Deadlocks

CPSC 457: Principles of Operating Systems
Winter 2024

Contains slides from Pavol Federl, Mea Wang, Andrew Tanenbaum and Herbert Bos, Silberschatz, Galvin and Gagne

Jonathan Hudson, Ph.D.
Instructor
Department of Computer Science
University of Calgary

Tuesday, 28 November 2024

UNIVERSITY OF CALGARY

Topics

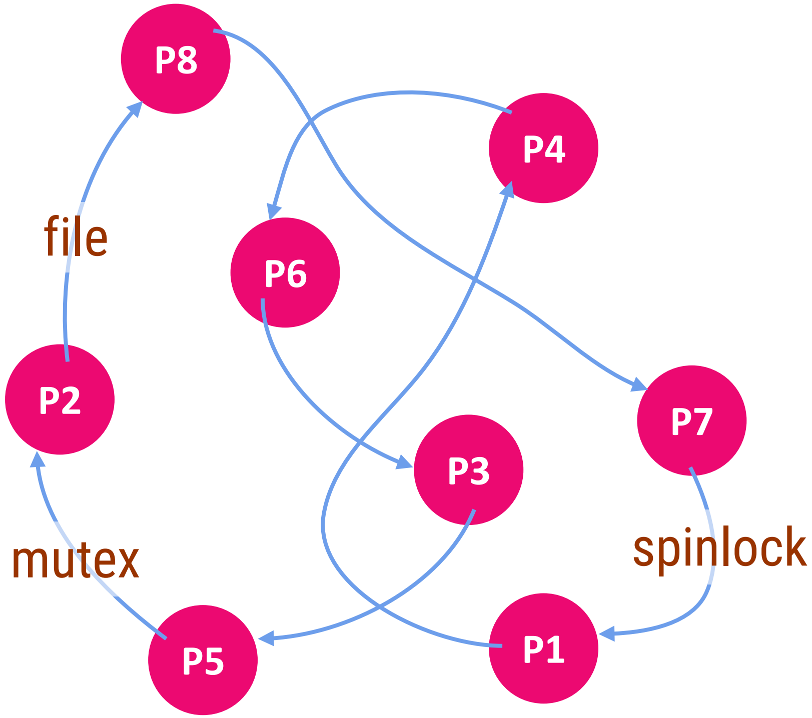
- system model (assumptions and simplifications)
- deadlock characterization
- methods for handling deadlocks
  - deadlock prevention
  - deadlock avoidance
  - deadlock detection
  - recovery from deadlock

Definition

Deadlock definition

- a set of processes are in a deadlock if:
  - each process in the set is waiting for an event; **and**
  - that event can be caused only by another process in the set.

- event could be anything, e.g.
  - resource becoming available
  - mutex/semaphore/spinlock being unlocked
  - message arriving



System Model

System model

- system consists of N processes and M resources
- resources could be files, global variables, etc. or mutexes protecting them
- in most systems each resource type has a single instance
  - we'll mostly focus on this scenario, since usually we assign unique mutexes to each resource instance
  - e.g. multiple shared counters, each protected by individual mutexes
- in some systems we could have multiple instances per resource type
  - a process could request "an instance" of a type
  - e.g. 5 identical disks, 3 identical printers, and a process could request *"one printer, does not matter which one"*

UNIVERSITY OF CALGARY

System model

- we assume processes/threads are well behaved (programs are well written)
- each process utilizes a resource in the same manner:
    1. a process **requests** the resource, and OS may block such process
    2. a process **uses** the resource for a **finite amount** of time
    3. a process **releases** the resource, OS may unblock related process(es)

Conditions

Deadlock – sufficient and necessary conditions

- **mutual exclusion** condition
  - resources are not shareable (max. one process per resource)
- **hold and wait** condition
  - a process holding at least one resource is waiting to acquire additional resources
- **no preemption** condition
  - resource cannot be stolen (can only be released voluntarily by the process holding it)
- **circular wait** condition
  - there is an ordering of processes { P_1 , P_2 , … , P_n }, such that
    - P_1 waits for P_2
    - P_2 waits for P_3

      …
    - P_n waits for P_1
  - i.e. there is a cycle

Deadlock can arise if and only if all four conditions hold simultaneously!

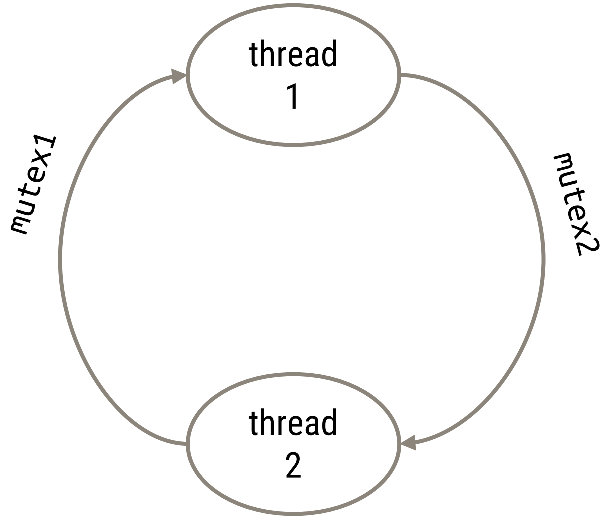
These are also known as **Coffman conditions**.

Deadlock with mutex locks

- deadlocks can occur in many different ways, usually due to locking
- simple example – deadlock with 2 mutexes:

```
Thread 1:

lock( mutex1 );
lock( mutex2 );
// critical section
unlock( mutex2 );
unlock( mutex1 );
```

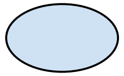
```
Thread 2:

lock( mutex2 );
lock( mutex1 );
// critical section
unlock( mutex1 );
unlock( mutex2 );
```



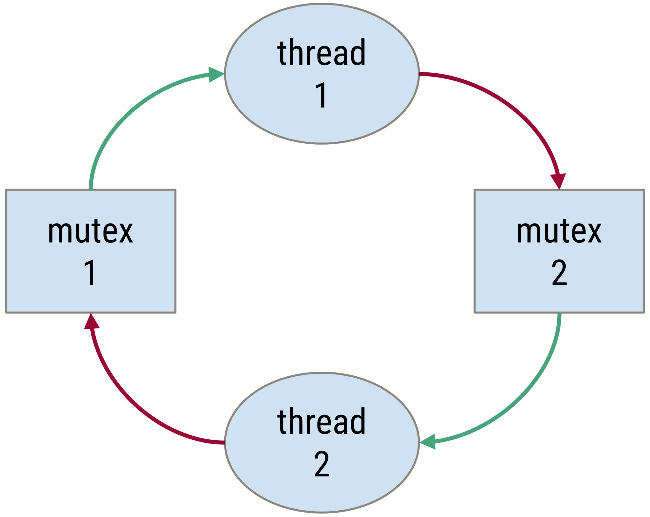
- all 4 necessary conditions present:
  mutual exclusion, hold and wait, no pre-emption, circular wait

Resource-Allocation Graph

Resource-Allocation Graph with 1 instance per resource type

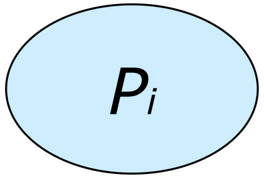
- system state can be represented by a directed graph G(V,E)
- two types of vertices
  - processes represented as ellipsoids 
  - resources represented as rectangles 
- two types of edges
  - request edge — pointing from process → resource
    representing a process requesting unique access to resource

  - assignment edge — pointing from resource → process
    representing process having unique access to resource

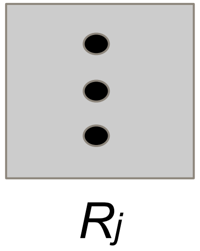


Resource-Allocation Graph with multiple instances per resource

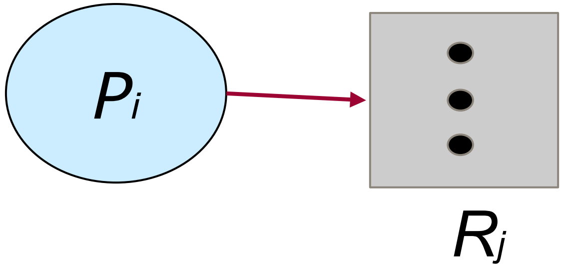
- **process** P_i :



- **multiple instances** of **resource type** are represented as dots inside resources, e.g. resource R_j with 3 instances:

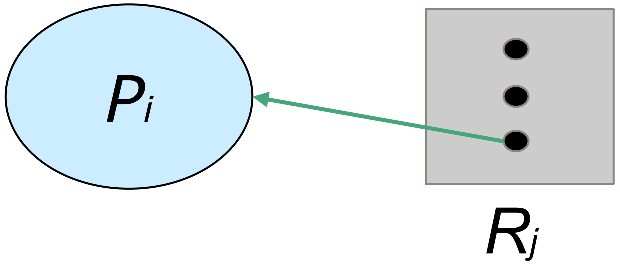


- *Pi* **requests** an instance of R_j :



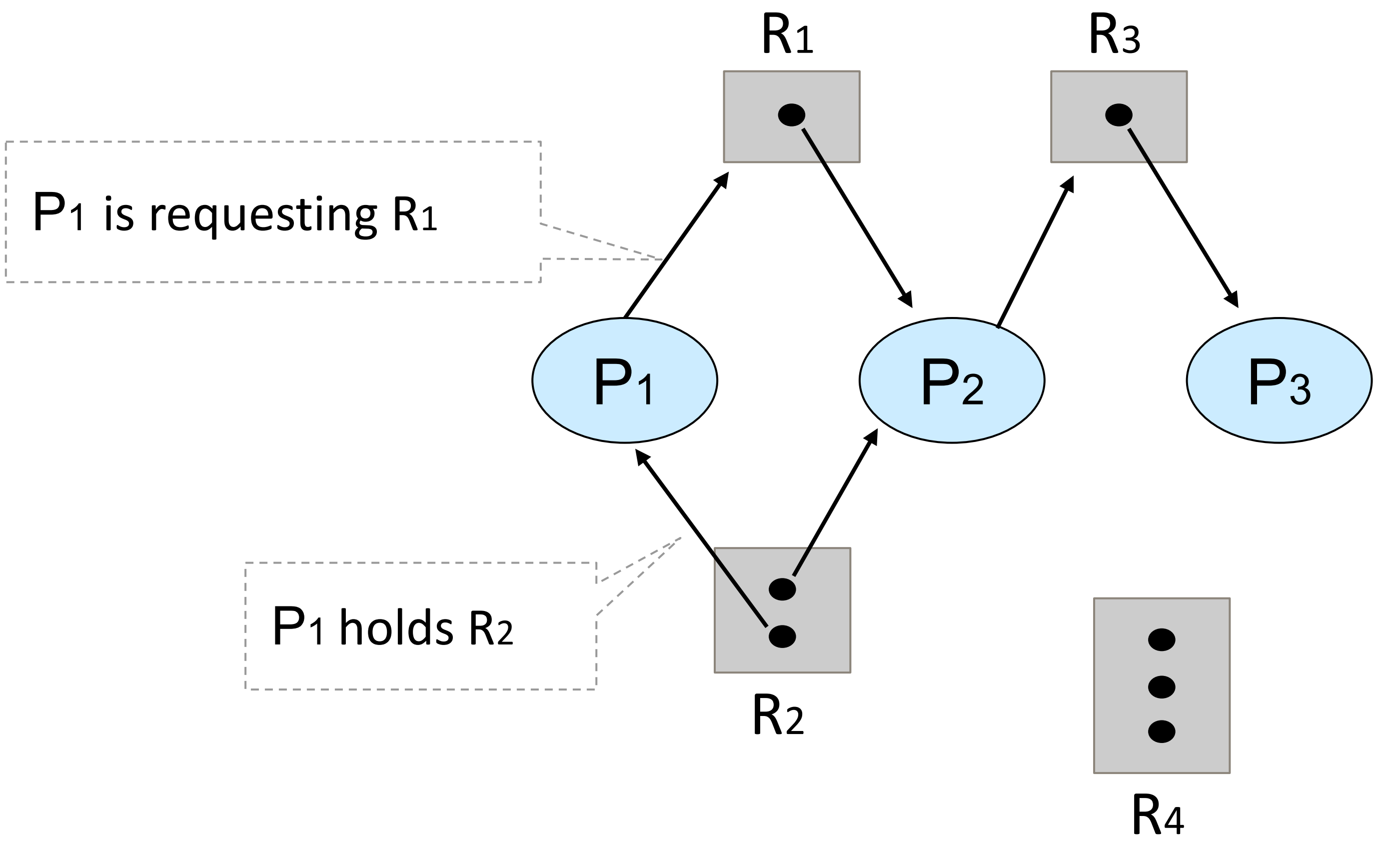
request edge points to resource type, not resource instance

- *Pi* is **holding** an instance of R_j :



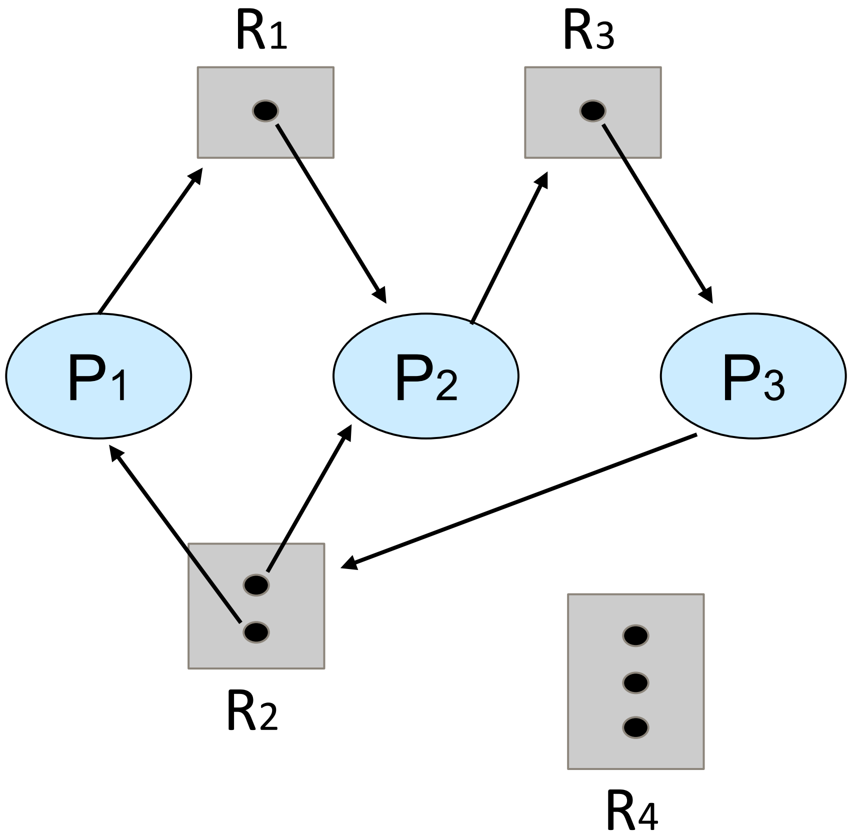
- assignment edge originates from instance, not type

Resource Allocation Graph Example



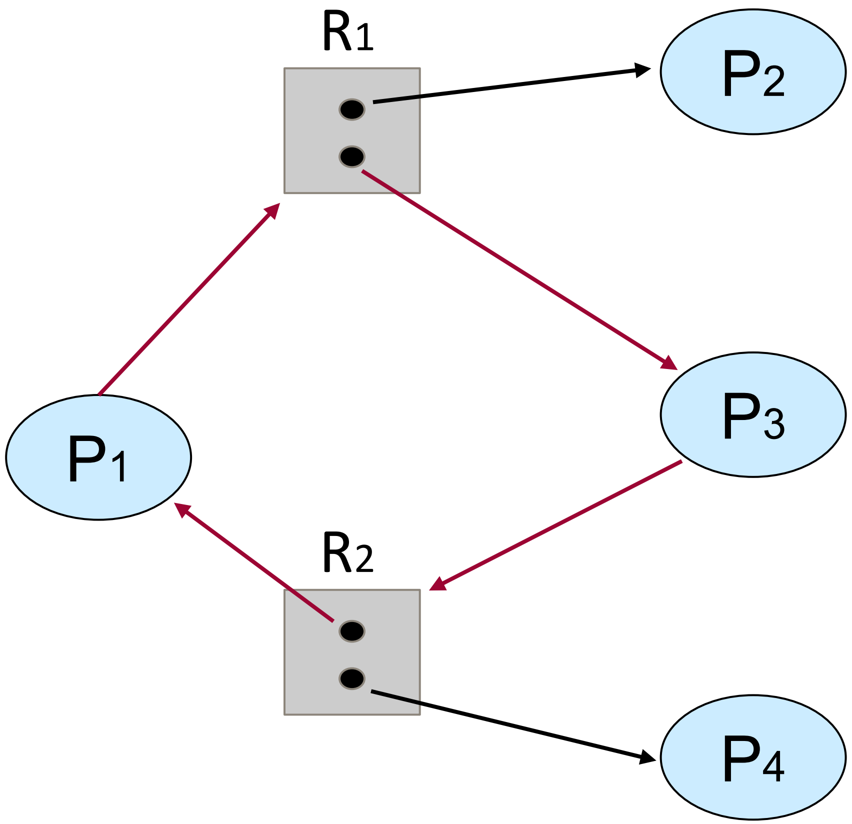
no cycle ⇒ no deadlock

Resource Allocation Graph With A Deadlock



deadlock ⇒ cycle

Graph With A Cycle But No Deadlock



but … cycle $\not\Rightarrow$ deadlock

Deadlock vs Cycle

- if graph contains no cycles ⇒ no deadlock
  - holds for both single-instance and for multiple-instances per resource type
- if graph contains a cycle ...
  - if only one instance per resource type ⇒ **guaranteed** deadlock
  - if multiple instances per resource type ⇒ **possible** deadlock

Example

Example

- consider 3 processes A, B and C which want to perform operations on resources R, S and T:

```
Process A:

1. request R
2. request S
3. release R
4. release S
```

```
Process B:

1. request S
2. request T
3. release S
4. release T
```

```
Process C:

1. request T
2. request R
3. release T
4. release R
```

- depending on the order in which we process and grant the requests, we may end up:
  - with a deadlock
  - or no deadlock

Example: sequence of operations leading to deadlock

```
Process A:

request R
request S
release R
release S
```

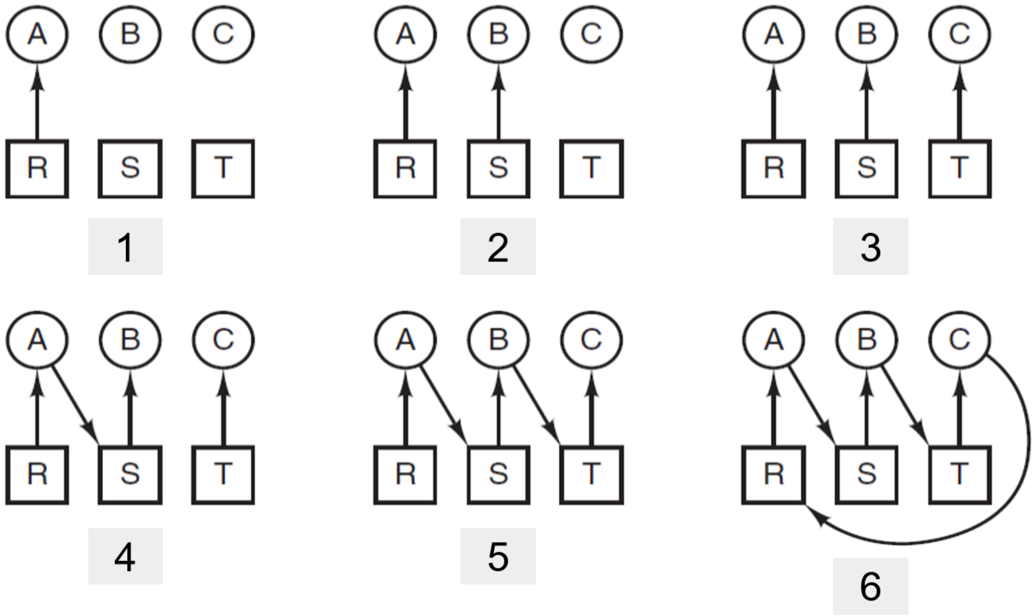
```
Process B:

request S
request T
release S
release T
```

```
Process C:

request T
request R
release T
release R
```

This sequence:

1. A requests R - granted
2. B requests S - granted
3. C requests T - granted
4. A requests S - blocked
5. B requests T - blocked
6. C requests R - blocked

⇒ leads to a **deadlock**



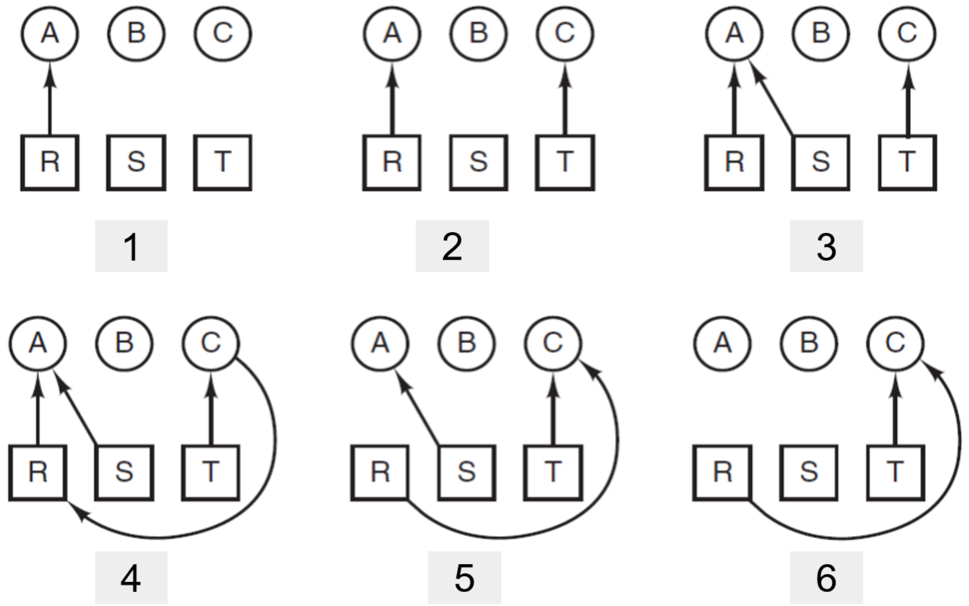
Example: sequence of operations **not** leading to deadlock

Process A:	Process B:	Process C:
request R	request S	request T
request S	request T	request R
release R	release S	release T
release S	release T	release R

This sequence:

1. A requests R - granted
2. C requests T - granted
3. A requests S - granted
4. C requests R - blocked
5. A releases R - unblocks C
6. A releases S
7. …

⇒ does not lead to a **deadlock**



Handling Deadlocks

Methods for dealing with deadlocks

- ignore the problem:
  - pretend that deadlocks never occur in the system
  - approach of many operating systems, including UNIX
  - it's up to applications to address their deadlock issues

- ensure that the system will never enter a deadlock state:
  - deadlock **prevention**
  - deadlock **avoidance**

- allow the system to enter a deadlock state and then recover:
  - deadlock **detection**
  - **recovery** from deadlock

Prevention

Deadlock prevention

- deadlock prevention = any technique that prevents one of the 4 necessary conditions

- avoiding **mutual exclusion** condition:
  - mutual exclusion not required for shareable resources
    e.g. no shared resources, read-only global variables or files, lock-free operations
  - spooling can help for some resource types (e.g. printers) to convert them to shareable
  - not practical in many/most cases

- attacking **hold and wait** condition:
  - whenever a process requests a resource, it cannot hold any other resources
    - option 1: process must request all needed resources at the beginning
    - option 2: process can request resources only when it has no resources
  - often leads to low resource utilization & possibility of starvation,
    especially with large number of resources

attacking hold and wait condition

Deadlock prevention - example

- how do we fix the deadlock possibility below by avoiding hold and wait condition?

```
Thread 1:

lock( mutex1 );
  /* use resource 1 */
lock( mutex2 );
  /* use resources 1 and 2 */
unlock( mutex2 );
unlock( mutex1 );
```

```
Thread 2:

lock( mutex2 );
  /* use resource 2 */
lock( mutex1 );
  /* use resources 1 and 2 */
unlock( mutex1 );
unlock( mutex2 );
```

Deadlock prevention - example

- option 1: acquire all resources at the beginning
- if we had `lockn()` - that atomically locks multiple mutexes at once:

```
Thread 1:

lockn( mutex1, mutex2);
  /* use resources 1 and 2 */
unlock( mutex2 );
unlock( mutex1 );
```

```
Thread 2:

lockn( mutex2, mutex1);
  /* use resources 1 and 2 */
unlock( mutex1 );
unlock( mutex2 );
```

Deadlock prevention - example

- option 2: release resources before acquiring more
- if we had `unlockAndLock()` - unlock all locked mutexes, then lock them all atomically

```
Thread 1:

unlockAndLock( mutex1 )
  /* use resource 1 */
unlockAndLock( mutex2, mutex1);
  /* use resources 1 and 2 */
unlock( mutex2 );
unlock( mutex1 );
```

```
Thread 2:

unlockAndLock( mutex2 )
  /* use resource 2 */
unlockAndLock( mutex1, mutex2);
  /* use resources 1 and 2 */
unlock( mutex1 );
unlock( mutex2 );
```

- both options could lead to non-optimal resource utilization and even starvation

see `std::lock` and `std::scoped_lock` for a partial solution

avoiding no preemption condition

Deadlock prevention

- avoiding **no preemption** condition:
  - if a process that is holding some resources requests another resource that cannot be immediately allocated to it, the process is suspended, and all resources currently held by it are released
  - these preempted resources are added to the list of resources for which the process is waiting
  - process will be resumed when it can regain its old & new resources
  - only works with resources for which we can save/restore the state (e.g. CPU registers)
- complicated mechanism, possible starvation, non-optimal use of resources

avoiding circular wait condition

Deadlock prevention

- avoiding **circular wait** condition
  - most practical condition to avoid
  - accomplished by establishing an ordering of resources, e.g. via resource hierarchy,
  - each process must requests resources in an increasing order of enumeration
  - e.g. lock mutexes in the same order by all threads,
    quite practical for small number of mutexes / resources

```
Thread 1:

lock( mutex1);
lock( mutex2);
  /* critical section */
unlock( mutex2 );
unlock( mutex1 );
```

```
Thread 2:

lock( mutex1);
lock( mutex2);
  /* critical section */
unlock( mutex1 );
unlock( mutex2 );
```

Resource ordering in C/C++

- if all resources are protected using global mutexes/semaphores/spinlocks, then resource ordering can be implemented by comparing their addresses

```
if ( & m1 < & m2) {
    lock(m1); lock(m2);
} else {
    lock(m2); lock(m1);
}
```

- similarly, if all resources have unique IDs that can be compared to each other, we can use these IDs to determine the order in which we lock the resources
    - e.g. for files we could use paths as IDs

```
if ( id(file1) < id(file2)) {
    lock(file1); lock(file2);
} else {
    lock(file2); lock(file1);
}
```

Deadlock Example

- imagine we are writing code that performs multiple transfers between different accounts
- we need to make it multithreaded
- class `Account` represents someone's account, e.g. an entry in a database
- it has 2 thread-safe methods: `.get()` and `.set()` for retrieving/setting the value, e.g.

```
class Account {
  ...
public:
  double get();
  void set(double amount);
  ...
};
```


UNIVERSITY OF CALGARY

Deadlock Example

- let's implement `transaction()` that transfers `amount` from account `a1` into account `a2`

```
void transaction(Account a1, Account a2, double amount) {
    adjust(a1, -amount); // withdraw from a1
    adjust(a2,  amount); // deposit into a2
}
```

- where a helper function `adjust(a,v)` adjusts account's amount by some value

```
void adjust(Account a, double value) {
    double tmp = a.get();
    tmp = tmp + value;
    a.set(tmp);
}
```

thread safe

adjust() is **not** thread safe!!!

`adjust`() is not thread-safe, therefore `transaction`() is also not thread-safe

- how do we make `transaction()` thread-safe, so that we can transfer money in parallel?

  i.e. we want to call `transaction()` from multiple threads

Deadlock Example

- let's write a thread-safe version of `adjust()`
- if we can get a unique mutex per account, e.g. using `get_mutex(a)`, we could write:

```
void adjust_r(Account a, double value) {
   mutex m = get_mutex(a);
   lock(m);
     a.set(a.get() + value);
   unlock(m);
}
```

- but `transaction()` might still not be thread safe…

```
void transaction(Account a1, Account a2, double amount) {
   adjust_r(a1, -amount); // withdraw
   /* trouble region - money missing */
   adjust_r(a2,  amount); // deposit
}
```

e.g. maybe there is a thread that periodically sums up all accounts in the database, it may get the wrong sum

Deadlock Example

- instead of fixing `adjust()` to prevent race condition, let's fix `transaction()` by locking both mutexes before modifying any accounts

```
void transaction_r(Account a1, Account a2, double amount) {
   mutex m1 = get_mutex(a1); // get exclusive access to account a1
   mutex m2 = get_mutex(a2); // get exclusive access to account a2
   lock(m1); lock(m2);
     adjust(a1, -amount);  // withdraw
     adjust(a2,  amount);  // deposit
   unlock(m2); unlock(m1);   // order does not matter here
}
```

no more race conditions for parallel invocations of transaction()

- summation thread could also work — provided it locks the mutex to read the account
- New problem: possible **deadlock**, can you spot it?

Note: to increase performance if we need to support readers, we could use read-write locks, e.g. `pthread_rwlock_t`, or `std::shared_mutex`

Deadlock Example with Lock Ordering

- imagine 2 transactions execute concurrently:
  - thread 1 calls `transaction( "a", "b", 20);`
  - thread 2 calls `transaction( "b", "a", 10);`
- depending on the order of execution, we could get a deadlock

```
Thread 1:

    mutex m1 = get_mutex("a");
    mutex m2 = get_mutex("b");
    lock(m1);
    lock(m2);

    adjust(a1, - amount);
    adjust(a2,   amount);
```

```
Thread 2:

    mutex m1 = get_mutex("b");
    mutex m2 = get_mutex("a");
    lock(m1);
    lock(m2);

    adjust(b, - amount);
    adjust(a,   amount);
```

Deadlock Example

- we can change the the locking order based on resource ordering
- imagine the `Account` class offers a unique ID using `.id()` method

```
void transaction(Account a1, Account a2, double amount)
{
    mutex m1 = get_mutex(a1);
    mutex m2 = get_mutex(a2);
    if( a1.id() < a2.id())
        swap(m1, m2);
    lock(m1); lock(m2);
        adjust(a1, -amount);
        adjust(a2,  amount);
    unlock(m2); unlock(m1);
}
```

alternative to using swap:

```
if( a1.id() < a2.id()) {
    lock(m1); lock(m2);
} else {
    lock(m2); lock(m1);
}
```

- 2 extra lines of code ⟶ no more deadlocks

UNIVERSITY OF CALGARY

Deadlock Example

- if `Account` exposes `.lock()` and `.unlock()` methods instead of raw mutexes, we could write:

```
void transaction(Account a1, Account a2, double amount)
{
   if( a1.id() < a2.id()) {
      a1.lock(); a2.lock();
   } else {
      a2.lock(); a1.lock();
   }

   adjust(a1, -amount);
   adjust(a2,  amount);

   a1.unlock(); a2.unlock();
}
```


UNIVERSITY OF CALGARY

Avoidance

Deadlock Avoidance

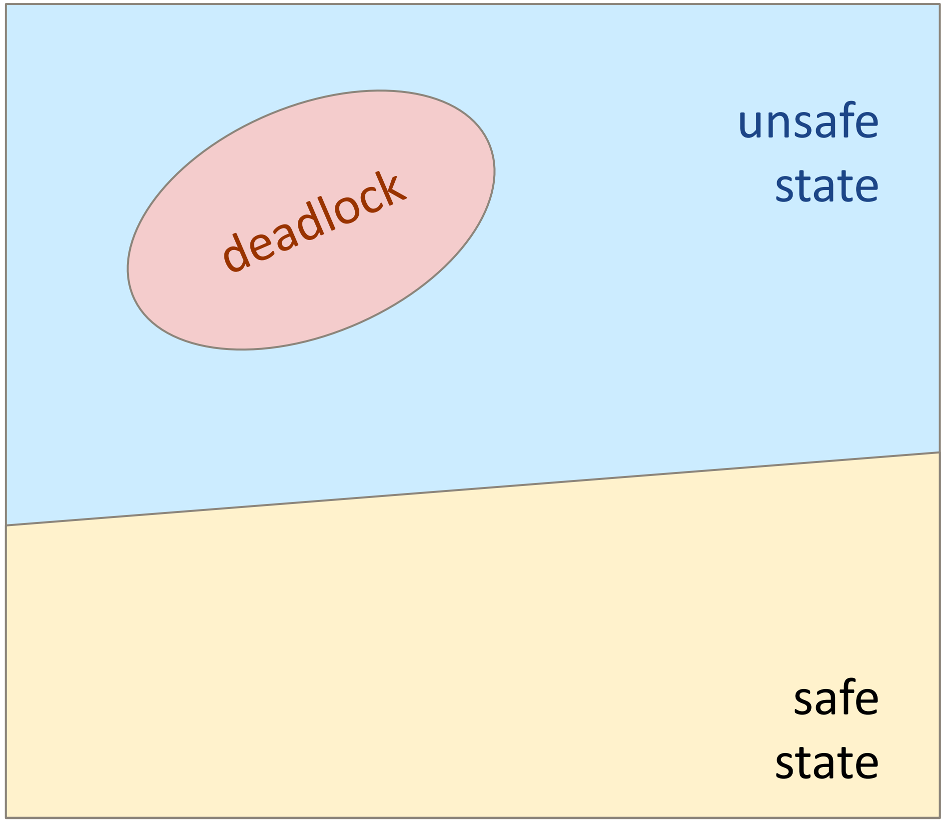
- deadlock prevention schemes can lead to low resource utilization
- **deadlock avoidance** can increase resource utilization if some **a priori information** is available, e.g. each process declares the maximum number of resources of each type that it may need
- a deadlock-avoidance algorithm dynamically examines the resource-allocation **state** to ensure that there can never be a circular-wait condition
- the state is defined by:
  1. the number of available resources
  2. already allocated resources, and
  3. the maximum demands of the processes (the a priori information)

Safe State

- a system is in a **safe state** if there exists a sequence <P_1, P_2, …, P_n> of all running processes in the system where they can all finish, while allowing them to claim their maximum resources
  - i.e. the processes can finish even under the **worst case** scenario — where every process requests its maximum declared resources as its next step
- a system is in an **unsafe state** if there does not exist such an execution sequence
- when a process requests an available resource, the system determines if granting such request would lead to a safe state
  - if new state is safe, request is granted
  - if new state is not safe, request is denied and process waits
    - i.e. a process may be blocked even if it requests a resource that is currently available

Safe, Unsafe, Deadlock State

- if a system is in a safe state → deadlocks are not possible (because they will be avoided by the system)
- if a system is in an unsafe state → deadlocks are possible (but not guaranteed)
- avoidance algorithm ensures that a system never enters an unsafe state, by rejecting/blocking some requests even if resources are available



Deadlock Avoidance Algorithms

- for single instance per resource type
  - we use a **resource-allocation graph algorithm**
  - i.e. we'll assume worst case scenario, create a graph and look for cycles in this graph

- for multiple instances per resource type
  - we use the **banker's algorithm**
  - not covered in this course

Resource-Allocation Graph Algorithm

Resource-Allocation Graph Algorithm

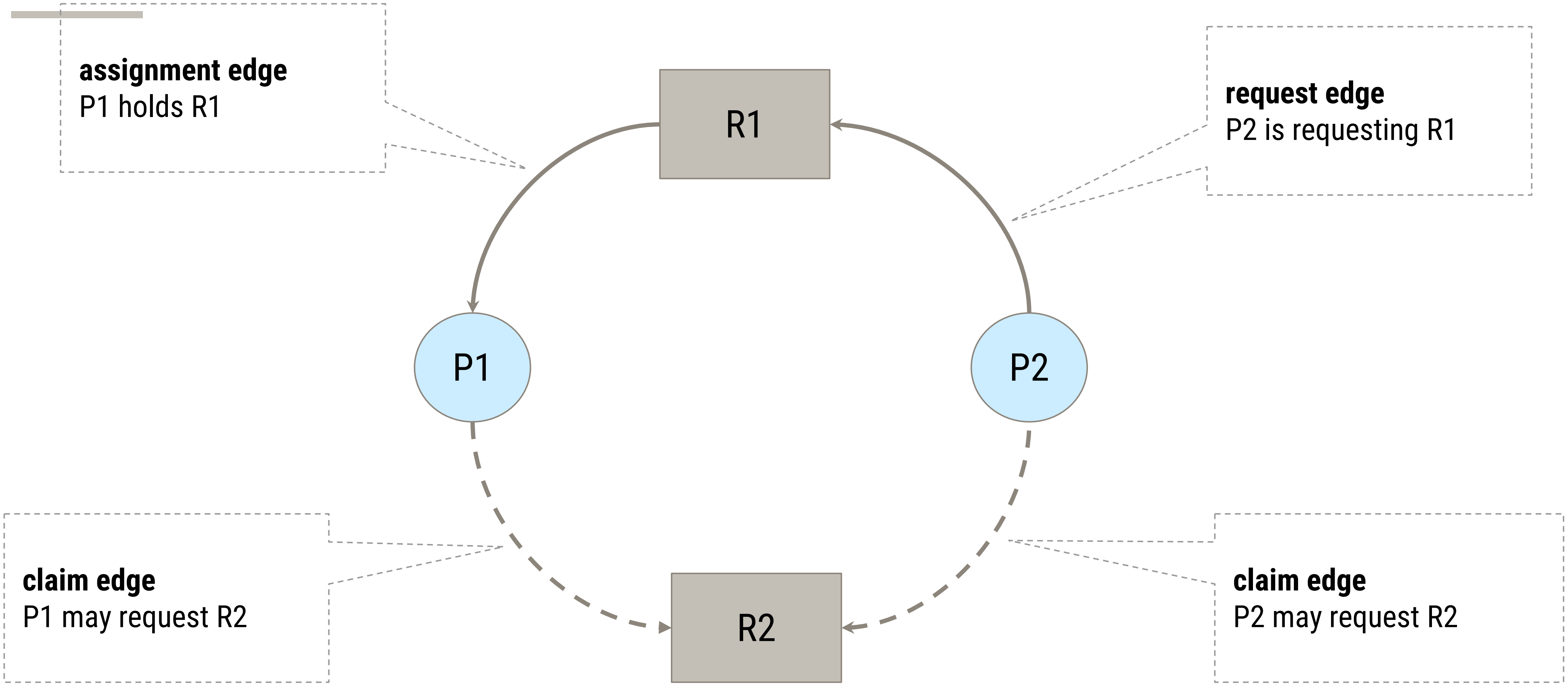
- **claim edge** $P_i \rightarrow R_j$ indicates that process P_j **may** request resource R_j
  - represented by a dashed line
  - this is the a priori knowledge

- claim edge converts to **request edge** when a process actually requests a resource
  - represented by a solid line

- request edge converts to an **assignment edge** when the resource is allocated to the process
  - represented by a solid line, reversed direction

- when a resource is released by a process, assignment edge reconverts to a claim edge
  - reverse direction & becomes dashed
- resources must be claimed **a priori** in the system

Resource-Allocation Graph



Resource-Allocation Graph

P_2 may request R_2

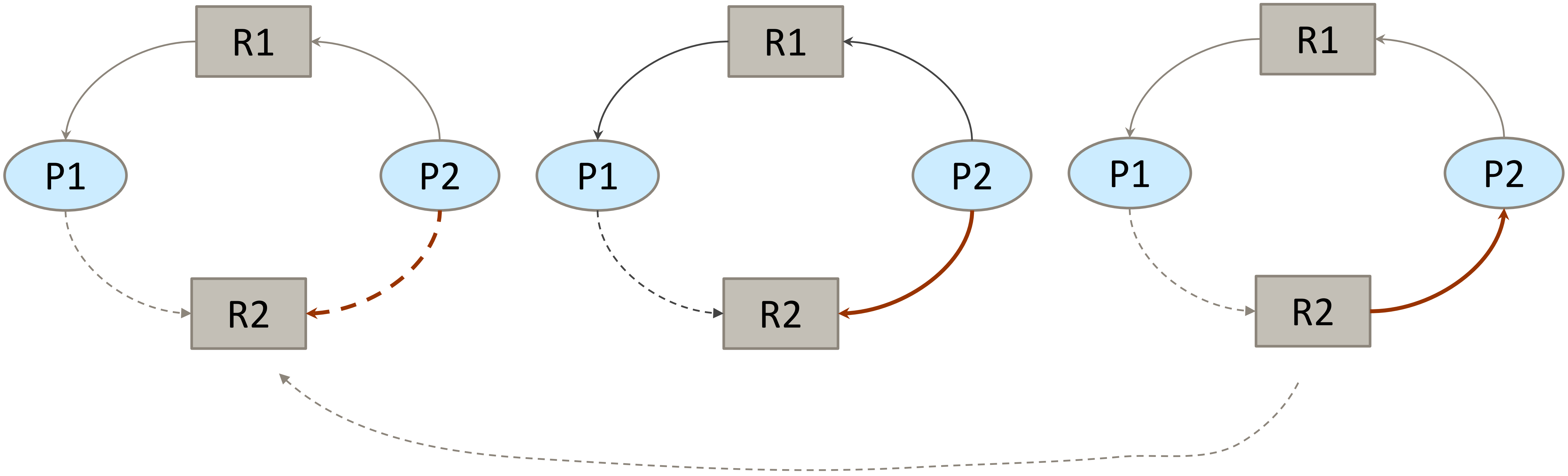
represented as claim edge

P_2 actually requests R_2

claim edge converts to request edge

P_2 holds R_2

claim edge converts to assignment edge

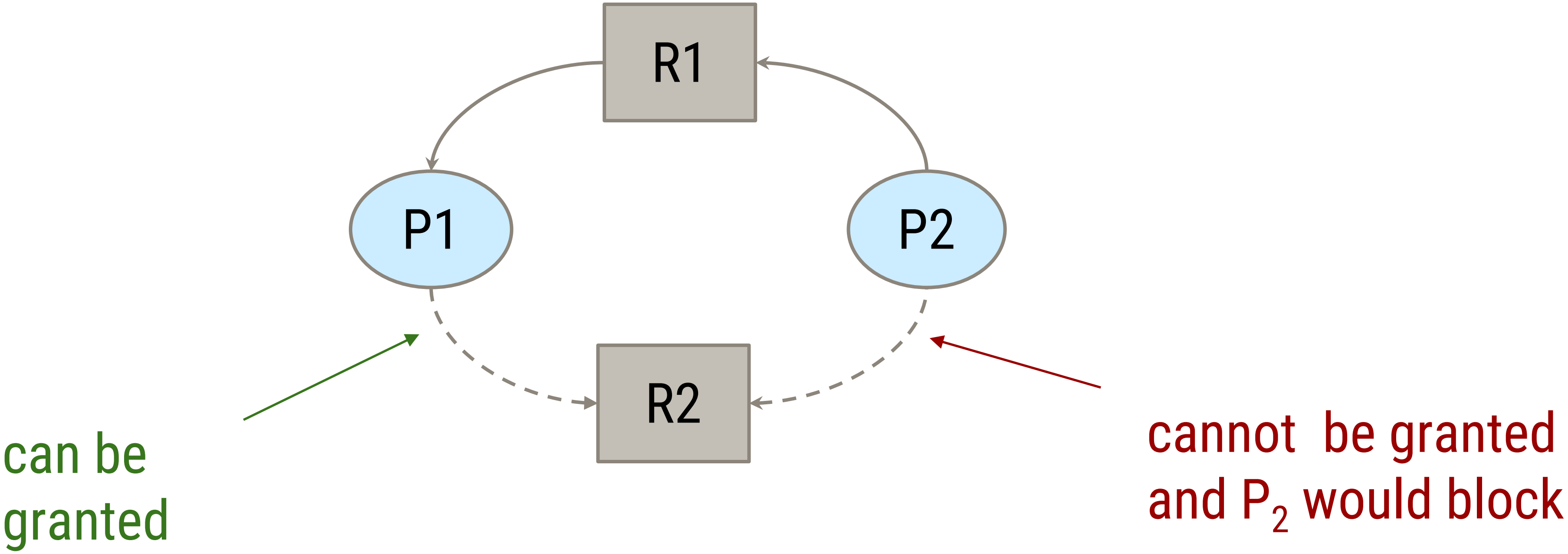


P_2 releases R_2

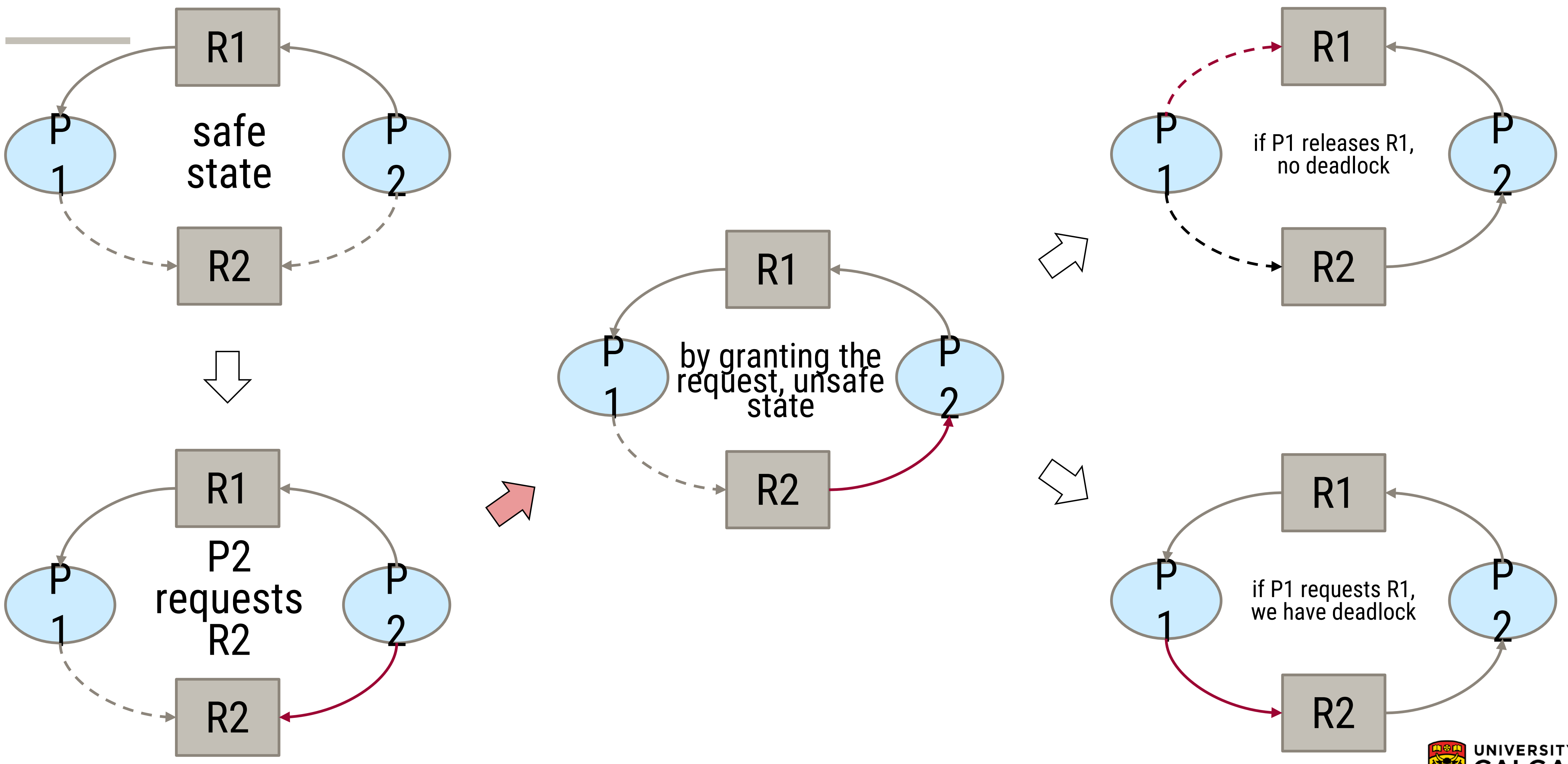
assignment edge converts back to claim edge

Resource-Allocation Graph Algorithm

- suppose that process requests a resource
- we decide whether to grant it by assuming the worst-case scenario
  - the request can be granted only if allowing such request will not violate **safe state**
  - we make sure that converting the request edge to an assignment edge does not result in **formation of a cycle**
- complexity: same as cycle-detection algorithm in a directed graph, i.e. O(|V| + |E|) e.g. using topological sort algorithm



Unsafe state could lead to deadlock



Banker’s Algorithm

Banker's Algorithm

- another avoidance algorithm
- more general than resource-allocation graph algorithm
- works with multiple instances per resource type
- even slower than the graph algorithm
  - banker's: $O(|processes|^2 * |resources|)$
  - graph: $O(|V| + |E|)$

Detection

Deadlock Detection

- we allow the system to enter a deadlock state
- but **eventually** we **detect** the deadlock and **recover** from it

- motivation:
  - prevention leads to non-optimal resource utilization/starvation
  - avoidance is expensive, and still non-optimal resource utilization
  - deadlocks are not that common to begin with

- detection algorithm – tells us which processes are involved in a deadlock, if any
  - with single instance per resource type
  - multiple instances per resource type

- recovery scheme

Deadlock detection with single instance per resource type

- if there is a cycle in resource allocation graph, then there is a deadlock
  - the graph could be big...
- periodically invoke an algorithm that searches for a cycle in the graph
- if there is a cycle, there exists a deadlock

Deadlock detection with multiple instances per resource type

- similar to banker's algorithm
- we try to determine if a sequence exists in which all running processes can **finish** executing
- we assume **best case scenario** - a process that is given its requested resources will finish without asking for more resources, and then releases all its resources
- Algorithm requires an order of $O(m * n^2)$ operations to detect whether the system is in deadlocked state.

UNIVERSITY OF CALGARY

Detection-Algorithms - when and how often?

- detection algorithms are quite expensive, $O(n^2)$ or even $O(n^3)$
- we probably cannot invoke them on every resource request
- other ideas for invoking detection:
  - check every few minutes in a background task
  - check when CPU goes idle (or drops below certain utilization?)
- when, and how often depends on:
  - how often a deadlock is likely to occur?
  - how many processes will be affected?
    - one for each disjoint cycle
- if we check too often - we spend too many CPU cycles on useless work
- if we don't check often enough - there may be many cycles in the resource graph and we would not be able to tell which of the many deadlocked processes “caused” the deadlock

Recovery

Deadlock recovery

1. process termination
2. process rollback
3. resource preemption

Recovery from deadlock: Process Termination

- we could abort all deadlocked processes
  - simple, but rarely necessary

- better solution is to abort one process at a time until the deadlock is eliminated
- some ideas for the order in which we abort processes:
  - priority of the process
  - age of the process
  - how much longer to completion
  - resources the process has used
  - resources process needs to complete
  - how many processes will need to be terminated
  - is process interactive or batch

Recovery from deadlock: Process Rollback

- more gentle than process termination
  - programs can be implemented to cooperate with termination
  - a program can periodically or on demand save its current state (**checkpoint**)
  - when restarted, the program detects a checkpoint and resumes computation from last checkpoint (**rollback**)
  - programs can then checkpoint themselves just before requesting resources, or inside signal handlers

- when deadlock is detected, a program can be terminated and re-scheduled to run later
  - e.g. after the other affected deadlocked processes are done

- does not work well with all resource types (e.g. printer)

- useful for long running computations / simulations

Recovery from deadlock: Resource Preemption

- similar idea to rollback, but instead of checkpointing the program, we checkpoint the resources of the program

- when deadlock occurs:
  - pick a victim process
  - suspend victim process
  - save state of victim's resources
  - give victim's resources to other deadlocked processes
  - when the other processes release the resources, restore resource states
  - return resources to the victim
  - unsuspend the victim process

- obviously, this only works with some resource types

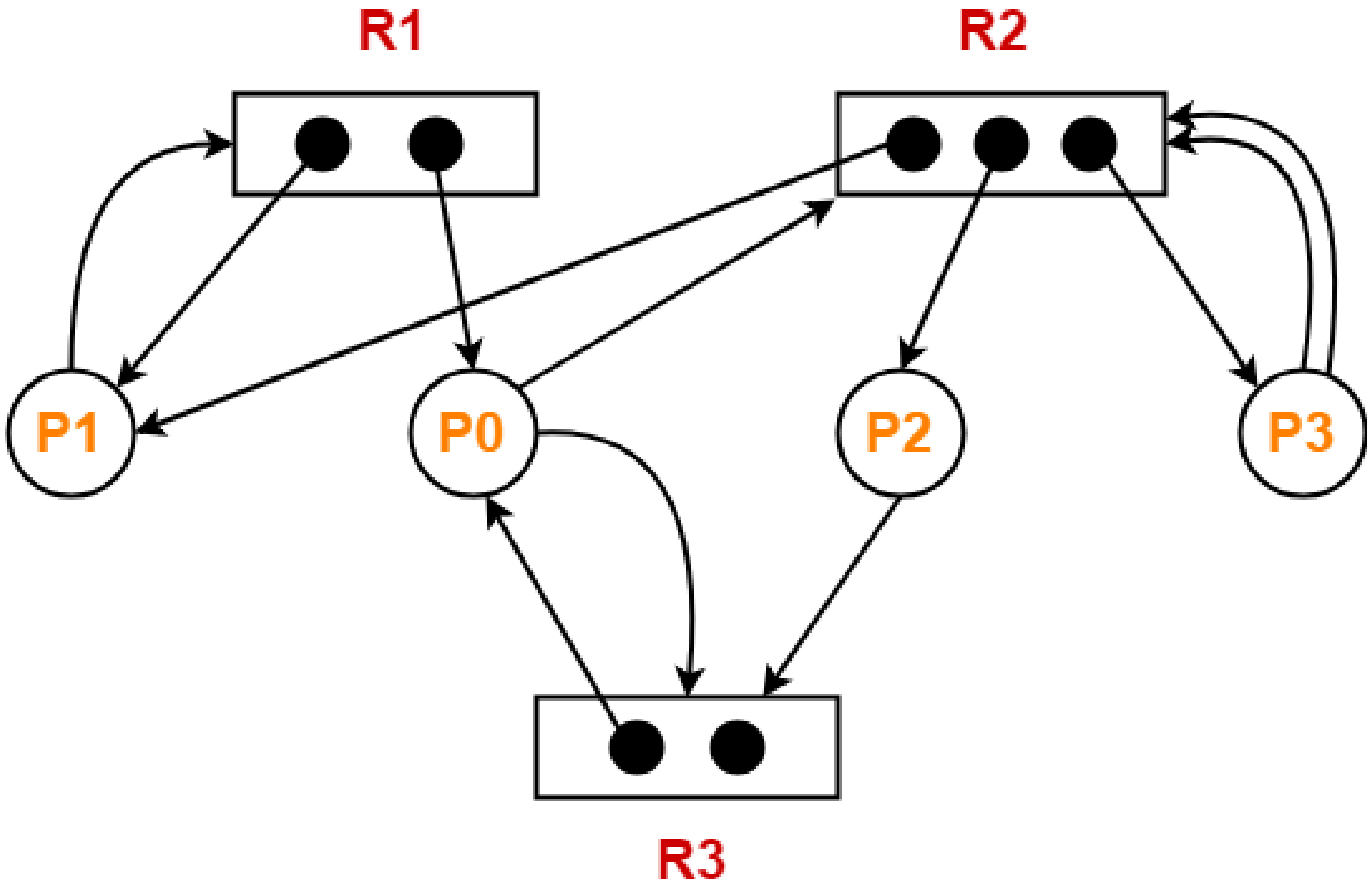
- quite complicated to implement

UNIVERSITY OF CALGARY

Recovery from deadlock

- starvation can be a problem with rollback & checkpointing
  - we might continually pick the same process to preempt/checkpoint
  - possible solution: keep count of preemptions/checkpoints
  - when picking the next process, take this count into consideration

Deadlock?



https://www.gatevidyalay.com/resource-allocation-graph-deadlock-detection/

Dining Philosophers

2 dining philosophers without deadlock avoidance

```
Thread 1:
  P1 requests F1
  P1 requests F2

Thread 2:
  P2 requests F2
  P2 requests F1
```

Sequence 1:

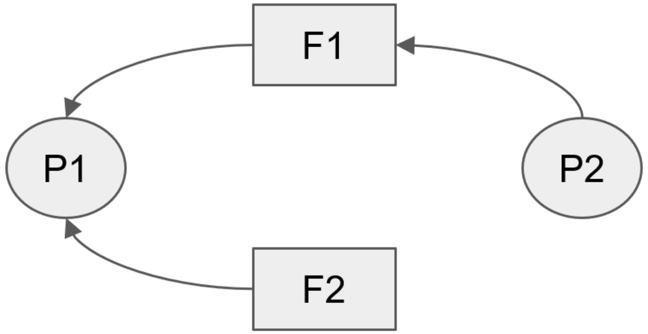
P1 requests F1
P1 requests F2
P2 requests F2
P2 requests F1

No deadlock

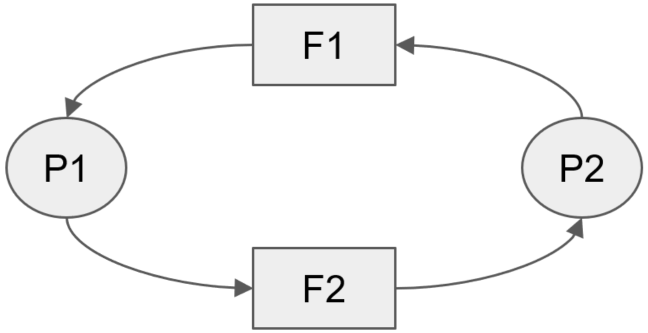
Sequence 2:

P1 requests F1
P2 requests F2
P1 requests F2
P2 requests F1

Deadlock !!!



no cycle



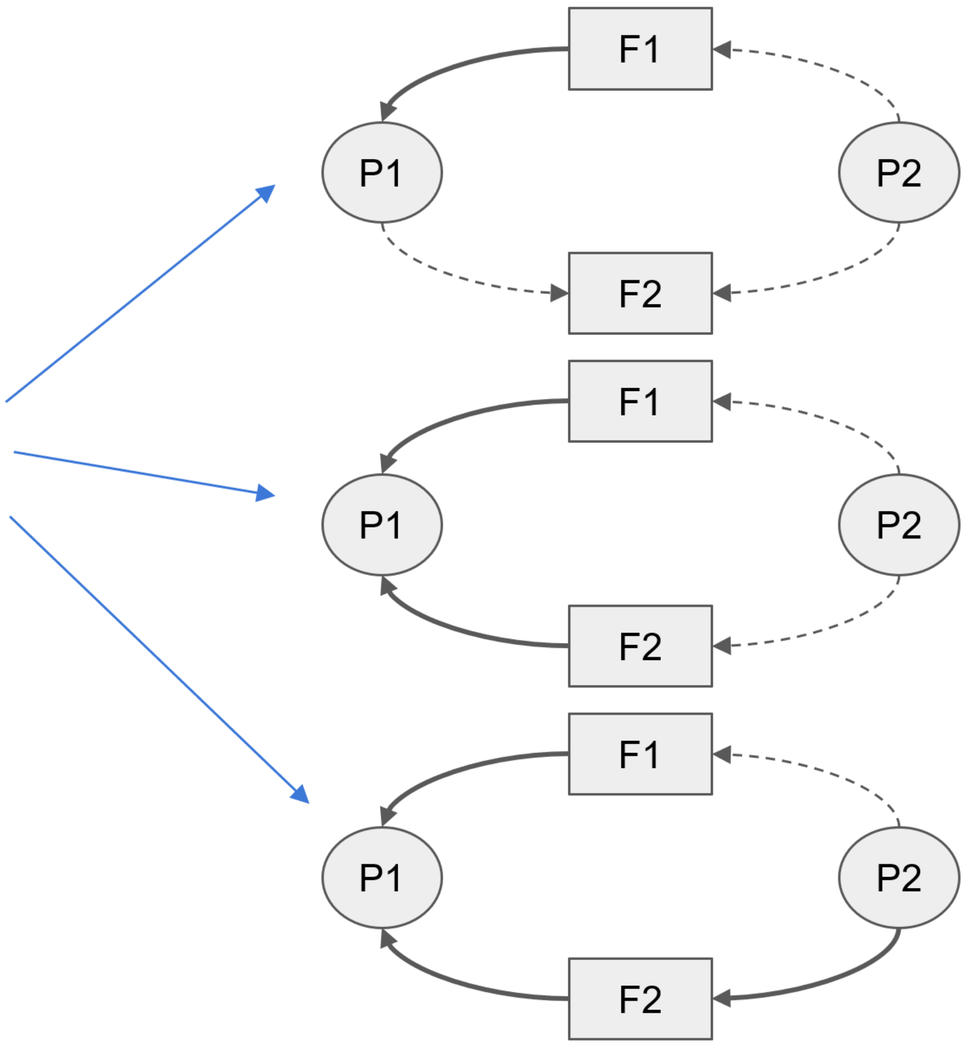
cycle

2 dining philosophers with deadlock avoidance

Sequence 1:

P1 requests F1
P1 requests F2
P2 requests F2 (blocked)
P2 requests F1

No deadlock



- system gives F1 to P1
- P1 continues to execute

- system gives F2 to P1
- P1 continues to execute

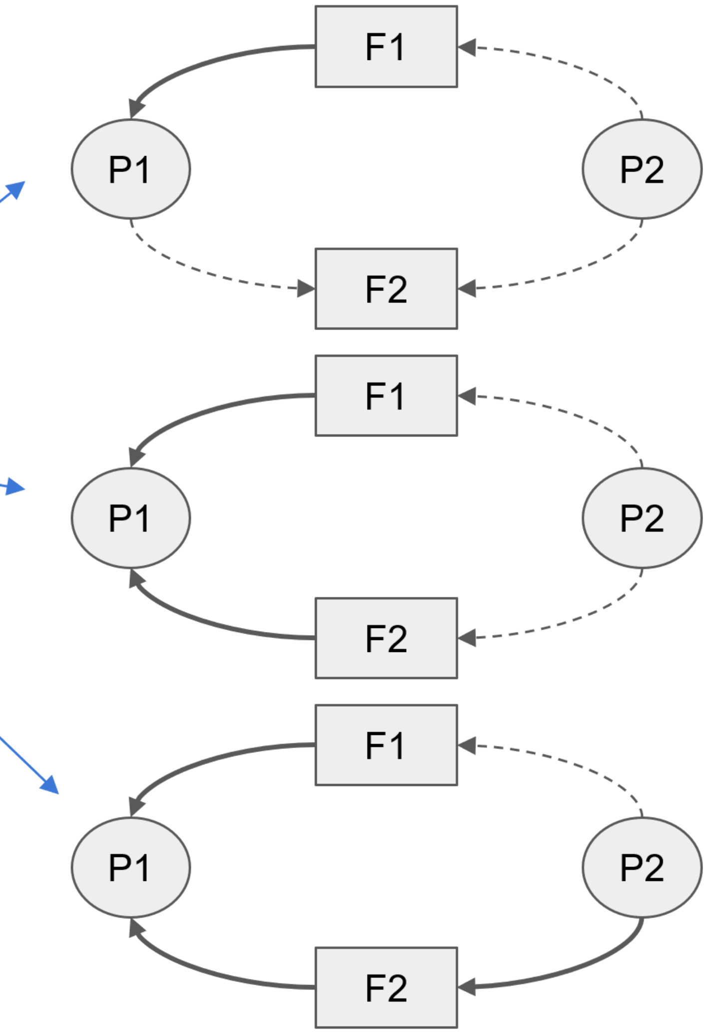
- F2 not available
- P2 is blocked
- P1 eventually finishes and releases F1 and F2 ...

2 dining philosophers with deadlock avoidance

Sequence 1:

P1 requests F1
P1 requests F2
P2 requests F2 (blocked)
P2 requests F1

No deadlock



- system gives F1 to P1
- P1 continues to execute

- system gives F2 to P1
- P1 continues to execute

- F2 not available
- P2 is blocked
- P1 eventually finishes and releases F1 and F2 ...

Topological sort

Topological sort

- a topological sort (toposort), labels vertices of a directed acyclic graph (DAG) from 1 to |V| such that if there is path from vertex i to vertex j, then label of vertex i < label of vertex j
- in other words, the result of a toposort is an ordering of all vertices in such a way, that if there is an edge from A to B, then A will be listed before B in the final ordering
- common use of toposort is to sort tasks based on their dependencies, e.g. taking university courses while satisfying their prerequisites

Topological sort

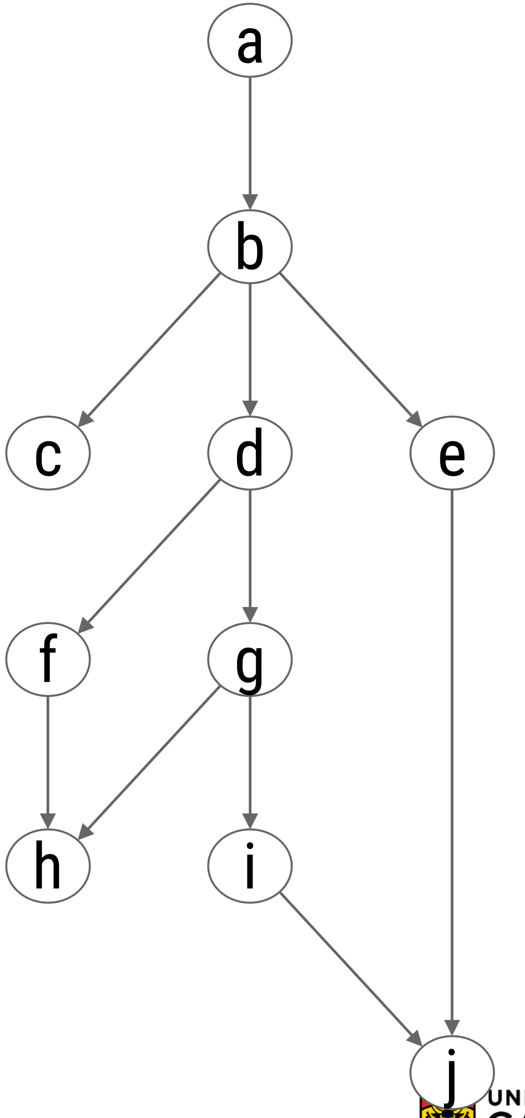
- interesting: if we vertically lay out the vertices in a DAG in topological order, all edges will point downwards
- a topological order may not be unique

  e.g. for the graph on the right, both

  {a,b,d,g,i,f,h,e,j,c} and {a,b,c,d,g,i,e,j,f,h}

  are valid topological orders
- if a graph contains a cycle, topological order does not exist, i.e.

  toposort will fail to finish



Toposort in english

- repeat:
  - find task that can be completed now (it does not depend on anything)
  - if no such task exists, exit loop
  - otherwise print & remove this task

- if we removed all tasks, we successfully finished toposort
  - we printed tasks in topological order

- otherwise, there must be a cycle
  - all remaining (unremoved) tasks are waiting for at least one task

Toposort algorithm

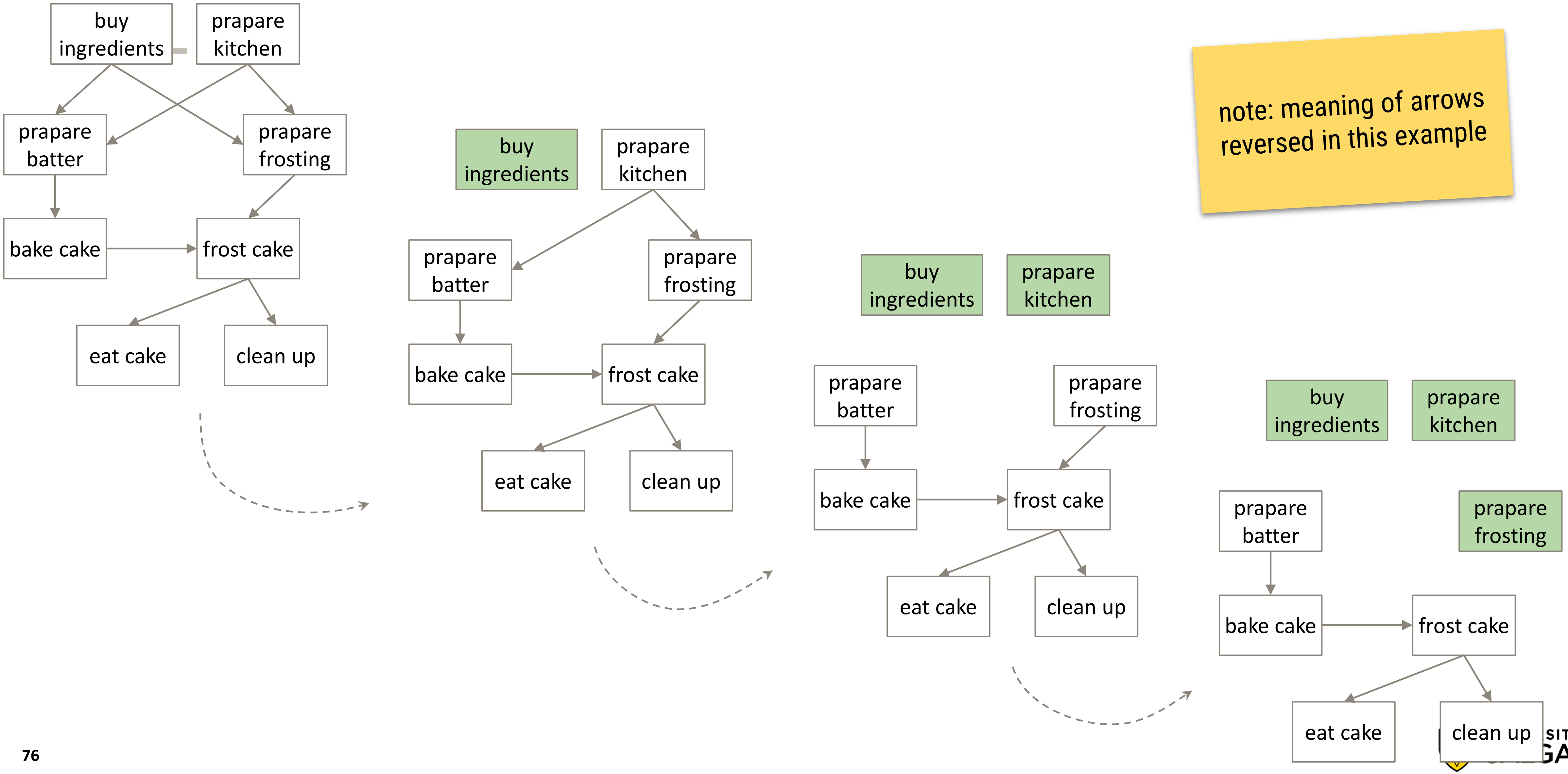
- repeat the following steps
  - find a node **n** in the graph that has no arrows pointing out from (towards*) it
  - if there is no such node, break loop
  - add node **n** to the result (e.g. linked list)
  - remove any edges from the graph that end (start*) at node **n**,

    this is enough to simulate removal of **n** from graph

- if the result contains all vertices of the grap, the result represents the topological order

- otherwise return an error - indicating there must be a cycle in the graph
  - any remaining nodes in graph are directly or indirectly part of the cycle

Toposort illustration

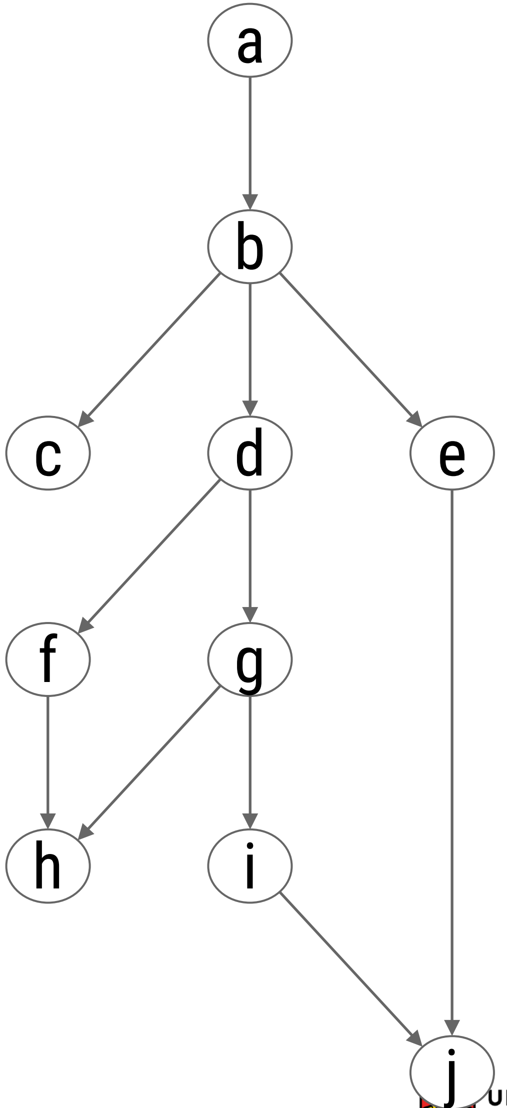


note: meaning of arrows reversed in this example

Toposort pseudocode

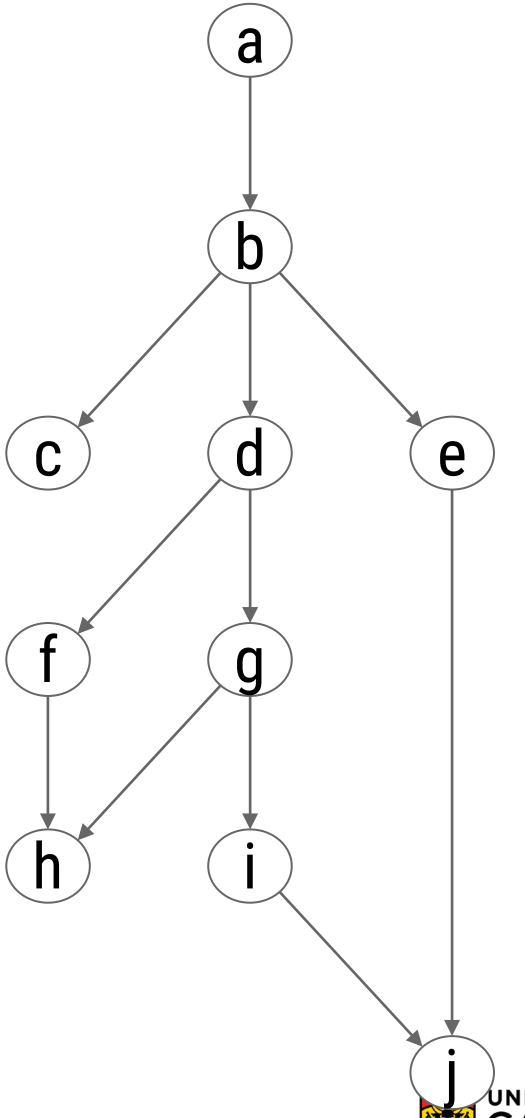
```
g = graph that we wish to topologically sort
result = []
s = stack of all all vertices m such that in-degree(m)=0
while len(s) > 0:
    n = s.pop()
    result.append(n)
    for every edge e in adjacency-list(n):
        remove edge e from the graph g
        if in-degree(m)==0:
            s.insert(m)
if len(res) != number_of_vertices(g):
    print graph contains a cycle
else:
    print res
```

with the right data structures, it is possible to implement so that runtime complexity = O(|V|+|E|)



Graph representation

n	in-degree(n)	adj-list(n)
a	0	b
b	1	c,d,e
c	1	
d	1	f,g
e	1	j
f	1	h
g	1	h,i
h	2	
i	1	j
j	1	



Review

Review

- Define deadlock.
- If there is a deadlock, that means there is a circular wait between processes. True or False
- If there is a circular wait between processes, than means there is a deadlock. True or False
- Which of the following methods is used to prevent circular waiting among processes and resources?
    - Spooling
    - Request all resources at the beginning
    - Take resources away
    - Order resources & lock in order
- How do we detect a deadlock?
- Name three approaches for deadlock recovery.
- What is a checkpoint?

Onward to …
CPU scheduling

Jonathan Hudson
jwhudson@ucalgary.ca
https://pages.cpsc.ucalgary.ca/~jwhudson/

UNIVERSITY OF CALGARY