Deadlocks

Overview

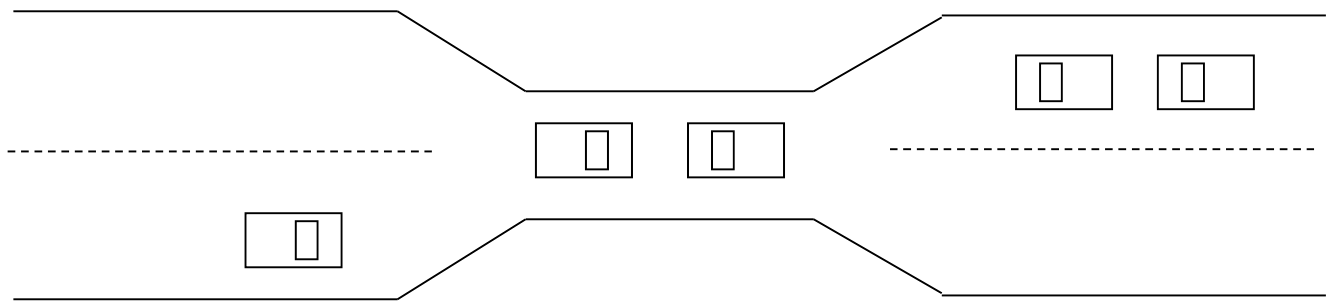
- Resources.
- Why do deadlocks occur?
- Dealing with deadlocks.
  - Ignoring them: ostrich algorithm.
  - Detecting.

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 disk drives
  - P_1 and P_2 each hold one disk 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)

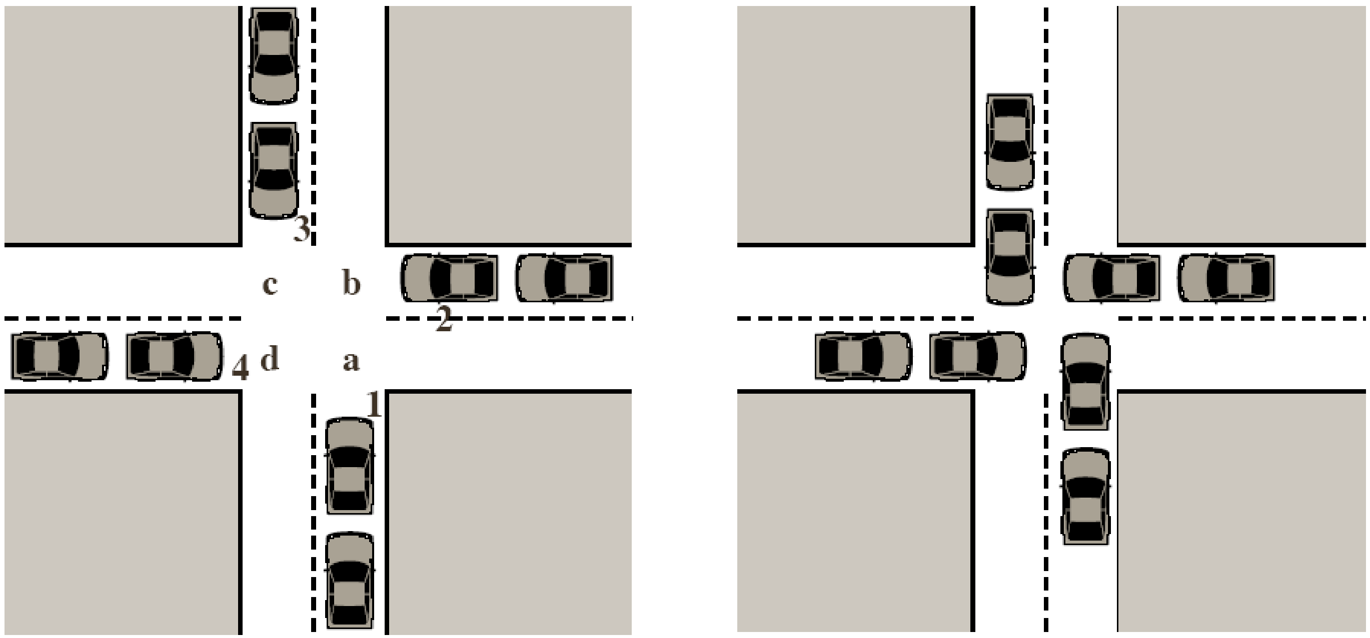
Bridge Crossing Example



- 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
- **Note** : Most OSes do not prevent or deal with deadlocks

Deadlock Principles

- A deadlock is a permanent blocking of a set of threads
  - a deadlock can happen while threads/processes are competing for system resources or communicating with each other



(a) Deadlock possible

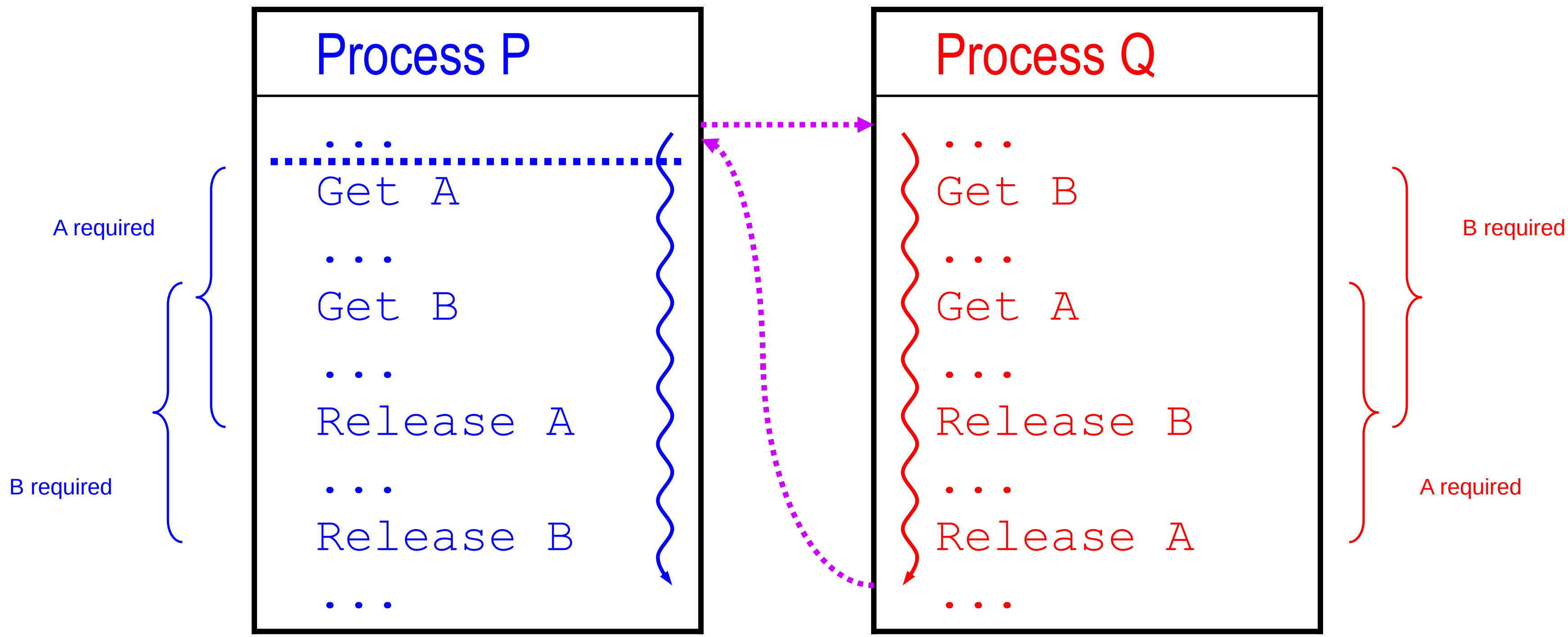
(b) Deadlock

Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

Illustration of a deadlock

Deadlock Principles

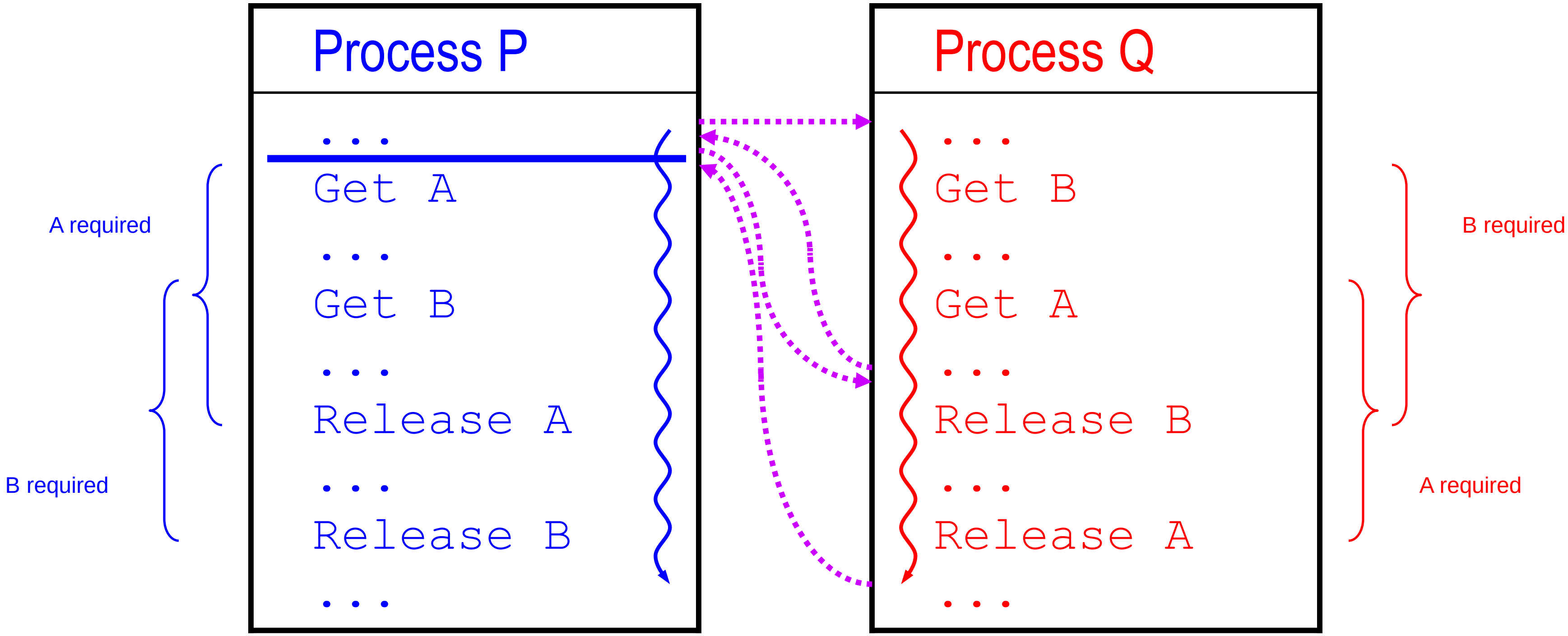
- Illustration of a deadlock – scheduling path 1 ☺
  - Q executes everything before P can ever get A
  - when P is ready, resources A and B are free and P can proceed



Happy scheduling 1

Deadlock Principles

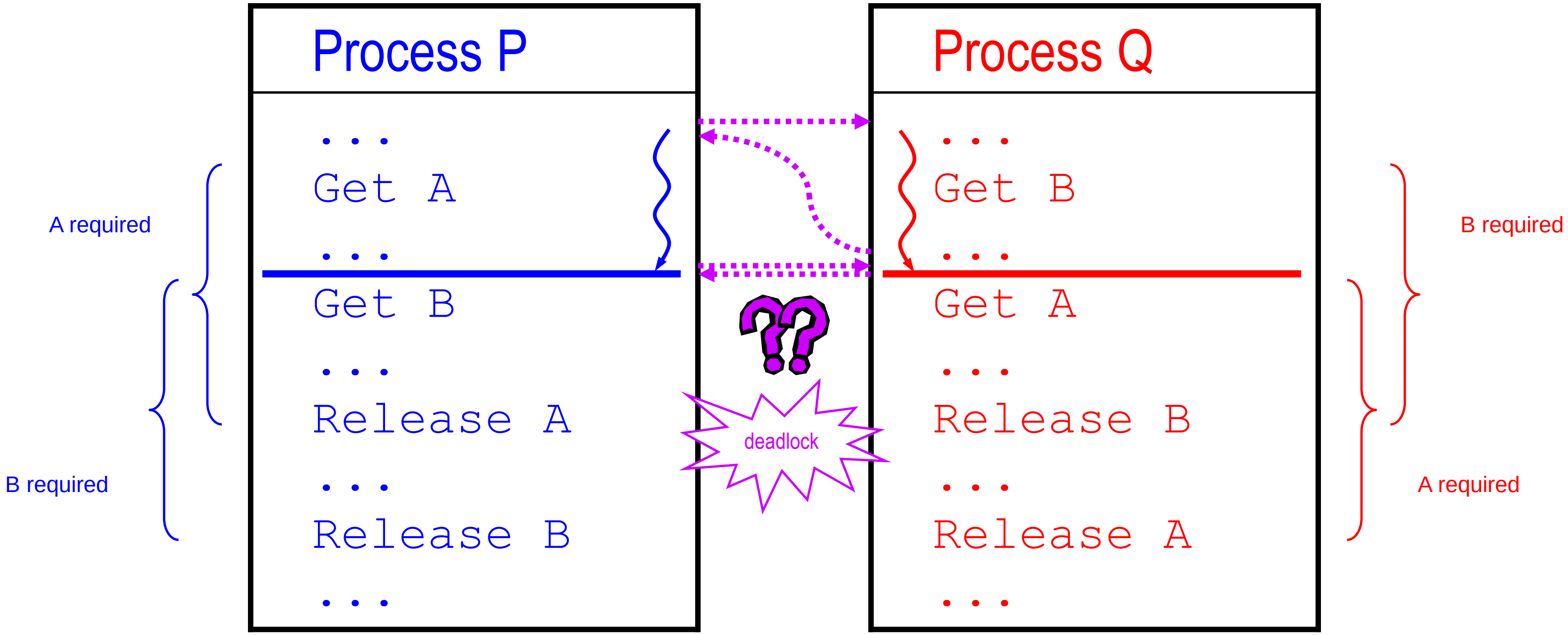
- Illustration of a deadlock – scheduling path 2 ☺
  - Q gets B and A, then P is scheduled; P wants A but is blocked by A's mutex; so Q resumes and releases B and A; P can now go



Happy scheduling 2

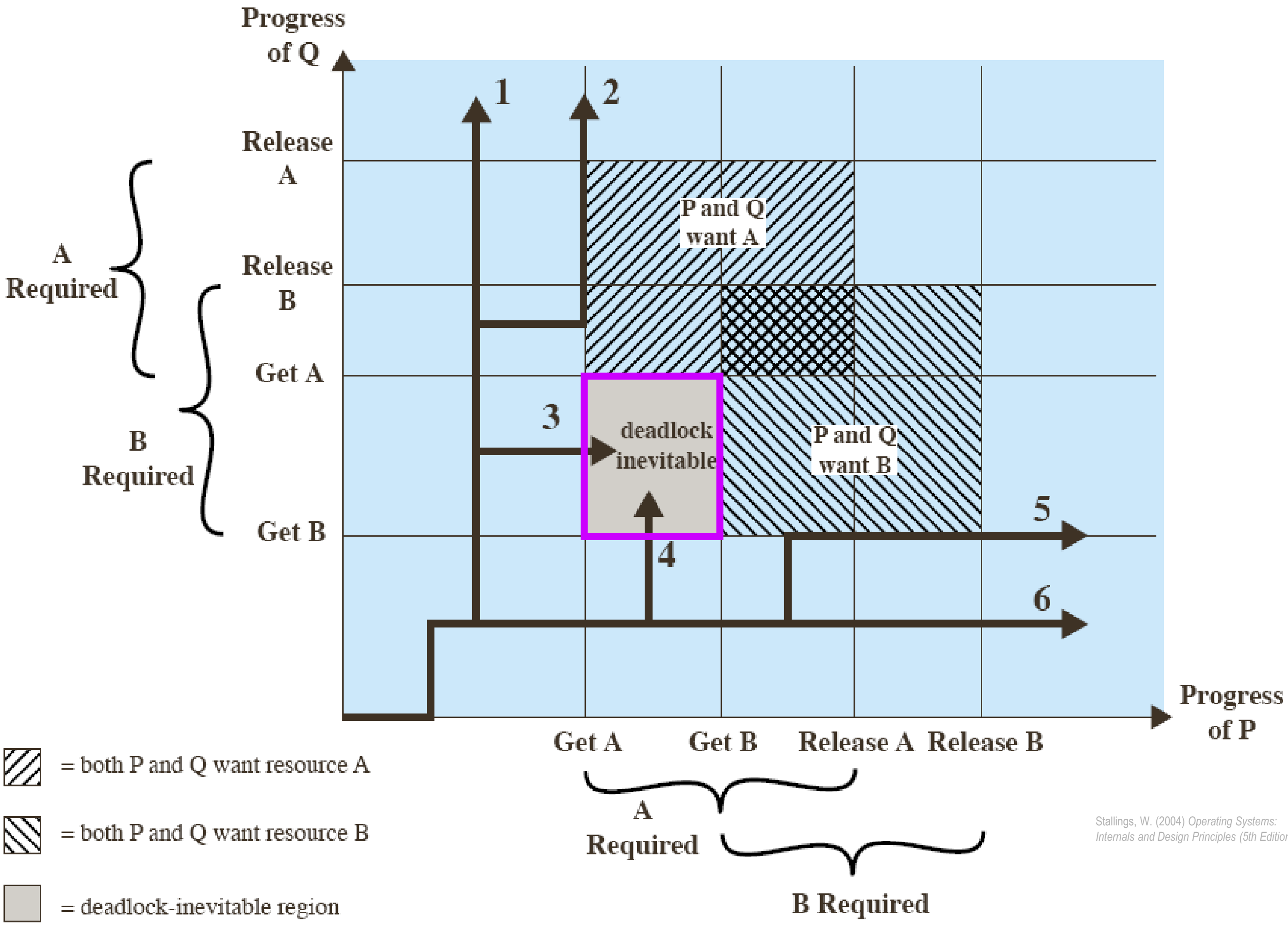
Deadlock Principles

- Illustration of a deadlock – scheduling path 3 ☹
  - Q gets only B, then P is scheduled and gets A; now both P and Q are blocked, each waiting for the other to release a resource



Bad scheduling → deadlock

Deadlock Principles

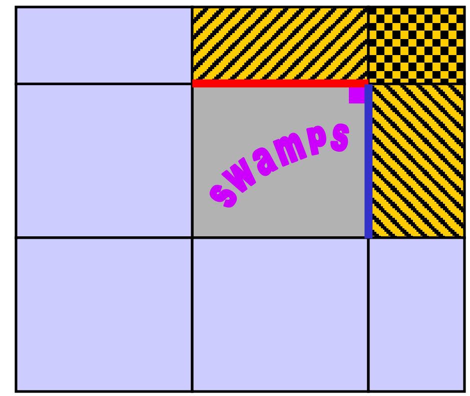


Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

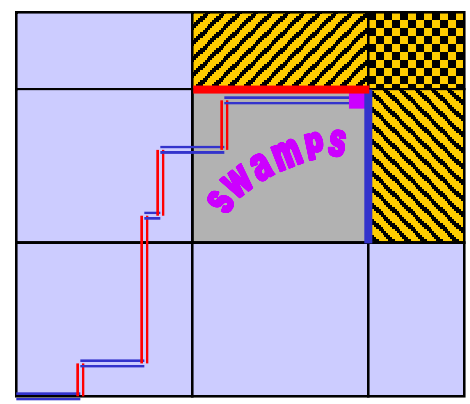
Joint progress diagram

Deadlock Principles

- Deadlocks depend on the program and the scheduling
  - **program design**
    - the order of the statements in the code creates the "landscape" of the joint progress diagram
    - this landscape **may** contain gray "swamp" areas leading to deadlock

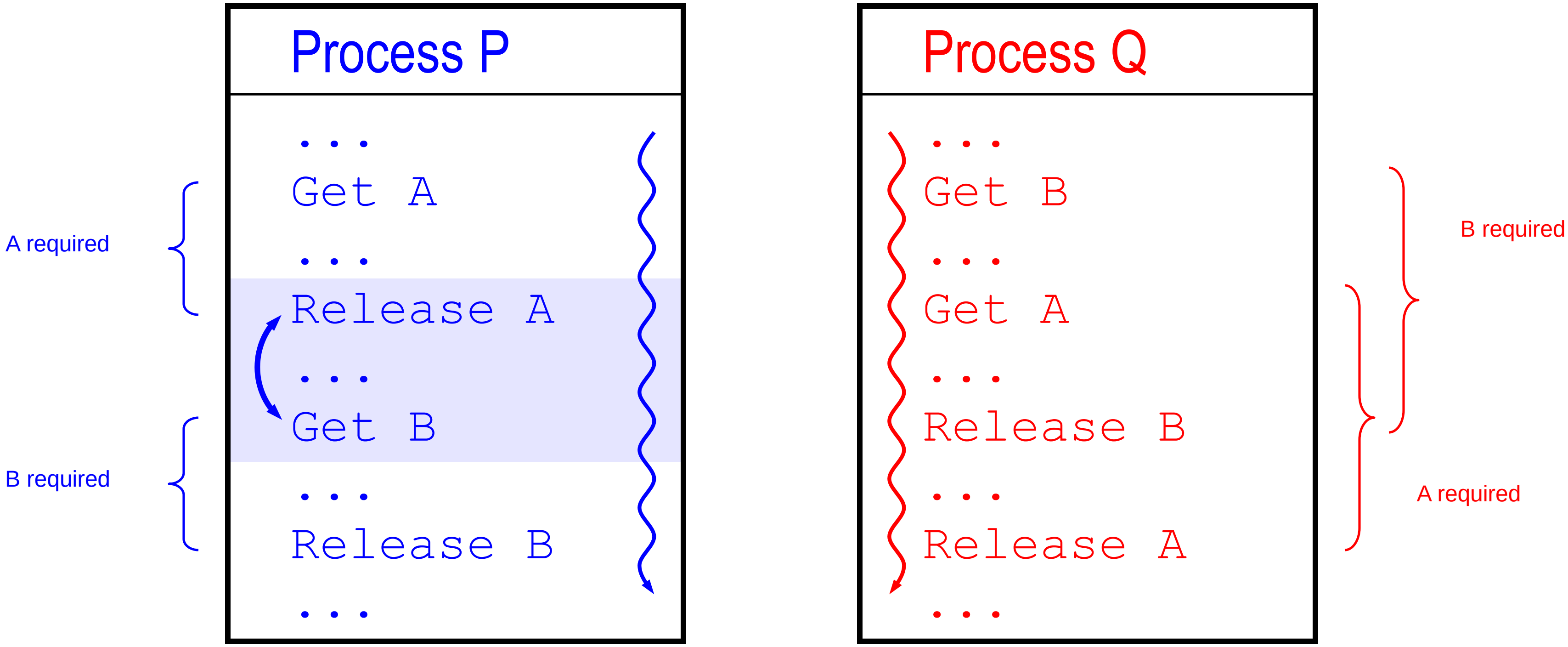


  - **scheduling condition**
    - the interleaved dynamics of multiple executions traces a "path" in this landscape
    - this path **may** sink in the swamps



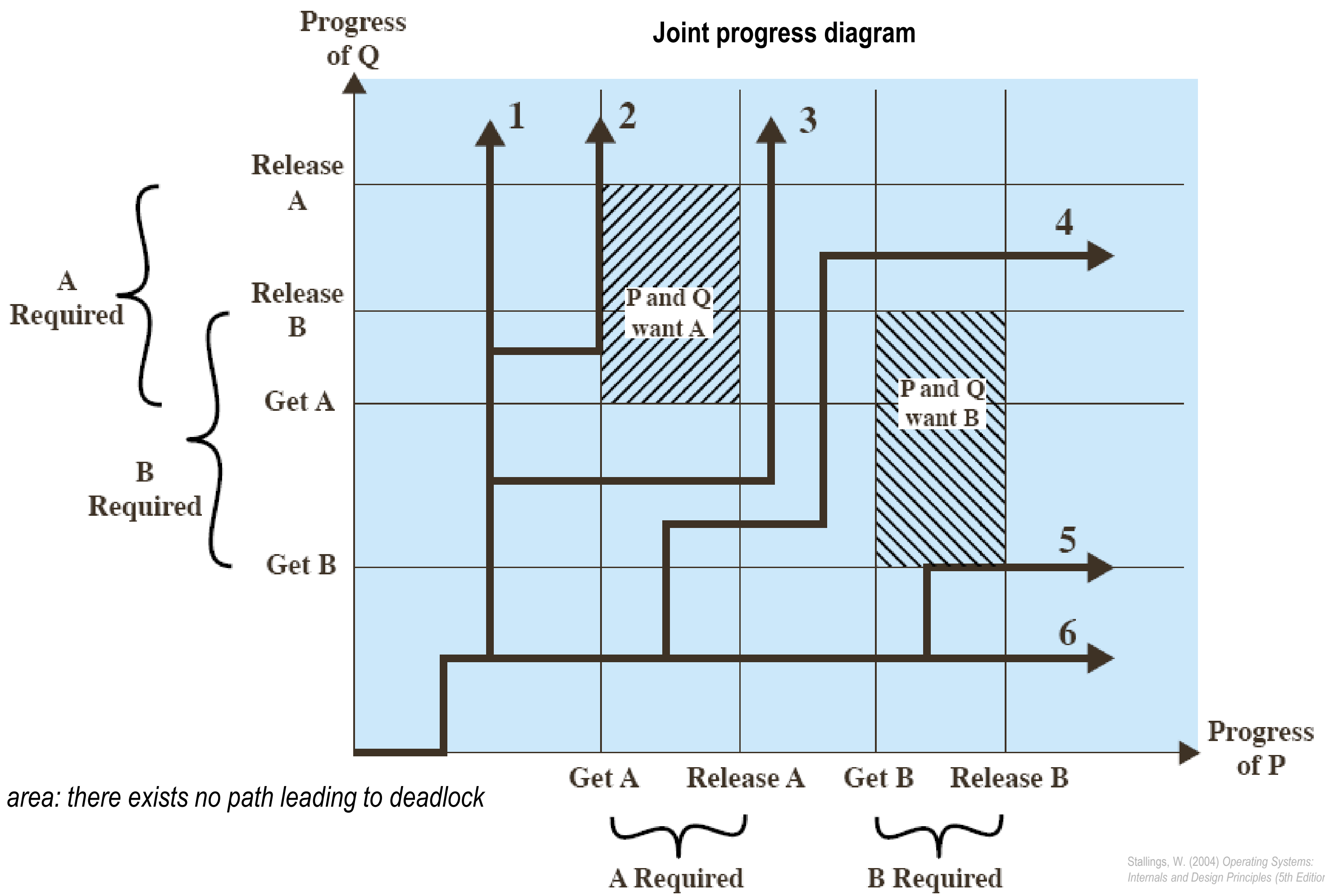
Deadlock Principles

- <u>Changing the program changes the landscape</u>
  - here, P releases A before getting B
  - deadlocks between P and Q are not possible anymore



Competing processes

Deadlock Principles



→ *no swamp area: there exists no path leading to deadlock*

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

System Model

- 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**

Four conditions for deadlock

- Mutual exclusion
  - Each resource is assigned to at most one process
- Hold and wait
  - A process holding resources can request more resources
- No preemption
  - Previously granted resources cannot be forcibly taken away
- Circular wait
  - There must be a circular chain of 2 or more processes where each is waiting for a resource held by the next member of the chain

Deadlock Characterization

- **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

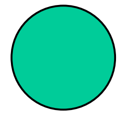
  $P_2, \ldots, P_{n-1}$ is waiting for a resource that is held by P_n, and P_0 is waiting for a resource that is held by P_0.

Resource-Allocation Graph

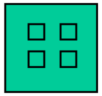
- V is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set consisting of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set consisting of all resource types in the system
- request edge – directed edge $P_1 \rightarrow R_j$
- assignment edge – directed edge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

Resource-Allocation Graph (Cont.)

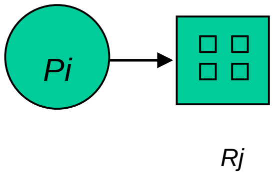
- Process



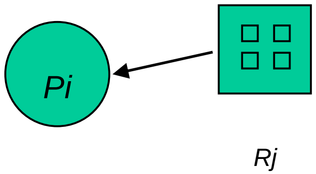
- Resource Type with 4 instances



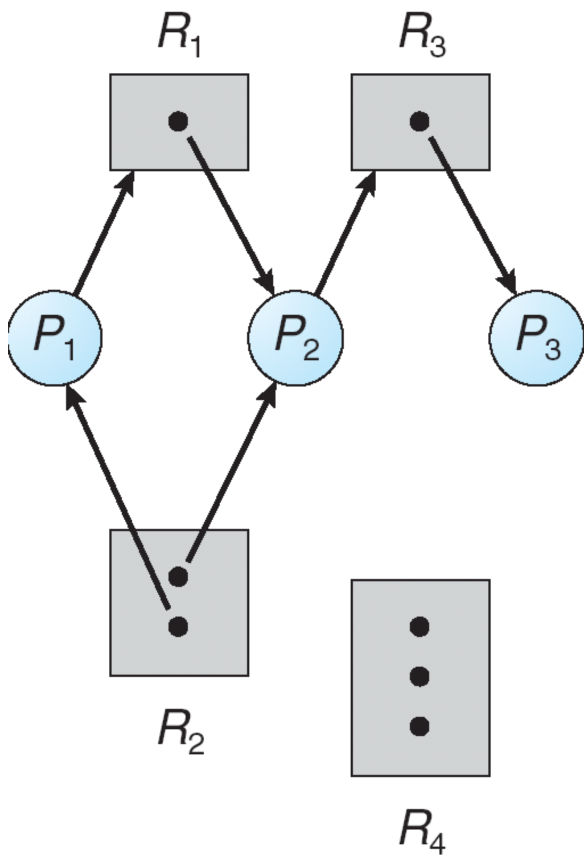
- P_i requests instance of R_j



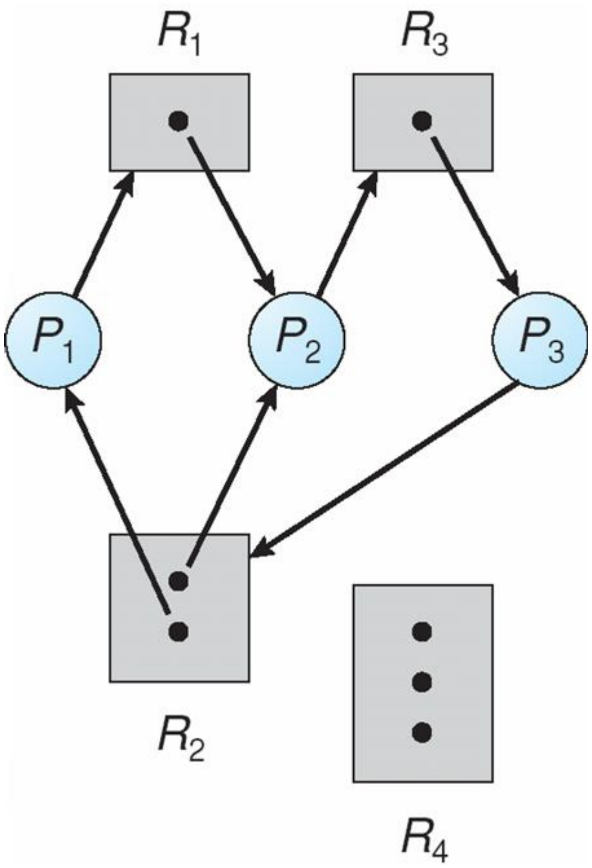
- P_i is holding an instance of R_j



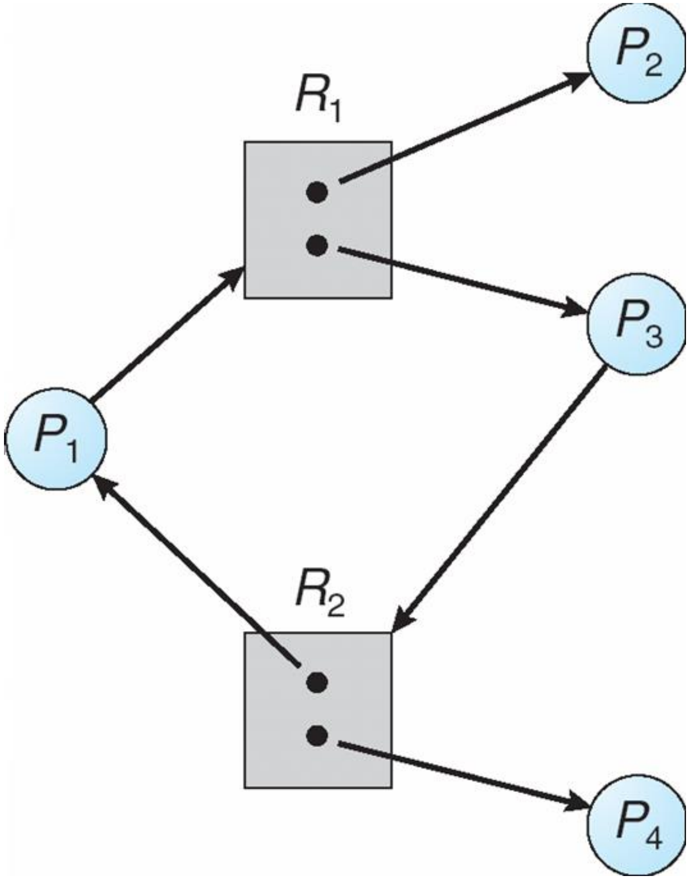
Example of a Resource Allocation Graph



Resource Allocation Graph With A Deadlock



Graph With A Cycle But No Deadlock



Basic Facts

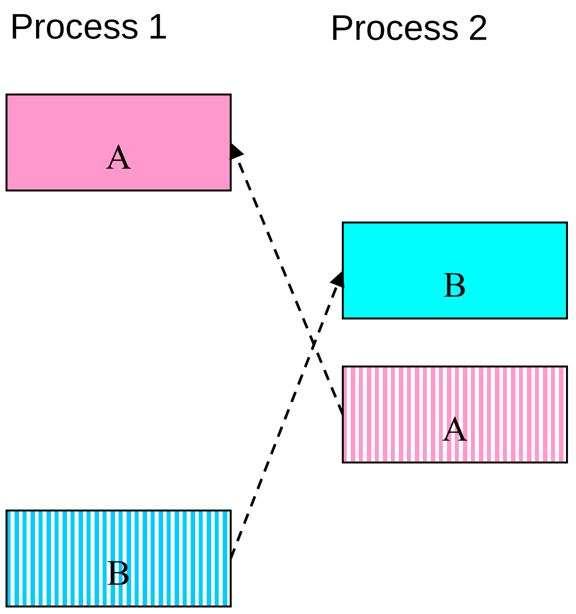
- If graph contains no cycles $\Rightarrow$ no deadlock
- If graph contains a cycle $\Rightarrow$
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

Resources

- Resource: something a process uses
  - Usually limited (at least somewhat)
- Examples of computer resources
  - Printers
  - Semaphores / locks
  - Tables (in a database)
- Processes need access to resources in reasonable order
- Two types of resources:
  - Preemptable resources: can be taken away from a process with no ill effects
  - Nonpreemptable resources: will cause the process to fail if taken away

When do deadlocks happen?

- Suppose
  - Process 1 holds resource A and requests resource B
  - Process 2 holds B and requests A
  - Both can be blocked, with neither able to proceed
- Deadlocks occur when …
  - Processes are granted exclusive access to devices or software constructs (resources)
  - Each deadlocked process needs a resource held by another deadlocked process



DEADLOCK!

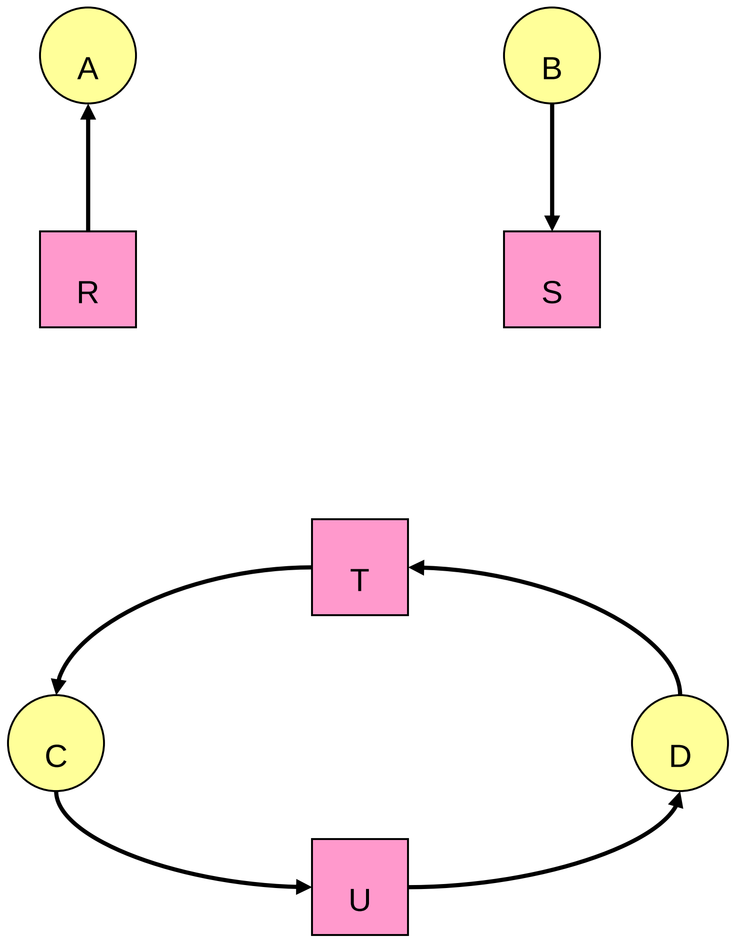
Using resources

- Sequence of events required to use a resource
  - Request the resource
  - Use the resource
  - Release the resource
- Can't use the resource if request is denied
  - Requesting process has options
    - Block and wait for resource
    - Continue (if possible) without it: may be able to use an alternate resource
    - Process fails with error code
  - Some of these may be able to prevent deadlock…

What is a deadlock?

- Formal definition:
  "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 release of a currently held resource
- In deadlock, none of the processes can
  - Run
  - Release resources
  - Be awakened

Resource allocation graphs



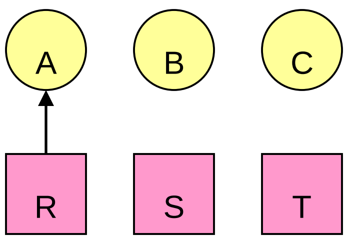
- Resource allocation modeled by directed graphs
- Example 1:
  - Resource R assigned to process A
- Example 2:
  - Process B is requesting / waiting for resource S
- Example 3:
  - Process C holds T, waiting for U
  - Process D holds U, waiting for T
  - C and D are in deadlock!

Dealing with deadlock

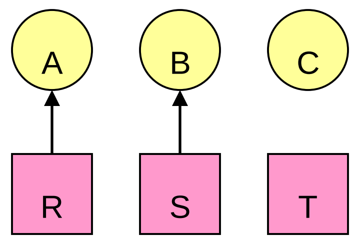
- How can the OS deal with deadlock?
  - Ignore the problem altogether!
    - Hopefully, it'll never happen…
  - Detect deadlock & recover from it
  - Dynamically avoid deadlock
    - Careful resource allocation
  - Prevent deadlock
    - Remove at least one of the four necessary conditions
- We'll explore these tradeoffs

Getting into deadlock

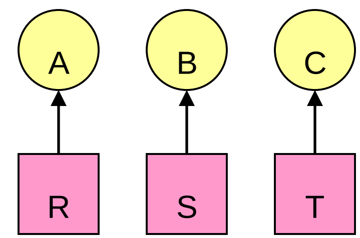
A	B	C
Acquire R Acquire S Release R Release S	Acquire S Acquire T Release S Release T	Acquire T Acquire R Release T Release R



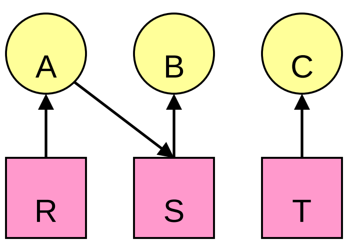
Acquire R



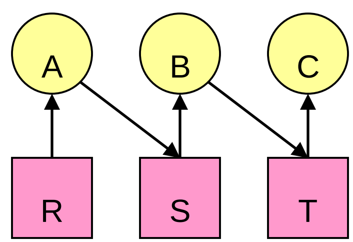
Acquire S



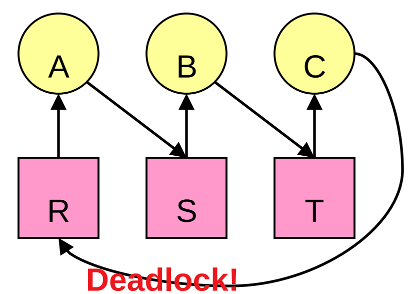
Acquire T



Acquire S



Acquire T



Acquire R

Not getting into deadlock…

- Many situations *may* result in deadlock (but don't have to)
  - In previous example, A could release R before C requests R, resulting in no deadlock
  - Can we always get out of it this way?
- Find ways to:
  - Detect deadlock and reverse it
  - Stop it from happening in the first place

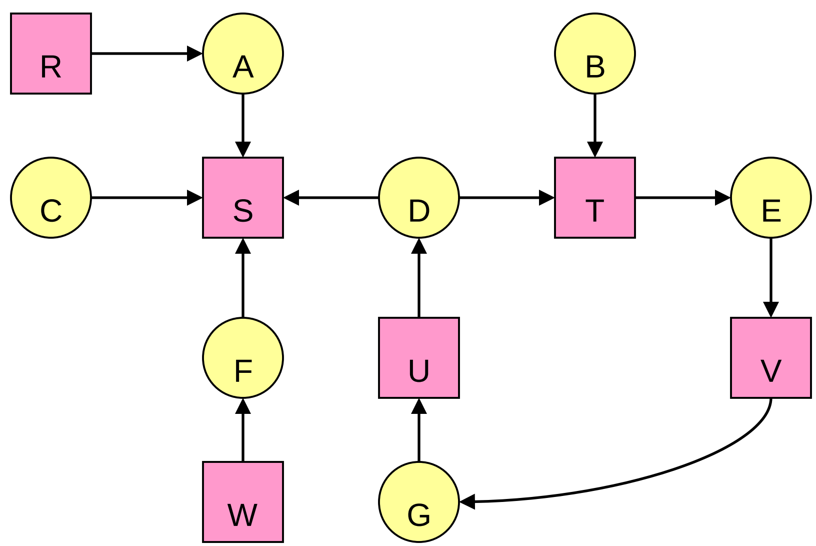
The Ostrich Algorithm

- Pretend there's no problem
- Reasonable if
  - Deadlocks occur very rarely
  - Cost of prevention is high
- UNIX and Windows take this approach
  - Resources (memory, CPU, disk space) are plentiful
  - Deadlocks over such resources rarely occur
  - Deadlocks typically handled by rebooting
- Trade off between convenience and correctness

Detecting deadlocks using graphs

- Process holdings and requests in the table and in the graph (they're equivalent)
- Graph contains a cycle => deadlock!
  - Easy to pick out by looking at it (in this case)
  - Need to mechanically detect deadlock
- Not all processes are deadlocked (A, C, F not in deadlock)

Process	Holds	Wants
A	R	S
B		T
C		S
D	U	S,T
E	T	V
F	W	S
G	V	U



Deadlock detection algorithm

- General idea: try to find cycles in the resource allocation graph
- Algorithm: depth-first search at each node
  - Mark arcs as they're traversed
  - Build list of visited nodes
  - If node to be added is already on the list, a cycle exists!
- Cycle == deadlock

```
For each node N in the graph {
  Set L = empty list
  unmark all arcs
  Traverse (N,L)
}
If no deadlock reported by now, there isn't any

define Traverse (C,L)  {
  If C in L, report deadlock!
  Add C to L
  For each unmarked arc from C {
    Mark the arc
    Set A = arc destination
    /* NOTE: L is a
       local variable */
    Traverse (A,L)
  }
}
```

Deadlock Avoidance

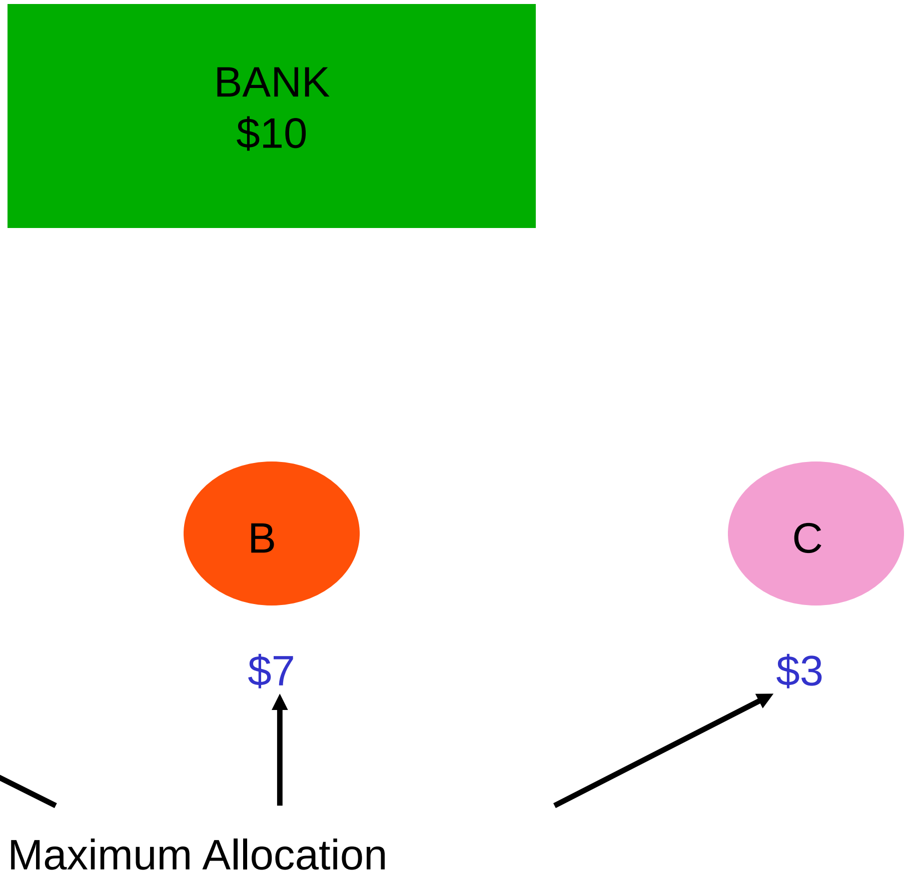
- makes sure the system stays in a safe state if a request is satisfied
- prevents circular waits
- requires additional information at the beginning of the execution of a process
  - maximum number of instances per resource type for each process

Safe State

- A system is in a **safe state** when given the current allocation the system is able to handle any number of requests, in some order, without getting into a deadlock.

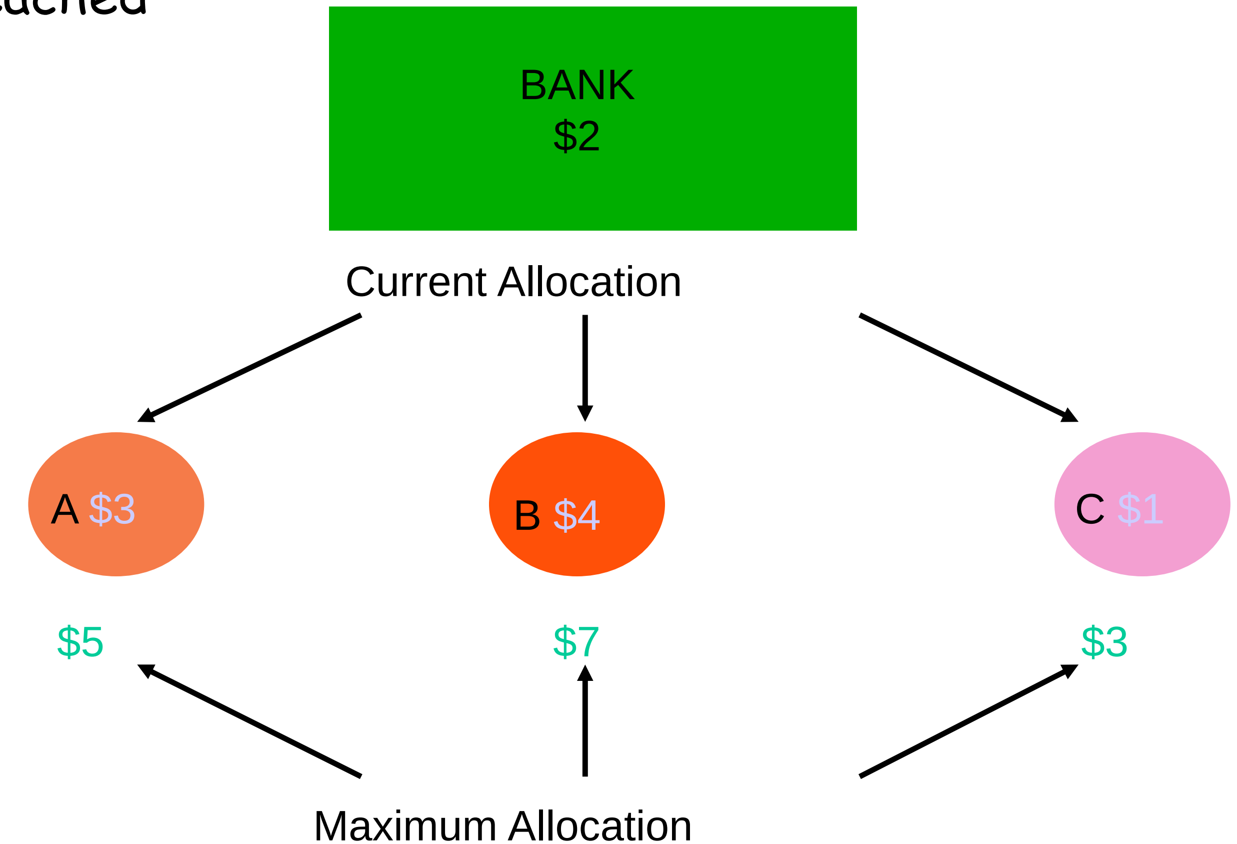
Example

- Bank gives loans to customers
  - maximum allocation = credit limit



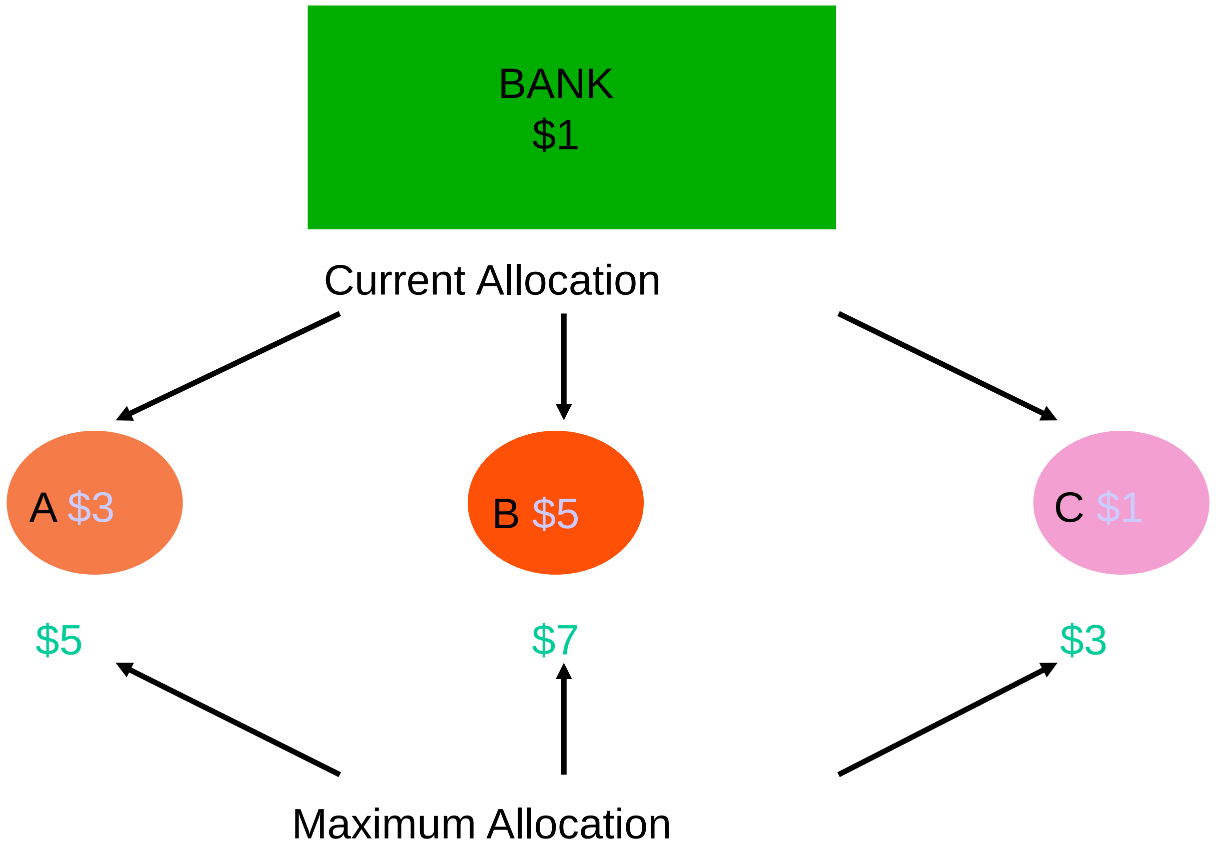
❐ Safe State?

- Will the bank be able to give each customer a loan up to the full credit limit?
  - not necessarily all customers simultaneously
  - order is not important
  - customers will pay back their loan once their credit limit is reached



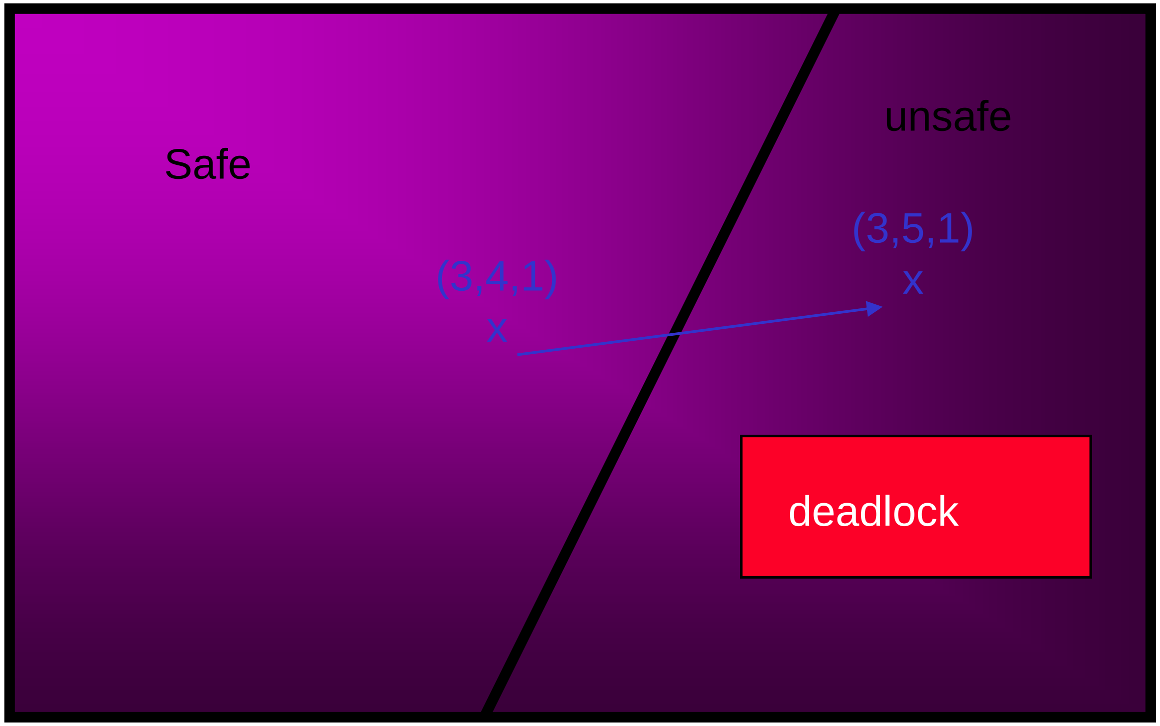
❑ Still Safe?

- after customer B requests and is granted $1, is the bank still safe?



Safe State Space

Bank Safe State Space



Banker's Algorithm

- before a request is granted, check the system's state
  - assume the request is granted
  - if it is still safe, the request can be honored
  - otherwise the process has to wait
  - overly careful
    - there are cases when the system is unsafe, but not in a deadlock

Example Banker's Algorithm

- Initially given: current allocation, maximum allocation, available resources

Process	Current Allocation	Maximum Allocation	Current Work Available	Remaining Needed = Max -Aval	Total Available
	A B C	**A B C**	**A B C**	**A B C**	**A B C**
P0	0 1 0	7 5 3			**10 5 7**
P1	2 0 0	3 2 2			
P2	3 0 2	9 0 2			
P3	2 1 1	4 2 2			
P4	0 0 2	5 3 3			
	7 2 5				

Example Banker's Algorithm

- First check which process is in the safe state.
- $Need_i \leq Work \Rightarrow Work = Work + Allocation$
  - If(RemaingNeeded <= CurrentWorkAvailable)

    CurrentWorkAvailable = CurrentWorkAvailable + CurrentAllocation
- Update the safe sequence.

Process	Current Allocation	Maximum Allocation	Current Work Available	Remaining Needed = Max -Aval	Total Available
	A B C	**A B C**	**A B C**	**A B C**	**A B C**
P0	0 1 0	7 5 3			**10 5 7**
P1	2 0 0	3 2 2			
P2	3 0 2	9 0 2			
P3	2 1 1	4 2 2			
P4	0 0 2	5 3 3			
	7 2 5				

Example Banker's Algorithm

- Initially given current allocation, maximum allocation, available resources
  - If(RemaingNeeded <= CurrentWorkAvailable)
    CurrentWorkAvailable = CurrentWorkAvailable + CurrentAllocation

Process	Current Allocation	Maximum Allocation	Current Work Available	Remaining Needed = Max -Aval	Total Available
	A B C	**A B C**	**A B C**	**A B C**	**A B C**
P0	0 1 0	7 5 3	3 3 2	7 4 3	**10 5 7**
P1	2 0 0	3 2 2	5 3 2	1 2 2	
P2	3 0 2	9 0 2	7 4 3	6 0 0	
P3	2 1 1	4 2 2	7 4 5	2 1 1	
P4	0 0 2	5 3 3	7 5 5	5 3 1	
	7 2 5		**10 5 7**		