Western

UNIVERSITY · CANADA

Chapter 8 - Deadlocks

Spring 2023

Overview

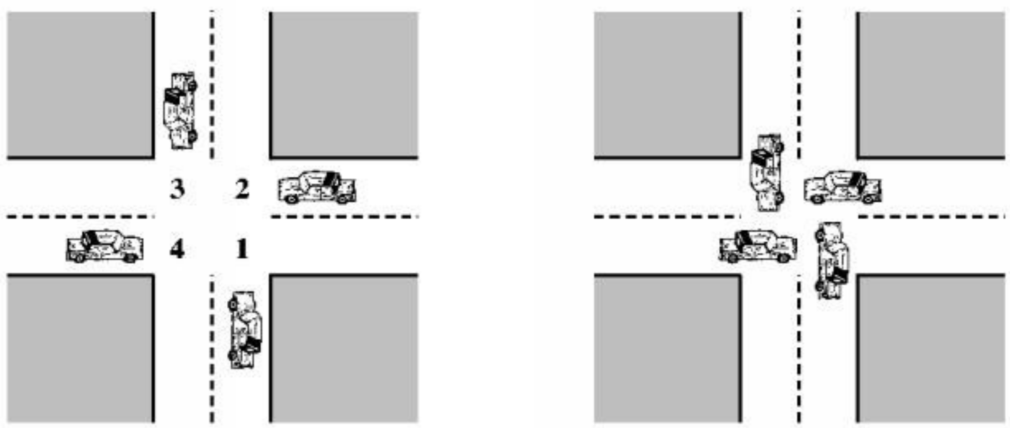
- System Model
- Deadlock Characterization
- Methods for Handling Deadlocks
- Deadlock Prevention
- Deadlock Avoidance
- Deadlock Detection
- Recovery from Deadlock

System Model

- Any system has a set of resources denoted R_1, R_2, … R_n
  - These could be CPU cycles, memory space, I/O devices, mutex locks, semaphores
- Each resource R_i has 1 or more instances
  - E.g. 4 CPUs, 2 Network interfaces, Semaphore value set to 5, etc.
- Each process or thread that requests a resources follows these steps:
  - 1) Request
  - 2) Use
  - 3) Release

System Model

- For example, consider a traffic jam



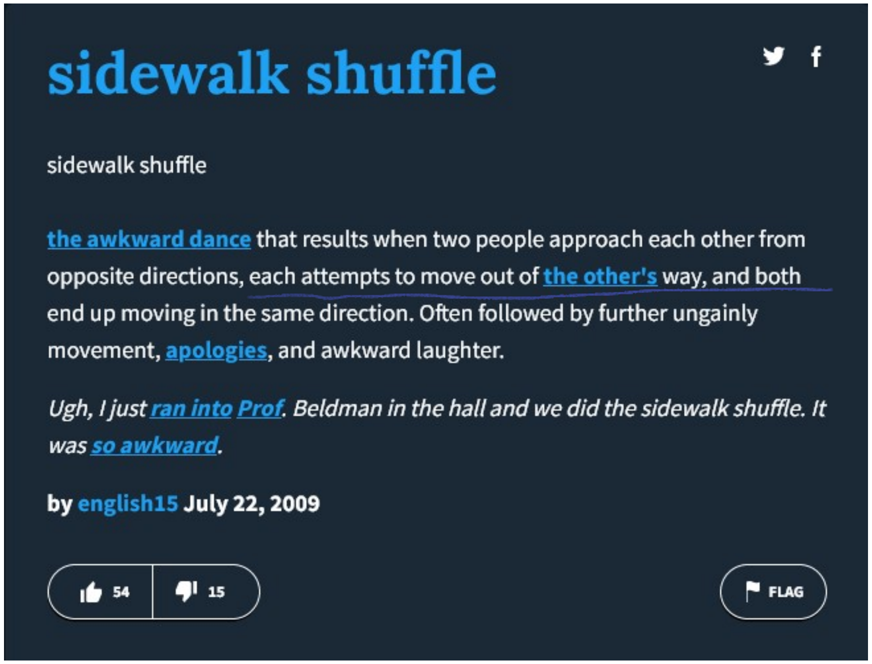
(a) Deadlock possible (b) Deadlock

System Model

- For example, return to the Dining Philosopher's problem
  - Suppose all five philosophers pick up their left chopstick. None of the philosophers can pick up their right chopstick. Some potential solutions:
    - Allow only 4 philosophers or less at the table
    - Allow a philosopher to pick up both chopsticks if and only if both are available
    - Odd numbered philosophers pick up their left chopstick, then the right. Even numbered philosophers pick up the right chopstick, then the left.

System Model

- For example, consider the "Sidewalk Shuffle". This is a special case known as **livelock**. There is no blocking but there is no progress either.



System Model

- For example, consider two threads with two semaphores
- T1:
  - wait(s1)
    wait(s2)
- T2:
  - wait(s2)
    wait(s1)

when T1 wait for s1, T2 would wait for s2,
so it is a deadlock.

T1: wait(s1)
=> wait for s1 to finish
=> turn to T2
T2: wait(s2)
=>

Deadlock Characterization

同时

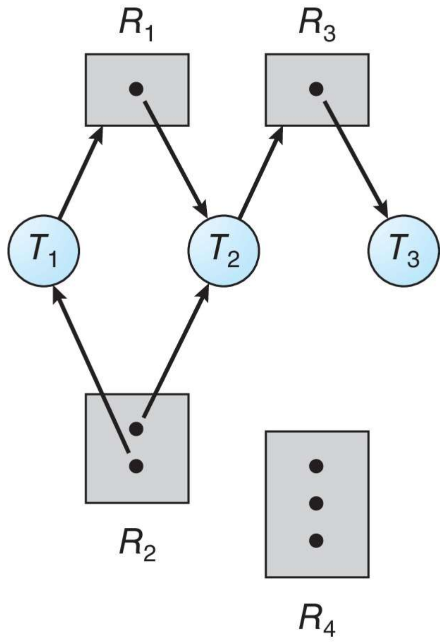
- A deadlock occurs under **all** the following four conditions **simultaneously**
  - **Mutual Exclusion** – only one thread at a time can use a resource
  - **Hold and Wait** – a thread holding at least one resource is waiting to acquire additional resources held by other threads
  - **No preemption** – a resource can be released only voluntarily by the thread holding it, after that thread has completed its task
  - **Circular wait** – Thread T_0 waits on a resource held by T_1, T_1 waits on a resource held by T_2, …, T_n waits on a resource held by T_0

Deadlock Characterization

- Identifying circular wait with resource allocation graphs
  - Consider two types of vertices
    - Threads – denoted $T = \{T_1, T_2, \ldots, T_n\}$
    - Resources – denoted $R = \{R_1, R_2, \ldots, R_m\}$
  - Two types of edges
    - Request edge – A directed edge $T_i \rightarrow R_j$ ("wants to hold")
    - Assignment edge – A directed edge $R_j \rightarrow T_i$ ("is holding")

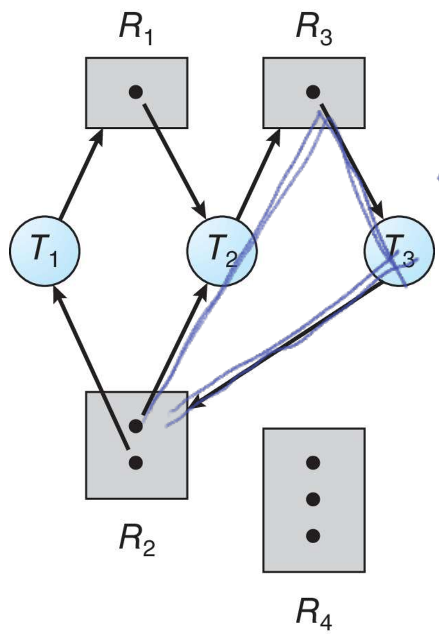
Deadlock Characterization

- Identifying circular wait with resource allocation graphs
  - 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 waiting for an instance of R_1
  - T_2 holds one instance of R_1, one instance of R_2, and is waiting for an instance of R_3
  - T_3 holds one instance of R_3



Deadlock Characterization

- Identifying circular wait with resource allocation graphs



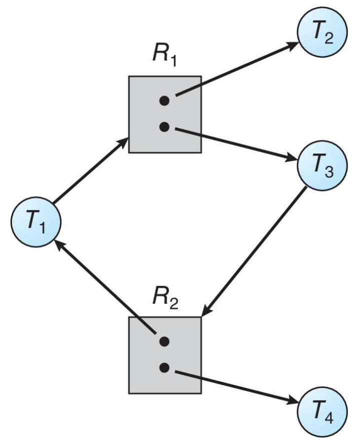
← it makes a circle, so it would be a deadlock.

Deadlock Characterization

- Identifying circular wait with resource allocation graphs
  - If a graph does not contain cycles, then we do not have deadlock
  - If a graph does contain cycles, we **may** have deadlock

not 100% having deadlock.

we have to check the case.



Handling Deadlocks

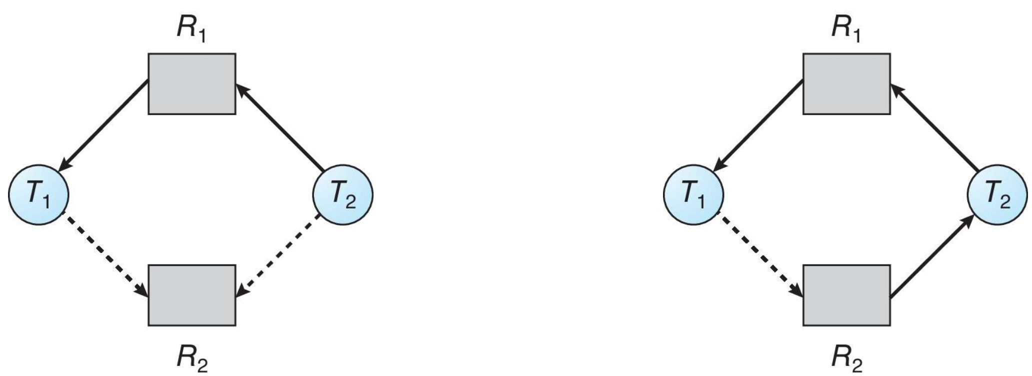
- Four options
  - **Ignore the deadlock** – Force application developers to ensure deadlock does not occur. For reasons of efficiency, this is the most common approach by most operating systems.
  - **Deadlock Prevention** – Ensure one of the four characteristics of deadlock does not occur. This typically underutilizes system resources.
  - **Deadlock Avoidance** – Declaring all resources that may be required before starting. Force threads to wait if there is a potential for overlap. Very inefficient.
  - **Deadlock Detection** – Periodically check to see if there is a deadlock and recover appropriately. Common in relational database management systems.

Deadlock Prevention

- Ensure one of the four characteristics of deadlock does not occur
  - 本質上
  - Mutual Exclusion – Some resources are just intrinsically unshareable. Preventing mutual exclusion is just not feasible
  - 习以为 prevent the circular wait.
  - Hold and Wait – Force a thread to hold all its resources before it can start. Most resources will be held for no reason. Popular resources will create starvation.
  - No preemption – If a resource cannot be secured, voluntarily release all existing resources and restart. Only feasible if state can be saved such as CPU registers and database transactions
  - Circular wait – Enforce an increasing order on all resources for all threads. It's up to the application developer to obey the order. E.g. R_1 then R_2. Never R_2 then R_1

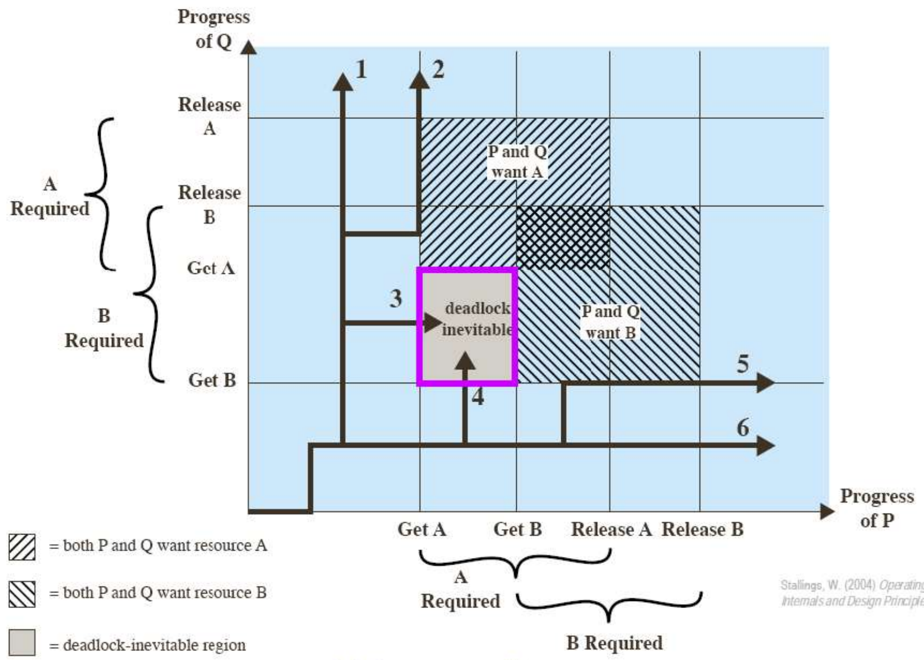
Deadlock Avoidance

- Resource Allocation Graphs – Do not allow any thread to enter the graph if it will create a cycle
  - E.g. T_2 must wait until T_1 finishes because introducing it could create a cycle



Deadlock Avoidance

- Deadlock Trajectory Diagram – Consider two Processes that require access to two resources

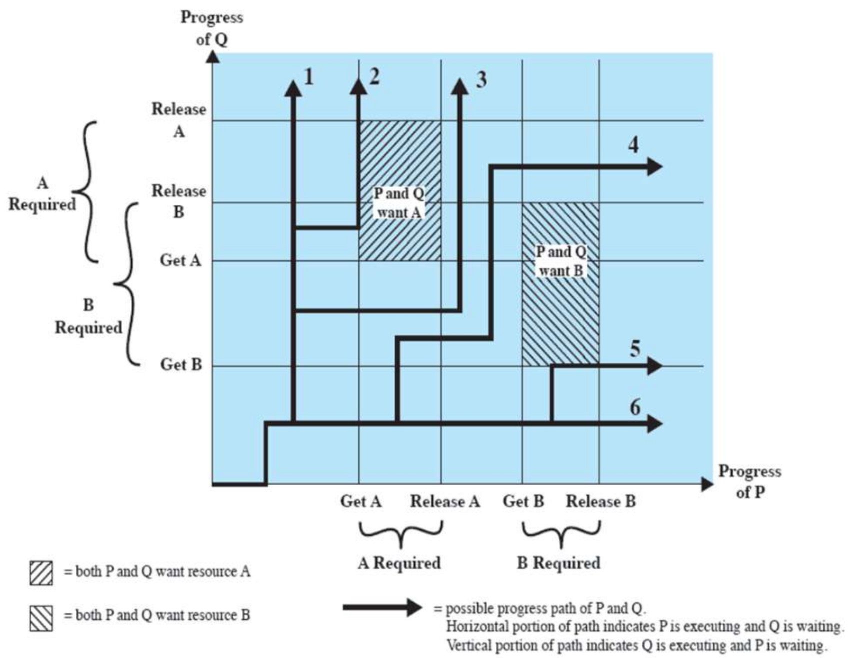


Joint progress diagram

Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

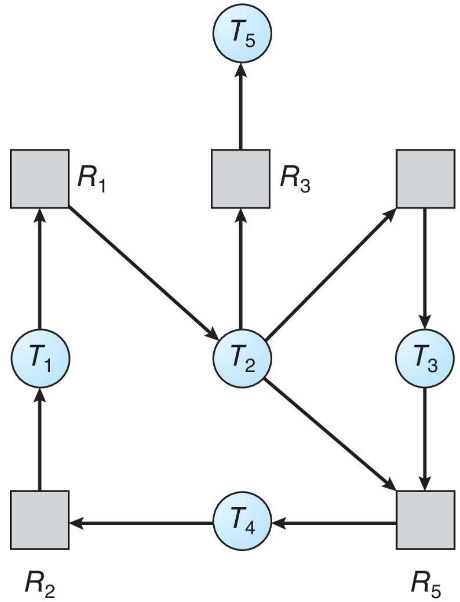
Deadlock Avoidance

- Deadlock Trajectory Diagram – Consider two Processes that require access to two resources

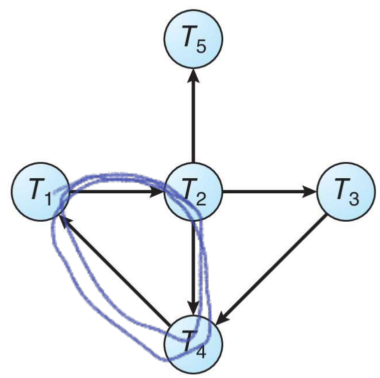


Deadlock Detection

- Collapse a resource allocation graph into a "wait-for" graph. Periodically invoke an algorithm to search for a cycle in the graph



(a)



(b)

Deadlock Detection

- How often to run the algorithm depends on a few factors
  - How often is a deadlock likely to occur?
  - When it does occur, how many threads will need to be aborted or rolled back?
- Running the algorithm uses precious CPU cycles, so we don't want to run it too often
- However, if the algorithm is not invoked often enough, deadlock will occur too frequently. The process to abort or roll back threads may take some time

Recovery from Deadlock

- Abort or rollback all deadlocked threads?
- Abort or rollback one thread at a time until deadlock stops?
  - Should aim to choose the **minimum cost**
    - Priority of the thread(s)?
    - Computation time completed? Computation time remaining?
    - Number of resources held? Number of remaining resources needed?
    - Fewest number of thread(s) to clear the deadlock
  - Starvation could occur if the same thread is always the victim

Western

UNIVERSITY · CANADA