Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department
1401

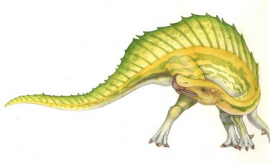
Zeinab Zali

Classical Problems of Synchronization

مشکلات کلاسیک همگام سازی

Classical Problems of Synchronization

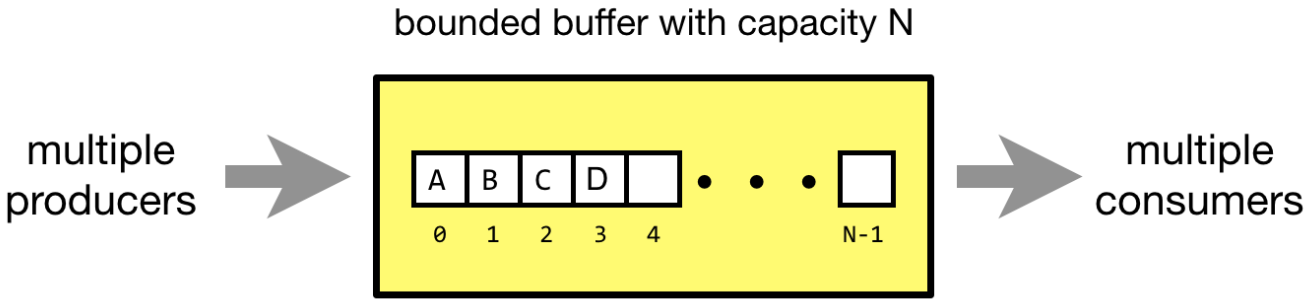
- Classical problems used to test newly-proposed synchronization schemes
  - Bounded-Buffer Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem



3 تا مسئله کلاسیک داریم توی همزمانی که بسیاری از مسائل همزمانی در دنیای واقعی یا... نمونه ای از یکی از این سه تا یا ترکیبی از این سه تا است

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n



-
راه حل با سمافور:
بسیاری از جاها مخصوصا توی کرنل ما مسئله Bounded-Buffer رو داریم
مسئله:
یک بافری با اندازه محدود داریم مثلا اندازه n
تعدادی ترد تولیدکننده داریم که این ها مقادیر رو درون بافر قرار میدن مثل همین A, B ,C, ...
از اون طرف مصرف کننده های زیادی ممکنه در هر آن واحد بخوان از این بافر مصرف بکنن و اطلاعات را از بافر بردارن
نکته: توی مسئله Bounded-Buffer وقتی که مصرف کننده یک اطلاعاتی رو از بافر استفاده میکنه کامل از توی بافر برش میداره و اون تیکه از بافر خالی میشه
سمافور full: برای مصرف کننده است برای اینکه مطمئن بشه مقدار این بافر صفر نیست و یک مقداری توی بافر وجود داره
سمافور empty: برای تولید کننده است برای اینکه تولید کننده مطمئن بشه بافر جای خالی داره
سمافور mutex: از دسترسی همزمان هر پروسسی به بافر جلوگیری بکنه پس برای جلوگیری از دسترسی همزمان پروسس ها به بافر است

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
        ...
        /* produce an item in next_produced */

        ...
        /* add next produced to the buffer */
        ...

}
```

-
قبلا این مسئله رو با CV حل کردیم حالا می خوایم با سمافور این مسئله رو حل بکنیم:

Bounded Buffer Problem (Cont.)

- The structure of the producer process

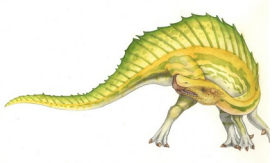
```
while (true) {
      ...
       /* produce an item in next_produced */
      ...
   wait(empty);
   wait(mutex);
      ...
      /* add next produced to the buffer */
      ...
   signal(mutex);
   signal(full);
}
```

-
wait(empty): برای اینکه مطمئن بشه جای خالی وجود داره که در حالت اولیه وجود داره و از این خط رد میشه ولی اگر تولید کننده n تا مقداری توی بافر قرار دارد و مصرف کننده هنوز هیچ کدوم رو مصرف نکرده بود برای n+1 جا مجبوره که بلاک بشه و بمونه توی این خط
بعد از اینکه مطمئن شد بافر خالی است و ازش رد شد
باید از wait(mutex) استفاده بکنه برای اینکه ممکنه دوتا تولید کننده همزمان بخوان مقداری رو توی بافر قرار بدن اینجا مشکلی پیش نیاد پس از دستور wait(mutex) استفاده میکنیم اینجا و بعد از اون مقداری که تولید کردیم می تونه توی بافر قرار بگیره
بعد از اینکه توی بافر مقدار قرار گرفت باید اون mutex رو سیگنال بکنیم که اگر کسی خواست وارد بافر بشه بتونه بشه چون دیگه خودمون با بافر کاری نداریم
و چون الان بافر رو پر کردیم باید سمافور full رو سیگنال بکنیم که اگر مصرف کننده ای منتظر مونده بود که بافر پر بشه حالا بتونه از اون خط wait عبور بکنه و از بافر برداره

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

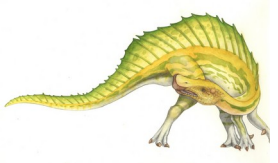
```
while (true) {

    ...
  /* remove an item from buffer to next_consumed */
    ...

   /* consume the item in next consumed */
    ...
}
```



Bounded Buffer Problem (Cont.)

- The structure of the consumer process

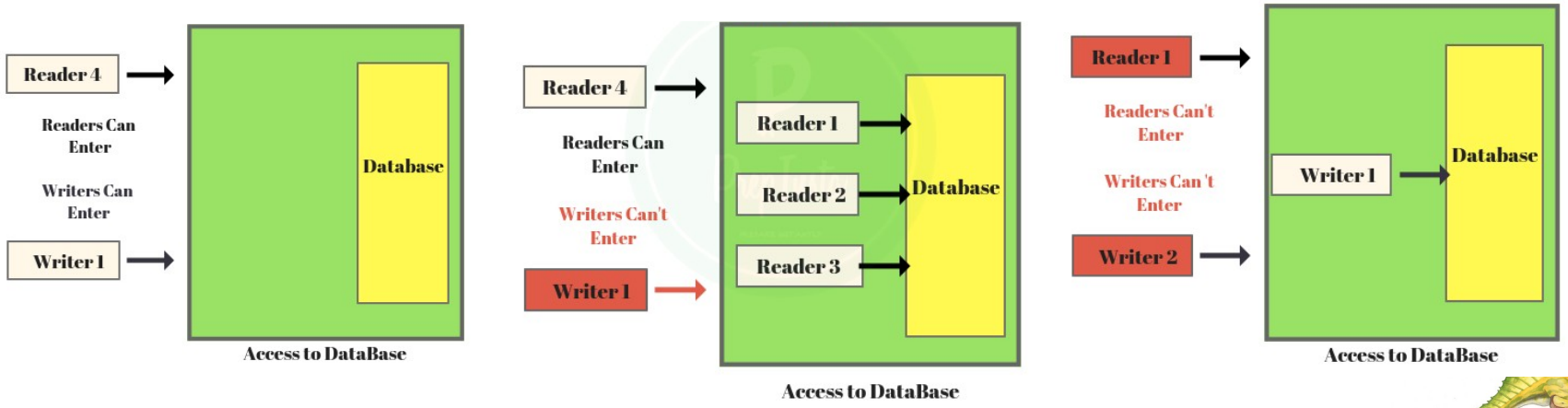
```
while (true) {
    wait(full);
    wait(mutex);
        ...
     /* remove an item from buffer to next_consumed */
        ...
    signal(mutex);
    signal(empty);
        ...
       /* consume the item in next consumed */
        ...
   }
```

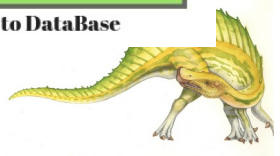


-
wait(full): اگر بافر خالی بود باید منتظر بمونه تا پر بشه پس باید مطمئن بشه چیزی توی بافر وجود داره پس روی full میاد wait رو می ذاره
بعد وقتی که خواست مقداری رو برداره باید مطمئن بشه که کس دیگه ای الان از بافر در حال برداشتن نیست پس روی mutex باید wait بذاره
و بعد از اینکه مقدار رو برداشت باید mutex رو سیگنال بکنیم
اینجا چون یک مقداری برداشته شده از بافر پس یک جایی خالی شده پس empty رو سیگنال می کنیم اینجا

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data set; they do ***not*** perform any updates
  - **Writers** – can both read and write
- Problem – allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time
- Several variations of how readers and writers are considered – all involve some form of priorities





-
مسئله Readers-Writers:
این مسئله معمولا توی مسائل دیتابیس استفاده میشه جایی که ما یک دیتابیس داریم و اپلیکیشن های زیادی همزمان دارن کوئری به دیتابیس میدن
حالا این کوئری می تونه کوئری اپدیت، دیلیت این ها باشه یا ...
وقتی که داریم select می کنیم از یک دیتابیسی صرفا داریم مقادیر رو از دیتابیس می خونیم پس به اون اپلیکیشن هایی که دارن از دیتابیس می خونن Readers گفته میشه ینی استفاده ای که دارن از دیتابیس میکنن خواندن از دیتابیس است ولی وقتی که داریم اپدیت یا دیلیتی انجام میدیم ینی داریم مقادیر داخل دیتابیس رو تغییر میدیم پس اپلیکیشن هایی که دارن مقداری رو پاک می کنن یا مقدار جدیدی رو دارن توی دیتابیس قرار میدن یا یکی از مقادیری که قبلا وجود داشت رو اپدیت می کنن این ها Writers محسوب می شن
پس دو نوع استفاده کننده از همچین ساختاری داریم: Readers و Writers
و این یک فرق اساسی که با مسئله قبلی داره این است که اینجا صرفا این هایی که دارن از دیتابیس استفاده می کنند خواننده و نویسنده هستن اینجوری نیست که وقتی که کسی یک مقداری رو می خونه در واقع برش داره از دیتابیس لزوما و برداشتن به معنی همون Writers کردن میشه ینی تغییر اطلاعات اون دیتابیس
پس Readers صرفا مقداری رو از توی دیتابیس می خونن و قرار نیست اون چیزی رو که می خونن حذفش بکنن از دیتابیس
Writers هم می تونن بخونن و هم بنویسن ولی Readers فقط می تون دیتا رو بخونن و هیچ اپدیتی انجام نمیدن

مسئله: ما یک دیتابیسی اینجا داریم از سمت چپ به سمت راست ببین-->
توی حالت اول هیچکس از دیتابیس استفاده نمیکنه
شکل دوم: یک تعدادی خواننده دارن از دیتابیس می خونن حالا اگر یک خواننده جدید وارد بشه و اون هم بخواد مقداری از دیتابیس بخونه هیچ مشکلی نداره و ما می تونیم بهش اجازه بدیم که از دیتابیس بخونه چون خواننده تغییری توی دیتابیس انجام نمیده در مقابل اگر یک Writers الان اومده باشه ایا Writers می تونه چیزی رو بنویسه توی دیتابیس وقتی که خواننده ها دارن می خونن؟
قطعا نه چون همزمانی Readers و Writers می تونه مشکل ایجاد بکنه پس اینجا باید یک قانونی داشته باشیم که وقتی که reader ها در حال خواندن هستن Writers اجازه نداره که بنویسه توی دیتابیس
شکل سوم: یک Writers است که در حال نوشتن توی دیتابیس است و الان چه Readers بیاد و چه پروسس Writers بیاد به هیچ کدوم ما اجازه نمیدیم که بیاد بخونه یا بنویسه

پس به تعداد زیادی readers همزمان اجازه میدیم از دیتابیس استفاده بکنن اما به readers و Writers همزمان اجازه نمیدیم که از دیتابیس استفاده بکنن

نکته: ما میتونیم حالت های مختلفی از این مسئله رو تعریف بکنیم و هر حالتی که در نظر بگیریم یا به Readers اولویت دادیم یا به Writers اولویت دادیم

این مسئله می تونه حالت های مختلفی داشته باشه حالا میخوایم مثالی که می زنیم روی این حالت زیر باشه:
شکل دوم: که چندتا Readers در حال خواندن هستن و یک Reader جدیدی هم اومده که اونم میخواد از دیتابیس بخونه و یک Writers هم هست و حالا ما اینجا باید تصمیم بگیریم که این readerها وقتی که خواندنشون تموم شد به این reader چهارم هم اجازه بدیم بخونه از دیتابیس و بعد نوبت رو به Writers بدیم یا نه اول نوبت به Writers و بعد به Reader چهارم بدیم --> الان میخوایم حالتی رو بنویسیم که تا وقتی که یک Reader جدید هست اجازه به Reader بدیم که از دیتابیس بخونه و Writers رو نگهش داریم بنابراین کد Reader که داریم می نویسیم اینو باید چک بکنیم که تا وقتی که یک Reader وجود داره به Reader ها اجازه بدیم و همینطور حواسمون هست که Reader ها اجازه دسترسی همزمان دارن ولی Writers نه پس یک جورایی Reader یه جایی باید یک Writers رو سیگنال بکنه اگر Writers اینجا منتظر مونده باشه

Readers-Writers Problem (Cont.)

- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

-
سمافور rw_mutex: مقدار دهی اولیه اش یک است
سمافور mutex:

Readers-Writers Problem (Cont.)

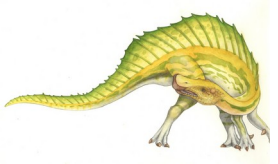
- The structure of a writer process

```
while (true) {

}
```

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
while (true) {
        wait(rw_mutex);
           ...
        /* writing is performed */
           ...
     signal(rw_mutex);
}
```



-
اگر writer در حال نوشتن بود reader نتونه بخونه پس اینجا باید روی mutex یک wait بذاریم
وقتی که writer میخواد کار بکنه بیاد روی rw_mutex ما wait بکنه
بعد از اون مقداری که میخواد رو می نویسه
و در اخر میاد rw_mutex رو سیگنال میکنه

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true){

}
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

```
while (true){
        wait(mutex);
        read_count++;
        if (read_count == 1) /* first reader */
          wait(rw_mutex);
              signal(mutex);
          ...
        /* reading is performed */
          ...
        wait(mutex);
        read count--;
        if (read_count == 0) /* last reader */
            signal(rw_mutex);
        signal(mutex);
}
```

این دستور سیگنال باید بیرون از if باشه چون اگر مقدار کانت باشه 2 و 3 داخل if که نمیشه پس باید این سیگنال رو بیرون بذاریم

-
توی reader:
اول اینکه اگر چندتا reader داشته باشیم همزمان می تونن بخونن پس mutex برای این قسمت که چندتا reader می خوان از دیتابیس استفاده بکنن نداریم
نکته: به reader ها اجازه دسترسی بدیم ولی به writer ها ندیم --> اینجا می تونیم تعداد Reader ها رو بشماریم و اگر تعداد reader ها بیشتر از یک است تا وقتی که اینطوری است نباید اجازه بدیم writer وارد بشه
برای اینکه تعداد Reader ها بشماریم می تونیم از یک متغییر جدیدی به نام read_count استفاده بکنیم
نکته: reader های مختلفی ممکنه همزمان از دیتابیس استفاده بکنند و همزمان بخوان مقدار read_count رو اضافه بکنند پس خود read_count یک متغییر شیردیتا است و خودش رو وقتی که داریم اپدیت می کنیم باید از یک mutex استفاده بکنیم که reader های همزمان نتونن همزمان read_count رو تغییر بدن و باعث اشتباهی اینجا بشه پس اینجا روی mutex میایم wait می کنیم
+ --> الان یک Reader داریم و نباید اجازه بدیم که writer وارد خوندن بشه پس اگر read_count=1 بود روی rw_mutex بیا wait بکن

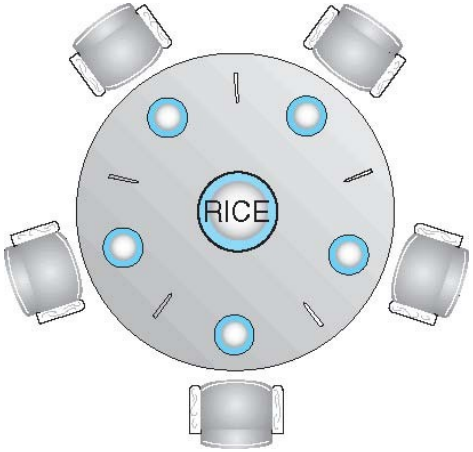
Readers-Writers Problem Variations

- The solution in previous slide can result in a situation where a writer process never writes. It is referred to as the "First reader-writer" problem.
  - **Once a reader is ready to read, no "newly arrived writer" is allowed to read**.
- The "Second reader-writer" problem is a variation the first reader-writer problem that state:
  - **Once a writer is ready to write, no "newly arrived reader" is allowed to read**.
- Both the first and second may result in starvation. leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

-
مسئله رو توی حالتی حل کردیم که اگر یک reader در حال خوندن است و یا یک reader اماده است و میخواد بخونه ما به writer که الان جدید وارد شده اجازه نمی دیم که وارد بشه --> این حالت اولویت رو به خوانندگان میده

اما یک حالت دیگه هم می تونیم برای این مسئله در نظر بگیریم که برعکس است ینی اگر یک writer اماده نوشتن باشه به هیچ reader جدیدی اجازه ندیم که ازدیتابیس استفاده بکنه --> این حالت اولویت رو به نویسندگان میده --> این حالت رو خودت بنویس

Classic problem: Dining-Philosophers Problem



- Philosophers spend their lives alternating thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers
  - Shared data
    - Bowl of rice (data set)
    - Semaphore chopstick [5] initialized to 1

-
مسئله Dining-Philosophers:
5 تا فیلسوف سر یک میز نشستن و 5 تا هم chopsticks دارن که هر فیلسوفی که میخواد غذا بخوره باید دوتا از chopsticks ها رو که دو طرف ظرفش است برداره و شروع به خوردن بخوره و این فیلسوف ها یا در حال فکر کردن هستن یا در حال خوردن ینی وقتی که در حال فکر کردن هستن غذا نمی خورن یا در غیر اینصورت میخوان غذا بخورن یا اینکه chopsticksهاشون ازاد نیست حالا یا یکیش یا هردوش و مجبورن صبر کنن که تا ازاد بشه و بتونن غذا بخون
پس این chopsticks ریسورس های کیریتیکال ما هستن

نکته: برای هر chopsticks یک سمافور در نظر می گیریم پس 5 تا سمافور داریم به نام chopsticks که با یک مقدار دهی اولیه شده

Dining-Philosophers Problem Algorithm

- The structure of Philosopher *i*:

```
do {
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

            //  eat

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

            //  think

} while (TRUE);
```

- What is the problem with this algorithm?

-
اگر برای فیلسوف iام داریم کد می نویسیم باید فیلسوف iام روی chopstick های i , i+1 بیاد wait بکنه و 5 <-- mode هم برای این گذاشتیم چون این 5 تا فیلسوف رو روی دایره فرض کردیم و وقتی هم که خوردنش تموم شد باید سیگنال بکنه و این chopsticks ها رو ازاد بکنه
مشکلی که این روش داره اینه که:
1- بن بست است --> اگر همه این فیلسوف ها i]chopstick] رو بردارن ینی خط اول رو همزمان بیان اجرا بکنن بنابراین همه chopsticks های 1 و 2 و 3 و 4 و 5 برداشته میشه و بعد همه بیان سر خط دوم که میخوان i+1 بیان بردارن هر کدوم که بخواد چوب سمت چپی رو برداره می بینه چوب سمت چپ دست نفر بعدی است و توی این نقطه wait مجبوره صبر بکنه چون chopstickاش ازاد نیست و هیچ وقت هم نمی تونه از این wait رد بشه چون همشون یه جورایی توی این خط wait موندن
پس مشکل اصلی این روش بن بستش است
روش غیر از اینکه بن بست داره می تونه گرسنگی هم داشته باشه
گرسنگی: اگر سمافورها صف هایی که در نظر می گیرن برای پیاده سازی سمافور به صورت FCFS نباشه امکان گرسنگی وجود داره برای یک پروسس--> اگر یک پروسس اولویت بالاتری داشته باشه و ما پروسس ها رو برمبنای اولویتشون خارج بکنیم از صف ها
و گرسنگی موقعی پیش میاد که 0 و 3 همیشه در حال خوردن باشن و مثلا 1 هم بخواد الان غذا بخوره ولی الان یکی از چوب ها ست صفر است و نمی تونه غذا بخوره و توی حالت wait می مونه
حالا وقتی که cpu رو به یک ندادیم و 0 و 3 هم ازاد شدن الان و بعد از اون 4 و 2 مشغول خوردن بشن باعث میشه که 1 نتونه غذا بخوره و اگراین کار باز تکرار بشه ینی 2 و 4 ازاد می کنن chopsticks هاشون رو ولی اولویت 0 و 3 بیشتره و به جای اینکه 1 بیاد و wake بشه به جاش 0 و 3 میاد wake میشه چون باز cpu به 0 و 3 داده میشه اینجا --> پس 1 همیشه گرسنه می مونه
مشکل گرسنگی رو با صف FIFO حلش میکنیم پس یا باید در سمافور صف FIFO وارد بکنیم که این دست ما نیست یا اینکه پس ما باید در کد خودمون به نوعی نوبت دهی رو وارد کنیم که اگر یکی از این ها خواست غذا بخوره و در صورتی که تمایلشو داشت و اگر الان چوب ها ازاد نبود وقتی که ازاد شد بتونه غذا بخوره --> ینی به نوبت در اختیار کس هایی قرار بگیره که تمایل به گرفتن چوب ها رو داشتن

Dining-Philosophers Problem Algorithm (Cont.)

- Deadlock handling
  - Allow at most 4 philosophers to be sitting simultaneously at the table.
    - Using a semaphore initialized to 4
  - Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section.
  - Use an asymmetric solution -- an odd-numbered philosopher picks up first the left chopstick and then the right chopstick. Even-numbered philosopher picks up first the right chopstick and then the left chopstick.
  - But starvation is still possible with these solutions

-
مسئله بن بست رو چجوری حلش بکنیم؟
چندین روش برای بن بست می تونه وجود داشته باشه:
مثلا به بیشتر از 4 تا فیلسوف اجازه ندیم که اصلا بخوان تمایل به خوردنشون رو نشون بدن یا بیشتر از 4 تا رو اجازه ندیم سر میز بشینن --> چرا این روش می تونه مفید باشه؟ چون مشکل وقتی پیش می اومد که همه چوب های سمت راستشون رو برمیداشتن و اگر اخری یا 5 امی برمیداشت باعث میشد این فیلسوف 3ام اگر بخواد چوب سمت چپش رو هم برداره دیگه نتونه این کارو بکنه ولی اگر اخری نباشه 3 می تونه غذا بخوره و دیگه بن بستی پیش نمیاد و به همین ترتیب بقیه هم می تونن غذا --> پس یکی از راه حل ها می تونست این باشه که از یک سمافور تکی با مقدار دهی اولیه 4 استفاده بکنیم و به این ترتیب اگر توی کد قبلی کار زیر را بکنیم بن بست مسئله حل میشه

```
while (true){ wait(5);
    wait (chopstick[i] );
    wait (chopStick[ (i + 1) % 5] );

     /* eat for awhile */

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );
     signal(5)
     /* think for awhile */

}
```

روش دوم:
اصلا مشکل وقتی پیش میاد که هر فیلسوفی اگر یک چوبش ازاد بود بتونه اونو برداره و بعد دنبال این می ره که ببینه چوب بعدی هم ازاد است یا نه پس اگر ما بیایم به نوعی این کار رو انجام بدیم که هر فیلسوفی مجبور باشه هر دوتا چوب رو بیاد چک بکنه و اگر هر دو ازاد بود بتونه برشون داره در این حالت هم باز به بن بست نمی خوریم --> پیاده سازی این رو با مانیتور انجام میدیم
روش سوم:
به نوعی ترتیب برداشتن چوب ها رو به نوعی تغییر بدیم برای فیلسوف ها مثلا فیلسوف های شماره فرد اول چوب سمت راست رو بردارن و بعد سمت چپ رو و فیلسوف های شماره زوج برعکس اول چپ رو بردارن و بعد راست و این باعث میشه که بن بست رخ نده

نکته: تمام راه حل هایی که میدیم باز هم می تونه گرسنگی داشته باشه مگر اینکه ما نوبت دهی رو وارد راه حل بکنیم

Dining-Philosophers application

- What is the real application of dining philosopher problem?
  - A transaction between two accounts

-
این مسئله کجاها پیش میاد؟
یکی از جاهای مهمی که این مسئله پیش میاد توی تراکنش های بانکی است
مثلا اگر بخوایم از یک کارتی به یک کارت دیگه ای واریز داشته باشیم --> وقتی که می خوایم واریز رو انجام بدیم این عملیات واریز نیاز به ازاد بودن این دوتا حساب کاربری داره مثلا ممکنه در همین لحظه از حساب A به حساب X یک مقداری رو وارد میکنیم ممکنه Y در حال برداشت از حساب X باشه یا باز ممکنه یک Z در حال واریز به حساب X
حالا ممکنه اطلاعات این حساب ها اشتباه بشه چون توی اون لحظه ای که داره مقدار حساب X داره زیاد میشه ممکنه همون لحظه داره کم میشه برای اینکه به Y واریز بکنه و این کم کردن و زیاد کردن اگر همزمان انجام بشه ممکنه که یکیشون miss بشه و جواب نهایی اشتباه باشه پس باید دنباله ای این کار انجام بشه مثلا اول واریز انجام بشه و بعد برداشت انجام بشه