HW4, OS

Dr Zali

Dey, 1403

Sepehr Ebadi

9933243

۱)

الف)

شرط Progress در راه‌حل صحیح مشکل بخش بحرانی (Critical Section Problem) بیان می‌کند که:

اگر هیچ فرآیندی در بخش بحرانی نباشد و تعدادی از فرآیندها تمایل به ورود به بخش بحرانی داشته باشند، فقط آن فرآیندهایی که آماده ورود به بخش بحرانی هستند باید در تصمیم‌گیری برای ورود به بخش بحرانی دخالت داشته باشندو این تصمیم‌گیری نباید تحت تأثیر فرآیندهایی باشد که تمایل به ورود به بخش بحرانی ندارند.

شرط Progress تضمین می‌کند که هیچ فرآیندی که آماده ورود به بخش بحرانی است، نباید به طور نامحدود منتظر بماند و انتخاب فرآیندی که وارد بخش بحرانی می‌شود باید در مدت زمان محدودی صورت بگیرد.

ب)

بله، گرسنگی می‌تواند شرط Progress را نقض کند به دلیل اینکه گرسنگی زمانی رخ می‌دهد که یک یا چند فرآیند علی‌رغم آماده بودن برای ورود به بخش بحرانی، به دلیل اولویت‌های پایین‌تر یا سیاست‌های نامناسب زمان‌بندی به طور مداوم از ورود به بخش بحرانی بازداشته می‌شوند. در این شرایط، شرط Progress نقض می‌شود زیرا:

فرآیندهایی که آماده ورود به بخش بحرانی هستند، نباید به‌طور نامحدود منتظر بمانند و همچنین اگر یک یا چند فرآیند به دلیل گرسنگی قادر به پیشرفت (Progress) نباشند، سیستم دیگر تضمین نمی‌کند که فرآیندهای منتظر در نهایت به بخش بحرانی دسترسی پیدا کنند.

ج)

شرط Mutual Exclusion می گوید فقط یک فرآیند می‌تواند در یک زمان مشخص وارد بخش بحرانی شود.
در این کد زمانی که فرآیند P0 قصد ورود به بخش بحرانی دارد، مقدار flag[0] برابر true می‌شود و تا زمانی که flag[1] نیز برابر true باشد، P0 در حلقه منتظر می‌ماند. و اگر P1 بخواهد وارد بخش بحرانی شود، در صورتی که flag[0] برابر true باشد، P1 نیز در حلقه منتظر می‌ماند.
پس این شرط برقرار است زیرا هیچگاه هر دو فرآیند به طور همزمان وارد بخش بحرانی نمی‌شوند.

شرط Progress پیشرفت می گوید اگر هیچ فرآیندی در بخش بحرانی نباشد، فرآیندهایی که تمایل به ورود دارند باید بتوانند در زمان محدودی وارد بخش بحرانی شوند. همچنین، فرآیندهایی که تمایلی به ورود ندارند نباید تصمیم‌گیری را تحت تأثیر قرار دهند.
فرض می کنیم هر دو فرآیند بخواهند همزمان وارد بخش بحرانی شوند:

p0 مقدار flag[0] را برابر true می‌کند. وP1مقدار flag[1] را برابر true می‌کند.
هر دو فرآیند در حلقه منتظر می‌مانند.
پس شرط Progress نقض می‌شود زیرا هر دو فرآیند ممکن است در وضعیت انتظار بی‌پایان گیر کنند و نتوانند وارد بخش بحرانی شوند.

شرط Bounded Waiting می گوید هر فرآیند باید بتواند در زمان محدودی وارد بخش بحرانی شود و نباید به طور نامحدود منتظر بماند.
همان‌طور که در شرط Progress توضیح داده شد، اگر هر دو فرآیند به طور همزمان تمایل به ورود به بخش بحرانی داشته باشند، ممکن است هر دو فرآیند در حلقه منتظر بمانند. در این حالت، هیچ تضمینی برای انتظار محدود وجود ندارد.
پس شرط Bounded Waiting نیز نقض می‌شود.

۲)

مقدار flag[i] برای هر فرآیند نشان‌دهنده تمایل آن فرآیند به ورود به بخش بحرانی است. و فرآیند i تنها در صورتی وارد بخش بحرانی می‌شود که مقدار flag[(i+1) % 2] برابر false باشد یا نوبت (turn) به آن فرآیند داده شده باشد.

این شرط به درستی عمل می‌کند و فرآیندها به صورت همزمان نمی‌توانند وارد بخش بحرانی شوند. وشرط Mutual Exclusion برقرار است.

در صورتی که هر دو فرآیند بخواهند وارد بخش بحرانی شوند، مکانیزم متغیر turn تعیین می‌کند که کدام فرآیند وارد شود. اما اگر فرآیند P0 وارد بخش بحرانی شود و سپس به دلایلی متوقف شود (مثلاً در بخش remainder زمان زیادی صرف کند)، فرآیند P1 ممکن است نتواند وارد بخش بحرانی شود. و شرط Progressبه‌طور کامل برقرار نیست.

مکانیزم turn می‌تواند باعث شود که یک فرآیند به دلیل تغییر متوالی نوبت (turn) بین دو فرآیند، وارد بخش بحرانی نشود. به عبارت دیگر، اگر فرآیند P0 در زمان مناسبی از بخش بحرانی خارج نشود، فرآیند P1 ممکن است دچار انتظار طولانی شود.
شرط Bounded Waiting نیز برقرار نیست.

در زیر کد اصلاح شده را می بینید.

```c
# include <stdio.h>
# include <string.h>
# include <stdlib.h>
# include <stdbool.h>

int turn;
bool flag[2];

Tabnine | Edit | Test | Explain | Document | Ask
void proc(int i) {
    while (true) {
        Compute;

        flag[i] = true;
        turn = (i + 1) % 2;

        while (flag[(i + 1) % 2] && turn == (i + 1) % 2);

        // Critical Section
        flag[i] = false;
        // Remainder Section
    }
}

Tabnine | Edit | Test | Explain | Document | Ask
int main() {
    turn = 0;
    flag[0] = false;
    flag[1] = false;

    proc(0) AND proc(1);

    return 0;
}
```

۳)

Spinlock

Spinlock ها قفل‌هایی هستند که نخ (Thread) یا فرآیند هنگام تلاش برای گرفتن قفل، در یک حلقه (Busy Waiting) باقی می‌ماند و به صورت مداوم بررسی می‌کند که آیا قفل آزاد شده است یا خیر. زمانی که زمان انتظار برای دسترسی به قفل کوتاه است، استفاده از Spinlock به دلیل حذف سربار Context Switching مناسب‌تر است.

Spinlock در سیستم‌هایی که چند پردازنده به صورت همزمان کار می‌کنند (SMP) مناسب است، زیرا یک پردازنده دیگر ممکن است قفل را آزاد کند و Context Switch کردن در این شرایط هزینه اضافی ایجاد می‌کند.

اگر کد داخل بخش بحرانی کوتاه و سریع اجرا شود، Spinlockکارآمدتر است.

سناریوهای مناسب:

دسترسی سریع به منابع در محیط‌های پردازشی پرسرعت(High-Performance Computing) . وکدهای همزمان‌سازی در سطح کرنل که نمی‌توانند عملیات مسدودسازی (Blocking) انجام دهند.

Blocking Lock

درBlocking Lock ، اگر نخ یا فرآیند نتواند قفل را به دست آورد، به جای انتظار مشغول، در وضعیت مسدود (Blocked) قرار می‌گیرد و کنترل CPU را آزاد می‌کند تا سایر نخ‌ها یا فرآیندها بتوانند از منابع استفاده کنند.
اگر انتظار برای قفل طولانی باشد، استفاده از Blocking Lock هزینه Context Switching را می‌پذیرد اما منابع CPU را تلف نمی‌کند. و در سیستم‌هایی که فقط یک پردازنده وجود دارد، مسدودسازی کارآمدتر از چرخش مداوم است. و همچنین اگر اجرای کد بخش بحرانی طولانی باشد، Blocking Lock مناسب‌تر است.

سناریوهای مناسب:

سیستم‌های چندوظیفه‌ای با تعداد زیادی نخ. و بخش‌های بحرانی طولانی در برنامه‌هایی با استفاده سنگین از منابع.

به طور خلاصه میتوان گفت :

Spinlock:

زمانی که سرعت بسیار مهم است و انتظار کوتاه است. و در سیستم‌های چندپردازنده‌ای برای کارهای کوتاه‌مدت. و در کرنل‌ها یا بخش‌های حساس که مسدود شدن ممکن نیست.

Blocking Lock:

زمانی که انتظار طولانی‌تر است و جلوگیری از هدر رفت CPU اولویت دارد. و در سیستم‌های تک‌پردازنده یا محیط‌های با تعداد زیادی نخ. و برای بخش‌های بحرانی طولانی.

(۴

انحصار متقابل تضمین می‌کند که در هر لحظه فقط یک فرآیند می‌تواند وارد بخش بحرانی شود. و سمافورها برای مدیریت این شرط استفاده می‌شوند.

اگر عملیات wait() و signal() اتمیک نباشند (یعنی عملیات کاهش یا افزایش مقدار S به طور کامل اجرا نشود قبل از اینکه فرآیند دیگری وارد شود، امکان وقوع شرایط رقابتی (Race Condition) وجود دارد. در نتیجه، دو یا چند فرآیند ممکن است همزمان وارد بخش بحرانی شوند و انحصار متقابل نقض شود.

فرض کنید مقدار اولیه سمافور S=1 است (یک منبع وجود دارد) و دو فرآیند P1 و P2 می‌خواهند وارد بخش بحرانی شوند.

P1:
وارد wait() می‌شود و مقدار S را بررسی می‌کند(S>0) . اما قبل از کاهش مقدار S مرحله S=S−1 ، اجرای آن متوقف می‌شود Context Switch رخ می‌دهد.

P2:
وارد wait() می‌شود و مقدار S را بررسی می‌کند(S>0) .چون هنوز P1 مقدار S را کاهش نداده، S همچنان ۱ است.P2 نیز وارد بخش بحرانی می‌شود.

هر دو فرآیند در بخش بحرانی:
P1 و P2 همزمان وارد بخش بحرانی می‌شوند و مقدار S دوبار کاهش می‌یابد، اما این نقض انحصار متقابل است.

در این سناریو، بررسی و کاهش مقدار S به صورت غیراتمیک انجام شده است. در نتیجه، هر دو فرآیند P1 و P2 به اشتباه تصور کرده‌اند که قفل باز است و وارد بخش بحرانی شده‌اند.

که برای رفع این خطا و مشکل میتوان به این صورت عمل کرد که :

عملیات wait() و signal() به صورت اتمیک (Atomic) اجرا شوند. این کار می‌تواند با استفاده از:
از دستورات خاص مانند Test-and-Set یا Compare-and-Swap استفاده شود. و اجرای عملیات‌های wait() و signal() باید در یک بخش غیرقابل پیش‌امپت اجرا شود.و از مکانیزم‌های دیگری مانند Mutex برای اجرای اتمیک این عملیات‌ها استفاده شود.

۵)

در ابتدا مقدار دهی اولیه به صورت رو به رو است:

```
struct lock {
    int available;
};

struct lock mutex = {0};
```

تابع acquire وظیفه دارد قفل را به دست آورد. اگر قفل در حال حاضر آزاد باشد (یعنی مقدار availableبرابر با صفر باشد)، فرآیند می‌تواند قفل را بگیرد و مقدار available را به ۱ تغییر دهد. این عملیات باید با استفاده از دستور اتمیک compare_and_swap انجام شود.

```
Tabnine | Edit | Test | Explain | Document | Ask
void acquire(struct lock *mutex) {
    while (compare_and_swap(&mutex->available, 0, 1) != 0)
    {

    }
}
```

اگر مقدار برگردانده‌شده غیرصفر باشد (یعنی قفل در حال حاضر توسط فرآیند دیگری گرفته شده)، حلقه ادامه پیدا می‌کند.

تابع release قفل را آزاد می‌کند. این کار با تغییر مقدار available از ۱ به ۰ انجام می‌شود.

```
Tabnine | Edit | Test | Explain | Document | Ask
void release(struct lock *mutex) {
    mutex->available = 0;
}
```

به صورت جمع بندی شده داریم:

```
struct lock {
    int available;
};

struct lock mutex = {0};

Tabnine | Edit | Test | Explain | Document | Ask
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected) {
        *value = new_value;
    }
    return temp;
}

Tabnine | Edit | Test | Explain | Document | Ask
void acquire(struct lock *mutex) {
    while (compare_and_swap(&mutex->available, 0, 1) != 0)
    {

    }
}

Tabnine | Edit | Test | Explain | Document | Ask
void release(struct lock *mutex) {
    mutex->available = 0;
}
```

۶)

دسترسی همزمان تولیدکننده و مصرف‌کننده به متغیر سراسری count ممکن است منجر به نتایج نادرست شود. به عنوان مثال، اگر دو یا چند فرآیند همزمان مقدار count را بخوانند و تغییر دهند، مقدار نهایی count ممکن است با مقدار واقعی سازگار نباشد.
و همچنین اگر فرآیندی قبل از خواب (sleep) بیدار شود، سیگنال بیدار شدن از دست می‌رود. در نتیجه، ممکن است یک فرآیند به طور دائم منتظر بماند، حتی اگر شرط مورد نظر آن برآورده شده باشد.

برای حل این مشکلات می‌توان از متغیرهای شرطی به همراه یک قفل (mutex) استفاده کرد. متغیرهای شرطی به فرآیندها اجازه می‌دهند به صورت امن منتظر وقوع شرایط خاصی باشند.

قفل تضمین می‌کند که فقط یک فرآیند در هر لحظه به متغیرهای مشترک مانند count دسترسی داشته باشد. و متغیر شرطی اجازه می‌دهد فرآیندها به صورت امن منتظر تغییر شرایط شوند (مانند پر شدن یا خالی شدن بافر).

متغیر شرطی تضمین می‌کند که هیچ سیگنالی از دست نمی‌رود. اگر شرط لازم (مثل خالی نبودن بافر) برقرار نباشد، فرآیند منتظر می‌ماند تا شرایط تغییر کند. و با استفاده از قفل، فرآیندها به صورت ایمن به متغیر count دسترسی دارند.

به صورت زیر میتوان این کد را اصلاح کرد:

```c
#define N 10
int count = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;

Tabnine | Edit | Test | Explain | Document | Ask
void *producer(void *arg) {
    while (1) {
        produce_item();

        pthread_mutex_lock(&mutex);

        while (count == N) {
            pthread_cond_wait(&not_full, &mutex);
        }

        enter_item();
        count++;

        pthread_cond_signal(&not_empty);
        pthread_mutex_unlock(&mutex);
    }
}

Tabnine | Edit | Test | Explain | Document | Ask
void *consumer(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);

        while (count == 0) {
            pthread_cond_wait(&not_empty, &mutex);
        }

        remove_item();
        count--;

        pthread_cond_signal(&not_full);
        pthread_mutex_unlock(&mutex);

        consume_item();
    }
}
```

۷)

مقدار اولیه سمافور S=1 .

دستورات P(s) عملیات Wait باعث کاهش مقدار سمافور می‌شوند، و اگر مقدار سمافور به صفر برسد، فرآیند در حالت انتظار قرار می‌گیرد.

دستورات V(s) عملیات Signal مقدار سمافور را افزایش می‌دهند و ممکن است فرآیندهای منتظر را بیدار کنند.

اولویت اجرا: فرآیندی که شماره کمتری دارد (P1 < P2 < P3) اولویت بالاتری در استفاده از سمافور دارد.

زمان (میلی ثانیه)	فرایند	دستور	مقدار سمافور	وضعیت فرایند
0	-	–	۱	P۱ :آزاد، P۲ :آزاد، P۳ :آزاد
100	p1	P(S)	۰	P۱ :آزاد، P۲ :آزاد، P۳ :آزاد
200	p1	P(S)	۰	P۱ :اجرا، P۲ :آزاد، P۳ :آزاد (درخواست مجدد P۱ نادیده گرفته می‌شود، زیرا قبلاً قفل گرفته است)
300	p2	P(S)	انتظار	P۱ :اجرا، P۲ :انتظار، P۳ :آزاد
400	p3	P(S)	انتظار	P۱ :اجرا، P۲ :انتظار، P۳ :انتظار
500	p1	v(S)	۱	P۱ :آزاد، P۲ :اجرا، P۳ :انتظار
600	p2	v(S)	۱	P۱ :آزاد، P۲ :آزاد، P۳ :اجرا
700	p2	P(S)	۰	P۱ :آزاد، P۲ :اجرا، P۳ :آزاد
800	p1	v(S)	۱	P۱ :آزاد، P۲ :آزاد، P۳ :آزاد
900	p1	v(S)	۲	P۱ :آزاد، P۲ :آزاد، P۳ :آزاد

که همانطور که می بینید مقدار نهایی سمافور برابر ۲ است.

و تمام فرایند ها ازاد و بدون انتظار هستند.

۸)

```
semaphore s1=1;
semaphore s2=15;

Tabnine | Edit | Test | Fix | Explain | Document | Ask
procedure Cave_exploration():
    Enter_the_cave();
    Tabnine | Edit | Test | Explain | Document | Ask
    Look_at_the_paintings();
    Tabnine | Edit | Test | Explain | Document | Ask
    Exit_the_cave();

Tabnine | Edit | Test | Explain | Document | Ask
procedure Enter_the_cave():
    wait(s1);
    Tabnine | Edit | Test | Explain | Document | Ask
    wait(s2);
    Tabnine | Edit | Test | Explain | Document | Ask
    signal(s1);

procedure Look_at_the_paintings():


Tabnine | Edit | Test | Explain | Document | Ask
procedure Exit_the_cave():
    wait(s1);
    Tabnine | Edit | Test | Explain | Document | Ask
    signal(s2);
    Tabnine | Edit | Test | Explain | Document | Ask
    signal(s1);
```

(٩

```
Tabnine | Edit | Test | Explain | Document | Ask
writerLock(){
    lock(mutex);
    writer_waiting += 1;
    while(readcount > 0 || writer_present){
        wait(writer_enter, mutex);
    }
    writer_waiting -= 1;
    writer_present = True;
    unlock(mutex);
}

Tabnine | Edit | Test | Explain | Document | Ask
readLock(){
    lock(mutex);
    while(writer_present || writer_waiting > 0){
        wait(reader_enter, mutex);
    }
    readcount += 1;
    unlock(mutex);
}

Tabnine | Edit | Test | Explain | Document | Ask
writeUnlock(){
    lock(mutex);
    writer_present = False;
    if(writer_waiting ==0){
        signal(reader_enter);
    }
    else{
        signal(writer_enter);
    }
    unlock(mutex);
}

Tabnine | Edit | Test | Explain | Document | Ask
readUnlock(){
    lock(mutex);
    readcount -= 1;
    if(readcount == 0)
    {
        signal(writwr_enter);
    }
    unlock(mutex);
}
```

(١٠

```
Tabnine | Edit | Test | Explain | Document | Ask
lock(m);
while(guest_count < N) {
    wait(cv_guest, m);
}
Tabnine | Edit | Test | Explain | Document | Ask
openDoor();
Tabnine | Edit | Test | Explain | Document | Ask
signal(cv_guest);
Tabnine | Edit | Test | Explain | Document | Ask
unlock(m);
```

```
Tabnine | Edit | Test | Explain | Document | Ask
lock(m);
guest_count += 1;
if(guest_count == N){
    signal(cv_host);
}
wait(cv_guest, m)
Tabnine | Edit | Test | Explain | Document | Ask
signal(cv_guest);
Tabnine | Edit | Test | Explain | Document | Ask
unlock(m);
Tabnine | Edit | Test | Explain | Document | Ask
enter();
```