اعضا : سید احمد رکنی حسینی ۸۱۰۱۰۰۱۵۴ محمدعلی شاهین فر ۸۱۰۱۰۰۱۶۹ امیرحسین راحتی ۸۱۰۱۰۰۱۴۴

سوال اول :

وقفه ها می توانند حتی در یک پردازنده باعث concurrency شوند. اگر وقفه ها فعال باشند، کد هسته را می توان در هر لحظه متوقف کرد تا به جای آن یک هندلر وقفه اجرا شود.

ممکن است یک برنامه چندین قفل را نگه دارد و با تکرار این کار توسط یک کد مشابه دیگر ، ممکن است به دلیل وابستگی منابع مشترک ، هردو نتوانند قفل هایی که دیگری نیاز دارد را آزاد کنند و deadlock رخ دهد.

سوال دوم :

تابع cli برای غیرفعال کردن وقفه‌ها و تابع sti برای فعال کردن وقفه‌ها استفاده می‌شود. توابع pushcli و popcli حین اجرا به ترتیب، در خود توابع cli و sti را صدا میزنند. با این تفاوت که یک متغیر به نام ncli وجود دارد که در pushcli مقدار آن یکی زیاد میشود و در popcli یکی کم میشود. هروقت این متغیر صفر باشد وقفه ها فعال میشوند و در غیر اینصورت غیر فعال هستند. اگر مثلا در برنامه ای همزمان از دو قفل استفاده کردیم و یکی را آزاد کردیم ، وقفه ها فعال نشوند و فرآیند آزاد سازی دیگری از حالت atomic خارج نشود.

سوال سوم :

این روش باعث busy waiting می‌شود. busy waiting در سیستم‌های چند هسته ای باعث هدر رفتن زمان پردازنده می شود؛ اما در سیستم‌های تک هسته ای این موضوع در بدترین حالت می‌تواند منجر به deadlock شود. مثلا حالتی که در آن یک پردازه قفلی را در اختیار می‌گیرد؛ سپس پردازه‌ای دیگر سعی می‌کند قفل را به بدست بیاورد؛ در این صورت پردازه دوم هیچگاه از حلقه خارج نشده و پردازه‌های دیگر زمان‌بندی نمی‌شوند.

سوال چهارم :

این دستور دو حالت ۳۲ و ۶۴ بیتی دارد . منظور از AMO ، atomic memory operation است . این دستور عملیات جایجایی مقداری در حافظه و رجیستر را به صورت atomic انجام میدهد. به عنوان مثال از این دستور میتوان در بازیابی مقدار قبلی lock استفاده کرد. زیرا اطمینان داریم که در میان عملیات جابجایی وقفه ای صورت نمیگیرد و مقدار قفل به مقدار نامطلوبی تغییر نمیکند

سوال پنجم :

در ساختار این قفل ، یک متغیر lock اصلی داریم و یک spinlock برای حفاظت از کل اجزای سیستم وجود دارد.

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.

رفتار تابع sleep به این شکل است که از یک spinlock 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.

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.

سوال ششم :

unused: خانه های لیست پردازه ها ممکن است خالی باشند و پردازه ای در آن ها وجود نداشته باشد . آن ها استیت unused هستند.

Embryo: هنگام ایجاد یک پردازه جدید،تابع allocproc یک عضوunused از لیست پردازه ها را انتخاب میکند وآن را به این استیت تغییر میدهد

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: این پردازه آماده اجرا است و زمانبند میتواند پردازنده را به ان اختصاص دهد . اگر این اتفاق بیوفتد به استیت running تبدیل میشود.

Running: این حالت یعنی پردازه درحال اجرا توسط cpu 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 state اش از RUNNING عوض شده باشد و قفل ptable را داشته باشد). در تابع وقفه ها فعال شده و بعد از آن تابع scheduler اجرا میشود.

سوال هفتم :

میتوانیم با استفاده از pid قرار داده شده در struct استفاده کنیم . به این صورت که هرگاه پردازه ای خواست قفل را آزاد کند ، در تابع چک شود که آیا این پردازه همان پردازه ای است که قفل را فعال کرده یا خیر.

در لینوکس ، معادل این قفل در فایل mutex.h تعریف میشود که در struct آن یک متغیر owner وجود دارد که آیدی پردازه ای که قفل را فعال کرده ذخیره شود و حین آزادسازی قفل ، آنرا چک میکند.

سوال هشتم :

الگوریتم هایی هستند که به صورت همروند انجام میشوند تا انجام عملیات ها در سیستم عامل بدون نیاز به روش ها همگام سازی blocking انجام شود. معمولا از این الگوریتم ها در multithreading استفاده میشود.عملیات ها در این حالت به صورت atomic انجام میشود

یکی از data structure های معروف در آن ها ، concurrent stack است که اجازه push و pop در استک توسط thread ها بدون نیاز به دریافت lock را میدهد.

این کار باعث افزایش performance میشود چون نیاز به قفل نداریم. همچنین منجر به رخ دادن بن بست نمیشود (مشابه بن بست در سوال ۳) Resilience در این روش بالاتر است . اگر برنامه ای در حین اجرا قفلی در اختیار داشته باشد و بدون ازاد کردن آن crash کند ، همه پردازه هایی که از آن منبع قفل شده استفاده میکردند از کار خواهند افتاد . در حالی که در روش lock free این اتفاق نخواهد افتاد.

از طرفی این الگوریتم ها پیچیده هستند و امکان پیاده سازی در هر حالتی و هر سیستمی را ندارند. زیرا ممکن است پشتیبانی سخت افزاری هم نیاز باشد .

پیاده سازی متغیر های مختص هر هسته پردازنده

الف) برای حل این مشکل در ساده ترین حالت ، برای دیتا هایی که به همزمان در چند هسته وجود دارند ، یک یا چند بیت برای ذخیره استیت آن دیتا در حافظه نهان آن پردازه در نظر میگیریم که تعیین میکند دیتای فعلی در حافظه نهان دیگر هسته ها با دیتای فعلی یا دیتای داخل حافظه نهان های مشترک یکسان است یا خیر . که یک کنترل کننده برای نظارت بر این دیتا ها در بیت هسته ها نیاز است که میتواند با پروتکل های مختلف مثل MSI , MSOI , MESI , MOESI این مشکل را حل کند.

ب) میتوانیم حافظه مسیر cache پردازنده ها را یکسان در نظر بگیریم . هر دفعه که پردازنده ای میخواهد روی cache بنویسد ، در همان لحظه فقط خودش بتواند بنویسد و به صورت انحصاری این کار را انجام دهد . سپس تغییرات را روی بقیه cache ها اعمال میکنیم . این انحصار را میتوان بوسیله ticketLock پیاده سازی کرد تا در لحظه فقط یک پردازنده بتواند cache را تغییر دهد

```
void enter() {
    const auto ticket = next_ticket.fetch_add(1, std::memory_order_relaxed);

    while (now_serving.load(std::memory_order_acquire) != ticket) {
        spin_wait();
    }
}
```

ج) میتوانیم متغیر های مربوط به هر هسته را به صورت DEFINE_PER_CPU(type , name) تعریف کرد . Type همان متغیر مربوط و name اسم پردازنده ای است که قرار است مقدار دهی شود .

همچنین میتوان با توابع put_cpu_var() و get_cpu_var() به مقادیر دسترسی داشته باشیم و تغییراتی را اعمال کنیم.

پیاده سازی متغیرهای مختص هر هسته پردازنده

یک متغیر **int** جدید به ساختمان داده **CPU** اضافه میشود که تعداد **systemcall** های اجرا شده در آن ذخیره میشود.

تغییراتی در تابع **()syscall** به صورت زیر انجام میشود که به ازای هر فراخوانی **systemcall** روی پردازنده ، متغیر مربوط در ساختمان داده پردازنده افزایش می یابد

```
struct cpu {
  uchar apicid;
  struct context *scheduler;
  struct taskstate ts;
  struct segdesc gdt[NSEGS];
  volatile uint started;
  int ncli;
  int intena;
  struct proc *proc;
  uint sys_call_numbers;
};
```

```
syscall(void)
{
  int num;
  struct proc *curproc = myproc();
  pushcli();
  struct cpu *my_cpu = mycpu();
  popcli();

  num = curproc->tf->eax;
  my_cpu->sys_call_numbers++;
```

در این قسمت ، **systemcall** زیر اضافه می شود . این **systemcall** مقدار فعلی متغیر **sys_call_numbers** را **return** می کند.

```
int sys_call_numbers(){
  struct cpu *my_cpu = mycpu();
  uint sys_call_numbers
    = my_cpu->sys_call_numbers;
  return sys_call_numbers;
}
```

برنامه سطح کاربر زیر هم اضافه میشود که در آن تعدادی پردازه ایجاد میشود و هرکدام تعدادی **systemcall** اجرا میکنند.

```
int main()
{
    for (int i = 0; i < 6; i++)
    {
        int pid = fork();
        if (pid > 0)
            continue;
        if (pid == 0)
        {
            sleep(1000);
            for (int j = 0; j < 500 ; j++)
```

```
            {
                long x = 0;
                for (long k = 0; k < 10000000000; k++)
                    x++;
            }
            exit();
        }
    }
    for(int i = 0 ; i < 3 ; i++)
        wait();
    exit();
}
```

راه حلی برای مشکل starvation در Priority Lock

این روش میتواند سبب **starvation** شود . به این صورت که هر دفعه پردازه ای با شناسه بزرگتر وارد شود و قفل را بخواهد و پردازه با شناسه کوچکتر به دلیل اولویت پایینتر نتواند قفل را در اختیار بگیرد.

یک راه حل برای این مشکل میتواند استفاده از **aging** باشد . به این صورت که هرگاه پردازه ای که در صف قرار دارد به خاطر وارد شدن پردازه پر اولویت تر ، در صف عقل تر رفت (اولویتش کمتر شد) به **age** آن یکی اضافه شود و مکانیسم تعیین اولویت در صف به گونه ای باشد که **age** هم با ضریب خاصی بتواند در روند اولویت بندی سهیم باشد.

در **ticketLock** به هرکس که میخواهد وارد صف شود یک تیکت یا شماره میدهیم و هرکس خارج شود پردازه بعدی حق دریافت قفل و ورود به ناحیه بحرانی را دارد . در این حالت صف ، به صورت **FIFO** است اما در حالت صف اولویت ، بر اساس ویژگی خاصی از پردازه اولویت بندی انجام میشود

پیاده سازی priority lock

ابتدا ساختمان داده های مربوط به پیاده سازی قفل اولویت به صورت زیر تعریف میشود:

pqueue یک لیست پیوندی از پردازه های منتظر دریافت قفل است و proritylock متغیر های مربوط به قفل را نگه میدارد

```
You, 2 days ago | 1 author (You)
struct pqueue
{
    struct proc* proc;
    struct pqueue* next;
};
```

```
struct proritylock
{
    unsigned int locked;       //
    struct spinlock lk; // spinloc
    // For debugging:
    char *name;        // Name of

    struct pqueue *priority_queue;
    int pid;
};
```

توابع مربوط به نگه داری قفل ، آزاد سازی قفل و دریافت قفل به صورت زیر تعریف میشوند .

یک تابع وظیفه پیدا کردن بالا ترین اولویت دریافت قفل بین پردازه های درخواست کننده را دارد .

```
struct proc* get_highest_priority(struct pqueue *queue){
    if (queue == 0){
        return 0;
    }
    struct proc* ans = queue->proc;
    struct pqueue* temp = queue->next;
    kfree((char *)queue);
    queue = temp;

    return ans;
}
```

یک تابع وظیفه مانیتور کردن فرآیند تخصیص قفل را دارد و هر دفعه پردازه ای که قفل را در اختیار دارد نمایش میدهد

تابع add_to_priority_queue() هم هردفعه که پردازه ای جدید قفل را درخواست میکند آن را به صف انتظار اضافه میکند .

```
void
releasepriority(struct proritylock *lk)
{
  acquire(&lk->lk);
  struct proc* proc = get_highest_priority(lk->priority
  lk->locked = 0;
  lk->pid = 0;
  if (proc != 0){
    wakeup(proc);
  }
  release(&lk->lk);
}
```

```
int
holdingpriority(struct proritylock *lk)
{
  int r;
  acquire(&lk->lk);
  r = lk->locked && (lk->pid == myproc()->pid);
  release(&lk->lk);
  return r;
}
```

```
void monitor_pqueue(struct proritylock *lk){

  struct pqueue* head = lk->priority_queue;
  cprintf("pid that uses lock : %d \n", lk->pid);
  cprintf("priority lock :\n");
  if (head == 0){
    return;
  }
  int i = 1;
  while(1){
    if (head == 0){
      break;
    }
    cprintf("%d- pid : %d\n", i, head->proc->pid);
    head = head->next;
  }
}
```

```
void add_to_priority_list(struct pqueue *queue, struct proc* proc){
    struct pqueue* current = (struct pqueue*)kalloc();
    if (current == 0){
        panic("can't allocate memory for priority queue");
    }
    current->proc = proc;
    struct pqueue* head = queue;
    if (head == 0){
        current->next = 0;
        queue = current;
        return;
    }
    if (head->proc->pid > current->proc->pid){
      current->next = head;
      head->next = 0;
      queue = current;
      return;      You, 3 minutes ago • Uncommitted changes
    }
    while(1){
        if (head->next == 0){
            current->next = 0;
            head->next = current;
            break;
        }
        if (head->next->proc->pid < current->proc->pid){
            head = head->next;
        }
        else{
            struct pqueue* temp = head->next;
            head->next = current;
            current->next = temp;
            break;
        }
    }
}
```

Wait() در این حالت به صورت زیر پیاده سازی میشود . در این تابع ، اگر قفل به پردازه اختصاص داده نشود پردازه به **sleep()** میرود و اجرا مشکل **busy waiting** بوجود نمی آید .

```
void acq_wait(struct prioritylock *lock){
  for (int i = 0; i < 5; i++)
  {
    acquirepriority(lock);
    cprintf("acquire %d\n", lock->pid);
    acquire(&tickslock);
    uint ticks0 = ticks;
    while(ticks - ticks0 < 10000){
      if(myproc()->killed){
        release(&tickslock);
        return;
      }
      sleep(&ticks, &tickslock);
    }
    release(&tickslock);

    cprintf("release %d\n", lock->pid);
    monitor_pqueue(lock);
    releasepriority(lock);
  }
```

```
void test_priority_lock(){
    const int NUMBER = 7;
    initprioritylock(&prilock, "test");
    for (int i = 0; i < NUMBER; i++)
    {
        int pid = fork();
        if (pid > 0)
            continue;
        if (pid == 0)
        {
            acq_wait(&prilock);
            exit();
        }
    }
    for(int i = 0 ; i < NUMBER; i++)
        wait();
    exit();
}
```

در فراخوانی سیستمی **test_priority_lock()** تعدادی پردازه ایجاد میشود که از این **systemcall** در برنامه سطح کاربر استفاده میشود دلیل این کار این است که بتوان در لحظه تغییرات را دید و لاگ های موردنیاز را ایجاد کرد . چون در سطح کاربر به اطلاعات قفل دسترسی نداریم باید به صورت **systemcall** پیاده سازی شود