Operating Systems

Isfahan University of Technology
Electrical and Computer Engineering Department

Zeinab Zali

Deadlock

System Model

- System consists of resources
- Resource types $R_1, R_2, \ldots, R_m$

  Examples: CPU cycles, memory space, I/O devices
- Each resource type R_i has W_i instances.
- Each process utilizes a resource as follows:
  - **request**
  - **use**
  - **release**

-
به طور کلی توی بحث Deadlock ما یک فرضیاتی در مورد سیستم داریم و با این مشخصات می ریم که مسئله Deadlock رو به خوبی بشناسیم و بهش فکر کنیم که چجور برخوردی باید باهاش داشته باشیم
فرض میکنیم: یک سیستمی شامل یک تعدادی نوع ریسورس است اینجا مثلا m نوع مختلف از ریسورس داریم که با R1 تا Rm نشون دادیم
ریسورس ها می تونن مثلا مموری باشن یا دیوایس I/O باشن یا cpu باشه یا..
ممکنه از یک ریسورسی هم بیشتر از یک عدد داشته باشیم مثلا هر ریسورس i براش Wi نمونه داشته باشیم مثلا وقتی که ما یک سمافور داریم و سمافورمون مقدارش بیشتر از یک است...
هر پروسسی توی اجراش یک تعدادی ریسورس برای اجرا نیاز داره ولی هر موقعی که نیاز به یک ریسورس پیدا میکنه اون رو ریکوست میده از سیستم ینی فرضمون بر اینه که یک سیستمی هست که ریکوست اون پروسس می فهمه که یک ریسورس است و بعد از ریسورس اگر سیستم بهش جواب داد و اون ریسورس رو بهش داد از ریسورس استفاده میکنه و بعد از استفاده هم ازادش میکنه پس اینم جز فرضیات سیستممون است

Deadlock in Multithreaded Application

- Two mutex locks are created an initialized:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex,NULL);
pthread_mutex_init(&second_mutex,NULL);
```

-
توی این کد ما یک مدل Deadlock رو می بینیم:

Deadlock in Multithreaded Application

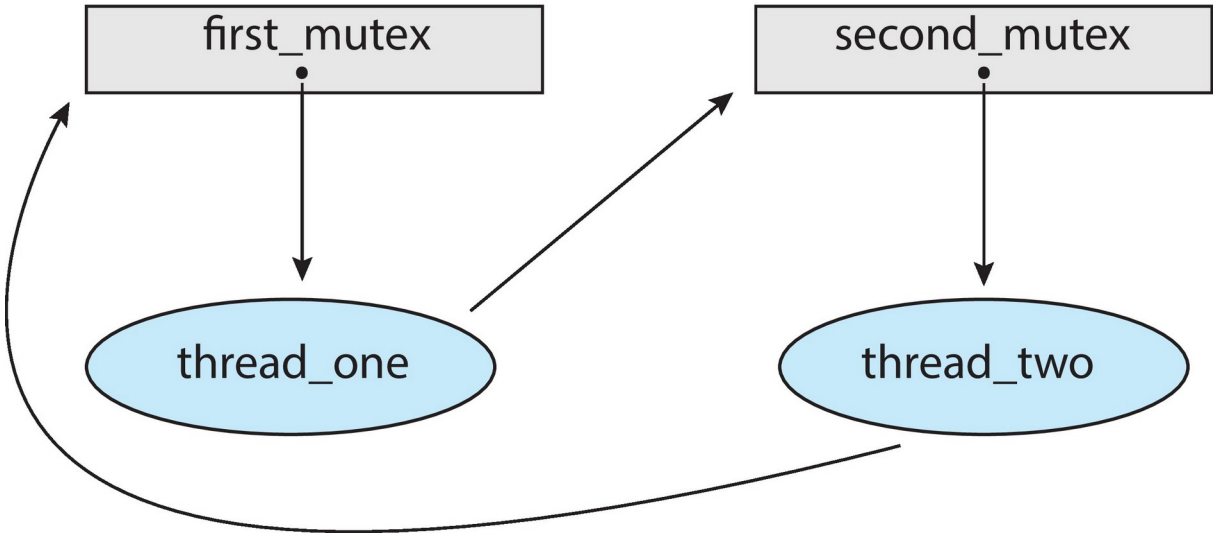
```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

ادامش:

دوتا تردی است که دارن از دوتا mutex استفاده می کنن ولی برعکس همدیگه ینی یکیشون اول mutex اول رو قفل میکنه بعد دومی رو و ترد دومی اول دومی رو قفل میکنه و بعد اولی رو که باعث میشه که این دوتا ترد اگر خط اول رو اجر کنند ینی یکیشون mutex اولی و دومی mutex دومی رو بگیره و بعد بخوان mutex بعدی رو بگیرن بخاطر این که mutex بعدی در دست ترد مقابل است دیگه اون خط قفل میشه و به بن بست می خوریم

Deadlock in Multithreaded Application

- Deadlock is possible if thread 1 acquires **`first_mutex`** and thread 2 acquires **`second_mutex`**. Thread 1 then waits for **`second_mutex`** and thread 2 waits for **`first_mutex`**.
- Can be illustrated with a **resource allocation graph**:



-
به همچین حالتی می تونیم صفحه قبل رو نشون بدیم:
وقتی که یک فلش از mutex به ترد زدیم ینی این mutex الان در اختیار این ترد است ینی تونسته ازش رد بشه مثلا first_mutex الان در اختیار ترد one است و ازش رد شده
و وقتی که یک فلش از ترد به mutex زدیم ینی الان این mutex رو درخواست داده که روش wait بکنه ولی نتونسته ازش رد بشه چون mutex الان دست ترد دیگری است مثلا ترد one الان درخواست داده که روی second_mutex بیاد wait بکنه
پس توی این شکل می تونیم اینو ببینیم که به علت انتظاری که این دوتا ترد برای mutex دارن که دست ترد مقابل هستن در یک بن بست افتادن
به این چیزی که اینجا کشیدیم که به صورت گراف است resource allocation graph میگیم

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

- **Mutual exclusion**: only one process at a time can use a resource
- **Hold and wait**: a process holding at least one resource is waiting to acquire additional resources held by other processes
- **No preemption**: a resource can be released only voluntarily by the process holding it, after that process has completed its task
- **Circular wait**: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

-
چه موقع ممکنه این Deadlock که گفتیم اتفاق بیوفته؟
در یک سیستمی با اون فرضیاتی که چند صفحه قبل گفتیم در صورتی که این 4 تا شرط همشون با هم برقرار باشه ما در Deadlock می افتیم ولی اگر حتی یکی از این شرط ها برقرار نباشه چیزی به نام Deadlock پیش نمیاد شاید مثلا ما یه جایی فک کنیم Deadlock پیش اومده ولی Deadlock پیش نمیاد پس این 4 تا شرط حتما باید در کنار هم حتما وجود داشته باشه تا Deadlock اتفاق بیوفته
1- Mutual exclusion: پروسس ها حتما در حال استفاده از ریسورس های شیر باشن و ریسورس های شیر به این معنی است که فقط یک پروسس می تونه در یک لحظه زمانی از ریسورس استفاده بکنه ینی مسئله انحصار نامتقابل رو داریم ینی نمیشه دوتا پروسس همزمان ریسورسی رو داشته باشن
2- Hold and wait: ینی شرایطی که یک ترد یا یک پروسس یک ریسورسی رو در اختیار داره مثل صفحه قبل ینی hold کرده و بعد منتظره برای یک ریسورس دیگه ای که الان ازاد نیست ینی wait کرده
3- No preemption: ینی اگر ما شرط کرده باشیم توی سیستممون که وقتی یک پروسس یک ریسورس رو گرفت ما از خارج اون پروسس نمی تونیم اون ریسورس رو ازش بگیریم بلکه خود اون پروسس باید به صورت اختیاری وقتی که کارش با ریسورس تموم شد ریسورس رو برگردونه به سیستم پس ما نمی تونیم یا سیستم عامل یا یک برنامه بالایی نمی تونه ریسورس ها رو بگیره از یک پروسسی --> اگر می تونستیم بگیریم می شد اون حالت بن بست رو شکست
4- Circular wait: ینی اگر این سه شرط اول موجود باشه ولی نحوی اجرای پروسس ها به صورتی باشه که این Circular wait اتفاق نیوفته پس بن بست هم پیش نمیاد پس این حالت Circular wait هم حتما باید باشه ینی به طور کلی اگر n تا پروسس داشته باشیم P0 منتظر ریسورسی باشه که دست P1 و بعد P1 منتظر ریسورسی باشه که دست P2 و P2 منتظر ریسورسی باشه که دست P3 و به همین ترتیب تا Pn که Pn منتظر ریسورسی است که دست P0 است ینی این ها توی یک دور افتاده باشن

Resource-Allocation Graph

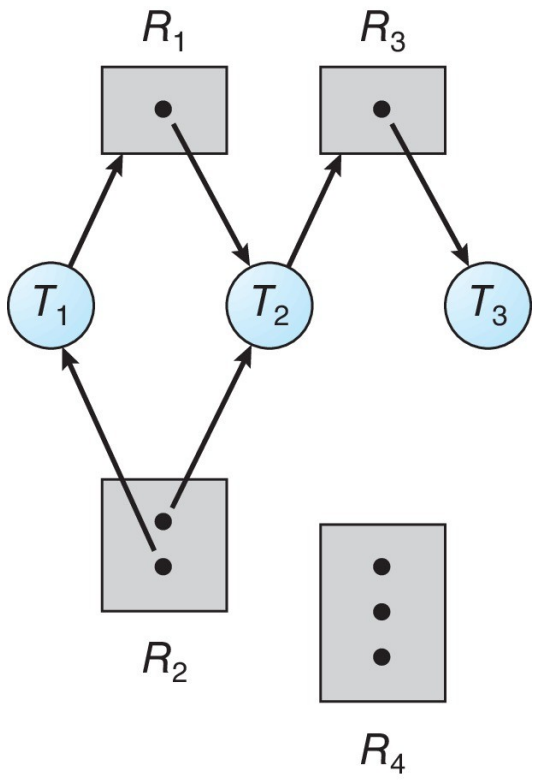
A set of vertices V and a set of edges E.

- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- **request edge** – directed edge $P_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow P_i$

-
برای اینکه این مسئله رو بتونیم بررسی کنیم و بتونیم تشخیص بدیم بن بست رو یک گرافی می سازیم که نشون میده که ریسورس ها به چه صورت به پروسس ها اختصاص داده شدن
صفحه بعدی...

Resource Allocation Graph Example

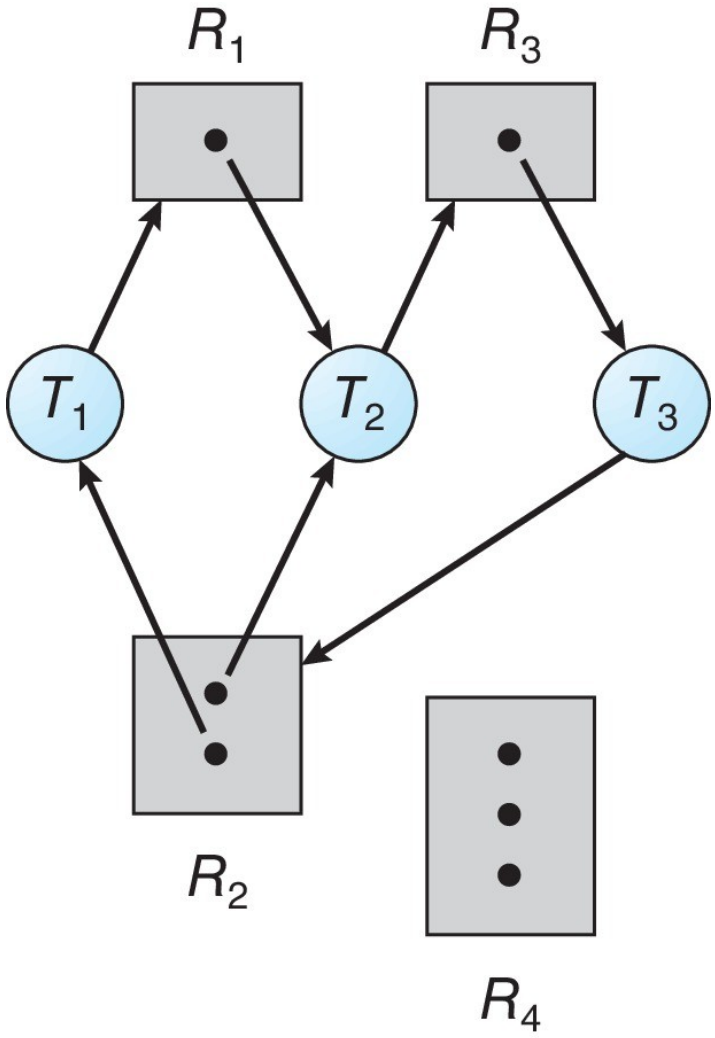
- One instance of R1
- Two instances of R2
- One instance of R3
- Three instance of R4
- T1 holds one instance of R2 and is waiting for an instance of R1
- T2 holds one instance of R1, one instance of R2, and is waiting for an instance of R3
- T3 is holds one instance of R3



-
اگر تردی یک ریسورس رو در اختیار گرفته باشه یک فلش از ریسورس به ترد وصله
و اگر تردی منتظر یک ریسورس باشه یک فلش از ترد به ریسورس وصله

توی شکل نشون میدیم که از هر ریسورسی چندتا نمونه داریم مثلا از R1 و R3 یک نمونه داریم
ولی از R2 دوتا نمونه و از R4 سه تا نمونه داریم

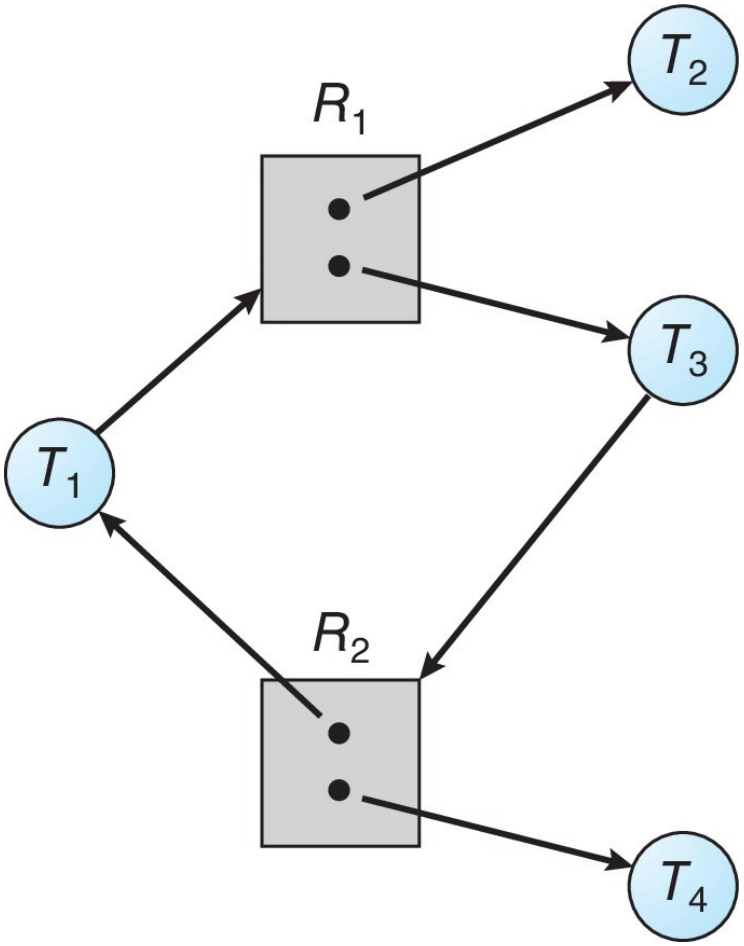
Resource Allocation Graph With A Deadlock



ایا تردهای این گراف توی بن بست افتادن؟
و چرا توی بن بست افتادن؟ و چجوری از روی این گراف بفهمیم؟

T2 منتظر یک ریسورسی است که دست T3 است و T3 هم منتظر یک ریسورسی است که الان دست T2 یا T1 است
توی این گراف یک دور می بینیم توی یال هاش و میشه از اینجا فهمید که توی بن بست افتادیم ینی ترد T2 و T3 درگیر یک بن بستی شدن

Graph With A Cycle But No Deadlock



ایا اینجا هم بن بست داریم؟ چون اینجا هم یک دور داریم

نه بن بست نداریم چون اینجا از ریسورس 2 ما دوتا نمونه داریم و T4 الان ریسورس دیگه ای برای اجرا نیاز نداره یا T2 هم همینطور
پس T4 , T2 می تونن وقتی که این R1 , R2 دستشون هست اجراشون رو ادامه بدن تا اجراشون تموم بشه و وقتی که T2 و T4 تموم بشن ریسورس هایی که دستشون است رو ازاد می کنن
اگر اینجا R2 ازاد بشه این T3 منتظر R2 بوده پس ما می تونیم R2 بهش بدیم چون R2 توسط T4 یکی از نمونه هاش ازاد شده و الان T3 هم R1 داره و هم R2 داره و به چیز دیگه ای هم نیاز نداره پس T3 هم می تونه اجرا بشه و بعد اجراش تموم بشه و بعد از اینکه اجرای این تموم شد R1 ازاد میشه و T1 می تونه با گرفتن R1 در حالی که R2 هم در اختیار داره اجرا بشه و اجرای این هم تموم بشه

Basic Facts

- If graph contains no cycles ⇒ no deadlock
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

-
پس اگر از هر ریسورسی فقط یک نمونه داشته باشیم و دور توی گرافمون ببینیم اینجا حتما deadlock اتفاق افتاده
ولی اگر از هر ریسورسی تعداد زیادی نمونه ینی بیشتر از یکی نمونه داشته باشیم با وجود اینکه توی یک گراف ممکنه دور ببینیم ممکنه که deadlock باشه و باید اینو چک بکنیم که deadlock هست یا نه ینی اینجا لزوما دور به معنای deadlock نیست

Methods for Handling Deadlocks

- Ensure that the system will never enter a deadlock state:
  - Deadlock prevention پیشگیری
  - Deadlock avoidance اجتناب
- Allow the system to enter a deadlock state and then recover (detect and recover)
  - Rollback? Is it useful? Is it possible?
- Ignore the problem and pretend that deadlocks never occur in the system.
  - Ostrich

Detect and recover

-
چجوری می تونیم deadlock رو کنترلش بکنیم؟
راه حل اول به طور کلی این است که ما کلا به نوعی سیستم رو طراحی بکنیم که مطمئن بشیم هیچ وقت اصلا وارد deadlock نمی شیم --> این که چجوری این کار رو انجام بدیم می تونه به دوتا روش باشه:
1- Deadlock prevention: توی این حالت ما باید اون فرضیاتی که در مورد سیستممون داشتیم که اگر اون فرضیات وجود داشته باشه بن بست پیش میاد رو یه جوری نفیشون کنیم ینی کلا پیشگیری کنیم که سیستم شرایطی داشته باشه که منجر به بن بست بشه
2- Deadlock avoidance: اما توی این حالت این پیشگیری رو نکردیم قبلا ولی وقتی که پروسس ها یا تردها دارن درخواست ریسورس میدن ما به نوعی به این ریسورس ها جواب میدیم که مطمئن باشیم که هیچ وقت منجر به بن بست نخواهد شد
هدف 1 و 2 این است که از این که کلا سیستم بره توی حالت بن بست جلوگیری بکنه
مثال:
در مورد کرونا: حالت پیشگیری حالتی است که از واکسن استفاده میکنیم توی کرونا که باعث میشه که اون ویروس واکسن کلا نتونه عمل بکنه روی بدن پس این ویروسه اینجا وجود داره ولی ما واکسنی می زنیم که به نوعی عملکرد اون ویرویس روی بدن رو مختل می کنه با تولید اون انتی بادی ها --> پس توی Deadlock prevention می خوایم در واقع روش هایی رو پیشنهاد بدیم که مطمئن باشیم که سیستممون با اون روش ها دیگه هیچ وقت دچار بن بست نمیشه
حالت اجتناب: ینی این که ما بدونیم که ویروس ها وجود دارند و از بینشون نبردیم ولی ما خودمون یه کارهایی بکنیم که این ویرویس وارد بدنمون نشه ینی خودمون اجتناب بکنیم از شرایطی که ممکنه ویروس وارد بدنمون بشه ینی مثلا ماسک بزنیم و فاصله رو رعایت بکنیم
نکته: کم کم حالت prevention منتهی به این میشه که کلا ویروسی موجود نباشه پس Deadlock prevention حالتی است که ما سیستم رو به نوعی طراحی بکنیم که اصلا به بن بست نخوره ولی Deadlock avoidance ینی سیستم امکان به بن بست رسیدن داره ما وقتی داریم ریسورس ها رو به پروسس ها میدیم به نوعی این ریسورس ها رو بدیم که در اینده دچار بن بست نشیم

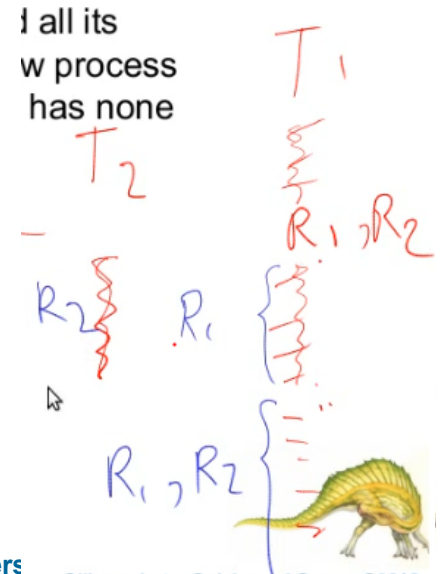
برخورد دوم با بن بست:
که نه پیشگیری کنیم و نه اجتناب کنیم و بذاریم که اگر پیش اومد بن بست اتفاق بیوفته و اگر بن بست اتفاق افتاد بتونیم تشخیص بدیم و recover بشیم از اون حالت ینی بتونیم یه جوری از اون حالت بن بست دربیایم ینی شاید مثلا ما توی سیستم یک Rollback داشته باشیم که باعث بشه که اون پروسس ها یا تردها که گیر افتادن توی همدیگه از اون حالت بن بست دربیان
که این موضوع بحث دارد --> که ایا درسته که بیوفتیم توی بن بست و بعد بخوایم recover بشیم و اصلا مفید است؟ یا اینکه اصلا امکان پذیر است؟

برخورد سوم:
روش Ostrich --> ینی سرش رو کرده توی برف :) ینی این که ما هیچ برخورد خاصی با بن بست نداشته باشیم خواست اتفاق بیوفته بذاریم بیوفته و خواست سیستم ما رو دچار هر مشکلی بکنه بذاریم بکنه
خود سیستم عامل به صورت مستقل داره روش سوم رو استفاده میکنه ینی کاری نداره که پروسس های سیستم جوری برخورد کردن که افتادن توی بن بست یا نیوفتادن کلا کاری نداره

Deadlock Prevention

Invalidate one of the four necessary conditions for deadlock:

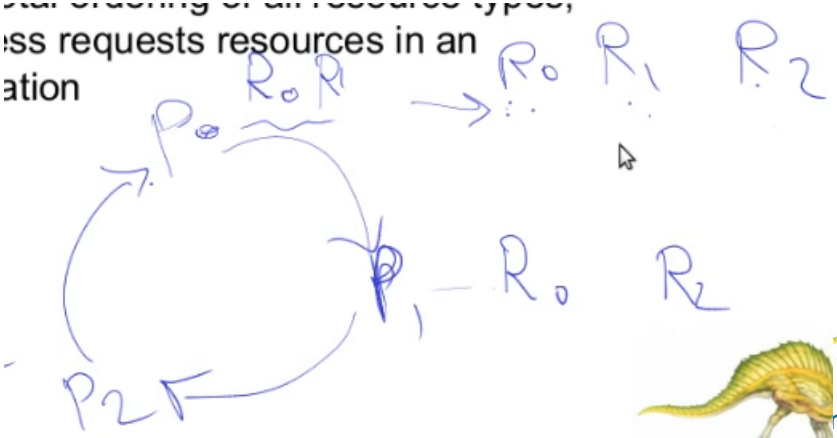
- **Mutual Exclusion** – not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait** – must guarantee that whenever a process requests a resource, it does not hold any other resources
  - Require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it.
  - Low resource utilization; starvation possible

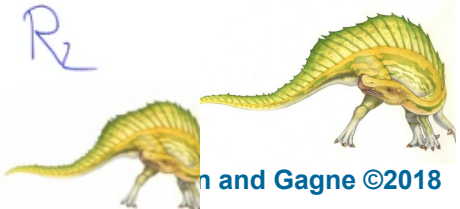


-
اگر بخوایم Deadlock Prevention داشته باشیم باید یکی از اون 4 شرطی که اونجا گفتیم برای اتفاق افتادن بن بست لازم هست رو نقض بکنیم توی سیستم و اگر هر کدوم نقض بشه دیگه مطمئنیم بن بست پیش نمیاد پس یکیش کافیه:
Mutual Exclusion: چجوری می تونیم اینو نقض بکنیم؟ اینکه اصلا ریسورس شیری نداشته باشیم یا اینکه ریسورس ها اگر توی یک سیستم همیشه read-only باشن هیچ وقت مشکلی پیش نمیاد ینی اشکالی نداره که افراد همزمان یا پروسس ها همزمان از اون ریسورس بخونن فقط اپدیت کردنشون مشکلی داره اما این روش منطقا معنی نداره ینی ما توی سیستم ها حتما ریسورس های شیری داریم که بحث Mutual Exclusion براشون مهمه چون تغییر داریم روی اون ریسورس های شیر انجام میدیم و بحثمون فقط خواندن از اون ریسورس ها نیست که مهم نباشه برامون Mutual Exclusion --> این روش ناکارامد بود
Hold and Wait: چجوری اینو نقض بکنیم؟ یکی از راه هاش اینه که اگر T1 به ریسورس R1 , R2 نیاز داشت ریسورس R1 , R2 کلا با هم بگیره نه اینکه اول یکیشو بگیره و بعد بعدی رو بگیره پس در کل سیستم جوری باشه که اگر ریسورس های یک پروسسی همشون با هم موجود بود بهشون بده وگرنه هیچ کدوم رو بهش نده --> مشکلی که این روش داره اینه که: اولا resource utilization رو اوردیم پایین مثال توی شکل مثلا توی T1 ما اولش فقط نیاز به R1 داشتیم ولی چون می خواستیم این شرط رو برقرار بکنیم بهش R2 رو دادیم در صورتی که توی بخش اول اصلا بهش نیاز نداشت ولی T2 که بهش نیاز داشت دیگه نمی تونه R2 رو بگیره مشکل دوم هم اینه که: T1 هم R1 , R2 باید داشته باشه و ممکنه هیچ وقت این دوتا با همدیگه همزمان ازاد نشن پس گرسنگی پیش میاد برای T1

Deadlock Prevention (Cont.)

- **No Preemption** –
  1) If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  2) Preempted resources are added to the list of resources for which the process is waiting
  3) Process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait** – impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration





-
No Preemption: چجوری اینو نقض بکنیم؟
راه حل: اگر یک پروسسی یک ریسورس هایی رو گرفته بود در اختیار خودش و منتظر یک ریسورس دیگه ای شد ما اون ریسورس دیگه رو بهش نمیدیم یا اینکه ریسورس های قبلیش رو هم ازش پس می گیریم --> توی این حالت Hold and Wait رو می شکنیم ینی اگر Hold and Wait اتفاق افتاد اون چیزایی که قبلا پروسس hold کرده بود رو preempt می کنیم مثلا توی مثال قبلی که T1 نیاز به R2 داشت و منتظرش بود نمی تونست پیش بره فعلاولی R1 دستش بود پس ما میایم R1 رو هم ازش میگیریم و بعد وقتی که R1 , R2 هر دو ازاد شد بهش پس میدیم
پس راه حل کلی این است که:
1- اگر یک پروسسی یک ریسورس هایی رو درخواست داد که دست پروسس های دیگه ای است و الان نمی تونه اون ریسورس رو بگیره ینی رسیده به حالت wait نگاه کنیم قبلا چه ریسورس های دیگه ای رو گرفته بود و بیایم اون ها رو ازاد بکنیم
2- و بعد اون ریسورس های قبلی رو که ازش گرفتیم اضافه میکنیم به لیست ریسورس هایی که الان این پروسس برای ادامه کار بهشون نیاز داره
3- پروسس وقتی می تونه دوباره ریستارت بشه که همه ریسورس هاش رو از جمله ریسورس های قبلی و از جمله ریسورس های جدیدی که درخواست داده یه جورایی ازاد بشه
Circular Wait: چجوری اینو نقض بکنیم؟ اگر اینجا این قید رو برای پروسس ها بذاریم که با هر ترتیبی نمی تونین مثلا این R0 , R1 ,R2 رو بگیرین
مثلا ریسورس ها رو سورت میکنیم مثلا میشه R0 ,R1, R2 و بعد پروسس ها مجبور میکنیم که فقط با همین ترتیب می تونن ریسورس بگیرن ینی مثلا اگر R0 رو گرفتن می تونن بعد روی R1 درخواست بدن و بعد اگر R1 رو گرفتن می تونن روی R2 هم درخواست بدن مثلا اگر P1 بهR0 , R2 نیاز داشت با اینکه اول نیازش روی R2 است و بعد روی R0 ولی مجبور باشه اول R0 رو بگیره و بعدا R2 رو بگیره
این روش هم داره resource utilization رو میاره پایین چون پروسس ها رو مجبور کردیم که با یک ترتیب خاصی ریسورس ها رو درخواست بدن ولی در عین حال قضیه به بدی اون hold and wait نیست

Circular Wait

- Invalidating the circular wait condition is most common.
- Simply assign each resource (i.e. mutex locks) a unique number.
- Resources must be acquired in order.
- If:

`first_mutex = 1`
`second_mutex = 5`

code for **`thread_two`** could not be written as follows:

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

-
روشی که صفحه قبلی گفتیم که مانع Circular Wait بشه باید توی پیاده سازیش دقت بکنیم:
مثلا اینجا میتونیم mutex ها رو شماره گذاری بکنیم و باید براساس همین شماره ها گرفته بشن
مثلا توی کد روبرو قسمت + برعکسه اول، دومی است و بعد اولی در صورتی که اول باید اولی رو بگیره و بعد دومی رو پس باید جای اون دو خط رو عوض بکنیم که این ها توی Circular Wait نیوفتن

Deadlock Example with Lock Ordering

```
void transaction(Account from, Account to, double amount)
{
   mutex lock1, lock2;
   lock1 = get_lock(from);
   lock2 = get_lock(to);
   acquire(lock1);
      acquire(lock2);
         withdraw(from, amount);
         deposit(to, amount);
      release(lock2);
   release(lock1);
}
```

Transactions 1 and 2 execute concurrently. Transaction 1 transfers $25 from account A to account B, and Transaction 2 transfers $50 from account B to account A

-
این تابع میخواد از اکانت from به اکانت to یک جابه جایی داشته باشه
در ابتدا باید هر دوی این اکانت ها اول قفل بشن و بعدا عملیات انجام بشه

Deadlock Avoidance

Requires that the system has some additional ***a prior*** information available

- Simplest and most useful model requires that each process declare the ***maximum number*** of resources of each type that it may need
- The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
- Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes

-
یکی از راه های برخورد با بن بست اجتناب از بن بست است ینی این که سیستم به نوعی ریسورس ها رو اختصاص بده به پروسس هایی که به اون ها نیاز دارن که از افتادن توی حالت بن بست جلوگیری بکنه
اگر بخوایم همچین کاری انجام بدیم ینی اگر بخوایم یک الگوریتمی برای سیستم ارائه بدیم باید یک مقدار اطلاعات بیشتری از پروسس ها داشته باشیم مثلا بدونیم که:
1- ماکزیممم درخواست هر پروسسی روی ریسورس های موجود سیستم چقدر خواهد بود پس یک اطلاع خیلی مفیدی که می تونیم ازش استفاده بکنیم اینه که بدونیم که هر پروسسی توی سیستم حداکثر چقدر ریسورس نیاز داره در این صورت باید به طور کلی بدونیم که کل ریسورس های سیستم چقدره که اینو همیشه مدنظر داشتیم ولی اینو قبلا مطرح نمی کردیم که از ابتدا یک پروسسی عنوان کنه که تا انتهای اجراش ماکزیمم چقدر ریسورس نیاز داره از هر کدوم از ریسورس های سیستم
2- الگوریتم Deadlock Avoidance که میخوایم بگیم: به این صورت کار میکنه که هر بار چک میکنه استیت سیستم از لحاظ ریسورس هایی که allocate شده و ریسورس هایی که داره درخواست میشه چک میکنه که یک حالت استیتی باشه که استیت امن است از لحاظ ما و چک میکنه همیشه اینو و جلوگیری می کنه از اینکه شرایطی پیش بیاد که سیستم توی حالت circular-wait بیوفته
3- ...

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- System is in **safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

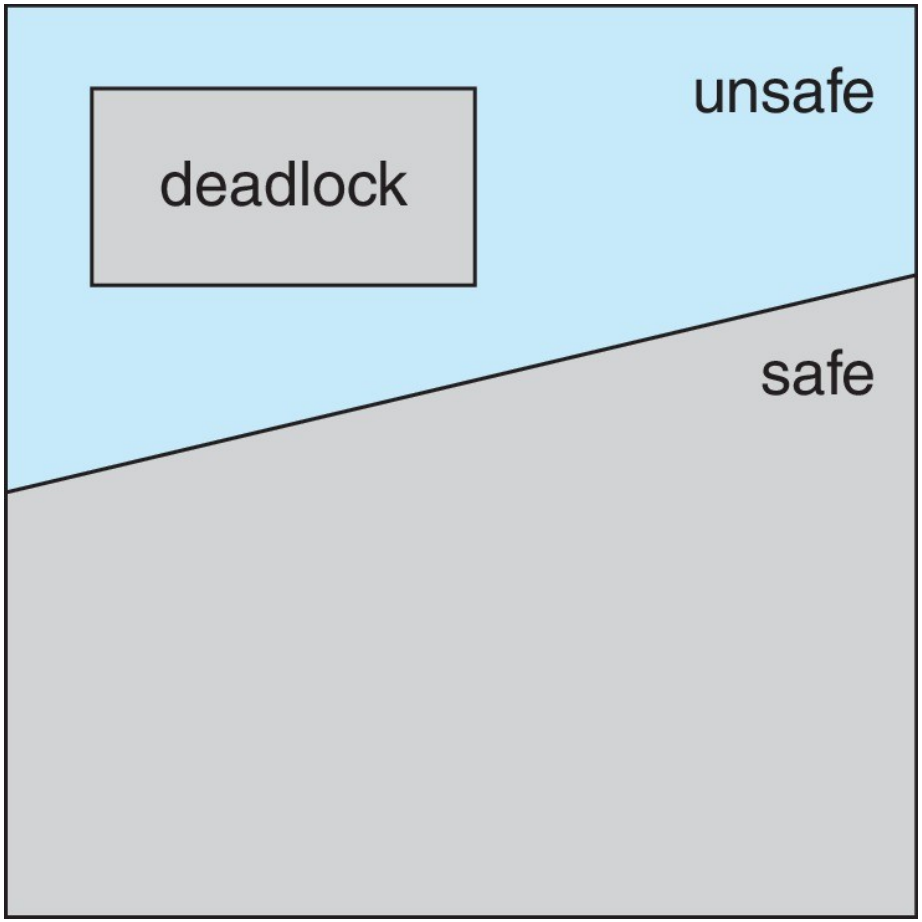
-
الگوریتم:
به طور کلی این الگوریتم اینطوری فرض میکنه هر بار که پروسسی ریکوست داد روی ریسورسی سیستم باید تصمیم بگیره که ایا این ریکوست فعلی اون پروسس رو باید بهش جواب بده یا نه و در صورتی جواب میده که اگر جواب داد سیستم در حالت امن باقی می مونه
پس به طور کلی یک وضعیت امنی برای سیستم تعریف میکنیم و بعد الگوریتم ما گارانتی میکنه که هر موقع که پروسسی درخواستی داشت در صورتی درخواستشو پاسخ میده که سیستم در حالت امن باقی بمونه
صفحه 21 ...
safe state: اگر یک دنباله ای از پروسس ها به صورت <P1, P2, …, Pn> داشته باشیم و این شرایط وجود داشت باشه که: به ازای هر کدوم از Pi ها ریسورس هایی که Pi می تونه ریکوست بده ریسورس هایی باشه که الان در اختیارشه ینی ازاده و سیستم می تونه بهش بده یا اینکه اگر در اختیارش نیست دست پروسس هایی باشه که قبل از خودش قرار دارن --> این شرایط برای همه Pi باید صادق باشه که ما بگیم این استیت سیستممون به طور کلی امن است
این چه مزیتی داره؟ می تونم اثبات بکنیم که اگر سیستم امن باشه و ما همیشه بین استیت های امن جابه جا بشیم توی سیستم در این صورت مطمئنیم هیچ وقت به بن بست نمی خوریم چرا؟

Basic Facts

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

-
پس:
اگر یک سیستم در حالت امن باشه هیچ وقت بن بست نداره
اگر توی حالت unsafe باشه ممکنه بن بست داشته باشه
پس بهترین کار برای اجتناب این است که همیشه به نوعی عمل بکنیم که مطمئن باشیم که سیستم همیشه توی حالت safe است و هیچ وقت توی حالت unsafe نمی افته در این صورت می تونیم تضمین کنیم که از بن بست اجتناب کردیم

Safe, Unsafe, Deadlock State



ادامه..
به نوعی این safe رو تعریف میکنیم که اگر سیستم در حالت Safe باشه مطمئنیم بن بستی هیچ وقت رخ نمیده ولی اگر سیستم در حالت Unsafe باشه احتمال وقوع بن بست وجود داره پس الگوریتم سعی میکنه سیستم رو توی حالت safe نگه داره که بتونیم مطمئن باشیم که به بن بست نمی خوریم

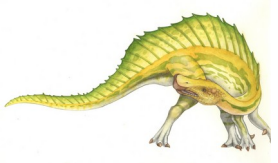
Avoidance Algorithms

- Single instance of a resource type
  - Use a resource-allocation graph
- Multiple instances of a resource type
  - Use the Banker's Algorithm

-
اگر بخوایم الگوریتم اجتناب از بن بست داشته باشیم برای سادگی می تونیم توی دو حالت مسئله رو بررسی کنیم:
1- حالتی که از هر ریسورسی فقط یک نمونه موجود باشه --> اینجا اگر از هر ریسورسی توی سیستم فقط یک نمونه داشته باشیم از گراف resource-allocation استفاده میکنیم تا الگوریتم را ارائه بکنیم و از بن بست اجتناب بکنیم
2- اما اگر از ریسورسی بیشتر از یک نمونه داشتیم باید از الگوریتم Banker استفاده بکنیم

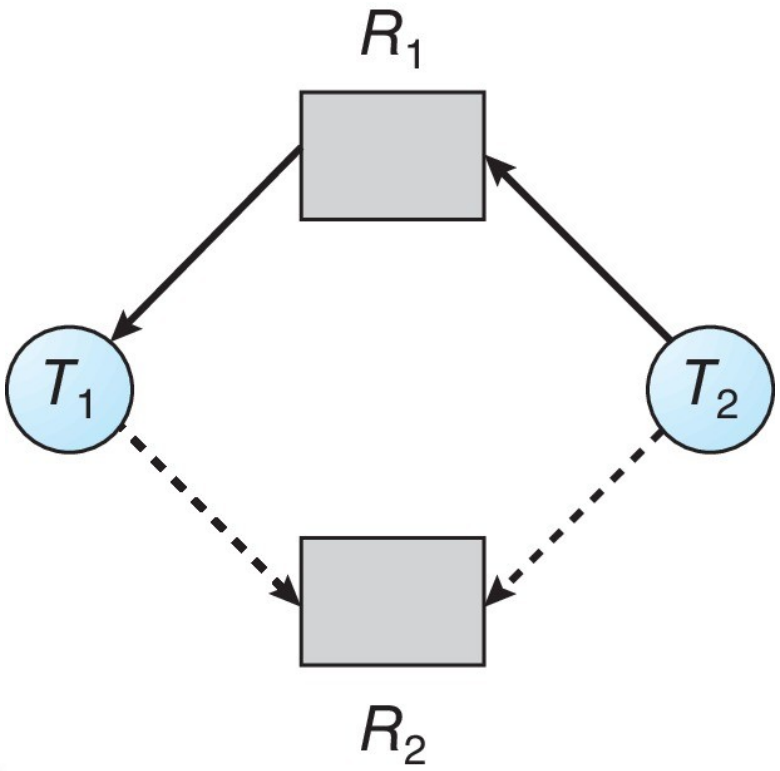
Resource-Allocation Graph Scheme

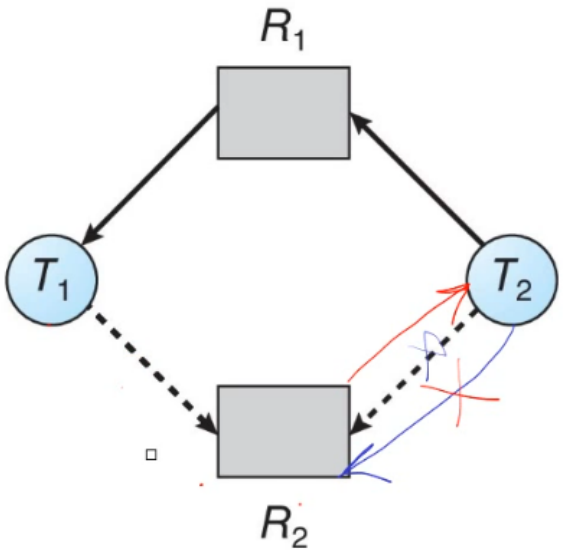
- **Claim edge** $P_i \rightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed *a priori* in the system



-
اگر بخوایم از بن بست اجتناب بکنیم باید یکسری اطلاعات بیشتری داشته باشیم توی فرضیاتمون:
صفحه بعدی...

Resource-Allocation Graph





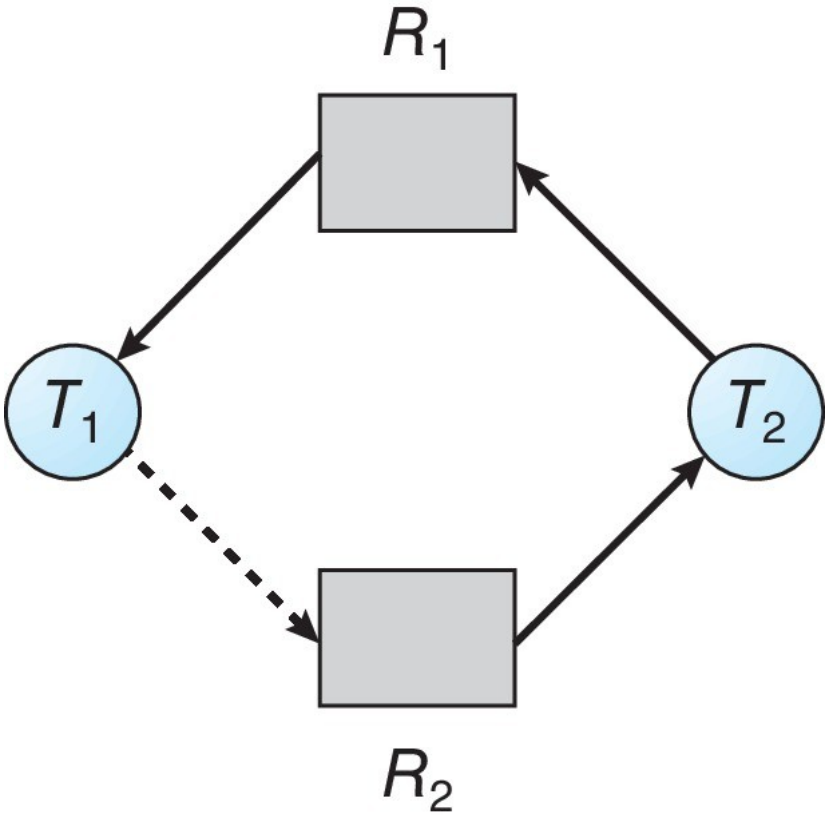
ما گراف Resource-Allocation رو از قبل داشتیم الان یه چیزی میخوایم به این Resource-Allocation اضافه بکنیم که اون همون اطلاعاتی است که میگیم برای اجتناب از بن بست نیازه:
یال خط چین: نشان دهنده این است که T2 ممکنه در اینده روی R2 ریکوست داشته باشه ینی الان ریکوست نداده و ریکوست فعلی T2 روی R1 است ولی بعدا ممکنه به R2 نیاز داشته باشه ولی هنوز اینو درخواست نکرده و میگیم T2 ادعا داره روی R2 و همینطور هم T1 ادعا داره روی R2
به چه صورت از این گراف میتونیم استفاده بکنیم که اجتناب بکنیم از بن بست؟
روش حلمون به طور کلی این است که سیستم رو توی حالت safe نگه داریم ینی هر بار میخوایم ریکوست رو پاسخی بدیم نگاه کنیم که ایا پاسخ دادن به اون ریکوست منجر میشه که سیستم از حالت safe به unsafe بره در اینصورت ریکوست رو جواب نمیدیم
توی همین شکل اگر T2 رسید به موقعی که واقعا روی R2 درخواست داشت این یال خط چین تبدیل میشه به یال ممتد که معنای ریکوست میده و اگر ریکوست رو ازاد کرد این میتونه تبدیل بشه به همون یال خط چین و همینطور اینجا که T2 به R1 نیاز داره اگر ریکوست رو بهش دادیم جهت یال برعکس میشه ینی میشه از R1 به T2 (ینی R1 اختصاص داده میشه به T2 که توی این شکل فعلا امکان پذیر نیست چون T1 در اختیار گرفته R1 رو و دیگه R1 ازاد نیست)
این الگوریتم چجوری باید کار بکنه؟
مثلا توی سیستم واقعا به شرایطی برسیم که T2 واقعا R2 رو درخواست بده ینی خط ابی بشه مثل عکس حالا سوال اینه که ایا سیستم بیاد R2 رو به T2 بده و الان R2 ازاده یا همچنان دست نگه داره و R2 رو نده به T2
چه چیزی رو باعث میشه که تصمیم بگیریم که R2 رو به T2 ندیم؟ اینکه اگر R2 رو به T2 بدیم سیستم بره توی حالت استیتی که اون استیت امن نیست
ادامش صفحه 26 ...

Resource-Allocation Graph Algorithm

- Suppose that process P_i requests a resource R_j
- The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocation graph

--

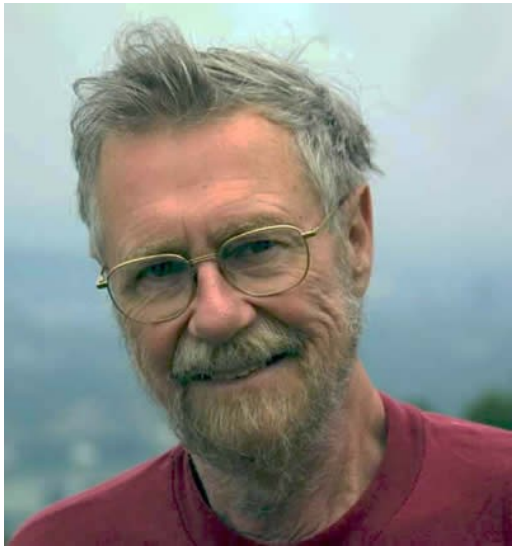
Unsafe State In Resource-Allocation Graph



ادامش...
فرض میکنیم R2 به T2 دادیم پس میشه الان خط قرمزه الان اتفاقی که افتاده اینه که R2 رو دادیم به T2 و T2 همچنان ریکوست روی R1 داره که هنوز بهش R1 داده نشده و R1 دست T1 است و T1 هم ممکنه در اینده روی R2 درخواست داشته باشه --> اگر برسیم به این حالت اتفاقی که میافته این میشه که گراف جدیدی درست میشه که دور داخلش است و این دور به ما نشون میده که بن بست اتفاق افتاده
پس ما وقتی که میخوایم تصمیم گیری بکنیم که یک ریکوست یک پروسس رو پاسخ بدیم یا نه باید فرض بکنیم که بهش پاسخ دادیم و بعد چک کنیم توی گراف Resource-Allocation که اگر اون ریکوست رو پاسخ دادیم ایا گراف تبدیل میشه به یک گرافی که دور داشته باشه و اگر دور داشته باشه به این معناست که در اینده به یک استیت unsafe می ره یا اینکه الان توی حالت unsafe است پس این ریکوست رو نباید جواب بدیم در غیر اینصورت می تونیم جواب بدیم
پس توی این مثالی که گفتیم نباید ریکوست T2 روی R2 رو جواب میدادیم چون حالت unsafe ایجاد کرد

Banker's Algorithm

- Multiple instances of resources
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time



-
الگوریتم Banker:
برای حالتی استفاده میشه که نمونه های بیشتر از یک برای هر ریسورس داشته باشیم که دیگه نتونیم از اون Resource-Allocation Graph استفاده بکنیم
اینجا ما باید بدونیم برای هر پروسسی برای هر ریسورسی حداکثر چندتا نمونه اش رو نیاز خواهد داشت که همون claim maximum use است
اگر یک پروسسی روی یک ریسورسی ریکوست داد این الگوریتم باید چک بکنه که اگر ریکوستش رو پاسخ بده سیستم می ره توی حالت امن یا حالت ناامن و اگر سیستم رفت توی حالت ناامن فعلا نباید ریکوستش رو جواب بدیم و این پروسس می ره توی حالت wait تا بعدا بتونیم ریکوستش رو جواب بدیم
در حالت عادی فکر می کردیم که اگر ریسورس ها ازاد باشن به هر حال اگر یک ریکوستی اومد و ریسورس هایی که میخواست اگر ازاد بود می تونیم بهش بدیم ولی اینجا میگیم نه نمیدیم اول چک میکنیم که اگر دادیم سیستم می ره توی حالت ناامن یا نه اگر رفت توی حالت ناامن ریکوست ها رو نباید جواب بدیم
فرض بعدی هم این است که اگر پروسسی کلیه ریسورس هایی که مورد نیازش بود رو گرفت به هر حال بعد از یک زمان محدودی اجراش تموم میشه و اون ریکوست ها رو میتونه ازاد بکنه پس فرضمون بر اینه که هر پروسسی به هر حال اجراش تموم میشه اگر ریسورس هایی که بهش نیاز داشت رو بهش بدیم

Data Structures for the Banker's Algorithm

Let n = number of processes, and m = number of resources types.

- **Available**: Vector of length m. If available [j] = k, there are k instances of resource type R_j available
- **Max**: $n \times m$ matrix. If Max [i,j] = k, then process P_i may request at most k instances of resource type R_j
- **Allocation**: $n \times m$ matrix. If Allocation[i,j] = k then P_i is currently allocated k instances of R_j
- **Need**: $n \times m$ matrix. If $Need$[i,j] = k, then P_i may need k more instances of R_j to complete its task

$$Need\ [i,j] = Max[i,j] - Allocation\ [i,j]$$

الگوریتم Banker:
فرض می کنیم که n تا پروسس داریم و m تا ریسورس تایپ داریم ینی نوع های مختلفی از ریسورس داریم
پس از هر ریسورسی ینی از هر کدوم از این m تا ممکنه یک تعداد مشخصی نمونه داشته باشیم و برای اینکه این نمونه ها رو مشخص بکنیم از یک وکتوری استفاده میکنیم به نام Available که بهمون میگه از هر ریسورسی چندتا نمونه موجود است
یک ماتریسی به نام Max که این ماتریس میگه هر کدوم از پروسس ها ماکزیمم نمونه ای که از هر ریسورس نیاز دارن چندتا است ینی اگر خونه i,j ام این ماتریس رو در نظر بگیریم ینی Max [i,j] برابر با k باشه ینی پروسس iام از ریسورس تایپ j به تعداد k تا نیاز خواهد داشت ینی ماکزیمم نیازش k تا است
ماتریس Allocation: که این هم n*m است که نشون مید که الان در حال حاضر هر کدوم از این پروسس ها چندتا نمونه از هر ریسورسی دستشون است پس Allocation[i,j] به این معنی است که پروسس iام چندتا ریسورس تایپ j دستش است
ماتریس Need: از ماتریس های قبلی به دست میاد - این ماتریس درایه i,j امش نشون میده که پروسس iام چندتا دیگه از نمونه نوع j نیاز داره پس فرمولش میشه: همونی که پایین نوشته خودش

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize:

   Work* = *Available

   ***Finish* [*i*] = *false* for *i* = 0, 1, …, *n*- 1**

2. Find an ***i*** such that both:

   (a) ***Finish* [*i*] = *false***

   (b) $\boldsymbol{Need_i \leq Work}$

   If no such ***i*** exists, go to step 4

3. $\boldsymbol{Work = Work + Allocation_i}$
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish* [*i*] == *true*** for all ***i***, then the system is in a safe state

الگوریتم Banker یا Safety:
می خوایم ببینیم که اگر ریکوست یک پروسس رو جواب دادیم توی حالت امن می مونیم یا نه
حالت مسئله در چه صورت امن است؟
ما باید چک بکنیم که توی استیتی که در حال حاضر می بینیم کدوم یک از پروسس ها رو می تونیم نیازهاش رو براورده کنیم ینی ماکزیمم نیازه که داره رو تا اجراش تموم بشه و ریسورس هاش رو برگردونه به سیستم و دوباره این کار رو تکرار میکنیم ینی ریسورس های این پروسس که برگشت به سیستم نگاه میکنیم که چه پروسس بعدی می تونه اجرا بشه و این کار رو تکرار میکنیم تا همه پروسس ها اجرا بشن--> این فرضمون است ینی داریم توی ذهن خودمون این کارو میکنیم نه اینکه واقعا این ریسورس ها رو به این پروسس ها بدیم داریم فرض میکنیم اگر دادیم سیستم توی چه حالتی می ره و این کارو تکرار میکنیم و اگر با تکرار این کار همه پروسس ها پاسخ داده شد و تمام شد می فهمیم اون استیت اولیه که ازش شروع کردیم استیت امنی است بنابراین اگر بخوایم فرموله بکنیم این الگوریتم رو: به ازای هر پروسسی مقدار Finish در نظر می گیریم که ینی یک وکتور برای پروسس ها در نظر میگیریم به نام Finish و اگر هر پروسسی کلا بتونه اجراش تموم بشه مقدار این Finish میشه true وگرنه false است
توی حالت اولیه این Finish برابر با False است
یک ماتریس هم در نظر میگیریم که مقدار اولیه اش برابره ینی یک وکتور Work مقدار اولیه اش برابر با ریسورس هایی است که Available است ینی وکتور Available
توی این حالت میایم i یا پروسسی رو پیدا میکنیم که ماتریس Needاش کمتر از Work مون است ینی با این چک میکنیم که ایا می تونیم با این ریسورس هایی که الان داریم پاسخش بدیم یا نه و اگر Needاش کمتر و مساوی بود می تونیم پاسخ بدیم پس پاسخ میدیم و فرضمون بر اینه که الان تموم شد پس Finish اش میشه true و وقتی که تموم شد ریسورس هاشو ازاد میکنه و چقدر ریسورس دستش بود؟ به اندازه Allocationi پس Work + Allocationi میکنه تا Work جدید به دست بیاد ینی بعد از اینکه این پروسس اجراش تموم شد مشخص میشه که چقدر الان ریسورس ازاد داریم و این کار رو باید تکرار بکنیم تا همه Finish ها true بشه
توی مرحله b اگر هیچ پروسسی وجود نداشت که Needاش کمتر و مساوی Work باشه میایم چک میکنیم که ایا همه پروسس ها الان Finish شون true شده یا نه اگر true شده ینی همه رو جواب دادیم و سیستم امن است وگرنه اگر حداقل یکیشون هم true نبود ینی سیستم رفته توی حالت ناامن

Example of Banker's Algorithm

- 5 processes P_0 through P_4;

  3 resource types:

  A (10 instances), B (5instances), and C (7 instances)

- Snapshot at time T_0:

	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	5 3 2
P_2	3 0 2	9 0 2	7 4 3
P_3	2 1 1	2 2 2	7 5 3
P_4	0 0 2	4 3 3	9 6 4

claim

need

A	B	C	
7	4	3	P_0
1	2	2	P_1
6	0	0	P_2
0	1	1	P_3
4	3	1	P_4

7 2 5

P_1 P_3 P_0 P_2 P_4

مثال:
5 تا پروسس داریم از P0 تا P4 و 3 تا تایپ ریسورس داریم به اسم A, B, C و از هرکدومشون به ترتیب 10 و 5 و 7 در اختیار داریم
الان استیت فعلی سیستم به این صورت است ینی T0 هست:
نکته: این Available رو می تونیم خودمون محاسبه بکنیم چجوری محاسبه بکنیم؟ الان توی این استیت سیستم اگر Allocation ها رو با هم جمع بزنیم می بینیم که از A الان 7 تا و از B هم 2 تا و از C هم 5 تا پس الان Available میشه 2 3 3 که الان ینی اینقدر ازاده
الان می خوایم ببینیم که سیستم توی حالت امن هست یا نه؟
اول نیازه که ماتریس Need رو بسازیم --> عکس
حالا نگا میکنیم از بین پروسس های p0, p1, p2, p3, p4 کدومشون رو می تونیم اجرا بکنیم با توجه به Available که در اختیار داریم
p0 رو نمی تونیم جواب بدیم چون Needاش بیشتر از Available است
p1 رو می تونیم جواب بدیم : پس اولین پروسسی که می تونیم اجرا کنیم می تونه p1 باشه و وقتی که p1 اجرا شد ریسورس هاش ازاد میشه که الان Available ما میشه 2 3 5
p2 نمی تونه اجرا بشه
p3 می تونه اجرا بشه حالا Available ما میشه 3 4 7
الان p0 می تونه اجرا بشه حالا Available میشه 3 5 7
الان p2 هم میتونه اجرا بشه حالا Available میشه 4 6 9
و در اخر p4 هم میتونه اجرا بشه
الان رسیدیم به حالتی که کل پروسس ها پاسخ داده شد پس می تونیم بگیم که سیستم توی حالت امن بوده

Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Example (Cont.)

- The content of the matrix ***Need*** is defined to be ***Max – Allocation***

	Need
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- The system is in a safe state since the sequence $< P_1, P_3, P_4, P_2, P_0 >$ satisfies safety criteria

نکته:
ممکنه دنباله های متفاوتی پیدا بکنیم که سیستم رو ببره به حالت امن مهم اینه که یکی از دنباله ها رو پیدا بکنیم

Resource-Request Algorithm for Process P_i

$Request_i$ = request vector for process P_i. If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$$Available = Available - Request_i;$$

$$Allocation_i = Allocation_i + Request_i;$$

$$Need_i = Need_i - Request_i;$$

- If safe $\Rightarrow$ the resources are allocated to P_i
- If unsafe $\Rightarrow$ P_i must wait, and the old resource-allocation state is restored

چه موقع سیستم توی حالت ناامن است؟
صفحه بعدی..
اگر ریکوستی روی Pi وجود داشت و ما الان می خوایم چک بکنیم که باید ریکوست Pi جواب بدیم یا نه باید ببینیم اون چیزی که الان ریکوست داده
اولا از Need اش کمتر باشه
بعد از Available مون هم کمتر باشه
در این حالت می تونیم بهش جواب بدیم و بعد که جواب دادیم ماتریس های Available و Allocation و Need رو اپدیت میکنیم
و چک میکنیم که ایا سیستم توی حالت امن است یا نه و اگر امن بود واقعا ریسورس ها رو allocated میکنیم به Pi
و اگر ناامن بود Pi باید الان wait کنه تا اینکه توی حالت امن باقی بمونیم

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

الان اینجا فرض کنیم اگر ریکوستی اومد فرض می کنیم این ریکوست رو جواب دادیم
با توجه به این که جواب دادیم ماتریس ها رو اپدیت بکنیم و بعد ببینیم استیت سیستم امن هست یا نه و اگر امن بود ینی این پاسخ دادنمون درست بوده پس میگیم این ریکوست رو جواب دادیم وگرنه جواب نمی دیم

مثال:
فرض میکنیم که همون مثال قبلی رو داریم حالا یک ریکوست جدید برامون اومده و p1 ریکوست داده به صورت 1,0,2
نگاه میکنیم می بینیم که این ریکوست کمتر از Available مون است پس می تونیم جوابشو بدیم
Availableمون همون مسئله قبلیه است ینی 2 3 3 بوده
حالا که جواب دادیم میایم این Available رو اپدیت میکنیم که میشه 0 3 2

مثل قبلی اینو حل میکنیم و تهش می بینیم که به یک دنباله ای رسیدیم که این دنباله می تونه به این ترتیب پاسخ داده بشه پس این که این ریکوست p1 رو جواب بدیم کار درستی بوده و سیستم ریکوست p1 رو جواب میده

Example: P_1 Request (1,0,2)

- Check that Request ≤ Available (that is, (1,0,2) ≤ (3,3,2) ⇒ true

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence $< P_1, P_3, P_4, P_0, P_2 >$ satisfies safety requirement
- Can request for (3,3,0) by P_4 be granted?
- Can request for (0,2,0) by P_0 be granted?

Deadlock Detection

- Allow system to enter deadlock state
- Detection algorithm
- Recovery scheme

-
رویکرد دوم: اجازه میده که بن بست اتفاق بیوفته و بعد بخوایم از شرایط بن بست recovery کنیم

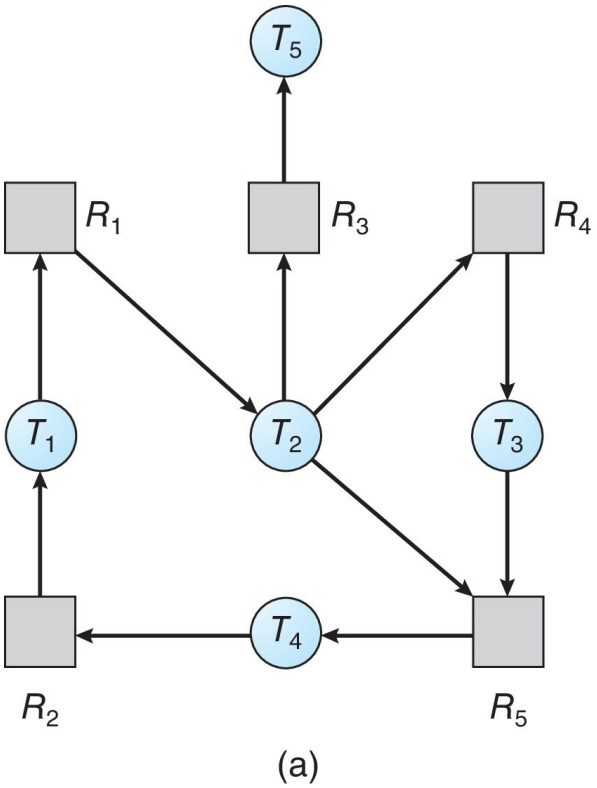
Single Instance of Each Resource Type

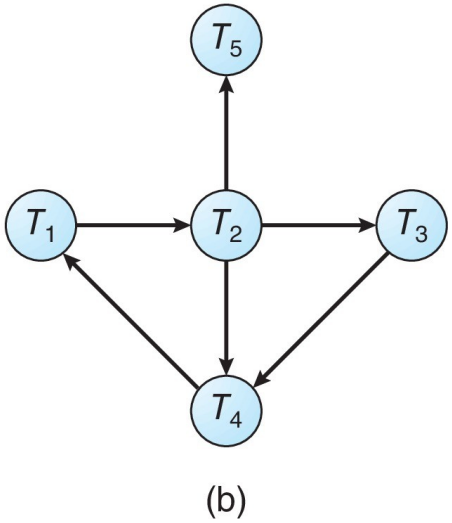
- Maintain **wait-for** graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph. If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

-
در این حالت مسئله اولمون این میشه که چجوری تشخیص بدیم که بن بست اتفاق افتاده توی سیستم؟
می تونیم از گراف Resource-Allocation استفاده بکنیم
صفحه بعدی...
برای همچین الگوریتمی برای یک گرافی با n راس --> برای پیدا کردن سیکل ها در همچین گرافی نیازبه الگوریتمی داریم از اردر n به توان دو

این حالت برای موقعی است که از هر ریسورسی یک نمونه بیشتر نداریم

Resource-Allocation Graph and Wait-for Graph





Resource-Allocation Graph

Corresponding wait-for graph

یک گراف جدیدی هم که از گراف Resource-Allocation درمیاد که تعداد نودهاش کمتره و فضای کمتری نگهداریش وجود داره گراف wait-for است که از همون گراف Resource-Allocation به دست میاد که نودهاش فقط تردهای پروسس ها هستن و ریسورس ها حذف شده
مثلا اگر T2 به ریسورسی نیاز داشته باشه که دست T5 است اونوقت T2 منتظر T5 است پس یک یال از T2 به T5 وصل میشه

اینجا مسئله این میشه که باز بتونیم دور رو تشخیص بدیم و اگر دوری وجود داشت بفهمیم که بن بست اتفاق افتاده

Several Instances of a Resource Type

- **Available**: A vector of length ***m*** indicates the number of available resources of each type
- **Allocation**: An ***n*** **x** ***m*** matrix defines the number of resources of each type currently allocated to each process
- **Request**: An ***n*** **x** ***m*** matrix indicates the current request of each process. If ***Request*** **[*i*][*j*] = *k***, then process P_i is requesting ***k*** more instances of resource type R_j.

-
برای تشخیص بن بست توی حالتی که از هر ریسورسی ممکنه تعداد بیشتر از یک نمونه داشته باشیم باید کاری شبیه الگوریتم Banker رو انجام بدیم اینجا برای تشخیص بن بست

Detection Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length ***m*** and ***n***, respectively Initialize:
   (a) ***Work* = *Available***
   (b) For ***i* = 1,2, …, *n***, if ***Allocation***$_i \neq$ **0**, then ***Finish*[i] = *false***; otherwise, ***Finish*[i] = *true***

2. Find an index ***i*** such that both:
   (a) ***Finish*[*i*] == *false***
   (b) ***Request***$_i \leq$ ***Work***

   If no such ***i*** exists, go to step 4

اینجا هم باید اول اون وکتور Available رو در نظر بگیریم و هر دفعه نگاه کنیم که ریکوست کدوم یکی از پروسس ها رو می تونیم جواب بدیم
پس توی حالت تشخیص بن بست فرض اینه که پروسس ها یکسری ریکوست هایی دارن و باید نگا کنیم که توی این شرایط که الان این ریکوست ها رو دارن و یکسری Allocation قبلی هم براشون وجود داره ایا بن بست وجود داره یا نه
که کافیه که توی چند مرحله توی یک حلقه هر دفعه چک کنیم که ایا پروسسی الان هست که ریکوستش رو جواب بدیم ینی وکتور ریکوست از اون وکتور Available ما کوچکتر باشه در صورتی که اینطور باشه فرضمون بر اینه که این کار پروسس انجام میشه و بعد Allocation اش رو میتونه برگردونه به Available های ما و یک فلگی هم true میکنیم برای این پروسس که مشخص بشه که کارش تموم شده ینی Finish رو میایم true میکنیم و دوباره این کارو تکرار میکنیم تا وقتی که مشخص شد که همه پروسس ها این فلگ Finish شون true شده
و اگر true نشده و به حالتی برسیم که یک پروسسی همچنان false و ما هم نمی تونیم دیگه جوابشو بدیم مشخصه که سیستم افتاده توی حالت بن بست

مشخصه اردر همچین الگوریتمی خواهد شد m*n^2 :
چون ما m تا تایپ ریسورس داریم و n تا پروسس داریم و هر بار باید برای این n تا پروسس بیایم m تا مقایسه انجام بدیم که وکتورش قابل جواب دهی هست یا نه و این کار چون باید تکرار بشه تا همه پروسس ها چک بشن نهایتا به اردر m*n^2 می رسیم

Detection Algorithm (Cont.)

3. ***Work* = *Work* + *Allocation*$_i$**
   Finish*[*i*] = *true
   go to step 2

4. If ***Finish[i]* == *false***, for some ***i***, $1 \leq i \leq n$, then the system is in deadlock state. Moreover, if ***Finish*[*i*] == *false***, then P_i is deadlocked

Algorithm requires an order of O(m x n^2) operations to detect whether the system is in deadlocked state

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

مثال:
در اخر به بن بست نمی خوریم

	Allocation	Request	Available
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	→ 2 0 2	0 1 0
P_2	3 0 3	0 0 0	3 1 3
P_3	2 1 1	1 0 0	5 1 3
P_4	0 0 2	0 0 2	7 2 4

P_0, P_2, P_1, P_3, P_4

Example of Detection Algorithm

- Five processes P_0 through P_4; three resource types
  A (7 instances), *B* (2 instances), and *C* (6 instances)

- Snapshot at time T_0:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in ***Finish[i] = true*** for all ***i***

Example (Cont.)

- P_2 requests an additional instance of type ***C***

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?

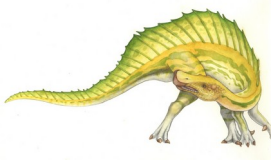
مثال:

Example (Cont.)

- P_2 requests an additional instance of type ***C***

	Request
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4



-
توی این مسئله فقط p0 می تونه اجرا بشه و بعد از اون p1, p2, p3, p4 نمی تونن اجرا بشن پس
ینی بن بستی اتفاق افتاده که شامل این 4 تا پروسس است --> p1, p2, p3, p4

Detection-Algorithm Usage

- When, and how often, to invoke depends on:
  - How often a deadlock is likely to occur?
  - How many processes will need to be rolled back?
    - one for each disjoint cycle
- If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

-
ما چقدر باید الگوریتم Detection را فراخوانی کنیم؟
این بستگی داره به این که بدونیم سیستممون چقدر احتمال بن بست توش وجود داره
چرا Detection میکنیم؟
برای اینکه بتونیم rolled back بکنیم از اون حالت و به استیت قبلی بریم که بن بست نداره

اگر فاصله های زمانی بلند باشه برای الگوریتم Detection اتفاقی که می افته اینه که شاید یکبار بن بست اتفاق بیوفته و بعد پروسس های جدید ریکوست های جدید می دن و این سیکل های بن بست هی بیشتر میشه یا بزرگتر میشه و بعد rolled back از توش سخت تر میشه بنابراین این که ما هر چند وقت یکباری توی زمان بیایم اون الگوریتم رو فراخوانی کنیم این بازه زمانی وابسته به این دوتا مورد میشه: 1- چقدر احتمال بن بست وجود داره 2- بن بستی که اتفاق می افته چندتا پروسس درگیرش هستن و نیازه که rolled back بشن --> هرچی تعداد این پروسس ها زیاد باشه توی بن بستی که اتفاق می افته معمولا rolled back کردنشون هم سخت تر میشه و بهتره که زودتر الگوریتم فراخوانی بکنیم که زودتر بتونیم تشخیصش بدیم

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  1. Priority of the process
  2. How long process has computed, and how much longer to completion
  3. Resources the process has used
  4. Resources process needs to complete
  5. How many processes will need to be terminated
  6. Is process interactive or batch?

-
بحث Recovery از بن بست می تونه چندتا رویکرد وجود داشته باشه:
1- Process Termination: ینی کلا اون پروسس هایی که درگیر بن بست شدن رو اصلا abort کنیم که این یکسری هزینه های بالایی برای ما داره که کلا پروسس رو ببندیم شاید اصلا دلمون نخواد اون پروسس بسته بشه یا مثلا اجرای دوباره اون پروسس امکان پذیر نباشه یا هزینه بر باشه
بعد اصلا کدوم یکی از پروسسس ها رو ببندیم که کلا بن بستمون شکسته بشه؟
1- اونی که اولویت پایین تری داره شاید
2- یا چقدر از یک پروسس مونده و اگر اجرای یک پروسسی که خیلی زمان ازش مونده رو اون رو مثلا ببندیم --> ینی پروسسی که زمان بیشتری از اجراش باقی مونده
3- بستگی به این داشته باشه که چه ریسورس هایی دست پروسس است یا اینکه چه ریسورس هایی نیاز داره که بتونه کارش رو تموم بکنه
6- ایا پروسسی که داریم می بندیم interactive بوده یا batch؟

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

-
اگر بخوایم rollback بکنیم ینی ریسورس هایی که دست پروسس ها است رو Preempte تا برگردیم به حالتی که دیگه بن بست نداریم باز اینجا بحث این میشه که:
1- ریسورس ها رو از کی بگیریم؟ ینی اون قربانی ما کی باشه؟ بازم باید نگاه بکنیم که گرفتن ریسورس ها از کدوم یکی از پروسس ها هزینه کمتری برای ما داره
2- اصلا امکان Rollback کردن اون پروسس مدنظر ما امکان پذیر هست یا نه؟
مثلا اگر از یک نقطه بخوایم Rollback بکنیم به یک نقطه قبلی اصلا باید ببینیم اون خط ها امکان برگشت دادنش وجود داره یا نه؟
3- یک قربانی وجود داشته باشه که همیشه دچار گرسنگی میشه ینی همیشه ما یکی از پروسس ها یا تعدادی از پروسس ها رو انتخاب بکنیم برای Rollback و این هیچ وقت اجراش تموم نشه

End of Chapter 8