آزمایشگاه سیستم عامل **پروژه 4**

سهیل حاجیان منش 810100119
امیرعلی رحیمی 810100146
مهدی نوری 810100231

github repo: https://github.com/SoheilHajianManesh/OS-Lab-Project4
last commit: e624f6915c96f404c489fe80fecdc005e02f5ba0

همگام سازی در xv6

۱. علت غیرفعال کردن وقفه ها در هنگام استفاده از قفل spinlock چیست؟ چرا ممکن است CPU با مشکل Deadlock مواجه شود؟

وقفه ها می توانند حتی در یک پردازنده باعث همزمانی[1] شوند: اگر وقفه ها فعال باشند، کد هسته را می توان هر لحظه متوقف کرد تا به جای آن یک کنترل کننده وقفه[2] اجرا شود.فرض کنید پردازنده قفل را نگه داشته باشد و در این حالت یک وقفه رخ دهد و وقفه سعی کند همان lock را قفل کند در این وضعیت، قفل هرگز آزاد[3] نخواهد شد چون فقط پردازه ای که قفل را در اختیار داشته میتواند آن را آزاد کند و آن پردازنده تا زمانی که کار وقفه تمام نشود نمی تواند به کار خود ادامه دهد بنابراین پردازنده و در نهایت کل سیستم به بن بست[4] می‌رسند.برای جلوگیری از این حالت آن پردازنده نباید قفل را در اختیار داشته باشد وقتی وقفه ای فعال است.در سیستم عامل xv6 به این منظور تمامی وقفه ها در آن

[1] concurrency
[2] interrupt handler
[3] released
[4] deadlock

پردازنده که میخواهد قفل را در اختیار بگیرد غیرفعال می گردد.البته در پردازنده های دیگر وقفه ها همچنان فعال هستند.

۲.توابع pushcli و popcli به چه منظور استفاده شده اند و چه تفاوتی با cli و sti دارند؟

توابع sti و cli به ترتیب برای غیرفعال کردن و فعال کردن وقفه ها استفاده میشوند.توابع pushcli و popcli هم به همین منظور استفاده میشوند به این ترتیب که مکانیزمی به مانند یک استک بوجود می آورند که برای سطوح تو در تو ی قفل ها کارایی دارند. داخل تابع pushcli تابع cli فراخوانی می شود که باعث میشود به ازای هر بار صدا زدن این تابع وقفه ها غیرفعال گردند البته اگر از قبل وقفه ها غیر فعال باشند منطقا با صدا زدن cli اتفاق خاصی رخ نمی دهد.سپس مقدار متغیر ncli برای پردازنده فعلی یکی زیاد میشود. در تابع popcli ابتدا مقدار ncli پردازنده ای که صدایش زده را یکی کم میکند و هر وقت این مقدار به صفر رسید تابع sti را صدا می کند که باعث میشود تا وقفه ها دوباره فعال شوند.پس در واقع به هر تعداد که از تابع pushcli استفاده کنیم به همان تعداد هم باید از تابع popcli استفاده کنیم تا وقفه ها دوباره فعال شوند. این کار کمک میکند تا اگر از چند قفل تو در تو استفاده کردیم و یکی از قفل ها آزاد شد وقفه ها فعال نگردند و تا آزاد شدن قفل آخر غیر فعال باشند.

۳. چرا Spinlock در سیستم ها تک هسته ای مناسب نیست؟

در سیستم عامل های چند هسته ای، عامل مشکل ساز استفاده از spinlock ها در بدترین حالت پایین آمدن efficiency است. اما در سیستم عامل تک هسته ای، استفاده از spinlock، میتوان موجب رخ دادن deadlock شود. برای مثال سناریو زیر را در نظر بگیرید:
پردازه A قفل را در اختیار میگیرد و در همین حین، پردازه ی B پردازه ی A را preempt میکند. حال پردازه ی B در حال اجرا است و این پردازه هم می خواهد همان قفلی که A در اختیار دارد را در اختیار بگیرد. پس سیستم کال گرفتن قفل را صدا میکند و وارد حلقه ای میشود که منتظر میماند قفل آزاد شود و به این پردازه اختصاص یابد. از آنجایی که در سطح کرنل، preemption رخ نمی‌دهد، این انتظار در حلقه پایان نمی یابد و به یک deadlock بر خورده‌ایم.

4. دلیل تعریف و نحوه کار دستور amoswap در RISCV

دستور amoswap که مختصر شده atomic memory allocation می باشد از مجموعه دستورات RISC-V است. تعریف این دستور به صورت زیر است:

amoswap rl, t1, (a0)

rl رجیستر مقصد است که با مقدار قدیمی موجود در آدرس a0 اپدیت میشود.
t1 رجیستر مبدا است که باید مقدار با مقدار موجود در آدرس a0 جابجا شود.
a0 آدرسی از حافظه است که باید مقدارش با t1 جابجا شود.

عملیات amoswap به صورت اتمی محتویات یک ثبات را با یک مکان حافظه تعویض می کند. این در محیط های چند هسته ای[5] یا چند رشته ای[6] که باید از شرایط مسابقه[7] جلوگیری کرد، مهم است. با استفاده از مبادله اتمی، سیستم تضمین می کند که هیچ پردازنده دیگری نمی تواند به مکان حافظه بین عملیات خواندن و نوشتن دسترسی داشته باشد.
استفاده از amoswap هنگام استفاده از قفل‌ها بسیار مهم است، زیرا باید اطمینان حاصل کنید که وقتی یک نخ در حال بررسی و تغییر وضعیت قفل است، هیچ نخ دیگری نمی‌تواند وارد شود و وضعیت را تغییر دهد تا زمانی که عملیات کامل شود.
به قطعه کد زیر که نمونه ای از حل مشکل انحصار متقابل[8] به کمک amoswap است دقت کنید:

```
    li          t0, 1       # Initialize swap value.
again:
    amoswap.w.aq t0, t0, (a0) # Attempt to acquire lock.
    bnez        t0, again   # Retry if held.
    # ...
    # Critical section.
    # ...
    amoswap.w.rl x0, x0, (a0) # Release lock by storing 0.
```

Figure 7.2: Sample code for mutual exclusion. a0 contains the address of the lock.

[5] multi-core
[6] multi-thread
[7] race-condition
[8] mutual exclusion

مقدار رجیستر t0 برابر ۱ شده است. سپس با دستور amoswap.w.aq مقدار قدیمی آدرس a0 که آدرس قفل است را در t0 می ریزیم. و مقدار ۱ را هم که به معنای بستن قفل است در آدرس a0 ریخته میشود.

حال در دستور بعدی اگر مقدار t0 برابر یک باشد یعنی قفل هنوز آزاد نشده است و دوباره به دستور قبلی پرش میکنیم و اگر t0 برابر صفر باشد یعنی قفل آزاد شده بوده است و در دستور قبلی ما آن را توانسته ایم در اختیار بگیریم و وارد ناحیه بحرانی[9] می شویم.

در حین خروج از ناحیه بحرانی هم با استفاده از دستور amoswap.w.rl مقدار رجیستر x0 که صفر است را در آدرس a0 حافظه میریزیم تا قفل آزاد شود.

۵. مختصری راجع به تعامل میان پردازه ها توسط دو تابع acquiresleep و releasesleep توضیح دهید.

استراکت sleeplock به صورت زیر تعریف شده است:

```
// Long-term locks for processes
struct sleeplock {
  uint locked;       // Is the lock held?
  struct spinlock lk; // spinlock protecting this sleep lock

  // For debugging:
  char *name;        // Name of lock.
  int pid;           // Process holding lock
};
```

متغیر locked برای این که وضعیت قفل را مشخص کند که قفل است یا آزاد.

یک spinlock برای خود قفل تعریف شده است که برای محافظت از متغیر های خود استراکت sleeplock در نظر گرفته شده است بخصوص متغیر locked.

متغیر name , pid بترتیب اسم قفل و آی دی پردازه ای که قفل را در اختیار دارد را نگه می دارند.

تابع aqcuiresleep :

[9] critical section

```
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 توسط پردازه گرفته می شود سپس چک میکند ببیند آیا قفل آزاد است یا خیر.
اگر آزاد باشد که آن را میگیرد و قفل میکند و spinlock را آزاد میکند و از تابع خارج می شود.
اما اگر قفل آزاد نباشد پردازه به حالت sleep باید در بیاید.

```
void sleep(void *chan, struct spinlock *lk)
{
  struct proc *p = myproc();

  if (p == 0)
    panic("sleep");

  if (lk == 0)
    panic("sleep without lk");
  if (lk != &ptable.lock)
  {
    acquire(&ptable.lock);
    release(lk);
  }
  p->chan = chan;
  p->state = SLEEPING;
  sched();
  p->chan = 0;
  if (lk != &ptable.lock)
  {
    release(&ptable.lock);
    acquire(lk);
  }
}
```

تابع sleep بصورت بالا تعریف شده است.

یکی از ورودی های این تابع یک spinlock است که ابتدا قبل از اینکه وضعیت پردازه را به SLEEPING تغییر دهیم آن را release میکنیم و بعد از اینکه پردازه توسط تابع wakeup بیدار شد و به بخش برگشت(با استفاده از scheduler) دوباره آن را acquire میکنیم. در اینجا همان spinlock مربوط به sleeplock را به عنوان این قفل پاس می دهیم چون در ابتدای کد aqcuiresleep آن را aqcuire کرده ایم و وقتی پردازه به حالت SLEEPING می رود باید قبل از آن این قفل آزاد شود تا پردازه های دیگر بتوانند آن را در اختیار بگیرند.

ورودی دیگر این تابع یک* void به نام chan است که در واقع مانند یک شاخص برای شناسایی عمل می کند و به پردازه می گوید روی چه کانالی باید sleep کند. در اینجا این مقدار را برابر خود خود قفل

قرار می دهیم. همچنین چون در حال تغییر ویژگی های پردازه هستیم در ابتدای این تابع قفل ptable را aqcuire می کنیم و در انتها آن را release می کنیم.

حال پردازه ای که قفل را در اختیار داشت وقتی کارش تمام شد تابع releasesleep را صدا می زند:

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

در این تابع به مانند تابع aqcuiresleep ابتدا spinlock توسط پردازه گرفته می شود .

سپس قفل را آزاد میکند و مقدار آی دی پردازه ای که قفل را در اختیار دارد را هم ریست میکند سپس تابع wakeup را فرا می خواند. این تابع بصورت زیر است:

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

این تابع هم با صدا زدن تابع wakeup1 تمام پردازه هایی که روی کانال chan در حالت SLEEPING قرار دارند را به حالت RUNNABLE تغییر می دهد تا دوباره توسط cpu زمانبندی[10] شوند در این تابع هم چون ویژگی های پردازه ها را نیاز است تغییر دهیم باید قفل ptable را در ابتدا aqcuire و سپس release کنیم.

به عنوان ورودی هم به این تابع خود قفل را می دهیم. کمی قبل تر ذکر شد که پردازه ها روی همین کانال به حالت SLEEPING درآمدند پس باید روی همین کانال هم wakeup شوند.

[10] scheduling

تابع wakeup1 هم مطابق زیر است که توضیح داده شد:

```
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;
}
```

6.حالات مختلف پردازه ها در xv6 را توضیح دهید.sched چه وظیفه ای دارد؟

پردازه ها در xv6 یکی از حالات زیر را می توانند داشته باشند:

UNUSED.1: از آنجایی که تمامی پردازه ها در یک جدول[11] با سایز ۶۴ نگه داری میشوند خانه هایی که در این جدول پردازه حقیقی ندارند در واقع شامل پردازه با حالت UNUSED می باشند.

EMBRYO.2:وقتی پردازه ای در ابتدا ساخته میشود اولین حالتی که به خود میگیرد این حالت است.تابع allocproc حالت پردازه را به EMBRYO تغییر می دهد و یک pid شاخص به پردازه اختصاص می دهد.

SLEEPING.3: وقتی پردازه ای در این حالت قرار می گیرد دیگر توسط scheduler زمانبندی نمی شود این حالت از هدر رفتن منابع cpu توسط پردازه ای که در انتظار مشغول است جلوگیری می کند.پردازه توسط کرنل یا با انتخاب خود می تواند به این حالت برود.

RUNNABLE.4:پردازه ای که تمام منابع مورد نیاز بجز یک پردازنده برای اجرا شدن را دارد و تنها منتظر scheduler است که یک پردازنده را به او اختصاص دهد در حالت RUNNABLE قرار می گیرد.

یک پردازه در وضعیت های مختلفی می تواند به وضعیت RUNNABLE برود.

- پردازه بعد از تشکیل اش از حالت EMBRYO به RUNNABLE تغییر یابد.
- پردازه در حالت RUNNING باشد و به دلیل تمام شدن زمان پردازنده اختصاص یافته به آن از این حالت خارج شده و دوباره به حالت RUNNABLE برود.

[11] ptable

- پردازه در حالت SLEEPING باشد و با فراخوانی wakeup از این حالت به RUNNABLE تغییر یابد.
- پردازهای که SLEEPING بوده kill شده و پس از 1 کردن فیلد killed آن پردازه، به حالت RUNNABLE می آید تا وقتی که دوباره اجرا شد، همان اول با توجه به کشته شدن، exit شود.

RUNNING.5: وقتی پردازه ای در این حالت قرار دارد یعنی توسط scheduler پردازنده ای به آن اختصاص یافته و در حال اجرا توسط پردازنده است.در هر زمان حداکثر به تعداد پردازنده ها می توانیم پردازه با حالت RUNNING داشته باشیم.

ZOMBIE:6: پردازه ای که کارش تمام شده است اما همچنان در جدول پردازه ها وجود دارد حالت ZOMBIE دارد. این حالت وقتی به وجود می آید که پردازه exit را صدا می زند اما پردازه پدرش wait را صدا نمی زند تا از اتمام کار پردازه فرزندش با خبر شود.

تابع ()sched برای تغییر متن از متن پردازه فعلی به متن زمانبند استفاده می شود.وقتی به هر دلیلی پردازه از حالت RUNNING خارج میشود این تابع فراخوانی می شود تا زمانبند پردازنده رها شده را به پردازه دیگری اختصاص دهد.نکته مهم این است که قبل از فراخوانی این تابع باید پردازه از حالت RUNNING خارج شده باشد و قفل ptable هم گرفته شده باشد وگرنه تابع panic میکند.

این تابع با استفاده از swtch که در آزمایش سوم کامل توضیح داده شده و در اسمبلی نوشته شده، متن را از متن پردازه فعلی به متن زمانبند تغییر می دهد و ادامه تابع scheduler اجرا می شود که به متن پردازه RUNNABLE دیگری تعویض می کند.

7. 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.

با توجه به عکسی که در ابتدای سوال پنج قرار داده شد از استراکت sleeplock متغیر pid که برابر آیدی پردازه ایست که قفل را در اختیار دارد در آن تعریف شده است. در xv6 این متغیر صرفا برای دیباگ کردن در تعریف شده است اما می توانیم از آن برای برطرف کردن مشکل گفته شده در این سوال استفاده کنیم.

کد releasesleep را به صورت زیر تغییر می دهیم:

```
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);
}
```

سه خط اضافه شده اینگونه است که اگر آی دی پردازه جاری با آی دی پردازه ای که قفل را در اختیار دارد برابر نبود بعد از release کردن spinlock از تابع خارج شود و نتواند قفل را release کند.

قفل معادل در هسته لینوکس mutex می باشد که در کتابخانه mutex.h تعریف شده است:

```
struct mutex {
        atomic_long_t           owner;
        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
};
```

متغیر owner به منظوری که گفته شد در نظر گرفته شده است که صاحب قفل را مشخص میکند اگر این متغیر مقدار null داشته باشد یعنی قفل متعلق به کسی نیست.

8. روشی دیگر برای نوشتن برنامه ها استفاده از الگوریتم lock-free است.مختصری راجع به آن ها توضیح داده و از مزایا و معایب آنها نسبت به برنامه نویسی با lock بگویید.

الگوریتم‌های بدون قفل از قفل یا mutex استفاده نمی‌کنند، بلکه به عملیات اتمی[12] یا دستورالعمل‌های مقایسه و تعویض[13] برای انجام به‌روزرسانی‌های همزمان فایل‌های اشتراک‌گذاری شده متکی هستند. عملیات اتمی تقسیم ناپذیر است و تضمین می کند که تنها یک فرآیند یا رشته می تواند یک فایل را در یک زمان، بدون دخالت دیگران، تغییر دهد. الگوریتم‌های بدون قفل می‌توانند عملکرد، مقیاس‌پذیری و پاسخ‌دهی را بهبود بخشند، اما چالش‌هایی نیز دارند. برای مثال، الگوریتم‌های بدون قفل می‌توانند باعث نشت حافظه،[14] تلاش مجدد یا پیچیدگی پیاده سازی شوند.

مزایا: الگوریتم های lock-free دارای چند مزیت نسبت به الگوریتم هایی که برمبنای قفل هستند دارند.

1. الگوریتم‌های بدون قفل می‌توانند عملکرد را بهبود بخشند، زیرا سربار(overhead) و مشاجره برای منابع مشترک را حذف می‌کنند.
2. الگوریتم‌های بدون قفل همچنین می‌توانند مقیاس‌پذیری[15] را بهبود بخشند، زیرا می‌توانند فرآیندها یا رشته‌های همزمان بیشتری را بدون مسدود کردن یا انتظار مدیریت کنند.
3. الگوریتم‌های بدون قفل همچنین می‌توانند پاسخگویی را بهبود بخشند، زیرا می‌توانند تضمین کنند که برخی از فرآیندها یا رشته‌ها همیشه پیشرفت کرده و به یک فایل دسترسی خواهند داشت.

معایب: الگوریتم های lock-free نسبت به الگوریتم هایی که مبتنی بر قفل در زمینه های مختلف دارای چند ایراد هستند.

1. پیاده‌سازی و اشکال‌زدایی الگوریتم‌های بدون قفل سخت‌تر است، زیرا به تکنیک‌های برنامه‌نویسی سطح پایین و پلتفرم خاص نیاز دارند.

---

[12] atomic operation

[13] compare-and-swap

[14] memory leaks

[15] scalability

2. الگوریتم‌های بدون قفل همچنین می‌توانند باعث نشت حافظه شوند، که زمانی رخ می‌دهد که یک فایل پس از به‌روزرسانی توسط یک فرآیند یا رشته، توسط سیستم آزاد یا بازیابی نشود.

پیاده سازی متغیر های مختص هر هسته پردازنده

الف)روشی جهت حل این مشکل در سطح سخت افزار وجود دارد. آن را توضیح دهید:

در حوزه علوم کامپیوتر به هماهنگی میان داده‌ها در حافظه‌های نهانی که همگی به منابع مشترکی وصل هستند، انسجام کش گفته می‌شود. حل این دغدغه حافظه در سطح سخت‌افزاری اغلب به کمک پروتکل‌هایی انجام می‌گیرد که برای حفظ انسجام کش طراحی شده‌اند.
ما در اینجا به بررسی سه نمونه از پروتکل‌های مطرح در این زمینه می‌پردازیم:

1.MESI (Modified, Exclusive, Shared, invalid):

پروتکل MESI با نظارت بر وضعیت هر سطر از کش، پایداری انسجام آن را تضمین می‌کند. هر سطر کش قادر است در یکی از چهار وضعیت: تغییر یافته (Modified)، انحصاری (Exclusive)، به اشتراک گذاشته شده (Shared) یا نامعتبر (Invalid) قرار گیرد. این پروتکل اطمینان می‌دهد که هر بلوک کش فقط یک بار به عنوان تغییر یافته یا انحصاری در نظر گرفته شود و در صورت نیاز، به‌روزرسانی‌ها به دیگر کش‌ها اعمال شود.

2.MOESI(Modified, Owned, Exclusive, Shared, Invalid):

پروتکل MOESI، با معرفی وضعیت "Owned" بهینه‌سازی‌های جدیدی را ارائه می‌دهد. وضعیت اختصاص یافته (Owned) به یک کش اجازه می‌دهد که نسخه‌ای از بلوک کش بدون تغییر دادن محتوا را حفظ کند، که این امکان می‌تواند در شرایط خاص کارایی را تقویت کند.

3.MESIF (Modified, Exclusive, Shared, Invalid, Forward):

پروتکل MESIF، که گسترش یافته‌ی MESI است، وضعیت "Forward" را به معادلات اضافه می‌کند. این وضعیت این امکان را به کش می‌دهد که اعلام کند نسخه‌ای از سطر کش را در اختیار دارد، در حالی که ممکن است همان سطر در کش‌های دیگری هم به صورت به اشتراک گذاشته شده (Shared) باشد. وضعیت Forward دسترسی‌های بی‌مورد به حافظه اصلی را کاهش می‌دهد، مخصوصاً زمانی که بیش از یک کش بخواهند به یک بلوک کش مشابه دست یابند.

ب) آیا قفل های بلیط، مشکل نامعتبر بودن یک داده ی تغییر یافته توسط یک پردازنده برای یک پردازنده دیگر را دارد؟

بله، این قفل ها مشکل مذکور را دارند. برای توضیح علت این موضوع، ابتدا به اختصار این قفل ها را توضیح میدهیم:

قفل بلیط یک مکانیسم همگام سازی است که در سیستم عامل ها برای کنترل دسترسی به یک منبع مشترک توسط رشته ها یا پردازنده های متعدد استفاده می شود. همچنین می تواند از گرسنگی قفل جلوگیری کند و عدالت را در تخصیص منابع فراهم کند.
اجزای اصلی قفل بلیط دو عدد صحیح هستند به آن "بلیط" و "نوبت" می گوییم.
بلیط : هنگامی که یک رشته می خواهد قفل را بدست آورد، یک شماره بلیط منحصر به فرد دریافت می کند.
نوبت: عدد دیگری وجود دارد که نشان‌دهنده نوبت پردازه ای است که قفل را در اختیار دارد.
هنگامی که یک پردازه سعی می کند قفل را بدست آورد، کارهای زیر را انجام می دهد:
بلیط را دریافت می کند و شماره بلیطی که نفر بعد باید دریافت کند را یک واحد افزایش میدهد. و منتظر میماند تا شماره بلیط آن با شماره نوبت مطابقت داشته باشد.
هنگامی که این اعداد مطابقت دارند، وارد بخش بحرانی می شود.
به محض خروج از بخش بحرانی، پردازه عدد نوبت را افزایش می دهد و به رشته منتظر بعدی اجازه می دهد تا قفل را بدست آورد.

همانطور که مشاهده میکنیم، پردازه های مختلف میخواهند مقادیر بلیط بعدی و نوبت حال حاضر را تغییر دهند. که این موضوع باز هم از cache coherency in multi-processor systems رنج میبرد.

ج) چگونه میتوان در لینوکس داده های مختص هر هسته را در زمان کامپایل تعریف نمود؟

در لینوکس، تعریف داده‌های خاص برای هر هسته در زمان کامپایل، یک روش معمول نیست، زیرا این کار بیشتر در زمان اجرا توسط زمان‌بندی لینوکس انجام می‌شود. با این حال، در هسته لینوکس، مفهومی به نام "متغیرهای هر CPU" وجود دارد که به هسته اجازه می دهد تا متغیرهایی را تعریف کند که مختص هر هسته CPU است. برای کار کردن با آن، باید مراحل زیر را انجام دهیم:

۱. تعریف آن متغیر:
متغیرهای هر CPU با استفاده از ماکرو DEFINE_PER_CPU تعریف می شوند. این ماکرو یک نمونه جداگانه از یک متغیر را برای هر CPU اختصاص می دهد. که استفاده آن به شکل زیر است:

```
DEFINE_PER_CPU(type, name);
```

برای مثال برای تعریف یک عدد صحیح در یک CPU، از دستور زیر استفاده میکنیم:

```
DEFINE_PER_CPU(int, myCpuData);
```

۲.مقداردهی اولیه این متغیر ها:
اگر مقداردهی اولیه باید هنگام راه اندازی سیستم اتفاق بیفتد، می توان متغیرهای هر CPU را با استفاده از __init و __initdata مقداردهی کرد، یا از ماکرو per_cpu در داخل تابعی استفاده کرد که در زمان اجرا فراخوانی می شود. برای مثال میتوان از قطعه کد زیر برای مقداردهی اولیه این متغیر ها استفاده کرد:

```
static int __init my_module_init(void) {
    int cpu;
    for_each_possible_cpu(cpu) {
        per_cpu(cpu_specific_data, cpu) = initial_value_for_core(cpu);
    }
    return 0;
}
```

۳. دسترسی به این متغیر ها:

برای دسترسی به یک متغیر هر CPU، می‌توان از ماکروهای get_cpu_var و put_cpu_var استفاده کرد، که pre-emption را غیرفعال می‌کنند تا اطمینان حاصل شود که یک CPU هنگام کار بر روی متغیر خود برنامه‌ریزی نمی‌شود: برای مثال:

```
int value = get_cpu_var(cpu_specific_data);
put_cpu_var(cpu_specific_data);
```

از طرف دیگر، می توان از ماکروهای this_cpu_read یا this_cpu_write استفاده کرد که به متغیر های هر CPU دسترسی دارند:

```
this_cpu_write(cpu_specific_data, new_value);
int value = this_cpu_read(cpu_specific_data);
```

پیاده سازی

در ابتدا برای شبیه سازی دو نوع حافظه دو پیاده‌سازی داریم. اولین نوع پیاده سازی حافظه نهان local پردازنده است که در struct cpu پیاده سازی شده است و نام آن **syscall_count** است.

```
struct cpu
{
  uchar apicid;              // Local APIC ID
  struct context *scheduler; // swtch() here to enter scheduler
  struct taskstate ts;       // Used by x86 to find stack for interrupt
  struct segdesc gdt[NSEGS]; // x86 global descriptor table
  volatile uint started;     // Has the CPU started?
  int ncli;                  // Depth of pushcli nesting.
  int intena;                // Were interrupts enabled before pushcli?
  struct proc *proc;         // The process running on this cpu or null
  uint syscall_count;
};
```

برای پیاده سازی حافظه مشترک در فایل mp.c که متغیر های مشترک میان cpu ها را داراست متغیر **syscall_count_total** را اضافه می‌کنیم.

```
struct cpu cpus[NCPU];
int ncpu;
uchar ioapicid;

uint syscall_count_total;
```

حال هربار که یک فراخوانی سیستمی صدا می‌شود باید هم به cpu ای که این تابع را فراخوانده و هم به متغیر گلوبال حافظه مشترک یکی اضافه کنیم.

توجه شود که چون ممکن است race condition رخ دهد باید از متغیر گلوبال مراقبت شود.

```
void syscall(void)
{
  int num;
  struct proc *curproc = myproc();

  cli();
  mycpu()->syscall_count++;
  sti();
  syscall_count_total++;
  __sync_synchronize();

  num = curproc->tf->eax;
  if (num > 0 && num < NELEM(syscalls) && syscalls[num])
  {
    curproc->tf->eax = syscalls[num]();
  }
  else
  {
    cprintf("%d %s: unknown sys call %d\n",
            curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
  }
}
```

در انتها یک فراخوانی سیستمی که مقادیر system call برای هر cpu و تمامی فراخوانی‌های کل cpu ها را نمایش داده و counter ها را ریست می‌کند طراحی می‌کنیم.

```
int sys_print_syscall_count()
{
  for (int i = 0; i < NCPU; i++)
  {
    cprintf("cpu number %d has run %d systemcalls\n", cpus[i].apicid, cpus[i].syscall_count);
  }

  cprintf("total number of system calls are %d\n", syscall_count_total);
  return 0;
}
```

حال برای اینکه متغیر ها هر بار که یه بار کاری به اتمام می‌رسد ریست شوند تابع سیستمی **reset_syscall_count** را تعریف کرده و بعد از هربار اجرای بخش EXEC در shell آن را صدا می‌کنیم.

```
case EXEC:
  ecmd = (struct execcmd *)cmd;
  if (ecmd->argv[0] == 0)
    exit();
  reset_syscall_count();
  exec(ecmd->argv[0], ecmd->argv);
  printf(2, "exec %s failed\n", ecmd->argv[0]);
  break;
```

(فایل sh.c)

```
void reset_syscall_count(void)
{
  cli();
  for (int i = 0; i < ncpu; i++)
    cpus[i].syscall_count = 0;
  sti();
  syscall_count_total = 0;
  __sync_synchronize();
}
```

(فایل proc.c)

```
int sys_reset_syscall_count()
{
  reset_syscall_count();
  return 0;
}
```

(فایل sysproc.c)

توجه شود که برای استفاده از ۴ پردازنده makefile را به صورت زیر تغییر می‌دهیم.

```
QEMUGDB = $(shell if $(QEMU) -help | grep -q '^-gdb'; \
    then echo "-gdb tcp::$(GDBPORT)"; \
    else echo "-s -p $(GDBPORT)"; fi)
ifndef CPUS
CPUS := 4
endif
QEMUOPTS = -drive file=fs.img,index=1,media=disk,format=raw
```

در انتها یک برنامه کاربر اجرا می‌کنیم که دارای ۴ process است و هر process روی یک cpu اجرا می‌شود.

```
void write_text(const char *text)
{
    char stringify_pid[20];
    itoa(getpid(), stringify_pid);
    int fd = open(strcat(strcat("output", stringify_pid), ".txt"), O_WRONLY | O_CREATE);
    if (fd < 0)
    {
        printf(2, "Failed to open file for writing\n");
        exit();
    }

    write(fd, text, strlen(text));
    close(fd);
}
```

```c
int main()
{
    int i;
    int pid;

    for (i = 0; i < NUM_PROCESSES; i++)
    {
        pid = fork();
        if (pid < 0)
        {
            printf(2, "Fork failed\n");
            exit();
        }
        else if (pid == 0)
        {
            char stringify_pid[20];
            itoa(getpid(), stringify_pid);
            char* message = strcat("Hello from Process ", stringify_pid);
            message = strcat(message, "\n");
            write_text(message);
            exit();
        }
    }

    for (i = 0; i < NUM_PROCESSES; i++)
    {
        wait();
    }

    print_syscall_count();

    exit();
}
```

خروجی برنامه به صورت زیر است:

```
$ syscall_count_test
cpu number 0 has run 8 systemcalls
cpu number 1 has run 12 systemcalls
cpu number 2 has run 12 systemcalls
cpu number 3 has run 6 systemcalls
total number of system calls are 38
$ syscall_count_test
cpu number 0 has run 6 systemcalls
cpu number 1 has run 6 systemcalls
cpu number 2 has run 13 systemcalls
cpu number 3 has run 13 systemcalls
total number of system calls are 38
```

پیاده سازی سازوکار همگام سازی با قابلیت اولویت دادن

برای قفل جدید دو فایل جدید تعریف می‌کنیم . یکی از فایل ها prioritylock.h است که در آن دو تعریف داریم. یکی تعریف struct قفل است. این struct شامل یک spinlock برای محافظت از داده های قفل هنگام تغییر تعبیه شده است. همچنین داخل این استراکت ابتدای صف process های در انتظار برای دریافت قفل را داریم. این صف به صورت یک linked list تعبیه شده و همواره مقدار ابتدای آن در اولویت برای دریافت قفل است. تعریف هر عضو لیست را در struct بالا یعنی node داریم. این struct دارای اولویت خانه ،process موجود در آن و خانه‌ی بعدی لیست است.

```
struct node
{
  uint priority;
  struct node *next;
  struct proc *process;
};

struct prioritylock
{
  uint locked;          // Is the lock held?
  struct spinlock slk; // spinlock protecting this priority lock
  char *name;           // Name of lock.
  int pid;              // Process holding lock
  struct node *queue;  // queue
};
```

(فایل prioritylock.h)

حال در ادامه برای استفاده از قفل سه تابع در فایل prioritylock.c تعریف می‌کنیم.
initpriorityلock برای ساخت قفل است. در این تابع یک قفل ایجاد می‌کنیم.

```
void initpriorityлock(struct priorityлock *lk, char *name)
{
  lk->name = name;
  lk->locked = 0;
  lk->pid = 0;
  lk->queue = 0;
  initlock(&lk->slk, "spin lock");
}
```

(تابع initpriorityлock)

در قسمت بعدی تابع acquirepriorityлock را داریم که برای دریافت قفل استفاده می‌شود. به این صورت که اگر process خواستار قفل وارد شود و کسی قفل را در اختیار نداشته باشد مستقیما قفل را دریافت کرده و خارج می‌شود. حال اگر قفل فعال باشد به این صورت عمل می‌کنیم که این process با توجه به مقدار آیدی در جای مناسب در لیست وارد می‌شود و سپس آن را sleep می‌کنیم تا نوبت به آن برسد.

```
void acquirepriority(struct prioritylock *lk)
{
  acquire(&lk->slk);
  if (lk->locked)
  {
    struct proc *cur_proc = myproc();
    struct node *p = (struct node *)kalloc();
    p->next = 0;
    p->priority = cur_proc->pid;
    p->process = cur_proc;

    struct node *q = lk->queue;
    if (q == 0 || p->priority > q->priority)
    {
      lk->queue = p;
      p->next = q;
    }
    else
    {
      while (q->next != 0 && p->priority <= q->next->priority)
      {
        q = q->next;
      }
      p->next = q->next;
      q->next = p;
    }
    sleep(cur_proc, &lk->slk);
  }
  else
  {
    lk->locked = 1;
    lk->pid = myproc()->pid;
  }
  release(&lk->slk);
}
```

(acquirepriorityلock تابع)

در releaseprioritylock برای آزاد سازی قفل تلاش می‌کنیم. به این صورت که اگر process ای در صف منتظر باشد قفل را به او منتقل می‌کنیم و آن process را wakeup می‌کنیم . در غیر این صورت قفل را آزاد می‌کنیم.

```
void releasepriority(struct prioritylock *lk)
{
  if (lk->pid != myproc()->pid || !lk->locked)
  {
    cprintf("the process %d does not own the lock to release it\n", myproc()->pid);
    return;
  }

  acquire(&lk->slk);

  if (lk->queue != 0)
  {
    struct node *p = lk->queue;
    print_prioritylock_queue(lk);
    lk->queue = p->next;
    p->process->state = RUNNABLE;
    lk->pid = p->process->pid;
    wakeup(p->process);
    kfree((char *)p);
  }
  else
  {
    lk->locked = 0;
    lk->pid = 0;
  }

  release(&lk->slk);
}
```

(releaseprioritylock تابع)

حال برای استفاده از این قفل سیستم کال های زیر را در prioritylock_test_util.c تعریف می‌کنیم .

توجه شود که در این متغیر قفل به صورت گلوبال در همین فایل قرار دارد.

```c
struct prioritylock pl;

int sys_init_prioritylock(void)
{
    initprioritylock(&pl, "priority_lock");
    return 0;
}

int sys_acquire_prioritylock(void)
{
    acquirepriority(&pl);
    return 0;
}

int sys_release_prioritylock(void)
{
    releasepriority(&pl);
    return 0;
}
```

(فایل prioritylock_test_util.c)

حال برنامه‌ی سطح کاربری طراحی می‌کنیم که کارایی این قفل را به نمایش بگذاریم.

```c
#include "types.h"
#include "user.h"

#define NCHILD 10

void process_function(int i)
{
  acquire_prioritylock();
  printf(1, "Process %d acquired the lock.\n", getpid());
  sleep(500);

  release_prioritylock();
  printf(1, "Process %d released the lock.\n", getpid());
  exit();
}
```

```c
int main()
{
  init_prioritylock();
  for (int i = 0; i < NCHILD; i++)
  {
    int pid = fork();
    if (pid < 0)
    {
      printf(1, "Fork failed.\n");
      exit();
    }
    else if (pid == 0)
    {
      process_function(i);
    }
  }
  for (int i = 0; i < 10; i++)
  {
    wait();
  }
  exit();
}
```

خروجی برنامه به صورت زیر است :

```
Group #31:
1. Mahdi Noori
2. AmirAli Rahimi
3. Soheil Hajian
$ prioritylock_test
Process 4 acquired the lock.
Process 4 released the lock.
the queue is : 13 12 11 10 9 8 7 6 5
Process 13 acquired the lock.
Process 13 released the lock.
the queue is : 12 11 10 9 8 7 6 5
Process 12 acquired the lock.
Process 12 released the lock.
the queue is : 11 10 9 8 7 6 5
Process 11 acquired the lock.
Process 11 released the lock.
the queue is : 10 9 8 7 6 5
Process 10 acquired the lock.
Process 10 released the lock.
the queue is : 9 8 7 6 5
Process 9 acquired the lock.
Process 9 released the lock.
the queue is : 8 7 6 5
Process 8 acquired the lock.
Process 8 released the lock.
the queue is : 7 6 5
Process 7 acquired the lock.
Process 7 released the lock.
the queue is : 6 5
Process 6 acquired the lock.
Process 6 released the lock.
the queue is : 5
Process 5 acquired the lock.
Process 5 released the lock.
```

آیا این پیاده سازی ممکن است که دچار گرسنگی شود؟ راه حلی برای برطرف کردن این مشکل ارائه دهید. روش ارائه شده توسط شما باید بتواند شرایطی را که قفل ها دارای اولویت یکسان می باشند را نیز پوشش دهد.

بله . در صورتی که process با آیدی بیشتر پشت سر هم وارد صف شود هر بار این process انتخاب شده و نوبت به process ها با آیدی کمتر نمی‌رسد.
برای رفع این مشکل از دو مکانیزم استفاده می‌کنیم. بخش اول اجرای عملیات Aging است به این صورت که هر بار به مقدار کوانتوم زمانی برسیم بررسی می‌کنیم که کدام یک از process ها بیشتر از این بازه در صف منتظر مانده‌اند. هر کدام که دارای این شرط باشند را یکی از مقدار الویتش زیاد می‌کنیم و جای جدیدی برای آن می‌یابیم.
توجه شود که در این قسمت ما این process را قبل از process ها با اولویت یکسان می‌گذاریم چرا که مدت زمان بیشتری در صف بوده است.
مکانیزم دوم این است که در حین اضافه کردن یک process جدید باید آن را بعد از سایر process ها با اولویت یکسان قرار دهیم بدین دلیل که دیرتر از مابقی وارد شده است.

یک نوع پیاده سازی همگام سازی توسط قفل بلیت انجام می‌شود. آن را بررسی کنید و تفاوتهای آن با روش همگام سازی بالا را بیان کنید.

قفل های بلیت مکانیزمی برای هماهنگ سازی است که برای تحقق عدالت در زمینه دسترسی به منابع است. این نوع از قفل ها اطمینان حاصل می‌کنند که هر process به ترتیبی که قفل را درخواست می‌کند آن را دریافت کرده که همین موجب جلوگیری از گرسنگی می‌شود.
روش پیاده سازی این قفل به شرح زیر است:
۱. دو متغیر ticket و turn را که بین تمامی process ها مشترکند تعریف می‌کنیم.

```
typedef struct
{
    atomic_int ticket; // Ticket number
    atomic_int turn;   // Turn number
} TicketLock;
```

۲. وقتی یک process می‌خواهد قفل دریافت کند به صورت atomic مقدار ticket را زیاد کرده و در اختیار process می‌گذاریم.

۳. process تا زمانی که شماره بلیط یکسانی با turn داشته باشد صبر می‌کند.

```
void ticket_lock_acquire(TicketLock *lock)
{
    int my_ticket = atomic_fetch_add(&lock->ticket, 1);
    while (atomic_load(&lock->turn) != my_ticket)
        ; // Wait until it's our turn
}
```

۴. حال وقتی نوبت به process رسید این process می‌تواند وارد قسمت حیاتی کد شود.

۵. بعد از خروج process برای آزاد سازی قفل مقدار turn را یکی زیاد می‌کند تا نوبت به process بعدی برسد.

```
void ticket_lock_release(TicketLock *lock)
{
    atomic_fetch_add(&lock->turn, 1); // Increment the turn number
}
```

از تفاوت های مهم این دو روش می‌توان به موارد زیر اشاره کرد :

۱. ترتیب اختصاص قفل برای دو مکانیزم متفاوت است که همین می‌تواند در قفل با اولویت موجب گرسنگی شود.
۲. در روش قفل بلیت دیگر نیازی به استفاده از آیدی process برای اولویت بندی نیست و ما را از دانستن آن بی‌نیاز می‌کند.
۳. میزان عدالت در روش قفل بلیت بیشتر است.
۴. در روش قفل بلیط از الگوی first-in first-served استفاده می‌شود در حالی که در قفل اولویت دار از الگویی برای رتبه بندی هر process استفاده می‌شود.