گزارش کار پروژه سوم درس آزمایشگاه سیستم عامل

آیلین جمالی ۸۱۰۸۹۵۰۲۶

فاطمه حقیقی ۸۱۰۱۹۵۳۸۵

۱. وجود وقفه باعث همزمانی وقایع حتی در سیستم های تک هسته ای می شود ، اگر وقفه ها غیرفعال نشوند، کدی در قسمت کرنل در حال اجرا است هر لحظه ممکن است به علت رخ داد وقفه متوقف شود و interrupt handler اجرا شود. مثلا حالتی را در نظر بگیریم که writer A قفل t را در اختیار دارد و وقفه ای برای اجرای writer B اتفاق می افتد ، writer B می خواهد قفل t را بگیرد ولی قفل آزاد نیست و صبر می کند تا آزاد شود. در این شرایط قفل t را فقط writer A می تواند آزاد کند و writer A نمی تواند به اجرای خود ادامه دهد تا قبل از اینکه writer B کارش تمام شود در نتیجه processor و حتی کل سیستم دچار deadlock می شود.

در سیستم عامل xv6غیر فعال کردن وقفه ها توسط صدا زدن تابع ()cli صورت می گیرد و توسط صدا زدن تابع ()sti این وقفه ها که متوقف شدند مجددا فعال می شوند . ولی برای فعال کردن و یا غیرفعال کردن وقفه ها به طور مستقیم این توابع را صدا نمی زنیم. کرنل xv6 برای غیرفعال کردن وقفه ها تابع ()pushcli را صدا می زند که این تابع ، تابع ()cli را صدا می زند، علاوه بر این تابع ()pushcli تعداد دفعاتی که تا کنون push صدا زده شده است را نگه می دارد.

هنگامی که spin lock آزاد می شود، تابع ()popcli از تعداد push صدا زده شده که تعدادشان در تابع ()pushcli محاسبه می شد، یکی کم می کند و با صدا کردن تابع ()sti درون خود وقفه های غیرفعال شده را فعال می کند ولی در صورتی این غیرفعال کردن اتفاق می افتد که تعداد push صفر شود. به عبارت دیگر توابع ()cli و ()pushcli تفاوتشان در این است که تابع ()pushcli تعداد push که تا کنون صدا زده شده را می شمارد و تفاوت توابع ()sti و ()popcli در این است که به ازای هر بار صدا زده شدن تابع ()popcliاز تعداد push یکی کم می شود و در صورتی که این مقدار صفر باشد تابع ()sti صدا زده می شود و وقفه ها فعال می شوند ولی تابع ()sti در هر صورت وقفه ها را غیرفعال می کند. اگر از توابع ()pushcli و ()popcli استفاده نکنیم و مستقیم توابع ()cli و ()sti را صدا بزنیم تعداد دفعاتی که یک وقفه غیرفعال شده را نمی دانیم و عملیات فعال کردن وقفه در زمان مناسبی صورت نمی گیرد.

۲. زمانی که می‌خواهیم برای مدت طولانی یک فرایند یا ریسمان را قفل کنیم، ازSleeplock استفاده می‌کنیم. به عنوان مثال می‌دانیم برای نوشتن و خواندن از دیسک، به دلیل اینکه دسترسی به حافظه کند است، زمان زیادی نیاز است. Sleeplock، یک استراکت است که فیلدی در آن وجود دارد به نام locked که مشخص می‌کند که آیا یک قفل را کسی گرفته است یا خیر. دو تابع acquiresleep و releasesleep، برای تعامل میان دو پردازه صورت می‌گیرد. اگر بخواهیم قفلی را بگیریم از تابع اول و برای رها کردن قفل، از تابع دوم استفاده می‌کنیم. برای اینکه توابع acquire وsleep خودشان با یکدیگر اجرا نشوند، باید با استفاده از راه حل‌های صحیح critical section از این توابع محافظت کنیم تا شرایط مسابقه رخ ندهد.

در تابع acquiresleep، ابتدا قفل sleeplock گرفته می‌شود. چون به دلایلی که در بالا گفته شد باید مطمئن باشیم که فرایند دیگری در حال اجرای این تابع نباشد. که این قفل یک spinlock است. سپس سعی می‌کند قفل sleeplock را بگیرد. اگر فرایند دیگری نباشد که قفل را گرفته باشد، این تابع به کار خود ادامه می‌دهد و می‌تواند وارد ناحیه‌ی بحرانی خود بشود. اما اگر فرایند دیگری وجود داشته باشد که قبلا قفل را گرفته باشد این فرایند به جای اینکه Busywait شود، به

sleep می‌رود. به این معنی که پردازنده از آن گرفته می‌شود و stateهای آن فرایند ذخیره می‌شود که در موقع بازیابی، بدانیم از کجا شروع به کار کنیم. حال اگر فرایندی که قفل را در اختیار داشت توسط تابع releasesleep آن را رها کند، مقدار locked و pid را صفر می‌کند و تمام توابعی که روی این قفل sleep شده بودند را wakeup می‌کند. حال یکی از این فرایندهای بیدار شده قفل را می‌گیرد و مقدار locked را یک می‌کند. به همین دلیل دوباره بقیه به خواب می‌روند تا زمانی که فرایند دوم هم release کند.

در مثال تولیدکننده/مصرف کننده، ممکن است هر فرایند مدت زمان زیادی نیاز داشته باشد تا کارش را انجام دهد. چون در این مثال عملیات I/O داریم و می‌دانیم که این عملیات نیازمند دسترسی به حافظه است و این کار، کند است و ممکن است خیلی زمان ببرد تا کارش تمام شود. چون تولید کننده، itemهایی تولید می‌کند و مصرف‌کننده از آن استفاده می‌کند که این کار توسط I/O انجام می‌شود. حال اگر در این شرایط بخواهیم از قفل‌های چرخشی استفاده کنیم، به این دلیل که در ابتدای این قفل تمام وقفه‌ها را خاموش می‌شوند، performance برنامه بسیار پایین می‌آیند. می‌دانیم که سیستم توسط interrupt از وقایع اطراف خود با خبر می‌شود. اگر مدت زمان زیادی این وقفه‌ها را خاموش کنیم، مشکلات جدی برای برنامه به وجود می‌آید. به عنوان مثال اگر تایم سیستم با وقفه کار کند، با خاموش کردن وقفه‌ها مشکلات در زمان‌بندی و... ایجاد می‌شود. به همین دلیل نمی‌توانیم در مثال تولیدکننده/مصرف‌کننده از قفل‌های چرخشی استفاده کنیم.

۳. الف) حالت های یک process در سیستم عامل xv6 عبارت است از:

UNUSED , EMBRYO , SLEEPING , RUNNABLE , RUNNING , ZOMBIE

بعد از که main زیرسیستم ها و device ها را مقداردهی اولیه می کند، اولین process را با صدا زدن userinit می سازد. اولین کاری که userinit انجام می دهد این است که allocproc را صدا می زند، کار allocproc این است که در process table به این process جدید یک جا اختصاص می دهد و برخی از قسمت های process state را که برای اجرا شدن kernel thread آن لازم است را مقداردهی اولیه می کند. Allocproc برای هر process جدید صدا زده می شود ولی userinit فقط برای اولین process صدا زده می شود. Allocproc ، proc table را scan می کند تا جایی که حالت آن unused است را پیدا کند. به عبارت دیگر هنگامی که process ،terminate می شود state آن در ptable به UNUSED تغییر می کند.

هنگامی که جایگاهی با این حالت پیدا کرد ، allocpro ، process را به حالت EMBRYO درمی آورد در این حالت process به عنوان process استفاده شده علامت گذاری می کند و یک pid جدید به آن اختصاص می دهد. سپس تلاش می کند به kernel thread این processs یک kernel stack اختصاص دهد، در صورت عدم موفقیت این فرآیند allocproc مجددا state ، process را به UNUSED تغییر می دهد.

یک process زمانی وارد حالت sleeping می شود که منابعی که نیاز دارد در حال حاضر در دسترس نباشد.در این شرایط یا خود process وارد حالت sleeping می شود یا kernel آن را در حالت sleeping قرار می دهد. در این حالت process

دسترسی خود را به CPU از دست می دهد .هنگامی که state یک process ، sleeping است ینی برنامه موقتا تعلیق شده و اطلاعات فعلی آن در جایی ذخیره شده . در این شرایط sched ، processor آزاد می کند. Process که در حالت sleeping قرار دارد قفل روی یک شرط یا ptable.lock یا هر دو را نگه می دارد، به عبارت دیگر قبل از آنکه process به بررسی درست بودن آن شرط برسد قفل را می گیرد.

در سیستم عامل xv6 بعد از آنکه main ،userinit را صدا می کند ، mpmain برای آنکه اجرای processها را شروع کند scheduler را صدا می زند، scheduler نیزحالت هر process را به RUNNABLE تغییر می دهد.

هنگامی که یک process فرزند تمام و exit شود، process در لحظه از بین نمی رود و وارد حالت zombie می شود تا زمانی که پدر آن process ، سیستم کال wait را صدا بزند. به عبارت دیگر پدر هر process مسئول آزاد کردن حافظه ی process فرزند پس از terminateشدن آن است. اگر process پدر قبل از process فرزند terminate شود، init process نقش پدر را برای process فرزند به عهده می گیرد و پس از exit شدن process برای آن process ، wait می کند.

ب) یک process برای اینکه CPU را رها کند تابع sched را صدا می زند. تابع sched یک contex switch را راه اندازی می کند و هنگامی که process ،در یه زمان دیگه switch back می کنه در sched برمی گرده به ادامه اجراش که قبل contex switch داشت انجام می داد. به عبارت دیگر صدا زدن تابع sched اجرای یک process را به طور موقت freez می کند. در ۳ حالت process میخواهد CPU را رها کند. ۱- وقتی که مدت زمانی که process ، CPU را در اختیار داشته زیاد بوده و یک timer interrupt اتفاق می افتد و CPU از process گرفته می شود. ۲- هنگامی که کار process تمام می شود، process برای آخرین بار sched را صدا می زند تا CPU را آزاد کند. ۳- هنگامی که process بخاطر یک اتفاق block می شود. تابع sched شرایط متفاوتی را چک می کند و swtch را صدا می زند برای switch کردن به scheduler thread.

۴. یکی از مزیت‌های اصلی ticketlock این است که starvation به مدت خیلی طولانی رخ نمی‌دهد و بالاخره به همه تایم برای اجرا می‌رسد و عدالت در مورد همه‌ی ریسمان‌ها رعایت می‌شود. چون در منطقی که پشت این نوع قفل وجود دارد، ترتیب ورود ریسمان‌ها FIFO است، این fairness رعایت می‌شود.

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

قفل معادل sleeplock در هسته‌ی xv6، قفل mutex در هسته‌ی لینوکس است. پیاده‌سازی این قفل در mutex.h نوشته شده‌است. در این قفل اگر کسی از قبل قفل را گرفته باشد، فرایندی که می‌خواهد قفل را بگیرد suspend می‌شود. suspend شدن به این معنی که cpu از آن گرفته می‌شود و stateهایش ذخیره می‌شود تا زمانی که mutex آزاد شود، بیدار شود و از همان جایی که قطع شده بود، به کارش ادامه دهد. یکی دیگر از ویژگی‌های این قفل این است که نمی‌توانیم به تعداد زیاد unlock کنیم. همچنین آن فرایندی که قفل را گرفته است و mutex دست او است، نمی‌تواند exit شود.

۵. یکی از مزیت‌های اصلی ticketlock این است که starvation به مدت خیلی طولانی رخ نمی‌دهد و بالاخره به همه تایم برای اجرا می‌رسد و عدالت در مورد همه‌ی ریسمان‌ها رعایت می‌شود. چون در منطقی که پشت این نوع قفل وجود دارد، ترتیب ورود ریسمان‌ها FIFO است، این fairness رعایت می‌شود.

در تابع sleep خود xv6، از spinlock استفاده شده است که می‌دانیم این قفل قابلیت به خواب رفتن ندارد و اگر پردازه‌ای موفق به دریافت قفل نشود، busywait می‌شود. بنابراین تابع ticketsleep خودمان را پیاده‌سازی کردیم.

```
void
ticketsleep(void *chan)
{

  struct proc *p = myproc();

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

  acquire(&ptable.lock);
  popcli();

  // Go to sleep.
  p->chan = chan;
  p->state = SLEEPING;
  cprintf("process %d slept\n" , myproc()->pid);

  sched();


  // Tidy up.
  p->chan = 0;
  pushcli();
  release(&ptable.lock);
}
```

```
void
initticketlock_func(struct ticketlock* tl)
{
    tl->now_running = 0;
    tl->next_ticket = 0;
    cprintf("ticketlock successfully initialized for process: %d \n", myproc()->pid );
}

void
acquireticketlock(struct ticketlock* tl)
{
    pushcli();
    int my_ticket = fetch_and_add(&(tl->next_ticket),1);
    cprintf("ticket number for process %d is : %d " ,myproc()->pid , my_ticket);
    cprintf("now running : %d , next ticket : %d ,for process: %d \n", tl->now_running ,
            tl->next_ticket,myproc()->pid);
    cprintf("now_running : %d " ,tl->now_running);
    cprintf("my_ticket : %d " ,my_ticket)   ;

    while(tl->now_running != my_ticket)
    {
        ticketsleep(tl);
    }
    cprintf("acquire for process: %d \n" , myproc()->pid);
    popcli();
}

void
releaseticketlock(struct ticketlock* tl)
{
    pushcli();
    fetch_and_add(&(tl->now_running), 1);
    cprintf("releasse for process : %d\n", myproc()->pid);
    wakeup(tl);
    popcli();
}
```

۶.

```
////////////////////////////////////////////////////////////
static inline int fetch_and_add(int* variable, int value)
{
    __asm__ volatile("lock; xaddl %0, %1"
      : "+r" (value), "+m" (*variable) // input+output
      : // No input-only
      : "memory"
    );
    return value;
}
////////////////////////////////////////////////////////////
```

این دستور اسمبلی میان کدهای به زبان c نوشته شده‌اند. بنابراین inline باید باشد. دو تا آرگومان داریم. اولی variableای است که می‌خواهیم مقدار آن را زیاد کنیم. دومین آرگومان مقداری است که می‌خواهیم به variable اضافه کنیم. در این کد

از دستور lock استفاده شده است. زیرا می‌خواهیم که این دستور قفل شود و کسی نتواند دیگر وسط اجرای این تابع اینتراپت بدهد و پردازنده از این تابع گرفته شود. در نتیجه منجر به شرایط مسابقه نمی‌شود. با دستور xaddl می‌توانیم دو متغیر را با هم جمع کنیم. همچنین ما *variable را می‌گیریم تا زمانی که value را با variable جمع می‌کنیم مقدار خود variable هم تغییر کند. درواقع پوینتر به variable است و مقدار آن را تغییر می‌دهد. نکته‌ای که در این قسمت وجود دارد این است که مقداری که از این تابع برمی‌گردد مقدار variable قبلی است و جمع شده‌ی آن با مقدار variable نیست.اما خود variable بعد از صدا شدن این تابع تغییر می‌کند. کامپایلر گاهی روند اجرای دستورات را عوض می‌کند و ممکن است reorder کند و دستورات را جابه‌جا اجرا کند. با زدن دستور memory از این اتفاق جلوگیری می‌کنیم و می‌گوییم که دستورات را با همین ترتیبی که خودمان نوشته‌ایم اجرا کند.