Universidade Estadual de Campinas
Instituto de Computação
**MC504 Sistemas Operacionais**

# T23a Common Concurrency Problems

*Referência principal*
Ch. 32 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

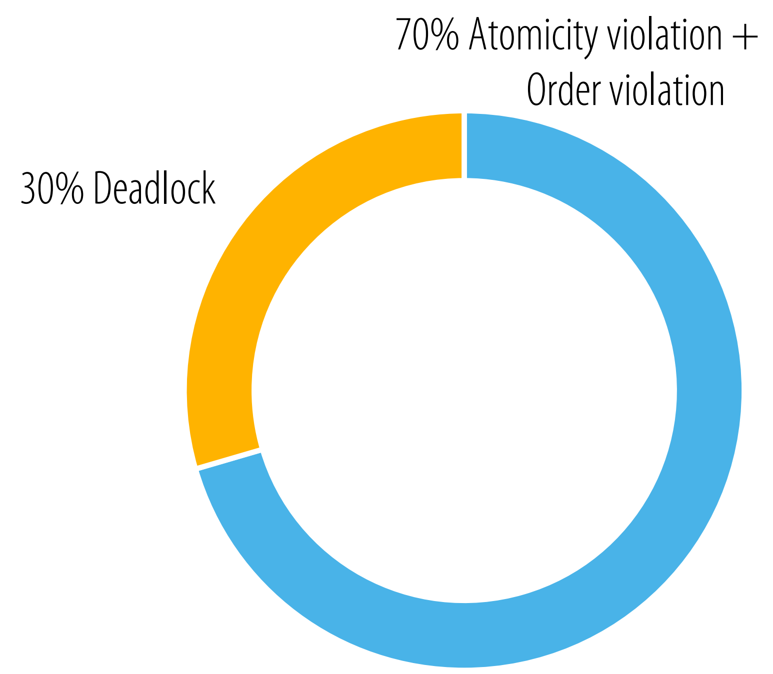
*Discutido em classe em 17 de outubro de 2018*

Arthur João Catto, PhD

2º semestre de 2018

# Common Concurrency Problems

- More recent work focuses on studying other types of common concurrency bugs.
- Non-deadlock bugs make up 70% of concurrency bugs:
  - Atomicity violation
  - Order violation
- Deadlocks account for 30% of all bugs.



# Atomicity-Violation Bugs

- In this sort of error, critical sections may not be executed atomically.
- This is a simple example found in MySQL:

```
Thread1::
    if(thd->proc_info){
        ...
        fputs(thd->proc_info, ...);
        ...
    }


Thread2::
    thd->proc_info = NULL;
```

- What is the problem here?
  - Tread1 and Thread2 may access field `proc_info` in struct `thd` non-atomically .

# Atomicity-Violation Bugs

- How can we solve that problem?
  - In this case it is rather straightforward: just protect the critical sections with locks.

```
mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

Thread1::
    mutex_lock(&lock);
    if(thd->proc_info) {
        ...
        fputs(thd->proc_info, ...);
        ...
    }
    mutex_unlock(&lock);

Thread2::
    mutex_lock(&lock);
    thd->proc_info = NULL;
    mutex_unlock(&lock);
```

# Order-Violation Bugs

- In this sort of error, instructions in different threads may be executed out of the expected order.

```
Thread1::
     void init(){
          mThread = PR_CreateThread(mMain, …);
     }


Thread2::
     void mMain(…){
          mState = mThread->State
     }
```

- What is the problem here?
  - The code in Thread2 seems to assume that the variable `mThread` has already been initialized (and is not `NULL`), but this may not be true.

# Order-Violation Bugs

- How can we solve that problem?
  - In this case, we can use a condition variable to enforce order of execution.

```
mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
cond_t mtCond  = PTHREAD_COND_INITIALIZER;
int mtInit = 0;


Thread 1::
void init(){
   ...
   mThread = PR_CreateThread(mMain,…);

   // signal that the thread has been created.
   mutex_lock(&mtLock);
   mtInit = 1;
   cond_signal(&mtCond);
   mutex_unlock(&mtLock);
   ...
}
```

```
Thread2::
void mMain(…){
   ...
   // wait for the thread to be initialized …
   mutex_lock(&mtLock);
   while(mtInit == 0)
      cond_wait(&mtCond, &mtLock);
   mutex_unlock(&mtLock);


   mState = mThread->State;
   ...
}
```

# Deadlocks

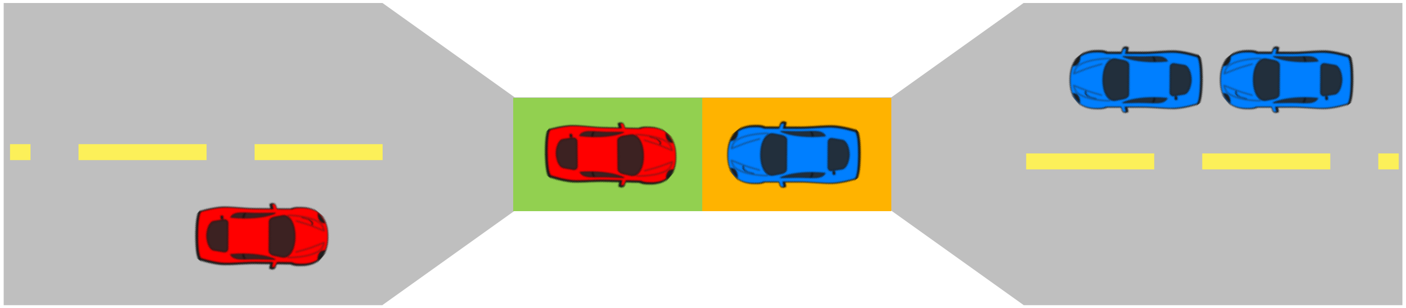
# Deadlocks…

- Are permanent blockings of two or more processes or threads that either
  - compete for system resources or
  - communicate with each other.
- Involve conflicting needs for resources by two or more processes or threads.
- Have no efficient solution.

# Deadlock Definition

- Resource
  - Any (passive) thing needed by a thread to do its job
    - CPU, disk space, memory, lock
  - Preemptable
    - can be taken away by OS
  - Non-preemptable
    - must stay with thread
- Starvation
  - Thread waits indefinitely
- Deadlock
  - Circular waiting for resources
  - Deadlock $\Longrightarrow$ starvation, but not vice-versa

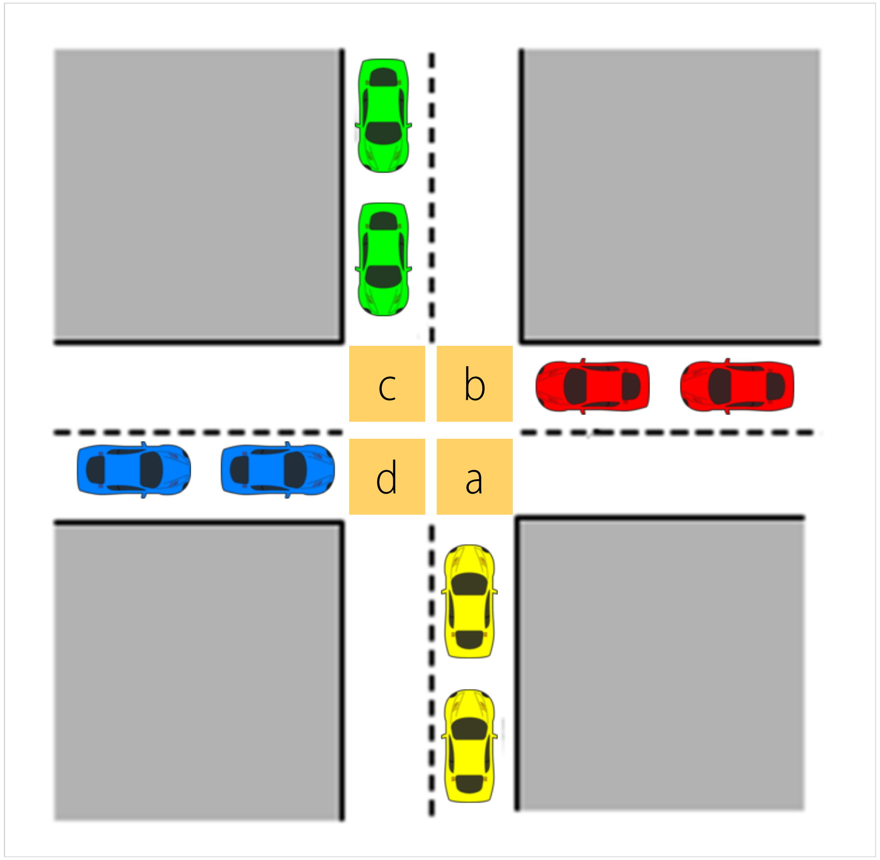
# Bridge Crossing



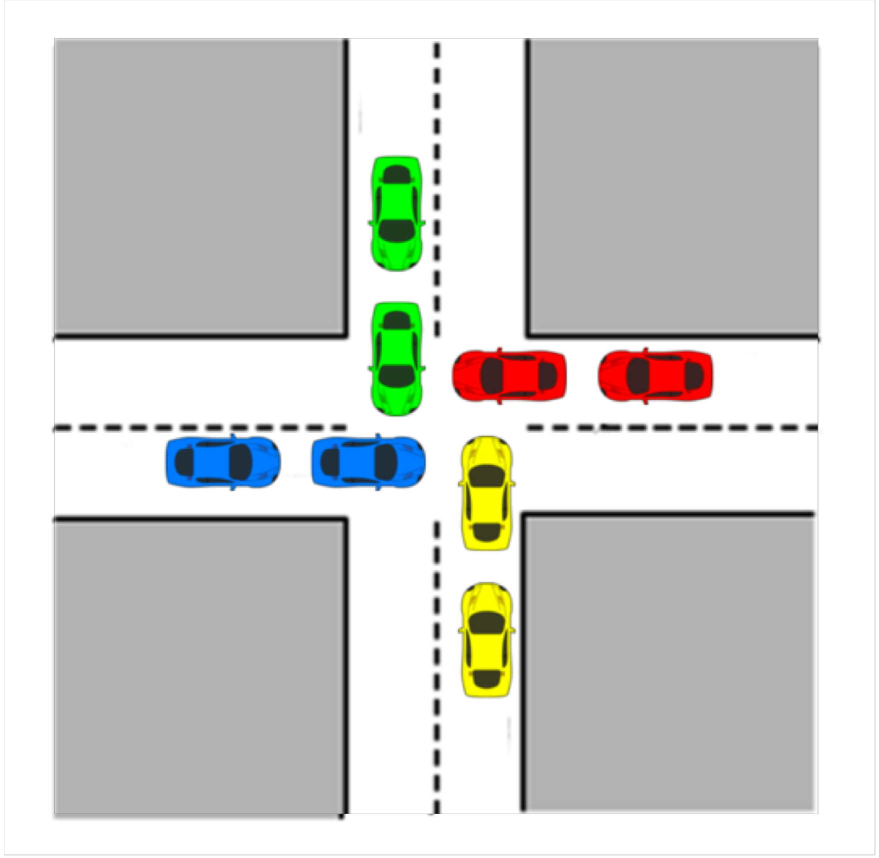
- Bridge with a single lane of traffic.
- Each section of the bridge can be viewed as a resource.
- If a deadlock occurs, it can be resolved if one car backs up (preempt resources and rollback).
- Several cars may have to be backed up if a deadlock occurs.
- Starvation is possible.

Deadlock Example

# Crossroads




Deadlock is possible



Deadlock has occurred

Deadlock Example

# Two locks

| Thread A | Thread B |
| --- | --- |
| `lock1.acquire();` | `lock2.acquire();` |
| `lock2.acquire();` | `lock1.acquire();` |
| `lock2.release();` | `lock1.release();` |
| `lock1.release();` | `lock2.release();` |

- Can deadlock happen?
- Will deadlock happen?

Deadlock Example

# Two locks and a condition variable

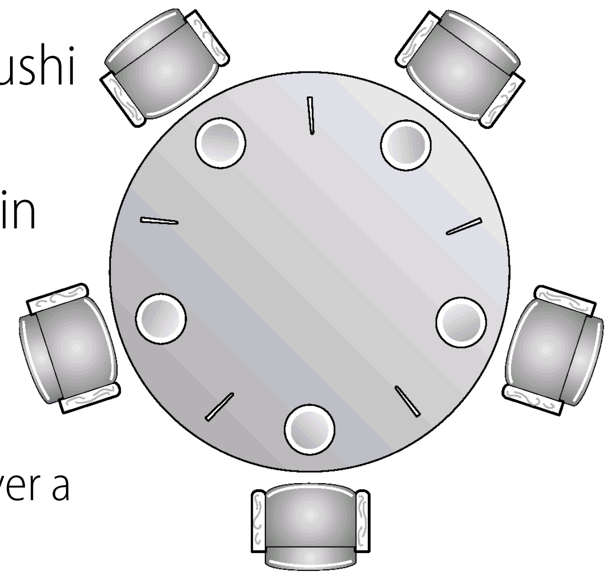
| Thread A | Thread B |
|---|---|
| `lock1.acquire();` | `lock1.acquire();` |
| `...` | `...` |
| `lock2.acquire();` | `lock2.acquire();` |
| `while (needToWait()) {` | `...` |
| `   condition.wait(lock2);` | `condition.signal(lock2);` |
| `}` | `...` |
| `lock2.release();` | `lock2.release();` |
| `...` | `...` |
| `lock1.release();` | `lock1.release();` |

- What happens here?

Deadlock example

# Dining Philosophers

- Problem
  - 5 people sit around a table and alternate between talking and eating from an infinite sushi tray placed at the center of the table.
  - There are 5 plates and 5 chopsticks as shown in the picture.
  - To eat, both chopsticks next to a plate are required.
    - Chopsticks are returned to their positions whenever a person stops or is not able to start eating.



- Can deadlock happen?
  - If so, does it always happen?
- Can a person starve?

# Deadlock can arise only if 4 conditions hold simultaneously

- Mutually exclusive finite resources
  - Resources are finite and can be used by only one process at a time.
- Hold and wait
  - A process is holding at least one resource and is waiting to acquire additional resources held by other processes.
- No preemption
  - A resource can be released only voluntarily by the process that is holding it.
- Circular wait
  - There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that
    - $P_0$ waits for a resource that is held by $P_1$
    - $P_1$ waits for a resource that is held by $P_2$ … and
    - $P_n$ waits for a resource that is held by $P_0$.

# A question about the Dining Philosophers

- How does the Dining Philosophers problem meet the necessary conditions for deadlock?
  - Mutually exclusive finite resources
  - Hold and wait
  - No preemption
  - Circular wait
- Can Dining Philosophers be modified to prevent deadlock?

# Methods for Handling Deadlocks

# Methods for Handling Deadlocks

- Prevention
  - Restrain requests to ensure that at least one necessary condition cannot hold so that the system can never enter a deadlock state.
- Avoidance
  - Delay granting requests that would take the system to a state where a deadlock might occur.
- Detection
  - Allow the system to enter a deadlock state and then recover.
- **Ignore the problem** and pretend that deadlocks never occur
  - Known as the “ostrich algorithm” and used by most operating systems, including UNIX and Windows.

# Deadlock Prevention:
## ensure that at least one necessary condition cannot hold

- Mutual Exclusion
  - Not required for sharable resources.
  - Must hold for non-sharable resources.
- Hold and Wait
  - Prevent a process from requesting a resource if it is already holding any other resources.
  - Require a process to request and receive all the resources it needs before it begins execution or allow process to request resources only when it has none.
  - Low resource utilization; starvation possible.

# Deadlock Prevention:
## ensure that at least one necessary condition cannot hold

- 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 process is waiting.
  - Process 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 process requests resources in an increasing order of enumeration.

# Deadlock Avoidance

- In order to avoid a deadlock, the system must have some additional *a priori* information available.
  - Simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition.
  - Resource-allocation *state* is defined by the number of available and allocated resources, and the maximum demands of the processes.

# System Model

- There are $m$ resource types $R_1, R_2, \ldots R_m$

  *CPU cycles, memory space, I/O devices*

- There are $W_i$ instances of each resource type $R_i$.

- Each process uses resources in the following sequence:
  1. Request
  2. Use
  3. Release

- Requesting and releasing system resources are implemented by *system calls*.

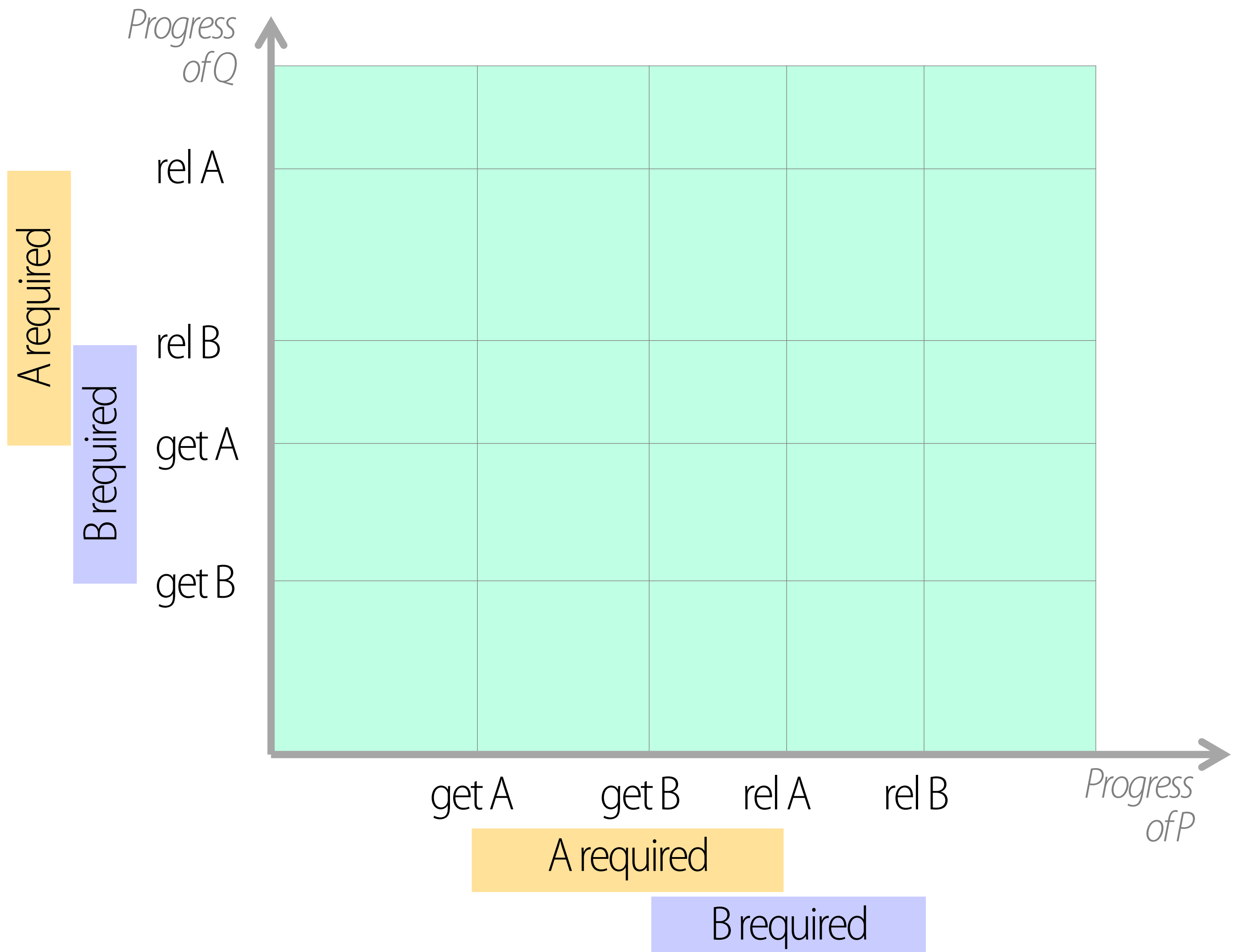
# Analysis of potential for deadlock (2 processes, 2 resources)

**Process P**

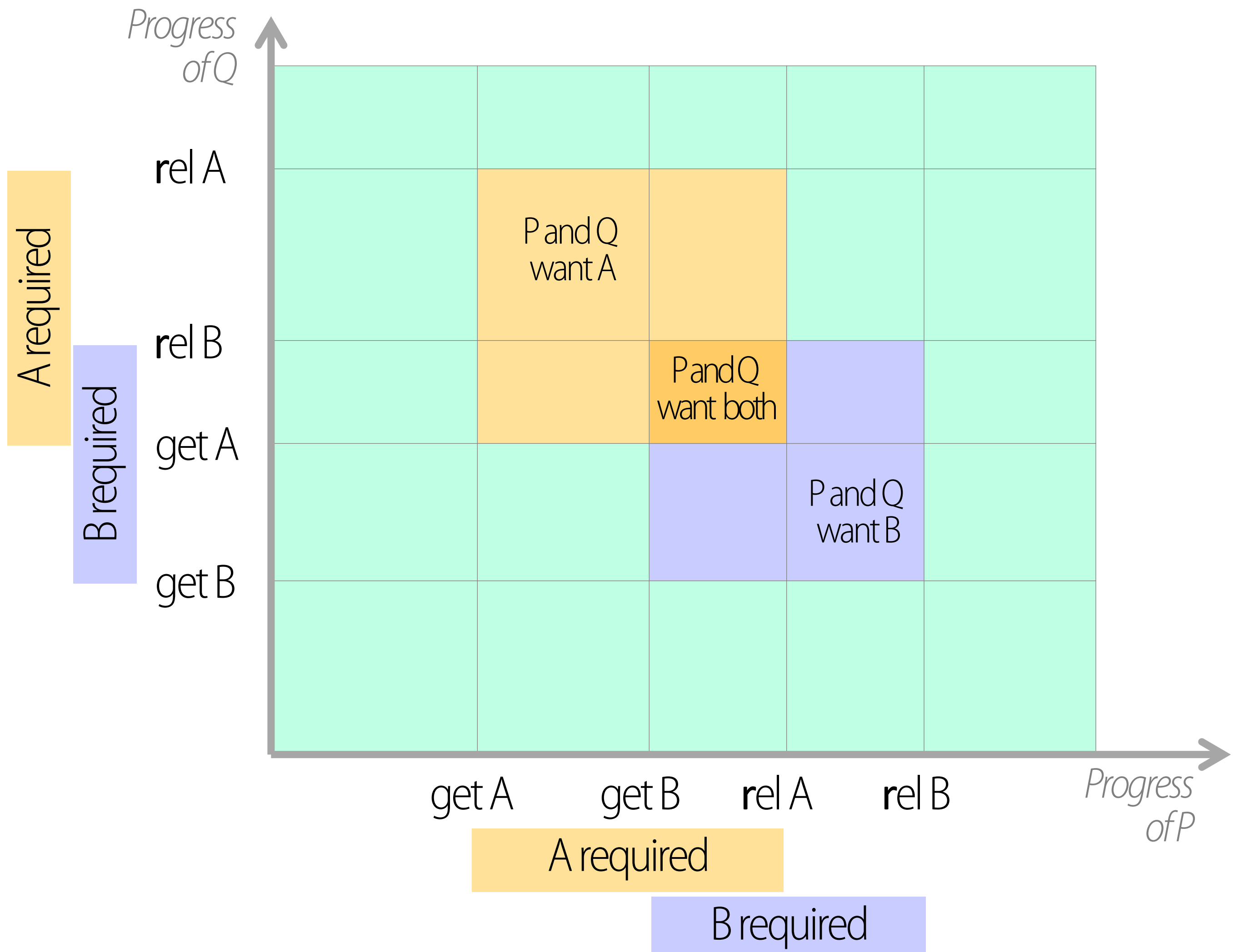
```
… get A;
… get B;
… rel A;
… rel B;
      …
```

**Process Q**

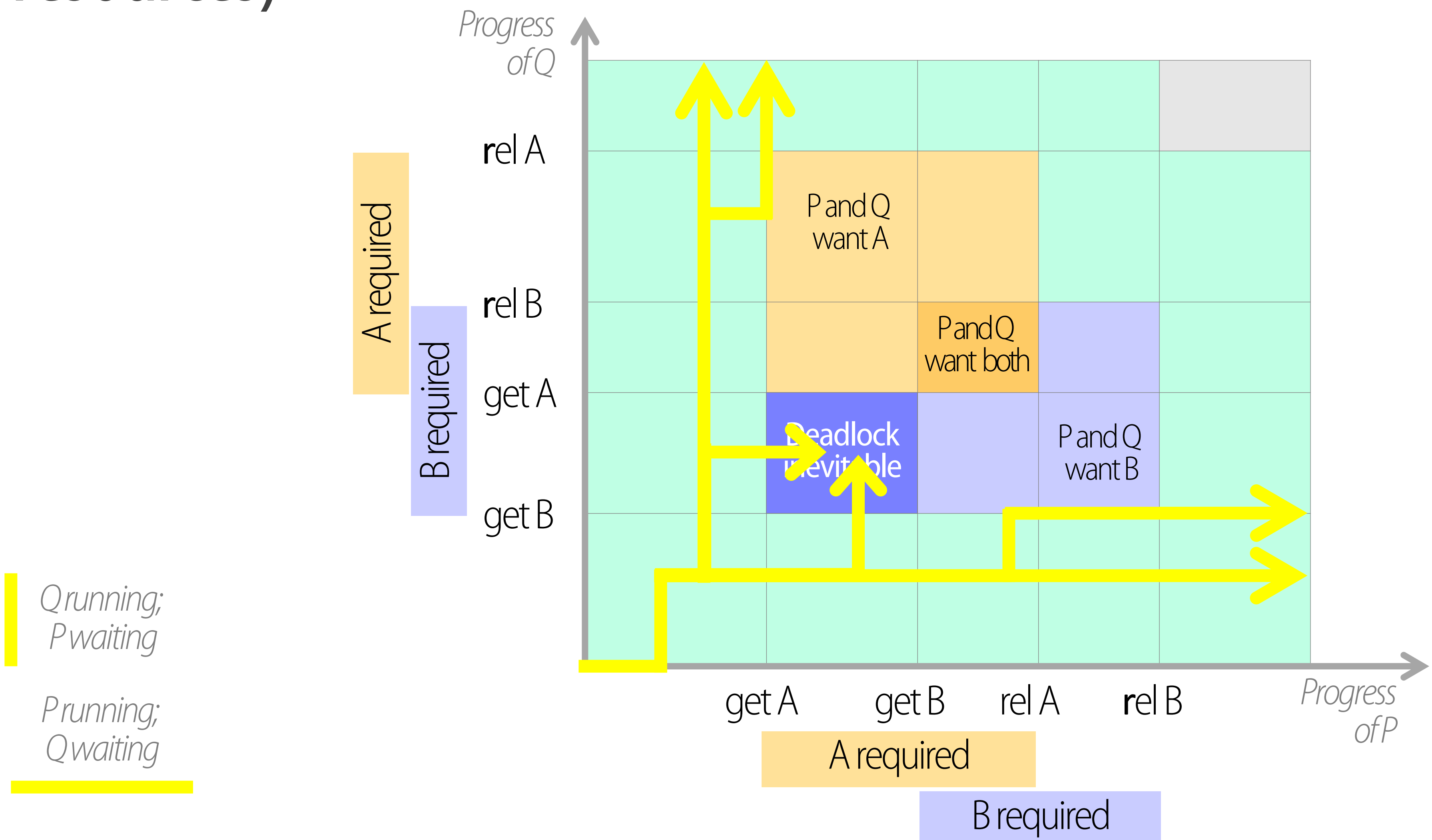
```
… get B;
… get A;
… rel B;
… rel A;
      …
```



# Analysis of potential for deadlock (2 processes, 2 resources)



# Analysis of potential for deadlock (2 processes, 2 resources)



# Deadlock avoidance: the Safe State concept

- When a process requests an available resource, the resource manager must decide if immediate allocation leaves the system in a safe state.

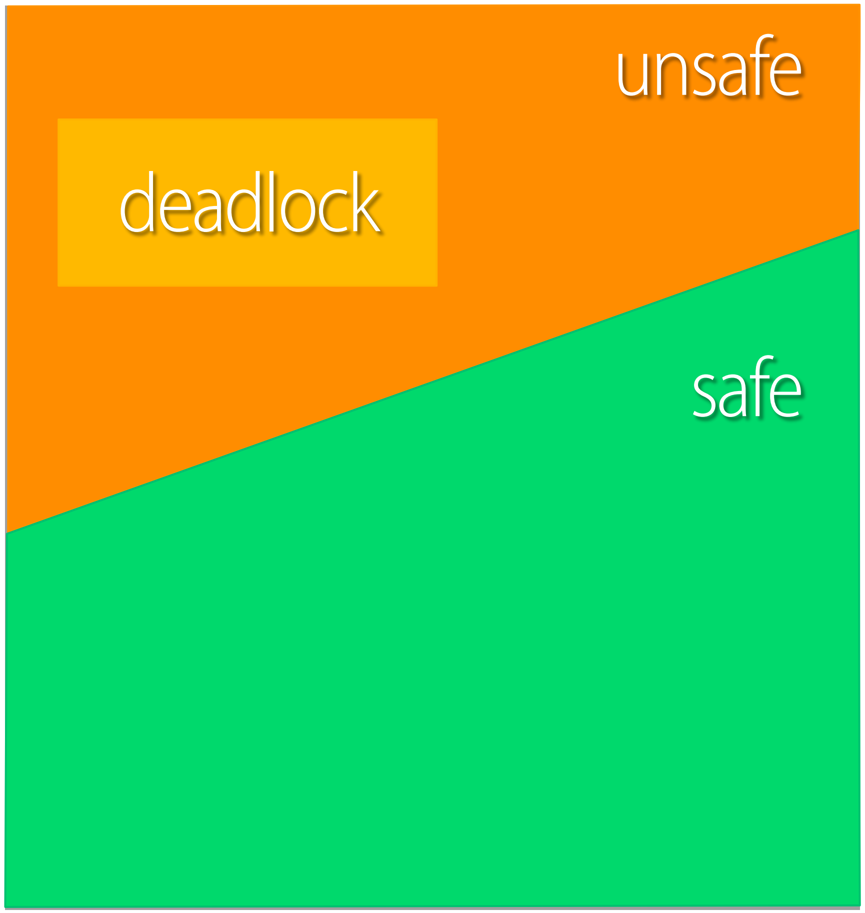
- System is said to be in a ***safe state*** if there exists a sequence of execution $\langle P_{i_1}, P_{i_2}, \ldots P_{i_n} \rangle$ of all the processes in the system such that, for each $P_{i_j}$, the resources that $P_{i_j}$ can still request can be satisfied by currently available resources plus resources held by some $P_{i_k}$, with $k < j$.

# Safe State

- In other words…
  - If $P_{i_j}$ needs a resource that is not immediately available, then $P_{i_j}$ can wait until all $P_{i_{k}}$, $k < j$, have finished.
  - When $P_{i_{k}}$, $k < j$, has finished, $P_{i_j}$ can obtain the resources that it needs, execute, return allocated resources and terminate.
  - When $P_{i_j}$ terminates, $P_{i_{j+1}}$ can obtain the resources that it needs, and so on.

# Basic Facts about Safe and Unsafe States

- System is in a safe state
  - No deadlocks.
- System is in an unsafe state
  - Possibility of deadlock.
- To avoid deadlocks
  - Ensure that the system will never enter an unsafe state.




System states

# Deadlock Dynamics

- Safe state
  - For any possible sequence of future resource requests, it is possible to eventually grant all requests
  - May require waiting even when resources are available!
- Unsafe state
  - Some sequence of resource requests can result in deadlock
- Doomed state
  - All possible computations lead to deadlock

# Food for thought...

- What are the doomed states for Dining Philosophers?
- What are the unsafe states?
- What are the safe states?