Checkpoints

- Logs keep growing. After every failure, we'd have to go back and replay the log. This can be time consuming.
- Checkpoint frequently
  - Output all log records currently in volatile storage onto stable storage
  - Output all modified data residing in volatile storage to the stable storage
  - Output a log record <checkpoint> into stable storage
- On failure, search backwards till we hit the first checkpoint. The first transaction start from the checkpoint (going back) is the start of replay

Serializability

- Transactions can be concurrent. Such concurrency may cause problems depending on the interleaving of the transactions. We introduce stricter notions of this phenomenon in order to predict system behavior
- Schedule is an execution sequence
- Serial schedule: Schedule where two concurrent transactions follow one after the other
  - For two transactions T1, T2: serial schedule is T1 then T2 or T2 then T1. For n transactions, we have n! choices, all of which is valid
  - Serial schedule cannot fully utilize the system resources and so we want to relax the schedule: non-serial schedule

Conflict

- We define a schedule to be in conflict if they both operate on the same data item and one of the operations is a write
- If there is no conflict, the schedule can be swapped.
- If after non-conflicting swaps we reach a serial schedule, then that schedule is called conflict serializable

Read(A)	
Write(A)	
Read(B)	
Write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Serial schedule

Read(A)	
Write(A)	
	read(A)
	write(A)
Read(B)	
Write(B)	
	read(B)
	write(B)

Conflict serializable schedule

Locking protocol to enforce order

- Shared: Transaction can read but not write
- Exclusive: Transaction can read and write
- Two phase protocol to ensure serializability:
  - Growing phase - transaction can obtain but not release locks
  - Shrinking phase - transaction can release lock but not acquire new ones
  - Ensures conflict serializability not is not free from deadlocks

Timestamp-based Protocols

- Timestamp transactions: Can be real wall clock time or logical clock
- The timestamp determines the serializability order
- For each data item (Q), associate two timestamps
  - W-timestamp denotes largest timestamp of any transaction that successfully executed write(Q).
  - R-timestamp for read(Q)
- Suppose Ti issues read(Q):
  - If TS(Ti) < W-timestamp(Q), rollback Ti
  - If TS(Ti) >= W-timestamp(Q), execute Ti, R-timestamp = maximum (R-timestamp(Q) and TS(Ti))
- Similarly for Ti issuing write(Q):

Chapter 7: Deadlocks

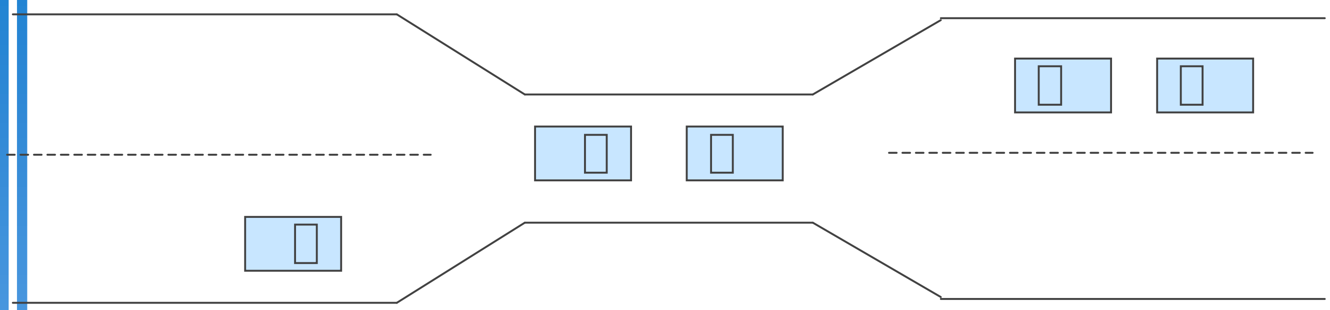
- To develop a description of deadlocks, which prevent sets of concurrent processes from completing their tasks
- To present a number of different methods for preventing or avoiding deadlocks in a computer system.

The Deadlock Problem

- A set of blocked processes each holding a resource and waiting to acquire a resource held by another process in the set.
- Example
  - System has 2 tape drives.
  - P_1 and P_2 each hold one tape drive and each needs another one.
- Example
  - semaphores *A* and *B*, initialized to 1

P_0	P_1
wait (A);	*wait(B)*
wait (B);	*wait(A)*

Bridge Crossing Example



- Traffic only in one direction.
- Each section of a bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

System Model

- Resource types R_1, R_2, . . ., 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

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 $\{P_0, P_1, \ldots, P_0\}$ 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

  $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

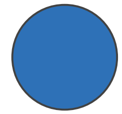
Resource-Allocation Graph

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

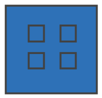
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system.
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system.
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed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

Resource-Allocation Graph (Cont.)

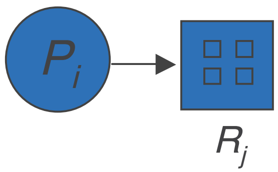
- Process



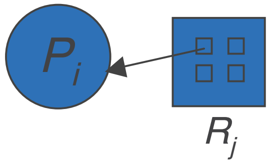
- Resource Type with 4 instances



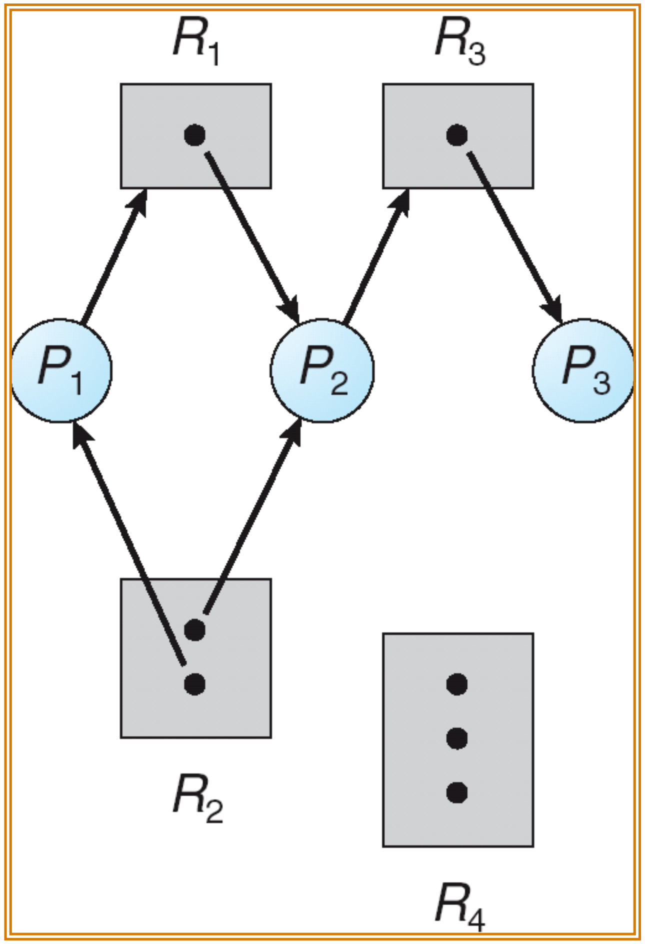
- P_i requests instance of R_j



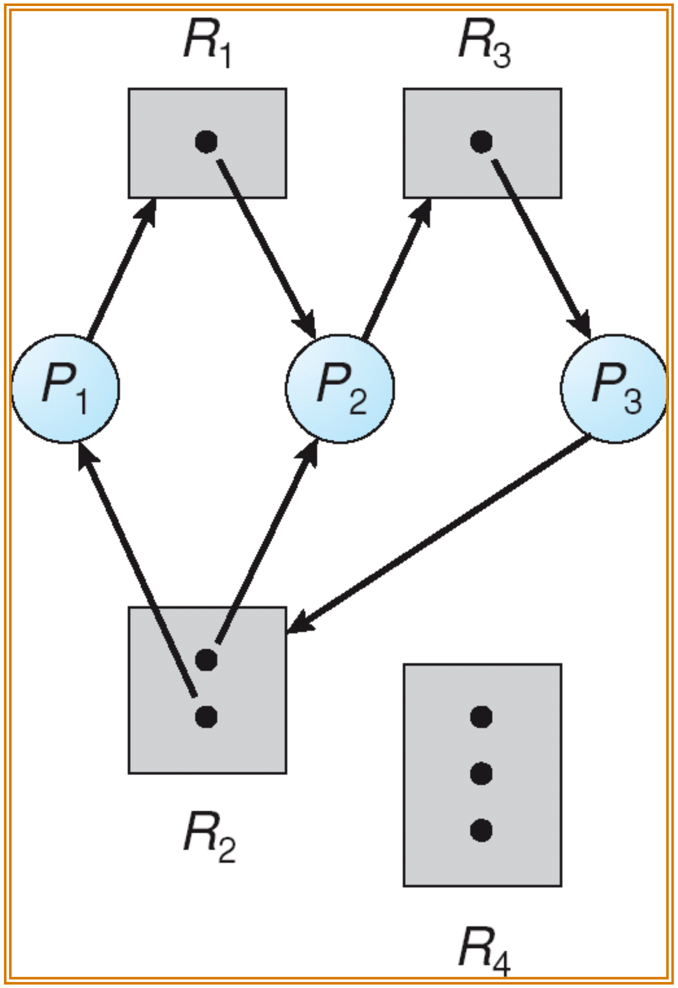
- P_i is holding an instance of R_j



Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Resource Allocation Graph With A Cycle But No Deadlock