Course 02158

Deadlocks

Hans Henrik Løvengreen

DTU Compute

Deadlock

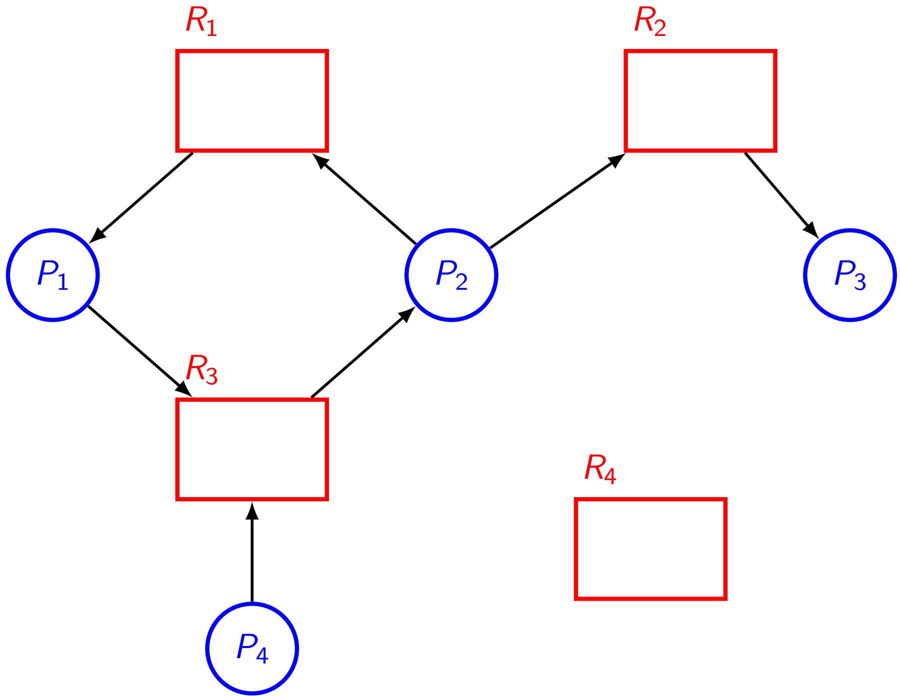
General Definition

- A set of processes S is *deadlocked* if each process in S is waiting for an event that can be caused only by a process in S.

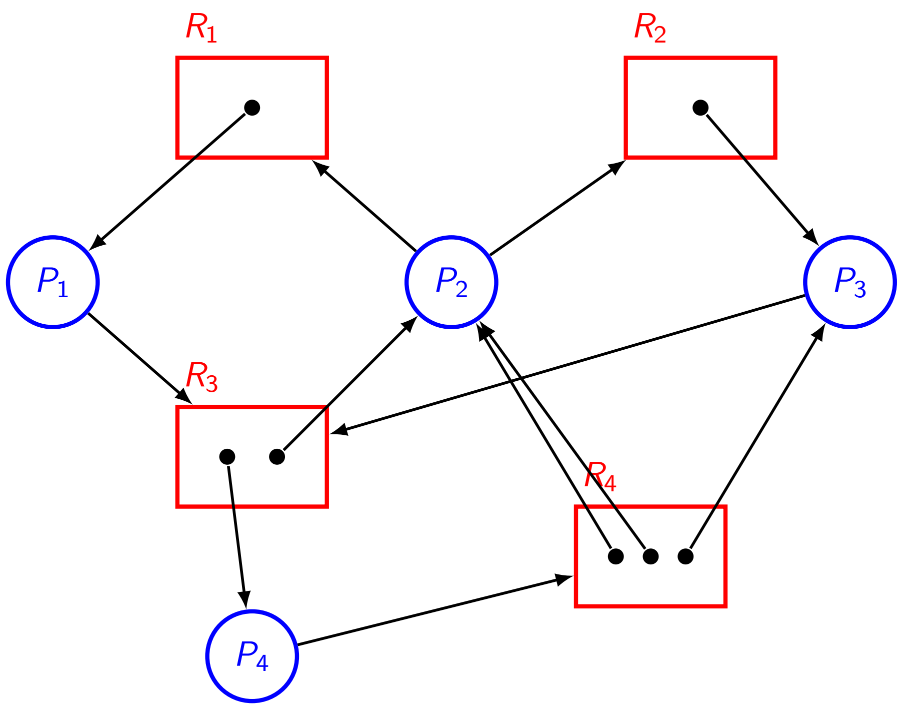
Resource Control

- Resource usage: *Request*; *Use*; *Release*
- Process waits for one or more resources held by others
- Necessary condition for deadlock:
  - **Mutual exclusion**
  - **Hold-and-wait**
  - **No preemption**
  - **Circular wait**
- Control may be *local* (locks) or *global* (resource manager, OS)

Resource Allocation Graph



Resource Allocation Graph — multiple instances



Principles of dealing with deadlock

- Deadlock *prevention*
- Deadlock *avoidance*
- Deadlock *detection* and *recovery*
- Ignore (hope for the best)

Deadlock Prevention

Idea

- To introduce *structural restrictions* that eliminates deadlock risk

Methods

- Mutual exlusion
  Enable simultaneous use, e.g. by spooling [not general]
- Hold-and-wait
  Reserve all resources at once [low utilization, risk of starvation]
- No-preemption
  Allow preemption, e.g. of CPU and memory [not general]
- Circular wait
  Assign ranks to resource types:
  A process may only request resources having strictly higher rank than already allocated ones.

Deadlock Avoidance

Idea

- To use *behavioural information* to dynamically avoid deadlock.

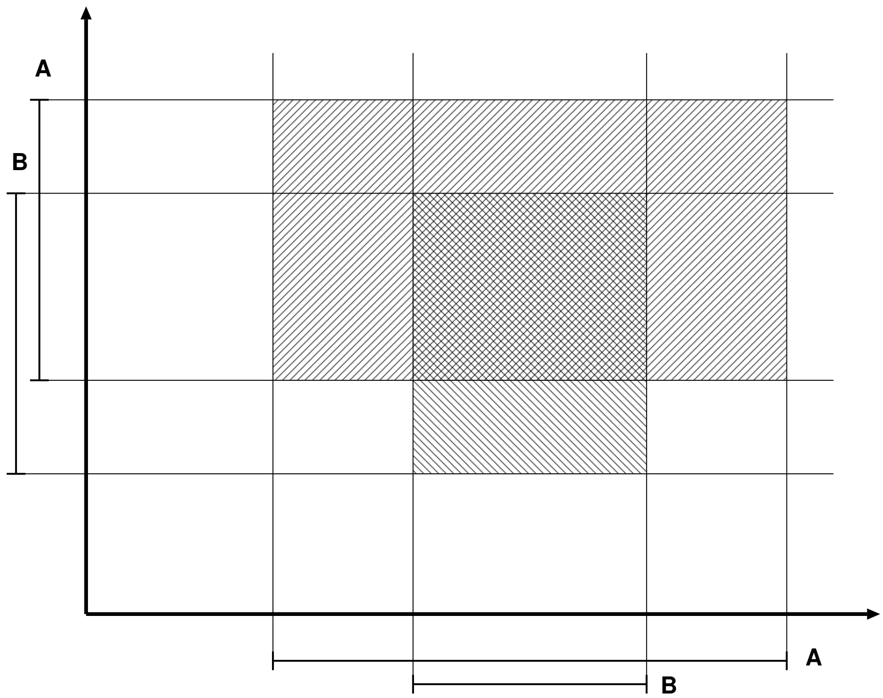
Prerequisites

- Behavioural information must be available for all processes
- Examples:
  - Max resource claim for each resource type
  - Resource usage pattern

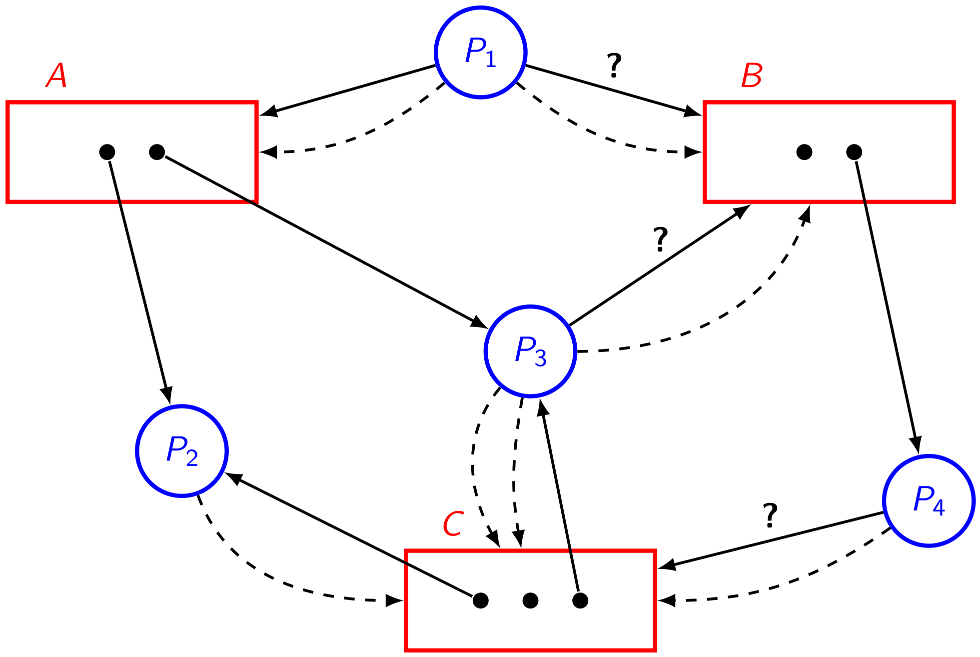
Method

- A *safe* state is a state from which there exists a way to terminate all processes (according to usage information).
- **Banker's Algorithm** A resource request is granted only if the resulting state remains safe.

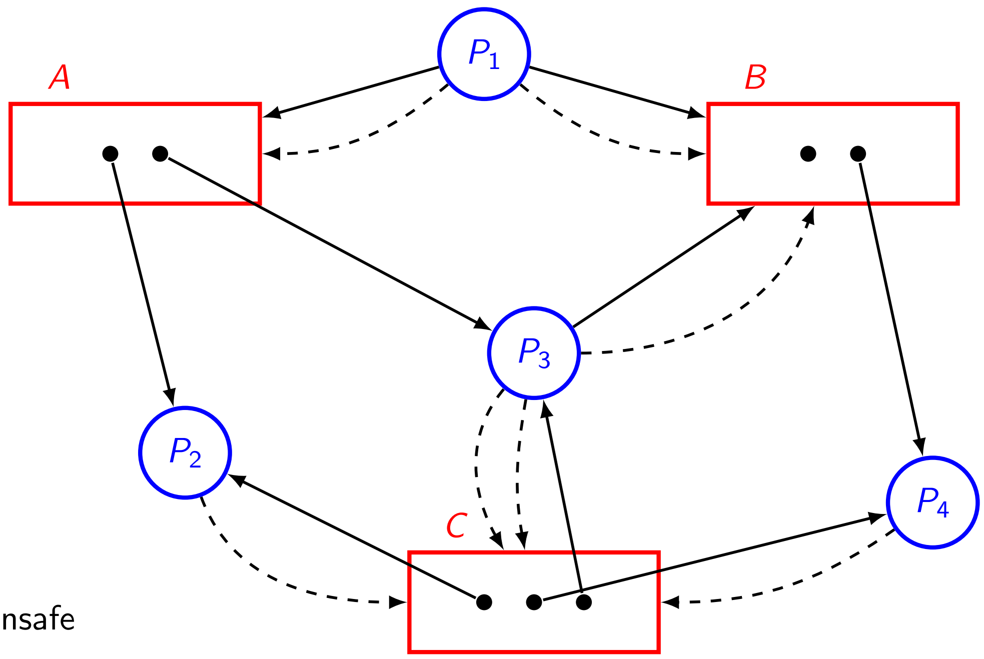
Safe/unsafe States



Banker's Algorithm

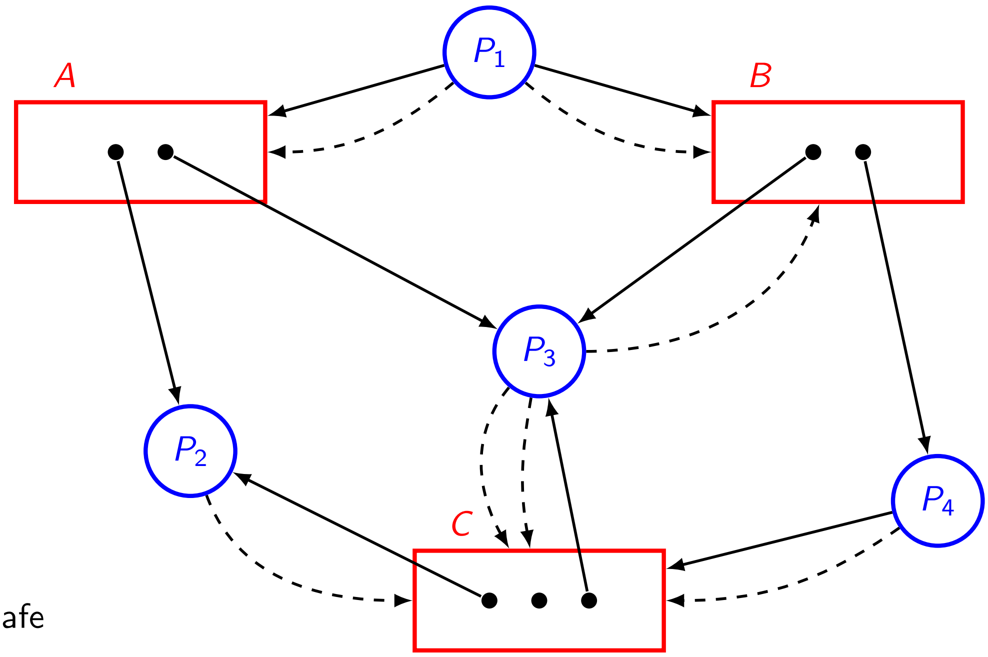


Banker's Algorithm



- Unsafe

Banker's Algorithm



- Safe

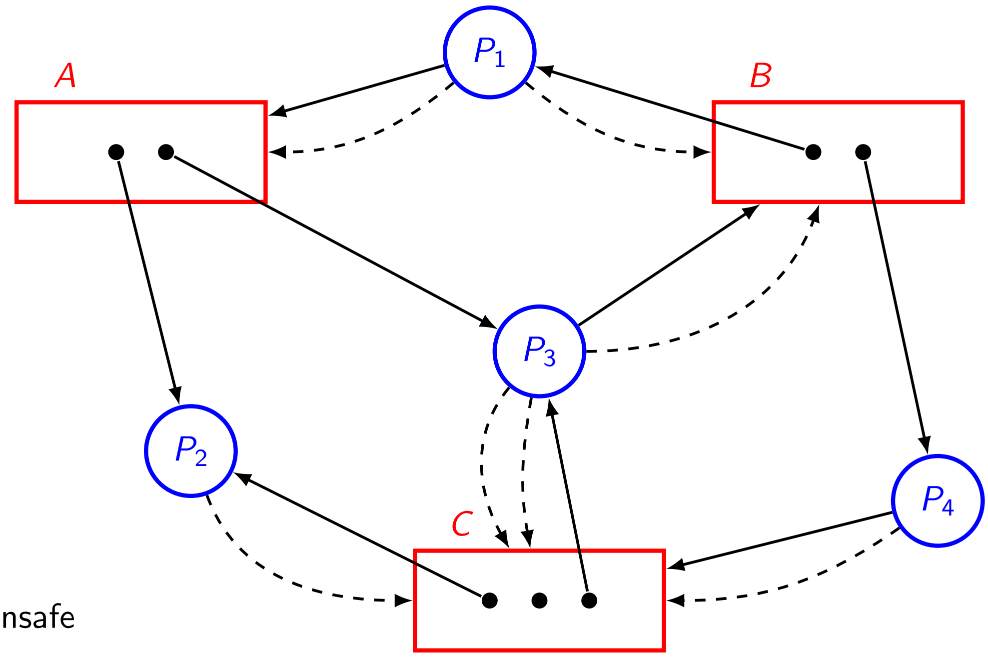
Banker's Algorithm

- After assigning the B-instance to P_3:

	Allocation			*Need*		
	A	B	C	A	B	C
P_1	0	0	0	2	2	0
P_2	1	0	1	0	0	1
P_3	1	1	1	0	1	2
P_4	0	1	0	0	0	2

Available			*Can finish*
A	B	C	
0	0	1	P_2
1	0	2	P_4
1	1	2	P_3
1	2	3	P_1
2	2	3	

Banker's Algorithm



- unsafe

Banker's Algorithm

- After assigning the B-instance to P_1:

	Allocation			*Need*		
	A	*B*	*C*	*A*	*B*	*C*
P_1	0	1	0	2	1	0
P_2	1	0	1	0	0	1
P_3	1	0	1	0	2	2
P_4	0	1	0	0	0	2

Available			*Can finish*
A	*B*	*C*	
0	0	1	P_2
1	0	2	P_4
1	1	2	

Deadlock Detection

Idea

- To detect deadlocks and handle them by automatic recovery

Deadlock Detection

- Maintain global allocation state and perform deadlock detection:
  - Regularly
  - When some process seems not to make progress
- Assume deadlock if no progress for a while

Recovery

- Select one or more victims based on *cost factors*
- Kill victim or *roll-back* to *check-point*
- Risk of starvation

Deadlock Summary

Principles

- Deadlock *prevention*
- Deadlock *avoidance*
- Deadlock *detection* and *recovery*
- Ignore (hope for the best)

Practice

- Often ignored — otherwise:
- Local control through *locks*
- Deadlock prevention qua *resource ordering*
- Example: Linux kernel locks

Feasible?

- Behavioural information may be used for deadlock avoidance in an ad-hoc way

Locking in the Linux Kernel

Development

- First uni-processor kernels: No need for lock in kernel (interrups disabled)
- First SMP kernels: A single *big kernel lock* (spin-lock)
- Preemptive kernels (kernel threads may be scheduled): Multiple locks
- Both *sleeping locks* and *spin-locks*, generally *r/w*

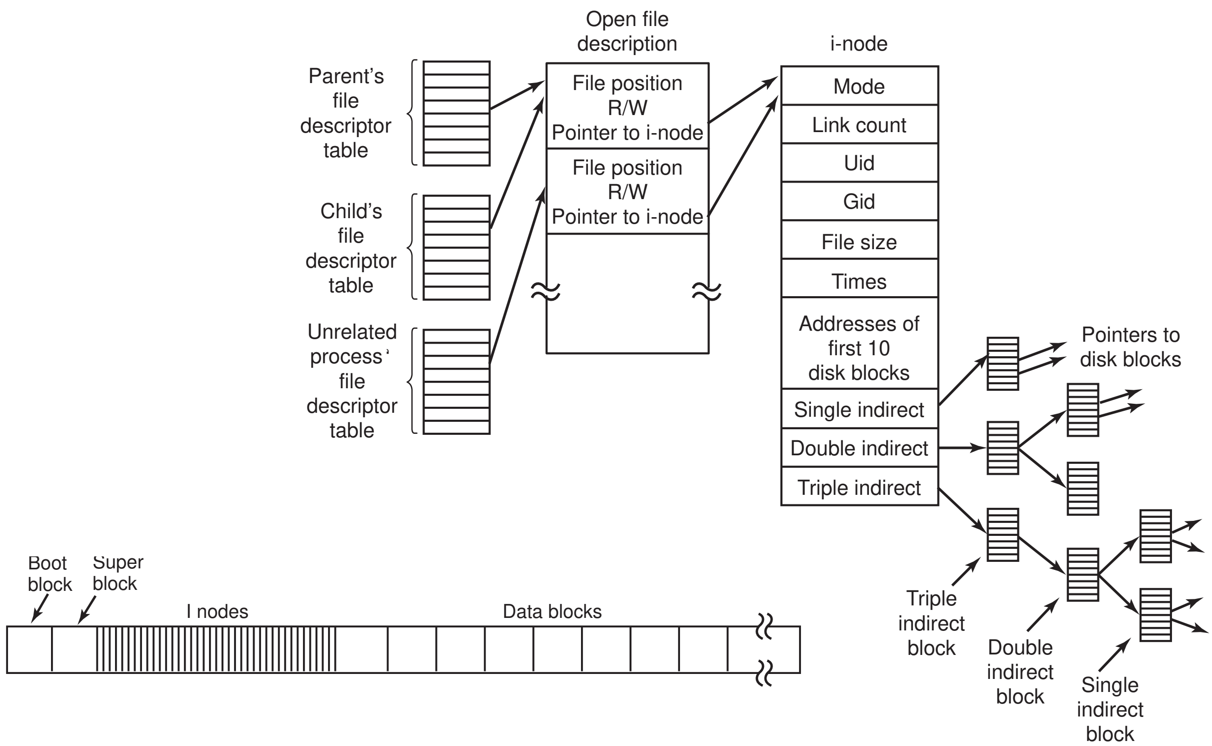
Status

- There are now *thousands of lock classes*
- Lock ordering only *sparsely documented* — in the code!
- No central documentation of locking order

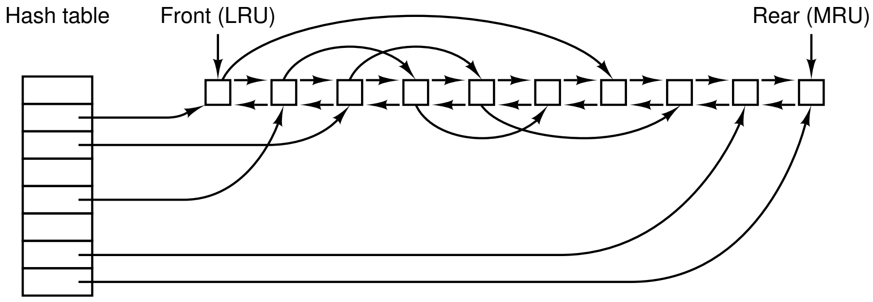
Tools

- *Static analysis* of the code is very difficult and incomplete
- The kernel may be instrumented for recording of locking/unlocking (lockdep)
- Potential lock cycles may be detected and reported at runtime *on-the-fly*
- The locking trace may be analyzed post execution (LockDoc)

Example: Linux ext-2



Example: Linux ext-2



Linux: Inode locking order I

Linux 6.0

- In fs/inode.c

```
/*
 * Inode locking rules:
 *
 * inode->i_lock protects:
 * inode->i_state, inode->i_hash, __iget(), inode->i_io_list
 * Inode LRU list locks protect:
 * inode->i_sb->s_inode_lru, inode->i_lru
 * inode->i_sb->s_inode_list_lock protects:
 * inode->i_sb->s_inodes, inode->i_sb_list
 * bdi->wb.list_lock protects:
 * bdi->wb.b_{dirty,io,more_io,dirty_time}, inode->i_io_list
 * inode_hash_lock protects:
 * inode_hashtable, inode->i_hash
 */
```

Linux: Inode locking order II

```
/*
 * Lock ordering:
 *
 * inode->i_sb->s_inode_list_lock
 * inode->i_lock
 * Inode LRU list locks
 *
 * bdi->wb.list_lock
 * inode->i_lock
 *
 * inode_hash_lock
 * inode->i_sb->s_inode_list_lock
 * inode->i_lock
 *
 * iunique_lock
 * inode_hash_lock
 */
```