یادداشت‌های درس سیستم‌های عامل - بخش ششم

در بخش قبل درس وضعیت رقابتی، قفل‌های Mutex و سمافور و چند مسئله‌ی کلاسیک همگام‌سازی را مطالعه کردیم. در این بخش موضوعات دیگری در ارتباط با همگام‌سازی را بررسی می‌کنیم.

- در بخش قبلی درس با دو ابزار برای همگام‌سازی آشنا شده‌ایم: قفل Mutex و سمافور.
- در ادامه با دو ابزار دیگر آشنا می‌شویم: مانیتور و عملیات اتمی.

- **مانیتور** (Monitor): ساختارهایی در زبان‌های سطح بالا که ویژگی انحصار متقابل و انتظار برای رخدادها را فراهم می‌کنند.
- گاهی محدود به یک تابع، گاهی محدود به یک کلاس در زبان‌های شیء گرا و گاهی محدود به یک فایل یا ماژول.
- در هر لحظه فقط یک ریسه می‌تواند هر قسمت از آن را اجرا کند.
- در ادامه، مانیتورها یک کلاس هستند.
- معمولا راحت‌تر از استفاده از سمافور یا قفل‌ها.

مانیتور - مثال اول

```
1  Monitor Example1 {
2       int count = 1;
3
4       void buy(void)
5       {
6            if (count == 0) {
7                 error("None left!\n");
8                 return;
9            }
10           count = count - 1;
11      }
12 }
```

- در بسیاری از زبان‌های برنامه‌نویسی شیء گرا مشابه تعریف یک کلاس می‌توانید یک مانیتور تعریف کنید (در برخی زبان‌ها همه‌ی اشیاء می‌توانند به یک مانیتور تبدیل شوند).
- از هر مانیتور می‌توان نمونه (Instance) ساخت.

```
Example1 ex1;
Example1 ex2;
ex1.buy();
```

مانیتور - انحصار متقابل

- ویژگی انحصار متقابل مانیتور.

```
Example1 ex1;
Example1 ex2;
```

- فرض کنید ریسه‌ها کد زیر را اجرا کنند.

Thread #1	Thread #2	Thread #3
... ex1.buy(); ...	... ex1.buy(); ...	... ex2.buy(); ...

مانیتور - مثال دوم

- قطعه کد زیر مربوط به مثالی هست که قبلا بررسی کرده‌ایم.

```
 1  Monitor Account {
 2        int balance;
 3
 4        void deposit(int x)
 5        {
 6             int n = balance + x;
 7             balance = n;
 8        }
 9        void withdraw(int x)
10        {
11             int n;
12             if (balance < x) {
13                  error("Account balance too low\n");
14                  return;
15             }
16             n = balance - x;
17             balance = n;
18        }
19  }
```

- به خاطر ویژگی انحصار متقابل توابع مانیتور، چند ریسه این این دو تابع را به صورت همزمان اجرا نمی‌کنند و در نتیجه، وضعیت رقابتی رخ نمی‌دهد.

- فرض کنید نمونه‌ی account در حافظه‌ی مشترک ایجاد شده باشد.

```
Account account;
```

Thread #1	Thread #2
... account.withdraw(10); ...	... account.deposit(20); ...

- فراخوانی توابع این نمونه از مانیتور توسط دو ریسه به صورت انحصار متقابل انجام می‌شود.

- برای انتظار در مانیتور باید از متغیر با نوع Condition استفاده کرد.

```
Condition cond;
```

- **تابع** cond.wait(): باعث می‌شود ریسه‌ی فراخوانی کننده Block شود.
- **تابع** cond.signal(): باعث می‌شود یکی از ریسه‌هایی که با فراخوانی cond.wait() منتظر هستند اجرایش ادامه یابد.

Thread #1	Thread #2
... cond.wait(); ...	... cond.signal(); ...

- گاهی نام notify() برای تابع signal به کار می‌رود.
- در برخی از پیاده‌سازی‌ها تابعی به نام notify_all() یا مشابه آن وجود دارد که همه‌ی ریسه‌های منتظر برای یک متغیر Condition را بیکار می‌کند.

- برای مثال مانیتور زیر را در نظر بگیرید.

```
Monitor PingPong {
     Condition cond;
     void ping(void) {
          cond.wait();
          printf("A\n");
     }
     void pong(void) {
          cond.signal();
          printf("B\n");
     }
}

PingPong pp;
```

- فرض کنید دو ریسه به شکل زیر از این مانیتور استفاده کنند.

Thread #1	Thread #2
... pp.ping(); ...	... pp.pong(); ...

- فرض کنید ریسه‌ی اول زودتر pp.ping() را اجرا کند و در cond.wait() منتظر شود.
- وقتی ریسه‌ی دوم pp.pong() تابع cond.signal() را فراخوانی می‌کند، دو ریسه‌ی آماده‌ی اجرا داریم: هم ریسه‌ی اول می‌تواند اجرا شود و هم ریسه‌ی دوم.
- اما با توجه به ویژگی انحصار متقابل مانیتور آنها نمی‌توانند همزمان اجرا شوند؛ یکی باید منتظر خروج دیگری از مانیتور شود تا بتواند اجرا شود.
- اما کدام ریسه باید زودتر اجرا شود؟ ریسه‌ی فراخوانی کننده‌ی signal() یا ریسه‌ای که از wait() بیرون آمده است؟
- به عبارت دیگر، در قطعه کد صفحه‌ی قبل B قبل یا بعد از A چاپ می‌شود؟
- این مسئله به پیاده‌سازی بستگی دارد؛ دو حالت داریم:

Signal and Wait: در این حالت ریسه‌ی فراخوانی کننده‌ی signal() منتظر می‌شود تا ریسه‌ی بیدار شده از مانیتور خارج شود.

Signal and Continue: در این حالت ریسه‌ی فراخوانی کننده‌ی signal() منتظر نمی‌شود و ریسه‌ی بیدار شده پس از خروج دیگری از مانیتور دیگری ادامه می‌یابد.

- از یک مانیتور برای مدیریت Buffer استفاده می‌کنیم (پیاده‌سازی در صفحه‌ی بعد) که دو تابع دارد: تابع produce برای افزودن داده و consume برای دریافت داده از صف انتقال.
- یک نمونه از این مانیتور باید در حافظه‌ی مشترک دو ریسه تعریف شود.

```
ProdCons pc;
```

- تولید کننده قطعه کد زیر را اجرا می‌کنند.

```
void producer(void) {
     while (1) {
          Item x = ...;       // # produce item
          pc.produce(x);
     }
}
```

- مصرف کننده قطعه کد زیر را اجرا می‌کنند.

```
void consumer(void) {
     while (1) {
          Item x = pc.consume();
          # consume item
          ...
     }
}
```

- در ادامه پیاده‌سازی مانیتور نمایش داده می‌شود.

```
1  Monitor ProdCons {
2        Queue queue;
3        Condition fcond;
4        Condition econd;
5        void produce(Item x) {
6             if (queue.isfull())
7                  fcond.wait();
8             queue.append(x);
9             econd.signal();
10       }
11       Item consume(void) {
12            Item x;
13            if (queue.isempty())
14                 econd.wait();
15            x = queue.remove();
16            fcond.signal();
17            return x;
18       }
19 }
```

- به خاطر ویژگی انحصار متقابل مانیتور نیازی به یک قفل برای دسترسی به متغیر queue وجود ندارد.
- از دو متغیر Condition برای انتظار تولید کننده و مصرف کننده استفاده می‌شود.

عملیات اتمی

- عملیات اتمی (Atomic Operations) عملیاتی هستند که در یک واحد غیر قابل وقفه توسط پردازنده اجرا می‌شوند.
- یک عمل اتمی به صورت یکجا انجام می‌شود و امکان ندارد پردازنده‌های دیگر به صورت ناقص خروجی این عملیات را بخوانند یا در وسط آن وقفه رخ دهد.
- این عملیات معمولا محدود و ساده هستند و توسط شرکت سازنده‌ی پردازنده می‌شوند.
- دو نمونه از این عملیات را در ادامه خواهیم دید.

عملیات اتمی - مثال

- بسیاری از پردازنده‌ها عمل test_and_set را به صورت اتمی پیاده‌سازی می‌کنند.
- این عمل یک آدرس از حافظه را به عنوان ورودی می‌پذیرد.
- دقت کنید که شبه کد زیر پیاده‌سازی نیست و فقط برای نمایش و توضیح است.

```
1  int test_and_set(int *p) {
2        int old = *p;
3        *p = 1;
4        return old;
5  }
```

عملیات اتمی - مثال

- عمل دیگری که برخی از پردازنده‌ها می‌توانند به صورت اتمی پیاده‌سازی کند compare_and_swap هست.

```
1  int compare_and_swap(int *p, int expected, int value) {
2       int old = *p;
3       if (*p == expected)
4            *p = value;
5       return old;
6  }
```

مقایسه‌ی عملیات اتمی با قفل‌ها

- عملیات اتمی مستقیما توسط پردازنده پیاده‌سازی می‌شوند و سریع‌تر هستند.
- اما دلایلی وجود دارند که نمی‌توان همیشه از آنها استفاده کرد:

الف) معمولا عملیات کمی در پردازنده هستند که به صورت اتمی پیاده‌سازی می‌شوند.

ب) معمولا عملیات اتمی ساده هستند و برای ناحیه‌های بحرانی پیچیده مناسب نیستند.

- در پیاده‌سازی قفل‌ها از عملیات اتمی استفاده می‌شود.

- سه ویژگی راه حل مناسب برای حل ناحیه‌ی بحرانی را قبلا بررسی کرده‌ایم:

  انحصار متقابل،

  پیشرفت و

  انتظار محدود.

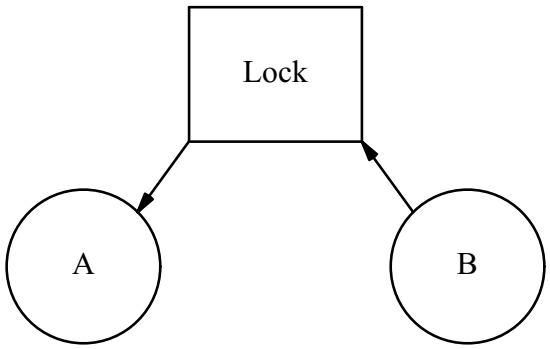
- در ادامه پیاده‌سازی آنها را مطالعه می‌کنیم.

- الف) قفل‌های مبتنی بر Busy-waiting:

معمولا حلقه‌ای وجود دارد که مقدار کلمه‌ای از حافظه را به صورت مکرر بررسی می‌کند.

در این قفل‌ها پردازنده مشغول نگه داشته می‌شود.

به این قفل‌ها گاهی قفل چرخشی (Spinlock) هم گفته می‌شود.



- ب) قفل‌های مبتنی بر Blocking:

پردازه یا ریسه‌ای که باید منتظر شود توسط سیستم عامل در حالت Waiting قرار می‌گیرد.

وقتی قفل آزاد شود، سیستم عامل پردازه را در حالت Running قرار می‌دهد تا اجرا شود.

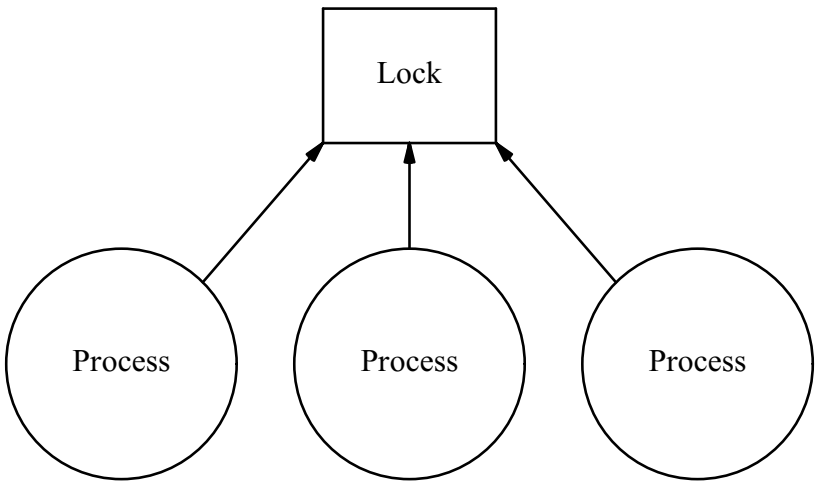
- اگر زمان انتظار کم باشد، قفل‌های چرخشی تأخیر کمتری دارند و انتخاب بهتری هستند.
- اگر زمان انتظار زیاد باشد، قفل‌های چرخشی زمان پردازنده را هدر می‌دهند؛ به جای اینکه پردازنده پردازه‌ها را اجرا کند، زمان را صرف بررسی حافظه می‌کند؛ بنابراین با زمان انتظار زیاد، قفل‌های Blocking انتخاب بهتری هستند.

- قفل‌های تطبیقی (Adaptive Mutex) قفل‌هایی هستند که هم می‌توانند از Busy-waiting و هم از Blocking استفاده کنند.
- وقتی یک ریسه درخواست در اختیار گرفتن قفل را می‌دهد:

اگر قفل باز باشد بدون انتظار در اختیارش قرار می‌گیرد.

اگر قفل بسته باشد و ریسه‌ی در اختیار دارنده‌ی قفل در وضعیت Waiting باشد، حالت Blocking برای انتظار انتخاب می‌شود.

اگر قفل بسته باشد و ریسه‌ی در اختیار دارنده‌ی قفل در وضعیت Running باشد، حالت Busy-waiting برای انتظار انتخاب می‌شود.



پیاده‌سازی قفل - تلاش اول

- در ادامه یک پیاده‌سازی قفل به روش Busy-waiting با کمک عمل اتمی test_and_set نمایش داده می‌شود.
- در این پیاده‌سازی اگر مقدار قفل یک باشد یعنی بسته است و اگر صفر باشد یعنی باز است.

```
1  void acquire(int *lock)
2  {
3       while (test_and_set(lock) == 1)
4            ;
5  }
6  void release(int *lock)
7  {
8       *lock = 0;
9  }
```

- سه شرط لازم برای درستی قفل را بررسی می‌کنیم.
- در این پیاده‌سازی انحصار متقابل برقرار است چون اگر قفل در اختیار یک ریسه باشد، ریسه‌های دیگری که خواهان گرفتن قفل باشند در حلقه‌ی تابع acquire منتظر می‌شوند.
- در این پیاده‌سازی شرط پیشرفت برقرار است چون بین ریسه‌های منتظر، یکی از آنها مقدار قفل را به یک تغییر می‌دهد و از حلقه‌ی انتظار خارج می‌شود.
- شرط انتظار محدود در این پیاده‌سازی برقرار نیست همه‌ی ریسه‌های منتظر (که تابع acquire را صدا زده‌اند) حلقه‌ی خط سوم و چهارم را اجرا می‌کنند. وقتی قفل آزاد شود مشخص نیست کدام مقدار را از صفر به یک تغییر می‌دهند و ممکن است ریسه‌ای باشد که همچنان در این حلقه منتظر باشد در حالی که ریسه‌های دیگری قفل را در اختیار بگیرند و آزاد کنند.

- ایده‌ی اصلی: بهره گیری از یک آرایه‌ی کمکی به نام waiting.
- اندازه‌ی این آرایه حداکثر تعداد ریسه‌ها یا پردازه‌ها (N) است.
- هر ریسه‌ای که بخواهد قفل را در اختیار بگیرد مقدار خانه‌ی متناظرش در آرایه‌ی waiting را به یک تغییر می‌دهد.

Waiting

0	1
1	0
2	0
3	1
4	0
5	0
6	0
7	1

- هر ریسه‌ای که قفل را آزاد می‌کند در آرایه‌ی waiting به دنبال اولین ریسه‌ای می‌گردد که منتظر هست (یعنی خانه‌ی متناظرش در waiting یک هست).
- اگر چنین ریسه‌ای موجود بود، بدون باز کردن قفل آن را در اختیار آن ریسه می‌گذارد.
- اگر چنین ریسه‌ای موجود نبود قفل را باز می‌کند.
- این پیاده‌سازی هر سه شرط قفل‌ها را دارد.
- هر ریسه حداکثر به اندازه‌ی استفاده‌ی N - 1 ریسه‌ی دیگر از قفل منتظر می‌ماند (چرا؟).

- این پیاده‌سازی در ادامه نشان داده می‌شود؛ فرض کنید i شناسه‌ی ریسه‌ای باشد که این قطعه کد را اجرا می‌کند.

```
 1  int waiting[N];
 2  int lock;
 3  void acquire(void) {
 4       waiting[i] = 1;
 5       while (test_and_set(lock) == 1 || waiting[i] == 0)
 6            ;
 7       waiting[i] = 0;
 8  }
 9  void release(void) {
10       int j = (i + 1) % N;
11       while (i != j && waiting[j] == 0)
12            j = (j + 1) % N;
13       if (j == i)
14            lock = 0;
15       else
16            waiting[j] = 0;
17  }
```

- هر ریسه‌ای که بخواهد قفل را در اختیار بگیرد مقدار خانه‌ی متناظرش در آرایه‌ی waiting را به یک تغییر می‌دهد (خط چهارم). سپس منتظر می‌شود یا قفل آزاد شود و آن را در اختیار بگیرد (فراخوانی test_and_set(&lock) مقدار صفر را بر گرداند) یا ریسه‌ای صاحب قفل بدون آزاد کردن آن قفل را در اختیارش بگذارد (مقدار waiting[i] صفر شود).
- هر ریسه‌ای که قفل را آزاد می‌کند در آرایه‌ی waiting به دنبال اولین ریسه‌ای می‌گردد که منتظر هست، یعنی خانه‌ی متناظرش در waiting یک هست (خط دهم تا دوازدهم).
- اگر چنین ریسه‌ای موجود بود، یعنی قبل از آنکه j برابر i شود حلقه‌ی خط یازدهم خاتمه یابد، بدون باز کردن قفل آن را در اختیار آن ریسه می‌گذارد (خط شانزدهم).
- اگر چنین ریسه‌ای موجود نبود، یعنی i برابر j شود قفل را باز می‌کند (خط چهاردهم).

- استفاده از ابزارهای همگام‌سازی مثل سمافور دشوار است.
- از این رو روش‌های دیگری برای همگام‌سازی وجود دارند.
- چند مورد از آنها را خواهیم دید.

- در حافظه‌ی تراکنشی (Transactional Memory) قسمت‌های از کد در یک بلوک تراکنش قرار می‌گیرند.
- تغییرات حافظه در یک بلوک به صورت یک تراکنش انجام می‌شوند: دسترسی‌ها به حافظه به صورت محلی انجام می‌شوند و پس از پایان بلوک اگر مقدار هیچ یک از خانه‌هایی از حافظه که بلوک به آنها دسترسی داشته است تغییر نکرده بود، تغییرات به حافظه اعمال می‌شود.
- در غیر این صورت، نتایج تراکنش دور انداخته می‌شود و اجرای تراکنش تکرار می‌شود.
- در زبان C و برای کامپایلر GCC می‌توان به صورت زیر یک بلوک حافظه‌ی تراکنشی تعریف کرد.

```
__transaction_atomic {
    a = a / b;
    c = a - b;
}
```

- حافظه‌ی تراکنشی گاهی توسط سخت‌افزار و گاهی توسط نرم‌افزار پیاده‌سازی می‌شود.

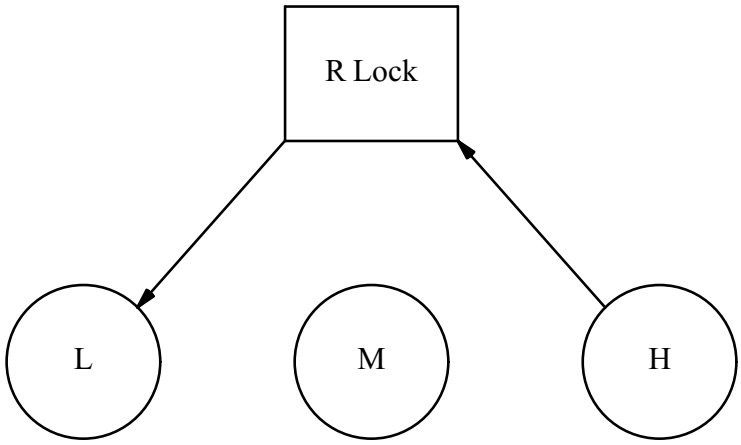
- در بخش‌های قبل درس OpenMP معرفی شده است.
- در OpenMP می‌توان ناحیه‌های بحرانی را به کمک #pragma omp critical مشخص کرد؛ مثال زیر حالت ساده‌ی این کار را نشان می‌دهد.

```
1  int main(void) {
2       #pragma omp parallel
3       {
4            #pragma omp critical
5            {
6                 // Critical section
7            }
8       }
9  }
```

- قسمت‌هایی که به این صورت مشخص می‌شوند به صورت انحصار متقابل اجرا می‌شوند.

- بیشتر زبان‌های برنامه‌نویسی که تا به حال دیده‌اید امری (Imperative) هستند.
- در زبان‌های تابعی (Functional) برنامه‌ها به شکل متفاوتی و در قالب تعدادی تابع بیان می‌شوند.
- چون در این زبان‌ها معمولا از متغیرها برای نگه‌داری حالت استفاده نمی‌شود، نیازی به دسترسی همزمان به متغیرها نیست.

- وارونگی اولویت (Priority Inversion) یک وضعیت ناخواسته هست که در آن به خاطر استفاده از قفل‌ها پردازه‌ای با اولویت بالا منتظر پردازه‌ای با اولویت پایین‌تر می‌شود.
- این وضعیت در ادامه شرح داده می‌شود.
- فرض کنید سه پردازه وجود دارند: پردازه‌ی H، پردازه‌ی M و پردازه‌ی L به صورتی که اولویت پردازه‌ی اول بیشتر از پردازه‌ی دوم و اولویت پردازه‌ی دوم بیشتر از پردازه‌ی سوم باشد.



- زمانبند سیستم عامل همواره پردازنده را در اختیار با اولویت‌ترین پردازه‌ی آماده‌ی اجرا قرار دهد.
- فرض کنید قفل R نیز موجود باشد.
- پردازه‌ی L قفل R را در اختیار می‌گیرد.
- سپس فرض کنید پردازه‌ی M نیز آماده‌ی اجرا باشد. چون اولویت M از L بیشتر است، پردازنده در اختیار M قرار می‌گیرد.
- سپس فرض کنید پردازه‌ی H بخواهد قفل R را در اختیار بگیرد.
- چون R در اختیار L هست، H منتظر می‌شود L قفل را آزاد کند ولی چون اولویت M بیشتر از L هست پردازنده در اختیار پردازه‌ی L قرار نمی‌گیرد تا قفل را آزاد کند.
- در نتیجه، پردازه‌ی H منتظر یک پردازه با اولویت پایین‌تر یعنی M می‌شود.

- یک را برای رفع وارونگی اولویت، به ارث بردن اولویت (Priority Inheritence) هست.
- در به ارث بردن اولویت، اگر یک پردازه‌ی با اولویت بالا A بخواهد قفلی را در اختیار بگیرد که در اختیار پردازه‌ای با اولویت پایین‌تر B است، اولویت پردازه‌ی B به اندازه‌ی پردازه‌ی A افزایش می‌یابد (اولویت A به ارث برده می‌شود).