان برنامه نویسی اضافه شده‌اند بدست می‌آید. زمانی که تعداد ریسه‌ها زیاد می‌شود، مکانیزم‌های همگام سازی نظیر قفل‌های mutex دچار مشکلاتی از قبیل بن بست می‌شوند. فایده استفاده از مکانیزم حافظه تراکنشی نسبت به قفل این است که در آن سیستم تراکنش حافظه و نه دولوپر، مسئول تضمین atomic بودن است و چون قفلی وجود ندارد بن بست اتفاق نمی‌افتد. همچنین این مکانیزم می‌تواند شناسایی کند که کدام جملات در atomic blocks می‌توانند به صورت concurrent اجرا شوند مثل concurrent access برای خواندن از متغیر مشترک. واضح است که این کارها و شناسایی این موقعیت‌ها برای یک برنامه‌نویس ممکن است ولی انجام اینکار زمانی که تعداد ریسه‌ها زیاد می‌شود بسیار مشکل است.

7) پیاده‌سازی ماکروی barrier() در لینوکس برای معماری x86 را فقط بنویسید.

```
/* The "volatile" is due to gcc bugs */
# define barrier() __asm__ __volatile__("": : :"memory")
```

پیاده‌سازی ذکر شده در لینک زیر وجود دارد.

https://github.com/torvalds/linux/blob/master/include/linux/compiler.h?plain=1#L84

8) آیا یک دستور مانع حافظه باید مانع بهینه‌سازی هم باشد؟ نام ماکروی پیاده‌سازی سه نوع مانع حافظه درلینوکس در معماری x86 را به همراه دستورالعمل‌های ماشین پیاده‌سازی آن ذکر کنید.

بله یک دستور مانع حافظه باید مانع بهینه‌سازی هم باشد چون کامپایلر برنامه‌ها را بهینه‌سازی می‌کند و ممکن است در این عملیات ترتیب دستورات را تغییر دهد و پردازه که تنها برنامه کامپایل شده را می‌بیند و از برنامه قبل از کامپایل خبر ندارد، دیگر به اینکه دستور درواقع مانع حافظه بوده است توجهی نمی‌کند.

سه نوع مانع حافظه در لینوکس در کد زیر آورده شده است و در ادامه به توضیح آن می‌پردازیم:

```
#define mb() asm volatile(ALTERNATIVE("lock; addl $0,-4(%%esp)", "mfence", \
                                      X86_FEATURE_XMM2) ::: "memory", "cc")
#define rmb() asm volatile(ALTERNATIVE("lock; addl $0,-4(%%esp)", "lfence", \
                                       X86_FEATURE_XMM2) ::: "memory", "cc")
#define wmb() asm volatile(ALTERNATIVE("lock; addl $0,-4(%%esp)", "sfence", \
                                       X86_FEATURE_XMM2) ::: "memory", "cc")
#else
#define __mb()  asm volatile("mfence":::"memory")
#define __rmb() asm volatile("lfence":::"memory")
#define __wmb() asm volatile("sfence" ::: "memory")
```

-**دستور mb**: از منتقل شدن دستورات read access و write access (به طور کلی دستورات memory access) به طرف دیگر مانع جلوگیری می‌کند و دستور ماشین پیاده‌سازی آن در mfence آورده شده است.

-**دستور rmb**: از منتقل شدن دستورات read access به طرف دیگر مانع جلوگیری می‌کند و دستور ماشین پیاده‌سازی آن در lfence آورده شده است.

-**دستور wmb**: از منتقل شدن دستورات write access به طرف دیگر مانع جلوگیری می‌کند و دستور ماشین پیاده‌سازی آن در sfence آورده شده است.

9) یک کاربرد از مانع در پردازش موازی ارائه دهید.

برای مثال در شبه کد Peterson's solution که در صفحه 263ز کتاب نیز آورده شده است، حط اول و دوم در حلقه باید به ترتیب اجرا شوند و توسط مانع می‌توان از صحت آن اطمینان پیدا کرد.

```
while (true) {
     flag[i] = true;
     turn = j;
     while (flag[j] && turn == j)
          ;

          /* critical section */

     flag[i] = false;

          /*remainder section */
}
```

پیاده‌سازی مساله 5 فیلسوف با استفاده از Condition Variable :

در ابتدا یک استراکت برای condition_var تعریف می‌کنیم که و یک آرایه‌ی 5 تایی از آن می‌سازیم که هر عضو آن متعلق به یک فیلسوف است و max_proc نشان‌دهنده‌ی ماکسیمم پردازه‌هایی است که می‌توانند باهم در ناحیه critical خود اجرا شوند که در این مساله برابر با 1 است و cur_proc نشان‌دهنده‌ی تعداد پردازه‌هایی است که اکنون یک چنگال خاص را در دست دارند (که در این مسله این تعداد نیز از 1 بیشتر نمی‌شود.) در queue پردازه‌هایی که خواستار یک چنگال هستند و برای آن در انتظارند، ذخیره می‌شوند و برای هر condition_var یک قفل از جنس spinlock در نظر گرفته شده است.

```
#define CONDITION_VAR_COUNT 5
static struct proc *initproc;
struct condition_var
{
  int max_proc;
  int cur_proc;
  int last;
  struct spinlock lock;
  struct proc* queue[NPROC];

} condition_vars[CONDITION_VAR_COUNT];
```

پیاده‌سازی sem_init:

در اینجا متغیرهای تعریف شده در استراکت condition_var را مقداردهی اولیه می‌کنیم و تعداد ماکسیمم پردازه‌هایی که می‌توانند در ناحیه بحرانی باشند و بقیه‌ی متغیرها (تعداد پردازه‌های در انتظار و تعداد پردازه‌های فعلی در ناحیه critical و اندیس آخرین پردازه‌ای که وارد صف شده) را برابر 0 قرار می‌دهیم.

```
void
sem_init(int i, int v)
{
  condition_vars[i].max_proc = v;
  condition_vars[i].last = 0;
  condition_vars[i].cur_proc = 0;

  for (int j = 0; j < NPROC; ++j)
    condition_vars[i].queue[j] = 0;

}
```

پیاده‌سازی sem_acquire:

این سیستم‌کال زمانی فراخوان می‌شود که یک پردازه موفق نشود تا به ناحیه بحرانی راه پیدا کند. در اینجا ابتدا روی قفل مشخص شده، acquire می‌کند و به تعداد پردازه‌هایی که خواستار ورود به ناحیه critical هستند اضافه می‌شود و پردازه فعلی به وارد صف انتظار آن ناحیه می‌کند و سپس روی همان قفل sleep می‌کند و پس از اتمام sleep، آن قفل را release می‌کند.

```
void
sem_acquire(int i)
{
  acquire(&condition_vars[i].lock);
  condition_vars[i].cur_proc++;
  condition_vars[i].queue[condition_vars[i].last] = myproc();
  condition_vars[i].last++;
  cprintf("Philosopher %d going to sleep\n", i+1);
  sleep(myproc(), &(condition_vars[i].lock));
  cprintf("Philosopher %d woke up\n", i+1);
  release(&condition_vars[i].lock);
}
```

پیاده‌سازی sem_release:

فراخوانی سیستمی sem_release زمانی فراخوانی می‌شود که پردازه‌ای موفق شده است قفل را در دست گیرد. در اینجا ابتدا قفل مشخص شده را accuire می‌کنیم و سپس با استفاده از یک شرط مطمئت می‌شویم که پردازه‌ای که اقدام به رها کردن قفل کرده است، آن را در دست داشته است؛ برای آزاد کردن قفل، پردازه‌ای که در ابتدای صف انتظار بوده است (یعنی مدت بیشتری در انتظار بوده است) را از آن خارج می‌کنیم و بقیه‌ی اعضای صف را یک واحد شیفت می‌دهیم و اندیس عضو آخر آن را نیز یک واحد کم می‌کنیم و سپس wakeup را فراخوانی می‌کنیم و قفل گرفته شده را release می‌کنیم.

```
void
sem_release(int i)
{
  acquire(&condition_vars[i].lock);

  if (condition_vars[i].cur_proc > 0 && condition_vars[i].cur_proc <= condition_vars[i].max_proc)
  {
    struct proc* p = condition_vars[i].queue[0];
    condition_vars[i].cur_proc--;
    for (int j = 1; j < condition_vars[i].last ; ++j)
      condition_vars[i].queue[j - 1] = condition_vars[i].queue[j];
    condition_vars[i].last--;
    wakeup(p);
  }
  release(&condition_vars[i].lock);
}
```

پیاده‌سازی تابع pickup:

ابتدا قفل را acquire می‌کنیم و وضعیت آن فیلسوف را به hungry تغییر می‌دهیم و سپس test را فراخوان می‌کنیم؛ درصورتی که فیلسوف موفق نشده بود تا چنگال را بردارد، sem_acquire را فراخوان می‌کنیم.

```
void pickup(int num)
{

    acquire(&statelock.lock);
    state[num] = HUNGRY;
    cprintf("Philosopher %d is Hungry\n", num + 1);
    test(num);
    release(&statelock.lock);
    if(state[num]!=EATING)
      sem_acquire(num);
}
```

پیاده‌سازی putdown:

در این تابع نیز در ابتدا قفل را acquire و در انتها release می‌کنیم؛ این تابع به این صورت عمل می‌کند که ابتدا وضعیت فیلسوف فعلی را به thinking تغییر می‌دهد و سپس یک بار برای چنگال سمت راست این فیلسوف و یک بار برای چنگال سمت چپ این فیلسوف، test را فراخوان می‌کنیم.

```
void putdown(int num)
{
    acquire(&statelock.lock);
    state[num] = THINKING;
    cprintf("Philosopher %d putting spoon %d and %d down\n",num + 1, (num)%5+1, (num+1)%5 + 1);
    cprintf("Philosopher %d is Thinking\n", num + 1);
    test((num+4)%5);
    test((num+1)%5);
    release(&statelock.lock);
}
```

در این پروژه علاوه بر تابع تست معمولی یک تابع deep_test نیز پیاده‌سازی شده است تا شرط bounding waiting رعایت شود و یک پردازه متقاضی چنگال به تعداد محدود برای بدست آوردن آن صبر کند.

پیاده‌سازی deep_test:

ابتدا یک آرایه 5تایی به نام phil_wai_counts می‌سازیم که در آن تعداد دفعاتی که هر پردازه در انتظار بوده و نوبت به آن نرسیده است نگهداری می‌شود و درصورتی که این مقدار به threshold تعیین شده (در اینجا عدد 3 تعیین شده) رسید و پردازه‌ی دیگری نیز درخواست آن چنگال را داشت ولی تعداد دفعات انتظارش کمتر بود، چنگال به پردازه فعلی می‌رسد.

```
int
deep_test(int num)
{
    int right_neigh = (num+1)%5;
    int left_neigh = (num+4)%5;
    int self_thresh = phil_wait_counts[num];
    if(state[right_neigh] == HUNGRY && phil_wait_counts[right_neigh] >= THRESHOLD && phil_wait_counts[right_neigh] > self_thresh) {
      cprintf("Philosopher %d going to sleep beacause of deep test for philosopher %d\n", num+1, right_neigh+1);
      return 0;
    }

    if(state[left_neigh] == HUNGRY && phil_wait_counts[left_neigh] >= THRESHOLD && phil_wait_counts[left_neigh] > self_thresh) {
      cprintf("Philosopher %d going to sleep beacause of deep test for philosopher %d\n", num+1, left_neigh+1);
      return 0;
    }
    return 1;
}
```

پیاده‌سازی تابع test:

در این تابع چک می‌شود درصورتی که همسایه چپ و راست فیلسوف درحال خوردن نباشند و خود فیلسوف فعلی

```
void test(int num)
{
    if (state[num] == HUNGRY && state[(num+4)%5] != EATING && state[(num+1)%5] != EATING && deep_test(num)) {
        state[num] = EATING;
        cprintf("Philosopher %d picking spoon %d and %d\n",num + 1, (num)%5+1, (num+1)%5 + 1);
        cprintf("Philosopher %d is Eating\n", num + 1);
        phil_wait_counts[num] = 0;
        sem_release(num);
    }
    else {
      phil_wait_counts[num]++;
      cprintf("Philosopher %d can't eat and wait_thresh: %d\n", num + 1, phil_wait_counts[num]);
    }
}
```

گرسنه باشد و deep_test، 1 را برگرداند، وضعیت فیلسوف فعلی را به درحال خوردن تغییر می‌دهیم و درغیر اینصورت یک واحد به تعداد دفعاتی که درحال انتظار بوده است اضافه می‌کنیم.

برنامه test_program:

در این برنامه 5 فرزند توسط fork تشکیل می‌دهیم و آنها را به برنامه philosophers، exec می‌کنیم و در انتها به تعداد پردازه‌های ایجاد شده wait می‌کنیم.

```c
int main(int argc, char *argv[])
{
    int id;
    char *in[]={"0"};
    for(int i=0;i<5;i++)
    {
        sem_init(i,1);
        in[0][0]=i+'0';
        id=fork();
        if(id==0)
            exec("philosophers",in);

        printf(1,"%d complete\n",i);
        sleep(10);
    }
    for(int i=0;i<5;i++)
        wait();
}
```

برنامه philosophers:

در یک حلقه پایان ناپذیر، به ازای هر فیلسوف، ابتدا درخواست pickup می‌کند و پردازه اول مدت بیشتری sleep می‌کند و پردازه‌های دیگر 30 میلی ثانیه sleep می‌کنند (تا بتوانیم حالتی که deep_test برای آن مطرح شد را امتحان کنیم) و سپس putdown را فراخوان می‌کنیم.

```c
int main(int argc, char *argv[])
{
    int i=argv[0][0]-'0';
    while (1)
    {
        sleep(20);
        pickup(i);
        printf(1, "philosopher %d in critical\n", i + 1);
        if (i==0)
            sleep(150);
        else
            sleep(30);
        putdown(i);
    }
}
```

نمونه‌ای از اجرای برنامه به همراه پیام‌هایی که متناسب با هرکار چاپ شده است در زیر آورده شده است.

```
$ test_program
0 complete
1 complete
2 complete
Philosopher 1 is Hungry
Philosopher 1 picking spoon 1 and 2
Philosopher 1 is Eating
philosopher 1 in critical
3 complete
Philosopher 2 is Hungry
Philosopher 2 can't eat and wait_thresh: 1
Philosopher 2 going to sleep
4 complete
Philosopher 3 is Hungry
Philosopher 3 picking spoon 3 and 4
Philosopher 3 is Eating
philosopher 3 in critical
Philosopher 4 is Hungry
Philosopher 4 can't eat and wait_thresh: 1
Philosopher 4 going to sleep
Philosopher 5 is Hungry
Philosopher 5 can't eat and wait_thresh: 1
Philosopher 5 going to sleep
Philosopher 3 putting spoon 3 and 4 down
Philosopher 3 is Thinking
Philosopher 2 can't eat and wait_thresh: 2
Philosopher 4 picking spoon 4 and 5
Philosopher 4 is Eating
Philosopher 4 woke up
philosopher 4 in critical
Philosopher 3 is Hungry
Philosopher 3 can't eat and wait_thresh: 1
Philosopher 3 going to sleep
Philosopher 4 putting spoon 4 and 5 down
Philosopher 4 is Thinking
Philosopher 3 picking spoon 3 and 4
Philosopher 3 is Eating
Philosopher 5 can't eat and wait_thresh: 2
Philosopher 3 woke up
philosopher 3 in critical
Philosopher 4 is Hungry
```