Repository Link: https://github.com/SamanEN/Operating-System-Lab-Projects
Latest Commit Hash: ad78c6a0daf7e87c9a3b06a05efbf9283395fa56

همگام‌سازی در xv6

1. علت غیرفعال شدن وقفه در حین اجرای ناحیه بحرانی، توضیح توابع pushcli و popcli و تفاوت آنها با cli و sti

تابع `cli` برای غیرفعال کردن وقفه‌ها و تابع `sti` برای فعال کردن وقفه‌ها استفاده می‌شود. توابع `pushcli` و `popcli` به ترتیب، به نوعی یک wrapper برای توابع `cli` و `sti` هستند با این تفاوت که می‌توانیم فرض کنیم یک stack مدیریت فعال کردن و یا غیرفعال کردن وقفه‌ها را به عهده می‌گیرد. تا زمانی که این استک خالی است، وقفه‌ها فعال‌اند و به محض اینکه با استفاده از تابع `pushcli` مقداری در استک push شود، وقفه‌ها غیرفعال می‌شوند. در واقع پیاده‌سازی توابع `pushcli` و `popcli` به این شکل است که تابع `pushcli` به ازای هربار فراخوانی، تابع `cli` را صدا می‌زند و وقفه‌ها را غیرفعال می‌کند اما تابع `popcli` تنها زمانی با استفاده از تابع `sti` وقفه‌ها را فعال می‌کند که استک کاملا خالی باشد. البته لازم به ذکر است که در واقعیت استکی وجود ندارد و فقط تعداد فراخوانی‌های هر یک از توابع در متغیری به نام `ncli` در هر پردازنده ذخیره می‌شود (به ازای فراخوانی تابع `pushcli`، مقدار این متغیر یک واحد افزایش پیدا می‌کند و به ازای فراخوانی تابع `popcli`، مقدار آن یک واحد کاهش می‌یابد) و زمانی که این متغیر برابر با 0 شود، وقفه‌ها فعال می‌شوند و هر موقع مقدار این متغیر بیشتر از 0 شود، وقفه‌ها غیرفعال می‌شوند. پس در واقع 2 بار فراخوانی تابع `pushcli`، نیازمند 2 بار فراخوانی تابع `popcli` برای فعال‌سازی مجدد وقفه‌ها است. کاربرد این توابع این است که اگر برای مثال به طور همزمان از دو قفل استفاده می‌کردیم، آزاد کردن یکی از قفل‌ها سبب فعال شدن وقفه‌ها نشود و این مورد فقط زمانی انجام شود که هر دو قفل آزاد شده باشند.

2. چرا Spinlock در سیستم‌های تک‌هسته‌ای مناسب نیست؟

در ابتدا نیاز به توضیح مختصری راجع به نحوه عملکرد تابع `acquire` می‌باشد؛ کد این تابع به شرح زیر است (کامنت‌ها در اینجا حذف شده‌اند):

```
void
acquire(struct spinlock *lk)
{
  pushcli();
  if(holding(lk))
    panic("acquire");

  while(xchg(&lk->locked, 1) != 0)
    ;

  __sync_synchronize();

  lk->cpu = mycpu();
  getcallerpcs(&lk, lk->pcs);
}
```

در این تکه کد، ابتدا تمام وقفه‌ها توسط تابع `pushcli` غیرفعال می‌شوند؛ سپس در یک حلقه `while` دستور اتمی `xchg` اجرا می‌شود. این دستور به صورت اتمی، محتوای یک خانه از حافظه را با یک رجیستر جابجا می‌کند. تابع `acquire` بصورت مداوم مقدار `lk->locked` را توسط دستور `xchg` یک می‌کند؛ در صورتی که قفل از قبل توسط پردازه‌ای دیگر نگه داشته شده باشد، مقدار آن از قبل یک بوده و با اجرای این دستور تفاوتی ایجاد نمی‌شود و تابع `xchg` مقدار یک را برمی‌گرداند؛ اما اگر مقدار `lk->locked` صفر باشد، این دستور مقدارش را یک کرده (که به معنای درگیر بودن قفل است) و تابع `xchg` مقدار صفر را بازمی‌گرداند. در این صورت تابع `acquire` از حلقه خارج شده و پردازه ادامه روند اجرایش را از سر می‌گیرد.
واضح است که این روش باعث انتظار مشغول[1] می‌شود. انتظار مشغول در سیستم‌های چند پردازنده‌ای باعث هدر رفتن زمان پردازنده و در نتیجه افت بهینگی سیستم شود؛ اما در سیستم‌های تک پردازنده‌ای این موضوع در بدترین حالت می‌تواند منجر به deadlock شود. حالتی را فرض کنید که در آن یک پردازه قفلی را در اختیار می‌گیرد؛ سپس پردازه‌ای دیگر سعی می‌کند قفل را به روش مذکور بدست بیاورد؛ در این صورت پردازه دوم هیچگاه از حلقه خارج نشده و پردازه‌های دیگر زمان‌بندی نمی‌شوند.

3. مختصری راجع به تعامل میان پردازه‌ها توسط توابع Sleeplock توضیح دهید. چرا استفاده از Spinlock در مثال producer-consumer ممکن نیست؟

یک sleeplock در xv6 از اجزای زیر تشکیل شده است:

```
struct sleeplock {
  uint locked;
  struct spinlock lk;
  char *name;
  int pid;
};
```

متغیر locked وضعیت قفل بودن را نشان داده و از یک spinlock برای حفاظت از کل اجزای استراکت استفاده می‌شود.
وقتی یک پردازه `acquiresleep` می‌کند، طبق محتوای تابع:

```
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);
}
```

ابتدا spinlock گرفته شده و سپس بررسی می‌کند که آیا sleeplock در دست پردازه دیگریست یا خیر.
در صورت آزاد بودن لاک، خود پردازه آن را گرفته و در غیر این صورت، پردازه sleep می‌شود.

[1] Busy waiting

نسخه ساده شده تابع sleep:

```
void
sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();
  if(p == 0 || lk ==0)
    panic("...");

  release(lk);
  p->chan = chan;
  p->state = SLEEPING;

  sched();

  p->chan = 0;
  acquire(lk);
}
```

رفتار این تابع به این شکل است که از یک spinlock (که اینجا همان spinlock داخل sleeplock است) که از قبل گرفته شده را دریافت کرده و قبل از تغییر وضعیت پردازه به SLEEPING، آن را release می‌کند. بعداً پس از wakeup شدن و بازگشت به این بخش از طرف scheduler، لاک دوباره acquire می‌شود.

این تابع یک آرگومان void* chan نیز دریافت می‌کند. این متغیر در struct proc هم وجود داشته و در sleep آن را به ورودی تغییر می‌دهد. اینجا خود sleeplock به عنوان chan پاس داده شده است.

حال که پردازه در حالت اجرا شونده نیست، پردازه‌ای که قفل را داشت releasesleep می‌کند:

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

وقتی که پردازه قبلی sleep کرد، لاک release شد پس اینجا می‌تواند آن را acquire کند. پردازه قفل را رها کرده و wakeup را روی خود متغیر sleeplock صدا می‌زند.

تابع wakeup یک void* chan را دریافت کرده و با پیمایش روی همه پردازه‌ها، وضعیت آنهایی که روی chan خوابیده اند را به RUNNABLE تغییر می‌دهد تا بعدا توسط scheduler قابلیت انتخاب شدن برای اجرا داشته باشند.

از آنجا که پردازه‌ای که در انتظار دریافت sleeplock روی خود لاک sleep کرده است، اینجا هم با صدا زدن wakeup روی خود لاک بیدار می‌شود.

دلیل اینکه در acquiresleep خود sleep کردن در یک چرخه است، برای این است که اگر به دلیلی wakeup شد ولی هنوز لاک قفل بود، به sleep بازگردد تا releasesleep صدا شود.

در مثال تولیدکننده/مصرف‌کننده، استفاده از قفل‌های چرخش ممکن است. یک راه حل می‌تواند به صورت زیر باشد (spinlock به صورت اتمی پیاده‌سازی شده است):

```
// Producer
void produce(/* arguments and the spinlock */) {
    while (true) {
        acquire_lock(spinlock);
        if (can_push()) break;
        release_lock(spinlock);

        while (!can_push());
    }
    // Push item to the buffer
    release_lock(spinlock);
}

// Consumer
void consume(/* arguments and the spinlock */) {
    while (true) {
        acquire_lock(spinlock);
        if (can_pull()) break;
        release_lock(spinlock);

        while (!can_pull());
    }
    // Pull item from the buffer
    release_lock(spinlock);
}
```

البته دقت شود در مسئله تولیدکننده/مصرف‌کننده، به دلیل نابهینه بودن قفل‌های چرخشی برای انتظار طولانی‌مدت، بهتر است از شیوه‌های دیگر مانند سمافور استفاده کنیم.

4. توضیح حالات مختلف پردازه‌ها در xv6 و وظیفه تابع sched()

حالت یک پردازه در متغیر state آن که از جنس `enum procstate` است نگه داشته می‌شود:

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

- UNUSED: از آنجا که پردازه‌ها در یک لیست 64 تایی نگه داشته می‌شوند، خانه‌هایی که درشان پردازه حقیقی‌ای نیست با این state مشخص شده‌اند و به معنی نبود پردازه است.
- EMBRYO: وقتی که پردازه جدیدی ساخته می‌شود (مثلا با `fork`) در ابتدا حالت پردازه این است. یعنی تابع `allocproc` از بین پردازه‌های `UNUSED` یکی را انتخاب و آن را `EMBRYO` می‌کند.
- SLEEPING: در این وضعیت، پردازه در بین انتخاب‌های scheduler برای تخصیص پردازنده به آن قرار نمی‌گیرد و بدون هیچ فعالیتی می‌ماند. پردازه می‌تواند به صورت داوطلبانه یا توسط کرنل به این حالت برود و در انتظار دسترسی به یک منبع بماند.
- RUNNABLE: وقتی پردازه در این حالت است، یعنی در صف اجرای scheduler قرار دارد و در یکی از راندهای زمان‌بندی بعدی CPU به آن می‌رسد و `RUNNING` می‌شود. چند حالت که منجر به ورود به این وضعیت می‌شوند:
    - پردازه تازه تشکیل شده و از `EMBRYO` به `RUNNABLE` می‌آید.

- پردازه در حال اجرا و RUNNING بوده و با اتمام time slice یا yield توسط کرنل پردازنده از آن گرفته می‌شود.
- پردازه در SLEEPING بوده و با wakeup قابل اجرا می‌شود.
- پردازه‌ای که SLEEPING بوده kill شده و پس از 1 کردن فیلد killed آن پردازه، به حالت RUNNABLE می‌آید تا وقتی که دوباره اجرا شد، همان اول با توجه به کشته شدن، exit شود.

- RUNNING: این حالت یعنی پردازه درحال اجرا توسط پردازنده است. تعداد پردازه‌های RUNNING در یک زمان حداکثر معادل تعداد پردازنده‌ها است.
- ZOMBIE: وقتی پردازه کارش تمام می‌شود و می‌خواهد exit بکند، ابتدا ZOMBIE می‌شود. یعنی مستقیم به UNUSED نمی‌رود و در حالتی می‌ماند که پدرش بتواند با استفاده از تابع wait از اتمام کار فرزندش باخبر شود.

تابع sched() برای context switch کردن به context زمان‌بند است. پردازه برای رها کردن CPU به این تابع می‌آید (که از قبل باید stateاش از RUNNING عوض شده باشد و قفل ptable را داشته باشد). در تابع فلگ interrupt enable ذخیره شده و پس از بازگشت برگردانده می‌شود.
این تابع با استفاده از swtch که در آزمایش سوم کامل توضیح داده شده و در اسمبلی نوشته شده، context را تغییر می‌دهد و ادامه تابع scheduler اجرا می‌شود که به context پردازه RUNNABLE دیگری تعویض می‌کند.

5. تغییری در توابع Sleeplock بدهید تا تنها پردازه صاحب قفل، قادر به آزادسازی آن باشد و قفل معادل در هسته لینوکس را به طور مختصر معرفی کنید

در استراکت sleeplock متغیر pid قرار داده شده که هدف اصلی ایجاد آن برای دیباگ کردن بوده است. از همین متغیر برای چک کردن صاحب قفل در releasesleep استفاده می‌کنیم. تکه کد زیر، تابع تغییر یافته است که قابلیت خواسته شده در آن ایجاد شده است:

```
void
releasesleep(struct sleeplock *lk)
{
  acquire(&lk->lk);

  if (lk->pid != myproc()->pid) {
    release(&lk->lk);
    return;
  }

  lk->locked = 0;
  lk->pid = 0;
  wakeup(lk);
  release(&lk->lk);
}
```

در هسته لینوکس از قفل mutex استفاده می‌شود. تعریف این قفل بصورت زیر می‌باشد (در فایل mutex.h موجود است):

```
    struct mutex {
        atomic_long_t owner;
        raw_spinlock_t wait_lock;
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
        struct optimistic_spin_queue osq; /* Spinner MCS lock */
#endif
        struct list_head wait_list;
#ifdef CONFIG_DEBUG_MUTEXES
        void* magic;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
        struct lockdep_map dep_map;
#endif
    };
```

همانطور که قابل مشاهده است، یک فیلد به نام `owner` برای این استراکت در نظر گرفته شده است؛ این فیلد در حین رها کردن قفل چک می‌شود تا تنها صاحب قفل مجاز به این کار باشد.

6. افزونه‌های همگام‌سازی تراکنشی (TSX) را شرح دهید و نقش حذف قفل را در آن بیان کنید

ایده transactional memory به این صورت است که تراکنش‌های حافظه داریم؛ هر تراکنش یک دنباله از خواندن و نوشتن‌ها در حافظه است که باید به صورت اتمی و پشت سر هم انجام بپذیرد.
در صورتی که تراکنش به صورت کامل انجام شود، تغییرات اعمال شده و در غیر این صورت، عملیات متوقف شده و اعمال نمی‌شوند.
در روش‌های عادی لاک کردن، وقتی که contention زیاد می‌شود مشکلاتی از جمله ددلاک و کند شدن به وجود می‌آید. افزایش تعداد پردازنده‌ها می‌تواند این contention را بیشتر هم بکند و برای همین به دنبال روش‌های دیگری برای هندل کردن critical section هستیم.
اگر در زبان برنامه‌نویسی بلاک کدی وجود داشته باشد که به کامپایلر بگوید که این بخش باید به صورت تراکنش انجام بپذیرد، می‌توان از مزایای این روش بهره‌مند شد؛ چون در این صورت، برنامه‌نویس دیگر درگیر لاک‌ها نمی‌شود و اتمی بودن عملیات در پشت صحنه هندل می‌شود.
حافظه تراکنشی در دو سطح نرم‌افزار و سخت‌افزار قابل پیاده شدن است. STM (Software TM) کاملا در نرم‌افزار بوده و کامپایلر کد را بررسی کرده و کدهای مورد نیاز را وارد می‌کند. HTM (Hardware TM) تدخلات داده‌های اشتراک گذاشته شده را در سمت سخت افزار حل می‌کند و مشکل کند بودن STM را ندارد.
TSX (Transactional Synchronization Extentions) یک افزونه برای ISA اینتل x86 است که ساپورت HTM را اضافه می‌کند. این HTM سرعت اجرای برنامه‌های مولتی‌ترد را با استفاده از lock elision بهبود می‌دهد. این افزونه باگ‌هایی داشته و در خیلی از پردازنده‌های اینتل که این قابلیت وجود داشته، غیر فعال شده است. به همین دلیل اکثر برنامه‌ها هنوز از TSX استفاده نمی‌کنند.
در این طراحی، با استفاده از اینستراکشن‌های جدید، سعی بر تراکنش می‌شود و اگر در زمان انجام نشد و abort شد، به لاک معمولی بر می‌گردد. به این کار lock elision می‌گویند.
یعنی با استفاده از این روش، ظاهر کد یکسان می‌ماند و لاک‌ها مثل یک wrapperاند که ابتدا سعی بر تراکنش می‌کنند و در صورت جواب ندادن به لاک عادی بر می‌گردند. و این یعنی لایبرری ارائه‌دهنده لاک‌ها باید تغییرات استفاده از TSX را اعمال کند.

مانع

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

این ماکرو در فایل linux/blob/master/include/linux/compiler.h به صورت زیر تعریف شده است:

```
#define barrier() __asm__ __volatile__("": : :"memory")
```

8. آیا یک دستور مانع حافظه باید مانع بهینه‌سازی هم باشد؟ نام ماکرو پیاده‌سازی سه نوع مانع حافظه در لینوکس برای معماری x86 را به همراه دستورهای ماشین پیاده‌سازی آنها ذکر کنید

بله یک مانع حافظه باید یک مانع بهینه‌سازی نیز باشد زیرا پردازنده فقط برنامه کامپایل شده را می‌بیند و ترتیب خطوط کدها پیش از بهینه‌سازی را نمی‌داند. برای مثال شبه کد زیر را در نظر بگیرید:

```
Statement1;
Statement2;
MemBarrier;
Statement3;
```

اگر فرض کنیم Statement2 و Statement3 به همدیگر وابسته نباشند، کامپایلر در هنگام بهینه‌سازی ممکن است ترتیب این 2 دستور را عوض کند. اما پردازنده به دلیل وجود مانع، تغییری در ترتیب دستورات نمی‌دهد و دستورات به ترتیب زیر اجرا می‌شوند:

```
Statement1;
Statement3;
Statement2;
```

در نتیجه ترتیب دستورات برخلاف خواسته توسعه‌دهنده، تغییر کرده است.

در لینوکس به طور کلی 3 دستور برای ایجاد مانع حافظه وجود دارد که در بخش زیر توضیح داده شده‌اند:

1. دستور rmb: این دستور از انتقال هرگونه دستور read access به سمت دیگر مانع، جلوگیری می‌کند. دستورالعمل ماشین آن در x86، lfence است که حرف l در آن، نشان‌دهنده کلمه load است.
2. دستور wmb: این دستور از انتقال هرگونه دستور write access به سمت دیگر مانع جلوگیری می‌کند. دستورالعمل ماشین آن در x86، sfence است که حرف s در آن، نشان‌دهنده کلمه store است.
3. دستور mb: این دستور از انتقال هرگونه دستور memory access به سمت دیگر مانع جلوگیری می‌کند (در واقع ترکیبی از 2 دستور قبلی است). دستورالعمل ماشین آن در x86، mfence است که حرف m در آن، نشان‌دهنده کلمه memory است.

برای یافتن دستورات مانع حافظه در لینوکس و دستورالعمل ماشین معادل آن‌ها در معماری x86، از فایل linux/arch/x86/include/asm/barrier.h استفاده شده است که کد آن در بخش زیر قابل مشاهده است:

```
#ifdef CONFIG_X86_32
#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")
#endif
```

در لینوکس توابع دیگری نیز برای ایجاد مانع وجود دارند. برای مثال ماکرو `barrier` یک مانع نرم‌افزاری (بهینه‌سازی) به وجود می‌آورد. ماکروهای `smp_rmb`، `smp_wmb` و `smp_mb` نیز در سیستم‌های چندهسته‌ای به ترتیب معادل دستورات 1 تا 3 هست و در سیستم‌های تک هسته‌ای معادل دستور `barrier` هستند.

9. کابرد مانع در پردازش موازی

یک کاربرد مانع در پردازش موازی می‌تواند در Peterson’s solution باشد. شبه کد این راه حل به صورت زیر می‌باشد:

```
int turn;
boolean flag[2];
while (true) {
      flag[i] = true;
      trun = j;
      while (flag[j] && turn == j);
      /* critical section */
      flag[i] = false;
      /* remainder section */
}
```

برای آنکه این الگوریتم درست کار کند، دو عبارت مقداردهی در بخش ورودی (دو خط اول پس از شروع حلقه اول) باید به ترتیب اجرا شوند. بنابراین با ایجاد یک مانع میان این دو عبارت، می‌توان از اجرای صحیح برنامه مطمئن شد.

یک مثال ساده دیگر می‌تواند برنامه زیر باشد:

```
// Core 1
while (f == 0);
// Memory barrier required here
print x;
```

```
// Core 2
x = 42;
// Memory barrier required here
f = 1;
```

در صورتی که مانع‌ها در بخش‌های مربوطه قرار نگیرند، ممکن است در هسته دوم مقدار f زودتر از x در حافظه قرار بگیرد و در نتیجه باعث اجرای غلط هسته اول شود. علاوه بر آن اگر در هسته اول نیز مانع قرار نگیرد، ممکن است ابتدا مقدار x و سپس مقدار f بارگذاری بشود. در این صورت مقدار ناصحیح و قدیمی x چاپ خواهد شد.

پیاده‌سازی Semaphore

پیاده‌سازی در فایل semaphore.h/c انجام شده و سپس سه سیستم‌کال اضافه شده‌اند.

```
struct semaphore {
  int value
  struct spinlock lk;
  struct proc* waiting[NPROC];
  struct proc* holding[NPROC];
  int wfirst;
  int wlast;
  char* name;
};
```

استراکت semaphore از چند فیلد تشکیل شده است:

- value: مقدار سمافور است که با عددی مثبت initialize شده و با هر بار گرفته شدن لاک، منهای 1 و با آزاد شدن لاک، به علاوه 1 می‌شود.
- lk: یک spinlock که از مقادیر استراکت محافظت می‌کند.
- waiting: یک صف دایره‌ای که پردازه‌هایی که در انتظار آزاد شدن سمافور و رفتن به critical sectionاند را نگه می‌دارد.
- holding: لیستی از پردازه‌هایی که سمافور را acquire کرده‌اند.
- wfirst wlast: دو اندیس برای پیاده‌سازی صف دایره‌ای waiting. wfirst به ابتدای صف اشاره می‌کند و با dequeue شدن یکی جلو می‌رود. wlast به انتهای صف اشاره کرده و با enqueue شدن وارد آن خانه می‌شود و یکی جلو می‌رود.
- name: این فیلد مانند بقیه لاک‌های xv6 قرار داده شده و به منظور قرار دادن اطلاعات برای دیباگ کردن است.

در semaphore.c توابع مورد نیاز برای کار با سمافورها پیاده‌سازی شده است:

```
void
semaphore_init(struct semaphore* sem, int value, char* name)
{
  sem->value = value;
  initlock(&sem->lk, "semaphore");
  memset(sem->waiting, 0, sizeof(sem->waiting));
  memset(sem->holding, 0, sizeof(sem->holding));
  sem->wfirst = 0;
  sem->wlast = 0;
  sem->name = name;
}
```

تابع `semaphore_init` یک سمافور و مقدار اولیه اش را گرفته و آن را initialize می‌کند. تابع `initlock` در spinlock.c تعریف شده و برای init کردن spinlock داخل سمافور استفاده شده است.

```
void
semaphore_acquire(struct semaphore* sem)
{
  acquire(&sem->lk);
  --sem->value;
  if(sem->value < 0){
    sem->waiting[sem->wlast] = myproc();
    sem->wlast = (sem->wlast + 1) % NELEM(sem->waiting);
    sleep(sem, &sem->lk);
  }
  struct proc* p = myproc();
  for(int i = 0; i < NELEM(sem->holding); ++i){
    if(sem->holding[i] == 0){
      sem->holding[i] = p;
      break;
    }
  }
  release(&sem->lk);
}
```

برای acquire کردن سمافور، ابتدا به دلیل لزوم تغییر مقادیر، spinlock استراکچر گرفته می‌شود. سپس value یک واحد کم شده و اگر حاصل منفی نبود، یعنی سمافور هنوز اجازه ورود پردازه به ناحیه بحرانی را می‌دهد. پس پردازه در اولین جای خالی در آرایه holding قرار می‌گیرد.
اگر حاصل value منفی شد، یعنی اجازه ورود ندارد و باید تا وقتی که پردازه‌ای سمافور را آزاد کند و نوبت به او برسد، sleep بماند. پس پردازه در خانه wlast صف waiting قرار گرفته و sleep می‌کند.

```
void
semaphore_release(struct semaphore* sem)
{
  acquire(&sem->lk);
  ++sem->value;
  if(sem->value <= 0){
    wakeupproc(sem->waiting[sem->wfirst]);
    sem->waiting[sem->wfirst] = 0;
    sem->wfirst = (sem->wfirst + 1) % NELEM(sem->waiting);
  }
  struct proc* p = myproc();
  for(int i = 0; i < NELEM(sem->holding); ++i){
    if(sem->holding[i] == p){
      sem->holding[i] = 0;
      break;
    }
  }
  release(&sem->lk);
}
```

برای release کردن سمافور، spinlock گرفته شده و value یک واحد اضافه می‌شود. اگر value کمتر مساوی 0 بود یعنی پردازه‌ای در انتظار گرفتن سمافور است و باید آن را wakeup کند. پس از این کار، خود را از لیست holding ها حذف می‌کند.

نحوه کار کردن sleep و wakeup در سوال سوم توضیح داده شد. سمافورها روی خودشان به عنوان chan می‌خوابند و اگر در release کردن روی سمافور wakeup زده شود، کل پردازه‌های sleeping بیدار می‌شوند درحالی که فقط می‌خواهیم سر صف را بیدار کنیم.

به همین دلیل یک تابع جدید در proc.c نوشته شد که پردازه خاصی را بیدار می‌کند و نیازی به chan ندارد:

```
void
wakeupproc(struct proc* p)
{
  acquire(&ptable.lock);
  p->state = RUNNABLE;
  release(&ptable.lock);
}
```

تابع `semaphore_holding` یک سمافور را گرفته و می‌گوید که آیا پردازه فعلی آن را گرفته است یا خیر. این کار با لیست holding سمافور که در توابع قبلی آپدیت می‌شد انجام می‌شود.

حال چون که xv6 ترد ندارد و همگام‌سازی در سطح پردازه‌ها انجام می‌شود، نمی‌توان یک instance از سمافور را در برنامه‌ای ساخت و از آن استفاده کرد. سمافور باید بین پردازه‌ها مشترک باشد و این یعنی در خود کرنل باید تعریف بشود.

پس یک آرایه 5 تایی از سمافورها در نظر گرفته شده که با استفاده از سیستم‌کال‌های `sem_init(i, v)` و `sem_acquire(i)` و `sem_release(i)` که i اندیس سمافور کرنل است، عملیات `semaphore_x`ها روی آنها انجام می‌شود.

```
struct semaphore sems[NSEMS];
```

شبیه‌سازی مسئله فلاسفه خورنده

برای پیاده‌سازی این مسئله از روش ذکر شده در کتاب استفاده شده است. در این روش، فیلسوف‌های با شماره زوج ابتدا چنگال سمت چپ خود و سپس چنگال سمت راست خود را برمی‌دارند. از طرفی، فیلسوف‌های با شماره فرد ابتدا چنگال سمت راست خود و سپس چنگال سمت چپ خود را برمی‌دارند. با این کار مشکل ددلاک حل می‌شود. هر کدام از 5 سمافور سیستمی در واقع نشان‌دهنده یک چنگال هستند که با مقدار اولیه 1 شروع به کار می‌کنند.

با توجه به نوع پیاده‌سازی سمافور (صف FIFO)، مشکل starvation نیز رخ نخواهد داد. مشکل دیگری که در حل این مسئله وجود داشت این است که تابع `printf` به صورت atomic اجرا نمی‌شود و در نتیجه چاپ log برنامه ممکن است باعث چاپ درهم شود. در نتیجه تعداد سمافورهای سیستم از 5 عدد به 6 عدد تبدیل شد تا سمافور ششم به عنوان mutex برای پرینت کردن استفاده شود. برای شبیه‌سازی بهتر نیز از تابع `random` استفاده شده که باعث می‌شود هر فیلسوف زمان رندومی را صرف خوردن غذا و یا تفکر کند. نمونه‌ای از خروجی مسئله در تصویر زیر قابل مشاهده است:

```
$ dining_philosophers
Philosopher 1 is going to pick up the right fork first
Philosopher 1 is going to pick up the left fork
Philosopher 0 is going to pick up the left fork first
Philosopher 3 is going to pick up the right fork first
Philosopher 4 is going to pick up the left fork first
Philosopher 2 is going to pick up the left fork first
Philosopher 1 has picked up both forks
Philosopher 0 is going to pick up the right fork
Philosopher 3 is going to pick up the left fork
Philosopher 1 will be eating for 274 ticks
Philosopher 3 has picked up both forks
Philosopher 3 will be eating for 764 ticks
Philosopher 1 has put down both forks
Philosopher 1 will be thinking for 395 ticks
Philosopher 0 has picked up both forks
Philosopher 2 is going to pick up the right fork
Philosopher 0 will be eating for 29 ticks
Philosopher 0 has put down both forks
Philosopher 0 will be thinking for 426 ticks
Philosopher 1 is going to pick up the right fork first
Philosopher 0 is going to pick up the left fork first
Philosopher 0 is going to pick up the right fork
Philosopher 0 has picked up both forks
Philosopher 0 will be eating for 659 ticks
Philosopher 3 has put down both forks
Philosopher 3 will be thinking for 333 ticks
Philosopher 2 has picked up both forks
Philosopher 4 is going to pick up the right fork
Philosopher 2 will be eating for 519 ticks
Philosopher 3 is going to pick up the right fork first
Philosopher 2 has put down both forks
Philosopher 2 will be thinking for 364 ticks
Philosopher 1 is going to pick up the left fork
Philosopher 0 has put down both forks
Philosopher 0 will be thinking for 0 ticks
Philosopher 0 is going to pick up the left fork first
Philosopher 4 has picked up both forks
```