# Concurrent Programming – Synchronisation

CISTER Summer Internship 2017

Luís Nogueira

lmn@isep.ipp.pt

# Introduction

- Multitasking
  - Concept of overlapping the computation of a program with another one
  - Central to modern operating systems
- Programming languages explore multitasking by the use of processes, threads or tasks
- Scheduler decides which program to run
  - Common tools: Priority, Time slicing
  - Common goals: Fairness, Response time (low latency), Maximal system utilisation (high throughput), Real-time guarantees, …

# Introduction

- Traditionally, the world parallel is used for systems in which executions of several programs overlap in time by running them on separate processors
- The word concurrent is reserved for potential parallelism, in which executions may, but need not, overlap
- Concurrent programming applies to systems with or without multiple processors
- Parallel programming applies only to systems with multiple processors

# Introduction

- Concurrency – aspect of the problem domain
- Parallelism – aspect of the solution domain
- Both go beyond the traditional sequential model in which things happen one at a time, one after another

# Concurrent programming

- It is difficult to implement safe and efficient synchronisation and communication in concurrent programs

- Correctness for sequential programs
  - Partial correctness – if a program P halts, the answer is "correct"
  - Total correctness – a program P does halt and the answer is "correct"

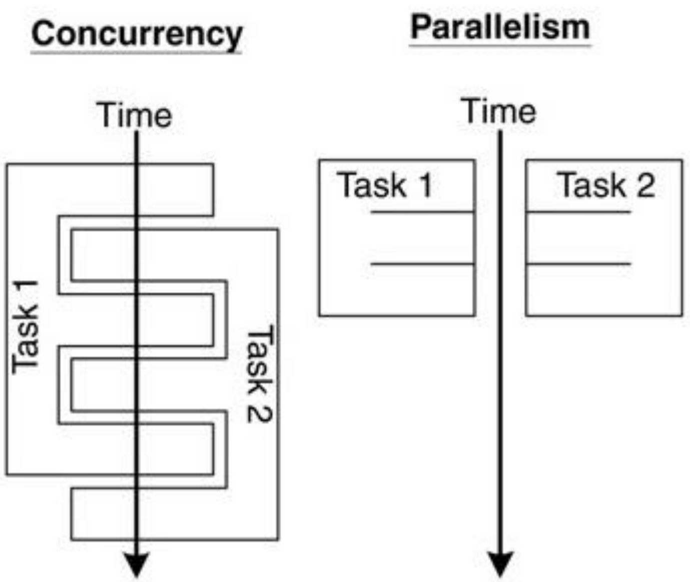
- This deals with correctness of computing a functional result

# Concurrent programming

- Concurrent programs often do not halt
- Correctness of (non-terminating) concurrent programs deal with properties of computation
  - Safety properties – something bad never happens (the program never enters an unacceptable state)
  - Liveness properties – something good eventually happens (the program eventually enters a desirable state)
- Concurrent programs must satisfy the liveness properties without violating the safety properties

# Challenge

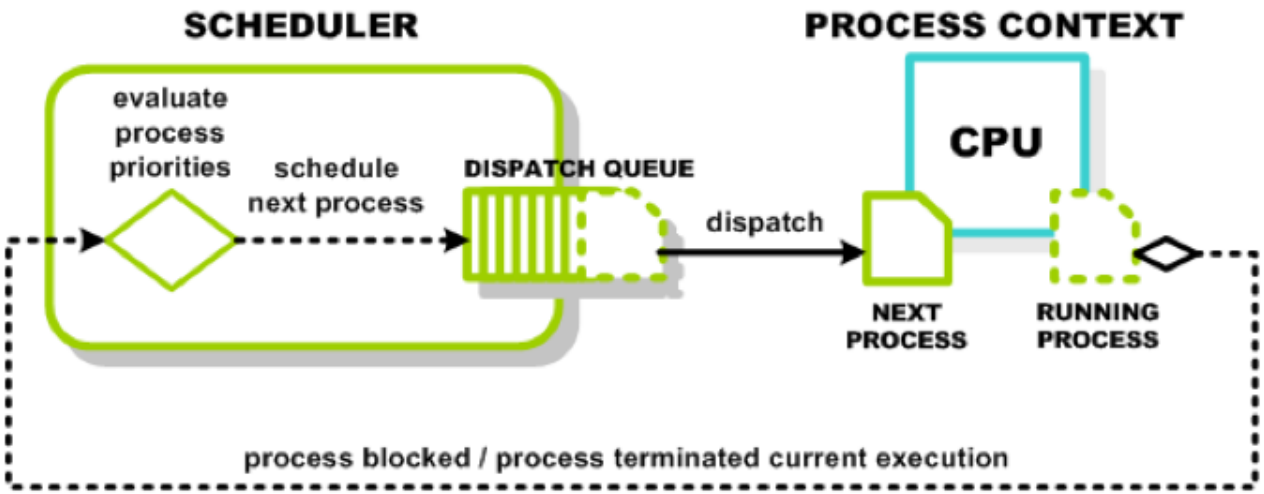
- The need to synchronise the execution of different processes and to enable them to communicate

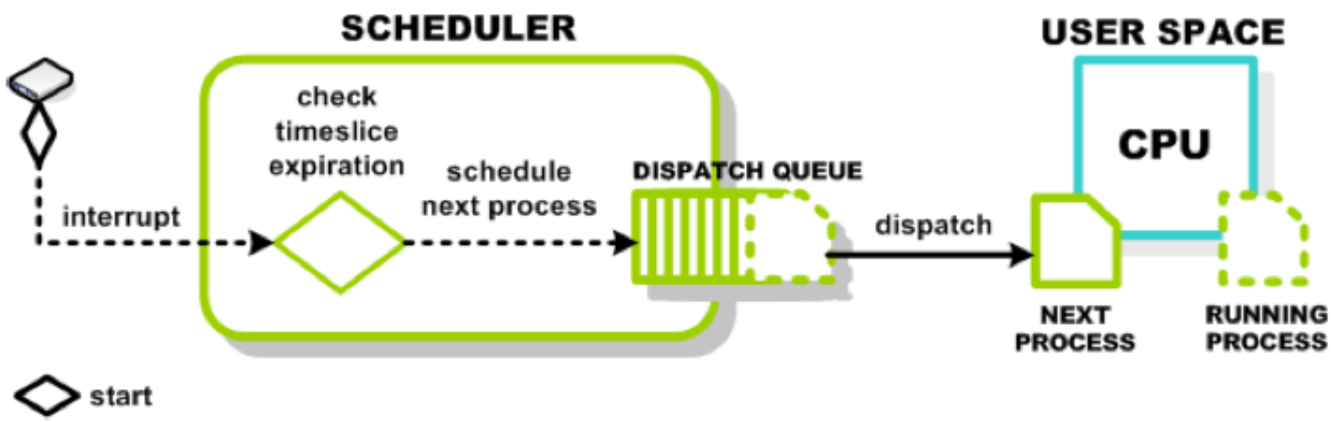


# Preemptive scheduling

- Preemption is the act of temporarily interrupting the current process, without requiring its cooperation, and with the intention of resuming the process at a later time
  - Involves the use of an interrupt mechanism which suspends the currently executing process and invokes the scheduler to determine which process should execute next
- Today, nearly all operating systems support preemptive scheduling
  - This includes the current versions of Windows, Mac OS, Linux, iOS and Android

# When does scheduling happens?



# Priority-based scheduling issues

- Starvation
- Deadlock
- Livelock
- Priority inversion

# Starvation

- Processes with lower priorities may not be given the opportunity to run (or access some other resource)

- A high priority process $P_1$ will always run before a low priority process $P_2$

- If $P_1$ never blocks, $P_2$ will (in some systems) never be scheduled

# Starvation

- Starvation is usually caused by an overly simplistic scheduling algorithm

- A scheduler should allocate resources so that no process perpetually lacks necessary resources
  - Modern scheduling algorithms normally guarantee that all processes will receive a minimum amount of each important resource (most often CPU time)

# Avoiding starvation

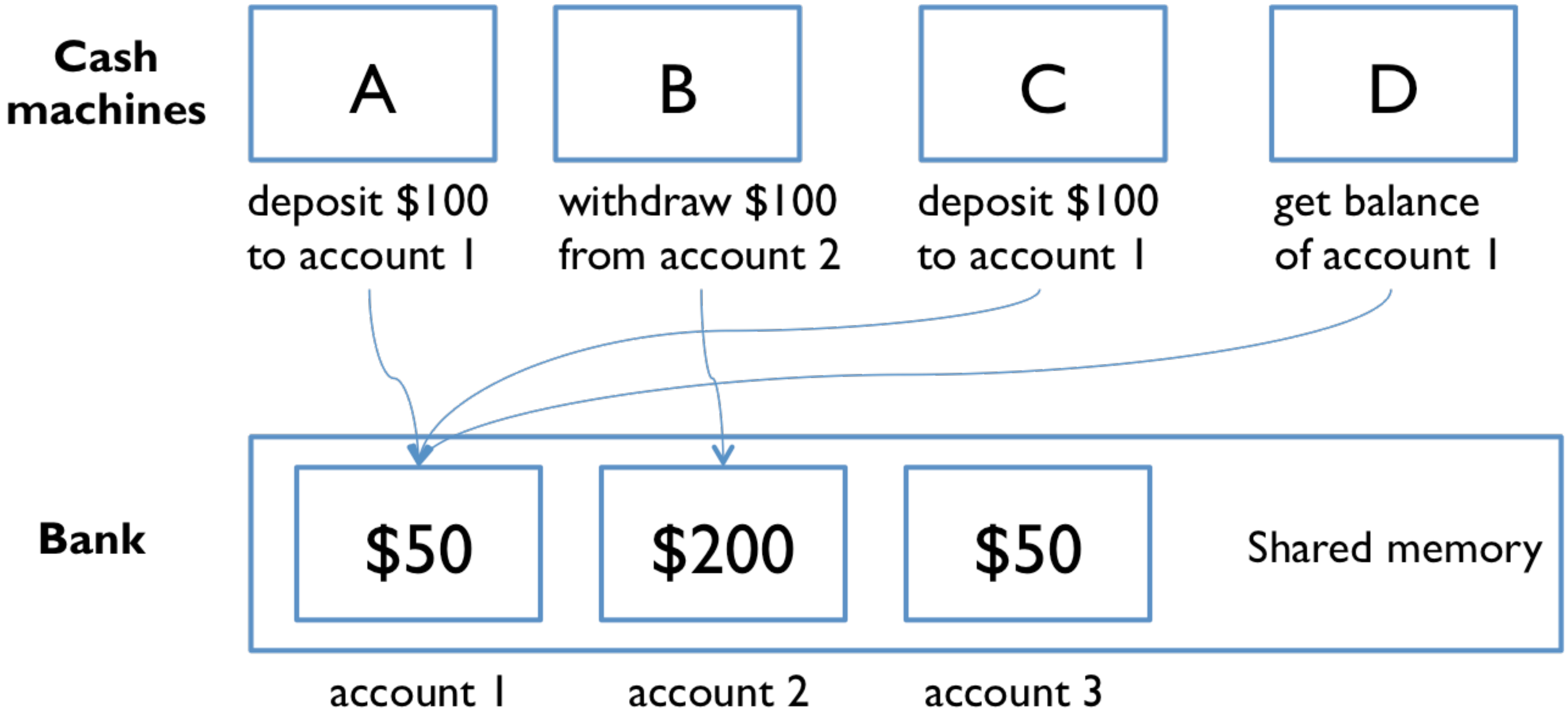
- One common solution is aging
  - One parameter to priority assignment is the amount of time the process has been waiting
- The longer a process waits, the higher its priority becomes

# Resource sharing

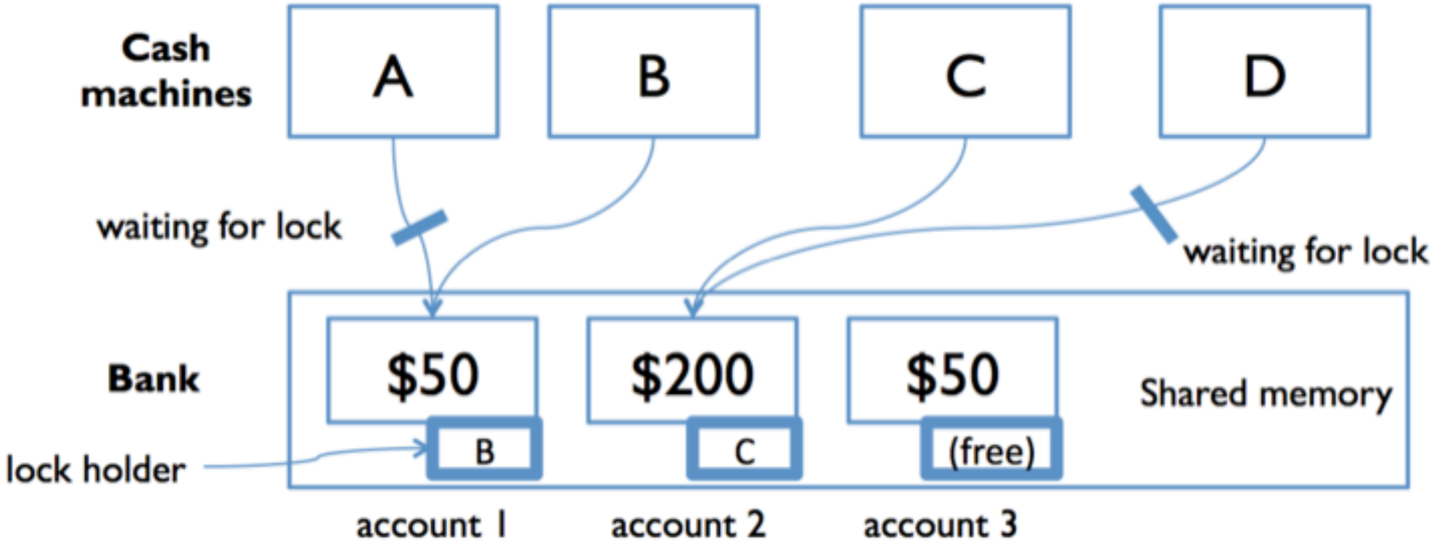
- In most systems, processes share resources apart from the processor
  - Memory areas, Files, Network, …

- Synchnronisation mechanisms (semaphores, locks, …) are used to manage shared resources

# The need for synchronisation



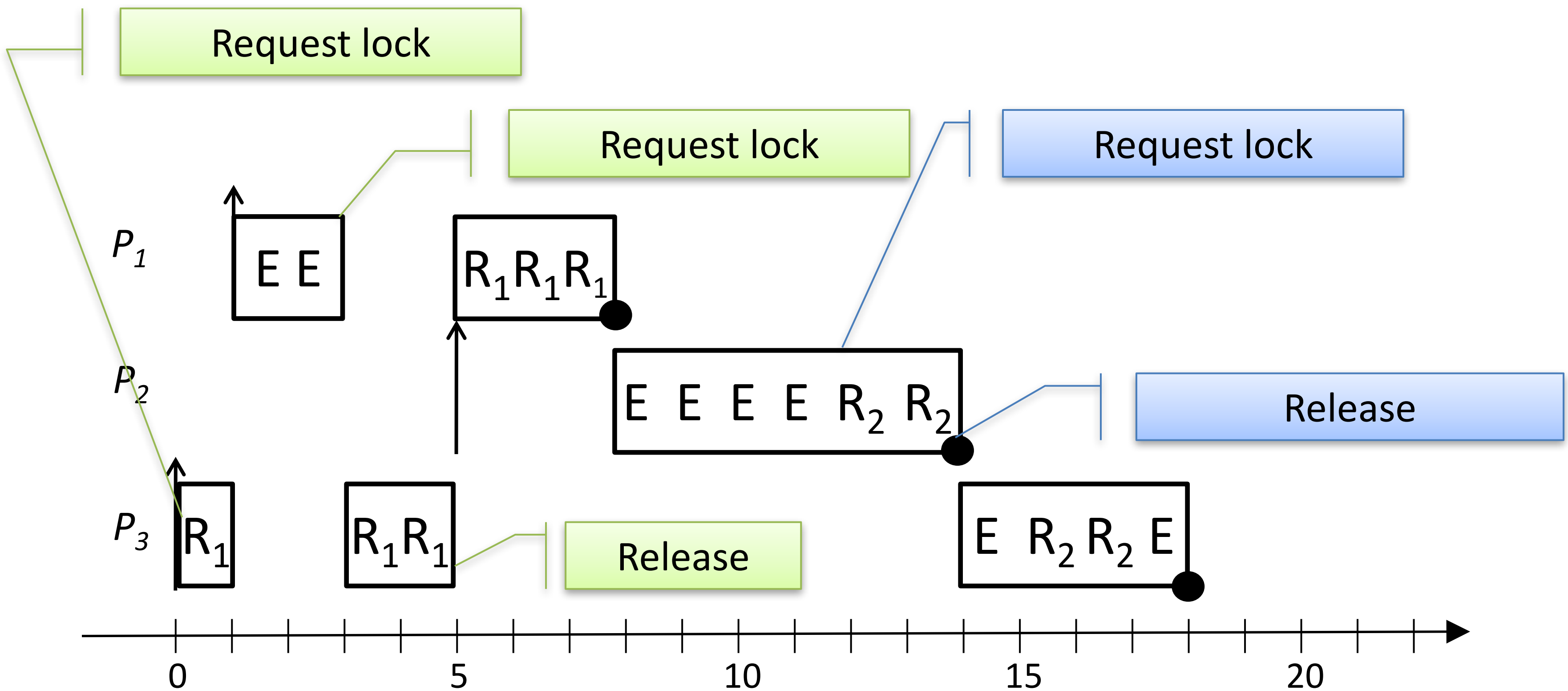
# The need for synchronisation




- Now, before they can access or update an account balance, cash machines must first acquire the lock on that account

# Resource sharing

| Process | Execution time | Priority | Arrival instant | Execution sequence |
|---|---|---|---|---|
| $P_1$ | 5 | 1 | 1 | E E $R_1$ $R_1$ $R_1$ |
| $P_2$ | 5 | 2 | 5 | E E E E $R_2$ $R_2$ |
| $P_3$ | 7 | 3 | 0 | $R_1$ $R_1$ $R_1$ E $R_2$ $R_2$ E |

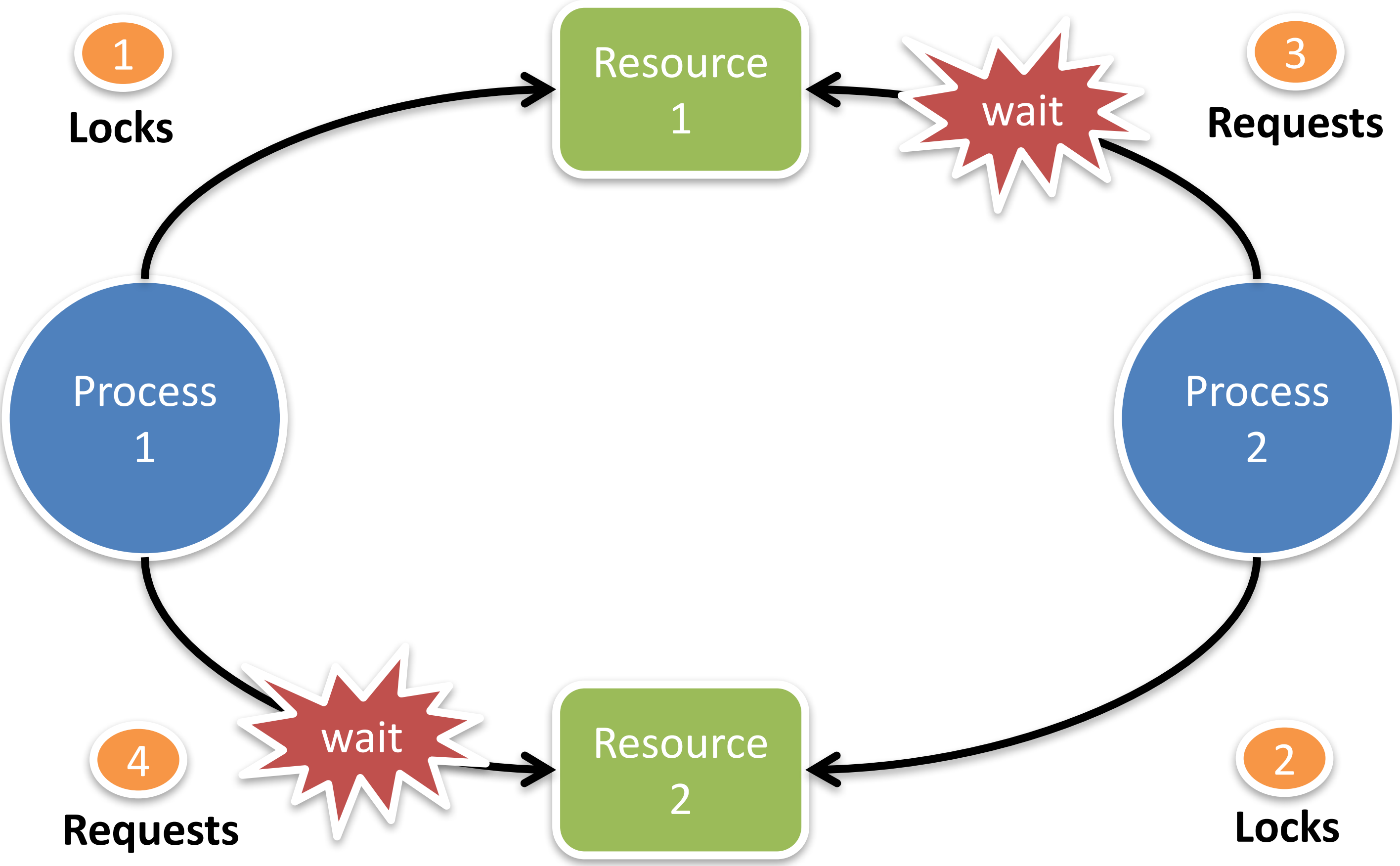


# Deadlock

- A condition involving one or more processes and one or more resources, such that each process waits for one of the resources, but all the resources are already held
  - Therefore, none of the processes can continue

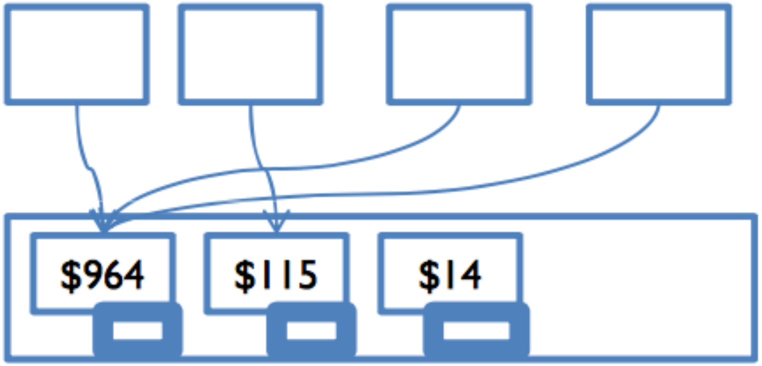
- The most common example is with two processes and two resources

# Deadlock

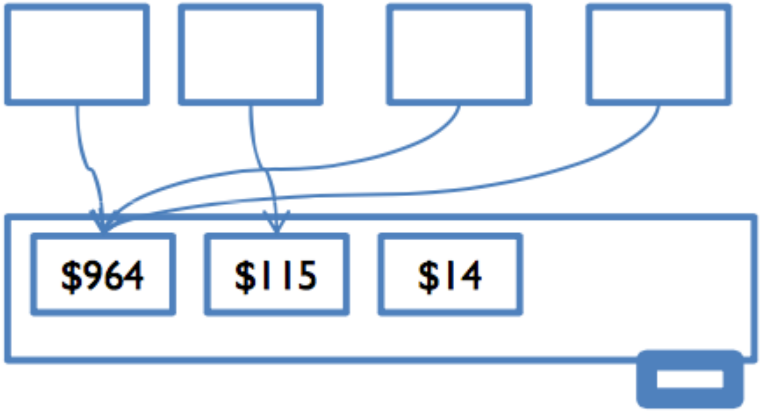

1
Locks
Resource
1
wait
3
Requests
Process
1
Process
2
4
Requests
wait
Resource
2
2
Locks

# Avoiding deadlocks

- Start with a coarse-grained approach, identify bottlenecks, and add finer-grained locking where necessary to alleviate the bottlenecks




one lock per account




one lock for the whole bank

# Avoiding deadlocks

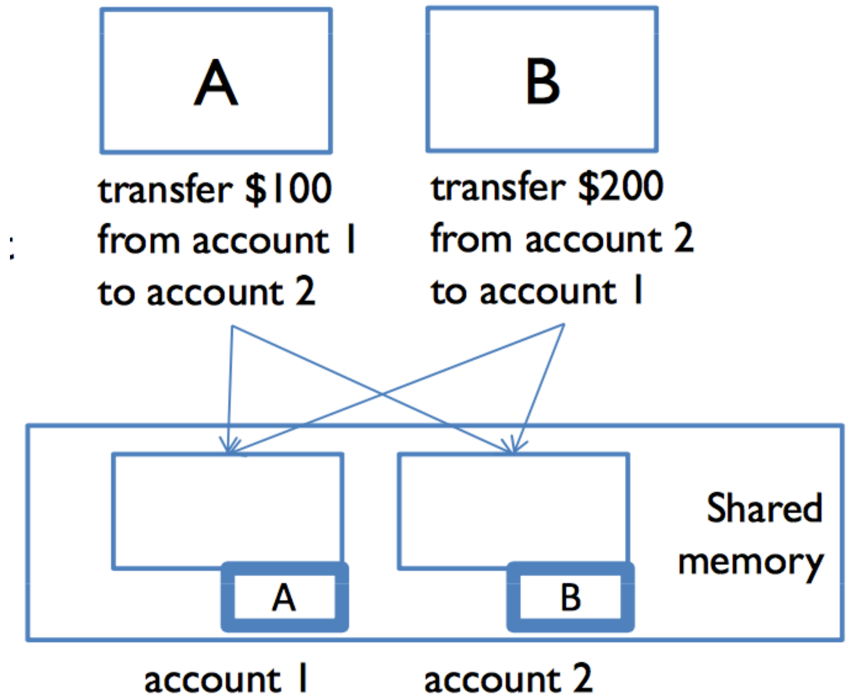
- Implement lock ordering when using multiple locks
  - Nested locks must always be obtained in the same order (not always easy in practice)

```
void transferMoney(Account *fromAccount, Account
      *toAccount, float amountToTransfer){

      sem_wait(fromAccount->lock);
      sem_wait(toAccount->lock);
      debit(fromAccount,amountToTransfer);
      credit(toAccount,amountToTransfer);
      sem_post(fromAccount->lock);
      sem_post(toAccount->lock);
}
```

# Problem

- Suppose A and B are making simultaneous transfers between two accounts in opposite directions (A: $C_1 \rightarrow C_2$; B: $C_2 \rightarrow C_1$)



# Possible solution

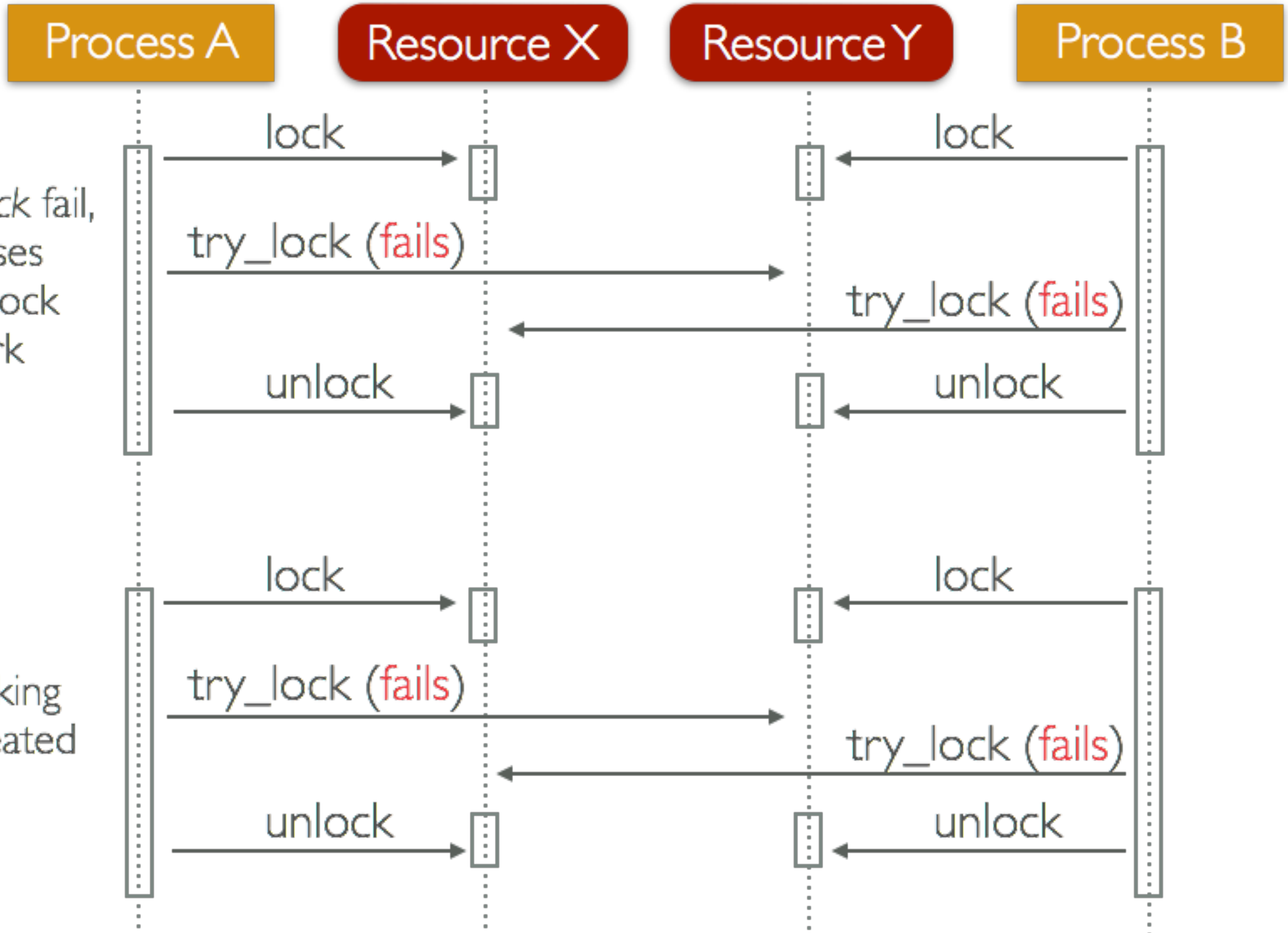
- Impose a maximum waiting time for acquiring the lock…
  - In POSIX, *sem_trywait()* and *sem_timedwait()*
- … and try again later
  - New attempt is usually done after a random waiting period, but several approaches are possible

# Livelock

- Similar to a deadlock, except that the states of the processes involved in the livelock constantly change with regard to one another, none progressing

- Livelock is a risk with some algorithms that detect and recover from deadlock
  - If more than one process takes action, the deadlock detection algorithm can be repeatedly triggered

# Livelock




After the *try_lock* fail, both processes release their lock and no work is done

The same locking pattern is repeated

# Priority inversion

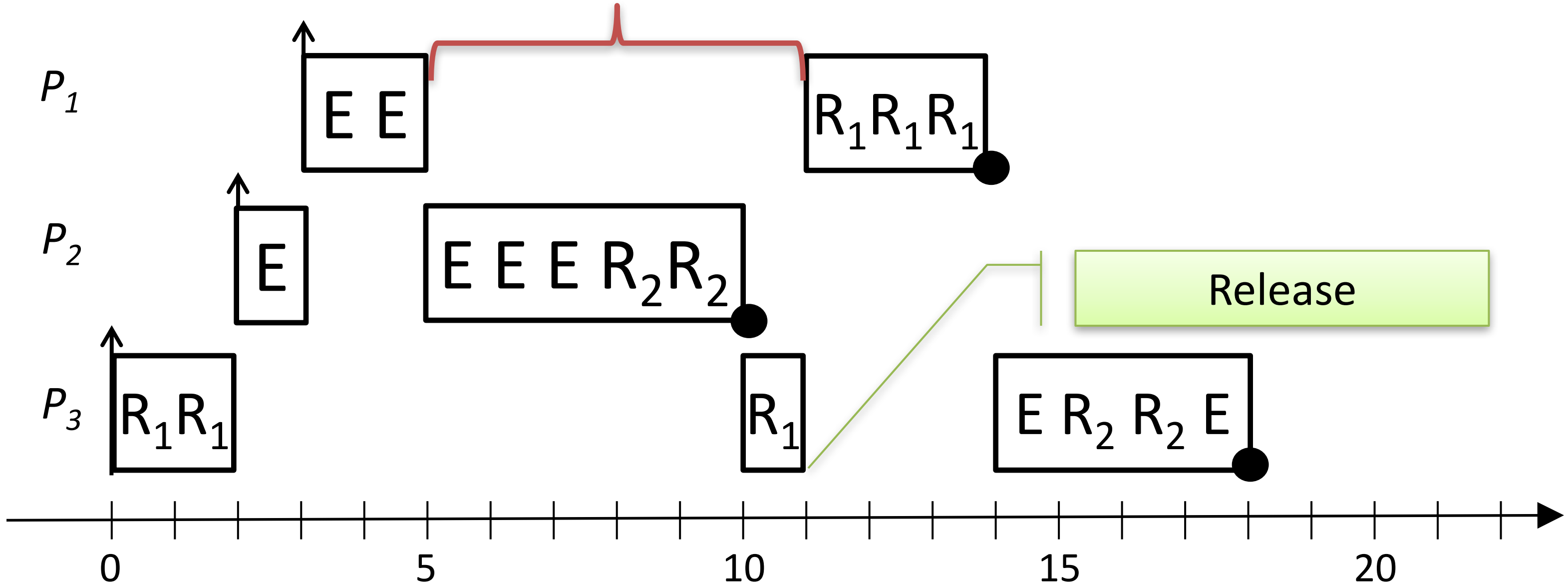
- When a higher priority process is indirectly preempted by a lower priority one, effectively "inverting" the relative priorities of the two processes

- This violates the priority model
  - High priority tasks can only be prevented from running by higher priority tasks

# Priority inversion

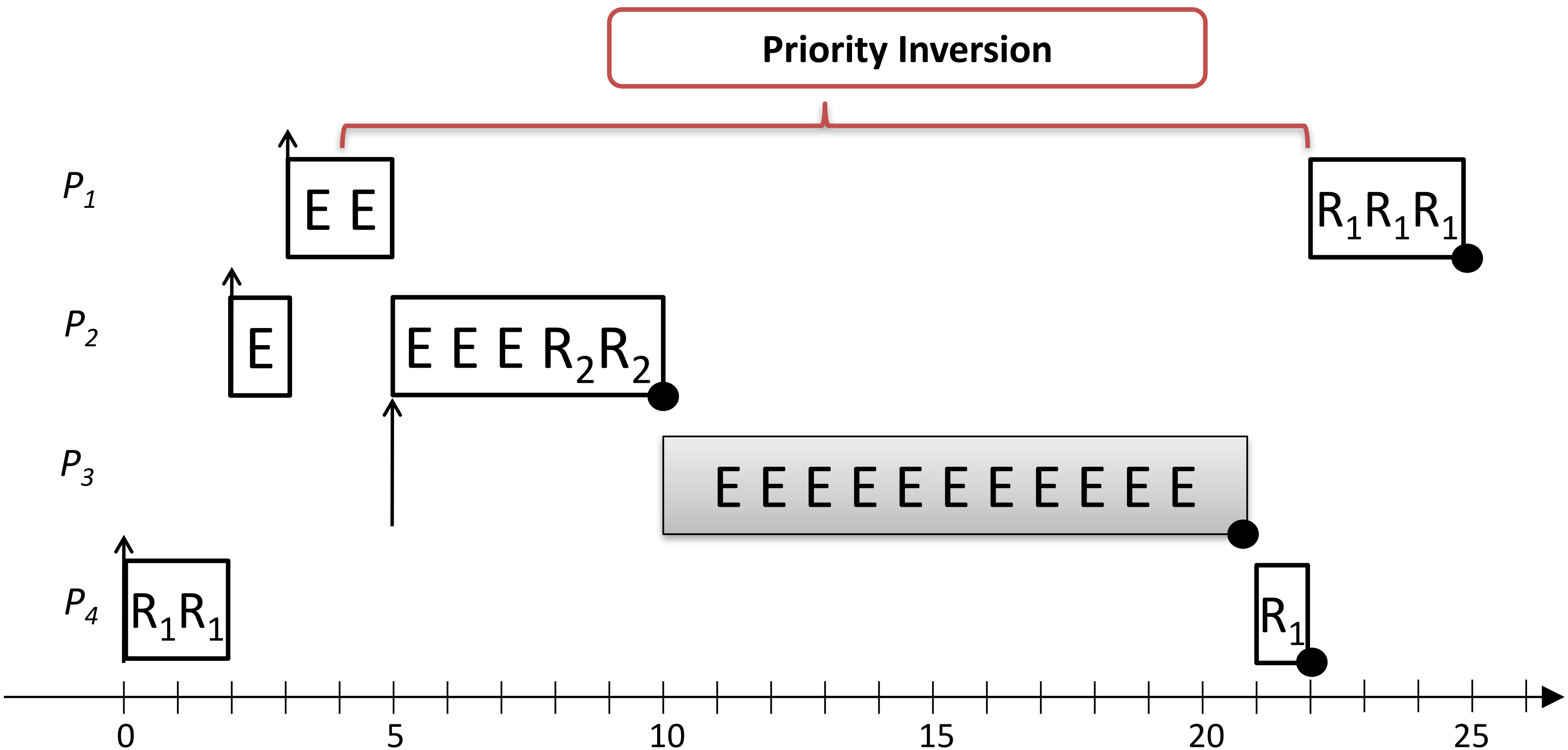
| Process | *Execution time* | *Priority* | *Arrival instant* | *Execution sequence* |
|---|---|---|---|---|
| $P_1$ | 5 | 1 | 3 | E E $R_1$ $R_1$ $R_1$ |
| $P_2$ | 5 | 2 | 2 | E E E E $R_2$ $R_2$ |
| $P_3$ | 7 | 3 | 0 | $R_1$ $R_1$ $R_1$ E $R_2$ $R_2$ E |

A higher priority process ($P_1$) waits for a lower priority process ($P_3$) while middle priority processes are allowed to execute



# Priority inversion

- If no rule is applied when sharing resources, it is impossible to determine the maximum blocking time



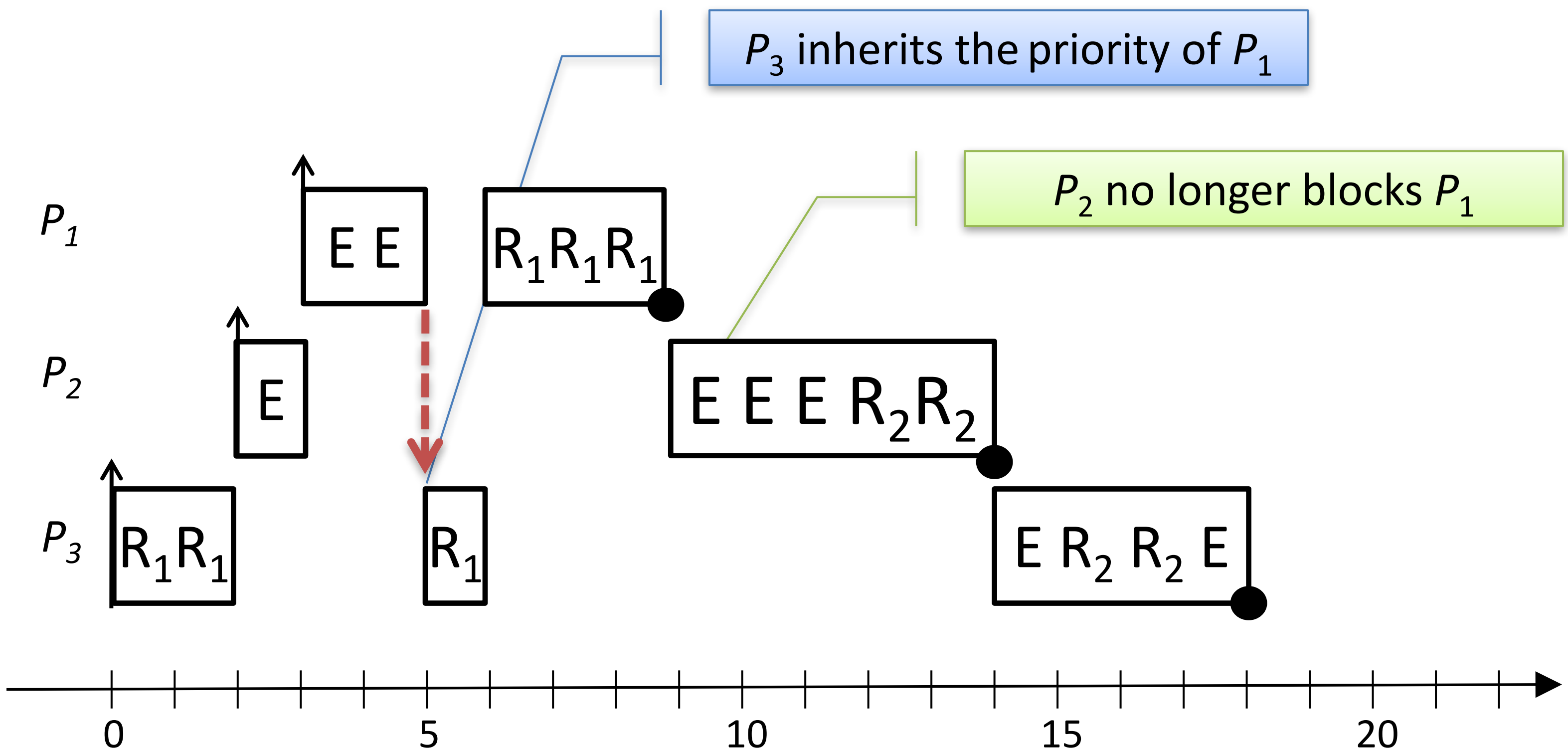
# Avoiding priority inversion

- Allow low priority processes to quickly complete their use of a shared resource
- Two main protocols
  - Priority Inheritance Protocol
  - Priority Ceiling Protocol

# Priority Inheritance Protocol

- When a higher priority process is blocked in a shared resource, the lower priority process using the resource "inherits" the higher priority (only when using)
- Allows several blocking periods but guarantees a maximum blocking period

# Priority Inheritance Protocol



# Priority Inheritance Protocol

- If a process has m critical sections that can lead to it being blocked, then the maximum number of times it can be blocked is m

- If B is the maximum blocking time and K is the number of critical sections, process i has an upper bound on its blocking given by:

$$B_i = \sum_{k=1}^{K} usage(k,i)C(k)$$

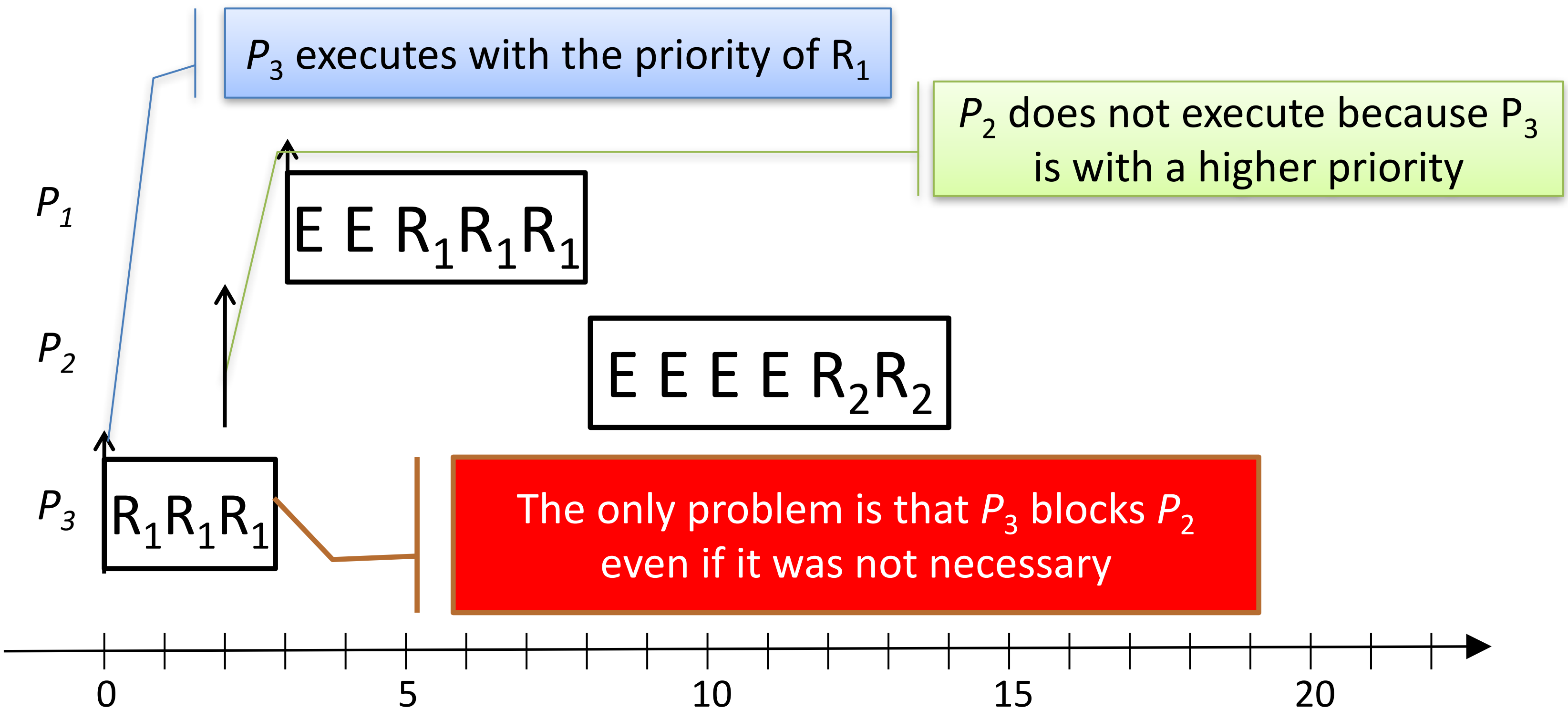
# Priority Inheritance Protocol

- Blocking is zero for the lowest priority process
- *usage(k,i)* is one if resource K is used by any process with equal or higher priority (or by i itself) than process i **AND** by any lower priority process
- Otherwise, *usage(k,i)* is zero

# Priority Ceiling Protocol

- Resources are given a priority which is equal to the highest priority of the processes that use the resource (ceiling)
- When holding the resource, processes execute with the priority of the resource (ceiling)
- Allows only one blocking period (but introduces unnecessary blocking)
  - Also prevents deadlocks

# Priority Ceiling Protocol



# Priority Ceiling Protocol

- As a consequence, a process will only suffer a block at the very beginning of its execution
  - Once the process starts actually executing, all the resources it needs must be free
  - If they were not, then some process would have an equal or higher priority and the process's execution would be postponed

$$B_i = \max_{k=1}^{k} usage(k,i)C(k)$$

# Response time with blocking

$$R_i = C_i + B_i + I_i$$

$$R_i = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

$$w_i^{n+1} = C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil C_j$$

# Conclusions

- Resource starvation, deadlock, livelock, and priority inversion are problems that all programmers of concurrent solutions must know and master

- They might not be obvious and occur in rare and unpredictable ways, imposing serious problems to applications

# Conclusions

- By dedicating a higher degree of attention in the design of the synchronisation solution and imposing clear rules of when and how to lock more than one resource, problems can be greatly reduced

- One critical aspect – ignored many times – is the documentation of the synchronisation solution, even in cases when a huge attention is dedicated to its design