

1) علت غیر فعال کردن وقفه ها هنگام اجرای مسیر بحرانی و امکان ایجاد *deadlock*

اصولا برای همگام سازی¹ بین پردازش های مختلف ، مسیر هایی در کد که بین داده ها در پردازش ها وابستگی وجود دارد یک مسیر بحرانی قائل می شویم. هنگامی که یک پردازش در بخش بحرانی قرار دارد هیچ پردازش دیگری نمی تواند به آن بخش ورود کند و در واقع داده های مشترک در هر لحظه توسط تنها و تنها یک پردازش قابل دسترسی میباشد.

چند دلیل برای غیرفعال شدن وقفه ها در زمانی که یک پردازش در مسیر بحرانی قرار دارد وجود دارد که به شرح زیر است:

- **اتمیک بودن برنامه** : اصولا وقتی ما وقفه را غیر فعال کنیم بدین معناست که پردازش ها وقتی در مسیر بحرانی خود هستند امکان انقطاع² برای آنها وجود ندارد و یک پردازش تا وقتی مسیر بحرانی خود را به اتمام نرسانده باشد نمیتوان آن را بوسیله وقفه دیگری از پردازنده خارج کرد.
- **همگام بودن برنامه** : کل فلسفه وجود این بخش این بود که وقتی چند پردازش همزمان به داده هایی دسترسی داشته باشند امکان تصادم³ وجود دارد. لذا ما مسیر بحرانی را تعریف کردیم تا ازین اتفاق جلوگیری کنیم و فقط یک پردازش به دیتاهای مشترک در آن واحد دسترسی داشته باشد.
- **جلوگیری از قفل**: فرض کنید دو پردازش P_1 و P_2 در سیستم باشند و پردازش P_1 در مسیر بحرانی خود توسط یک وقفه پردازش را به P_2 تحویل می دهد. حال این پردازش در جایی از متن خود به دیتایی از سوی پردازش P_1 نیاز دارد ولی چون این پردازش دچار وقفه شده است نمی تواند دیتا را به P_2 بدهد. حال هر دو پردازش متوقف شدند و سیستم دچار قفل می شود. لذا برای جلوگیری از قفل نشدن CPU باید تا پایان مسیر بحرانی در حال اجرا ، وقفه ها غیر فعال شوند.

2) توابع *pushcli* و *popcli* و تفاوت آنها با *sti* و *cli*

توابع *cli* و *sti* به ترتیب برای غیر فعال و فعال کردن وقفه ها استفاده می شوند. توابع *popcli* و *pushcli* اما با یک متغیر کمکی بنام *ncli* این کار را با یک استک مجازی انجام می دهند. در واقع ما هر بار با تابع *pushcli* یک وقفه در استک اضافه میکنیم این مقدار یک واحد افزایش یافته و هنگامی که با تابع *popcli* یک وقفه را خارج میکنیم یک واحد کاسته میشود. تا زمانی که مقدار متغیر *ncli* بزرگتر از صفر باشد یعنی استک پر بوده و در نتیجه تمامی وقفه ها غیرفعال هستند. به محض اینکه استک با دستور *popcli* خالی شد وقفه ها مجدداً فعال خواهند شد. تابع *pushcli* هر بار که فراخوانی شود تابع *cli* را صدا زده و وقفه ها را غیرفعال میکند. اما تابع *popcli* تنها وقتی استک خالی شد ، یعنی *ncli* برابر صفر شد تابع *sti* را صدا زده و وقفه ها را فعال میکند. کاربرد این توابع وقتی است که چند قفل وجود دارد و ما می خواهیم با رها شدن یک قفل ، وقفه ها فعال نشوند و این اتفاق تنها وقتی رخ دهد که تمامی قفل ها رها شده باشند.

¹ Synchronization

² Preempt

³ Conflict

(3) مناسب نبودن قفل چرخشی در سیستم های تک هسته

اگر نگاهی به تابع *acquire* که در فایل *spinlock.c* تعریف شده است بیندازیم ، میبینیم که در ابتدای آن با فراخوانی تابع *pushcli()* تمامی وقفه ها غیر فعال میشوند و سپس در یک حلقه دستور اتمی *xchg()* انجام میشود. این دستور یک خانه از حافظه را با یک رجیستر تعویض میکند:

```
3 void acquire(struct spinlock *lk)
4 {
5     pushcli(); // disable interrupts to avoid deadlock.
6     if (holding(lk))
7         panic("acquire");
8
9     // The xchg is atomic.
10    while (xchg(&lk->locked, 1) != 0)
11        ;
12
13    // Tell the C compiler and the processor to not move loads or stores
14    // past this point, to ensure that the critical section's memory
15    // references happen after the lock is acquired.
16    __sync_synchronize();
17
18    // Record info about lock acquisition for debugging.
19    lk->cpu = mycpu();
20    getcallerpcs(&lk, lk->pcs);
21 }
```

تابع *acquire* همیشه مقدار *lk-->locked* را بوسیله دستور *xchg* برابر 1 قرار می دهد. با اجرای این دستور اگر قفل قبلا توسط پردازنده دیگری درگیر بوده باشد و مقدار *locked* از قبل برابر یک باشد ، این تابع مقدار یک بر میگردد. اما اگر قفل رها بوده و مقدار *locked* برابر صفر باشد این تابع مقدار آن را به یک تغییر داده و صفر بر میگردد که یعنی قفل از قبل درگیر نبوده و فقط درین حالت است که پردازنده از حلقه خارج شده و به ادامه روند خود می پردازد.

روشن است درین روش انتظار مشغول⁴ داریم که در سیستم های چندپردازنده ای باعث هدر رفتن زمان و کاهش کارایی سیستم می شود. حال چنانچه بین دو پردازنده اولی قفل را بگیرد و دومی تلاش کند با قفل چرخشی و روشی که ذکر شد قفل را برای خودش بدست آورد ، چون قفل توسط اولی گرفته شده لذا مقدار *locked* یک است و دومین پردازنده هیچ گاه از حلقه خارج نمی شود. لذا مثال فوق نشان می دهد که در بدترین حالت در سیستم های تک پردازنده دچار *deadlock* خواهیم شد و این روش برای این گونه از سیستم ها مناسب نیست.

⁴ Busy waiting

(4) تعریف تابع *amoswap* و نحوه کارکرد آن

این تابع مخفف *atomic memory operation swap* است که محتوای خانه ای از حافظه را با مقدار یک رجیستر بصورت اتمیک جابجا میکند. سایر اپراتور های اتمی حافظه در تصویر زیر آمده اند:

9.4 Atomic Memory Operations

31	27	26	25	24	20	19	15	14	12	11	7	6	0
funct5	aq	rl	rs2	rs1	funct3	rd	opcode						
5	1	1	5	5	3	5	7						
AMOSWAP.W/D	ordering		src	addr	width	dest	AMO						
AMOADD.W/D	ordering		src	addr	width	dest	AMO						
AMOAND.W/D	ordering		src	addr	width	dest	AMO						
AMOOR.W/D	ordering		src	addr	width	dest	AMO						
AMOXOR.W/D	ordering		src	addr	width	dest	AMO						
AMOMAX[U].W/D	ordering		src	addr	width	dest	AMO						
AMOMIN[U].W/D	ordering		src	addr	width	dest	AMO						

این تابع 3 آرگومان *rs1* ، *rs2* و *rd* را دارد. ابتدا به آدرس ذخیره شده در *rs1* که خانه ای از حافظه است رفته و محتوای آن را *load* میکند. سپس این محتوا را در *rd* ذخیره کرده و در مرحله بعد مقدار *load* شده از *rs1* را با مقدار رجیستر *rs2* جابجا کرده و در آخر مقدار جدید را در آدرس حافظه موجود در *rs1* ذخیره میکند.

دلیل تعریف و استفاده از این توابع این است که در اصطلاح به آنها اپراتور های *read-modify-write* می گوئیم. یعنی اصولاً مقداری را از حافظه در رجیستر آورده تغییراتی روی آن اعمال کرده و مجدداً آن را در حافظه می نویسند و تمام این فرایند را بصورت اتمیک انجام می دهند. بدیهی است اتمیک بودن این اپراتور ها منجر به این میشود که در هر لحظه از برنامه محتواهای حافظه سیستم تنها در یک جا تغییر یابند و اگر این اپراتور های اتمیک نباشند ، مسئله *shared data* مجدد پیش خواهد آمد و ممکن است یک اپراتور در میانه فرایند *modify* خود بدلیل دسترسی اپراتور دومی به محتوای حافظه ای که داشته روی آن تغییرات اعمال میکرده دچار توقف شود.

(5) تعامل پردازنده ها در دسته دوم از توابع *sleeplock*

ابتدا نگاهی به بدنه ساختار این نوع قفل بیندازیم:

```
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
};
```

درین قفل یک متغیر *locked* داریم که نشان دهنده درگیر بودن یا نبودن قفل است و یک قفل چرخشی برای حفاظت از اجزای این ساختمان داده استفاده میشود.

در تابع *acquiresleep* طبق تصویر زیر ، ابتدا قفل چرخشی گرفته شده و سپس بررسی میشود آیا *sleeplock* قبلاً گرفته شده است یا خیر. در صورتی که گرفته نشده باشد ، پردازش آن را به خودش اختصاص داده و قفل چرخشی را رها می کند. اما در صورتی که قفل توسط پردازش دیگری گرفته شده باشد ، پردازش در حالت *sleep* خواهد ماند:

```
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);
}
```

اما برای *sleep* کردن پردازش از تابع *sleep* استفاده می کنیم و دو آرگومان مطابق شکل زیر به آن پاس داده میشود:

```
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);
    }
    // Go to sleep.
    p->chan = chan;
    p->state = SLEEPING;

    sched();

    // Reacquire original lock.
    if (lk != &ptable.lock)
    {
        // DOC: sleeplock2
        release(&ptable.lock);
        acquire(lk);
    }
}
```

اولین آرگومان آن یکی از متود های ساختار *proc* است که نشان می دهد پردازش روی آن خوابیده یا بیدار است و آرگومان دوم نیز قفل چرخشی مربوط به پردازش است.

درین تابع قبل از تغییر وضعیت پردازش ابتدا *lk* را رها می کنیم و سپس پردازش را روی *chan* به حالت خواب برده و وضعیت آن را روی *SLEEPING* قرار می دهیم و بعدا که دوباره *wakeup* کرد و توسط تابع زمان بند⁵ به این تابع برگشت (با فراخوانی تابع *sched*) مجددا قفل را *acquire* میکنیم. حال که این پردازش در حالت اجرا نیست ، پردازش ای که قفل را در اختیار دارد می تواند *releasesleep* کند:

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

درین تابع چون پردازش قبلی قفل را برای تغییر حالت رها کرد ، این یکی پردازش می تواند آن را *acquire* کند. سپس لاک را رها می کند و با فراخوانی *wakeup* این تابع روی ورودی *chan* void* پیمایش می کند و تمامی پردازش هایی که روی آن خواب بودند را به حالت *RUNNABLE* در میآورد تا توسط زمان بند قابل انتخاب شدن باشند. بعد از تغییر حالت پردازش ها توسط تابع زمان بند تابع *sched* صدا میشود و لذا تمام پردازش هایی که روی *sleeplock* خواب بودند (چون در تابع *acquiresleep* خود *sleeplock* را بعنوان *chan* پاس دادیم) روی این قفل بیدار می شوند.

دقت شود در تابع *acquiresleep* ما عملیات *sleep* را در یک حلقه *while* انجام میدهم و دلیل آن این است که اگر پردازش ای به هر دلیل *wakeup* کرد ولی همچنان قفل رها نشده بود مجدد به حالت *sleep* برگردد

⁵ Scheduler

(6) حالات مختلف پردازش در xv6 و تابع sched

حالات مختلف پردازش ها در فایل proc.h طبق تصویر زیر قابل دسترسی است:

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

طبق تصویر شش حالت برای پردازش ها متصور است که به اختصار هر یک را توضیح می‌دهیم:

- **UNUSED** : در کل یک آرایه 64 تایی از پردازش ها داریم و به خانه هایی که در آنها پردازش ای نباشد اطلاق میشود
- **EMBRYO** : حالت پردازش ای است که تازه (مثلا با fork) ساخته می شود. عملاً تابع *allocproc* از بین پردازش های UNUSED یکی را انتخاب می کند و حالت آن را به *EMBRYO* تغییر می دهد
- **SLEEPING** : درین حالت پردازش در بین انتخاب های زمانبند برای تخصیص پردازنده به خود قرار ندارد و بدون فعالیت باقی می ماند. پردازش یا به انتخاب خود و یا توسط کرنل می تواند به این حالت برود و در انتظار منبع بماند
- **RUNNABLE** : درین حالت پردازش در صف اجرای زمان بند قرار داشته و در یکی از راند های زمان بندی بعدی CPU به آن اختصاص داده خواهد شد و به حالت *RUNNING* می رود
- **RUNNING** : درین حالت پردازش در CPU در حال اجراست (بدیهی است تعداد پردازش های در حالت *RUNNING* نمی تواند از تعداد پردازنده ها بیشتر باشد)
- **ZOMBIE** : وقتی پردازش ای کارش تمام می شود و می خواهد *exit* کند ، قبل از آنکه *UNUSED* شود به این حالت می رود تا پدرش بتواند از طریق تابع *wait* بفهمد کار فرزندش تمام شده است.

وظیفه تابع *sched* تعویض متن⁶ به متن زمان بند می باشد. یک پردازش هنگام ترک CPU ابتدا باید *ptable* را لاک کند و به حالت *RUNNABLE* تغییر وضعیت داده و سپس به تابع *sched* برود.

سپس درین تابه با استفاده از دستور *swtch* که به زمان اسمبلی نوشته شده است ، عملیات تعویض متن صورت گرفته و در ادامه به تابع *scheduler* برمی گردیم و متن را به متن یکی از پردازش های *RUNNABLE* تغییر می دهیم.

⁶ Context switch

(7) آزادسازی قفل توسط پردازنده صاحب آن

همانطور که پیش تر در آخرین تصویر صفحه سوم دیدیم ، ساختار sleeplock یک *pid* دارد که ما نیز از همین فیلد استفاده میکنیم و با اضافه کردن یک قطعه شرط ساده بررسی میکنیم که اگر *pid* پردازنده و قفل یکی بودند سپس قفل را رها می کنیم و اگر نبود با دستور return بر میگردیم.

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

    //add if statement to check who has lock
    if (lk->pid != myproc()->pid)
    {
        release(&lk->lk);
        return;
    }

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

در هسته لینوکس در پوشه `include/linux/mutex.h` ساختار یک قفل متقابل⁷ قابل مشاهده است که دارای فیلدی بنام *owner* از تایپ `atomic_long_t` است که نشان می دهد صاحب این قفل کدام پردازنده است تا هنگام رها کردن فقط همان پردازنده قادر به release باشد:

```
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
};
```

⁷ mutex

8) الگوریتم های lock-free و مقایسه آن با locked-base

الگوریتم های lock-free طراحی شده اند تا بوسیله آنها در سیستم های چند ریشه ای⁸ بتوانیم بدون نیاز به ساز و کار های مرسوم همگام سازی مثل تقابل⁹ که تا اینجای کار دیدیم پیشرفت در سیستم را تضمین کنیم. در واقع هدف از ایجاد این الگوریتم ها تضمین این موضوع است که حتی در صورت وجود داشتن رقابت بین ریشه ها ، حداقل یک ریشه در حال پیشرفت وجود داشته باشد و برای مثال در هیچ کجای برنامه دچار deadlock نشویم .

در مقایسه با الگوریتم های صرفا مبتنی بر قفل¹⁰ ، این الگوریتم ها مزایا و معایبی دارند که مختصرا به آنها اشاره میکنیم:

مزایا

- ✓ **Concurrency** : این الگوریتم ها با اجازه دادن به ریشه ها برای انجام همزمان عملیات های شان ، هم روندی بالاتری را نسبت به الگوریتم های مبتنی بر قفل دارد که منجر به مقیاس پذیری و کارایی بالاتری در سیستم های چند هسته¹¹ است
- ✓ **Avoiding Deadlock** : این الگوریتم ها به جهت توازی پیشرفت ریشه ها از deadlock و انتظار ریشه ها برای آزاد شدن قفل توسط ریشه دیگر مخصوصا در سیستم های چند ریشه ای جلوگیری می کنند.
- ✓ **Predictable Performance** : با پیچیده شدن رقابت بین ریشه ها کارایی یک برنامه مبتنی بر قفل قابل پیش بینی و تقریب زدن نخواهد بود که این مشکل در الگوریتم های lock-free تا حد بسیار خوبی حل شده و کارایی قابل محک است
- ✓ **Scalability** : به جهت اینکه درین الگوریتم ها زمان انتظار ریشه ها برای قفل کمینه است ، لذا در مقایسه با سیستم های مبتنی بر قفل مقیاس پذیری بالاتری دارند و قابل استفاده تر هستند.

معایب

- ✗ **Complexity** : طراحی این الگوریتم ها به جهت استفاده از تکنیک های پیچیده مثل عملگر های اتمیک و نوبت دهی حافظه معمولا دارای پیچیدگی بیشتری نسبت به الگوریتم های lock-based هستند
- ✗ **Overhead** : به جهت استفاده از عملگر های اتمی و دیگر تکنیک هایی که برای همگام سازی درین نوع الگوریتم وجود دارد ، ممکن است با بالا رفتن تعداد ریشه ها یا پیچیده شدن عملیات ریشه ها ، سر بار زیادی به همراه داشته باشد
- ✗ **Limited Application** : این الگوریتم ها برای هر جایی مناسب نیستند و استفاده از آنها در جاهایی که کارایی مطلوبی با lock دریافت می کنیم ممکن است نسبت به هزینه ای که می کنیم پیشرفت چشمگیری نداشته باشد لذا در سیستم هایی که ریشه ها رقابت کمی با هم دارند استفاده ازین الگوریتم ها توصیه نمی شود.
- ✗ **Debugging** : به دنبال پیچیده بودن تکنیک های مورد استفاده درین الگوریتم ها ، طبعا دیباگ کردن آنها نیز سخت تر خواهد بود و پیدا کردن مشکلات برآمده از رقابت ریشه ها و حل کردن آنها با چالش بیشتری همراه است.

⁸ Multiple threads⁹ mutex¹⁰ Locked algorithms¹¹ Multi-core systems

Hardware Approach for cache Coherence

یکی از رایج ترین و پرکاربردترین رویکردهای سخت افزاری برای مدیریت انسجام cache در سیستم های چند پردازنده ای مدرن، پروتکل MESI است. پروتکل MESI یک پروتکل انسجام حافظه نهان است که به طور گسترده ای مورد استفاده است که به حفظ یک نگاه منسجم از حافظه مشترک در میان هسته های متعدد پردازنده کمک می کند. MESI مخفف Exclusive، Modified، Shared و Invalid است که بیانگر چهار حالت ممکن یک خط cache در حافظه نهان پردازنده است.

- ✓ **Modified** : خط cache در حافظه پنهان وجود دارد و اصلاح شده است. این بدان معناست که حافظه نهان، حاوی تنها نسخه معتبر داده است و قبل از اینکه پردازنده دیگری بتواند آن را بخواند، باید به حافظه اصلی بازگردانده شود.
- ✓ **Exclusive** : خط کش در حافظه پنهان وجود دارد و تغییری در آن انجام نشده است. این تنها نسخه معتبر است و داده های حافظه اصلی به روز نیستند. سایر کش ها کپی این داده ها را ندارند.
- ✓ **Shared** : خط cache در حافظه پنهان وجود دارد و اصلاح نشده است. سایر پردازنده ها نیز ممکن است یک کپی از همان داده ها را در حافظه پنهان خود داشته باشند
- ✓ **Invalid** : خط cache معتبر نیست یا حاوی اطلاعات قدیمی است. قبل از استفاده باید از حافظه اصلی یا حافظه پنهان دیگری بارگیری شود.

قفل بلیت

قفل بلیت نوعی مکانیزم همگام سازی است که برای اجرای دسترسی منصفانه به یک منبع مشترک در یک محیط چند رشته ای یا چند هسته ای استفاده می شود. در سیستم قفل بلیت، به هر ریسبه یا هسته زمانی که تلاش می کند قفل را بدست آورد، یک شماره بلیت داده می شود. ریسبه ها باید منتظر بمانند تا شماره بلیت آنها فراخوانی شود تا بتوانند قفل را بدست آورند. این یک اولویت بندی است که می تواند منجر به هماهنگی بین هسته ها شود. جزئیات پیاده سازی قفل بلیت ممکن است شامل عملیات اتمی باشد که توسط پروتکل های انسجام cache پشتیبانی می شود. به عنوان مثال، تخصیص شماره بلیت یا عملیات مقایسه و افزایش ممکن است شامل دستورالعمل های اتمی باشد که هماهنگی مناسب بین هسته ها را تضمین می کند.

تعریف داده های مختص هر هسته در لینوکس

برای تعریف داده های خاص برای هر هسته در لینوکس در طول زمان کامپایل، باید از ویژگی متغیرهای هر CPU استفاده کرد. متغیرهای هر CPU راهی برای تخصیص و دسترسی به داده هایی هستند که مختص هر هسته پردازنده در یک سیستم چند پردازنده ای است. این می تواند عملکرد را بهبود بخشد و با حذف نیاز به عملیات قفل یا عملگر های اتمی از شرایط مسابقه جلوگیری کند.

صف اولویت برای قفل

این الگوریتم به این دلیل که همواره پردازش با pid بیشتر اولویت بالاتری دارد ، ممکن است پردازنده را نتواند به پردازش های با اولویت کمتر تخصیص دهد و لذا این پردازش ها ممکن است دچار گرسنگی شوند.

برای حل این مشکل یا باید سیاست اولویت دهی را تغییر دهیم. مثلاً با فرض این که پردازش های جدیدی که ایجاد می شوند دارای pid بیشتری هستند میتوان اولویت بالاتر را به پردازش با pid کمتر داد و همچنین سیاست aging را هم دخیل کرد تا پردازش های با اولویت کمتر در گذر زمان و در صورت ایجاد پردازش های جدید اولویتشان بالاتر برود و دچار گرسنگی نشوند.

در روش قفل بلیت به جهت این که پردازش ها در صف اولویت هستند و به ترتیب نوبت دهی می شوند ما دچار مشکل گرسنگی نمی شویم اما در روشی که از طریق شماره هر پردازش به آن اولویت می دهیم با فرض آنکه پردازش های جدید دارای pid بیشتری باشند ، ممکن است با ایجاد پردازش جدید اولویت آن بالاتر برود (مثلاً از 63 به 0) و پردازش های قبلی موجود در صف اولویتشان پایین بیاید و نتوانند به منابع دسترسی پیدا کنند. لذا روش بالا ممکن است باعث گرسنگی پردازش ها شود.

منابع :

(1) https://en.wikipedia.org/wiki/Ticket_lock#Related_work

(2) Book-rev11-chapter 4

(3) Refrence-book-chapter6 & 7