R

بسم الله الرحمن الرحیم

سیستم عامل

جلسه نهم – بن‌بست

جلسه‌ی گذشته

مسائل هم‌روندی و مانیتور

بن‌بست راه حل اولیه‌ی شام فیلسوف‌ها

- هر ۵ فیلسوف چوب دستشان است!
- یک فیلسوف چوب برداشته بدون اینکه غذا بخورد ☹
- همه با دست راست شروع به غذا خوردن کردند.

- پس اجازه ندهیم حداقل یکی از این‌ها رخ بدهد.

Working Towards a Solution

وضعیت هر فیلسوف

```
int state[N]
semaphore mutex = 1
semaphore sem[i]
```

برای متغیرهای مشترک

برای منتظر غذا ماندن فیلسوف

Readers and Writers Problem

- Multiple readers and writers want to access a database (each one is a thread)
- Multiple readers can proceed concurrently
- Writers must synchronize with readers and other writers
  - *only one writer at a time !*
  - *when someone is writing, there must be no readers !*

Goals:

  - *Maximize concurrency*
  - *Prevent starvation*

دو سناریو

- First readers-writers problem
  - *no reader should wait for other readers to finish simply because a writer is waiting.*
- Second readers-writers problem
  - *once a writer is ready, that writer perform its write as soon as possible.*
  - *If a writer is waiting to access the object, no new readers may start reading.*

پیاده‌سازی ReadWriteLock

- از کتاب بخوانید... (بخش ۷٫۱٫۲)

پیاده‌سازی ReadWriteLock

- از کتاب بخوانید... (بخش ۷/۱/۲)

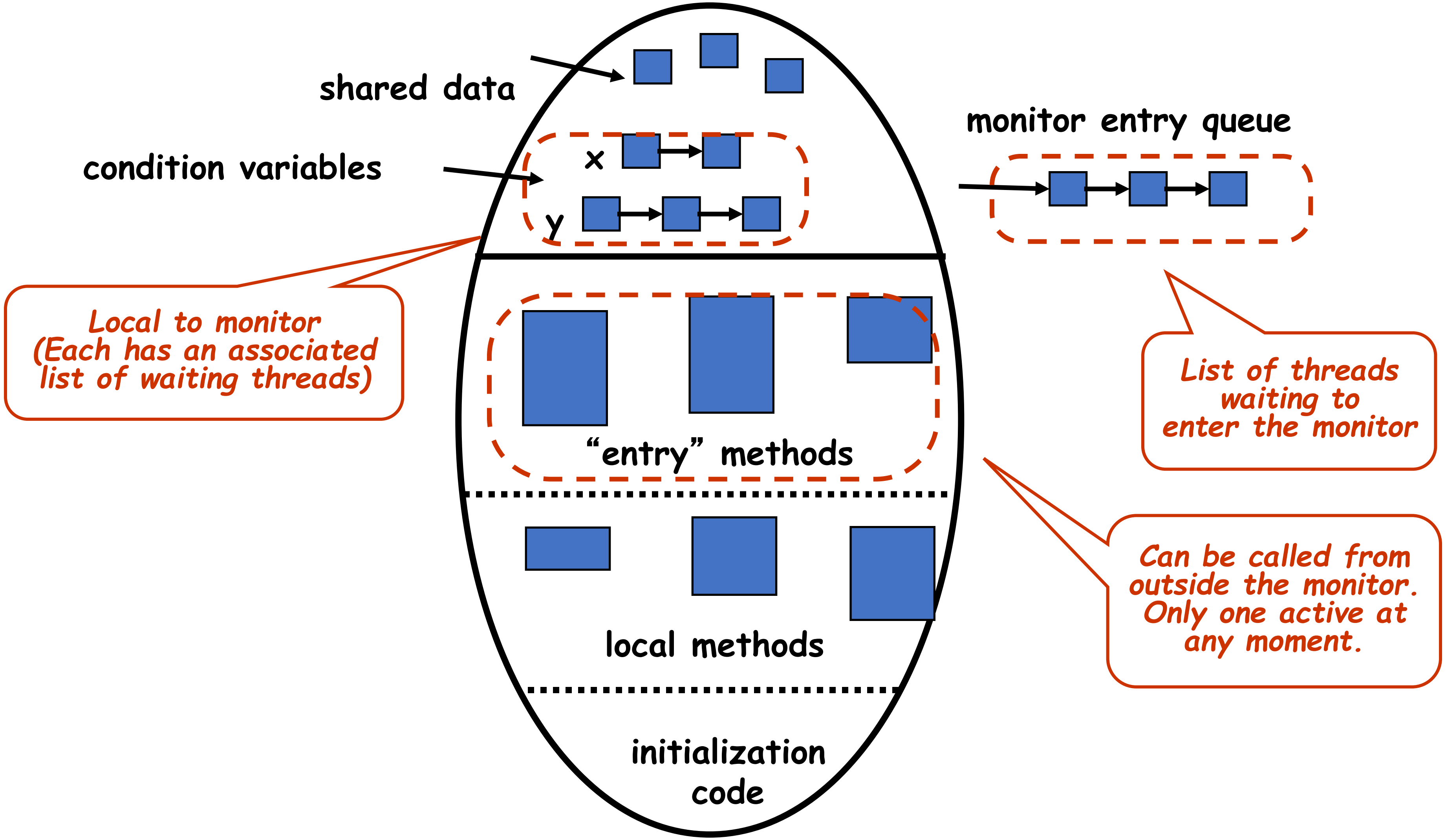
Monitors & Condition Variables

- We need two flavors of synchronization
- Mutual exclusion
  - *Only one at a time in the critical section*
  - *Handled by the monitor's mutex*
- Condition synchronization
  - *Wait until a certain condition holds*
  - *Signal waiting threads when the condition holds*

Monitors & Condition Variables

- Condition variables (cv) for use within monitors
  - *cv.wait(mon-mutex)*
    - Thread blocked (queued) until condition holds
    - Must not block while holding mutex!
    - Monitor mutex must be released!
    - Monitor mutex need not be specified by programmer if compiler is enforcing mutual exclusion
  - *cv.signal()*
    - Signals the condition and unblocks (dequeues) a thread

Monitor Structures



Monitor Example

```
monitor : BoundedBuffer
var buffer               : array[0..n-1] of char
    nextIn,nextOut       : 0..n-1 := 0
    fullCount            : 0..n   := 0
    notEmpty, notFull    : condition
```

```
  entry deposit(c:char)
  begin
    if (fullCount = n) then
        wait(notFull)
    end if

    buffer[nextIn] := c
    nextIn := nextIn+1 mod n
    fullCount := fullCount+1

    signal(notEmpty)
  end deposit
```

```
entry remove(var c: char)
begin
  if (fullCount = n) then
    wait(notEmpty)
  end if

  c := buffer[nextOut]
  nextOut := nextOut+1 mod n
  fullCount := fullCount-1

  signal(notFull)
end remove
```

```
end BoundedBuffer
```

Monitor Design Choices

- A signals a condition that unblocks B
  - *Does A block until B exits the monitor?*
  - *Does B block until A exits the monitor?*
  - *Does the condition that B was waiting for still hold when B runs?*
- A signals a condition that unblocks B & C
  - *Is B unblocked, but C remains blocked?*
  - *Is C unblocked, but B remains blocked?*
  - *Are both B & C unblocked, i.e. broadcast signal*
    - … if so, they must compete for the mutex!

Option 1: Hoare Semantics

- What happens when a Signal is performed?
  - *Signaling thread (A) is suspended*
  - *Signaled thread (B) wakes up and runs immediately*
- Result:
  - *B can assume the condition it was waiting for now holds*
  - *Hoare semantics give certain strong guarantees*
- When B leaves monitor, A can run
  - *A might resume execution immediately*
  - *... or maybe another thread (C) will slip in!*

Option 2: MESA Semantics

- What happens when a Signal is performed?
  - *The signaling thread (A) continues*
  - *The signaled thread (B) waits*
  - *When A leaves the monitor, then B resumes*
- Issue: What happens while B is waiting?
  - *Can the condition that caused A to generate the signal be changed before B runs?*
- In MESA semantics a signal is more like a hint
  - *Requires B to recheck the condition on which it waited to see if it can proceed or must wait some*

Example Use of Hoare Semantics

```
monitor BoundedBuffer
  var buffer: array[n] of char
     nextIn, nextOut: int = 0
     cntFull: int = 0
     notEmpty: Condition
     notFull: Condition

  entry deposit(c: char)
     if cntFull == N
       notFull.Wait()
     endIf
     buffer[nextIn] = c
     nextIn = (nextIn+1) mod N
     cntFull = cntFull + 1
     notEmpty.Signal()
   endEntry

  entry remove()
     ...

endMonitor
```

Hoare Semantics

Example Use of Mesa Semantics

```
monitor BoundedBuffer
  var buffer: array[n] of char
     nextIn, nextOut: int = 0
     cntFull: int = 0
     notEmpty: Condition
     notFull: Condition

  entry deposit(c: char)
     while cntFull == N
       notFull.Wait()
     endWhile
     buffer[nextIn] = c
     nextIn = (nextIn+1) mod N
     cntFull = cntFull + 1
     notEmpty.Signal()
   endEntry

  entry remove()
     ...

endMonitor
```

MESA Semantics

Message Passing

- Interprocess Communication
  - *Via shared memory*
  - *Across machine boundaries*
- Message passing can be used for synchronization or general communication
- Processes use send and receive primitives
  - *receive can block (like waiting on a Semaphore)*
  - *send unblocks a process blocked on receive (just as a signal unblocks a waiting process)*

Message Passing Example

- **Producer-consumer example:**
  - *After producing, the producer sends the data to consumer in a message*
  - *The system buffers messages (kept in order)*
  - *The producer can out-run the consumer*
- How does the producer avoid overflowing the buffer?
  - *The consumer sends empty messages to the producer*
  - *The producer blocks waiting for empty messages*
  - *The consumer starts by sending N empty messages*
    - N is based on the buffer size

Message Passing Example

```
const N = 100              -- Size of message buffer
var em: char
for i = 1 to N             -- Get things started by
  Send (producer, &em)     --    sending N empty messages
endFor
```

```
thread consumer
  var c, em: char
  while true
    Receive(producer, &c)    -- Wait for a char
    Send(producer, &em)      -- Send empty message back
    // Consume char...
  endWhile
end
```

Message Passing Example

```
thread producer
  var c, em: char
  while true
    // Produce char c...
    Receive(consumer, &em)    -- Wait for an empty msg
    Send(consumer, &c)        -- Send c to consumer
  endWhile
end
```

Buffering Design Choices

- **Option 1:** **Mailboxes**
  - *System maintains a buffer of sent, but not yet received, messages*
  - *Must specify the size of the mailbox ahead of time*
  - *Sender will be blocked if the buffer is full*
  - *Receiver will be blocked if the buffer is empty*

Buffering Design Choices

- <u>**Option 2:**</u> **No buffering**
  - *If Send happens first, the sending thread blocks*
  - *If Receive happens first, the receiving thread blocks*
  - *Sender and receiver must Rendezvous (ie. meet)*
  - *Both threads are ready for the transfer*
  - *The data is copied / transmitted*
  - *Both threads are then allowed to proceed*

جلسه‌ی جدید: بن‌بست

معرفی مدل تئوری بررسی، مثالی از بن‌بست، تعریف مسئله، معرفی روش‌های حمله برای حل مسئله (یا حذف صورت مسئله!)

مدل‌سازی مسئله

System Model

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

Example of Resources?

- Printers
- disk drives
- kernel data structures (scheduling queues …)
- locks/semaphores to protect critical sections

Resource Usage Model

- Sequence of events required to use a resource
  - *request the resource (eg. acquire mutex)*
  - *use the resource*
  - *release the resource (eg. release mutex)*
- Must wait if request is denied
  - *block*
  - *busy wait*
  - *fail with error code*

Preemptable Resources

- **Preemptable** resources
  - *Can be taken away with no ill effects*
- **Nonpreemptable** resources
  - *Will cause the holding process to fail if taken away*
  - *May corrupt the resource itself*
- Deadlocks occur when processes are granted exclusive access to non-preemptable resources and wait when the resource is not available

Definition of Deadlock

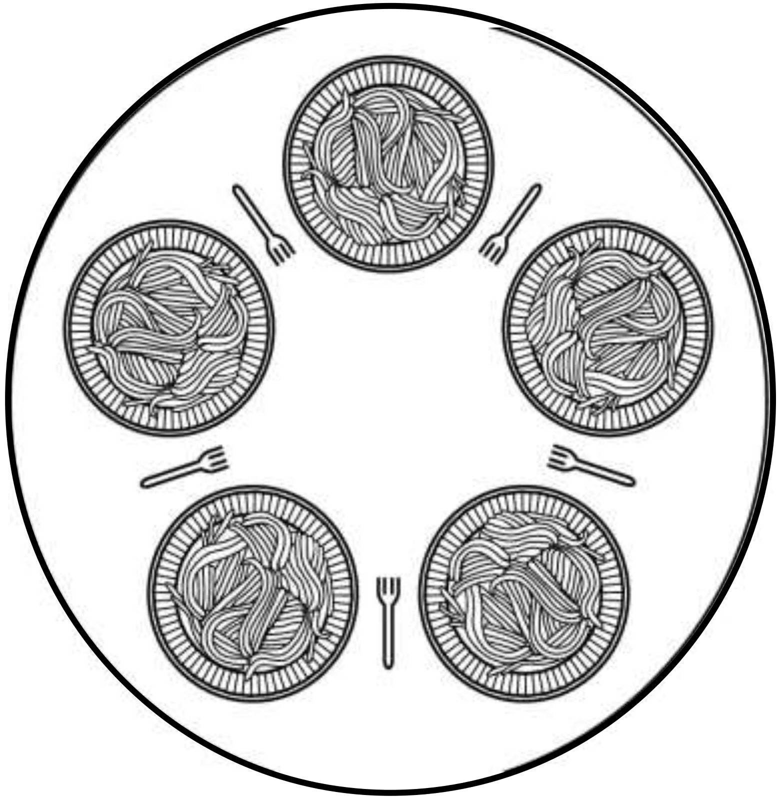
A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause

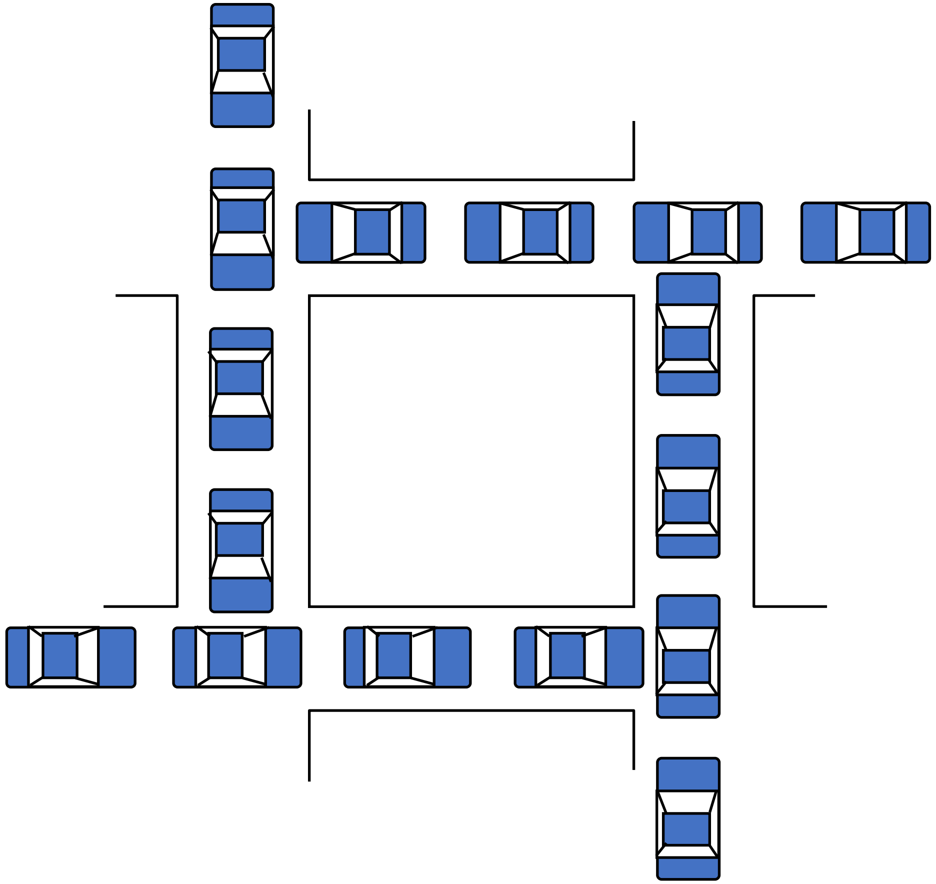
- Usually, the event is the release of a currently held resource
- None of the processes can …
  - *Be awakened*
  - *Run*
  - *Release its resources*

Deadlock Conditions

- A deadlock situation can occur *if and only if* the following conditions hold simultaneously
  - ***Mutual exclusion*** *condition – resource assigned to one process only*
  - ***Hold and wait*** *condition – processes can get more than one resource*
  - ***No preemption*** *condition*
  - ***Circular wait*** *condition – chain of two or more processes (must be waiting for resource from next one in chain)*

Examples of Deadlock



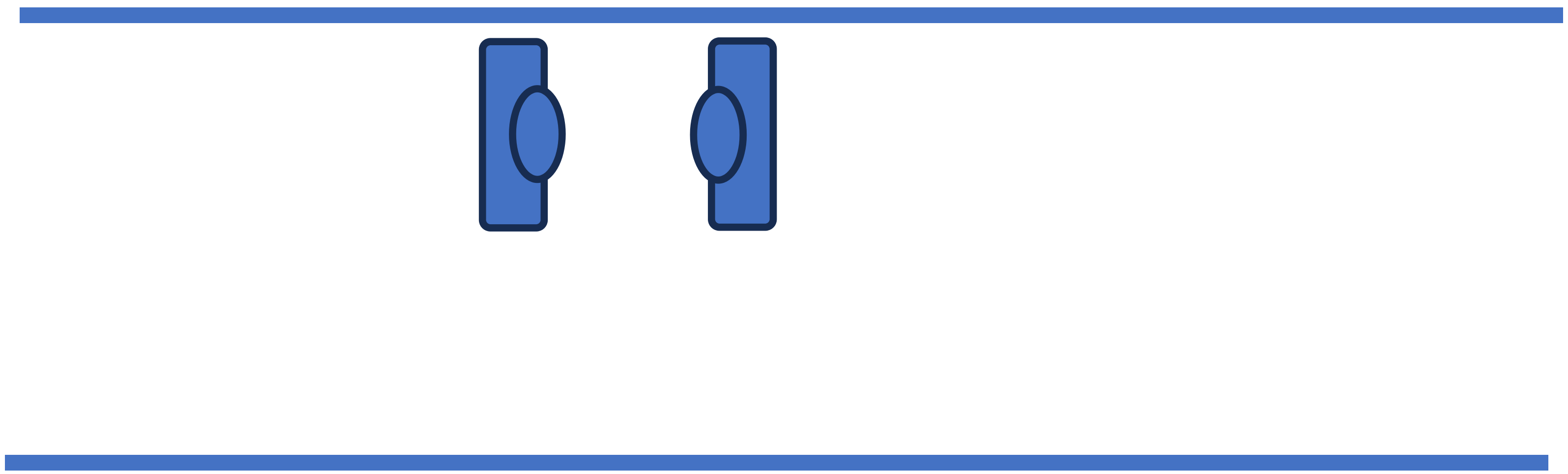


Livelock!

- A livelock is a situation where two or more processes are constantly changing their state in response to each other, but none of them can make any progress

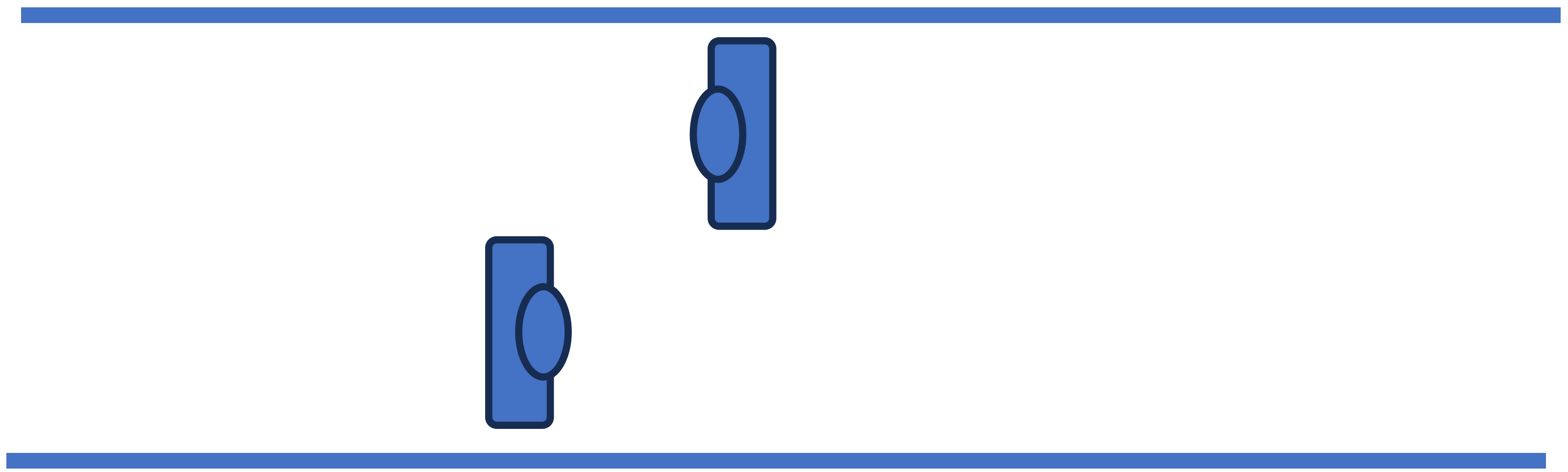
Livelock!

- A livelock is a situation where two or more processes are constantly changing their state in response to each other, but none of them can make any progress



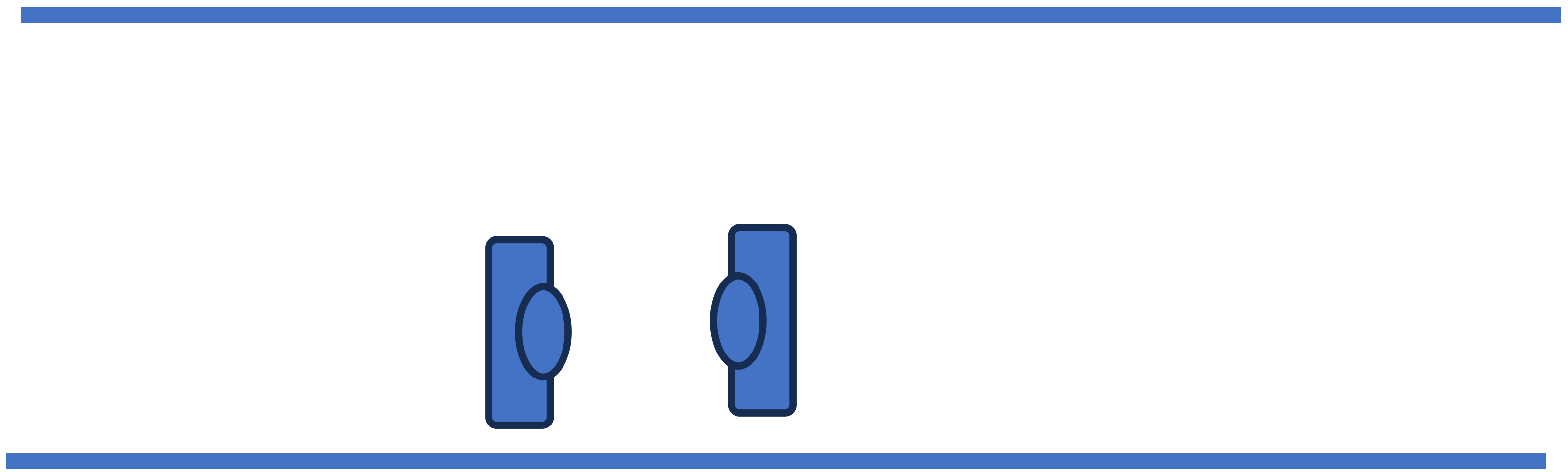
Livelock!

- A livelock is a situation where two or more processes are constantly changing their state in response to each other, but none of them can make any progress



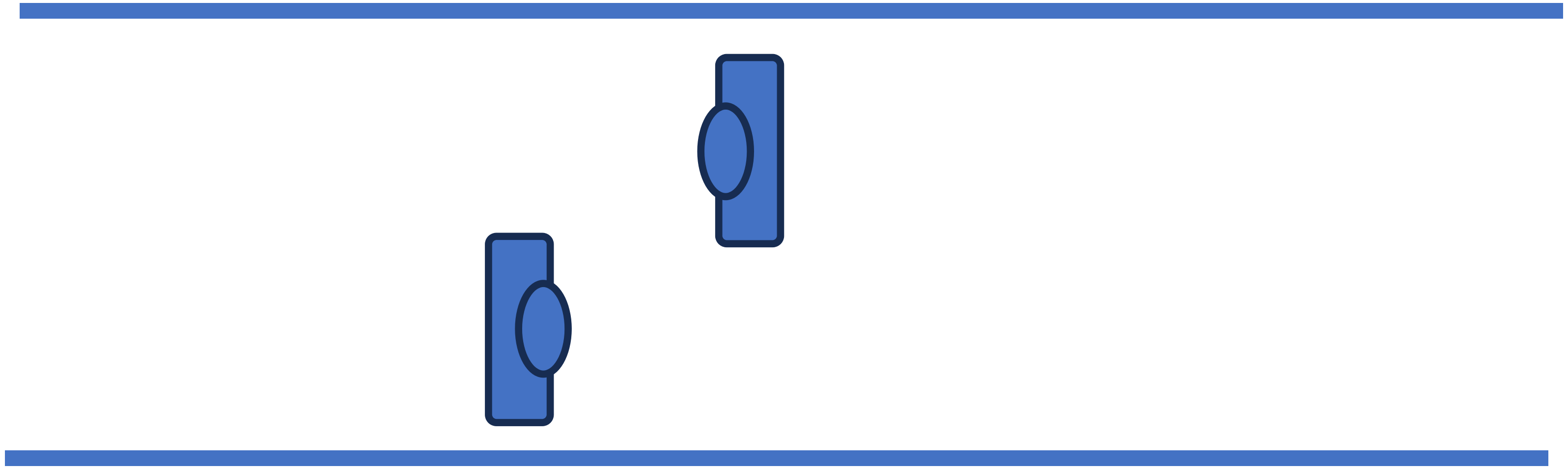
Livelock!

- A livelock is a situation where two or more processes are constantly changing their state in response to each other, but none of them can make any progress



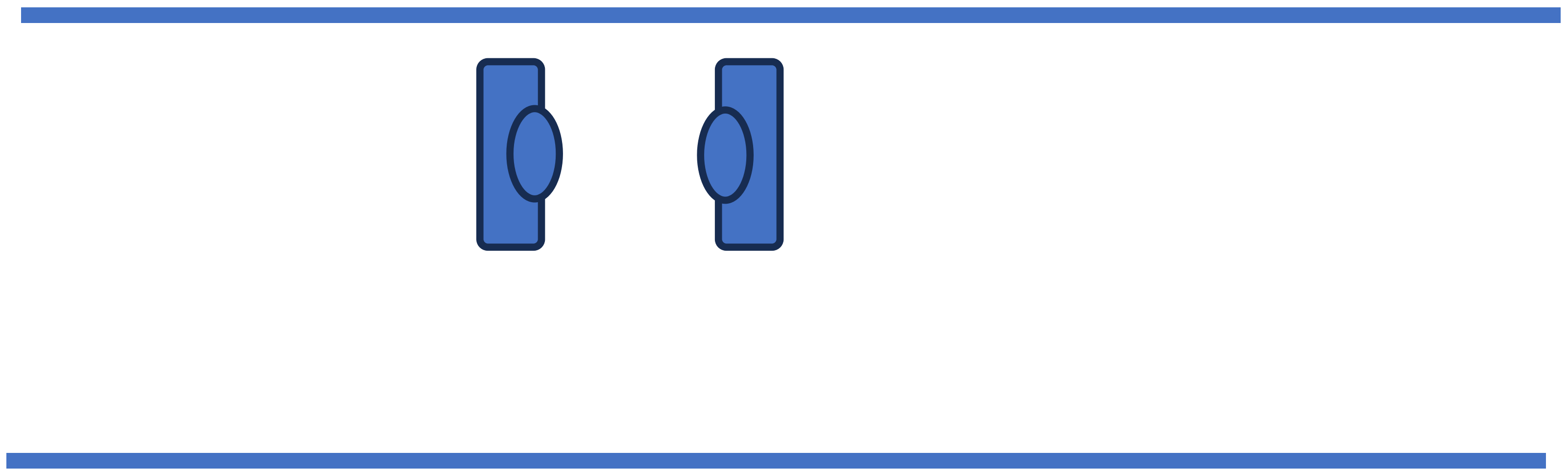
Livelock!

- A livelock is a situation where two or more processes are constantly changing their state in response to each other, but none of them can make any progress



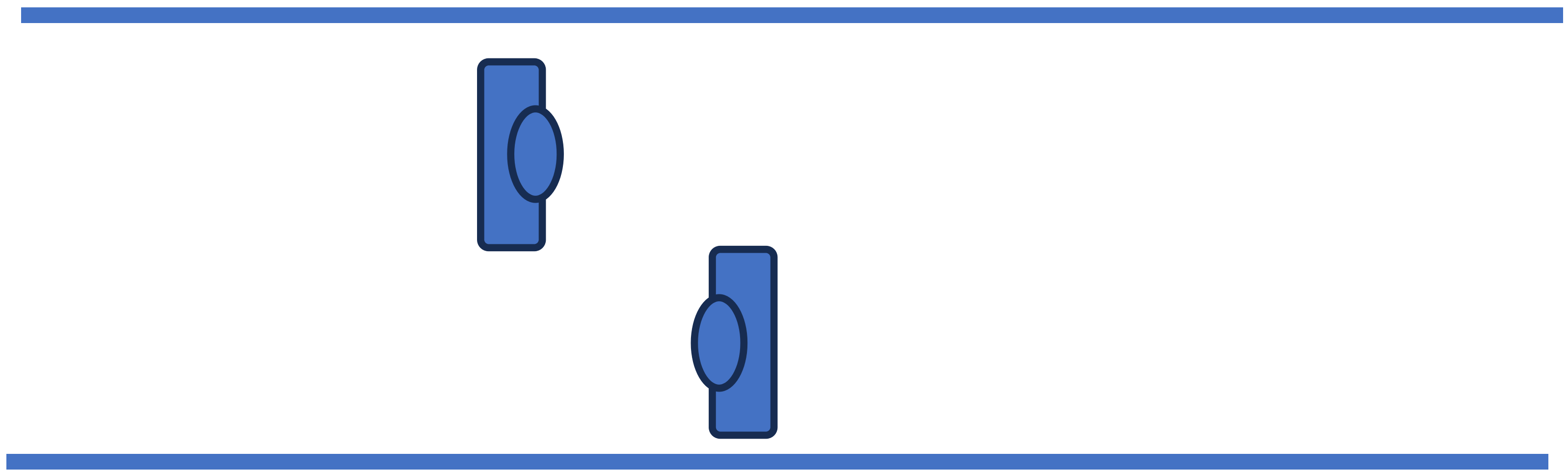
Livelock!

- A livelock is a situation where two or more processes are constantly changing their state in response to each other, but none of them can make any progress



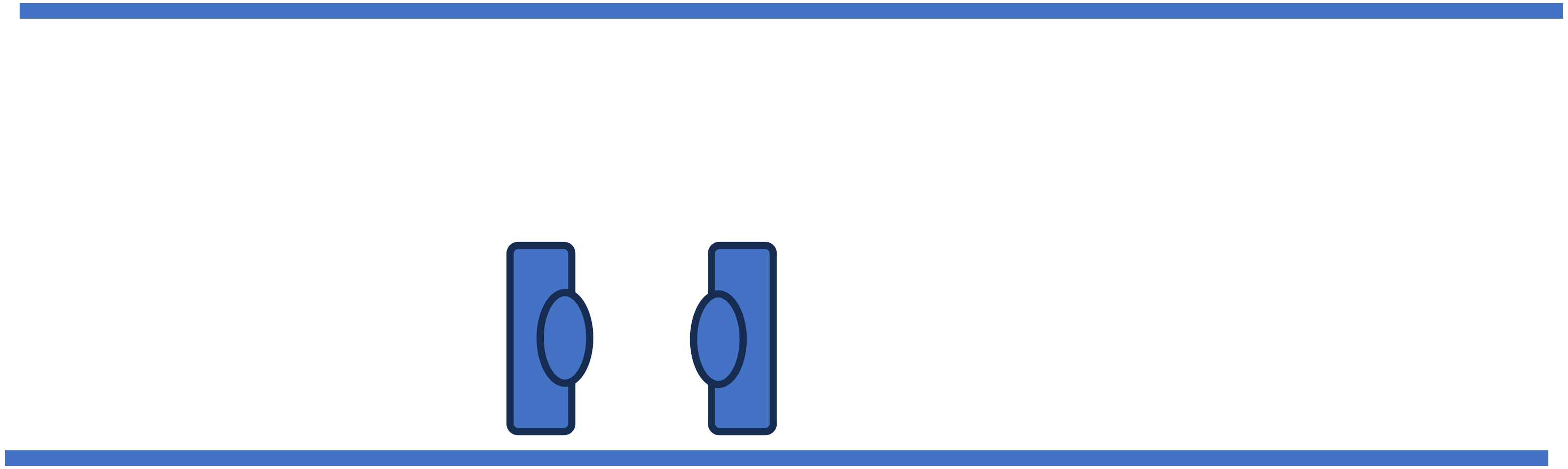
Livelock!

- A livelock is a situation where two or more processes are constantly changing their state in response to each other, but none of them can make any progress



Livelock!

- A livelock is a situation where two or more processes are constantly changing their state in response to each other, but none of them can make any progress



چند مثال از گرفتن منابع

Resource Acquisition Scenarios

Thread A:

```
acquire (resource_1)
use resource_1
release (resource_1)
```

Example:

```
var r1_mutex: Mutex
...
r1_mutex.Lock()
Use resource_1
r1_mutex.Unlock()
```

Resource Acquisition Scenarios

Thread A:

```
acquire (resource_1)
use resource_1
release (resource_1)
```

Another Example:

```
var r1_sem: Semaphore
r1_sem.Up()
...
r1_sem.Down()
Use resource_1
r1_sem.Up()
```

Resource Acquisition Scenarios

Thread A:

```
acquire (resource_1)
use resource_1
release (resource_1)
```

Thread B:

```
acquire (resource_2)
use resource_2
release (resource_2)
```

Resource Acquisition Scenarios

Thread A:

```
acquire (resource_1)
use resource_1
release (resource_1)
```

Thread B:

```
acquire (resource_2)
use resource_2
release (resource_2)
```

No deadlock can occur here!

Resource Acquisition Scenarios

Thread A:

```
acquire (resource_1)
acquire (resource_2)
use resources 1 & 2
release (resource_2)
release (resource_1)
```

Thread B:

```
acquire (resource_1)
acquire (resource_2)
use resources 1 & 2
release (resource_2)
release (resource_1)
```

Resource Acquisition Scenarios

__Thread A:__

```
acquire (resource_1)
acquire (resource_2)
use resources 1 & 2
release (resource_2)
release (resource_1)
```

__Thread B:__

```
acquire (resource_1)
acquire (resource_2)
use resources 1 & 2
release (resource_2)
release (resource_1)
```

No deadlock can occur here!

Resource Acquisition Scenarios

Thread A:

```
acquire (resource_1)
use resources 1
release (resource_1)
acquire (resource_2)
use resource 2
release (resource_2)
```

Thread B:

```
acquire (resource_2)
use resources 2
release (resource_2)
acquire (resource_1)
use resource 1
release (resource_1)
```

Resource Acquisition Scenarios

Thread A:

```
acquire (resource_1)
use resources 1
release (resource_1)
acquire (resource_2)
use resource 2
release (resource_2)
```

Thread B:

```
acquire (resource_2)
use resources 2
release (resource_2)
acquire (resource_1)
use resource 1
release (resource_1)
```

No deadlock can occur here!

Resource Acquisition Scenarios

Thread A:

```
acquire (resource_1)
acquire (resource_2)
use resources 1 & 2
release (resource_2)
release (resource_1)
```

Thread B:

```
acquire (resource_2)
acquire (resource_1)
use resources 1 & 2
release (resource_1)
release (resource_2)
```

Resource Acquisition Scenarios

Thread A:

```
acquire (resource_1)
acquire (resource_2)
use resources 1 & 2
release (resource_2)
release (resource_1)
```

Thread B:

```
acquire (resource_2)
acquire (resource_1)
use resources 1 & 2
release (resource_1)
release (resource_2)
```

Deadlock is possible!

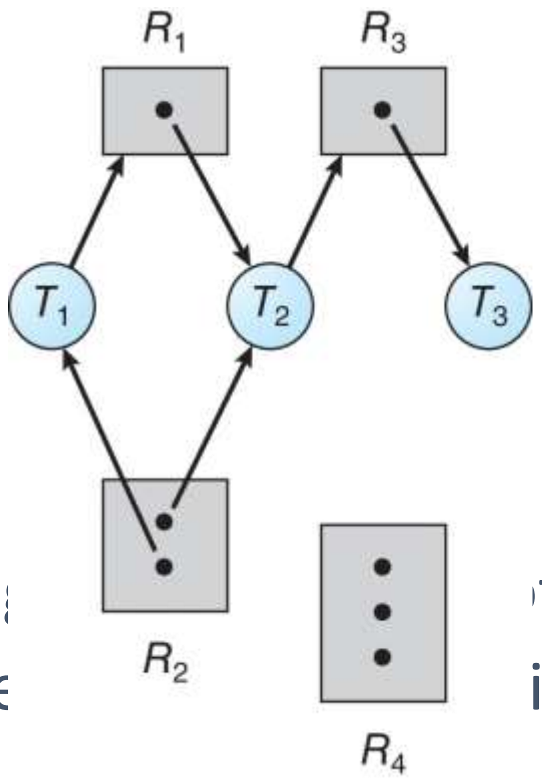
گراف تخصیص منابع

Resource-Allocation Graph

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

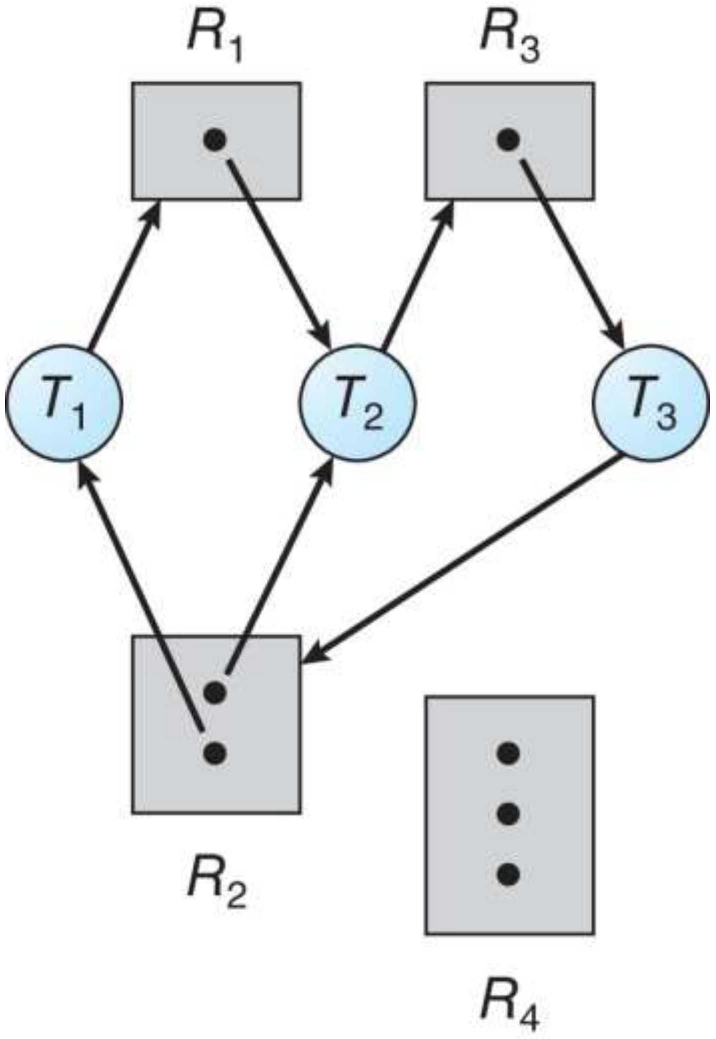
- V is partitioned into two types:
  - $T = \{T_1, T_2, \ldots, T_n\}$, *the set consisting of all the threads in the system.*
  - $R = \{R_1, R_2, \ldots, R_m\}$, *the set consisting of all resource types in the system*
- **request edge** – directed edge $T_i \rightarrow R_j$
- **assignment edge** – directed edge $R_j \rightarrow T_i$

Resource Allocation Graph Example

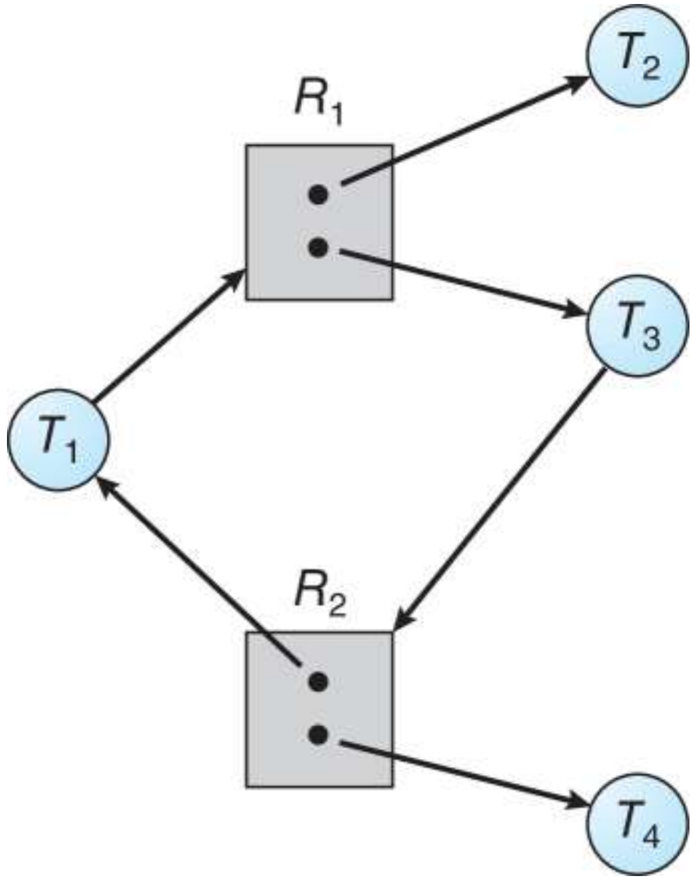


- One instance of R_1
- Two instances of R_2
- One instance of R_3
- Three instance of R_4
- T_1 holds one instance of R_2 and is waitin f R_1
- T_2 holds one instance of R_1, one instance ing for an instance of R_3
- T_3 is holds one instance of R_3

Resource Allocation Graph with a Deadlock



Graph with a Cycle But no Deadlock



خب، حالا چیکار کنیم؟

Dealing With Deadlock

- Ignore the problem
- Detect it and recover from it
- Dynamically avoid is via careful resource allocation
- Prevent it by attacking one of the four necessary conditions

Deadlock Prevention

Prevent it by attacking one of the four necessary conditions

- **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 thread requests a resource, it does not hold any other resources
  - *Require threads to request and be allocated all its resources before it begins execution or allow thread to request resources only when the thread has none allocated to it.*
  - *Low resource utilization; starvation possible*

Deadlock Prevention (Cont.)

- **No Preemption**:
  - *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*
  - *Preempted resources are added to the list of resources for which the thread is waiting*
  - *Thread 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 thread requests resources in an increasing order of enumeration*

جلسه‌ی بعد

- Deadlock Detection
- Deadlock Avoidance