به نام خدا

برنامه‌سازی پیشرفته

آرش شفیعی

برنامه‌سازی همروند

مقدمه

- برنامه‌سازی همروند یا همزمان [1] به معنای اجرای برنامه‌هاست به گونه‌ای که تعدادی از عملیات به طور همزمان اجرا شوند.
- اگر قسمت‌هایی از برنامه به صورت همزمان اجرا شوند، زمان پاسخ (یا تاخیر) [2] و توان عملیاتی [3] برنامه بهبود می‌یابد.

[1] concurrenct programming
[2] response time (or delay)
[3] throughput

مقدمه

- به طور مثال محاسبه ضرب دو ماتریس را در نظر بگیرید. از آنجایی که درایه‌های ماتریس حاصلضرب مستقل از یکدیگر قابل محاسبه هستند بنابراین هر یک از درایه‌ها می‌توانند به طور مستقل و موازی با درایه‌های محاسبه شوند (البته در صورتی که تعداد پردازنده‌ها به تعداد کافی باشد) و در این صورت ضرب دو ماتریس با تاخیر کمتری محاسبه خواهد شد.
- حال یک سیستم پردازش تصویر را در نظر بگیرید که در آن تصاویر به ترتیب از ورودی خوانده می‌شوند و پس از چند مرحله پردازش در خروجی نمایش داده می‌شوند. پس از این که اولین مرحلهٔ پردازش توسط یک پردازنده انجام شد، در صورتی که تعداد پردازنده‌ها به تعداد کافی باشد، پردازنده اول می‌تواند تصویر دوم را پردازش کند و تصویر اول برای پردازش به پردازندهٔ دوم برود. بدین ترتیب در یک فاصله زمانی معین تعداد بیشتری تصویر پردازش می‌شوند. در این حالت می‌گوییم توان عملیاتی افزایش سیستم افزایش یافته است.

مقدمه

- در برخی مواقع نیاز به همزمانی دو قسمت از برنامه پیدا می‌کنیم، زیرا هر قسمت وظیفه‌ای معین و متفاوت را انجام می‌دهند در حالی که آن دو قسمت باید با هم به طور همزمان اجرا شوند و در ارتباط باشند.
- برای مثال برنامه‌ای را در نظر بگیرید که مقادیری را در خروجی چاپ می‌کند و در صورت دریافت مقداری معین از ورودی برنامه را خاتمه می‌دهد. در این صورت قسمتی از برنامه که مسئولیت چاپ را بر عهده دارد باید به طور موازی با قسمتی از برنامه که مسئولیت دریافت ورودی را بر عهده دارد اجرا شود.

مقدمه

- همهٔ زبان‌های برنامه‌سازی قابلیت‌هایی برای اجرای برنامه‌های موازی دارند. در زبان سی++ نیز در کتابخانهٔ استاندارد، کلاس‌ها و توابعی برای برنامه‌سازی همروند فراهم شده است.

- برنامه‌سازی همروند به دو صورت می‌تواند انجام شود: چند پردازه‌ای [1] و چندریسه‌ای (چندنخی یا چند ریسمانی) [2].

- یک پروسه یا پردازه [3] نمونه‌ای از برنامه است که بر روی فضایی متمایز در حافظه اجرا می‌شود، بنابراین پروسه‌ها با یکدیگر حافظه را به اشتراک نمی‌گذارند. اگر قسمت‌هایی از برنامه به طور کاملا مستقل بر روی پروسه‌های متمایز اجرا شوند، می‌گوییم برنامه به صورت چندپردازه‌ای اجرا می‌شود.

- یک ریسه (نخ یا ریسمان) [4] واحدی از دستورات برنامه است که به طور مستقل اجرا می‌شود و حافظه را با دیگر ریسه‌ها در پروسهٔ خود به اشتراک می‌گذارد. اگر قسمت‌های مختلف برنامه بر روی ریسه‌های متمایز اجرا شوند، می‌گوییم برنامه به صورت چندریسه‌ای اجرا می‌شود.

[1] multiprocessing
[2] multithreading
[3] process
[4] thread

ریسه‌ها

- هر واحد محاسباتی را که می‌تواند به طور همزمان با واحدهای محاسباتی دیگر انجام شود، یک وظیفه [1] می‌نامیم.
- یک وظیفه یا task را یک ریسه یا thread اجرا می‌کند.
- در کتابخانه استاندارد کلاس `thread` پیاده سازی شده است. یک ریسه تابعی را برای اجرا دریافت و اجرا می‌کند. برای همگام‌سازی [2] ریسه‌ها گاه لازم است برای به پایان رسیدن یک ریسه صبر کنیم تا به پایان برسد. توسط تابع `join` بر روی یک ریسه، اجرای برنامه متوقف می‌شود تا ریسه مورد نظر به پایان برسد.

[1] task
[2] synchronization

ریسه‌ها

- در کتابخانه استاندارد کلاس thread پیاده سازی شده است.

```
۱ void hello() { for(int i=0; i<1000; i++) cout << "Hello" << endl; }
۲ void world() { for(int i=0; i<1000; i++) cout << "World" << endl; }
۳ void execute() {
۴   thread t1 {hello}; // hello() executes on one thread
۵   thread t2 {world}; // world() executes on another separate thread
۶   t1.join(); // wait for t1
۷   t2.join(); // wait for t2
۸ }
```

- پس وقتی اجرای برنامه به توابع join می‌رسد، اجرا متوقف می‌شود تا اجرای ریسه‌ها به پایان برسند و پس از آن اجرای برنامه ادامه پیدا کرده و تابع execute به اتمام می‌رسد.

ریسه‌ها

- همچنین می‌توانیم مقدار ورودی توابع را به ریسه‌ها ارسال کنیم.

```
۱ void print(const string & s) { for(int i=0; i<1000; i++) cout << s << e
۲ void execute() {
۳   thread t1 {print, "Hello"}; // print("Hello") executes on one thread
۴   thread t2 {print, "World"}; // print("World") executes on another thr
۵   t1.join(); // wait for t1
۶   t2.join(); // wait for t2
۷ }
```

ریسه‌ها

- مقدار خروجی یک تابع را می‌توانیم توسط یک متغیر مرجع در ورودی تابع دریافت کنیم.

```
۱ void multiply(int x, int y, int & res) { res = x * y; }
۲ void execute() {
۳   int res1, res2;
۴   thread t1 {multiply, 2, 3, res1};
۵   thread t2 {multiply, 4, 5, res2};
۶   t1.join(); // wait for t1
۷   t2.join(); // wait for t2
۸ }
```

حافظهٔ مشترک

- ریسه‌ها می‌توانند با یکدیگر حافظه را به اشتراک بگذارند. در کد زیر ریسه‌ها چه قسمتی را به اشتراک گذاشته‌اند؟

```
۱ void hello() { for(int i=0; i<1000; i++) cout << "Hello" << endl; }
۲ void world() { for(int i=0; i<1000; i++) cout << "World" << endl; }
۳ void execute() {
۴   thread t1 {hello}; // hello() executes on one thread
۵   thread t2 {world}; // world() executes on another separate thread
۶   t1.join(); // wait for t1
۷   t2.join(); // wait for t2
۸ }
```

حافظهٔ مشترک

- وقتی قسمتی از حافظه بین دو ریسه به اشتراک گذاشته می‌شود، ممکن است در هنگام اجرا نتیجهٔ مطلوب برنامه نویس به دست نیاید.
- به طور مثال هر دو تابع `hello` و `world` در کد زیر از شیء `cout` برای چاپ در خروجی استاندارد استفاده می‌کنند. بنابراین ممکن است قبل از اینکه تابع اول چاپ را به پایان برساند، تابع دوم اجرا شود و تابع دوم کنترل خروجی استاندارد را به دست بگیرد و در نتیجه قبل از اتمام چاپ یک رشته در تابع اول، یک رشته در تابع دوم چاپ شود.

```
1 void hello() { for(int i=0; i<1000; i++) cout << "Hello" << endl; }
2 void world() { for(int i=0; i<1000; i++) cout << "World" << endl; }
```

حافظهٔ مشترک

- به عنوان یک مثال دیگر، فرض کنید دو تابع به طور همزمان مقدار یک متغیر `counter` را افزایش می‌دهند.
- بنابراین هر ریسه در کد خود `++counter` را اجرا می‌کند.
- در زبان اسمبلی این کد به سه قسمت تقسیم می‌شود: (۱) خواندن متغیر از حافظه و ذخیره آن بر روی رجیستر پردازنده، (۲) افزایش مقدار، (۳) ذخیره مقدار رجیستر بر روی حافظه.
- حال دو ریسه ممکن است به طور همزمان این عملیات را انجام داده و در نتیجه هر دو همزمان این عملیات را انجام داده و به جای اینکه دو واحد به مقدار متغیر افزوده شود، تنها یک واحد به مقدار آن افزوده شود.
- بنابراین به سازوکاری نیاز داریم که از حافظهٔ مشترک محافظت کنیم و اجازه ندهیم ریسه‌های مختلف به طور همزمان بر روی حافظه‌های مشترک عملیات انجام دهد.
- چنین سازوکاری فراهم شده است به گونه‌ای که هر ریسه قبل از شروع به کار با یک متغیر مشترک، قفلی را در دست گرفته و به دیگران اجازه نمی‌دهد تا هنگام رهاسازی آن قفل، از متغیر مشترک استفاده کنند.

حافظهٔ مشترک

- فرض کنید چندین ریسه تابع زیر را اجرا کنند. به دلیلی که ذکر شد، پس از اتمام اجرای همهٔ ریسه‌ها، مقدار متغیر counter به مقدار مورد نظر افزایش نمی‌یابد.

```
۱  int counter = 0;
۲  void count() {
۳      for(int i=0; i<1000; i++)
۴          counter++;
۵  }
۶  int main() {
۷      thread t1 {count};
۸      thread t2 {count};
۹      t1.join();
۱۰     t2.join();
۱۱     cout << counter << endl; // counter is less than 2000
۱۲     return 0;
۱۳ }
```

حافظهٔ مشترک

- در زبان سی++ کلاسی به نام `mutex` به معنی انحصار متقابل [1] سازوکاری را فراهم می‌کند که هر ریسه بتواند دسترسی به یک دادهٔ مشترک را قفل کرده و منحصرا بر روی داده کار کند و پس از اتمام کار بر روی داده مشترک قفل را آزاد کرده تا ریسه‌های دیگر بتوانند از آن داده استفاده کنند.
- بدین ترتیب قبل از دسترسی به حافظه مشترک باید متغیری از کلاس `mutex` را توسط تابع `lock()` قفل و پس از دسترسی باید آن را توسط تابع `unlock()` آزاد کرد.

[1] mutual exclusion

حافظهٔ مشترک

- در مثال زیر هر ریسه در هنگام استفاده از شیء cout می‌تواند اطمینان حاصل کند که ریسهٔ دیگری به آن دسترسی ندارد.

```
۱ mutex m;
۲ void print(string s) {
۳    for(int i=0; i<1000; i++) {
۴        m.lock();
۵        cout << s << endl;
۶        m.unlock();
۷    }
۸ }
```

حافظهٔ مشترک

- در مثال شمارنده، می‌تواند متغیر counter در هر ریسه با استفاده از یک mutex قفل کرد. بدین ترتیب متغیر به مقدار مورد نظر افزایش پیدا می‌کند.

```
۱  int counter = 0;
۲  mutex m;
۳  void count() {
۴      for(int i=0; i<1000; i++) {
۵          m.lock(); counter++; m.unlock();
۶      }
۷  }
۸  int main() {
۹      thread t1 {count}; thread t2 {count};
۱۰     t1.join(); t2.join();
۱۱     cout << counter << endl; // counter is equal to 2000
۱۲     return 0;
۱۳ }
```

حافظهٔ مشترک

- یک راه حل دیگر این است که متغیر counter با استفاده از کلاس atomic به عنوان یک متغیر تجزیه‌ناپذیر در سطح کد اسمبلی تعریف شود و بدین ترتیب کد اسمبلی مربوط به محاسبات بر روی متغیر به طور تجزیه‌ناپذیر اجرا شده و مشکل مذکور مرتفع می‌شود. در هر ریسه با استفاده از یک mutex قفل کرد. بدین ترتیب متغیر به مقدار مورد نظر افزایش پیدا می‌کند.

```
۱  atomic<int> counter = 0;
۲  mutex m;
۳  void count() {
۴      for(int i=0; i<1000; i++) {
۵          counter++;
۶      }
۷  }
۸  int main() {
۹      thread t1 {count}; thread t2 {count};
۱۰     t1.join(); t2.join();
۱۱     cout << counter << endl; // counter is equal to 2000
۱۲     return 0;
۱۳ }
```

حافظهٔ مشترک

- می‌توان از کلاس scoped_lock برای قفل کردن mutex استفاده کرد. بدین ترتیب هنگامی که حوزه تعریف scoped_lock خارج می‌شویم، قفل به طور خودکار آزاد می‌شود.

```
۱ mutex m; // controlling mutex
۲ void print(string s) {
۳    for(int i=0; i<1000; i++) {
۴        scoped_lock lck {m}; // acquire mutex
۵        cout << s << endl; // manipulate shared data
۶    } // release mutex implicitly after each iteration
۷ }
```

حافظهٔ مشترک

- برای ارسال یک متغیر با ارجاع به تابع یک ریسه از ref() استفاده می‌کنیم.

```
۱  mutex m; // controlling mutex
۲  void print(const string & s) {
۳     for(int i=0; i<1000; i++) {
۴          scoped_lock lck {m}; // acquire mutex
۵          cout << s << endl; // manipulate shared data
۶     } // release mutex implicitly after each iteration
۷  }
۸  void execute() {
۹    string s1 = "Hello";
۱۰   string s2 = "World";
۱۱   thread t1 {print, ref(s1)};
۱۲   thread t2 {print, ref(s2)};
۱۳   t1.join();
۱۴   t2.join();
۱۵ }
```

حافظهٔ مشترک

- گاهی داده‌ای بین چندین ریسه به اشتراک گذاشته شده است در حالی که تعدادی از ریسه‌ها فقط متغیر را می‌خوانند و یک ریسه بر روی متغیر می‌نویسد. در چنین مواقعی ریسه‌هایی که فقط متغیر را می‌خوانند می‌توانند همگی قفل را دریافت کرده و متغیر را بخوانند. نویسنده تنها در صورتی می‌تواند بر روی متغیر بنویسد که هیچ خواننده‌ای مشغول خواندن آن نباشد. در چنین سناریویی می‌توانیم از `shared_lock` و `unique_lock` استفاده کنیم.

```
۱ shared_mutex mx; // a mutex that can be shared
۲ void reader() {
۳   shared_lock lck {mx}; // willing to share access with other readers
۴   // ... read ...
۵ }
۶ void writer() {
۷   unique_lock lck {mx}; // needs exclusive (unique) access
۸   // ... write ...
۹ }
```

رویدادها

- گاهی یک ریسه نیاز دارد قبل از ادامهٔ کار برای یک رویداد خارجی صبر کند. این رویداد خارجی می‌تواند پیام یا سیگنالی باشد که از یک ریسهٔ دیگر ارسال می‌شود و یا صرفا این رویداد یک زمان خاص باشد.
- در صورتی که رویداد زمان باشد می‌توان با استفاده از کتابخانهٔ chrono وقفه در اجرای ریسه ایجاد کرد.

```
۱ using namespace std::chrono;
۲ auto t0 = high_resolution_clock::now();
۳ this_thread::sleep_for(milliseconds{20});
۴ auto t1 = high_resolution_clock::now();
۵ duration<double> diff = t1 - t0;
۶ cout << diff.count() << " seconds passed\n";
۷ cout << duration_cast<nanoseconds>(diff).count()
۸        << " nanoseconds passed\n";
```

رویدادها

- برای ارتباط دو ریسه و ارسال سیگنال از یک ریسه به ریسهٔ دیگر کتابخانهٔ `condition_variable` طراحی شده است.
- یک ریسه می‌تواند بر روی شیئی از کلاس `condition_variable` منتظر یک رویداد بماند. این رویداد سیگنالی است که از یک ریسهٔ دیگر ارسال می‌شود. با دریافت سیگنال و اطلاع از رویداد، ریسهٔ در حال انتظار، فعالیت مورد نظر را ادامه می‌دهد.

رویدادها

- برای مثال دو ریسه را در نظر بگیرید که به طور همزمان بر روی یک صف عملیات انجام می‌دهند. یکی از ریسه‌ها اطلاعات را از روی ریسه می‌خواند و ریسهٔ دیگر اطلاعات را بر روی صف می‌نویسد. پس یک ریسهٔ خواننده و یک ریسهٔ نویسنده بر روی این صف عملیات انجام می‌دهند.

```
۱ class Message { // object to be communicated
۲ // ...
۳ };
۴
۵ queue<Message> mqueue; // the queue of messages
۶ condition_variable mcond; // the variable communicating events
۷ mutex mmutex; // for synchronizing access to mcond
```

رویدادها

- ریسهٔ خواننده برای اینکه بتواند پیامی را از روی صف بخواند نیاز دارد از خالی نبودن صف اطمینان پیدا کند.
- برای این کار بر روی `condition_variable` تابع `wait` را فراخوانی می‌کند.

```
۱  void consumer() {
۲    while(true) {
۳      unique_lock lck {mmutex}; // acquire mmutex
۴      mcond.wait(lck,[] { return !mqueue.empty(); });
۵        // release lck and wait;
۶        // re-acquire lck upon wakeup
۷        // 'dont wake up unless mqueue is non-empty
۸      auto m = mqueue.front(); // get the message
۹      mqueue.pop();
۱۰     lck.unlock(); // release lck
۱۱     // ... process m ...
۱۲   }
۱۳ }
```

رویدادها

- همچنین ریسهٔ نویسنده هر بار پیامی را در صف وارد می‌کند سیگنالی را به خواننده ارسال می‌کند.
- برای این کار بر روی condition_variable تابع notify_one را فراخوانی می‌کند.

```
۱ void producer() {
۲   while(true) {
۳     Message m;
۴     // ... fill the message ...
۵     scoped_lock lck {mmutex}; // protect operations
۶     mqueue.push(m); // notify
۷     mcond.notify_one(); // release lock (at end of scope)
۸   }
۹ }
```

مثال

- می‌خواهیم برنامه‌ای بنویسیم که مقداری را در خروجی چاپ کند و به طور همزمان مقداری را از ورودی دریافت کرده و در صورتی که مقدار ورودی برابر با حرف q باشد از برنامه خارج شود.

مثال

```
۱  bool active = true;
۲  void input() {
۳      char ch;
۴      while(active) {
۵          ch=getchar();
۶          if (ch == 'q') active = false;
۷      }
۸  }
۹  int main() {
۱۰     thread t {input};
۱۱     while(active) {
۱۲         cout << "Hello World!" << endl;
۱۳     }
۱۴     t.join();
۱۵     return 0;
۱۶ }
```

مثال

- برنامه‌ای بنویسید که ضرب دو ماتریس را به طور موازی بر روی چند ریسه انجام دهد.