حوری دهش 9821413

سوال 1:

استفاده از روش غیر فعال سازی وقفه برای پیاده سازی اصولی همزمانی در سیستم های چند هسته ای باعث محدودیت هایی و مشکلاتی در عملکرد سیستم میشود برای همین این روش مناسب نمی باشد به عبارت دیگر استفاده از این روش در اینجا باعث اتفاقات زیر میشود:

1. کاهش عملکرد سیستم:

استفاده از روش غیر فعال سازی وقفه باعث میشود که همه وقفه ها متوقف شوند و این کار به معنای توقف همه ی اطلاعات و وظایف هسته ها در هنگام اجرای کد حساس به وقفه است در نتیجه این موضوع باعث افت شدید عملکرد سیستم می شود.

2. عدم هماهنگی و تضاد:

در سیستم های چند هسته ای، هر هسته ممکن است به صورت همزمان و مستقل کار کند. اما غیر فعال سازی وقفه باعث می‌شود که هسته ها نتوانند با یکدیگر هماهنگ شوند و به طور همزمان کار کنند در نتیجه ممکن است منجر به تضاد و افزایش زمان اجرای وظایف شود.

3. از دست رفتن امکان اجرای وظایف همزمان:

در سیستم های چند هسته ای اجرای همزمان وظایف مختلف اهمیت زیادی دارد و اگر از روش غیر فعال سازی وقفه در این سیستم ها استفاده شود امکان اجرای همزمان وظایف در هسته های مختلف کاهش می یابد

4. عدم قابلیت پاسخگویی به وقفه ها:

استفاده از روش غیر فعال سازی وقفه باعث میشود که سیستم قادر نباشد که به وقفه ها پاسخ بدهد. از طرفی وقفه ها مکانیزم مهمی برای اطلاع رسانی و حل مشکلات در سیستم ها هستند و غیر فعال سازی آنها می‌تواند به مشکلات ناپدید شدن وقفه و عدم تشخیص خطاها منجر شود.

سوال 2:

الف)

1. قفل برای مدت کوتاهی نگهداری میشود:

Spin lock چون در این حالت پروسس ها، کوتاه ترین زمان ممکن را در حلقه منتظر می مانند و به جای اینکه به خواب بروند به طور فعال انتظار می کشند

2. قفل برای مدت طولانی نگهداری میشود:

Mutex lock چون در این حالت انتظار در یک حلقه ممکن است بهینه نباشد و به جای آن استفاده از حالت خواب برای پروسس ها که در حالت منتظر قفل هستند، مناسب تر است.

3. یک thread ممکن است هنگامی که قفل را در اختیار دارد به خواب برود:

Mutex lock چون در این حالت انتظار در یک حلقه ممکن است مناسب نباشد و باید از خواب استفاده بشود تا منابع سیستم بهینه تر مدیریت شوند.

ب)

یک upper bound برای استفاده از spin lock در محیطی با context switch به توازن بین دو موضوع مرتبط وابسته است:

1. هزینه Context switch:

اگر هزینه انجام context switch زیاد باشد ممکن است از spin lock استفاده شود یعنی اگر زمان T برای انجام context switch زیاد باشد، حداکثر مقداری مانند T ممکن است مرزی باشد که از آن به عنوان یک upper bound برای استفاده از spin lock در نظر گرفته می شود.

2. هزینه حلقه انتظار در Spin lock:

اگر هزینه اجرای حلقه انتظار در spin lock کمتر از هزینه context switch باشد ممکن است تا زمانی که، زمان انجام حلقه انتظار کمتر از T باشد از spin lock استفاده کرد.

در نتیجه یک upper bound برای استفاده از spin lock در محیطی با context switch ممکن است **برابر با T یا کمتر از آن باشد**، اگر موازنه بین هزینه context switch و هزینه حلقه انتظار در spin lock به نفع spin lock باشد.

سوال 3:

Spinlock یک نوع قفل همگانی است که معمولا در برنامه های همزمان سیستم های چند هسته‌ای مورد استفاده قرار می گیرد. اما در سیستم های سینگل پروسسور استفاده از Spinlock ممکن است مناسب نباشد و حتی ممکن است یک نقطه ضعف هم حساب شود. دلایلی اصلی که Spinlock برای سیستم های سینگل پروسسور مناسب نیست عبارتند از:

1. کاهش توانایی پاسخگویی به وقفه ها:

استفاده از Spinlock ممکن است باعث کاهش توانایی پاسخگویی به وقفه ها شود چون زمانی که یک ترد در حال انتظار برای Spinlock است این ترد هیچ وظیفه دیگری را انجام نمیدهد و تا زمانی که به دسترسی به قفل نخواهد رسید، سیستم به کلی مسدود می شود

2. هدر رفتن منابع:

استفاده از Spinlock ممکن است باعث هدر رفتن منابع شود چون زمانی که یک ترد قفل را چک میکند و متوجه می شود که قفل در دسترس نیست به جای اینکه منتظر بماند به صورت فعال منابع را مصرف می کند و تا زمانی که قفل در دسترس نشود به این کار ادامه می دهد

3. پردازش های طولانی مدت:

استفاده از Spinlock ممکن است باعث پردازش های طولانی مدت شود به علت اینکه اگر پردازش های طولانی مدت با استفاده از Spinlock اجرا شود این امر باعث میشود که بقیه تردها زمان بیشتری در انتظار باقی بمانند در نتیجه تاخیرهای بیشتری در پاسخگویی سیستم به وجود می آید

سوال 4:

Mutual Exclusion:

الگوریتم با استفاده از متغیرهای turn و wants_to_enter مطمئن میشود که تنها یک پروسس در بخش Critial Section قرار دارد. زمانی یک پروسس مثلا پروسس p0 قصد ورود به Critical Section را دارد مقدار wants_to_enter[0] را به true تغییر میدهد سپس اگر پروسس p1 هم قصد ورود Critial Section را داشته باشد وارد یک حلقه while میشود و منتظر می ماند تا نوبت به آن برسد و اگر در این حالت نوبت پروسس p0 باشد از حلقه داخلی (trun !=0) خارج شده و به حلقه while اول باز می گردد و منتظر می ماند تا نوبت به پروسس فعلی برسد و وقتی نوبت رسید وارد Critial Section میشود و بعد از انجام عملیات های لازم trun را به شماره پروسس دیگر تغییر میدهد یعنی turn <- 1 پس شرط Mutual Exclusion تضمین میشود.

Absence of Deadlock:

با توجه به طراحی حلقه های while در کد، اگر یک پروسسی مانع ورود دیگری به Critical Section باشد، این موقعیت به صورت موقت است و هیچ کدام از پروسس ها به طور نادرست درون حلقه بی پایان گیر نمی کنند. اگر یک پروسس بخواهد وارد بخش Critical Section شود، اما دیگری این اجازه را ندهد (با توجه به wants_to_enter و turn)، آن پروسس به دلیل متغیر turn مجبور به صبر می شود و در نتیجه هیچ گونه deadlock ایجاد نمی شود.

:Progress

این الگوریتم اطمینان می دهد که هر دو پروسس به طور منصفانه فراخوانی می شوند. وقتی یک پروسس از Critical Section خارج می‌شود (wants_to_enter را به false تغییر می دهد)، پروسس دیگر می تواند وارد شود. این به این معناست که هر دو پروسس فرصت برابری برای دسترسی به منطقه Critial Section را دارند و هیچ کدام به طور ناعادلانه ای بر روی دیگری تاثیر نمی گذارد.

همچنین اگر پروسس دیگری مثلا p1 در حالت علاقه مندی به ورود به Critical Section باشد ولی نوبت p0 باشد بدون اینکه تداخلی به وجود آید p0 وارد بخش Critical Section می شود.

در نتیجه الگوریتم Dekker شرایط Critical Section را ارضا می کند.

سوال 5:

الگوریتم آیزنبرگ-مک‌گوایر به تدریج دو وضعیت را برای هر پروسس مدیریت می کند: WantCS و CS

WantCS: این متغیر نشان دهنده تمایل یک پروسس برای ورود به Critical Section است پس اگر یک پروسس تمایل به ورود به Critical Section را داشته باشد WantCS تغییر می کند.

CS: این متغیر نشان دهنده وضعیت Critical Section است. مقدار 1- به معنای این است که هیچ پروسسی در حال حاضر در Critical Section نیست، در غیر این صورت مقدار CS نشان دهنده پروسسی است که در حال حاضر در Critical Section وجود دارد.

:Mutual Exclusion .1

اگر یک پروسسی متغیر WantCS را تغییر داده باشد و قبلا هیچ پروسسی در بخش Critical نباشد یعنی CS = -1 باشد این به معنای تمایل پروسس به ورود به بخش Critical است و Mutual Exclusion حفظ می شود.

:Progress .2

زمانی که یک پروسس تمایل به ورود به بخش Critical را دارد (WantCS را تغییر داده است) و در حال حاضر هیچ پروسسی در بخش Critical نیست (CS = -1) پس پروسس فرصتی برای ورود به بخش Critical خواهد داشت. این امر به این معناست که الگوریتم از Progress حمایت می کند و هر پروسسی که تمایل به ورود داشته باشد در نهایت به بخش Critical وارد خواهد شد.

:Bounded Waiting .3

این ویژگی تضمین می کند که هیچ پروسسی نمی تواند بی پایان منتظر بماند. زمانی که یک پروسس تمایل به ورود به بخش Critical را دارد وارد وضعیت WantCS میشود. اگر یک پروسسی در صف انتظار برای ورود به

بخش Critical باشد تا زمانی که پروسس دیگری در بخش Critical نیست (CS = -1) وضعیت WantCS خود را تغییر خواهد داد و در نهایت به بخش Critical وارد خواهد شد.

سوال 6:

Reordering به تغییر ترتیب اجرای دستورات در یک برنامه اشاره دارد در حالت کلی این امکان وجود دارد که سیستم یا بهینه ساز کامپایلر، برخی از دستورات را به ترتیبی متفاوت از آنچه در کد اصلی نوشته شده است اجرا کند. این تغییر ترتیب اجرا ممکن است منجر به خروجی ها و کارکرد غیر منتظره در برنامه شود. مشکل اصلی که Reordering در الگوریتم پترسون ایجاد میکند ورود همزمان پروسس ها به Critical Section است که این باعث تداخل های نامطلوبی می شود. الگوریتم پترسون به منظور جلوگیری از تداخل در Critical Section از تست و تعیین نوبت (turn variable) استفاده می کند. به عنوان مثال فرض میکنیم که دو پروسس A و B همزمان به بخش تست نوبت می رسند. اگر Reordering اتفاق بیافتد و ابتدا پروسس A تست نوبت را انجام دهد و بعد پروسس B، ممکن است هر دو پروسس به عنوان نتیجه تست فرض کنند که نوبت در دسترس است (turn != i) و وارد Critical Section شوند. این باعث ورود همزمان دو پروسس به بخش Critical Section می شود که شرط Mutual Exclusion را تضمین نمی کند

استفاده از memory_barrier() در محل های مناسب می تواند مشکلات Reordering را حل بکند و این دستور برای ایجاد یک مرز حافظه استفاده میشود. این دستور اطمینان می دهد که تمام عملیات های حافظه قبل از آن انجام شده باشند و هیچ Reordering قبل و بعد از آن اتفاق نیفتد.

در الگوریتم پترسون این دستور به کار می رود تا اطمینان حاصل شود که تغییرات در متغیرهای گلوبال، تا آن لحظه اعمال شده اند و هیچ تداخل ناخواسته ای در ترتیب اجرای دستورات رخ ندهد.

برای مثال در کد زیر دستور memory_barrier() بعد از تنظیم wants_to_enter[i] و turn قرار گرفته است تا اطمینان حاصل شود که تمام پروسس ها از ترتیب درست اجرای دستورات مطلع شده اند تا هیچ تداخل غیرمنتظره ای درCritical Section رخ ندهد:

```
int wants_to_enter[2];
int turn;

void peterson_algorithm(int i) {
    wants_to_enter[i] = 1;
    memory_barrier();
    turn = 1 - i;
    while (wants_to_enter[1 - i] && turn == (1 - i)) {
        // busy wait
    }

    //Critical Section
    // ...

    wants_to_enter[i] = 0;
}
```

سوال 7:

```
// Acquire Lock
void acquire(lock *mutex) {
    while (test_and_set(&(mutex->available)) == 1) {
        // منتظر ماندن تا قفل آزاد شود
        ......
    }
    // حالا قفل گرفته شده است
}

// Release Lock
void release(lock *mutex) {
    mutex->available = 0;  // آزاد شدن قفل
}
```

سوال 8:

در این کد تردها به صورت همزمان تابع calc را اجرا کرده و به متغیر critical اضافه می کنند و با توجه به این که هر ترد آرایه ای مخصوص به خود را به عنوان ورودی به تابع calc می‌دهد، اضافه کردن اعداد آن آرایه به critical انجام میشود و در نهایت مقدار نهایی critical برابر با مجموع اعداد آرایه A و B خواهد شد:

critical = (1 + 2 + 3 + 4) + (5 + 6 + 7 + 8) = 36

اگر بخواهیم کد بهینه شده را با جابه جایی توابع mutex.release و mutex.acquire بهینه کنیم، باید ابتدا قفل را آزاد کنیم (release) و سپس دوباره قفل را بگیریم (acquire). این رویکرد انعطاف پذیری بیشتری را به تردها در اجرای همزمان، اجازه می دهد. در این حالت هر ترد می تواند به طور مستقل از دیگر تردها Critical Section دسترسی یابد.

```
int critical = 0;
int *A = {1, 2, 3, 4};
int *B = {5, 6, 7, 8};

void *calc(int *array) {
    mutex.release();
    int temp = 0;
    for (int i = 0; i < 4; i++) {
        temp += array[i];
    }
    critical += temp;
    mutex.acquire();
}
```

۲

و به دنبال آن اجرای کد هم سریعتر میشود به علت اینکه:

1. کاهش تداخل:

در کد جدید تردها ابتدا قفل را آزاد می کنند و سپس دوباره آن را می گیرند که این کار باعث کاهش تداخل بین تردها میشود چون هر ترد در زمانی که به Critical Section دسترسی پیدا میکند قفل را آزاد کرده و به ترد دیگر این امکان را می دهد که قفل را بگیرد این کار می تواند در کاهش انتظار تردها برای ورود به Critical Section تاثیرگذار باشد

2. افزایش همزمانی:

این کار به تردها این امکان را میدهد که به صورت همزمان اجرا شوند و هر کدام به Critical Section دسترسی پیدا کنند در نتیجه این افزایش همزمانی می تواند، بهبود کارایی کد را به همراه داشته باشد زیرا تردها می توانند به صورت همزمان بدون منتظر ماندن برای یکدیگر اجرا شوند.

سوال 9:

برای محدود کردن تعداد اتصالات همزمان در یک سرور، می‌توان از سمافورها استفاده کرد. این سمافورها به سرور این امکان را می دهند که تعداد اتصالات فعلی را کنترل کنند و اجازه ایجاد یا قبول اتصالات جدید را بر اساس تعداد مجاز تعیین کند.

1. در ابتدا یک سمافور برای نگهداری تعداد اتصالات فعلی و مجاز باید ایجاد شود

2. سپس تعداد مجاز اتصالات مطابق با نیاز سرور تنظیم می شود (N)

3. در هنگام برقراری یک اتصال جدید، سرور قبل از قبول اتصال، باید یک مقدار از سمافور کم کند. همچنین در هنگام قطع یک اتصال باید یک مقدار به سمافور اضافه کند

4. قبل از قبول یک اتصال جدید، سرور باید مقدار سمافور را بررسی و چک کند و اگر مقدار برابر با تعداد مجاز اتصالات یعنی N باشد، این به این معناست که حداکثر تعداد اتصالات همزمان به دست آمده و سرور نباید اتصال جدید را قبول بکند

5. هنگامی که اتصالی قطع می شود و یا یک اتصال جدیدی ایجاد می شود، سرور باید توجه داشته باشد که مقدار سمافور در محدوده تعداد مجاز اتصالات یعنی 0 تا N باشد تا اطمینان حاصل شود که سرور تنها تا حداکثر تعداد مجاز اتصالات اجازه برقراری اتصال می دهد.

در زیر یک نمونه ساده از استفاده از سمافور برای محدود کردن تعداد اتصالات همزمان در یک سرور آورده شده است:

خلاصه ای از کد زیر:

در این کد از sem_init برای ایجاد سمافور با تعداد اولیه مشخص (برابر با تعداد حداکثر اتصالات همزمان) و از sem_wait و sem_post برای افزایش و کاهش مقدار سمافور استفاده شده است همچنین از std::lock_guard برای قفل کردن متغیرهای اشتراکی استفاده شده است تا مشکلات همگام‌سازی حل شوند.

کلاس ConnectionManager مدیریت کننده اتصالات با استفاده از سمافور و قفل، تعداد اتصالات همزمان را محدود کرده و در هنگام اتصال یا قطع، ایمنی عملیات‌های مرتبط با اتصالات را تضمین می‌کند

توابع Connect و Disconnect به ترتیب برای ایجاد و قطع اتصال استفاده می شوند.

تابع ClientThread در این مثال نمونه ای از عملیاتی است که یک کلاینت در یک سناریوی واقعی ممکن است انجام دهد. در اینجا ClientThread یک کلاینت را نمایش می دهد که به ConnectionManager متصل میشود و سپس عملیات دلخواه خود را انجام می دهد و در انتها اتصال خود را قطع می کند

```
#include <iostream>
#include <thread>
#include <mutex>
#include <semaphore.h>

class ConnectionManager {
public:
    ConnectionManager(int maxConnections) : maxConnections_(maxConnections)
{
        sem_init(&connectionSemaphore, 0, maxConnections);
    }

    ~ConnectionManager() {
        sem_destroy(&connectionSemaphore);
    }

    void Connect() {
        sem_wait(&connectionSemaphore);  // از واحد یک گرفتن برای انتظار
سمافور
        // است شده برقرار اتصال
        std::lock_guard<std::mutex> lock(mutex_);
        currentConnections_++;
        std::cout << "Connection established. Total connections: " <<
currentConnections_ << std::endl;
    }

    void Disconnect() {
        {
            std::lock_guard<std::mutex> lock(mutex_);
            // اتصال قطع
            currentConnections_--;
            std::cout << "Connection closed. Total connections: " <<
currentConnections_ << std::endl;
        }
        sem_post(&connectionSemaphore);  // سمافور واحد یک کردن آزاد
    }

private:
    int maxConnections_;
    int currentConnections_;
    std::mutex mutex_;
    sem_t connectionSemaphore;
};
```

```
// استفاده مثال
void ClientThread(ConnectionManager& connectionManager) {
    connectionManager.Connect();
    // دیگر های‌عملیات
    connectionManager.Disconnect();
}

int main() {
    const int maxConnections = 3;
    ConnectionManager connectionManager(maxConnections);

    // تردها ساخت
    std::thread threads[5];
    for (int i = 0; i < 5; ++i) {
        threads[i] = std::thread(ClientThread,
std::ref(connectionManager));
    }

    // تردها چیدمان
    for (int i = 0; i < 5; ++i) {
        threads[i].join();
    }

    return 0;
}
```