TxOS

Emmett Witchel

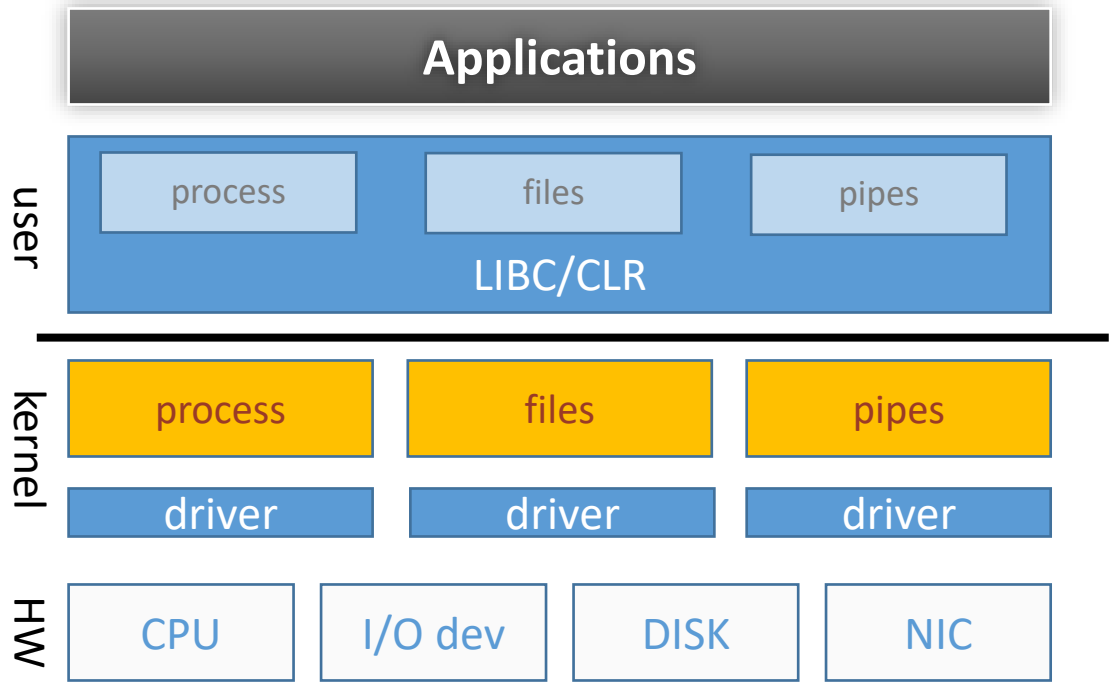
CS380L

TxOS faux quiz

- What is a TOCTOU bug? Give an example. Can you think of solutions other than transactions?
- Why does TxOS “split” kernel objects such as inodes?
- What techniques does TxOS use to handle output commit?
- List some performance tradeoffs TxOS makes in exchange for supporting general-purpose transactions. Can you think of scenarios where it is a good/poor tradeoff?
- How does TxOS support transactions in user-space?
- What is “strong isolation”? Asymmetric conflict? False conflict?

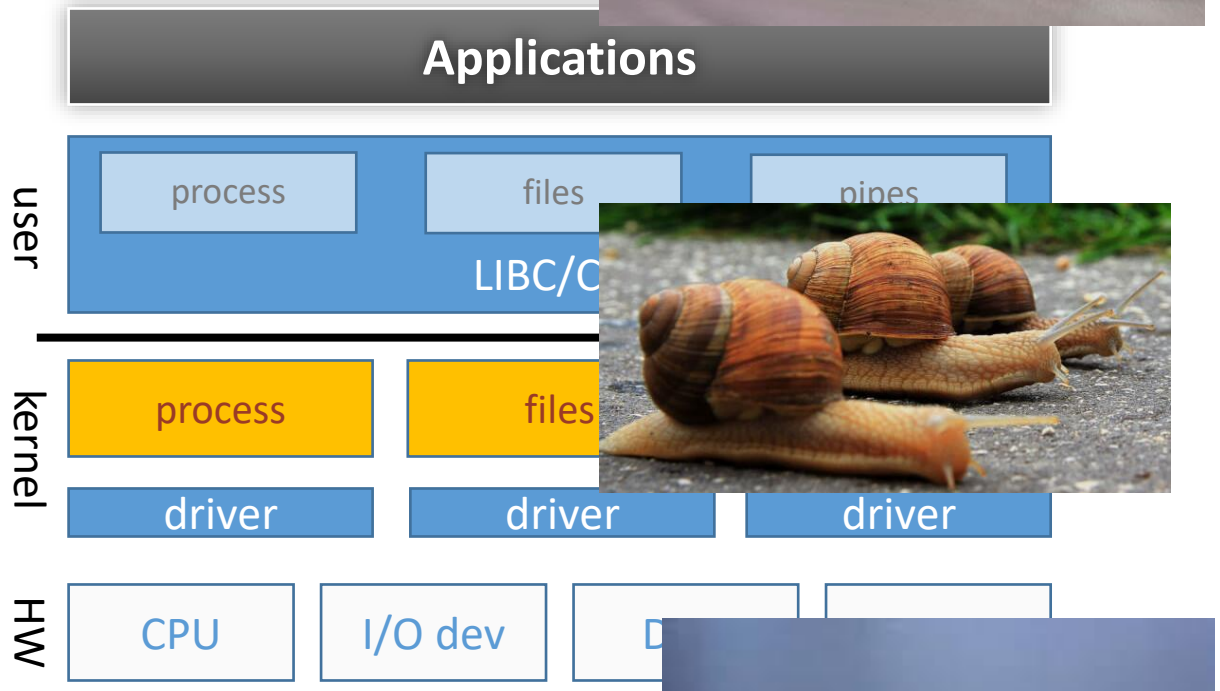
OS: a slowly evolving foundation

- All applications need the OS
  - Makes evolving the OS API difficult
- Fast evolution at stack extrema
  - User's needs change → old OS API ***creates security problems***
  - Hardware increasingly parallel → old OS API ***fails to unlock performance***
- But…must solve the problems of today with the API of yesterday!



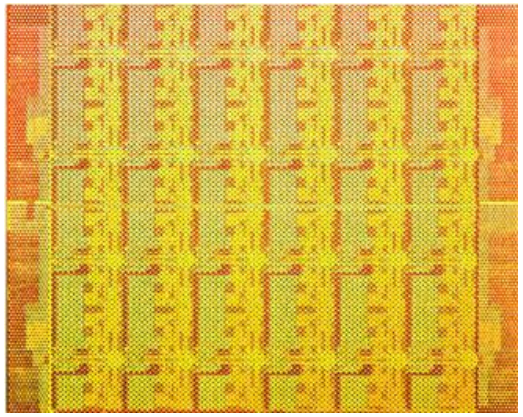
OS: a slowly evolving foundation

- All applications need the OS
  - Makes evolving the OS API difficult
- Fast evolution at stack extrema
  - User's needs change → old OS API ***creates security problems***
  - Hardware increasingly parallel → old OS API ***fails to unlock performance***
- But…must solve the problems of today with the API of yesterday!

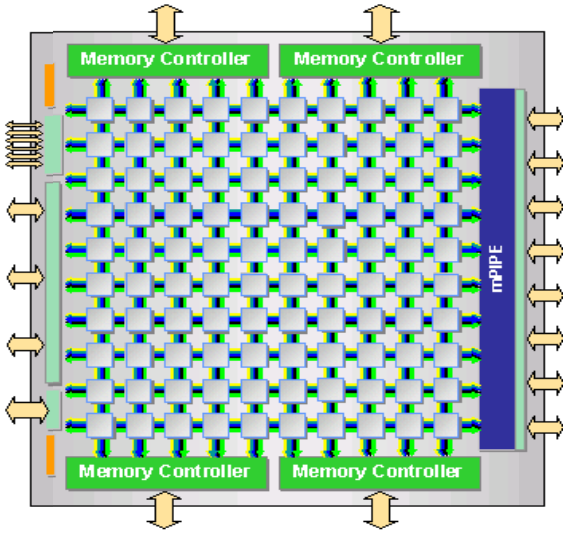


Trend toward concurrency

This laptop
2 Intel cores



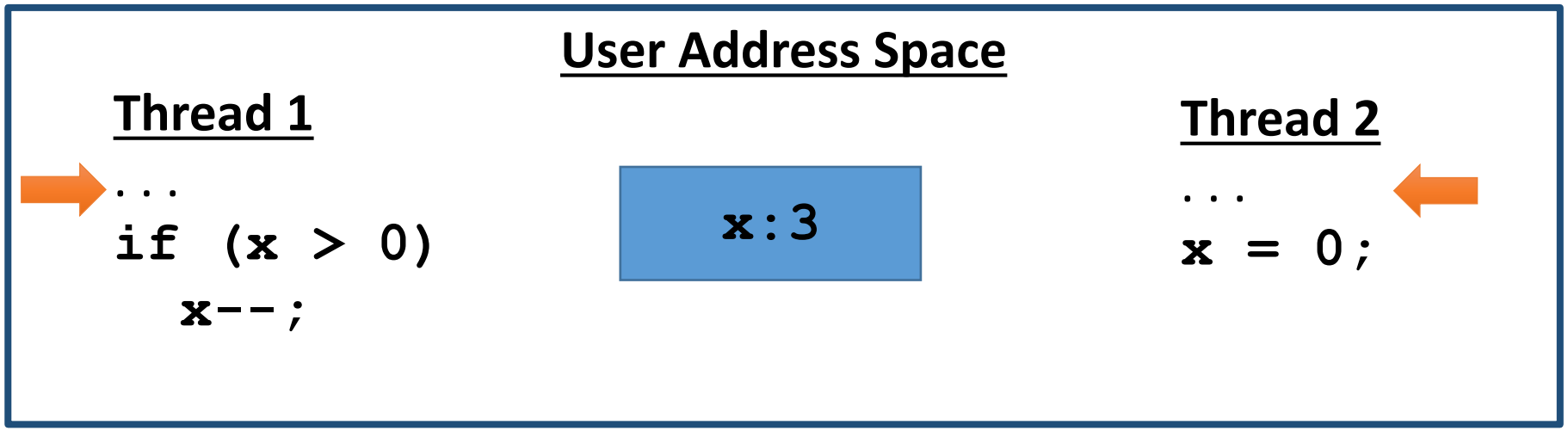
Intel Single-chip Cloud Computer
48 cores



Tilera Tile GX
100 cores

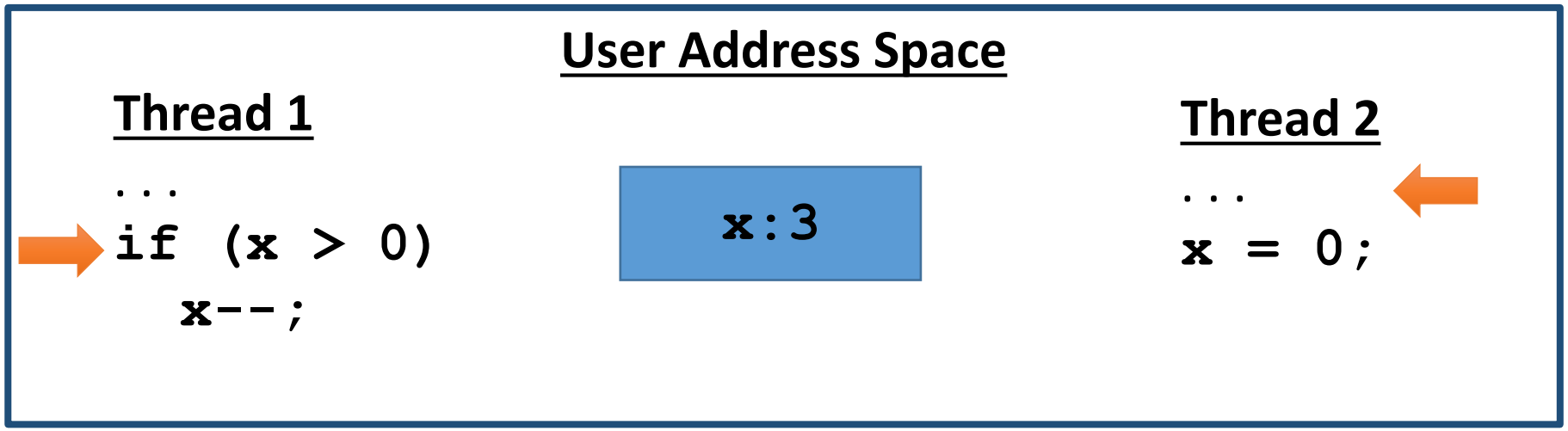
- New hardware is multi-core
- OS interface must provide concurrency control
  - Performance
  - Security

Example: Race condition in app



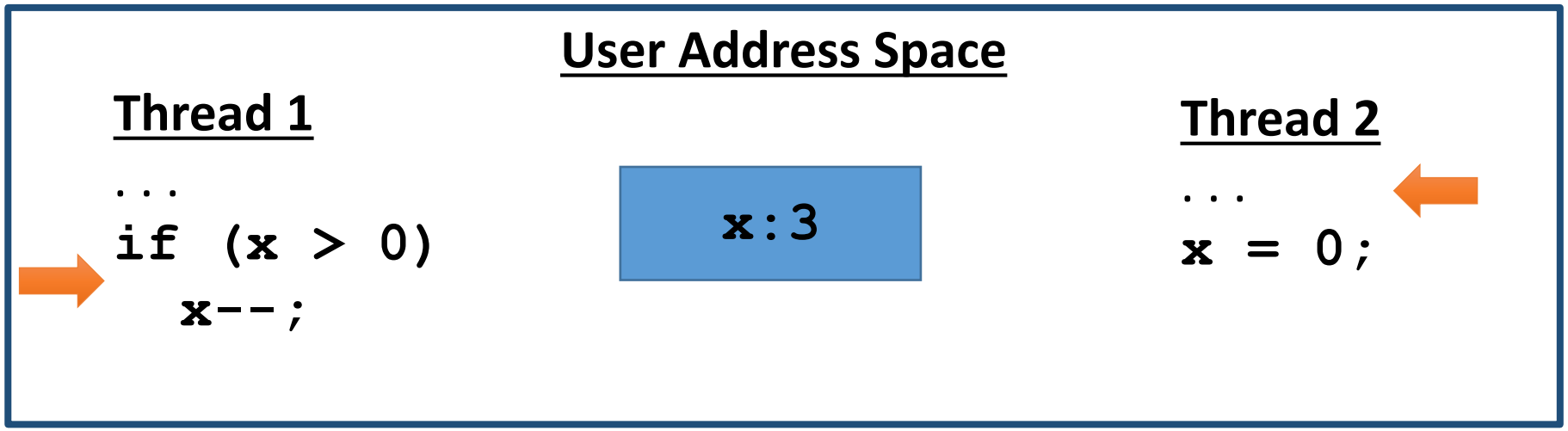
- Invariant: x >= 0
- Race conditions in the address space are a common mistake
- Commonly addressed with locks, synchronized keyword, condition variables, monitors, lock-free data structures, etc.

Example: Race condition in app



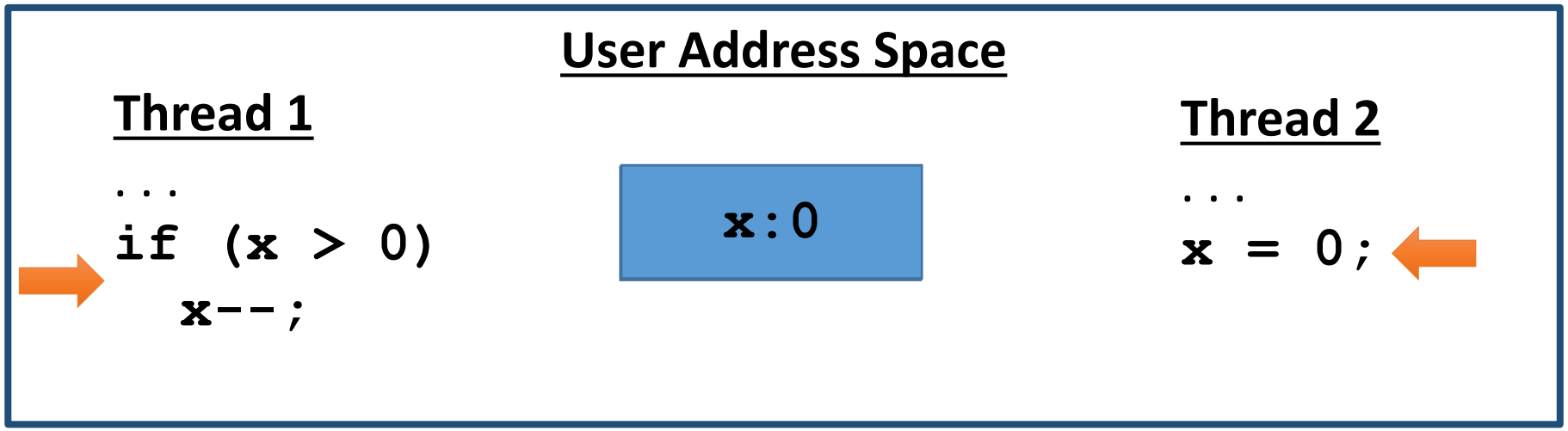
- Invariant: x >= 0
- Race conditions in the address space are a common mistake
- Commonly addressed with locks, synchronized keyword, condition variables, monitors, lock-free data structures, etc.

Example: Race condition in app



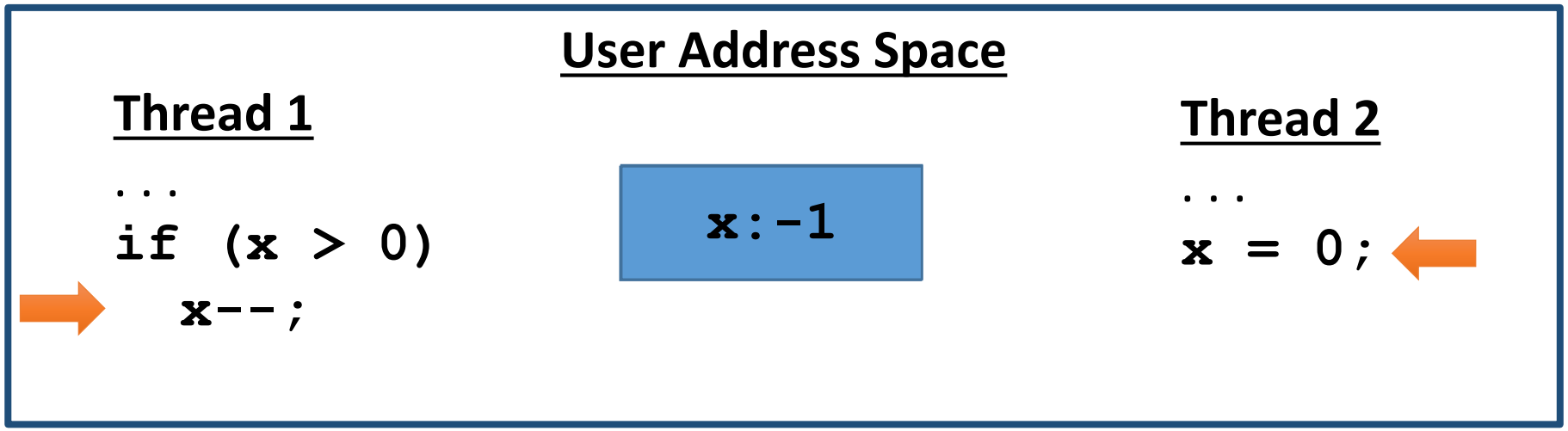
- Invariant: x >= 0
- Race conditions in the address space are a common mistake
- Commonly addressed with locks, synchronized keyword, condition variables, monitors, lock-free data structures, etc.

Example: Race condition in app



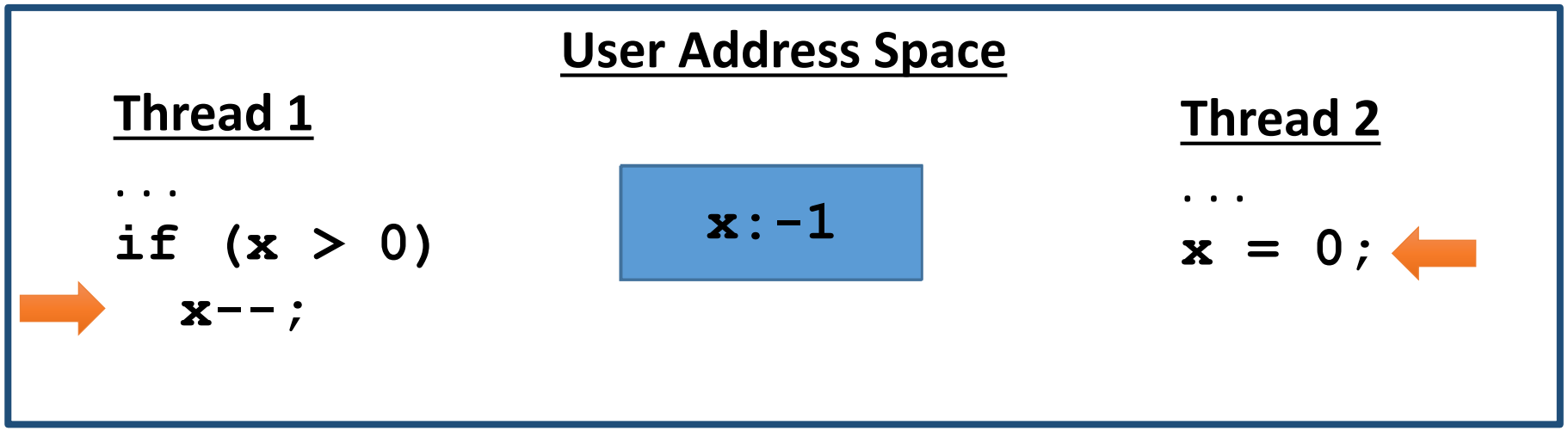
- Invariant: x >= 0
- Race conditions in the address space are a common mistake
- Commonly addressed with locks, synchronized keyword, condition variables, monitors, lock-free data structures, etc.

Example: Race condition in app



- Invariant: x >= 0
- Race conditions in the address space are a common mistake
- Commonly addressed with locks, synchronized keyword, condition variables, monitors, lock-free data structures, etc.

Example: Race condition in app

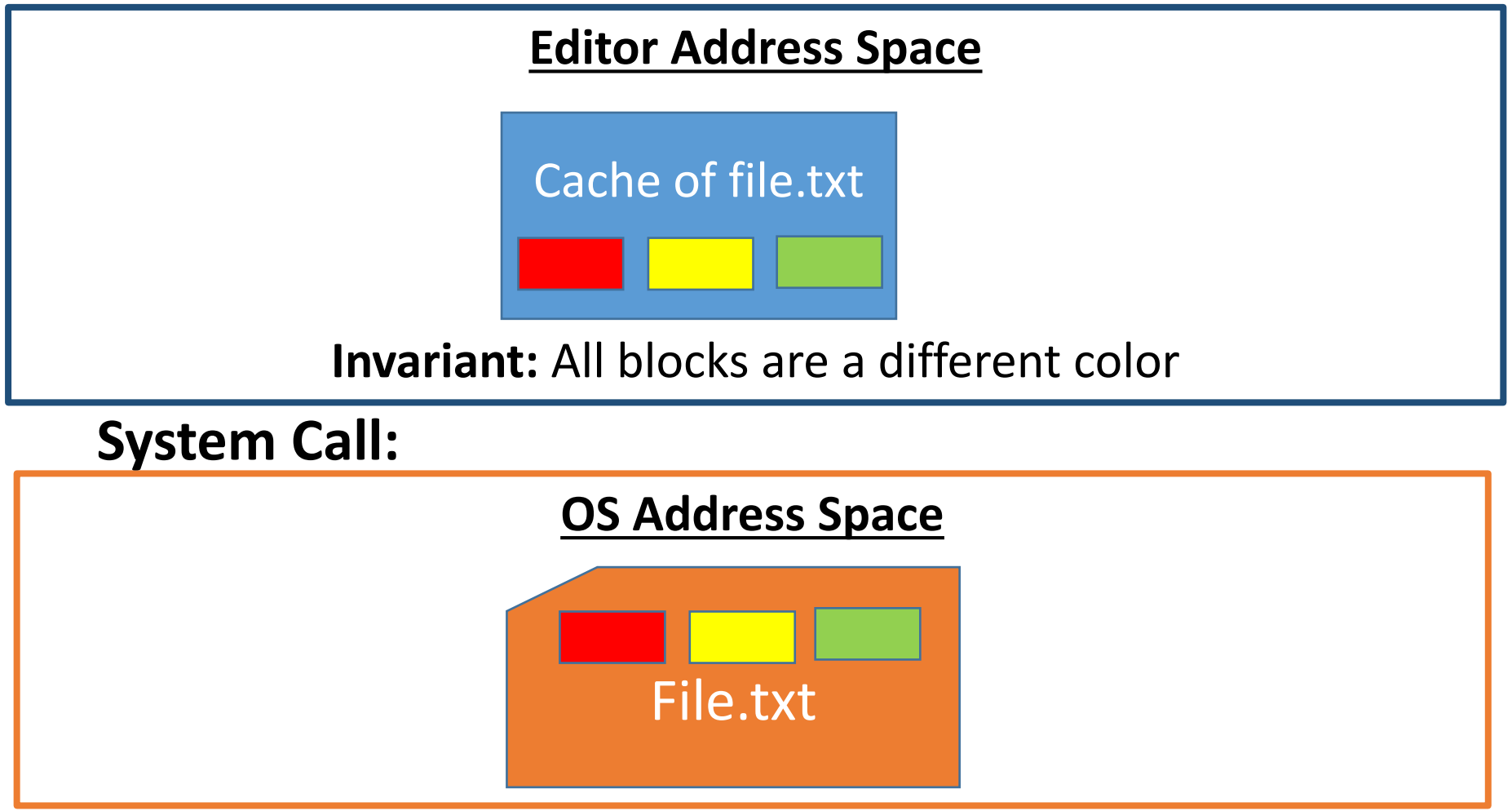


- Invariant: x >= 0
- Race conditions in the address space are a common mistake

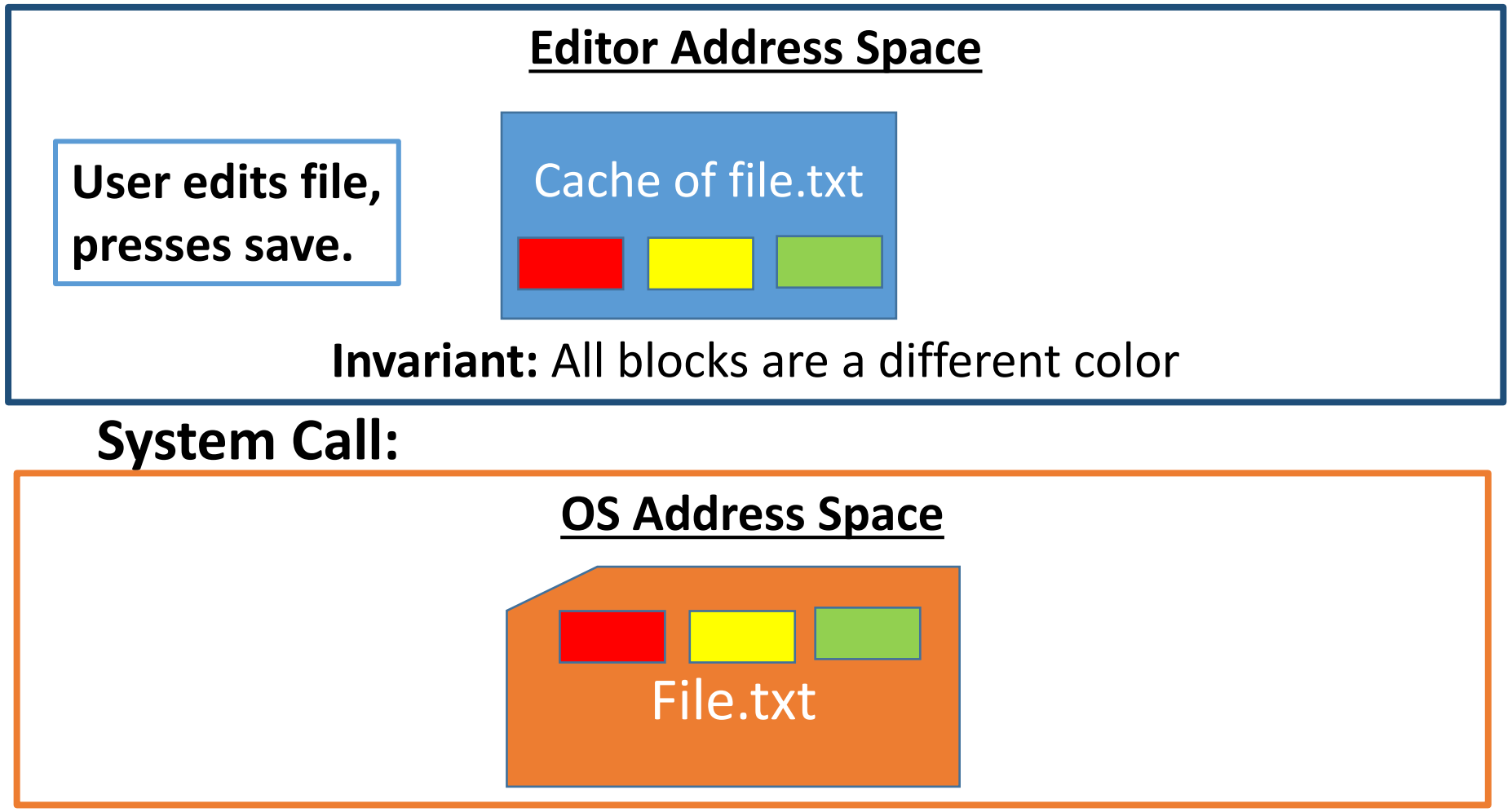
Applications implicitly keep state in OS too
TxOS: race conditions ***on application state*** in OS API

zed keyword,
a structures,

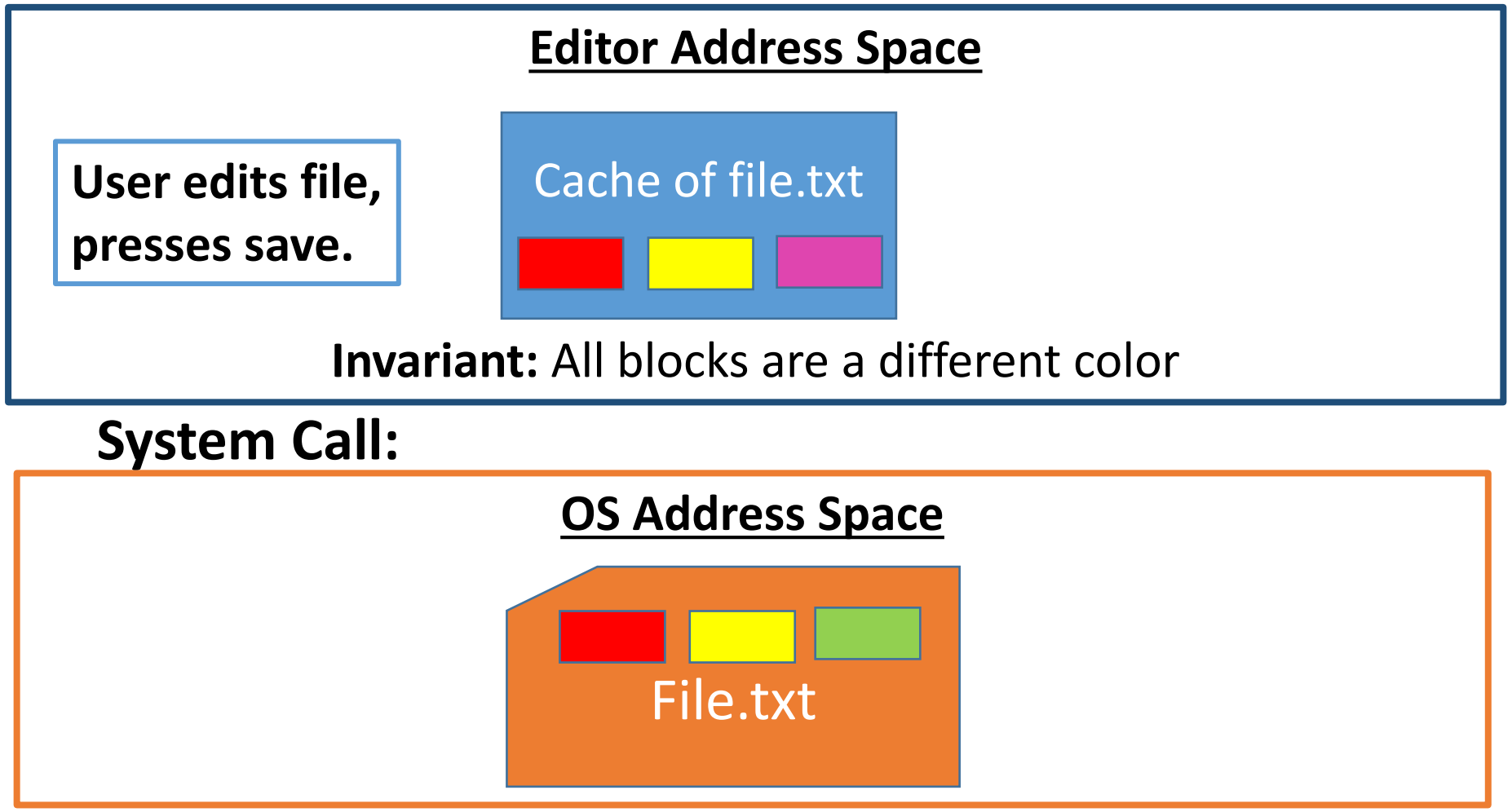
Example: Text editor



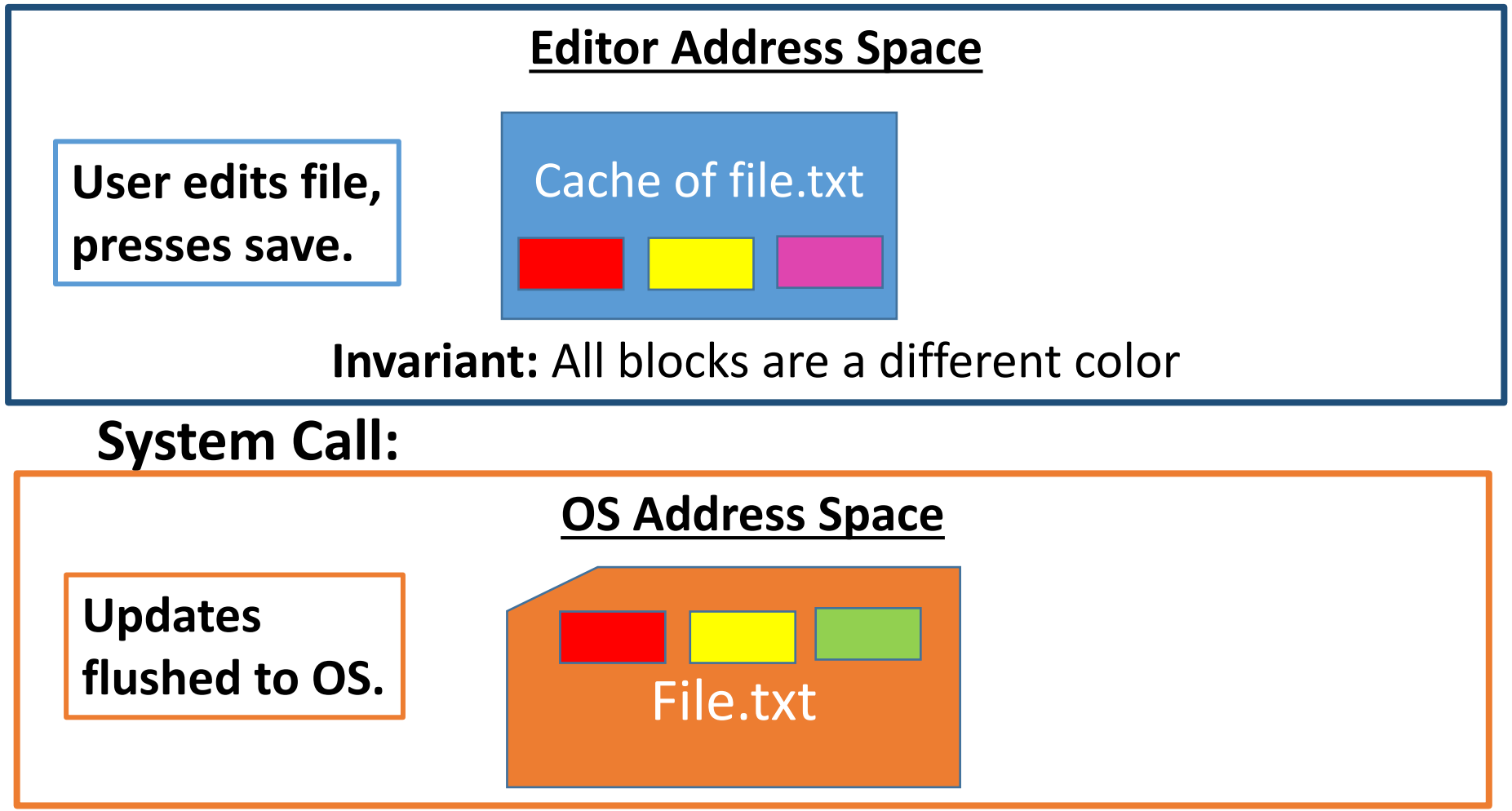
Example: Text editor



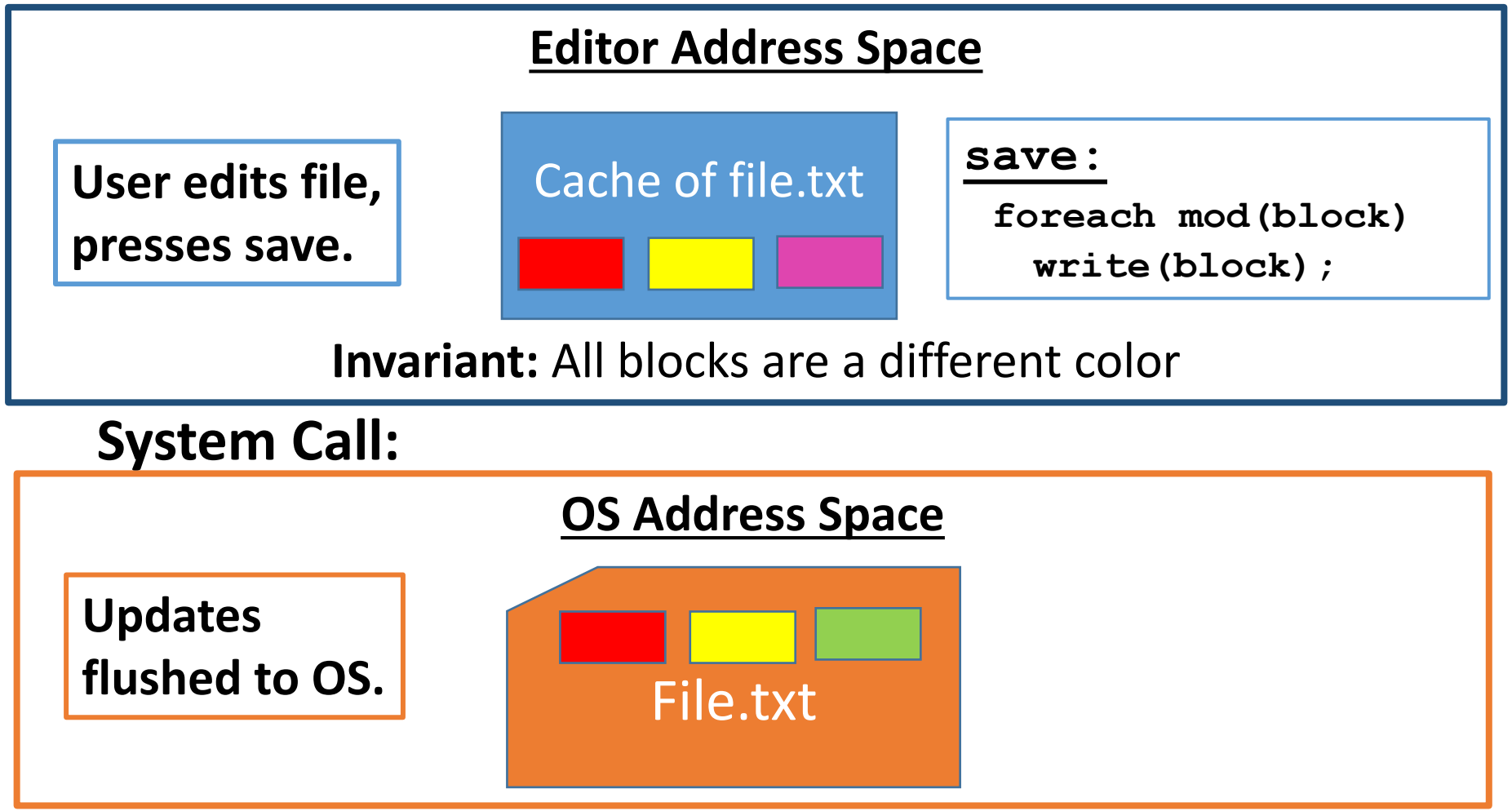
Example: Text editor



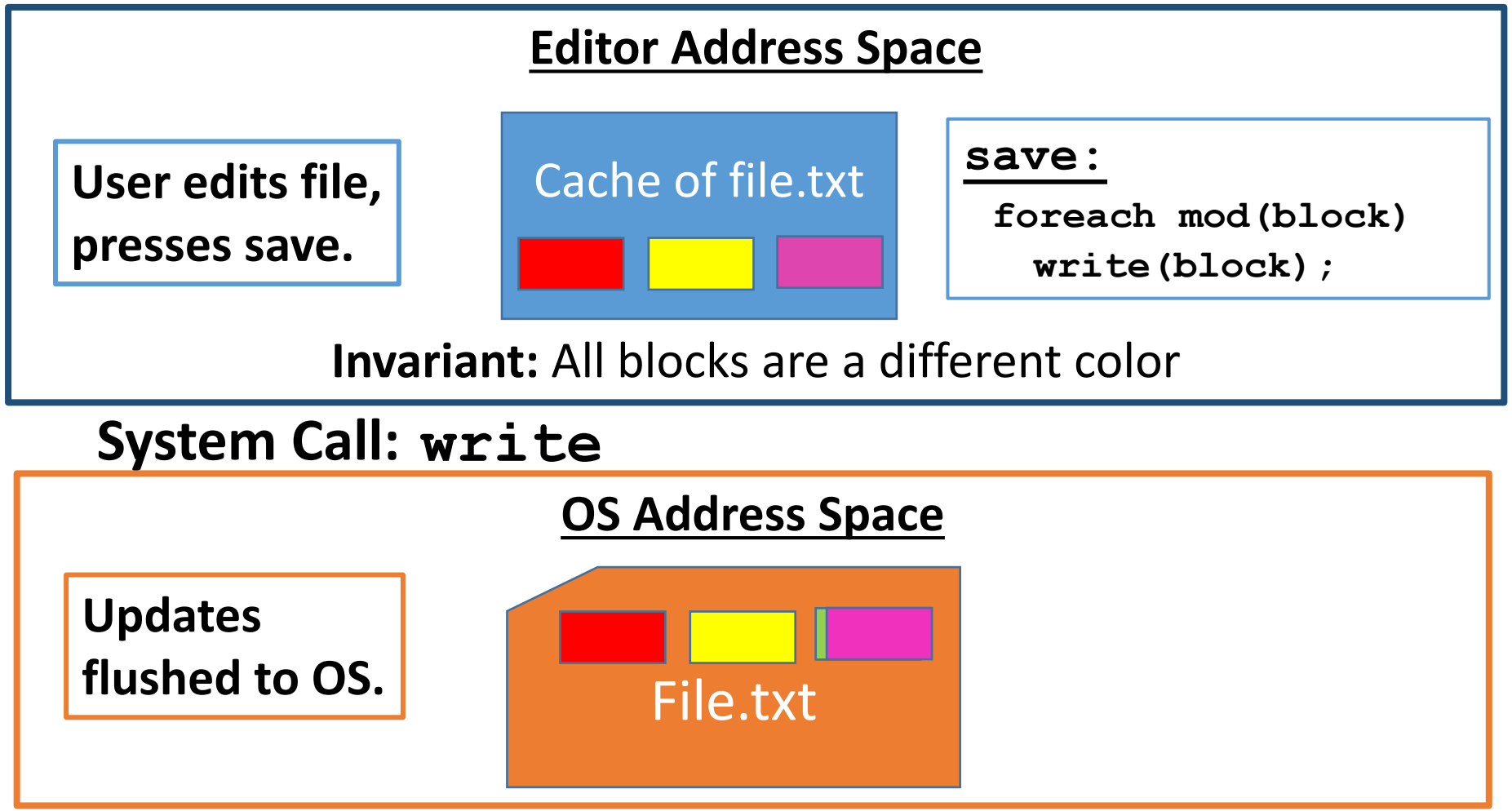
Example: Text editor



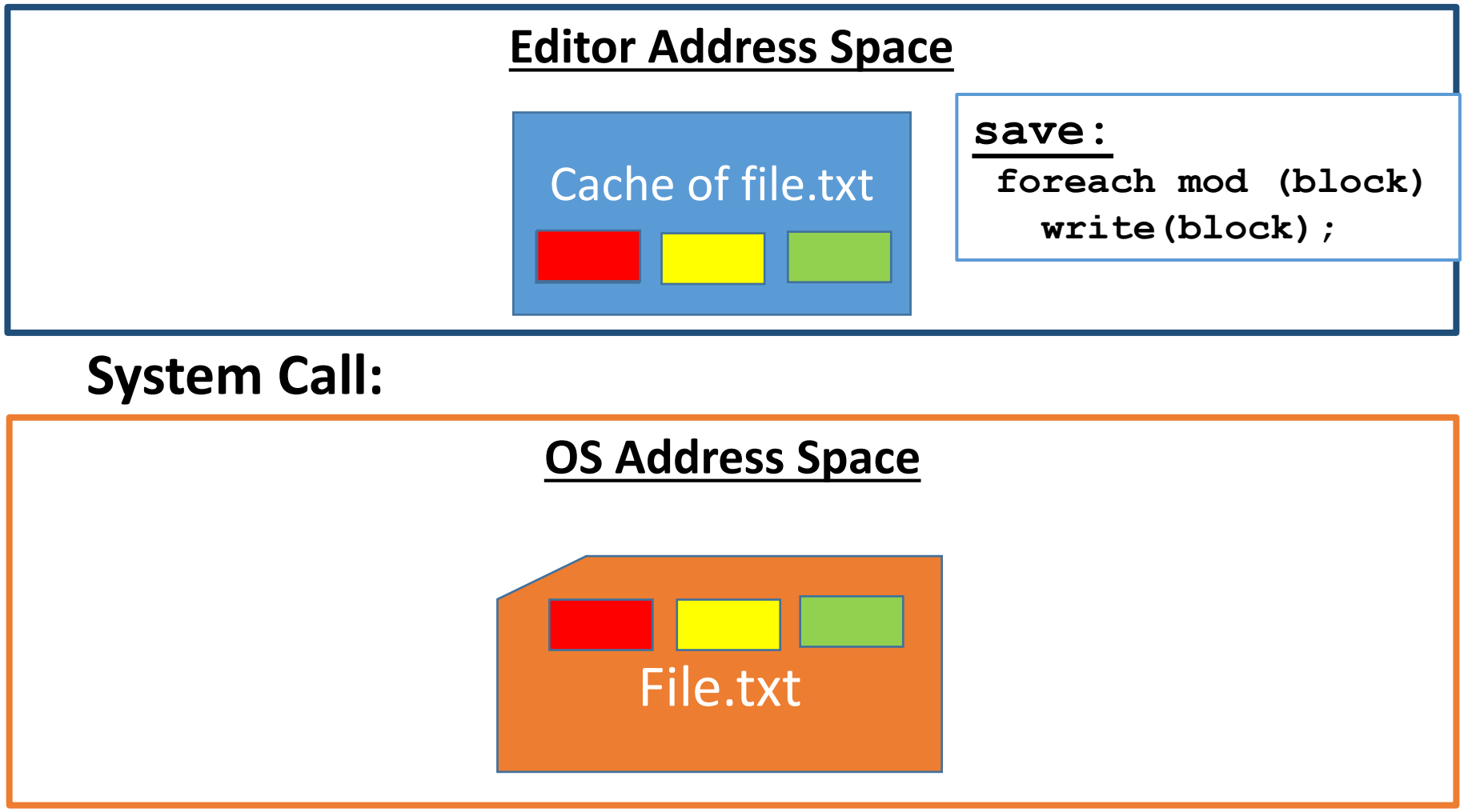
Example: Text editor



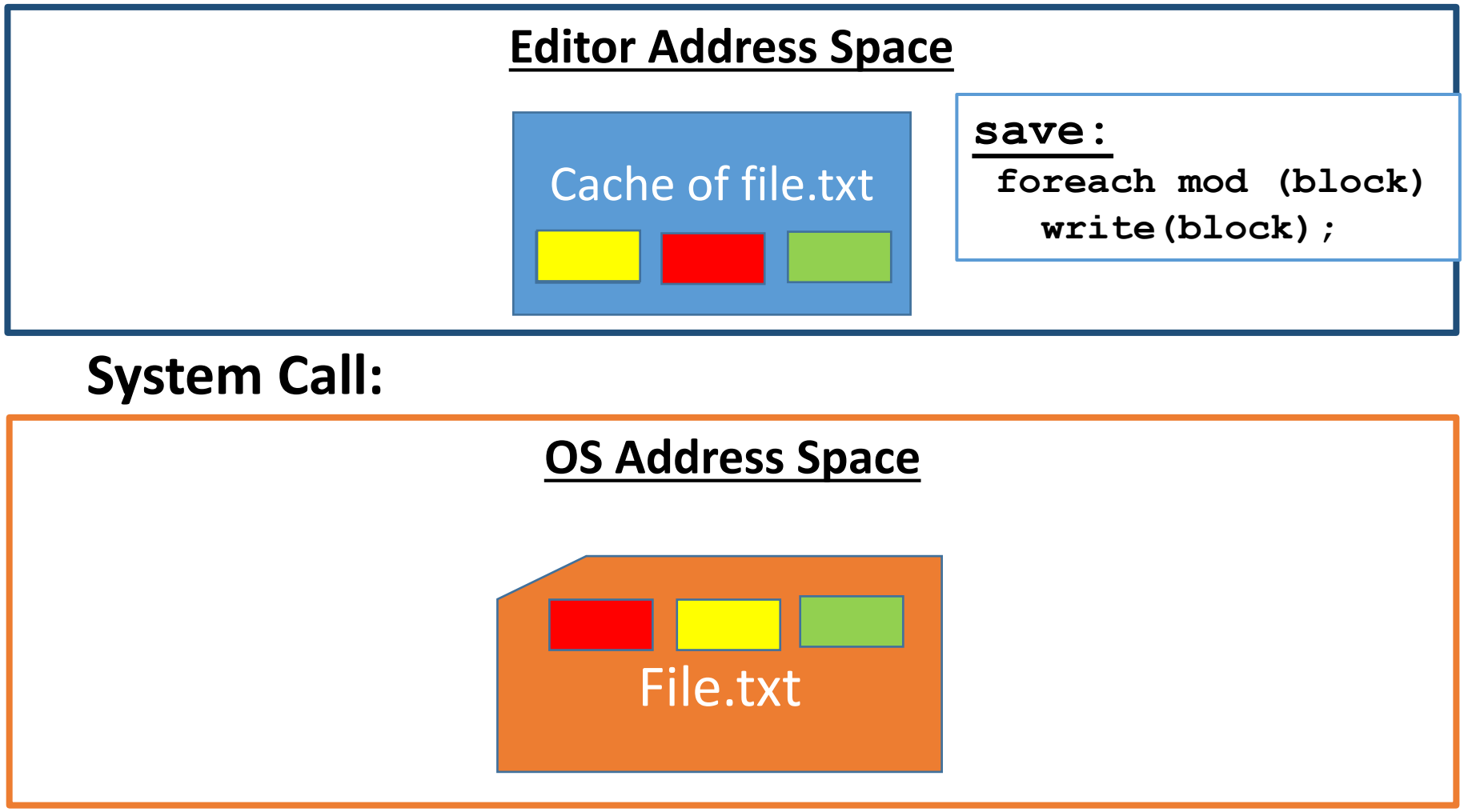
Example: Text editor



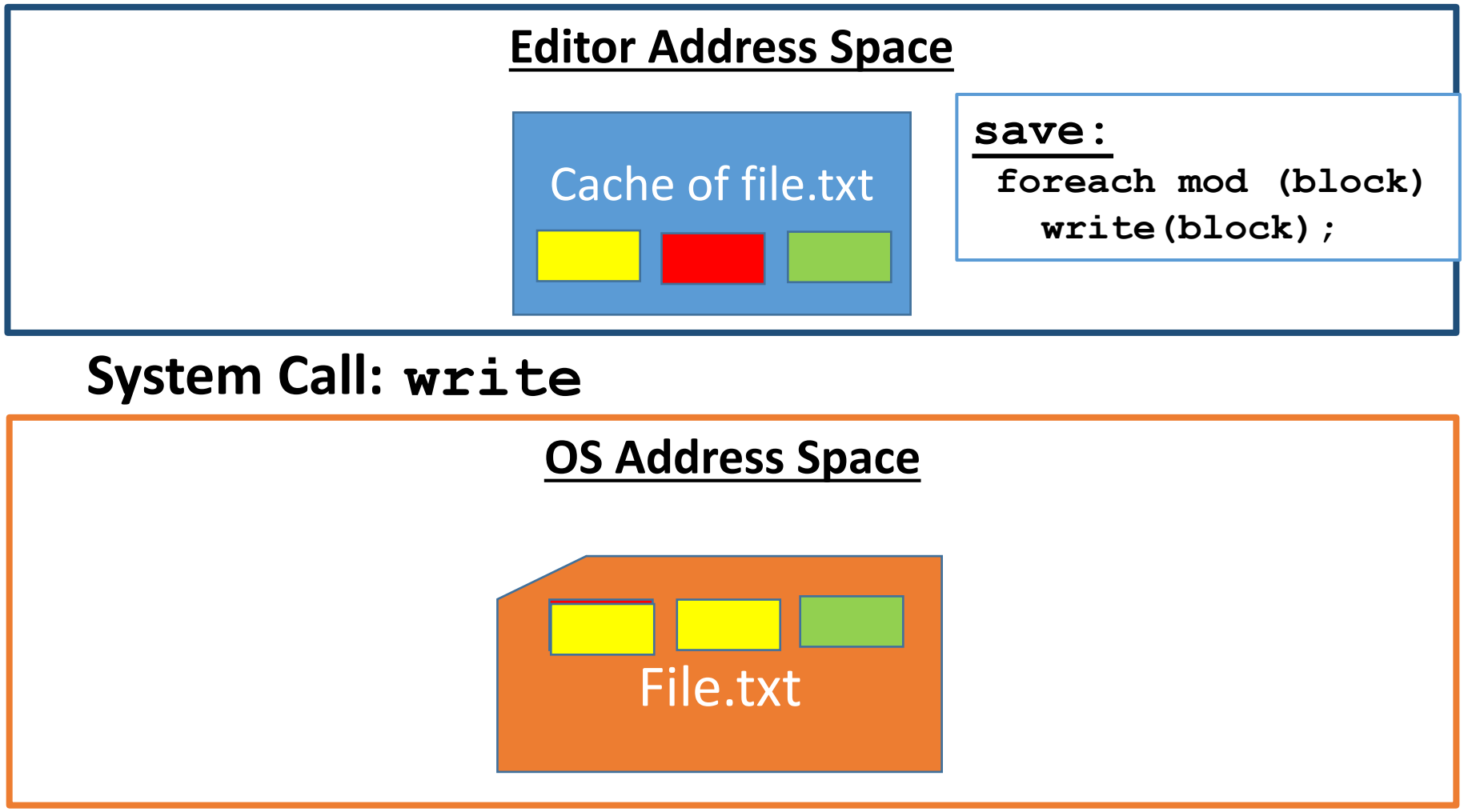
Saving two updates



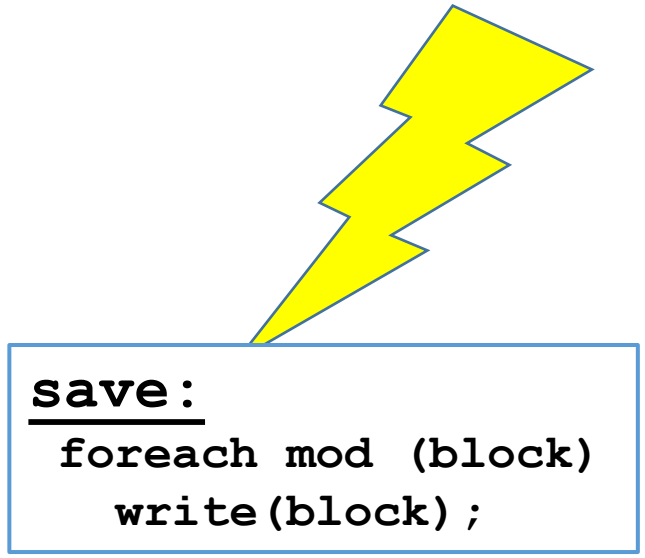
Saving two updates



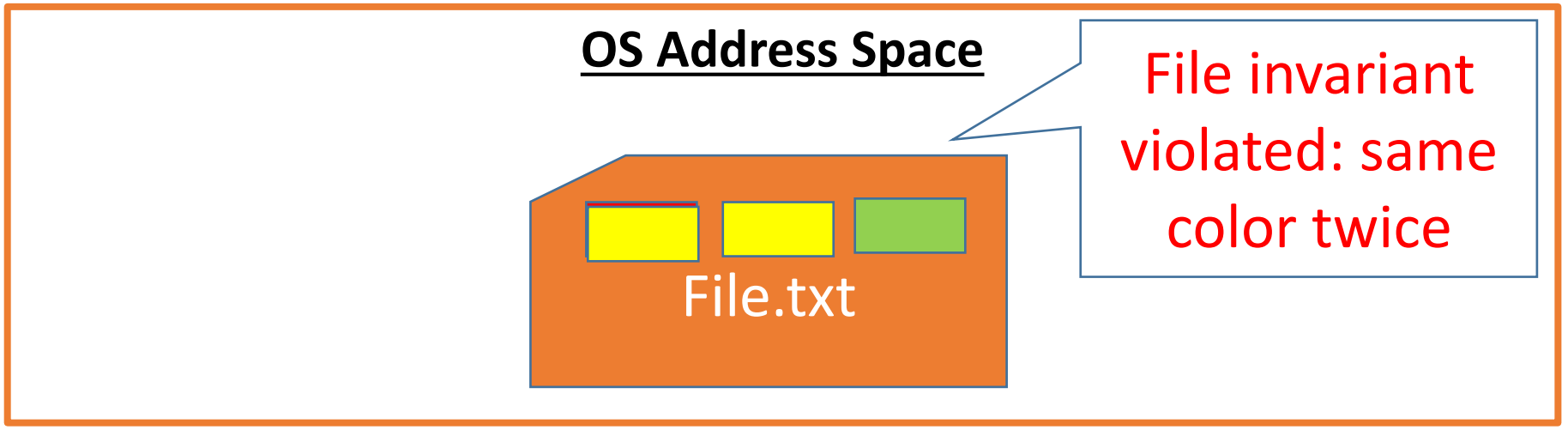
Saving two updates



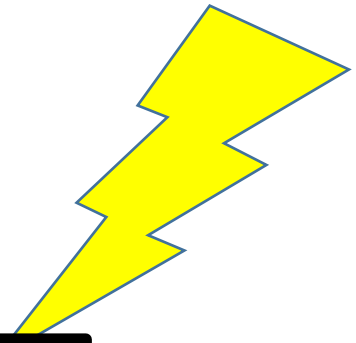
Saving two updates



System Call: `write`

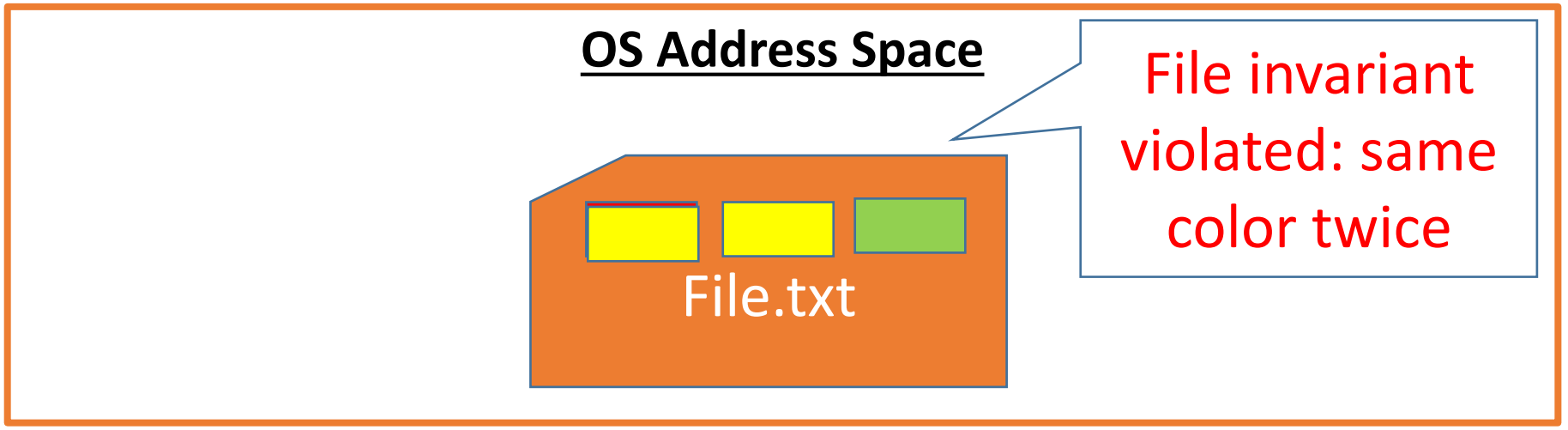


Saving two updates

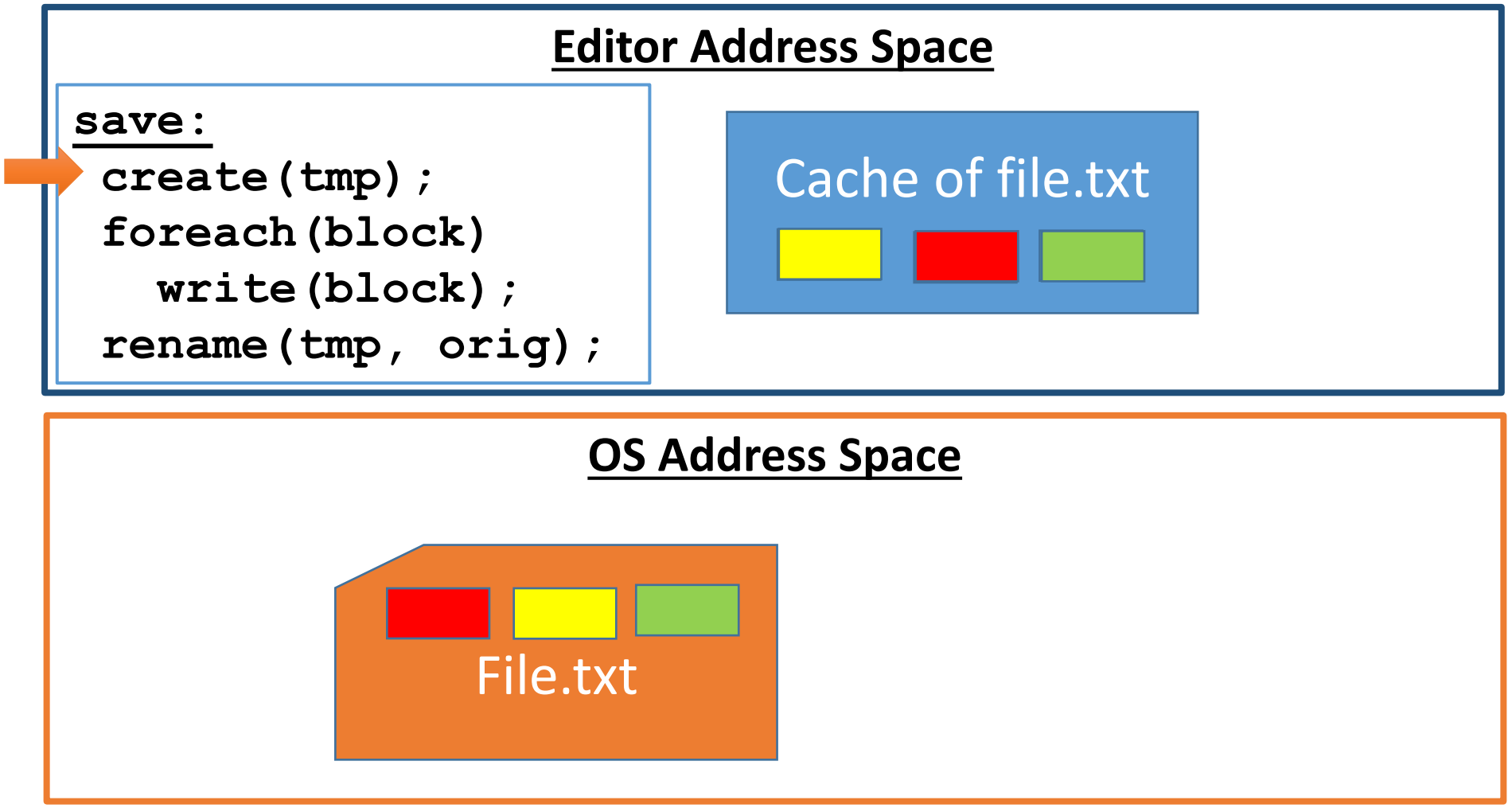


Desired property:
Atomicity: All updates stored or none.

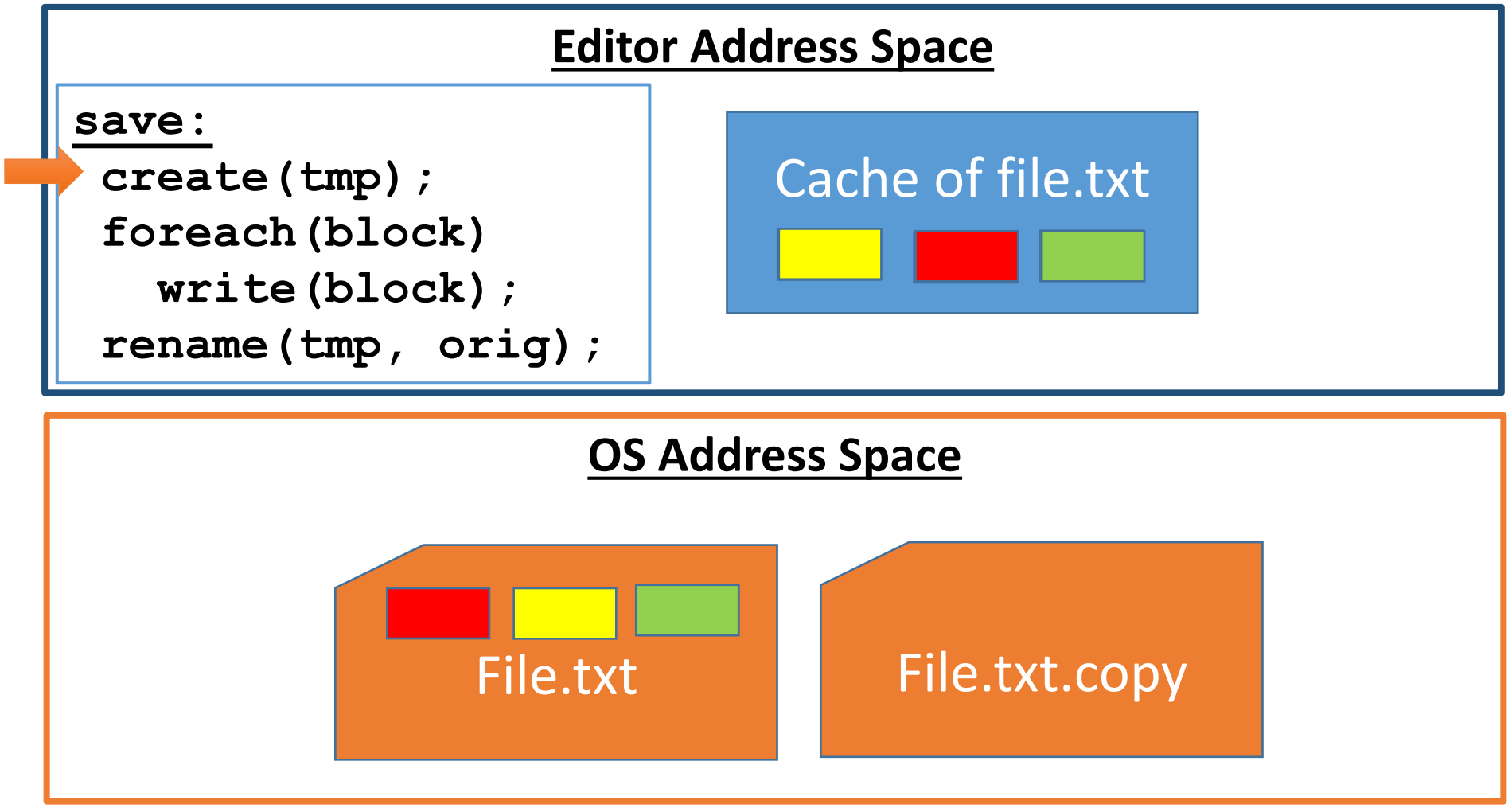
System Call: `write`



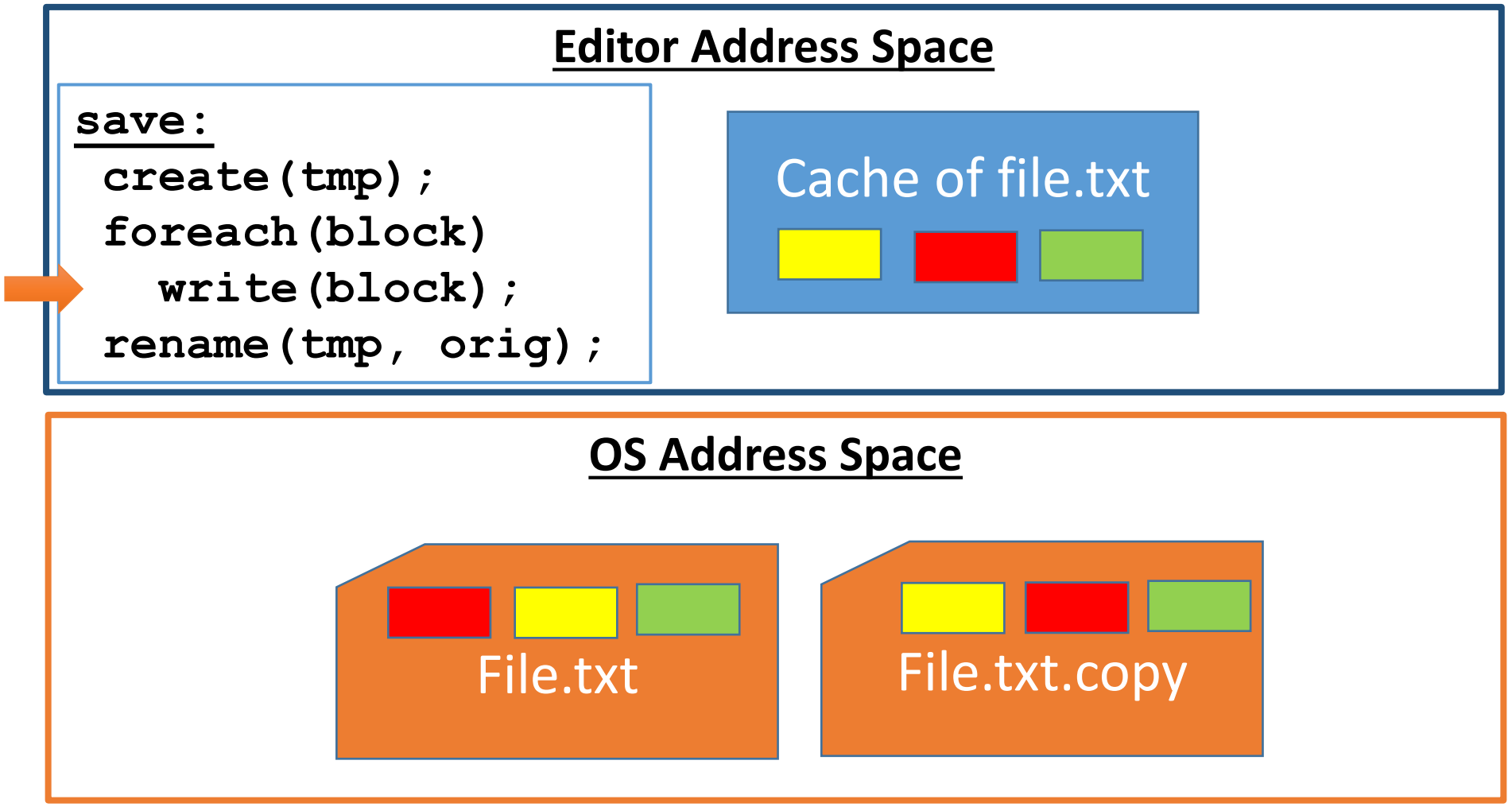
How does an editor really save a file?



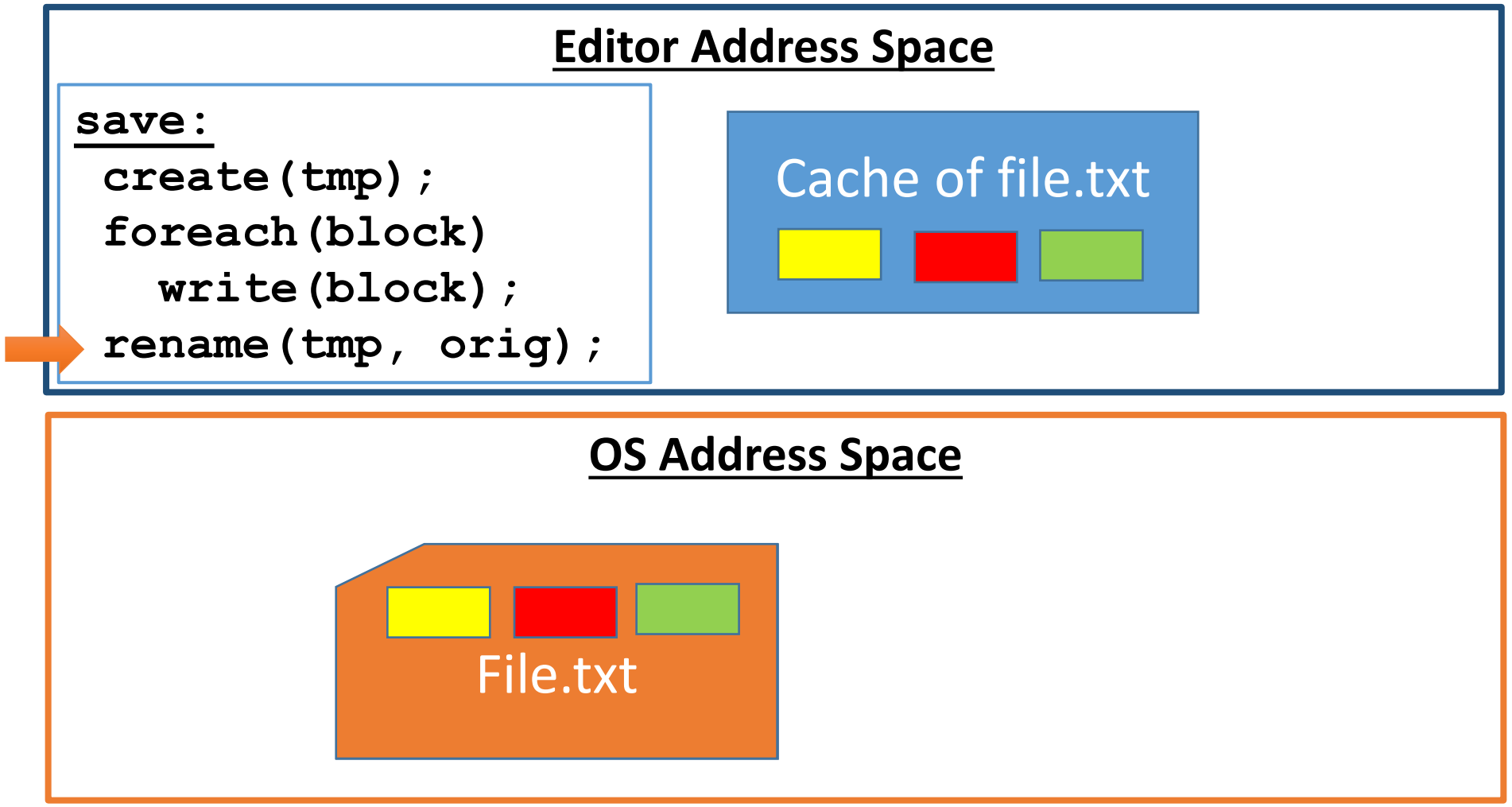
How does an editor really save a file?



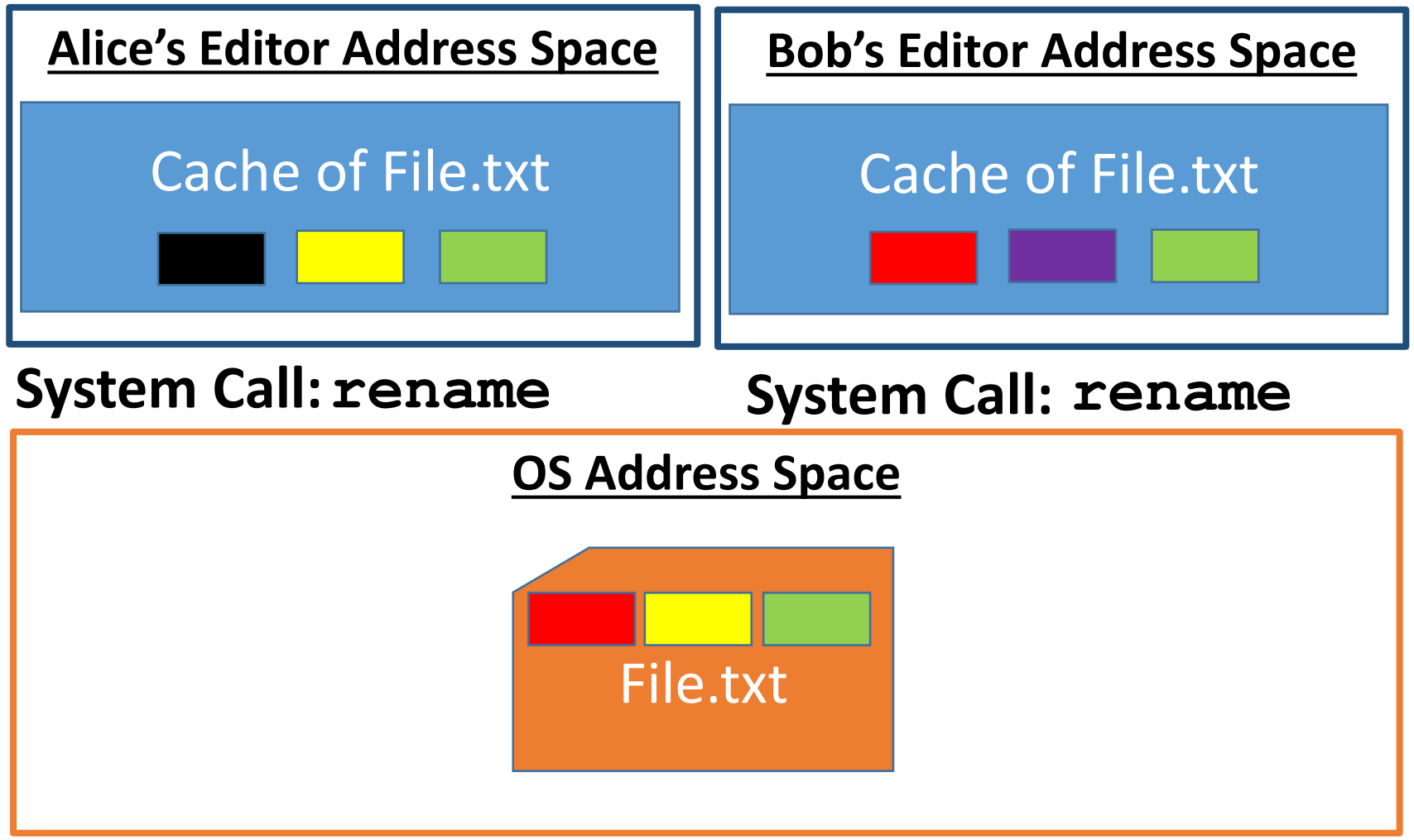
How does an editor really save a file?



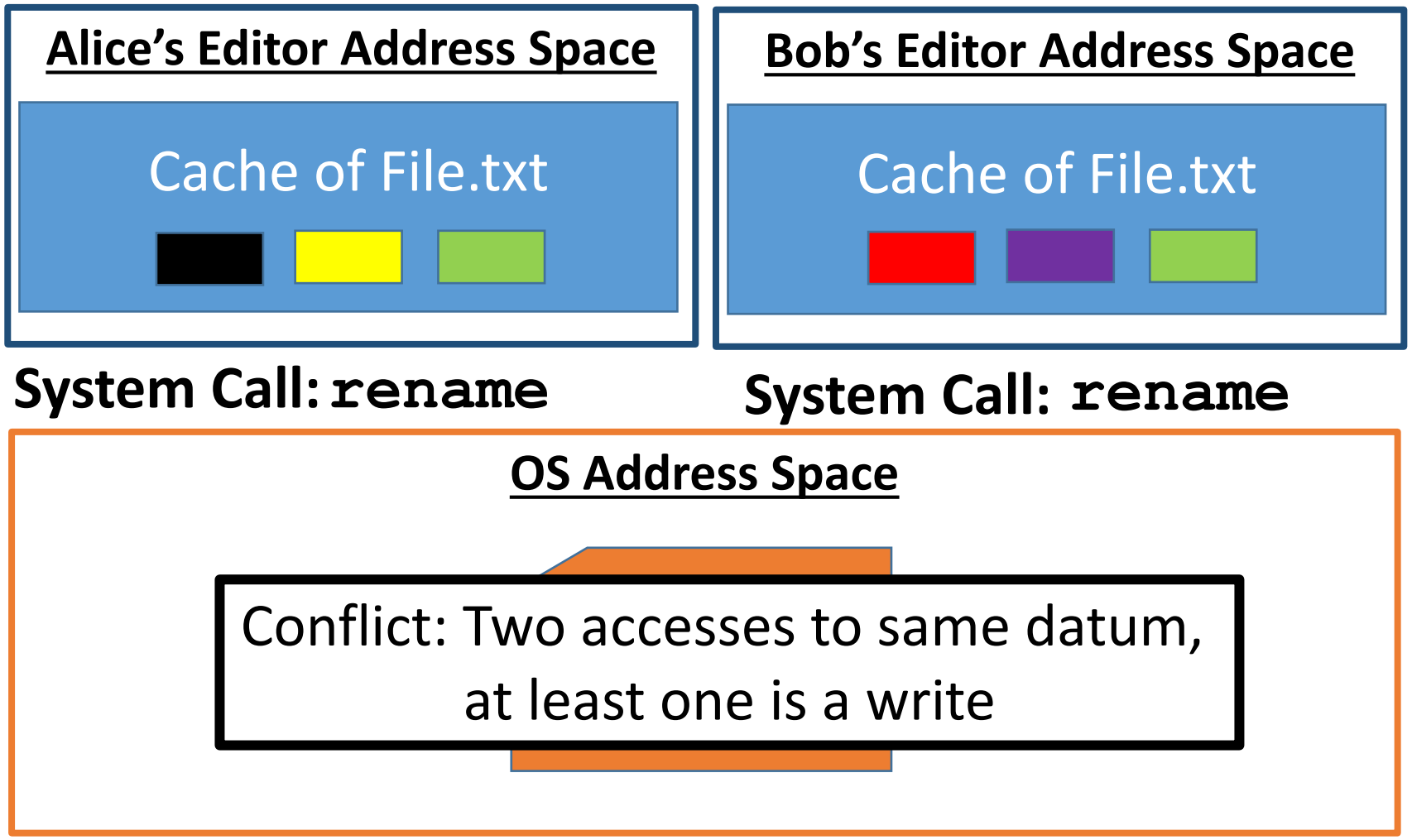
How does an editor really save a file?



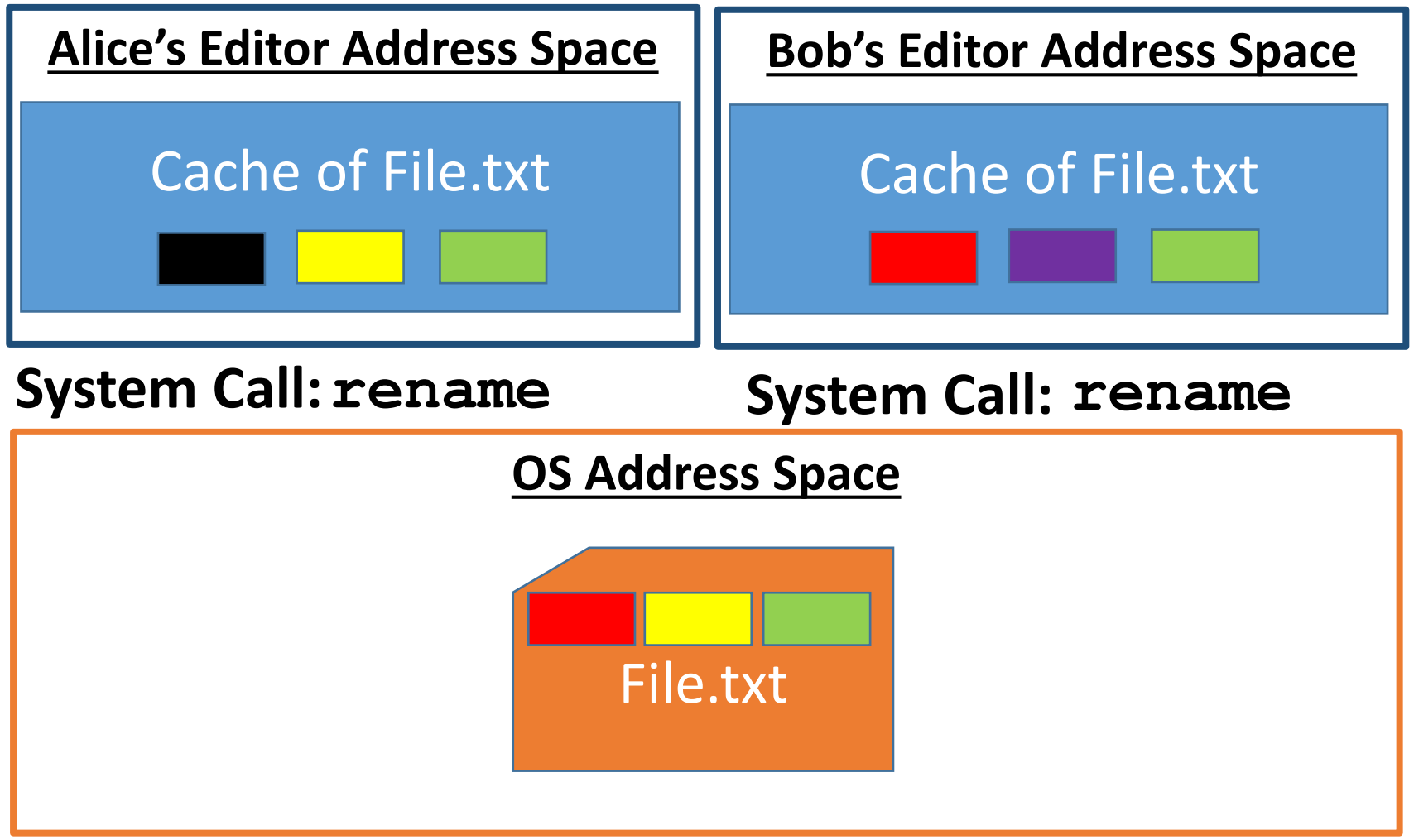
Inherent race condition in OS API



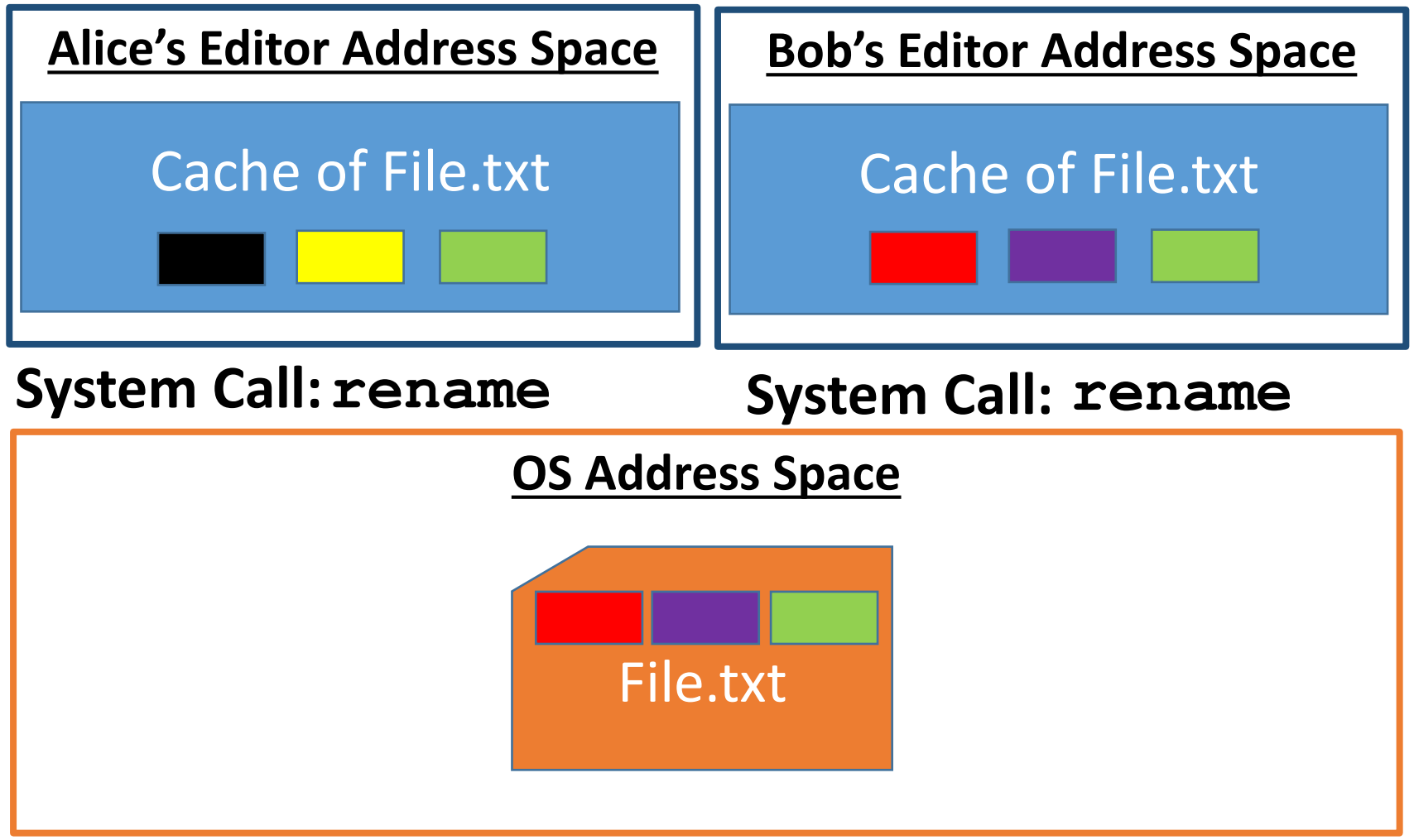
Inherent race condition in OS API



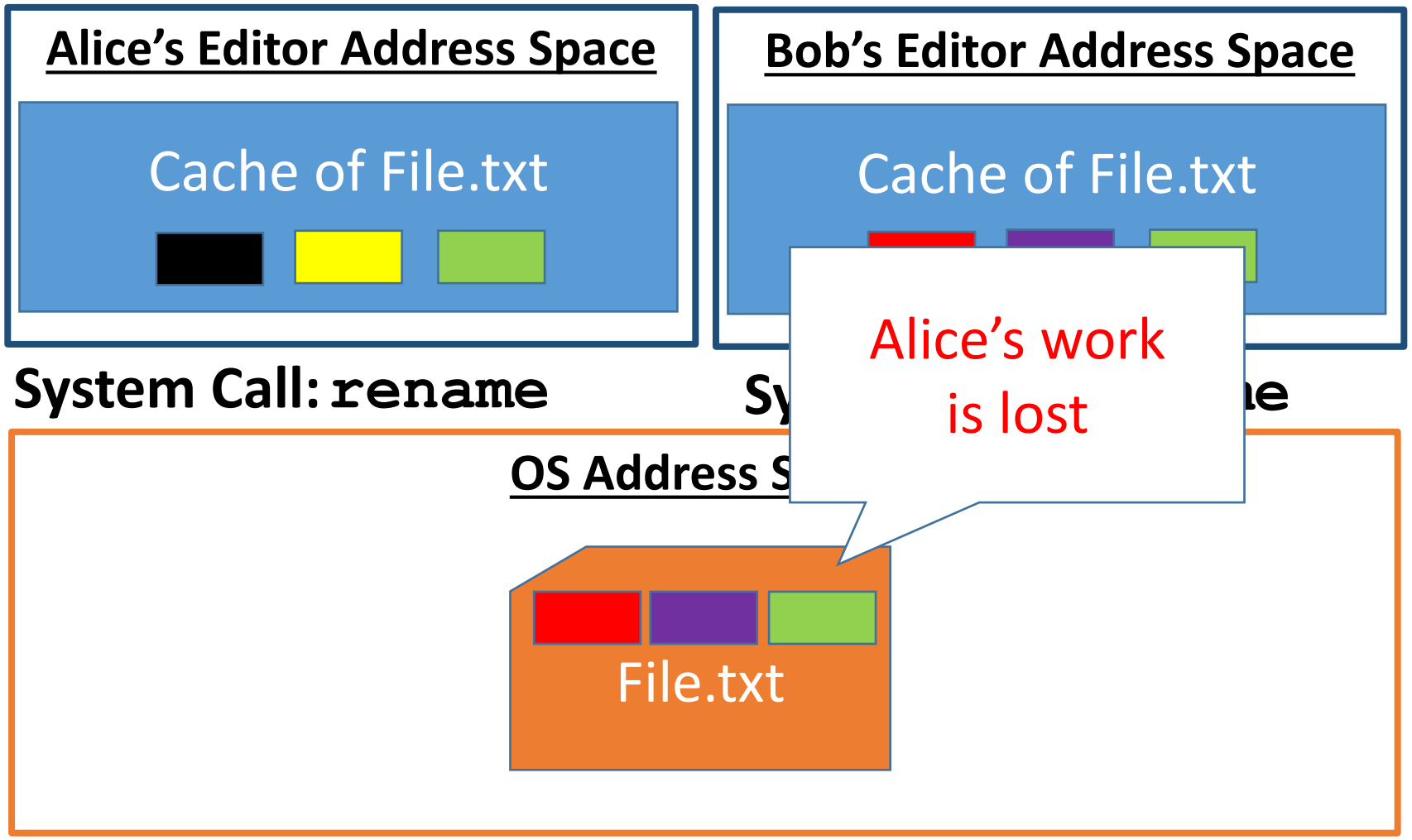
Inherent race condition in OS API



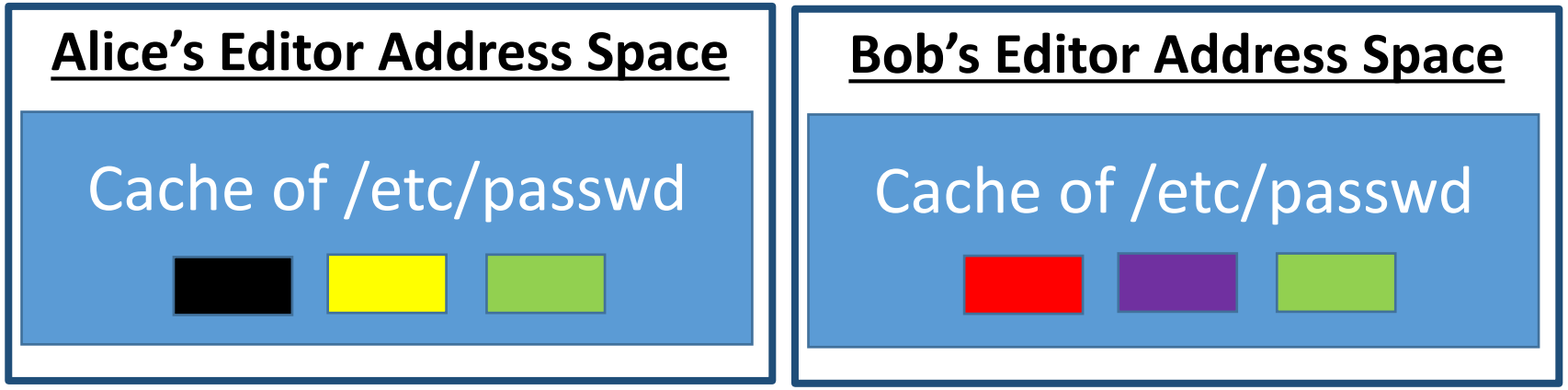
Inherent race condition in OS API



Inherent race condition in OS API



Real problem for system admins



- System administrators can race to save a system file
  - Can lose data
  - Can break invariants across configuration files
- Partial solutions exist; none fully work

Solution: System transactions

- The OS API needs a concurrency control abstraction
- Many potential solutions
  - Locking will not work
    - User-level locks not visible to OS
    - Users can’t be trusted with OS locks
- Transactions are a great success story in other layers of the technology stack

System transactions

- Simple API: **`sys_xbegin, sys_xend, sys_xabort`**

```
save:
  sys_xbegin();
  create(tmp);
  foreach(block)
    write(block);
  rename(tmp, orig);
  sys_xend();
```

- All system calls executed in system transaction take effect “at once.”

System transactions

- Simple API: **`sys_xbegin, sys_xend, sys_xabort`**

```
save:
  sys_xbegin();

  foreach(block)
    write(block);

  sys_xend();
```

- All system calls executed in system transaction take effect “at once.”

System transactions

- Transactions execute with ACID properties
  - **Atomicity**: Either all effects committed or none
  - **Consistency**: System state remains consistent
  - **Isolation**: All other operations appear to execute before or after a transaction
  - **Durability**: Committed operations are saved to non-volatile storage (where appropriate)

Transactional interfaces

Technology	Interface	Implementation
Database	Application to storage	Application
Distributed Transactions	Application to remote application	Application
Transactional FS	Application to subset of OS	OS module
Transactional Memory	Application to data structures	Hardware or user-level software
System Transactions	**Application to OS (FS, pipes, signals, etc.)**	**OS**

- System transactions enable consistent composition of system calls

Transactions solve a myriad of ills

- Reliable software
  - Transactional software installation
- Efficient software
  - Lightweight alternative to a database
- Correct software
  - Eliminate race conditions
  - 600+ instances of race conditions leading to security compromises in National Vulnerability Database
- The time has come for system transactions

System races undermine security

(root)

```
if(access("foo")) {

   fd = open("foo");
   write(fd,…);
   …
}
```

Time-of-check-to-time-of-use (TOCTTOU) race condition [McPhee 1974]

System races undermine security

(root)

```
if(access(“foo”)) {
        symlink(“/etc/passwd”, “foo”);
   fd = open(“foo”);
   write(fd,…);
   …
}
```

Time-of-check-to-time-of-use (TOCTTOU) race condition [McPhee 1974]

System races undermine security

(root)

```
if(access(“foo”)) {
      symlink(“/etc/passwd”, “foo”);
   fd = open(“foo”);
   write(fd,…);
   …
}
```

foo == /etc/passwd

Time-of-check-to-time-of-use (TOCTTOU) race condition [McPhee 1974]

TOCTTOU is a hard problem

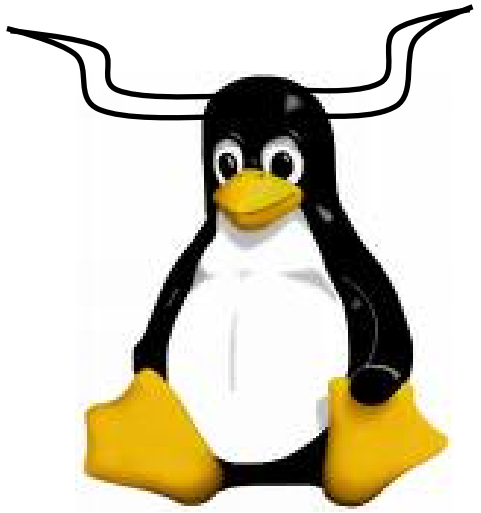
- Conceptually simple
  - 600+ instances in National Vulnerability Database
  - Often addressed by complex hacks or **removing functionality** from the application
- No portable, deterministic solution with current POSIX API – Dean and Hu 2004
- Ongoing arms race
  - Tsafrir et al.'s defense won best paper at FAST '08
  - Cai et al. defeated this defense and others in '09

Outline

- **TxOS design and implementation**
- Evaluation and applications
- Related work
- Ongoing and future work

TxOS prototype

- Extend Linux 2.6.22 to support system transactions
  - Add 8,600 lines of code to Linux
  - Minor modifications to 14,000 lines of code
- Runs on commodity hardware
- Transactional semantics for a range of resources:
  - File system, signals, processes, pipes
- More information in papers
  - [HotOS ’09, SOSP ‘09]



Building a transactional system

- **Detect conflicts**
  - Safety: Detect and serialize all conflicting transactions
  - Performance: Don’t serialize non-conflicting transactions
- Version management
  - Private copies instead of undo log
- All code must respect transaction isolation
  - Legacy code does not use transactions

Building a transactional system

- **Detect conflicts**
  - Safety: Detect and serialize all conflicting transactions
  - Performance: Don’t serialize non-conflicting transactions
- Version management
  - Private copies instead of undo log
- All code must respect transaction isolation
  - Legacy code does not use transactions

Wait…what?
Detect conflicts?
Version management?
Why are we talking about this?

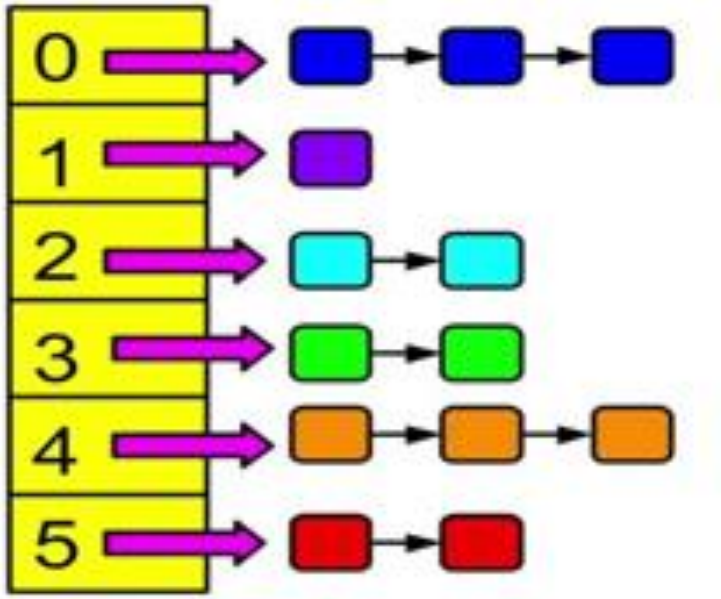
Let’s talk concurrency control

Let’s talk concurrency control

Consider a hash-table

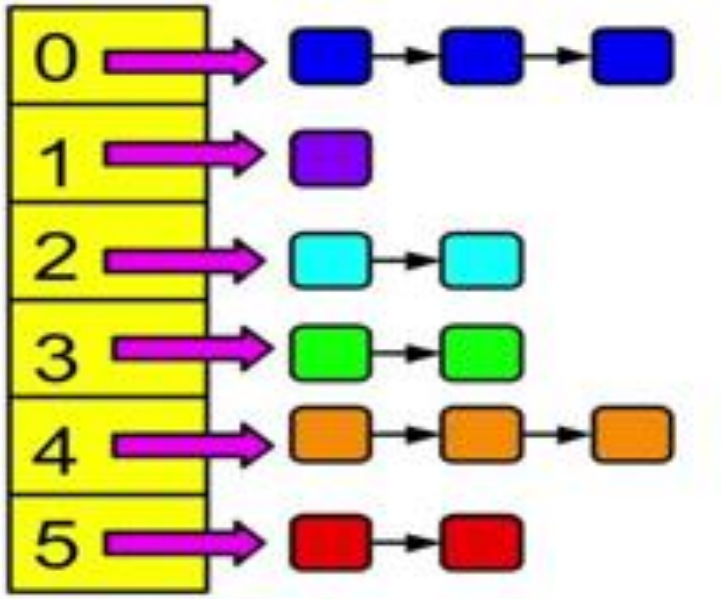
Let’s talk concurrency control

Consider a hash-table



Let’s talk concurrency control

Consider a hash-table



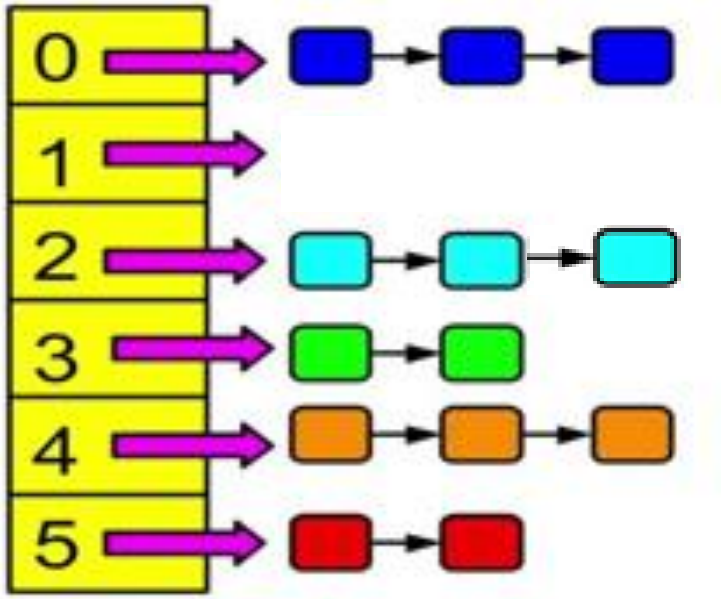
thread T1

```
ht.add( );

if(ht.contains( ))
    ht.del( );
```

Let’s talk concurrency control

Consider a hash-table



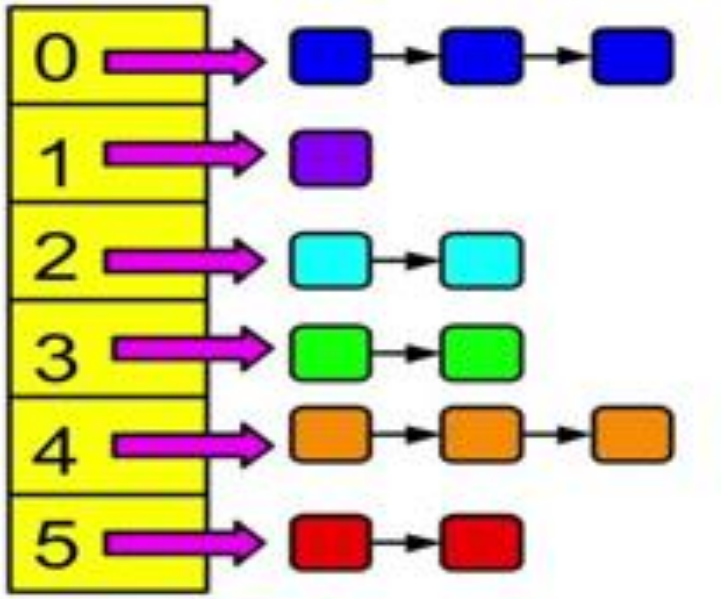
thread T1

```
ht.add( );

if(ht.contains( ))
    ht.del( );
```

Let’s talk concurrency control

Consider a hash-table



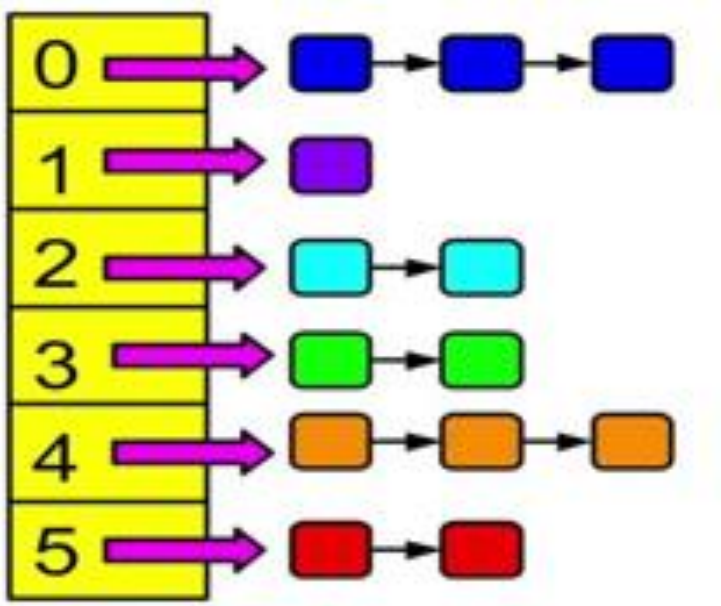
thread T1

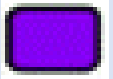
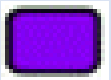
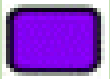
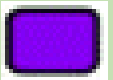
```
ht.add( );

if(ht.contains( ))
    ht.del( );
```

Let’s talk concurrency control

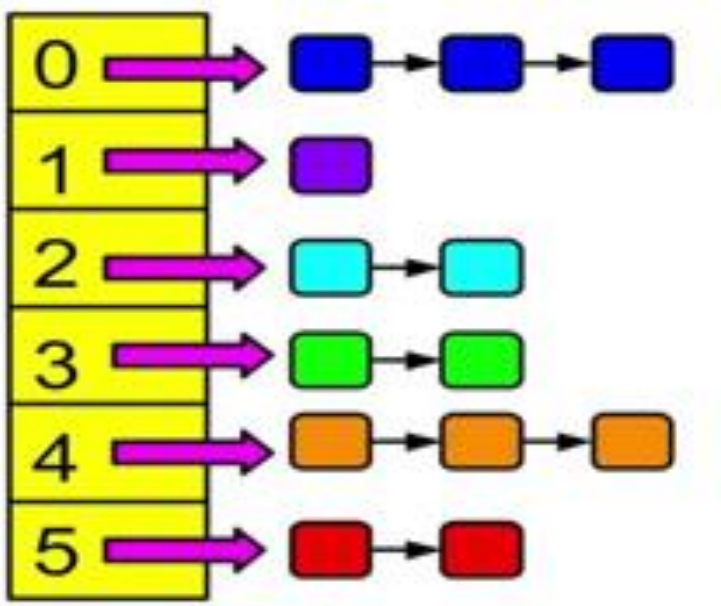
Consider a hash-table



thread T1	thread T2
`ht.add(`  `);`	`ht.add(`  `);`
`if(ht.contains(`  `))` `ht.del(`  `);`	`if(ht.contains(`  `))` `ht.del(`  `);`

Let’s talk concurrency control

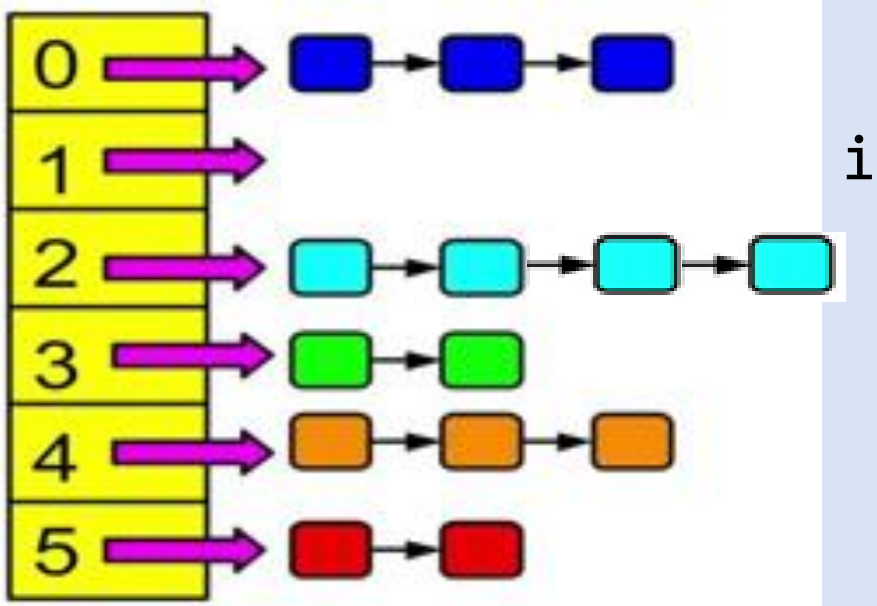
Consider a hash-table



thread T1	thread T2
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );

Let’s talk concurrency control

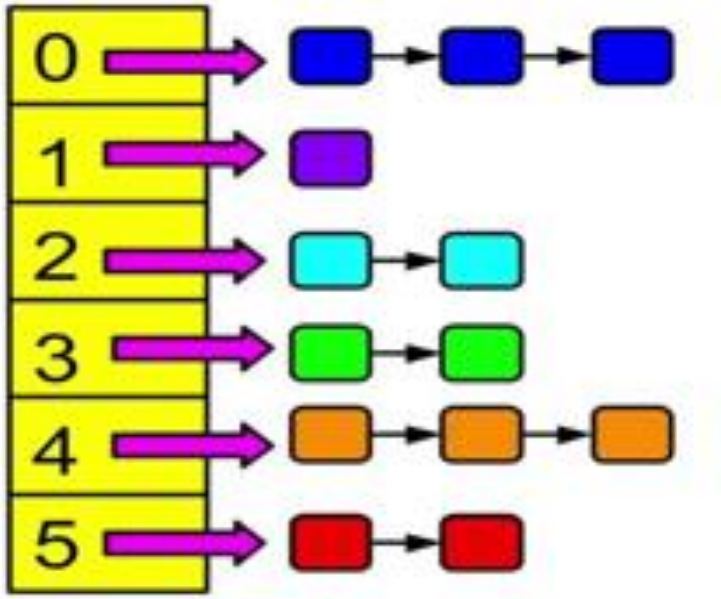
Consider a hash-table



thread T1	thread T2
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );

Pessimistic concurrency control: coarse locks

Consider a hash-table



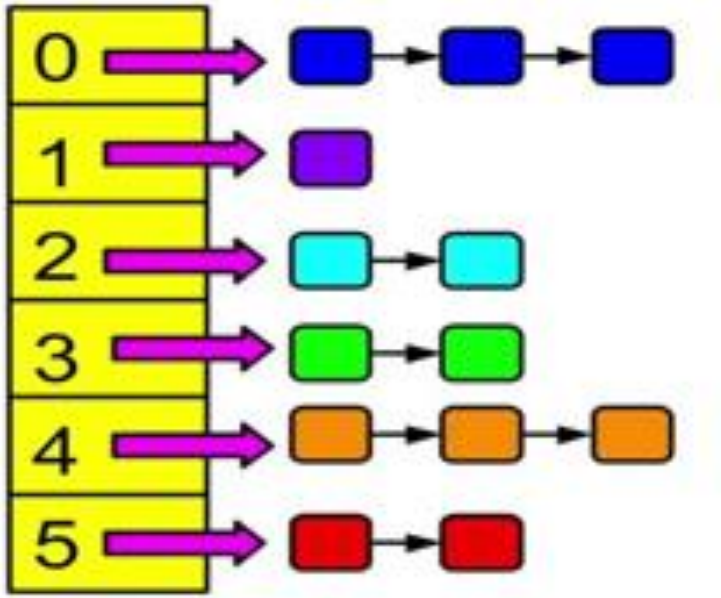
thread T1

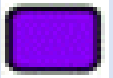
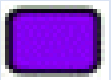
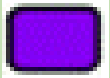
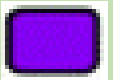
```
ht.add( );

if(ht.contains( ))
    ht.del( );
```

Pessimistic concurrency control: coarse locks

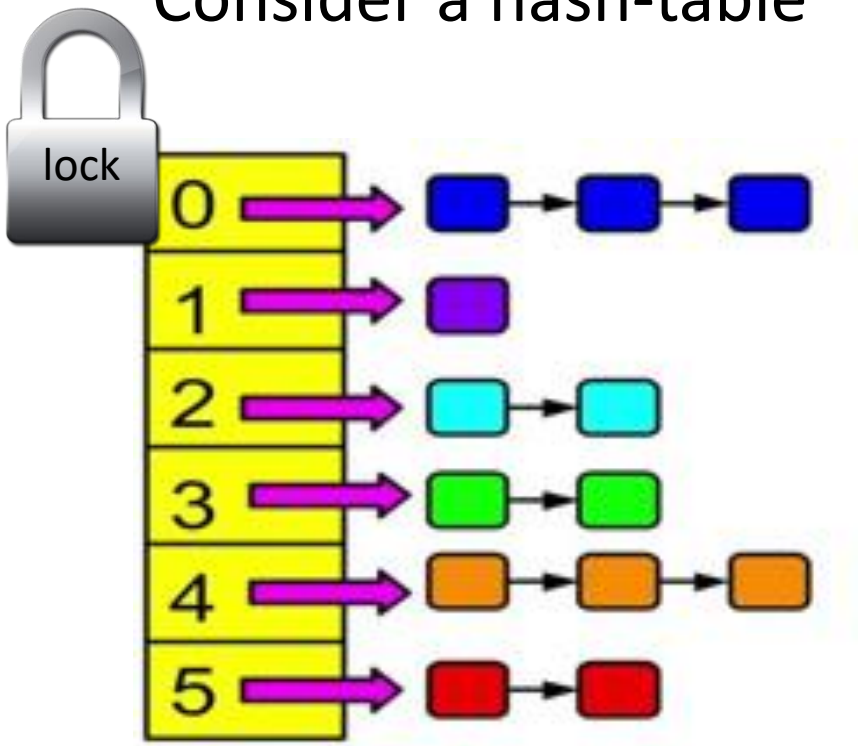
Consider a hash-table

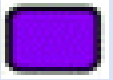
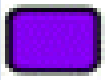
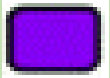
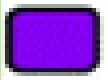


thread T1	thread T2
`ht.add(`  `);`	`ht.add(`  `);`
`if(ht.contains(`  `))` `ht.del(`  `);`	`if(ht.contains(`  `))` `ht.del(`  `);`

Pessimistic concurrency control: coarse locks

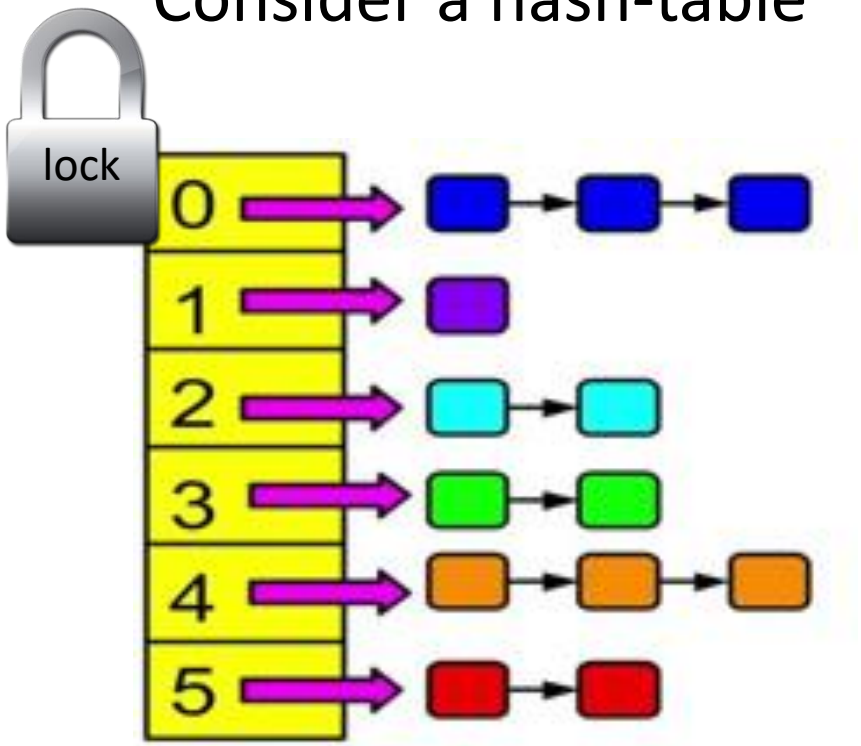
Consider a hash-table



thread T1	thread T2
`ht.add(`  `);`	`ht.add(`  `);`
`if(ht.contains(`  `))` `ht.del(`  `);`	`if(ht.contains(`  `))` `ht.del(`  `);`

Pessimistic concurrency control: coarse locks

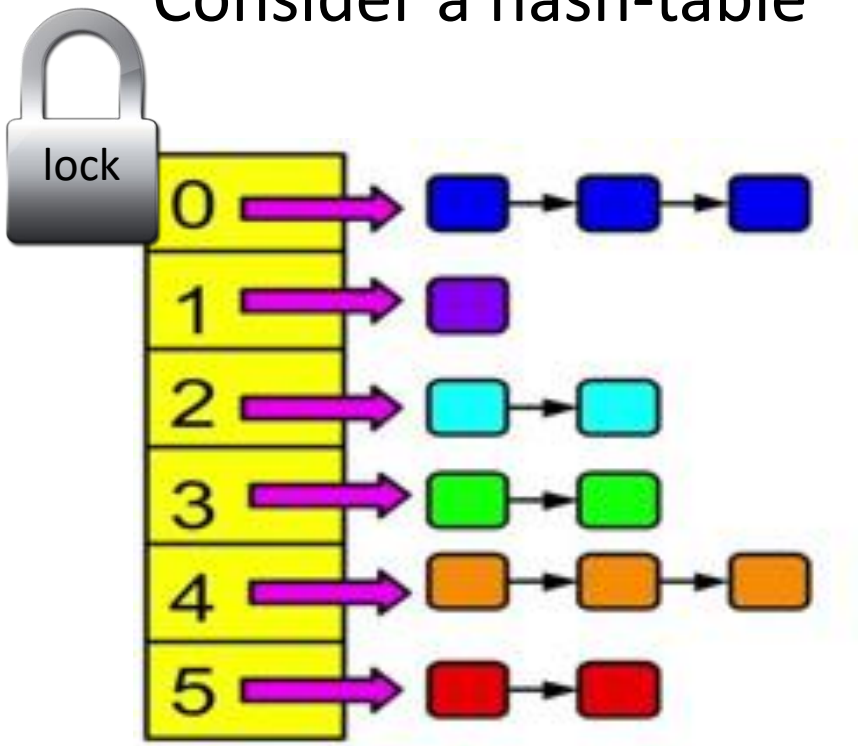
Consider a hash-table



thread T1	thread T2
ht.lock();	
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );

Pessimistic concurrency control: coarse locks

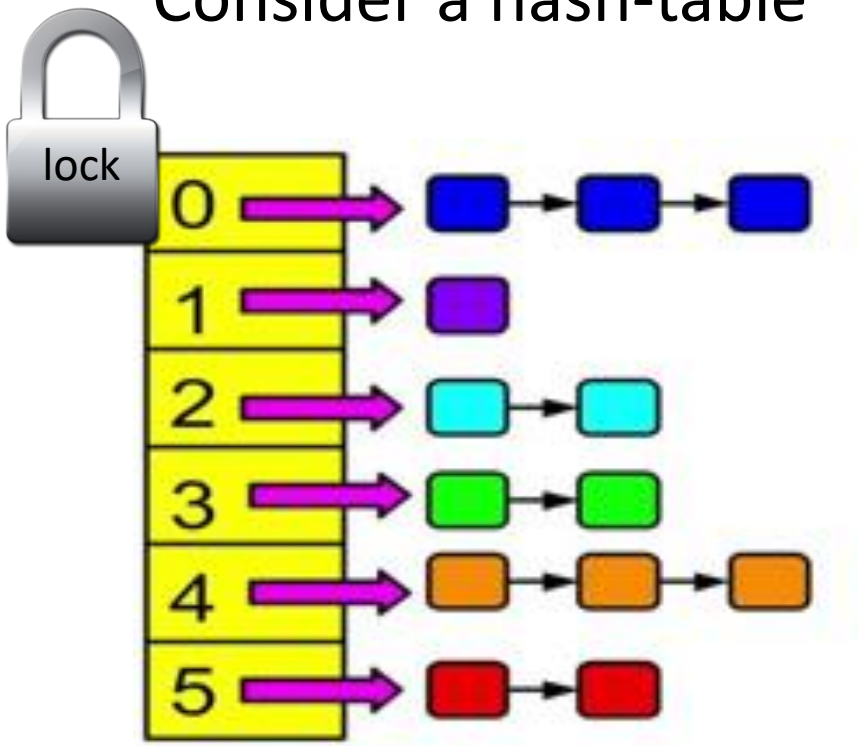
Consider a hash-table



thread T1	thread T2
ht.lock();	
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
ht.unlock();	

Pessimistic concurrency control: coarse locks

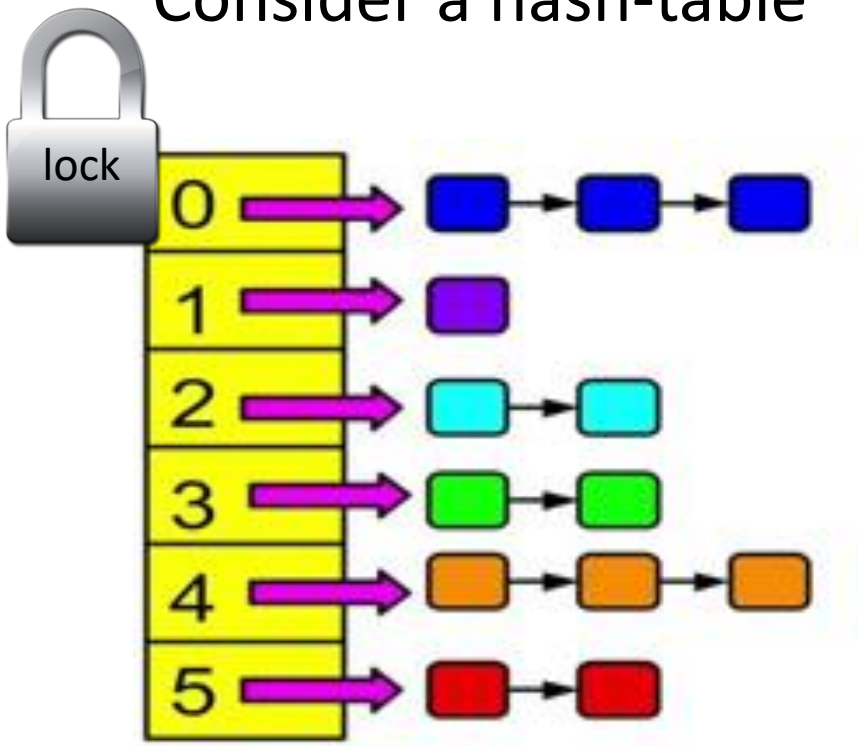
Consider a hash-table



thread T1	thread T2
ht.lock();	**ht.lock();**
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
ht.unlock();	

Pessimistic concurrency control: coarse locks

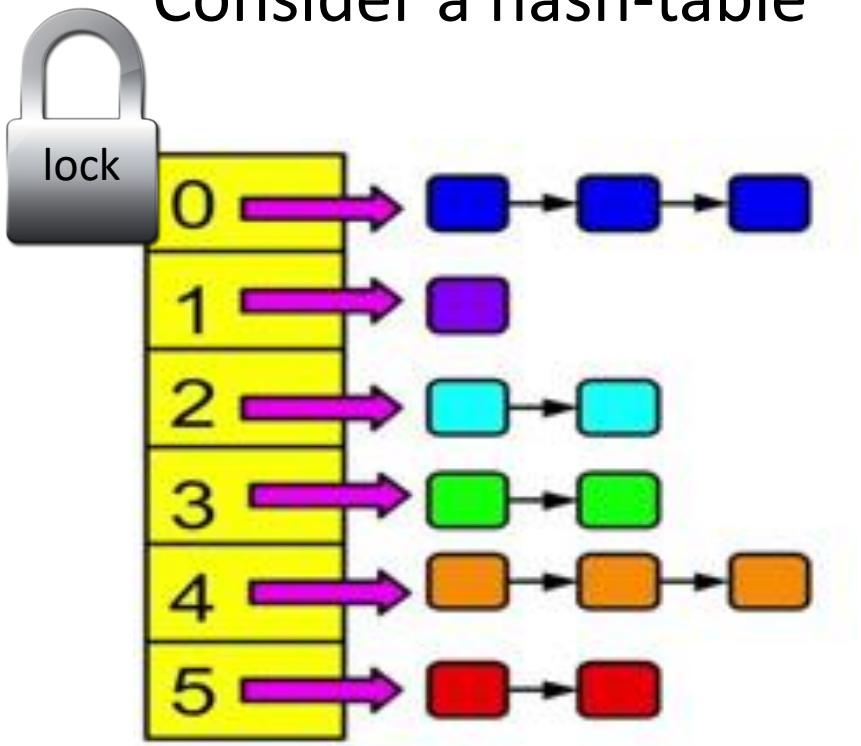
Consider a hash-table



thread T1	thread T2
ht.lock();	**ht.lock();**
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
ht.unlock();	**ht.unlock();**

Pessimistic concurrency control: coarse locks

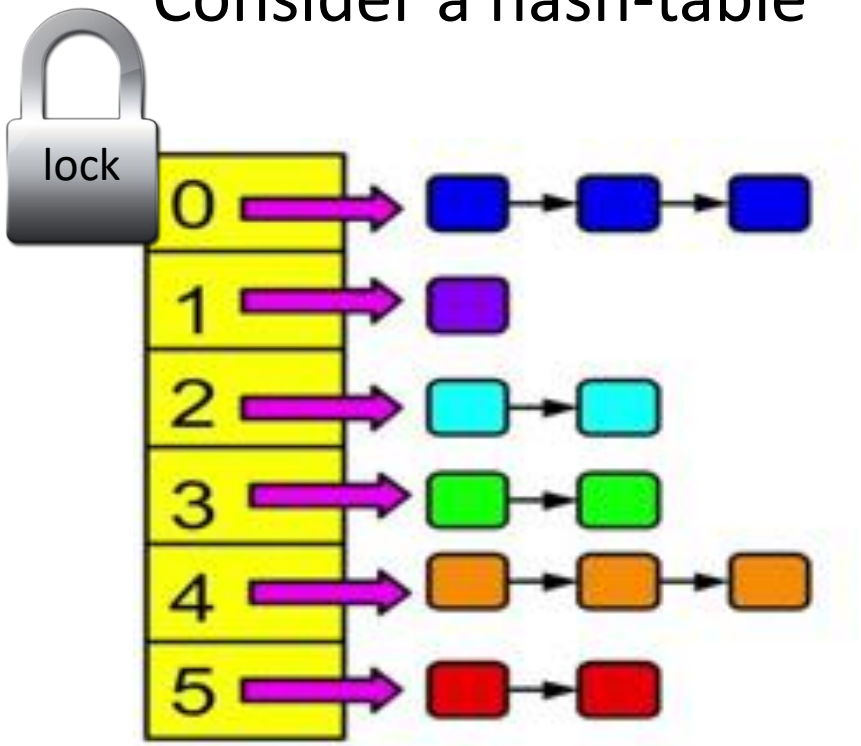
Consider a hash-table



thread T1	thread T2
ht.lock();	**ht.lock();**
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
ht.unlock();	**ht.unlock();**

Pessimistic concurrency control: coarse locks

Consider a hash-table

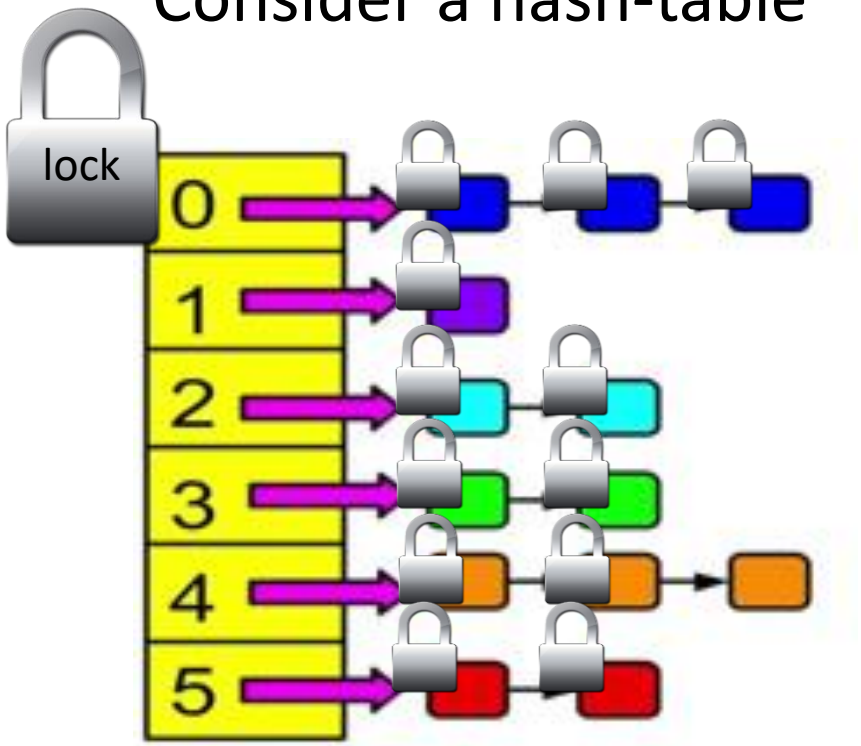


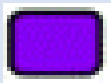
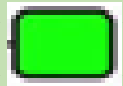
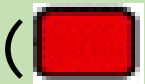
thread T1	thread T2
ht.lock();	**ht.lock();**
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
ht.unlock();	**ht.unlock();**

Coarse lock:
Non-conflicting ops serialized
Low Complexity -- Low Performance

Pessimistic concurrency control: fine locks

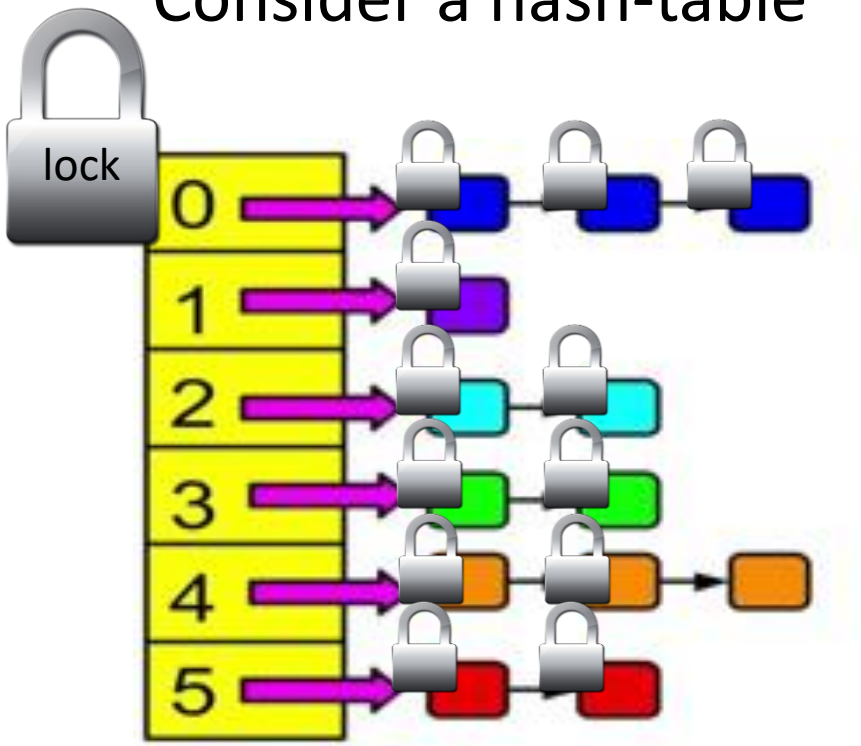
Consider a hash-table

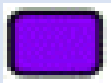
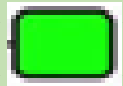
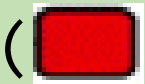


thread T1	thread T2
`figure-out-locks();`	`figure-out-locks();`
`lock-them-inorder();`	`lock-them-inorder();`
`ht.add(`  `);`	`ht.add(`  `);`
`if(ht.cont`  `s( ))`	`if(ht.contains(`  `))`
`ht.del( );`	`ht.del(`  `);`
`unlock-locks();`	`unlock-locks();`

Pessimistic concurrency control: fine locks

Consider a hash-table

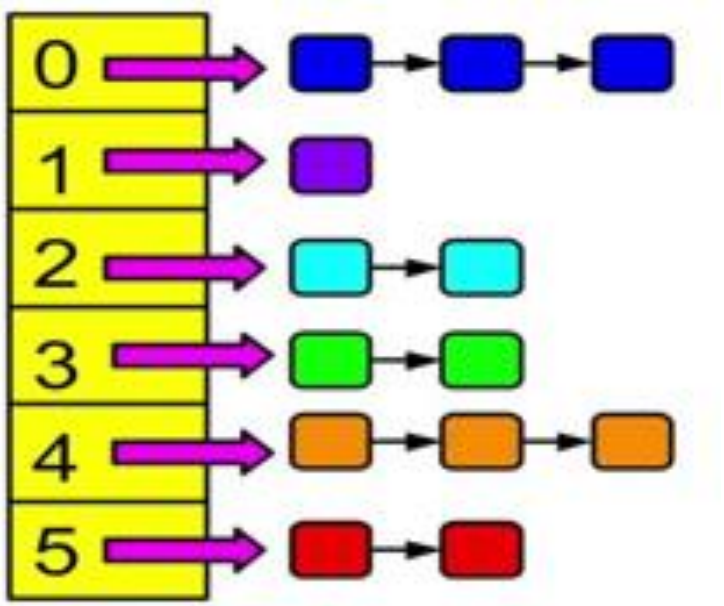


thread T1	thread T2
`figure-out-locks();`	`figure-out-locks();`
`lock-them-inorder();`	`lock-them-inorder();`
`ht.add(`  `);`	`ht.add(`  `);`
`if(ht.cont`  `s( ))`	`if(ht.contains(`  `))`
`ht.del( );`	`ht.del(`  `);`
`unlock-locks();`	`unlock-locks();`

Fine-grain lock:
Non-conflicting parallel
High Complexity -- High Performance

Transaction-based concurrency control

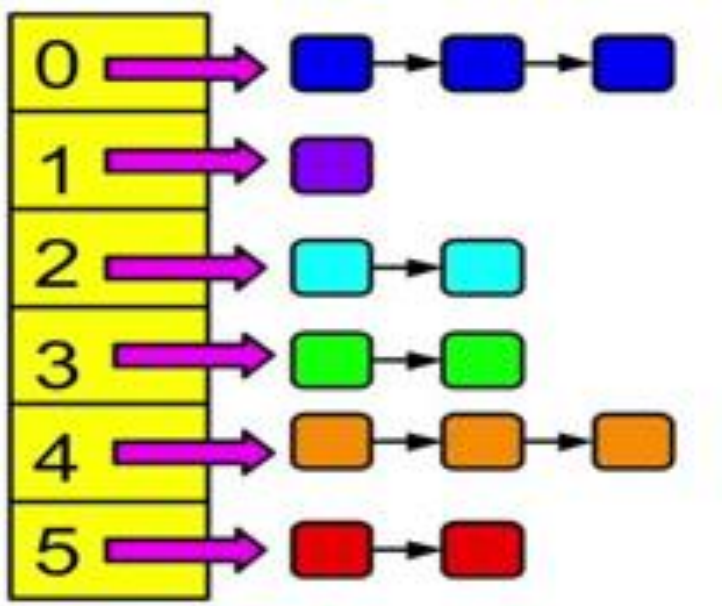
Consider a hash-table



thread T1	thread T2
BeginTX();	**BeginTX();**
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
EndTX();	**EndTX();**

Transaction-based concurrency control

Consider a hash-table



thread T1	thread T2
BeginTX();	**BeginTX();**
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
EndTX();	

How to make this work?
Transactions are an ***abstraction***
Can be ***implemented*** in many ways:

- Locks (SGL semantics, 2PL)
- OCC, MVCC
- OCC → low complexity, high performance

Optimistic Concurrency Control

Key Idea:

- Critical sections concurrent
- Conflicts detected dynamically
  $\varnothing \neq \{W_a\} \cap \{R_b \cup W_b\}$
- On conflict abort/retry

∩

Optimistic Concurrency Control

Key Idea:

- Critical sections concurrent
- Conflicts detected dynamically

  $\varnothing \neq \{W_a\} \cap \{R_b \cup W_b\}$
- On conflict abort/retry

```
On-BeginTX:
   r-set = {}
   w-set = {}
```

∩

Optimistic Concurrency Control

Key Idea:

- Critical sections concurrent
- Conflicts detected dynamically

  $\varnothing \neq \{W_a\} \cap \{R_b \cup W_b\}$
- On conflict abort/retry

```
On-BeginTX:
  r-set = {}
  w-set = {}
On-Write(v, val):
  w-set = w-set U {<v,val>}
```

∩

Optimistic Concurrency Control

Key Idea:

- Critical sections concurrent
- Conflicts detected dynamically

  $\varnothing \neq \{W_a\} \cap \{R_b \cup W_b\}$
- On conflict abort/retry

```
On-BeginTX:
  r-set = {}
  w-set = {}
On-Write(v, val):
  w-set = w-set U {<v,val>}
On-Read(v):
  r-set = r-set U {v}
  if(w-set.contains(v))
    return w-set.get(v)
```

∩

Optimistic Concurrency Control

Key Idea:

- Critical sections concurrent
- Conflicts detected dynamically

  $\varnothing \neq \{W_a\} \cap \{R_b \cup W_b\}$
- On conflict abort/retry

```
On-BeginTX:
  r-set = {}
  w-set = {}
On-Write(v, val):
  w-set = w-set U {<v,val>}
On-Read(v):
  r-set = r-set U {v}
  if(w-set.contains(v))
    return w-set.get(v)
On-Commit():
  for all txns a
    if a.w-set ∩ b.rw-set != {}
       abort
  for all <v,val> in w-set
    apply(v, val)
```

Optimistic Concurrency Control

Key Idea:

- Critical sections concurrent
- Conflicts detected dynamically

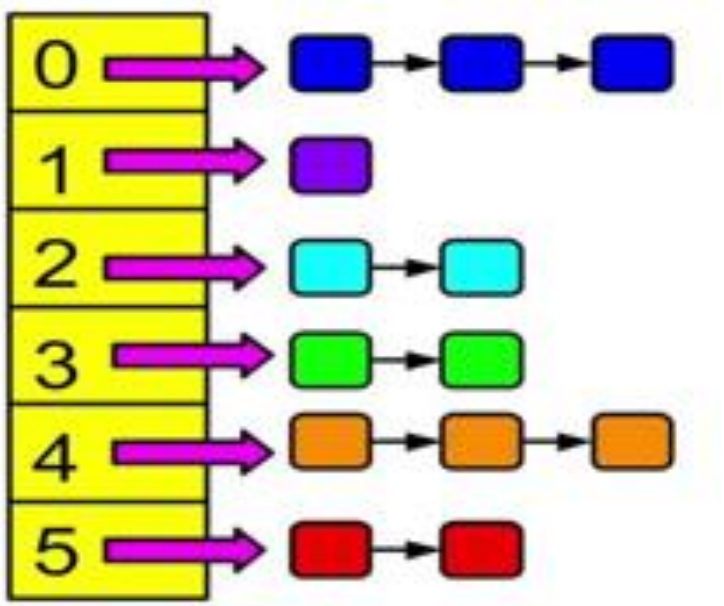
$$\varnothing \neq \{W_a\} \cap \{R_b \cup W_b\}$$

- On conflict abort/retry

OCC systems decomposed into:
Conflict detection: mechanism, policy
Version management: mechanism, policy
Granularity: what is being read/written?

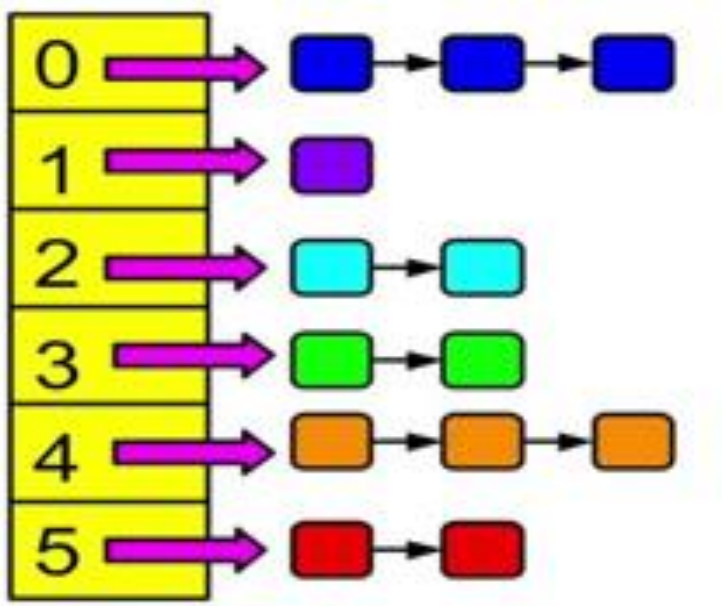
```
On-BeginTX:
  r-set = {}
  w-set = {}
On-Write(v, val):
  w-set = w-set U {<v,val>}
On-Read(v):
  r-set = r-set U {v}
  if(w-set.contains(v))
    return w-set.get(v)
On-Commit():
  for all txns a
    if a.w-set ∩ b.rw-set != {}
       abort
  for all <v,val> in w-set
    apply(v, val)
```

Coarse optimistic concurrency control



thread T1	thread T2
BeginTX();	**BeginTX();**
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
EndTX();	**EndTX();**

Coarse optimistic concurrency control

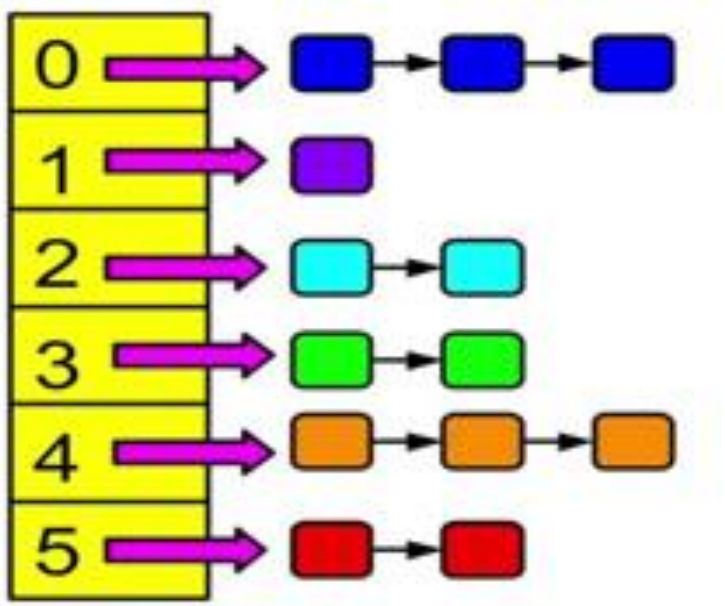


thread T1	thread T2

BeginTX();
ht.add( );
 	
if(ht.contains( ))
ht.del( );
EndTX();
	BeginTX();
	ht.add( );
	
	if(ht.contains( ))
	ht.del( );
	EndTX();

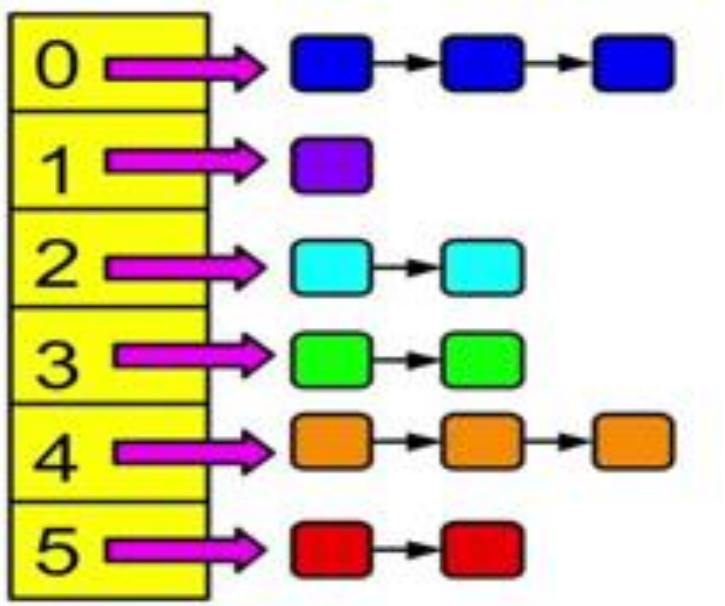
Read-set: [ ] Write-set: [ ]	Read-set: [ ] Write-set: [ ]

Coarse optimistic concurrency control



thread T1	thread T2
BeginTX();	**BeginTX();**
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
EndTX();	**EndTX();**
Read-set: [ ]	Read-set: [ ]
Write-set: [ ]	Write-set: [ ]

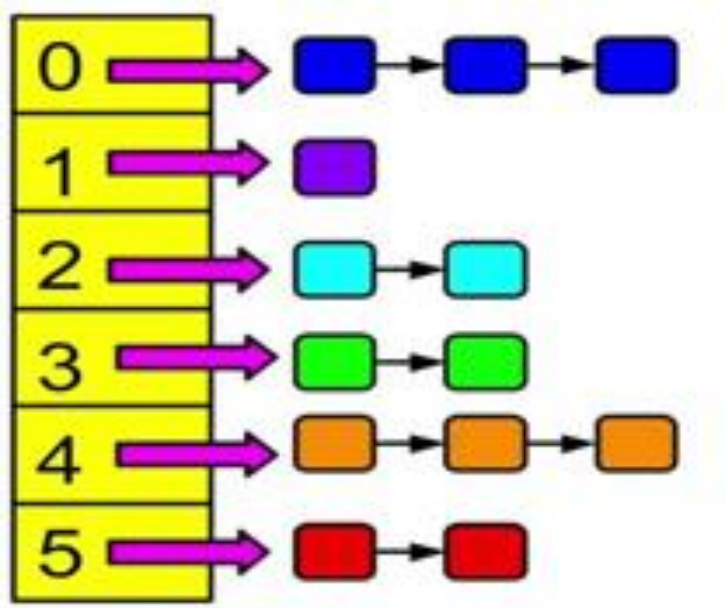
Coarse optimistic concurrency control



thread T1	thread T2
BeginTX();	**BeginTX();**
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
EndTX();	**EndTX();**
Read-set: [ ]	Read-set: [ ]
Write-set: [ ]	Write-set: [ ]

$$\varnothing \neq \{W_a\} \cap \{R_b \cup W_b\}$$

Coarse optimistic concurrency control

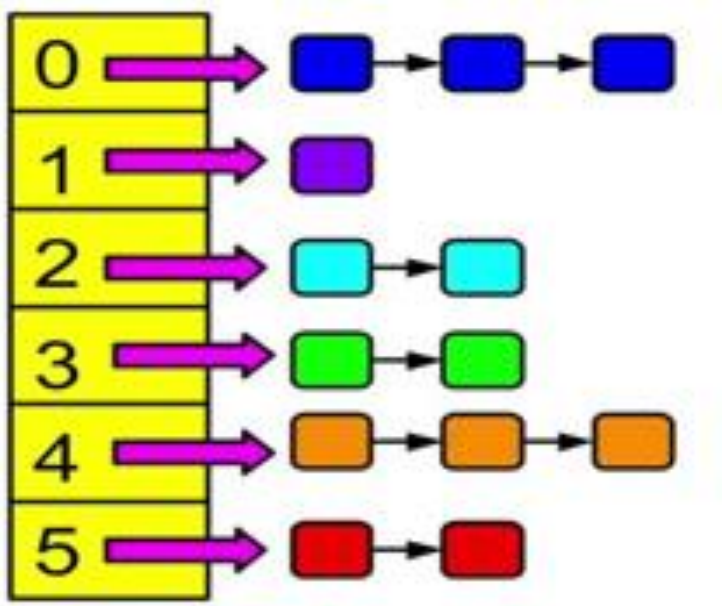


Coarse OCC:
Non-conflicting ops still serial!
ARGHHH!!!

thread T1	thread T2
BeginTX();	**BeginTX();**
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
EndTX();	**EndTX();**
Read-set: [ ]	Read-set: [ ]
Write-set: [ ]	Write-set: [ ]

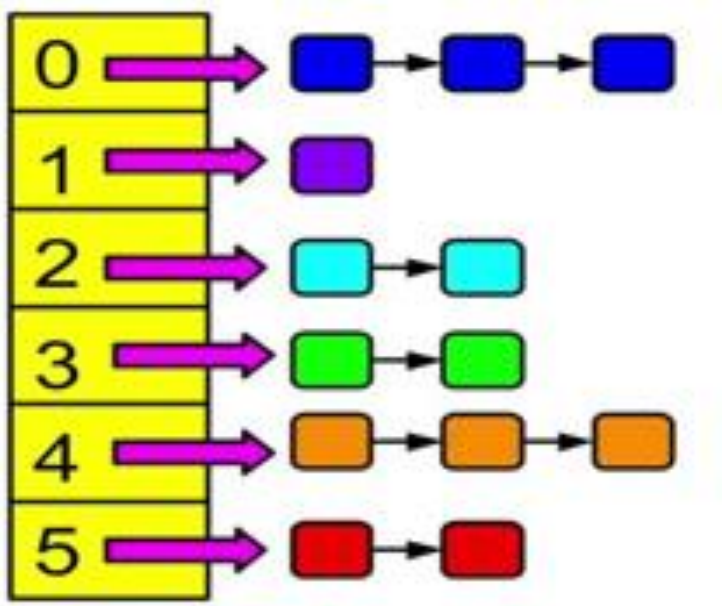
$$\varnothing \neq \{W_a\} \cap \{R_b \cup W_b\}$$

Optimistic concurrency control: transactions



thread T1	thread T2
BeginTX();	**BeginTX();**
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
BeginTX();	**BeginTX();**

Optimistic concurrency control: transactions

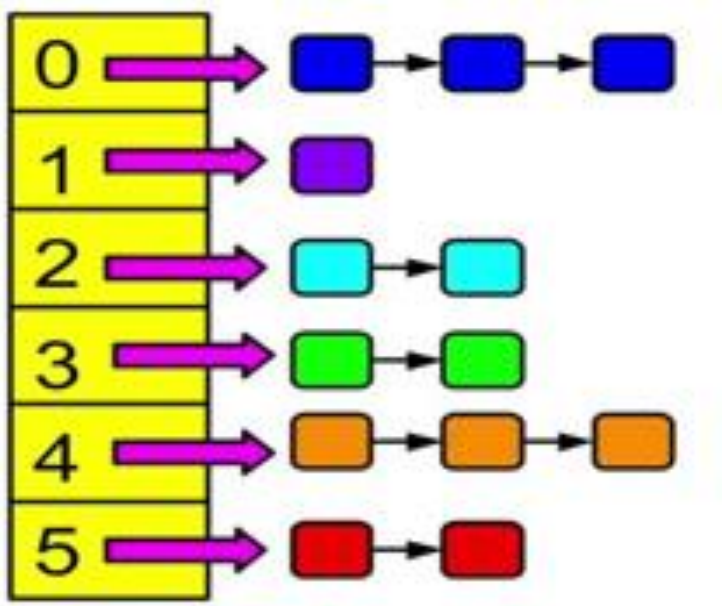


thread T1	thread T2

BeginTX();
ht.add( );
 	
if(ht.contains( ))
ht.del( );
BeginTX();
	BeginTX();
	ht.add( );
	
	if(ht.contains( ))
	ht.del( );
	BeginTX();

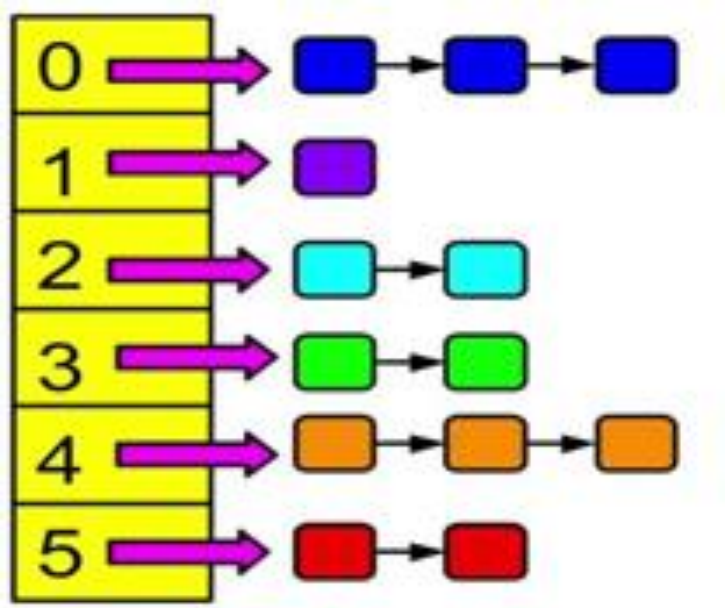
Read-set: [ ] Write-set: [ ]	Read-set: [ ] Write-set: [ ]

Optimistic concurrency control: transactions



thread T1	thread T2
BeginTX();	**BeginTX();**
ht.add( );	ht.add( );
if(ht.contains( ))	if(ht.contains( ))
ht.del( );	ht.del( );
BeginTX();	**BeginTX();**
Read-set: [ ]	Read-set: [ ]
Write-set: [ ]	Write-set: [ ]

Optimistic concurrency control: transactions



Fine-grain OCC:
Non-conflicting ops parallel
Low Complexity -- High Performance

thread T1	thread T2

BeginTX();
ht.add( );
 	
if(ht.contains( ))
ht.del( );
BeginTX();
	BeginTX();
	ht.add( );
	
	if(ht.contains( ))
	ht.del( );
	BeginTX();

Read-set: [ ] Write-set: [ ]	Read-set: [ ] Write-set: [ ]

$$\varnothing == \{W_a\} \cap \{R_b \cup W_b\}$$

OK, now…building a transactional system

- **Detect conflicts**
  - Safety: Detect and serialize all conflicting transactions
  - Performance: Don’t serialize non-conflicting transactions
- Version management
  - Private copies instead of undo log
- All code must respect transaction isolation
  - Legacy code does not use transactions

False conflicts ruin performance

- Transactions execute concurrently for performance
  - Conflicting transactions must serialize for safety
  - Conflicts defined by rules, e.g., many readers, 1 writer
- False conflict: rules mark safe operations as conflict
  - E.g., concurrent increments of a shared counter
  - Ruined performance of previous transactional OSes
- Precisely identifying conflicts is the first-order performance concern for system transactions

Minimizing false conflicts

	Read	Write
Read		
Write		

Minimizing false conflicts

	Read	Write
Read		
Write		

```
sys_xbegin();
create("/tmp/foo");
sys_xend();
```

```
sys_xbegin();
create("/tmp/bar");
sys_xend();
```

Minimizing false conflicts

	Read	Write
Read		
Write		


```
sys_xbegin();
create("/tmp/foo");
sys_xend();
```

```
sys_xbegin();
create("/tmp/bar");
sys_xend();
```

Minimizing false conflicts

	Read	Add/Del
Read		
Add/Del		

```
sys_xbegin();
create("/tmp/foo");
sys_xend();
```

```
sys_xbegin();
create("/tmp/bar");
sys_xend();
```

Minimizing false conflicts

	Read	Add/Del
Read		
Add/Del		

OK if different files created, Dir not read

```
sys_xbegin();
create("/tmp/foo");
sys_xend();
```

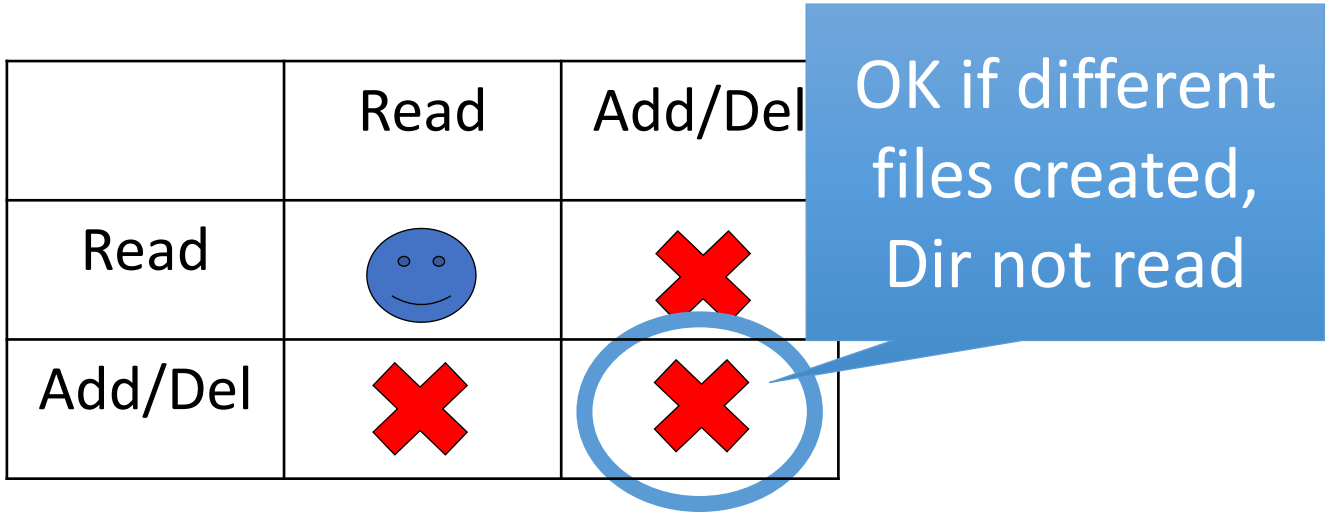
```
sys_xbegin();
create("/tmp/bar");
sys_xend();
```

Minimizing false conflicts

	Read	Add/Del
Read		
Add/Del		

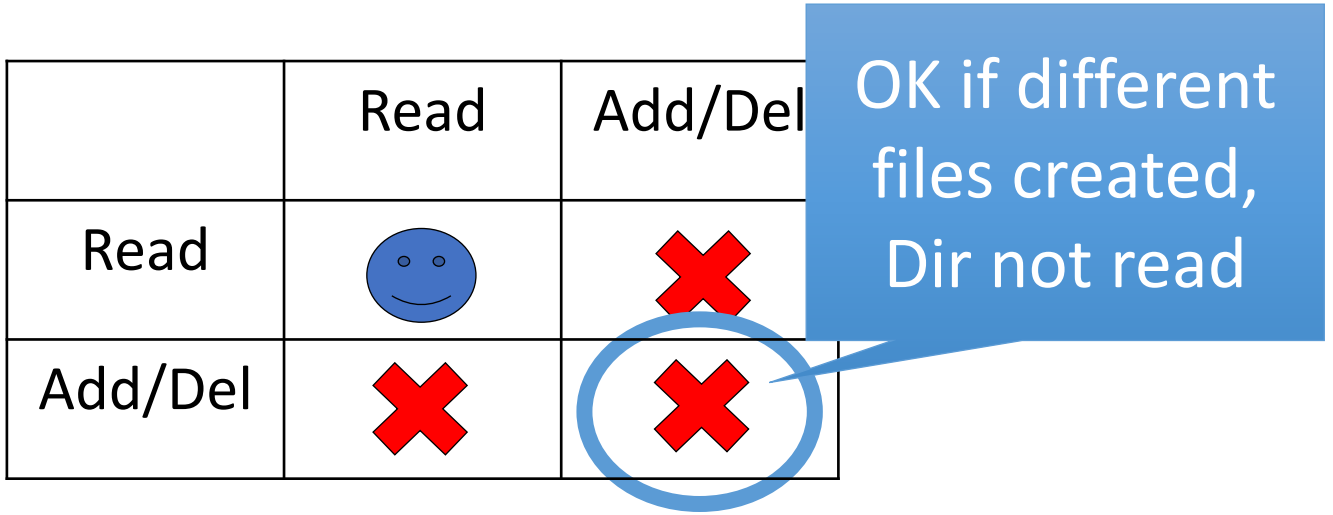
OK if different files created, Dir not read

Minimizing false conflicts



- Insight: object semantics allow more permissive conflict definition and therefore more concurrency

Minimizing false conflicts



- Insight: object semantics allow more permissive conflict definition and therefore more concurrency
- TxOS supports precise conflict definitions per object type

Minimizing false conflicts

	Read	Add/Del	Add/Del+R
Read			
Add/Del			
Add/Del+R			

- Insight: object semantics allow more permissive conflict definition and therefore more concurrency
- TxOS supports precise conflict definitions per object type

Minimizing false conflicts

	Read	Add/Del	Add/Del+R
Read			
Add/Del			
Add/Del+R			

- Insight: object semantics allow more permissive conflict definition and therefore more concurrency
- TxOS supports precise conflict definitions per object type
- Increases concurrency without relaxing isolation

Kernel objects prone to false conflicts

```
struct inode {
        struct hlist_node i_hash;
        struct list_head i_list;
        struct list_head i_sb_list;
        struct list_head i_dentry;
        unsigned long i_ino;
        atomic_t i_count;
        unsigned int i_nlink;
        uid_t i_uid;
        gid_t i_gid;
        dev_t i_rdev;
        unsigned long i_version;
        loff_t i_size;
        struct timespec i_atime;
        struct timespec i_mtime;
        struct timespec i_ctime;
        unsigned int i_blkbits;
        blkcnt_t i_blocks;
        unsigned short i_bytes;
        umode_t i_mode;
        spinlock_t i_lock;
        struct mutex i_mutex;
        struct rw_semaphore i_alloc_sem;
        const struct inode_operations *i_op;
        const struct file_operations *i_fop;
        struct super_block *i_sb;
        struct file_lock *i_flock;
        struct address_space *i_mapping;
        struct address_space i_data;
        struct list_head i_devices;
        union {
           struct pipe_inode_info *i_pipe;
           struct block_device *i_bdev;
           struct cdev *i_cdev; };
        int i_cindex;
        __u32 i_generation;
        unsigned long i_dnotify_mask;
        struct dnotify_struct *i_dnotify;
        struct list_head inotify_watches;
        struct mutex inotify_mutex;
        unsigned long i_state;
        unsigned long dirtied_when;
        unsigned int i_flags;
        atomic_t i_writecount;
        void *i_security;
        void *i_private;

};
```

- In addition to file metadata, inodes store:

Kernel objects prone to false conflicts

```
struct inode {
        struct hlist_node i_hash;
        struct list_head i_list;
        struct list_head i_sb_list;
        struct list_head i_dentry;
        unsigned long i_ino;
        atomic_t i_count;
        unsigned int i_nlink;
        uid_t i_uid;
        gid_t i_gid;
        dev_t i_rdev;
        unsigned long i_version;
        loff_t i_size;
        struct timespec i_atime;
        struct timespec i_mtime;
        struct timespec i_ctime;
        unsigned int i_blkbits;
        blkcnt_t i_blocks;
        unsigned short i_bytes;
        umode_t i_mode;
        spinlock_t i_lock;
        struct mutex i_mutex;
        struct rw_semaphore i_alloc_sem;
        const struct inode_operations *i_op;
        const struct file_operations *i_fop;
        struct super_block *i_sb;
        struct file_lock *i_flock;
        struct address_space *i_mapping;
        struct address_space i_data;
        struct list_head i_devices;
        union {
           struct pipe_inode_info *i_pipe;
           struct block_device *i_bdev;
           struct cdev *i_cdev; };
        int i_cindex;
        __u32 i_generation;
        unsigned long i_dnotify_mask;
        struct dnotify_struct *i_dnotify;
        struct list_head inotify_watches;
        struct mutex inotify_mutex;
        unsigned long i_state;
        unsigned long dirtied_when;
        unsigned int i_flags;
        atomic_t i_writecount;
        void *i_security;
        void *i_private;

};
```

- In addition to file metadata, inodes store:
- Garbage collection bookkeeping

Kernel objects prone to false conflicts

```
struct inode {
    struct hlist_node i_hash;
    struct list_head i_list;
    struct list_head i_sb_list;
    struct list_head i_dentry;
    unsigned long i_ino;
    atomic_t i_count;
    unsigned int i_nlink;
    uid_t i_uid;
    gid_t i_gid;
    dev_t i_rdev;
    unsigned long i_version;
    loff_t i_size;
    struct timespec i_atime;
    struct timespec i_mtime;
    struct timespec i_ctime;
    unsigned int i_blkbits;
    blkcnt_t i_blocks;
    unsigned short i_bytes;
    umode_t i_mode;
    spinlock_t i_lock;
    struct mutex i_mutex;
    struct rw_semaphore i_alloc_sem;
    const struct inode_operations *i_op;
    const struct file_operations *i_fop;
    struct super_block *i_sb;
    struct file_lock *i_flock;
    struct address_space *i_mapping;
    struct address_space i_data;
    struct list_head i_devices;
    union {
      struct pipe_inode_info *i_pipe;
      struct block_device *i_bdev;
      struct cdev *i_cdev; };
    int i_cindex;
    __u32 i_generation;
    unsigned long i_dnotify_mask;
    struct dnotify_struct *i_dnotify;
    struct list_head inotify_watches;
    struct mutex inotify_mutex;
    unsigned long i_state;
    unsigned long dirtied_when;
    unsigned int i_flags;
    atomic_t i_writecount;
    void *i_security;
    void *i_private;
};
```

- In addition to file metadata, inodes store:
- Garbage collection bookkeeping
- References to other kernel objects

Kernel objects prone to false conflicts

```
struct inode {
    struct hlist_node i_hash;
    struct list_head i_list;
    struct list_head i_sb_list;
    struct list_head i_dentry;
    unsigned long i_ino;
    atomic_t i_count;
    unsigned int i_nlink;
    uid_t i_uid;
    gid_t i_gid;
    dev_t i_rdev;
    unsigned long i_version;
    loff_t i_size;
    struct timespec i_atime;
    struct timespec i_mtime;
    struct timespec i_ctime;
    unsigned int i_blkbits;
    blkcnt_t i_blocks;
    unsigned short i_bytes;
    umode_t i_mode;
    spinlock_t i_lock;
    struct mutex i_mutex;
    struct rw_semaphore i_alloc_sem;
    const struct inode_operations *i_op;
    const struct file_operations *i_fop;
    struct super_block *i_sb;
    struct file_lock *i_flock;
    struct address_space *i_mapping;
    struct address_space i_data;
    struct list_head i_devices;
    union {
        struct pipe_inode_info *i_pipe;
        struct block_device *i_bdev;
        struct cdev *i_cdev; };
    int i_cindex;
    __u32 i_generation;
    unsigned long i_dnotify_mask;
    struct dnotify_struct *i_dnotify;
    struct list_head inotify_watches;
    struct mutex inotify_mutex;
    unsigned long i_state;
    unsigned long dirtied_when;
    unsigned int i_flags;
    atomic_t i_writecount;
    void *i_security;
    void *i_private;
};
```

- In addition to file metadata, inodes store:
- Garbage collection bookkeeping
- References to other kernel objects
- Timestamps (typically monotonically increasing)

Kernel objects prone to false conflicts

```
struct inode {
    struct hlist_node i_hash;
    struct list_head i_list;
    struct list_head i_sb_list;
    struct list_head i_dentry;
    unsigned long i_ino;
    atomic_t i_count;
    unsigned int i_nlink;
    uid_t i_uid;
    gid_t i_gid;
    dev_t i_rdev;
    unsigned long i_version;
    loff_t i_size;
    struct timespec i_atime;
    struct timespec i_mtime;
    struct timespec i_ctime;
    unsigned int i_blkbits;
    blkcnt_t i_blocks;
    unsigned short i_bytes;
    umode_t i_mode;
    spinlock_t i_lock;
    struct mutex i_mutex;
    struct rw_semaphore i_alloc_sem;
    const struct inode_operations *i_op;
    const struct file_operations *i_fop;
    struct super_block *i_sb;
    struct file_lock *i_flock;
    struct address_space *i_mapping;
    struct address_space i_data;
    struct list_head i_devices;
    union {
        struct pipe_inode_info *i_pipe;
        struct block_device *i_bdev;
        struct cdev *i_cdev; };
    int i_cindex;
    __u32 i_generation;
    unsigned long i_dnotify_mask;
    struct dnotify_struct *i_dnotify;
    struct list_head inotify_watches;
    struct mutex inotify_mutex;
    unsigned long i_state;
    unsigned long dirtied_when;
    unsigned int i_flags;
    atomic_t i_writecount;
    void *i_security;
    void *i_private;
};
```

- In addition to file metadata, inodes store:
- Garbage collection bookkeeping
- References to other kernel objects
- Timestamps (typically monotonically increasing)
- Internal synchronization

Kernel objects prone to false conflicts

```
struct inode {
    struct hlist_node i_hash;
    struct list_head i_list;
    struct list_head i_sb_list;
    struct list_head i_dentry;
    unsigned long i_ino;
    atomic_t i_count;
    unsigned int i_nlink;
    uid_t i_uid;
    gid_t i_gid;
    dev_t i_rdev;
    unsigned long i_version;
    loff_t i_size;
    struct timespec i_atime;
    struct timespec i_mtime;
    struct timespec i_ctime;
    unsigned int i_blkbits;
    blkcnt_t i_blocks;
    unsigned short i_bytes;
    umode_t i_mode;
    spinlock_t i_lock;
    struct mutex i_mutex;
    struct rw_semaphore i_alloc_sem;
    const struct inode_operations *i_op;
    const struct file_operations *i_fop;
    struct super_block *i_sb;
    struct file_lock *i_flock;
    struct address_space *i_mapping;
    struct address_space i_data;
    struct list_head i_devices;
    union {
        struct pipe_inode_info *i_pipe;
        struct block_device *i_bdev;
        struct cdev *i_cdev; };
    int i_cindex;
    __u32 i_generation;
    unsigned long i_dnotify_mask;
    struct dnotify_struct *i_dnotify;
    struct list_head inotify_watches;
    struct mutex inotify_mutex;
    unsigned long i_state;
    unsigned long dirtied_when;
    unsigned int i_flags;
    atomic_t i_writecount;
    void *i_security;
    void *i_private;
};
```

- In addition to file metadata, inodes store:
- Garbage collection bookkeeping
- References to other kernel objects
- Timestamps (typically monotonically increasing)
- Internal synchronization
- FS notification bookkeeping

Kernel objects prone to false conflicts

```
struct inode {
    struct hlist_node i_hash;
    struct list_head i_list;
    struct list_head i_sb_list;
    struct list_head i_dentry;
    unsigned long i_ino;
    atomic_t i_count;
    unsigned int i_nlink;
    uid_t i_uid;
    gid_t i_gid;
    dev_t i_rdev;
    unsigned long i_version;
    loff_t i_size;
    struct timespec i_atime;
    struct timespec i_mtime;
    struct timespec i_ctime;
    unsigned int i_blkbits;
    blkcnt_t i_blocks;
    unsigned short i_bytes;
    umode_t i_mode;
    spinlock_t i_lock;
    struct mutex i_mutex;
    struct rw_semaphore i_alloc_sem;
    const struct inode_operations *i_op;
    const struct file_operations *i_fop;
    struct super_block *i_sb;
    struct file_lock *i_flock;
    struct address_space *i_mapping;
    struct address_space i_data;
    struct list_head i_devices;
    union {
        struct pipe_inode_info *i_pipe;
        struct block_device *i_bdev;
        struct cdev *i_cdev; };
    int i_cindex;
    __u32 i_generation;
    unsigned long i_dnotify_mask;
    struct dnotify_struct *i_dnotify;
    struct list_head inotify_watches;
    struct mutex inotify_mutex;
    unsigned long i_state;
    unsigned long dirtied_when;
    unsigned int i_flags;
    atomic_t i_writecount;
    void *i_security;
    void *i_private;
};
```

- In addition to file metadata, inodes store:
- Garbage collection bookkeeping
- References to other kernel objects
- Timestamps (typically monotonically increasing)
- Internal synchronization
- FS notification bookkeeping

Simple rules at object granularity will not perform

How to eliminate false conflicts

- Can start with simple rules and refine as needed
- Techniques:
  - Decomposition of objects
    - Separate kernel garbage collection bookkeeping
  - Idiosyncratic rules
    - Defer updates of timestamps until commit
    - Buffer file system notification events
- Price for working with Linux

Building a transactional system

- Detect conflicts
- **Version management**
  - Private copies instead of undo log
- All code must respect transaction isolation
  - Legacy code does not use transactions

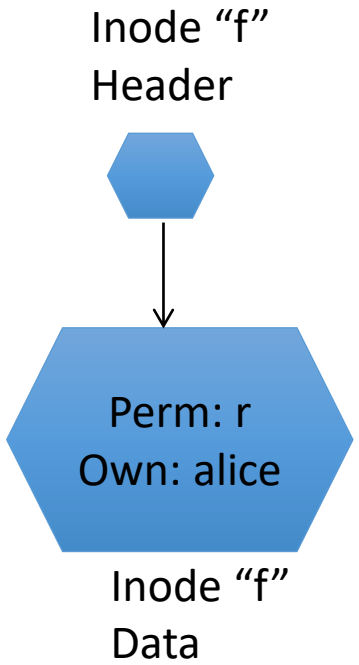
TxOS version management

CPU 0 (low priority)

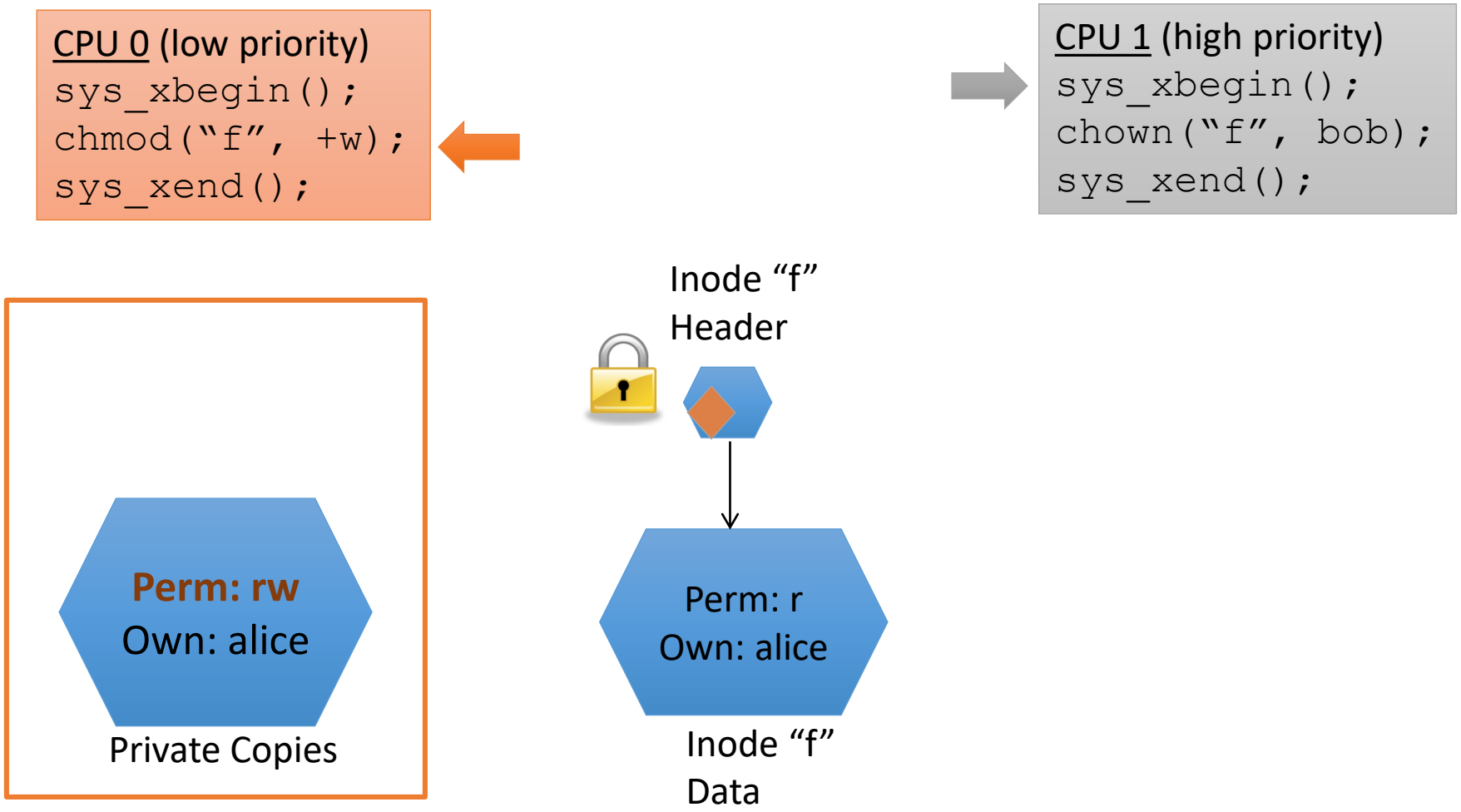
```
sys_xbegin();
chmod("f", +w);
sys_xend();
```

CPU 1 (high priority)

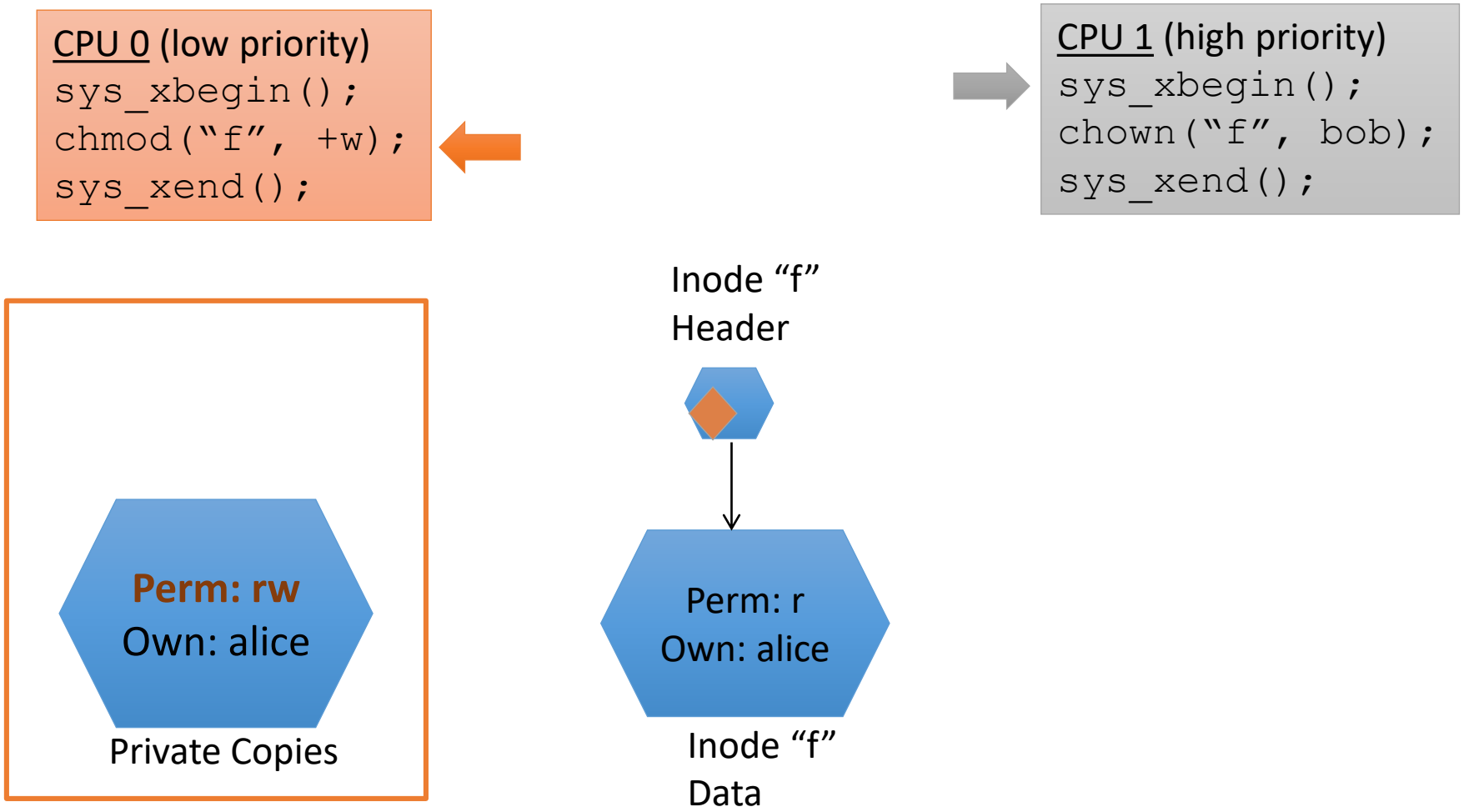
```
sys_xbegin();
chown("f", bob);
sys_xend();
```



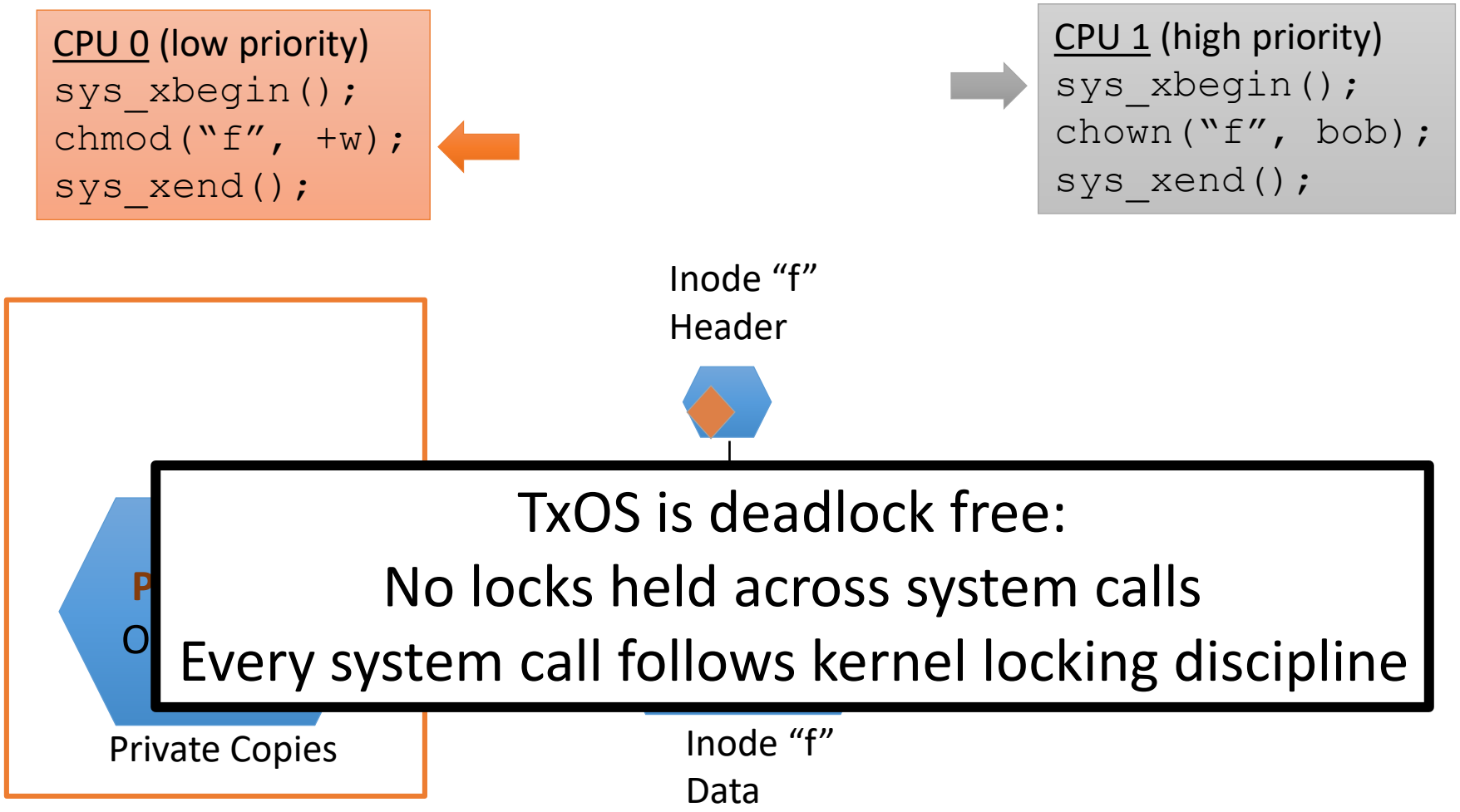
TxOS version management



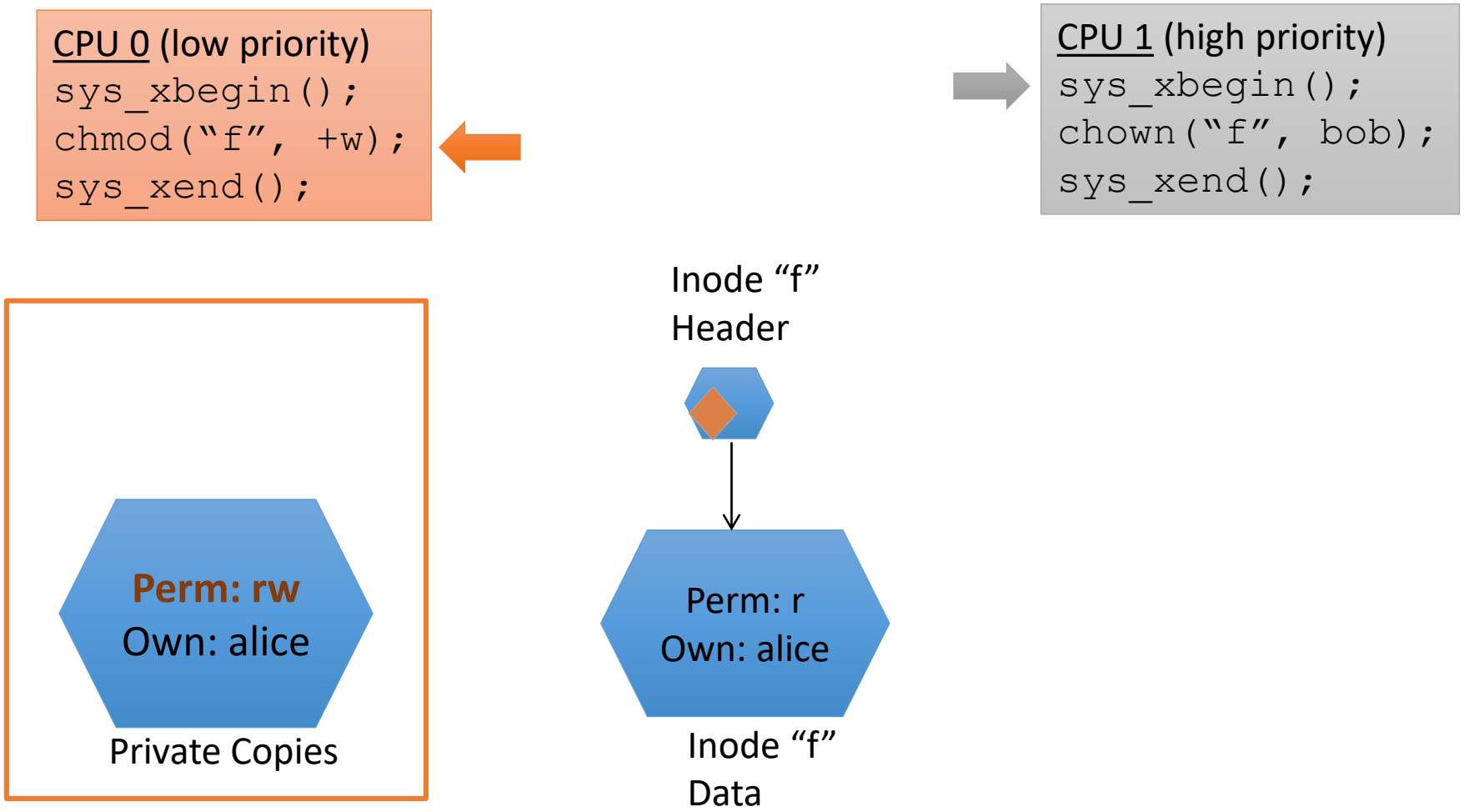
TxOS version management



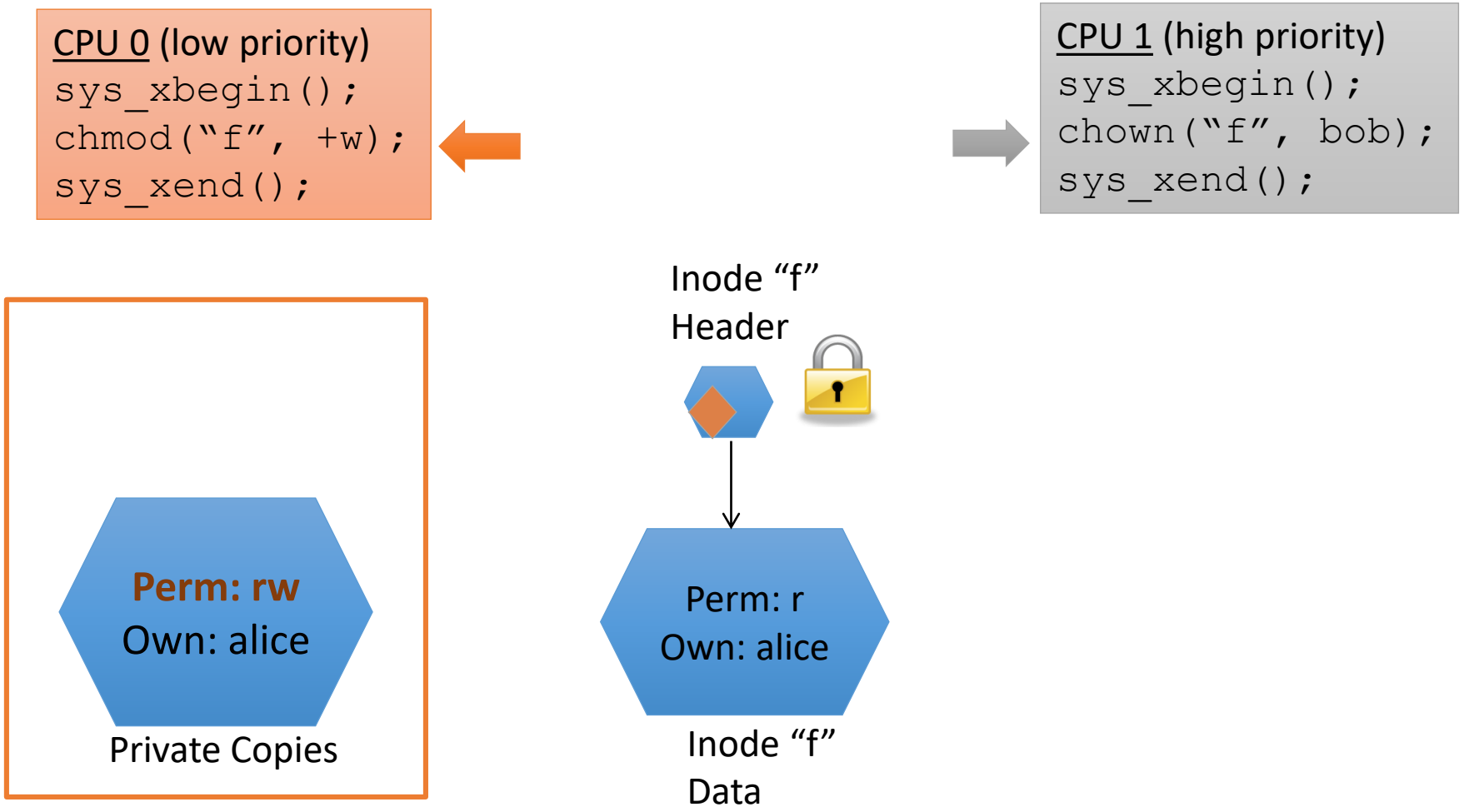
TxOS version management



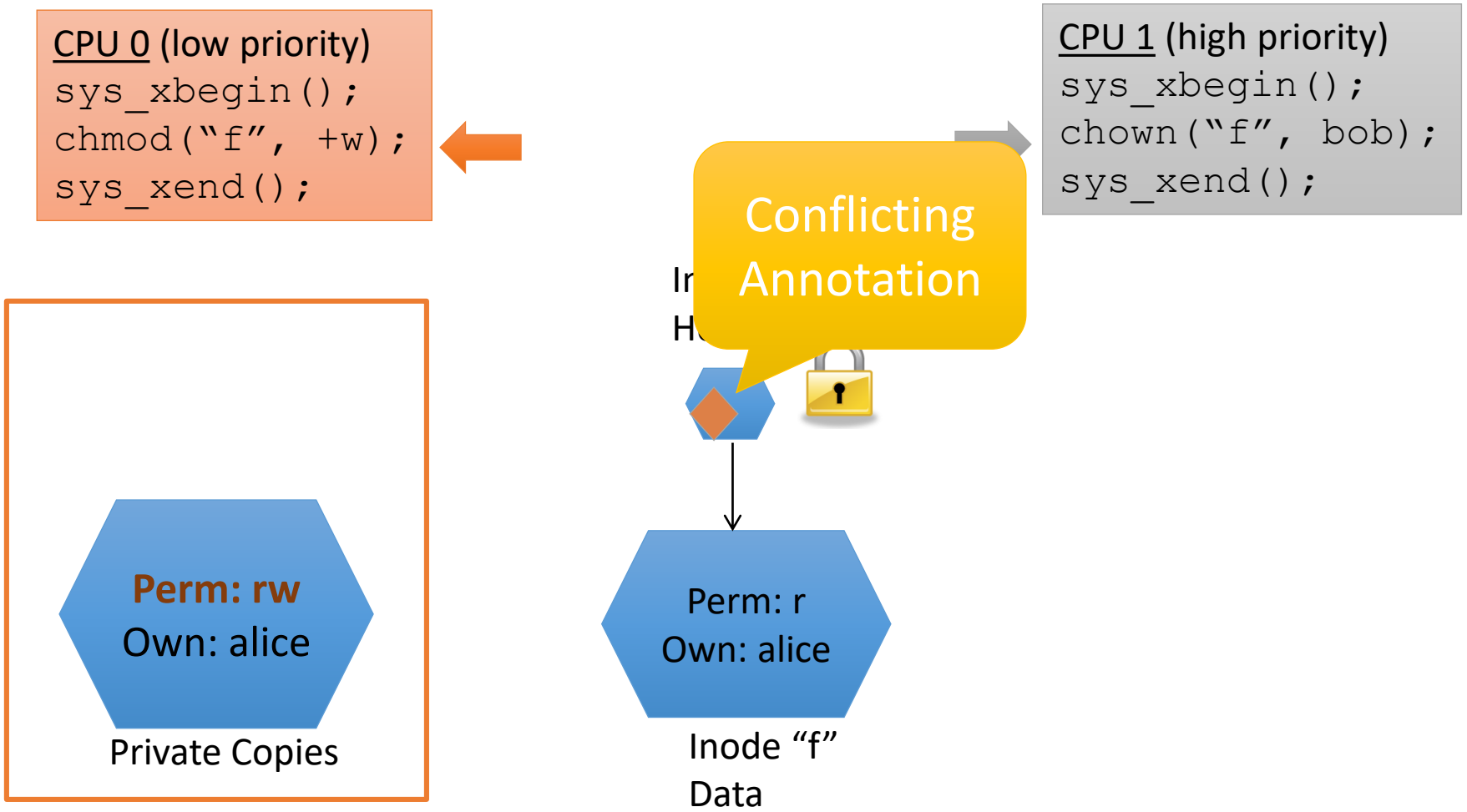
TxOS version management



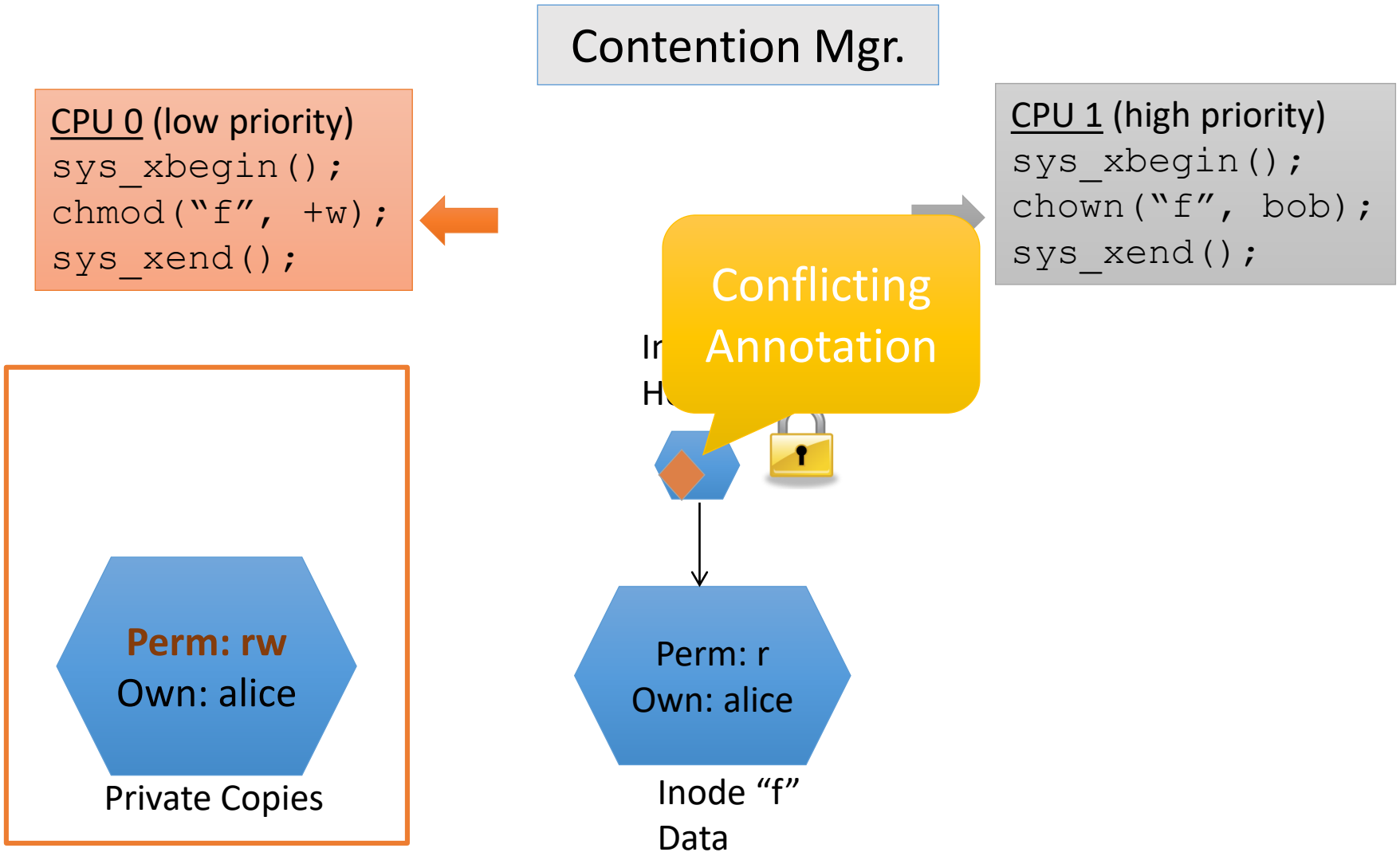
TxOS version management



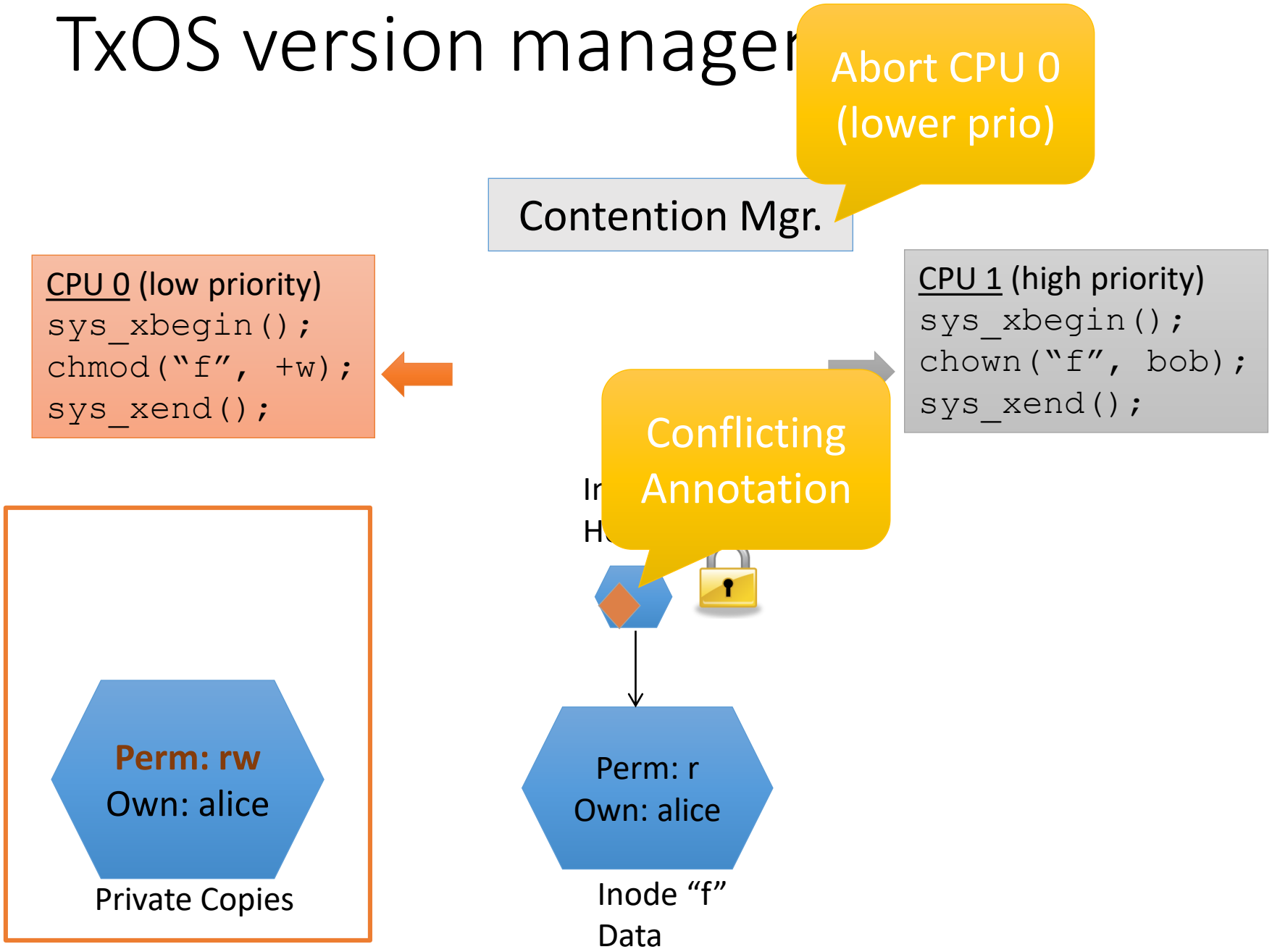
TxOS version management



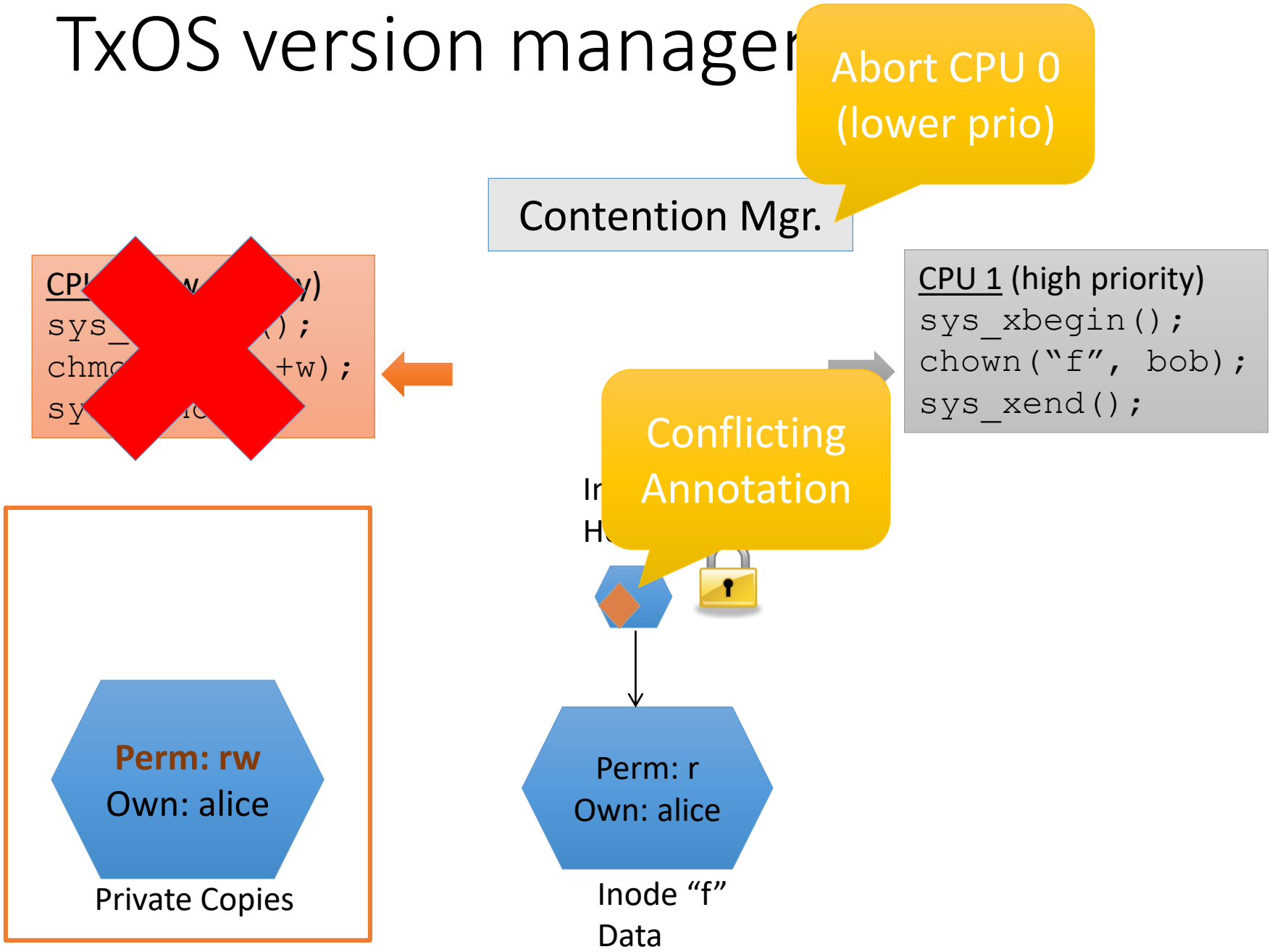
TxOS version management



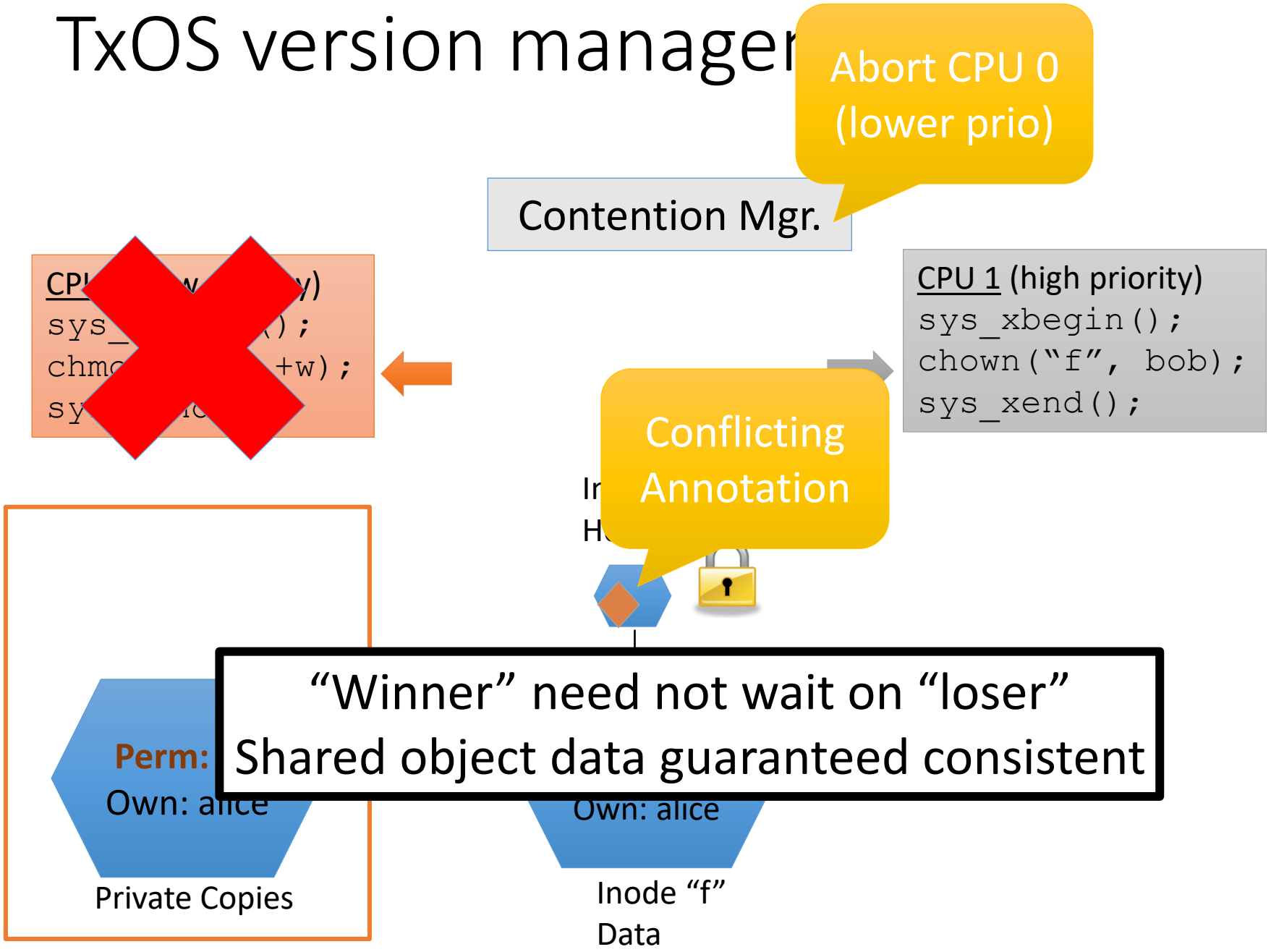
TxOS version manager



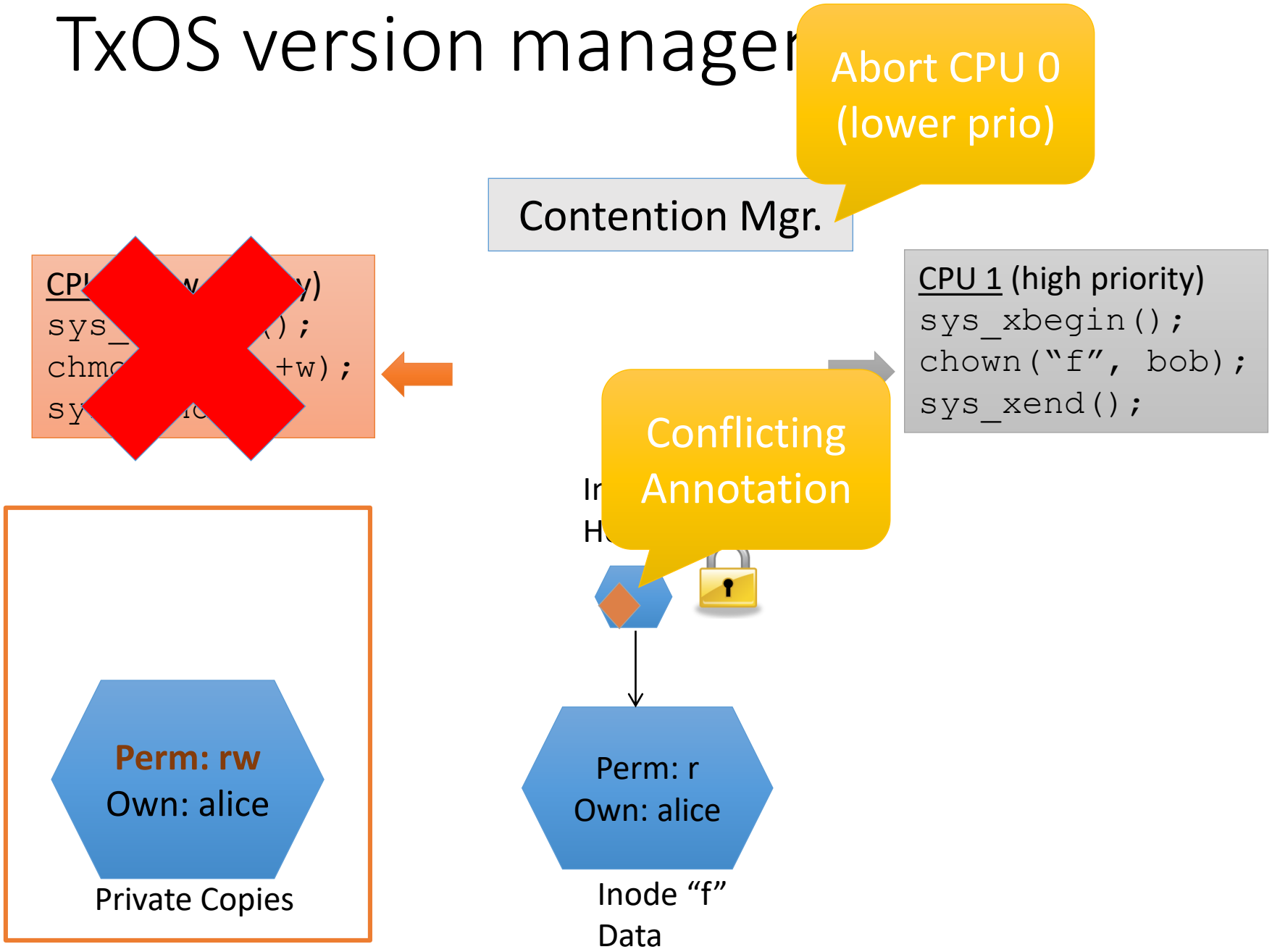
TxOS version manager



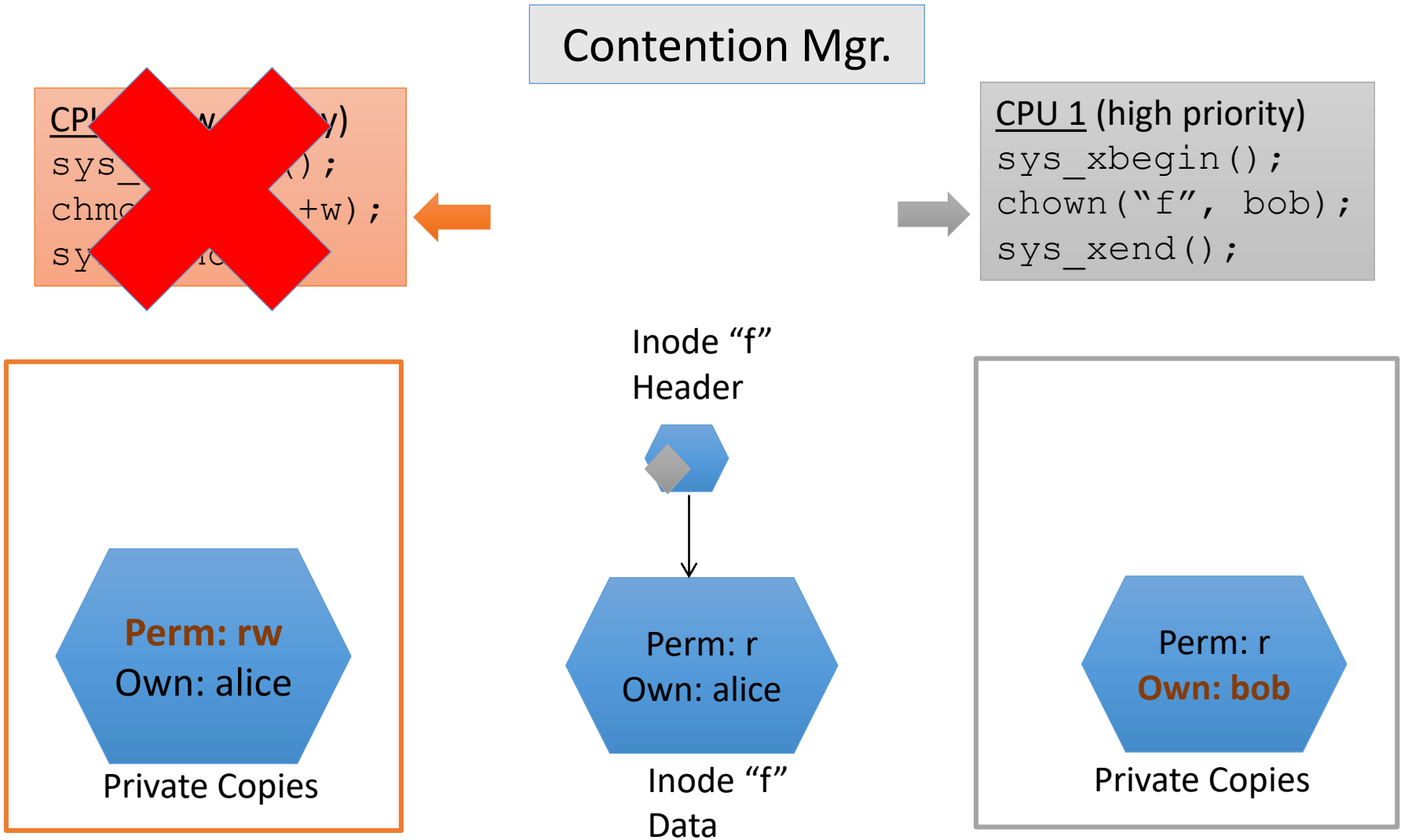
TxOS version manager



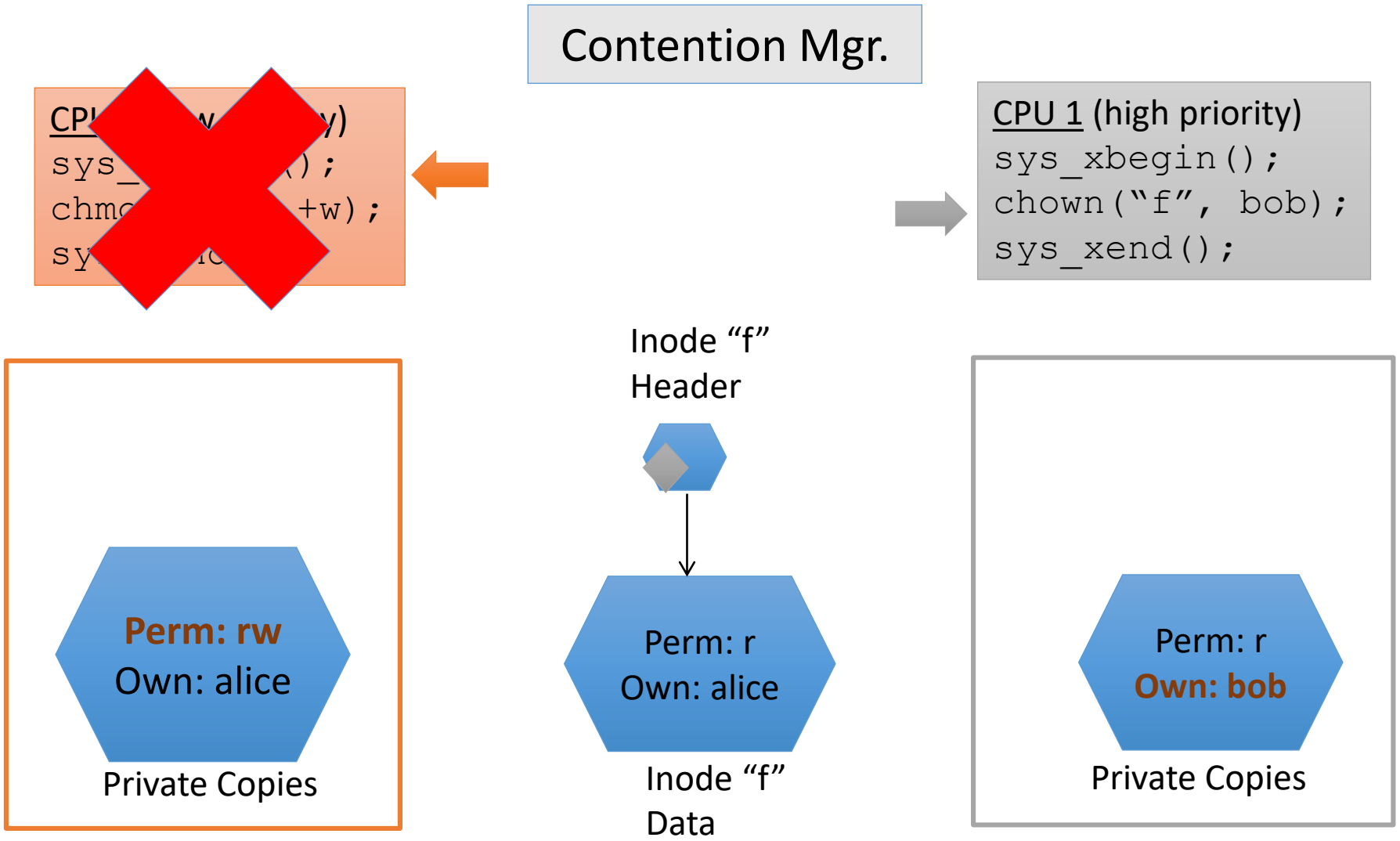
TxOS version manager



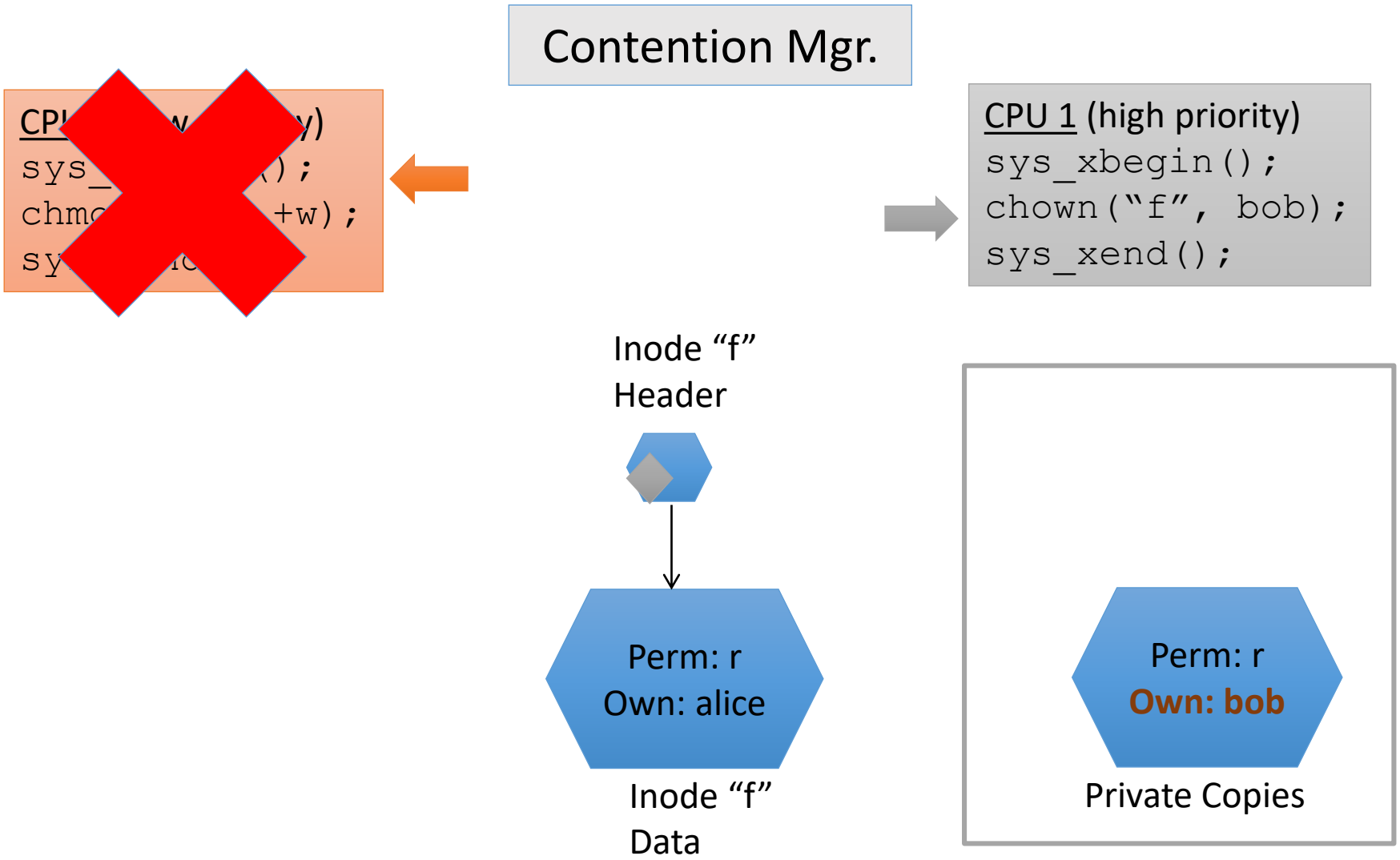
TxOS version management



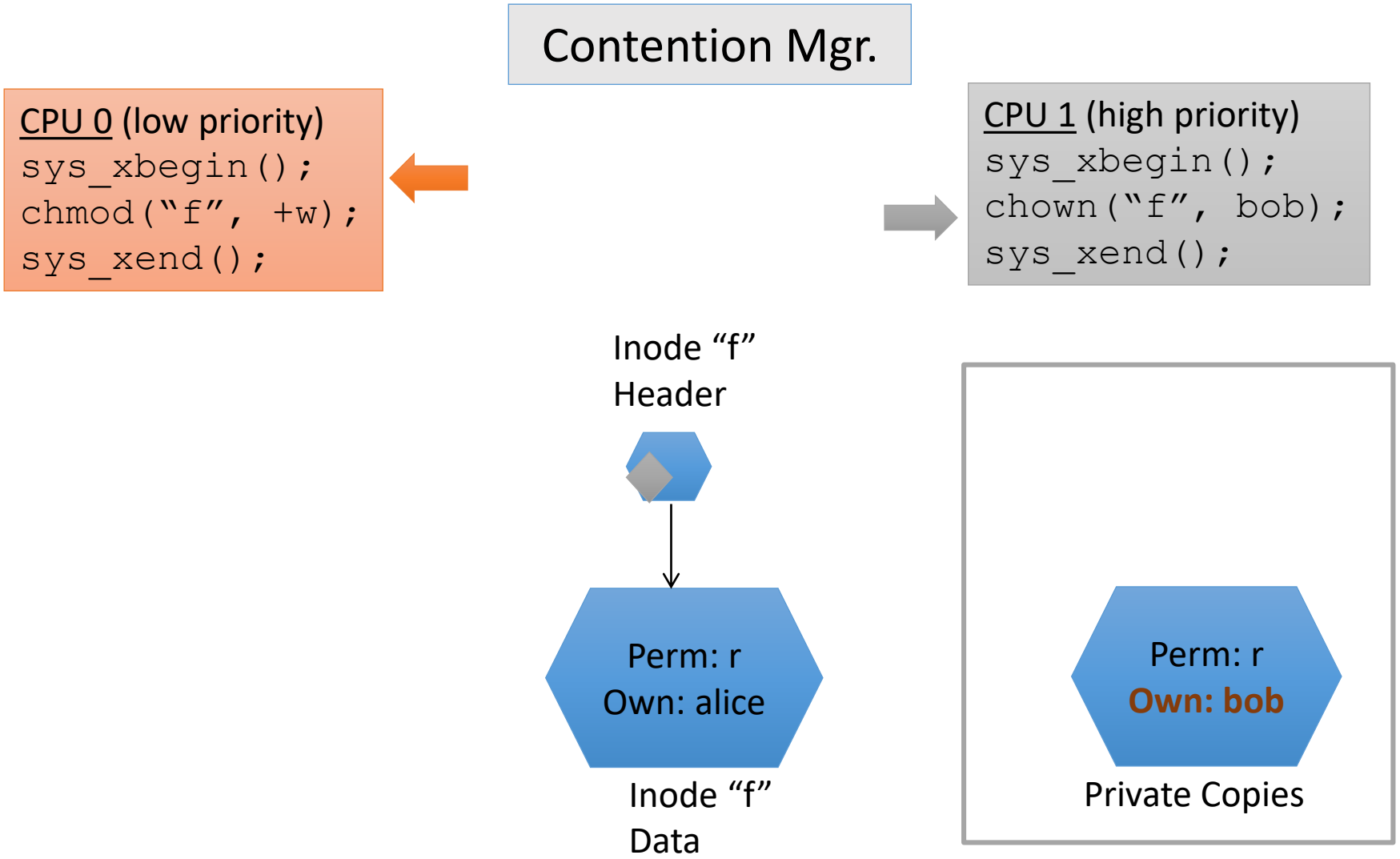
TxOS version management



TxOS version management



TxOS version management

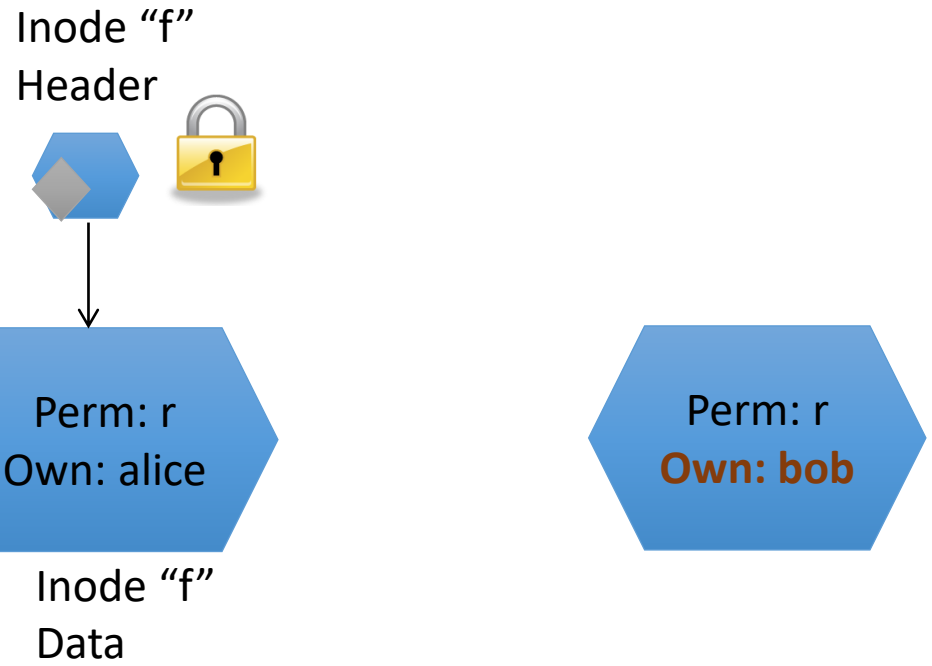


TxOS version management

Contention Mgr.

CPU 0 (low priority)
```
sys_xbegin();
chmod("f", +w);
sys_xend();
```

CPU 1 (high priority)
```
sys_xbegin();
chown("f", bob);
sys_xend();
```



TxOS version management

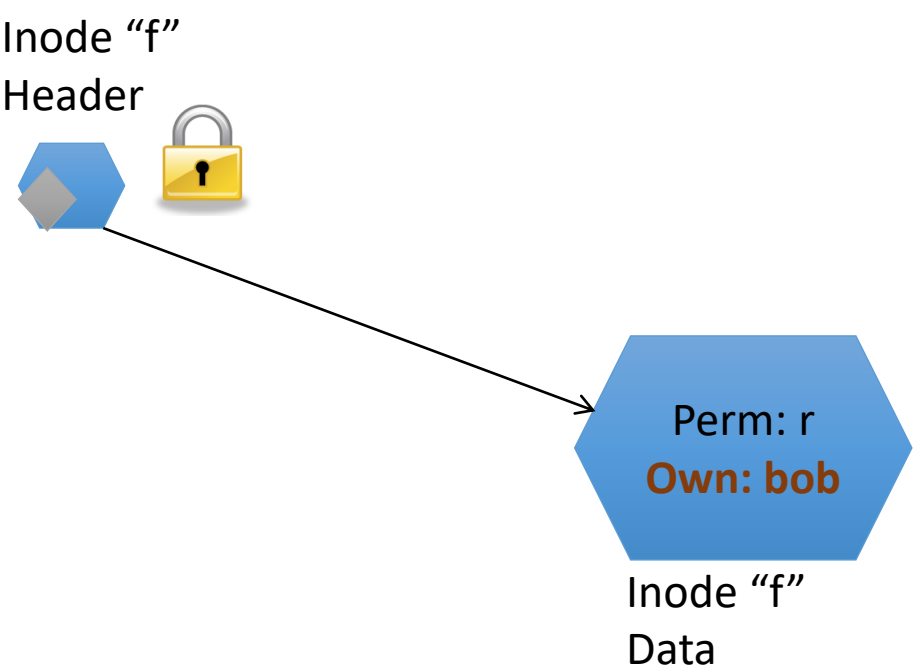
Contention Mgr.

CPU 0 (low priority)

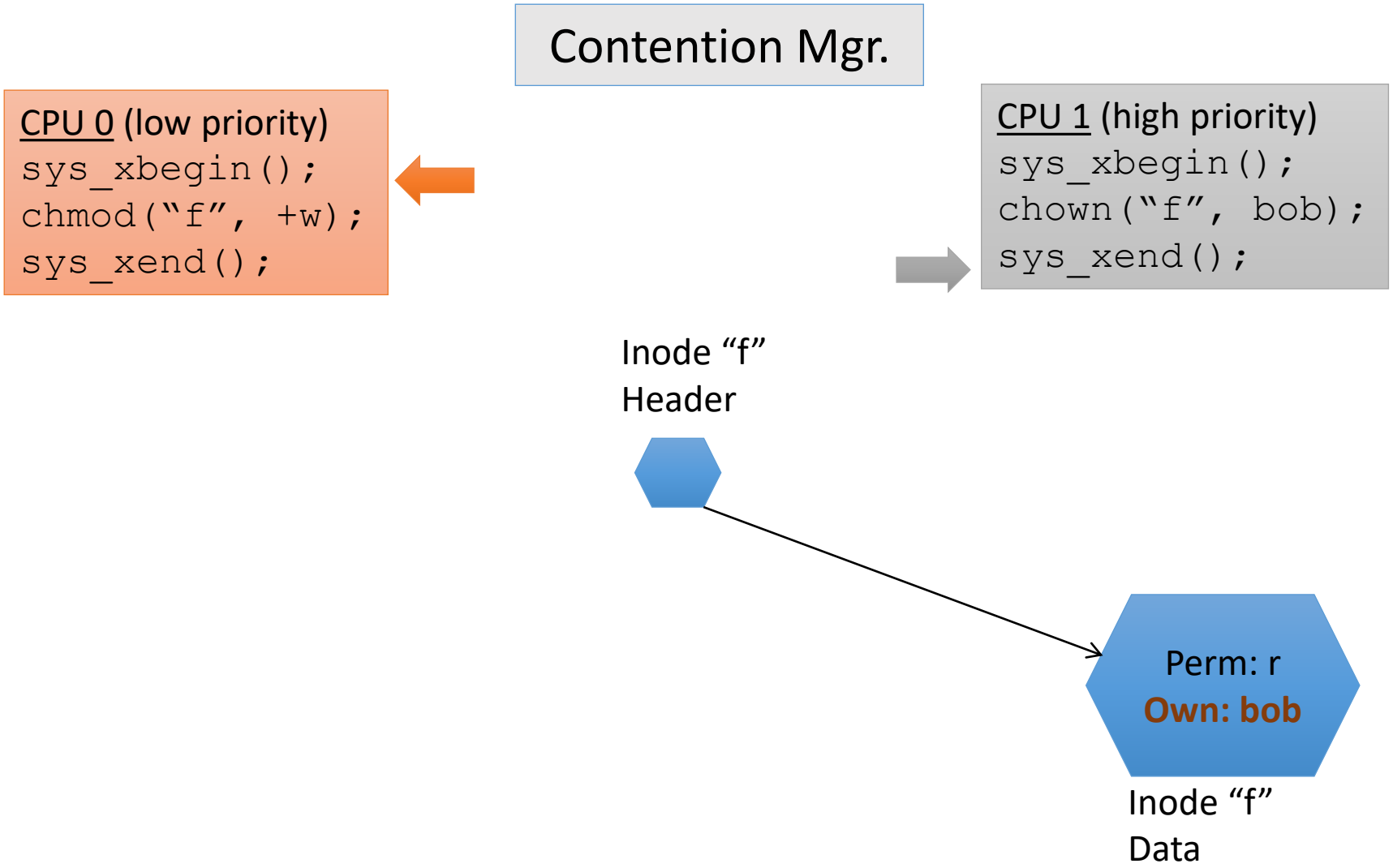
```
sys_xbegin();
chmod("f", +w);
sys_xend();
```

CPU 1 (high priority)

```
sys_xbegin();
chown("f", bob);
sys_xend();
```



TxOS version management



A novel transactional OS design

	Previous Systems	**TxOS**
Speculative write location		
Isolation mechanism		
Rollback mechanism		
Commit mechanism		

A novel transactional OS design

	Previous Systems	**TxOS**
Speculative write location	Shared data structures	
Isolation mechanism		
Rollback mechanism		
Commit mechanism		

A novel transactional OS design

	Previous Systems	**TxOS**
Speculative write location	Shared data structures	
Isolation mechanism	Two-phase locking (deadlock)	
Rollback mechanism		
Commit mechanism		

A novel transactional OS design

	Previous Systems	**TxOS**
Speculative write location	Shared data structures	
Isolation mechanism	Two-phase locking (deadlock)	
Rollback mechanism		
Commit mechanism		

Deadlock prone

A novel transactional OS design

	Previous Systems	**TxOS**
Speculative write location	Shared data structures	
Isolation mechanism	Two-phase locking (deadlock)	
Rollback mechanism	Undo log (priority inversion)	
Commit mechanism		

Deadlock prone

A novel transactional OS design

	Previous Systems	**TxOS**
Speculative write location	Shared data structures	
Isolation mechanism	Two-phase locking (deadlock)	
Rollback mechanism	Undo log (priority inversion)	
Commit mechanism		

Deadlock prone

Can cause priority inversion

A novel transactional OS design

	Previous Systems	**TxOS**
Speculative write location	Shared data structures	
Isolation mechanism	Two-phase locking (deadlock)	
Rollback mechanism	Undo log (priority inversion)	
Commit mechanism	Discard undo log, release locks	

Deadlock prone

Can cause priority inversion

A novel transactional OS design

	Previous Systems	**TxOS**
Speculative write location	Shared data structures	
Isolation mechanism	Two-phase locking (deadlock)	
Rollback mechanism	Undo log (priority inversion)	
Commit mechanism	Discard undo log, release locks	

A novel transactional OS design

	Previous Systems	**TxOS**
Speculative write location	Shared data structures	**Private copies of data structures**
Isolation mechanism	Two-phase locking (deadlock)	**Private copies + annotations**
Rollback mechanism	Undo log (priority inversion)	
Commit mechanism	Discard undo log, release locks	

A novel transactional OS design

	Previous Systems	**TxOS**
Speculative write location	Shared data structures	**Private copies of data structures**
Isolation mechanism	Two-phase locking (deadlock)	**Private copies + annotations**
Rollback mechanism	Undo log (priority inversion)	**Discard private copies**
Commit mechanism	Discard undo log, release locks	**Publish private copy by ptr swap**

Handling aborts in an application

- After abort in kernel, system calls will return a special error code
  - Until an **`xend()`** or **`xabort()`** call issued
  - Application state must be rolled back
- Simplicity: Let the system do it for you
  - Copy-on-write address space (single thread)
  - Transactional memory (multiple threads)
- Performance: Manually roll back application state
  - Mature code already has error handlers for system calls

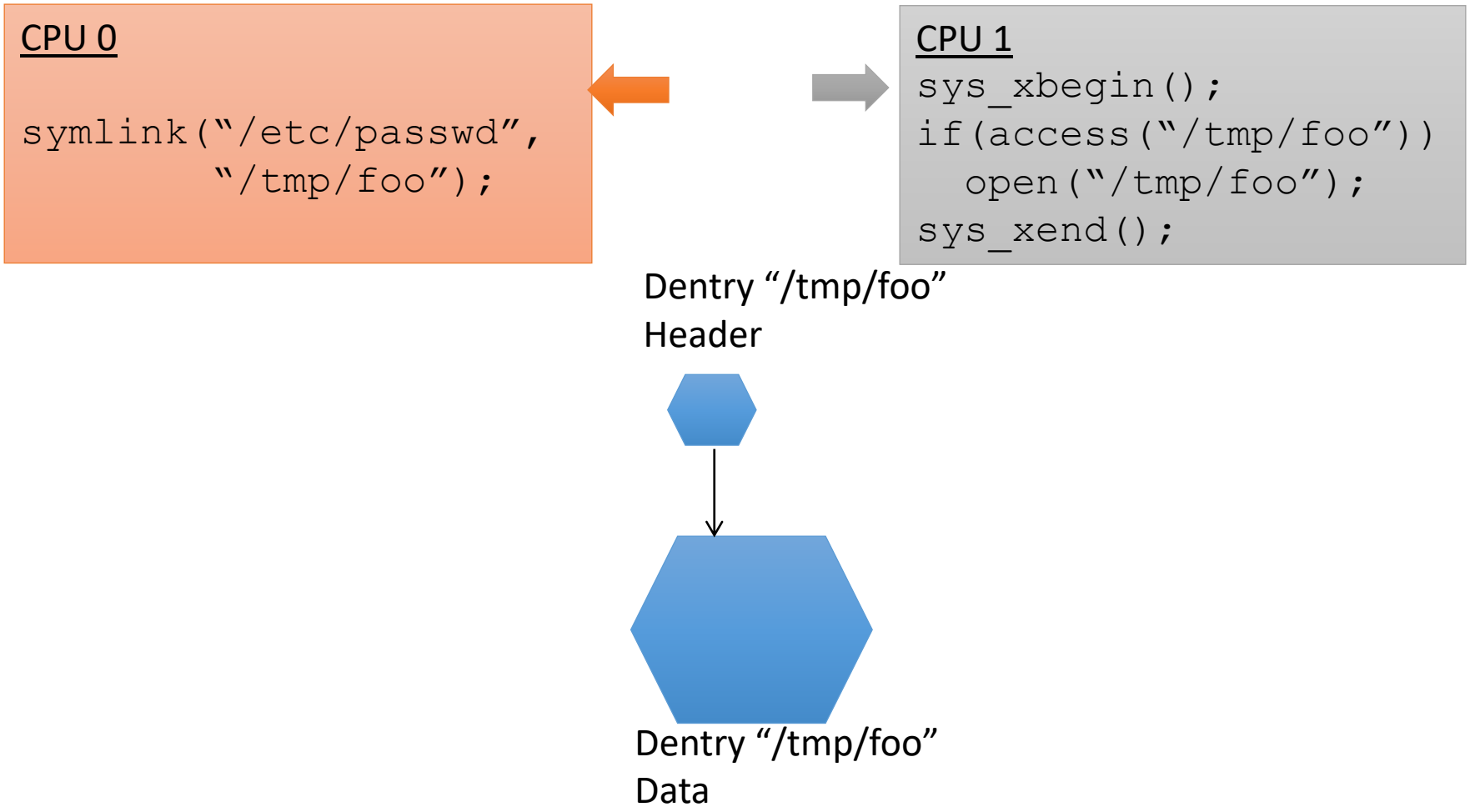
Building a transactional system

- Detect conflicts
- Version management
- **All code must respect transaction isolation**
  - Legacy code does not use transactions

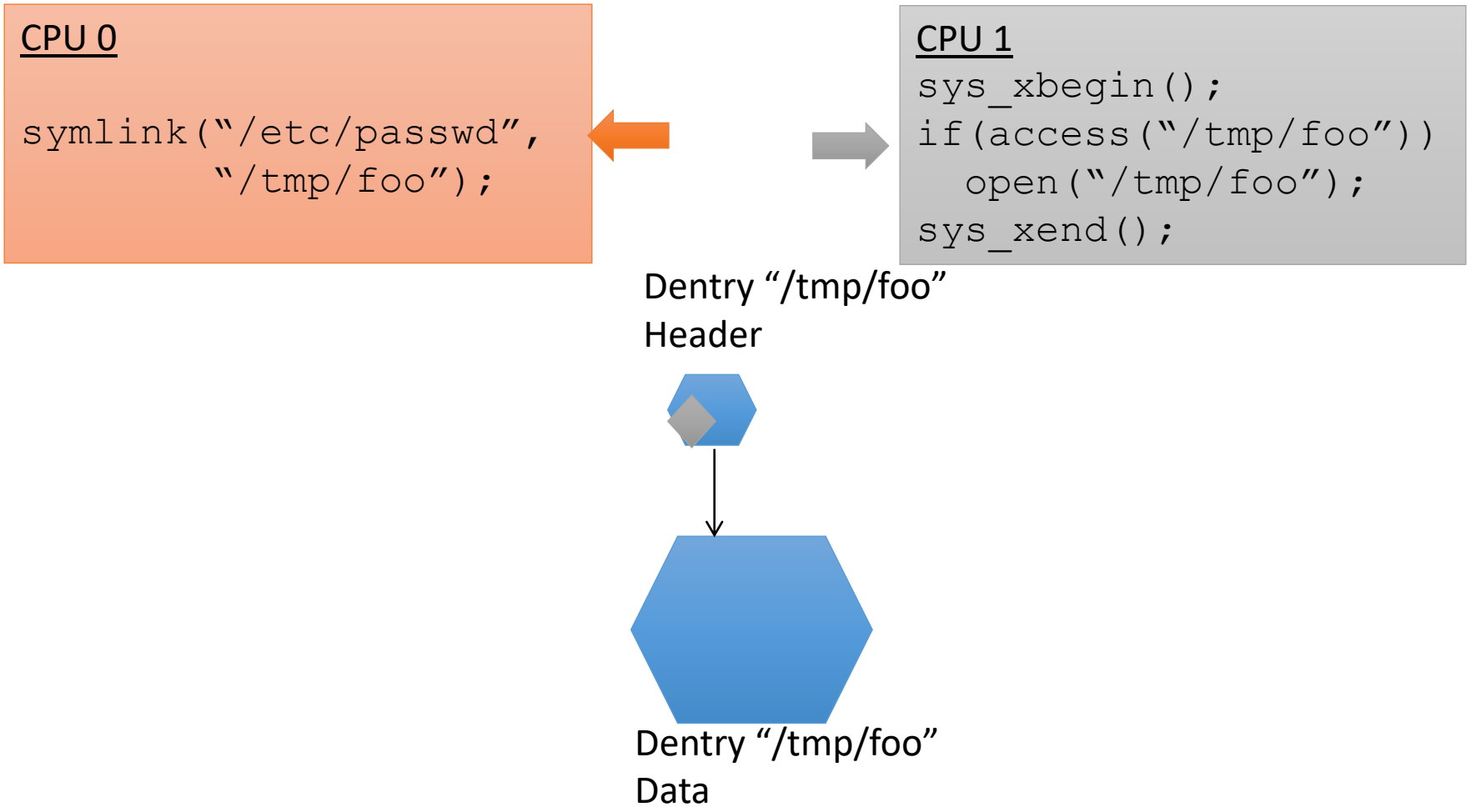
Serializing transactions and non-transactions (strong isolation)

- TxOS mixes transactional and non-tx code
- Supports incremental adoption of transactions
- Problem: Can’t roll back non-transactional syscall
  - Always aborting transaction undermines fairness

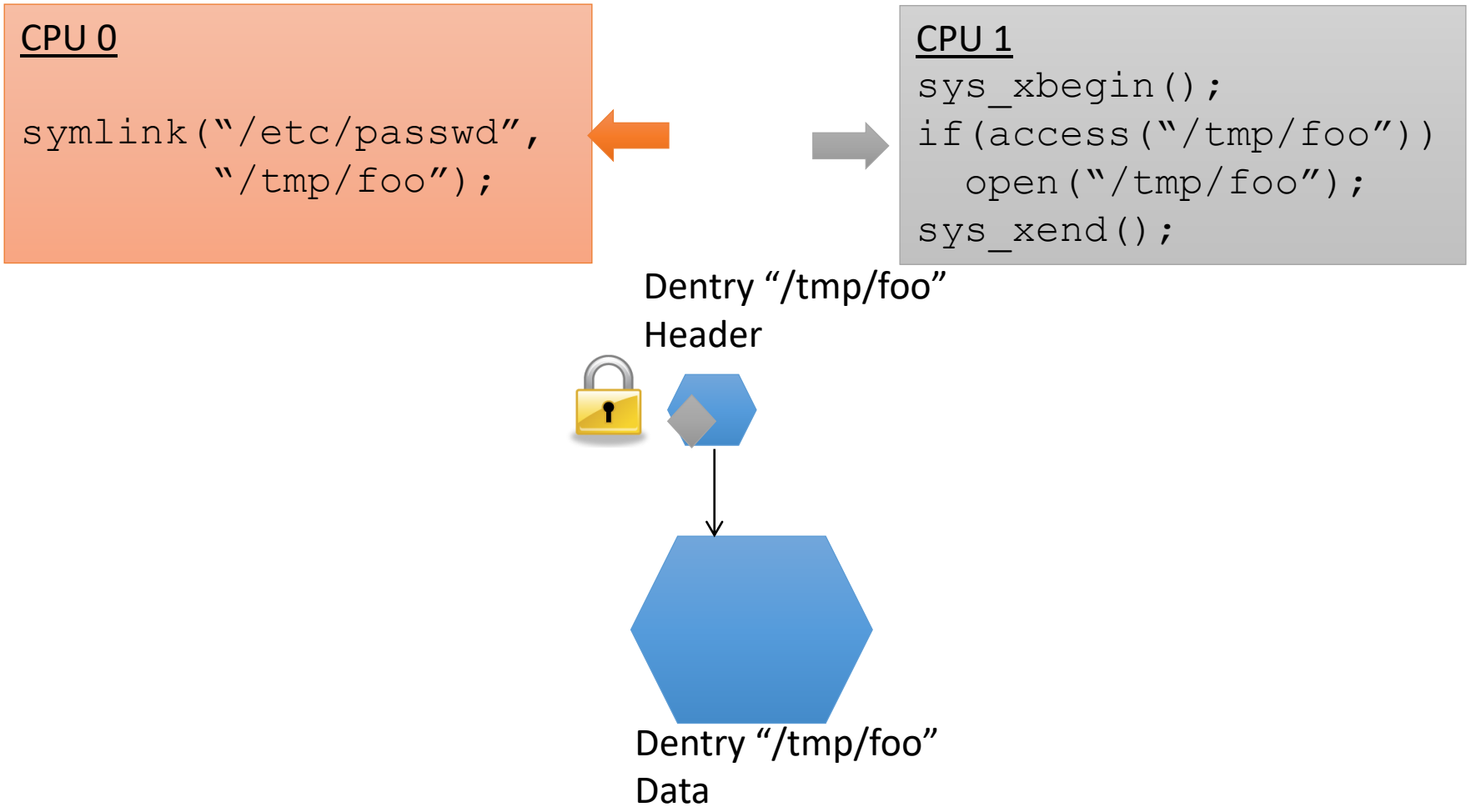
Strong isolation in TxOS



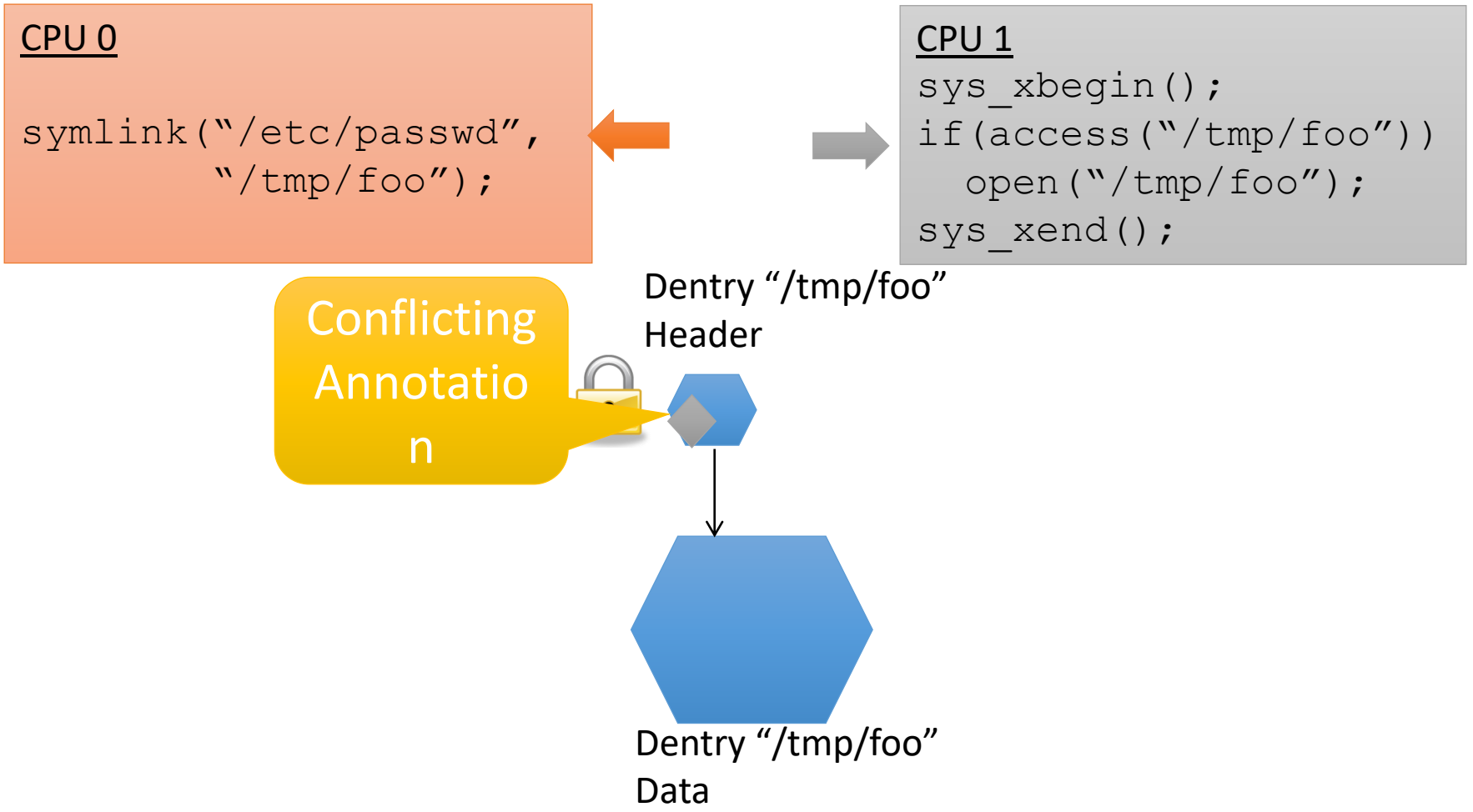
Strong isolation in TxOS



Strong isolation in TxOS



Strong isolation in TxOS



Strong isolation in TxOS

CPU 0

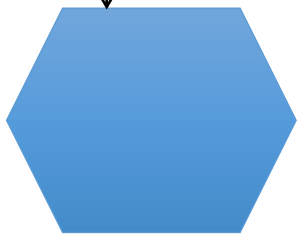
```
symlink("/etc/passwd",
        "/tmp/foo");
```

CPU 1

```
sys_xbegin();
if(access("/tmp/foo"))
  open("/tmp/foo");
sys_xend();
```

Conflicting Annotation

Dentry "/tmp/foo" Header



Dentry "/tmp/foo" Data

Contention Manager

- Options:
  - Abort CPU1
  - Deschedule CPU0

Strong isolation in TxOS

CPU 0

```
symlink("/etc/passwd",
        "/tmp/foo");
```

CPU 1

```
sys_xbegin();
if(access("/tmp/foo"))
  open("/tmp/foo");
sys_xend();
```

Conflicting Annotation

Dentry "/tmp/foo" Header

Dentry "/tmp/foo" Data

Contention Manager

Insight: kernel preemption enables fair contention management

- Options:
  - Abort CPU1
  - Deschedule CPU0

Evaluation

- TxOS design and implementation
- **Evaluation and applications**
  - What is the cost of using transactions?
  - What overheads are imposed on non-transactional applications?
- Related work
- Ongoing and future work

Hardware and benchmarks

- Quadcore 2.66 GHz Intel Core 2 CPU, 4 GB RAM, 7200 RPM SATA Disk

Benchmark	Description
install	install of svn 1.4.4
make	Compile nano 2.06 inside a tx
dpkg	dpkg install OpenSSH 4.6
LFS large/small	Wrap each phase in a tx
RAB	Reimplemeted Andrew Benchmark Each phase in a tx

Transactional software install

```
sys_xbegin();
dpkg –i openssh;
sys_xend();
```

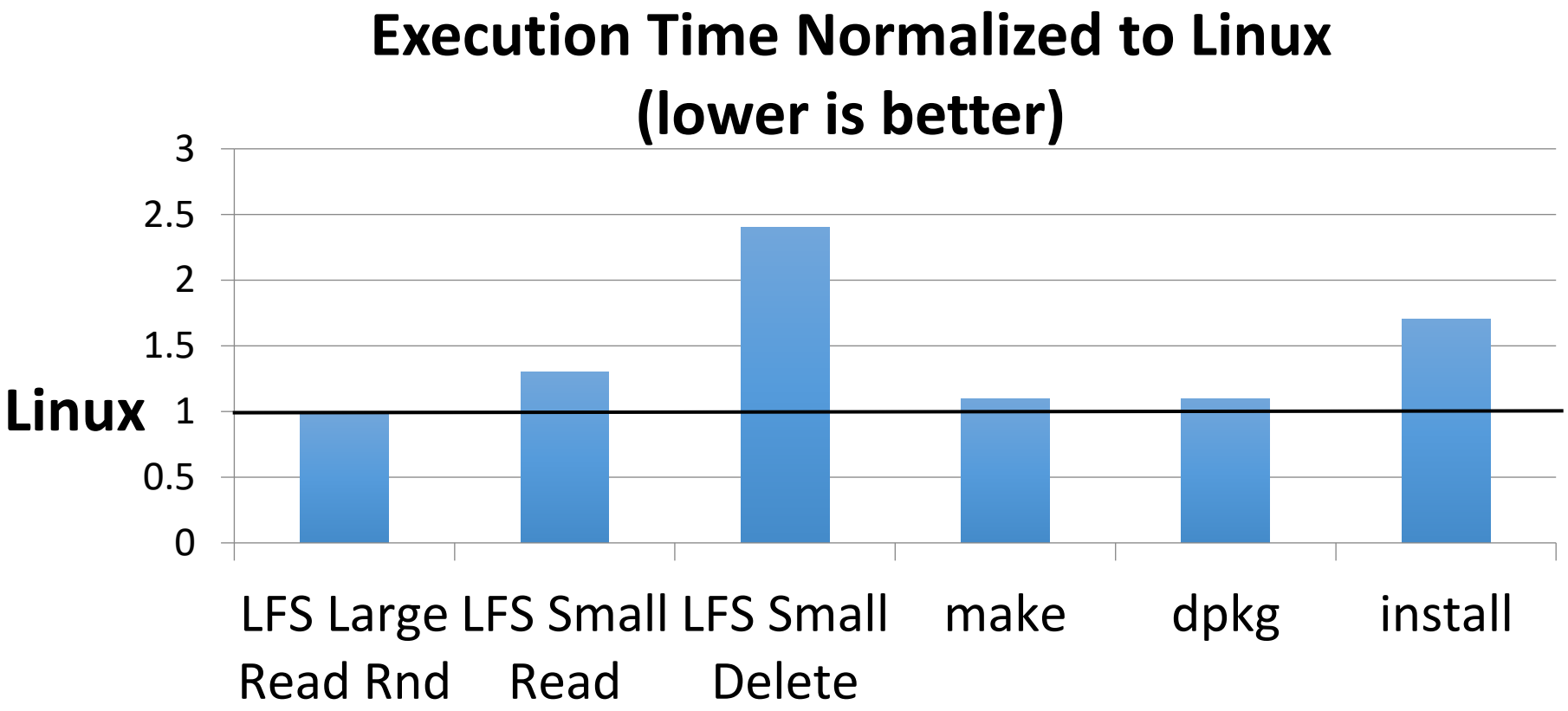
10% overhead

```
sys_xbegin();
install svn;
sys_xend();
```

70% overhead

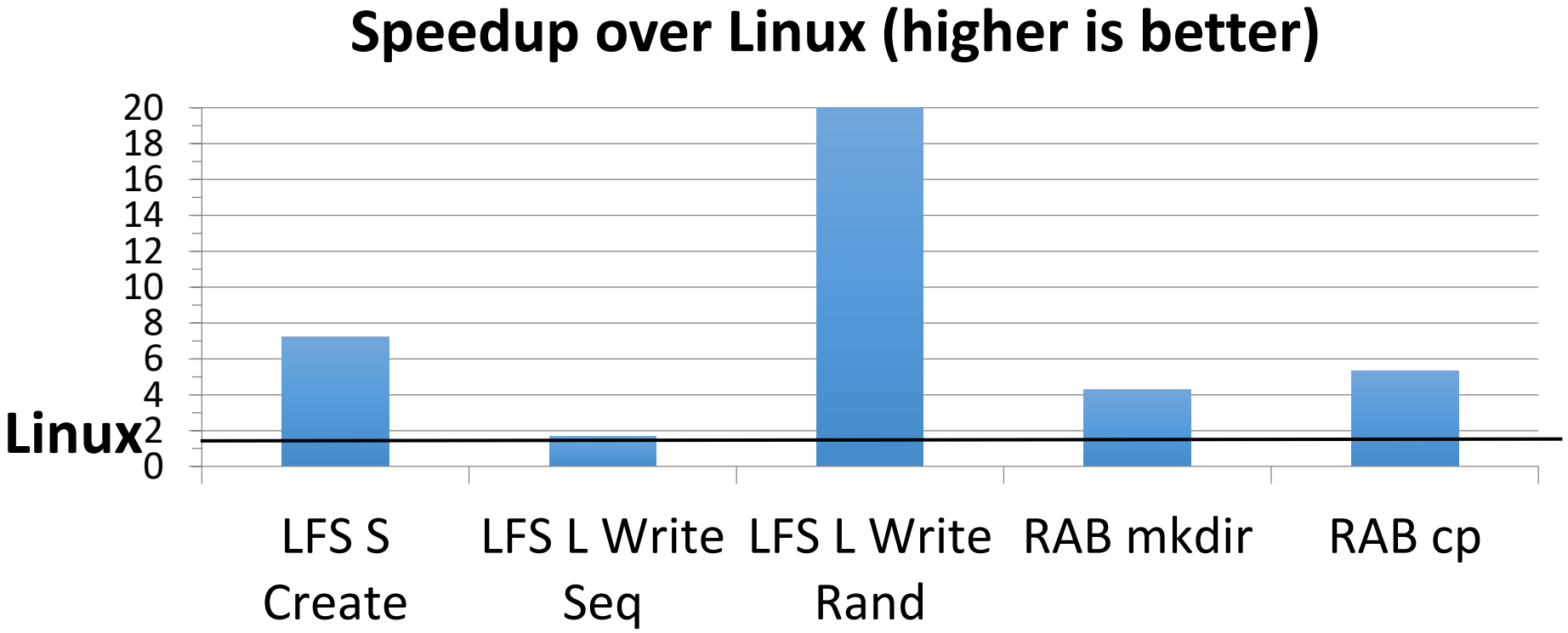
- A failed install is automatically rolled back
  - Concurrent, unrelated operations are unaffected
- System crash: reboot to entire upgrade or none
- Reliable software install impossible without OS help

Acceptable performance overheads



- Memory overheads on LFS large:
  - 13% high, 5% low (kernel)

Write speedups

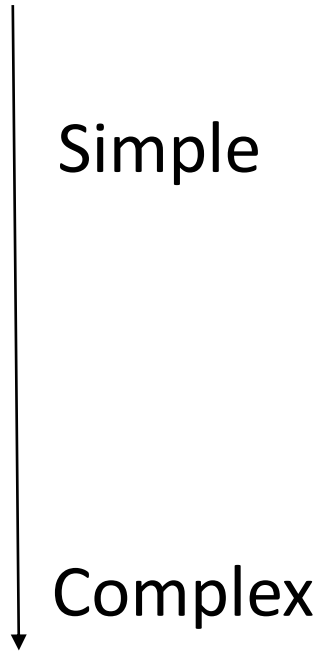


- Better I/O scheduling – not luck
- Tx boundaries provide I/O scheduling hint to OS

Better application design

- How to make consistent updates to stable storage?

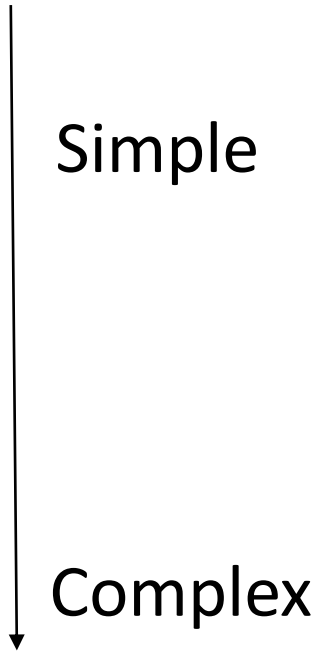
Application	Technique



Better application design

- How to make consistent updates to stable storage?

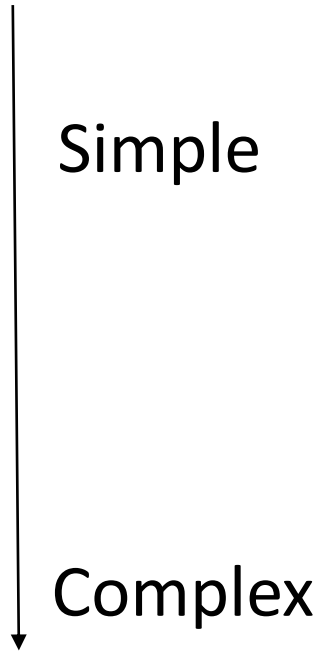
Application	Technique
Editor (1 file)	



Better application design

- How to make consistent updates to stable storage?

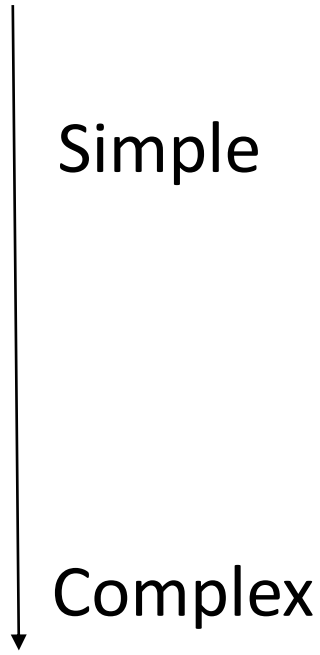
Application	Technique
Editor (1 file)	rename()



Better application design

- How to make consistent updates to stable storage?

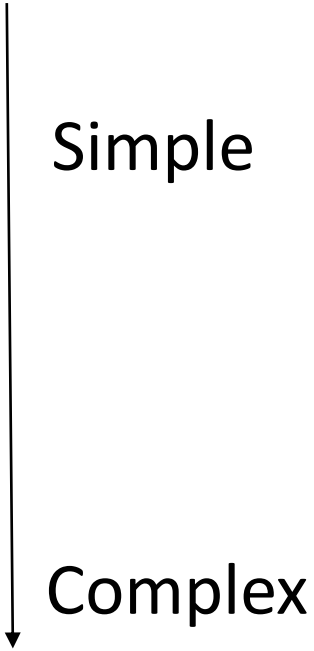
Application	Technique
Editor (1 file)	rename()
Web service (2+ files)	



Better application design

- How to make consistent updates to stable storage?

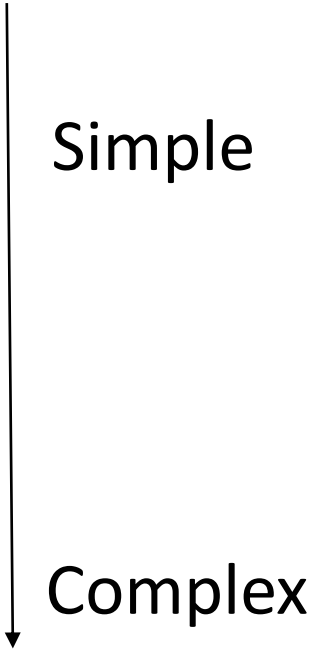
Application	Technique
Editor (1 file)	rename()
Web service (2+ files)	Database



Better application design

- How to make consistent updates to stable storage?

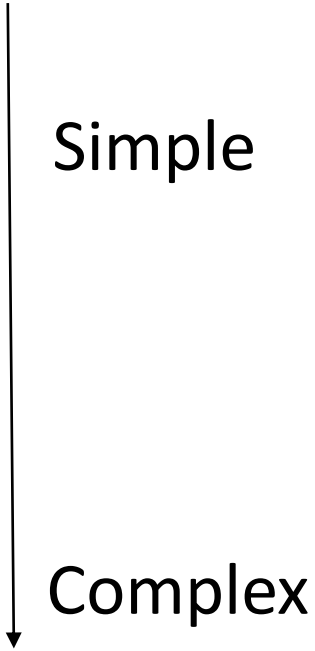
Application	Technique
Editor (1 file)	rename()
Web service (2+ files)	Database
Enterprise data storage	Database



Better application design

- How to make consistent updates to stable storage?

Application	Technique
Editor (1 file)	rename()
Web service (2+ files)	Sys Tx
Enterprise data storage	Database



Lightweight DB alternative

- OpenLDAP directory server
  - Replace BDB backend with transactions + flat files
- 2-4.2x speedup on write-intensive workloads
- Comparable performance on read-only workloads
  - Primarily serviced from memory cache

- System transactions easier to use, perform better

Low non-transactional overheads

- Single system call microbenchmark:
  - 42% mean overhead compared to Linux
- Non-transactional Linux compile: <2% on TxOS
  - System calls infrequent enough that harm on overall execution time is minimal
- Transactions are pay-to-play

Evaluation summary

- Performance overhead of transactions is acceptable
- Developers need transactions

Outline

- TxOS design and implementation
- Evaluation and applications
- **Related work**
  - Previous transactional operating systems
  - Transactional file systems
- Ongoing and future work

Transactional operating systems

- System transactions have been proposed and built
  - LOCUS [Weinstein et al. ’85]
  - QuickSilver [Schmuck and Wylie ‘91]
- Serious usability and performance problems
  - Applied database implementation techniques to OS
- Key contribution: new design and implementation
  - Uphold strong guarantees and good performance

Transactional file systems

- Limited by lack of integration with rest of system
- Examples:
  - Cannot write a binary and execute it in one transaction
  - Forces applications to reason about deadlock in FS
- System API simplicity requires kernel integration

Transactional FS Comparison

Feature	TxF	Amino	Valor	TxOS
Simple API	✗	✓	✗	✓
Can be root file system	✓	✗	✓	✓
Other kernel resources in a transaction	✓	✗	✗	✓
Code reuse across file systems	✗	✗	✓	✓

Outline

- TxOS design and implementation
- Evaluation and applications
- Related work
- **Ongoing and future work**

External adoption of TxOS

- Supporting a release at http://txos.code.csres.utexas.edu
- Users:
  - Researchers at UT-Austin – Oakland Security ‘11
  - External requests
- Promoting adoption in Linux community
  - Talk at Ottawa Linux Symposium 2010

Future work on TxOS

- Build distributed system with TxOS
  - Imap
    - Use sys transactions instead of lock files
    - 40 lines changed of 138,723
    - Better performance 1,500 messages, 1-4 clients, write %
      - Single threaded (12-22%)
      - Multi-threaded (20-106%)
  - Byzantine fault tolerance
    - Faster replication with ordered transactions

Conclusion

- System transactions are practical on modern OSes
  - Incrementally deployable
  - Reasonable overheads
- Solve long standing problems
- Build better concurrent programs

Backup Slides

Transactional file systems

- Provide transactional interface to storage
  - Solves important problems (limited SW install)
  - Adopted by Windows Vista
- Much related research:
  - Transactions in LogFS [Seltzer ‘93], OdeFS [Gehani et al. ’94], Inversion [Olson ‘93], DBFS [Murphy et al. ‘02], Transactional Flash [Gal and Toledo ‘05], Amino [Wright et al. ’07], Valor [Spillane et al. ’09]
  - Complementary focus: efficient atomic, durable disk writes
- TxOS provides transactions for a range of system resources (file system, processes, pipes, signals)

Comparison with Windows KTM

- Complimentary at many points
  - KTM: 2-phase commit, tx file system
  - TxOS: kernel design to eliminate deadlock, reuse code
- KTM: Explicitly transacted system calls
  - Developers cannot write unsupported code
  - More work to adopt
- TxOS: Same code transacted by context
  - Safety: Dynamic checks for unsupported operations

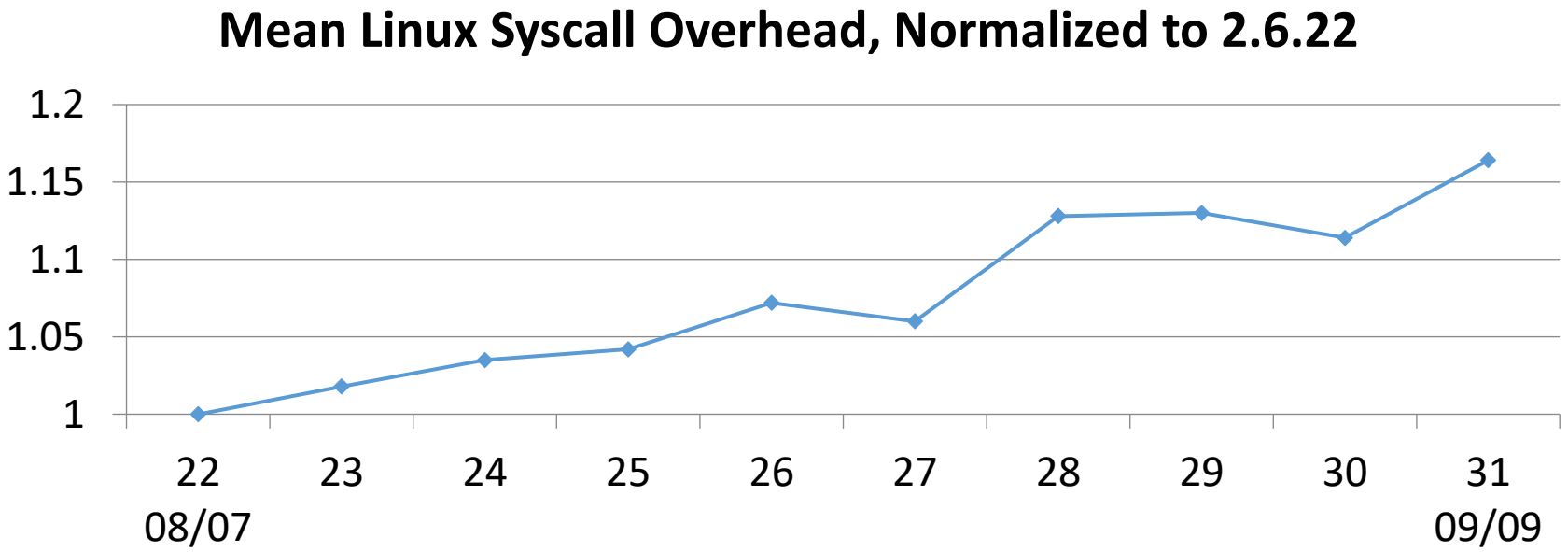
Transactional file systems

- Good idea, difficult to implement
  - Challenging to implement below VFS layer
  - Valor [FAST ‘09] introduces OS support in page cache
- Lack simple abstractions
  - Users must understand implementation details
    - Deadlock detection (Transactional NTFS)
    - Logging and locking mechanism (Valor)
- Lack support for other OS resources in transactions
  - Windows KTM supports transactional registry

Well, rename works ***unless***:

- Using directories
  - EISDIR, ENOTEMPTY, EEXIST
- Concurrent updates
  - Only a swap, not a compare-and-swap
  - “Lost updates” in database parlance
- Using multiple file systems
  - Not atomic across file systems
  - Unix directory tree different file systems

What is practical?



- Feature creep over 2 years costs 16%
- Developers are willing to give up performance for useful features
- Transactions are in same range (14%), more powerful

Challenge 3: Strong isolation

Task 1

```
sys_xbegin();
fd = open("file");
write(fd, "a", 1);
write(fd, "b", 1);
sys_xend();
```

Task 2

```
cat "file"
```

- What does the Task 2 read?

Challenge 3: Strong isolation

Task 1

```
sys_xbegin();
fd = open("file");
write(fd, "a", 1);
write(fd, "b", 1);
sys_xend();
```

Task 2

```
cat "file"
```

- What does the Task 2 read?
  - Ideally, "" or "ab", but not "a"

Challenge 3: Strong isolation

Task 1

```
sys_xbegin();
fd = open("file");
write(fd, "a", 1);
write(fd, "b", 1);
sys_xend();
```

Task 2

```
cat "file"
```

- What does the Task 2 read?
  - Ideally, "" or "ab", but not "a"
- Critical to correctness

Challenge 2: Conflict resolution

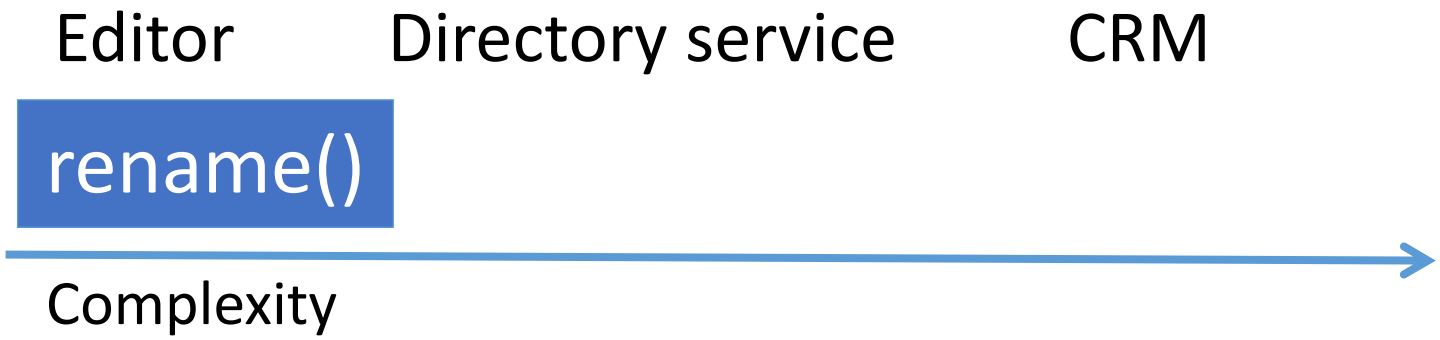
- Goal: Gracefully serialize conflicting accesses
  - Ideally upholding OS scheduling priority
- TxOS: transactions operate on private data copies
  - Leaves shared data structures in a consistent state
- Detect conflicts with annotations on object headers
  - Eliminate two-phase locking and deadlock

Transactions 101

- Isolation: speculatively modified data is not visible outside of a transaction
- Serializable: effects of concurrent transactions are equivalent to a serial execution of the transactions
- Conflict: data access that violates serializability
  - Most often a concurrent read and write of same datum
- Safety follows from enforcing serializability
  - Too much serialization hurts performance

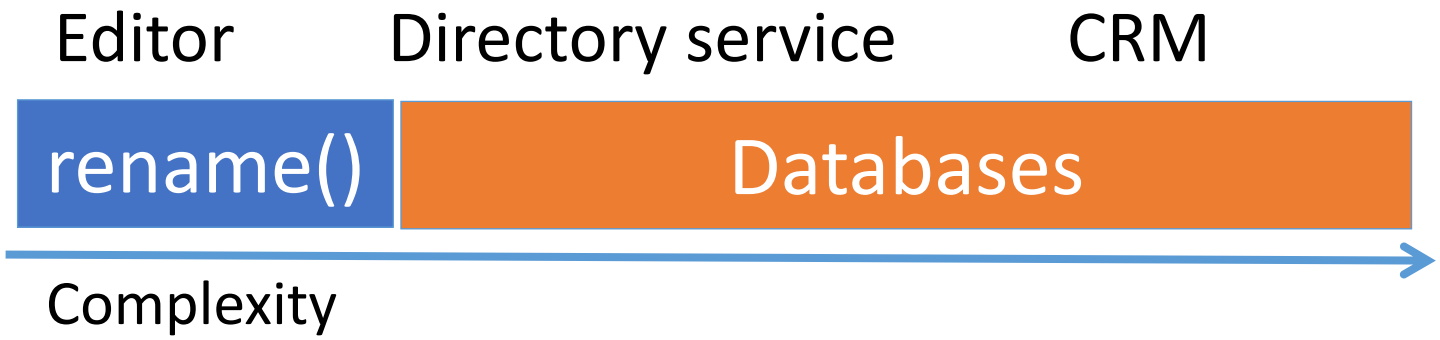
Example 1: better application design

- How to make consistent updates to stable storage?



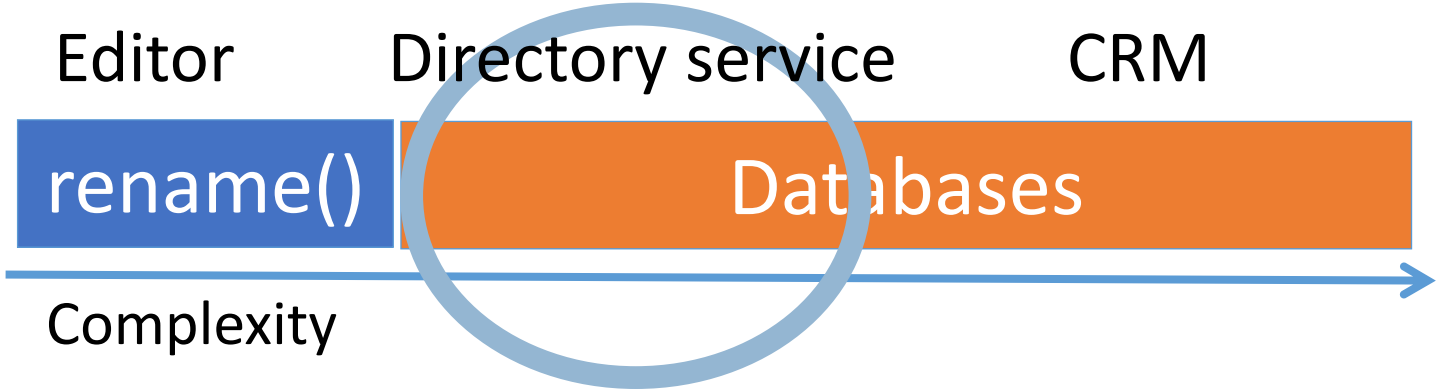
Example 1: better application design

- How to make consistent updates to stable storage?



Example 1: better application design

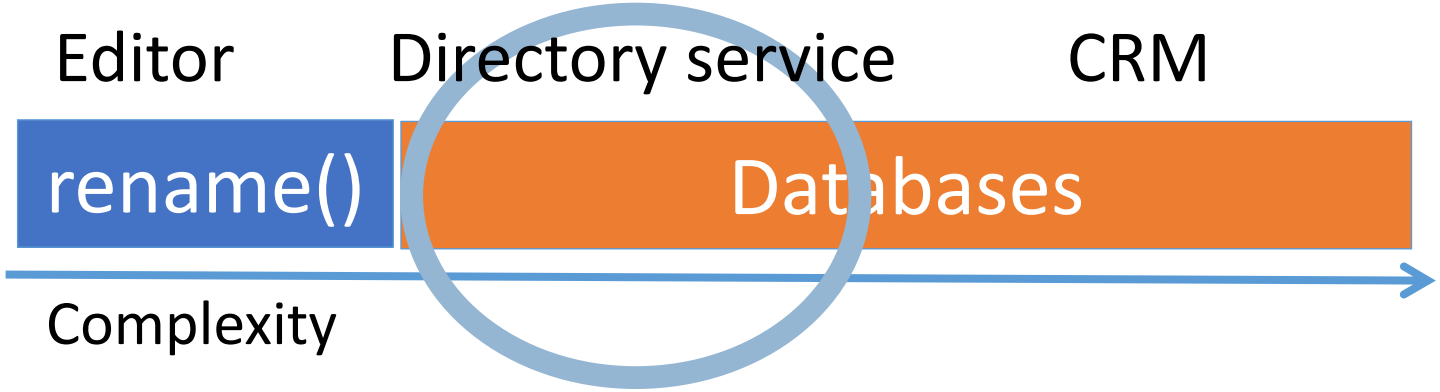
- How to make consistent updates to stable storage?



- Databases overkill for middle ground

Example 1: better application design

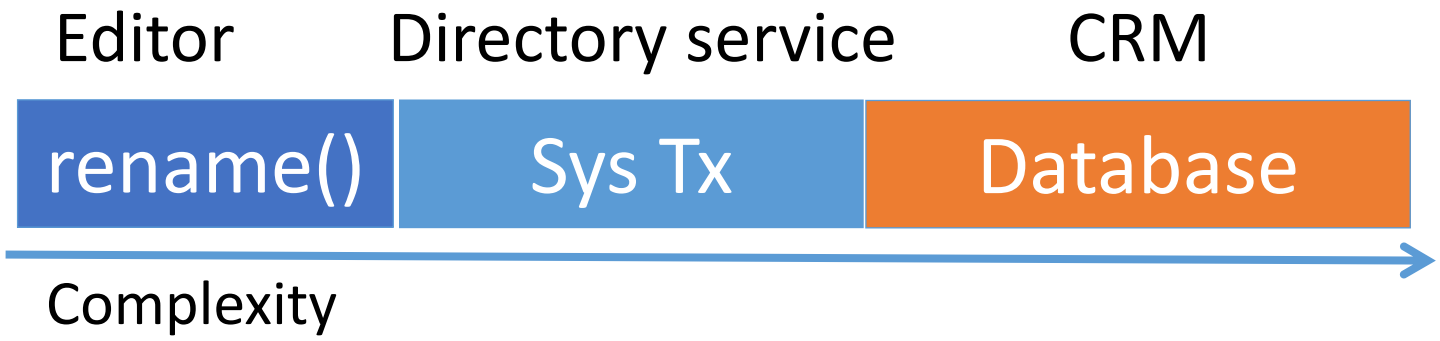
- How to make consistent updates to stable storage?



- Databases overkill for middle ground
- System transactions + flat files can be a lighter database alternative

Example 1: better application design

- How to make consistent updates to stable storage?



- Databases overkill for middle ground
- System transactions + flat files can be a lighter database alternative

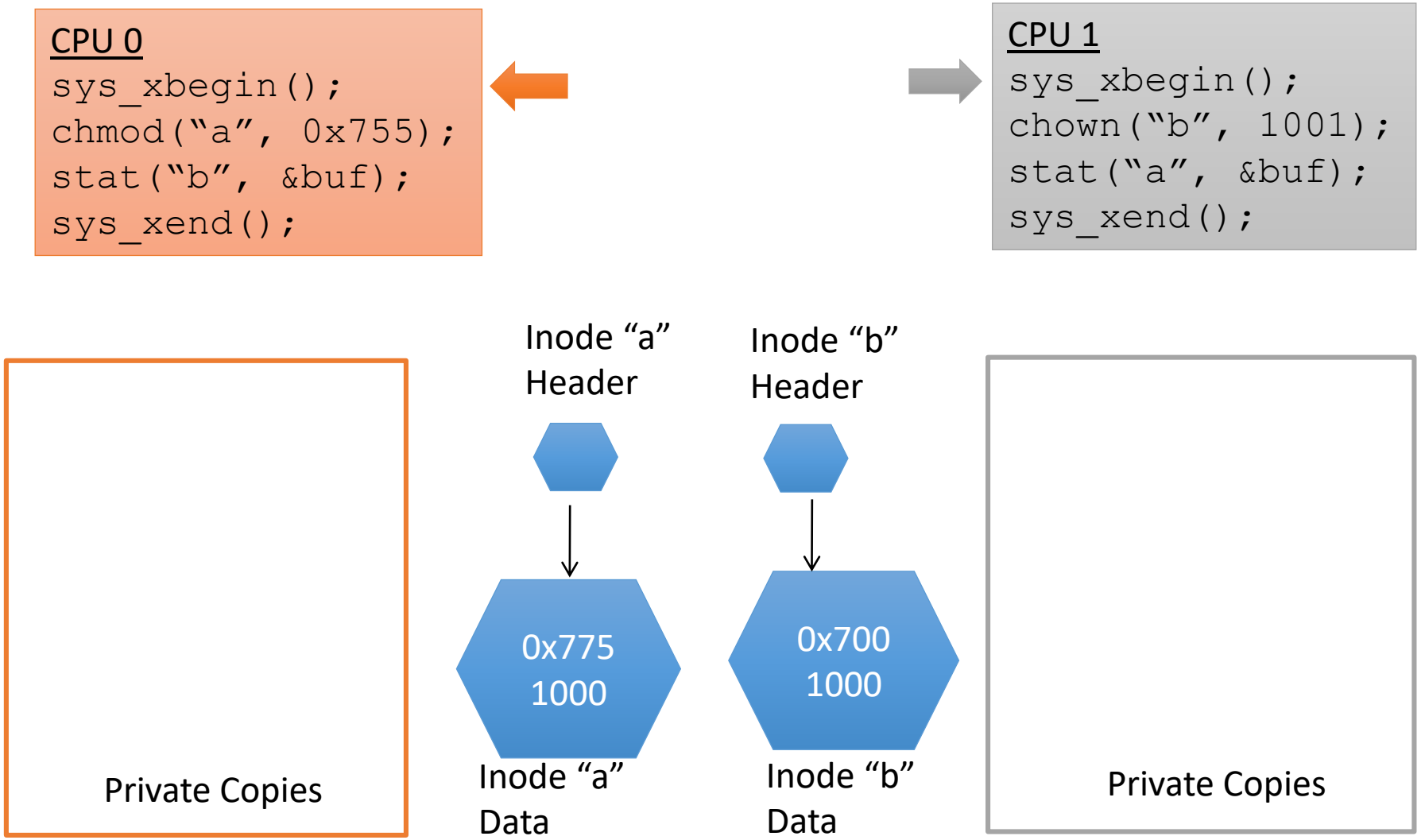
Conflict management in TxOS

- Replace 2PL with annotations on object header
  - Lock objects only for initial copy and commit
  - Eliminate deadlock
- Shared data structures always in committed state
  - Eliminate priority inversion on restarts
- Split objects into header and data for efficiency

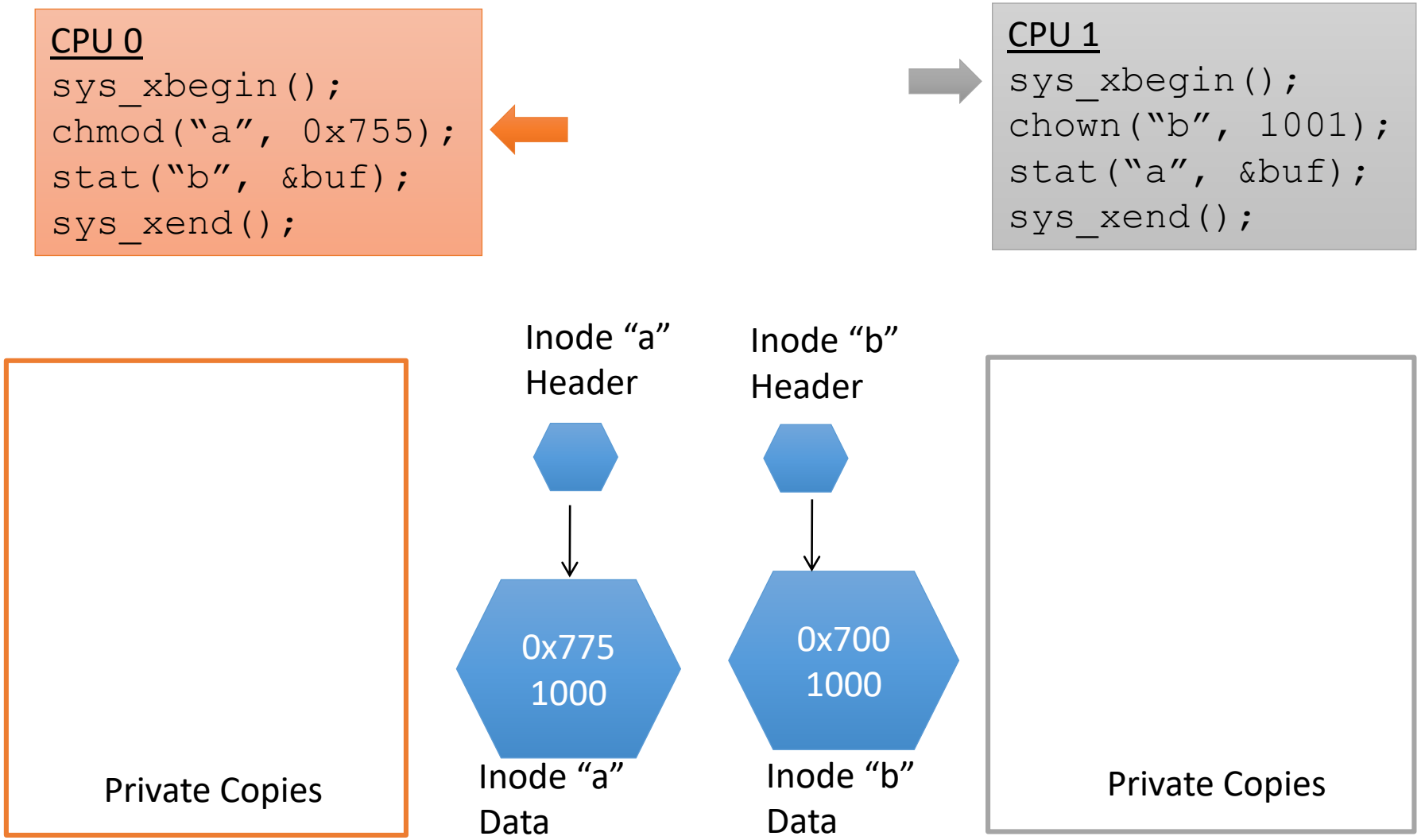
TxOS design

- Transactions operate on private copies of data
  - Made efficient by splitting objects
- Restarts do not stall other threads
- Deadlock-free
  - Lock only to copy object and commit
  - Follows existing locking discipline

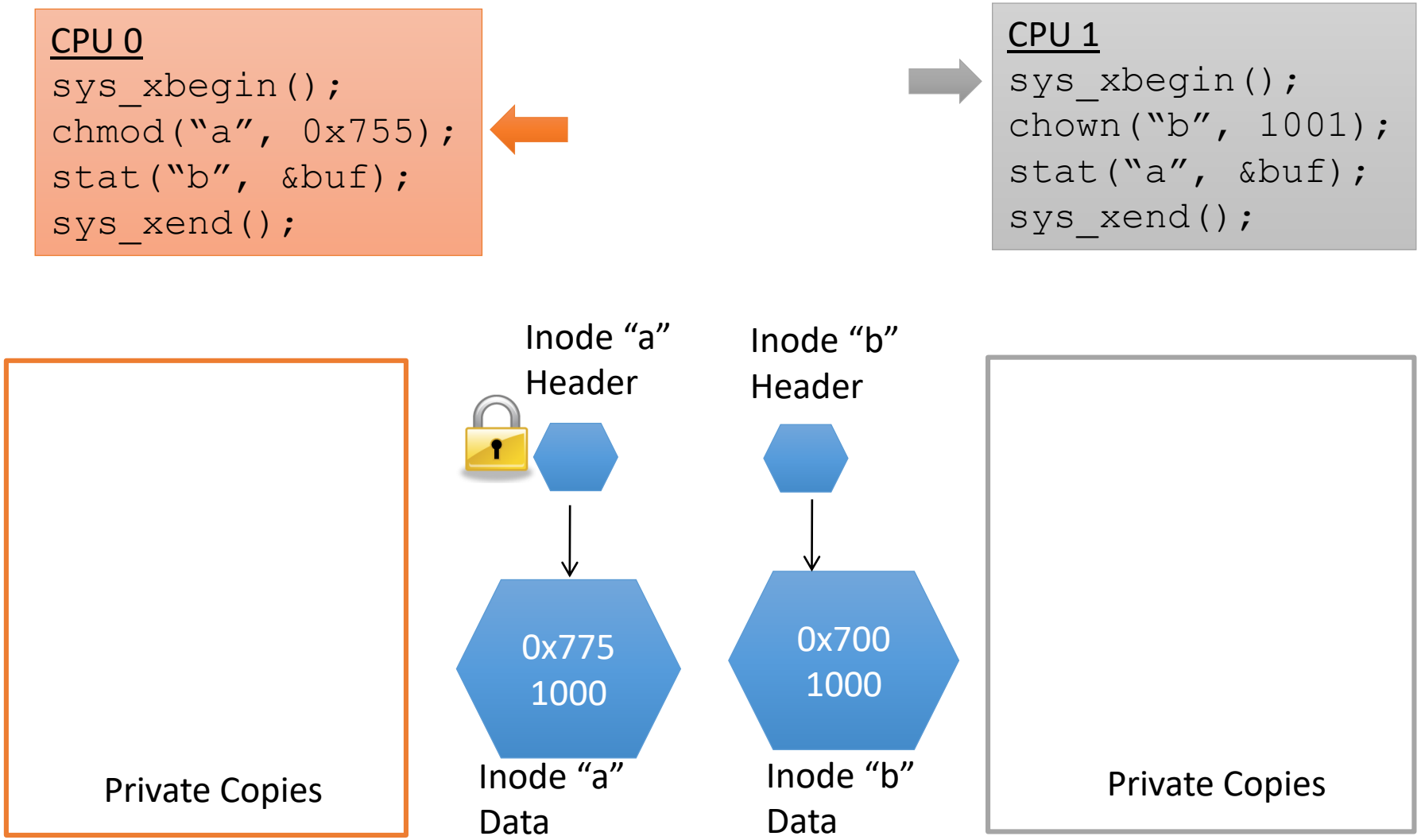
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x775 1000
0x700 1000
Inode "a" Data
Inode "b" Data
Private Copies
Private Copies

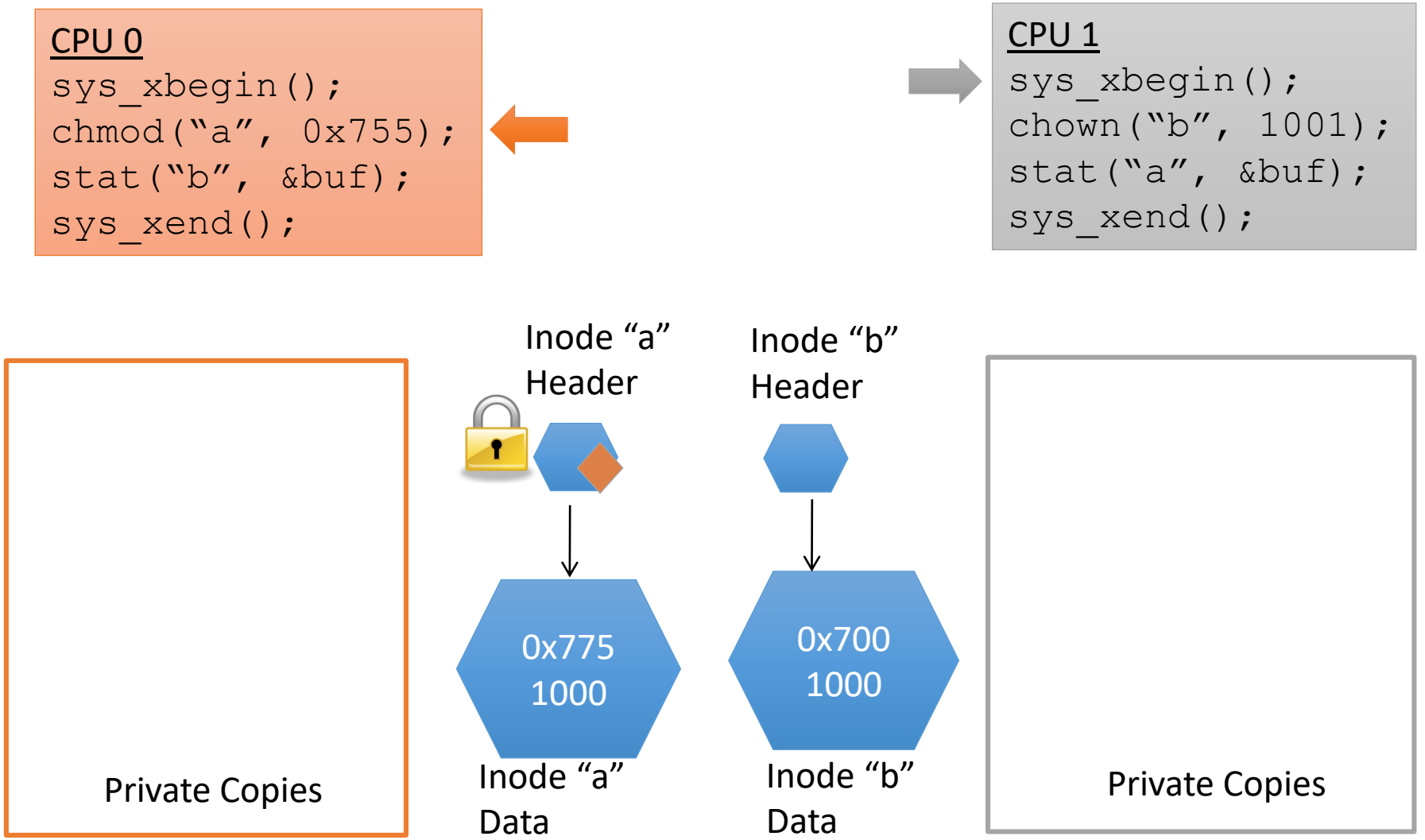
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x775 1000
0x700 1000
Inode "a" Data
Inode "b" Data
Private Copies
Private Copies

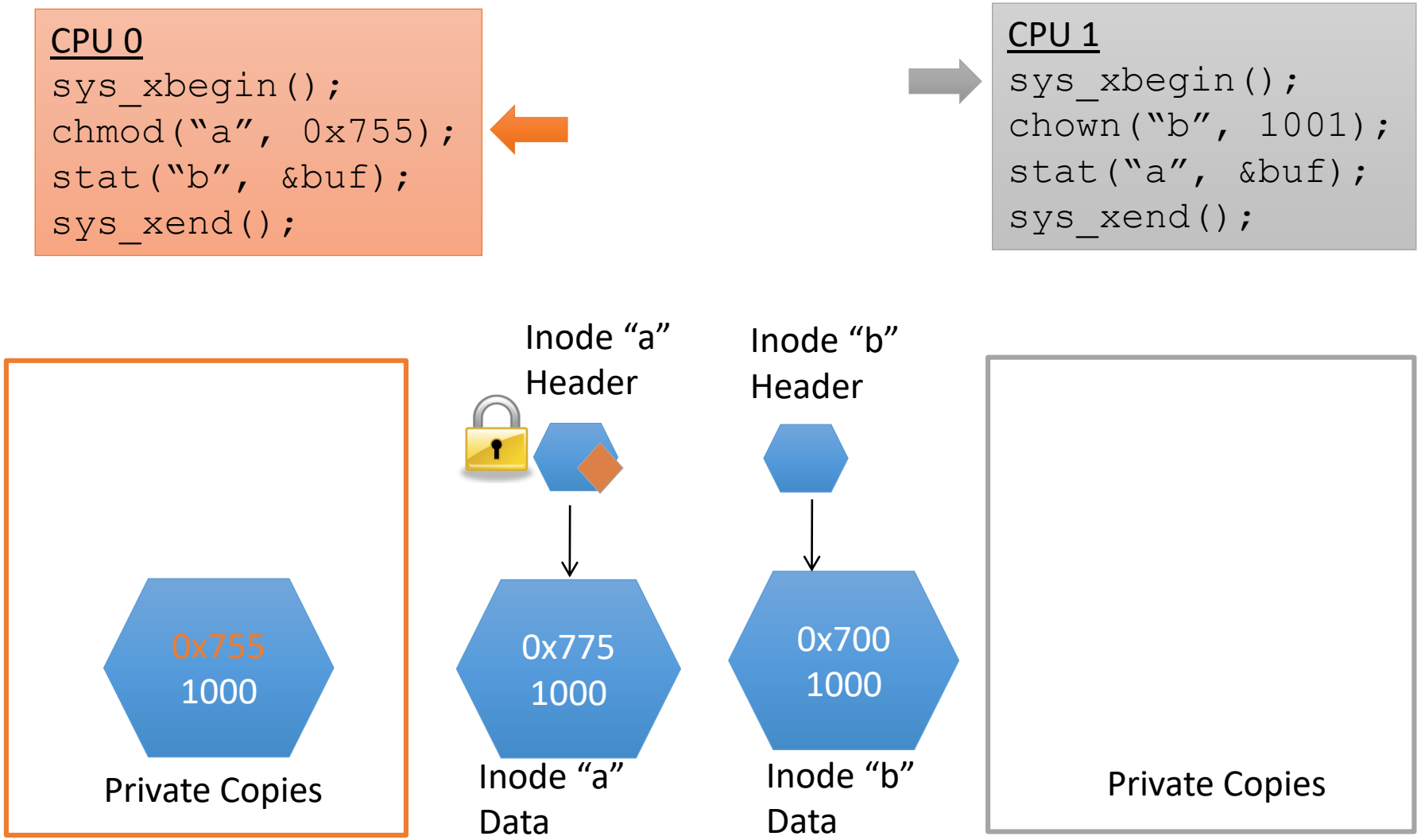
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a"
Header
Inode "b"
Header
0x775
1000
0x700
1000
Private Copies
Inode "a"
Data
Inode "b"
Data
Private Copies

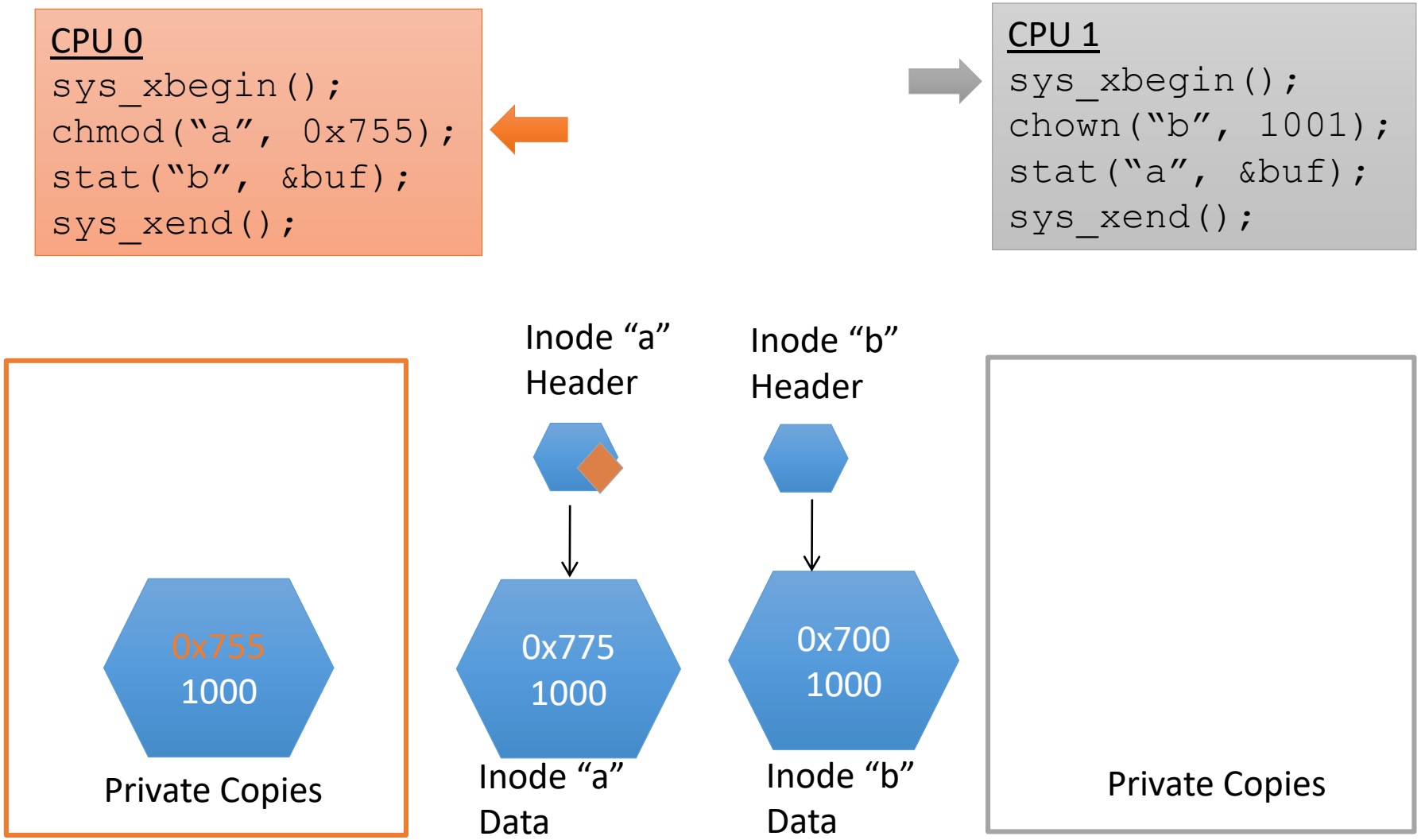
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a"
Header
Inode "b"
Header
0x775
1000
0x700
1000
Private Copies
Inode "a"
Data
Inode "b"
Data
Private Copies

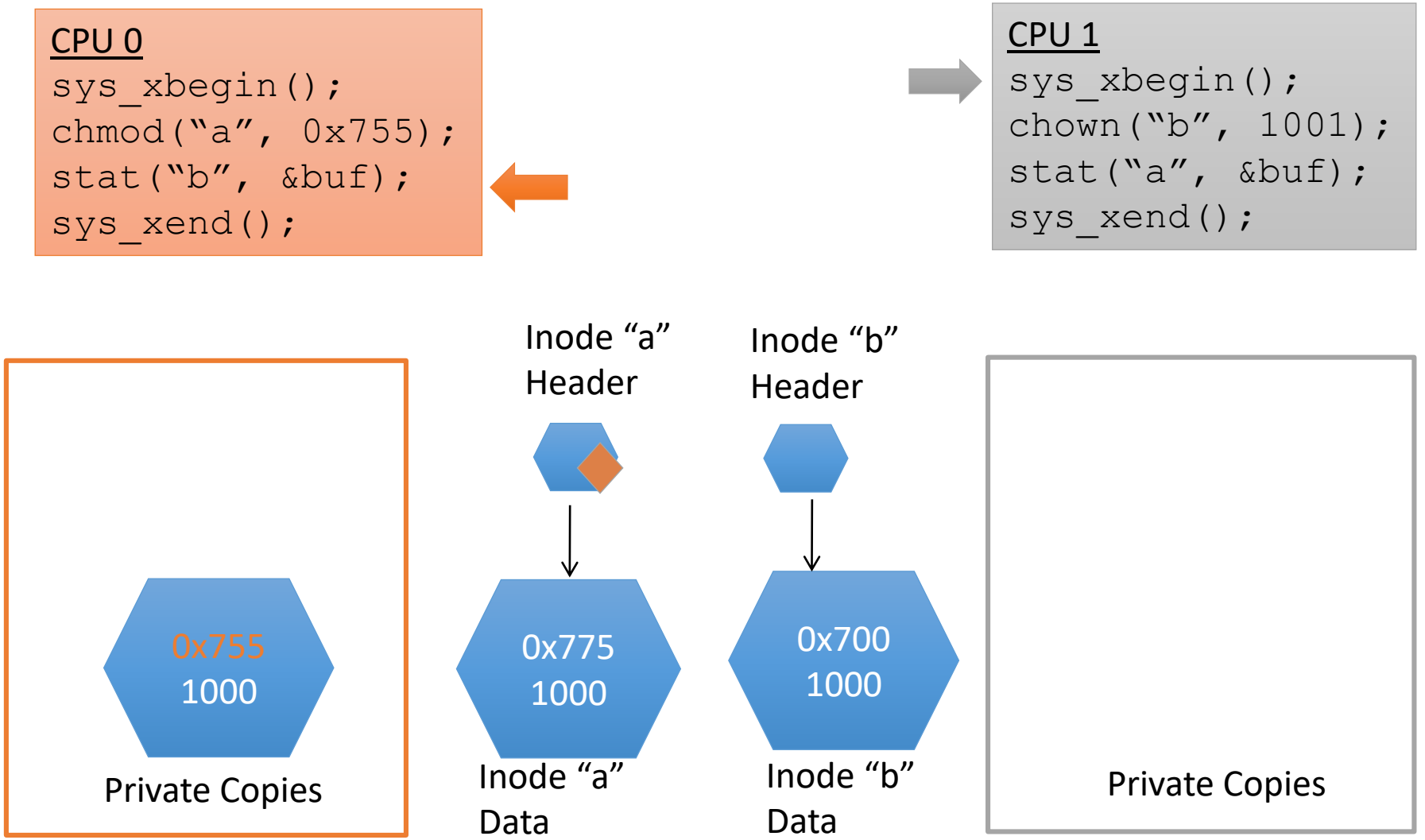
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a"
Header
Inode "b"
Header
0x755
1000
Private Copies
0x775
1000
Inode "a"
Data
0x700
1000
Inode "b"
Data
Private Copies

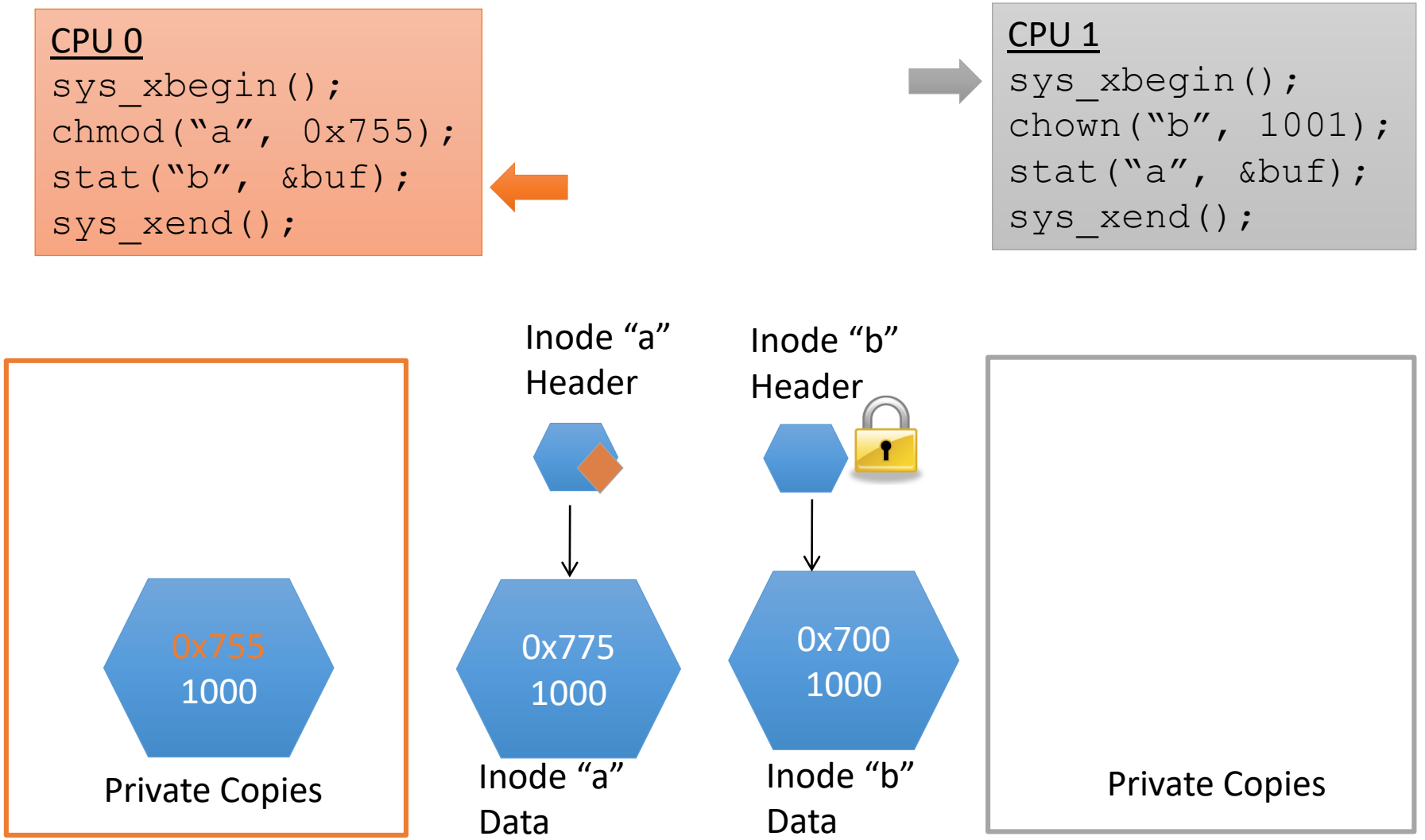
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
Private Copies

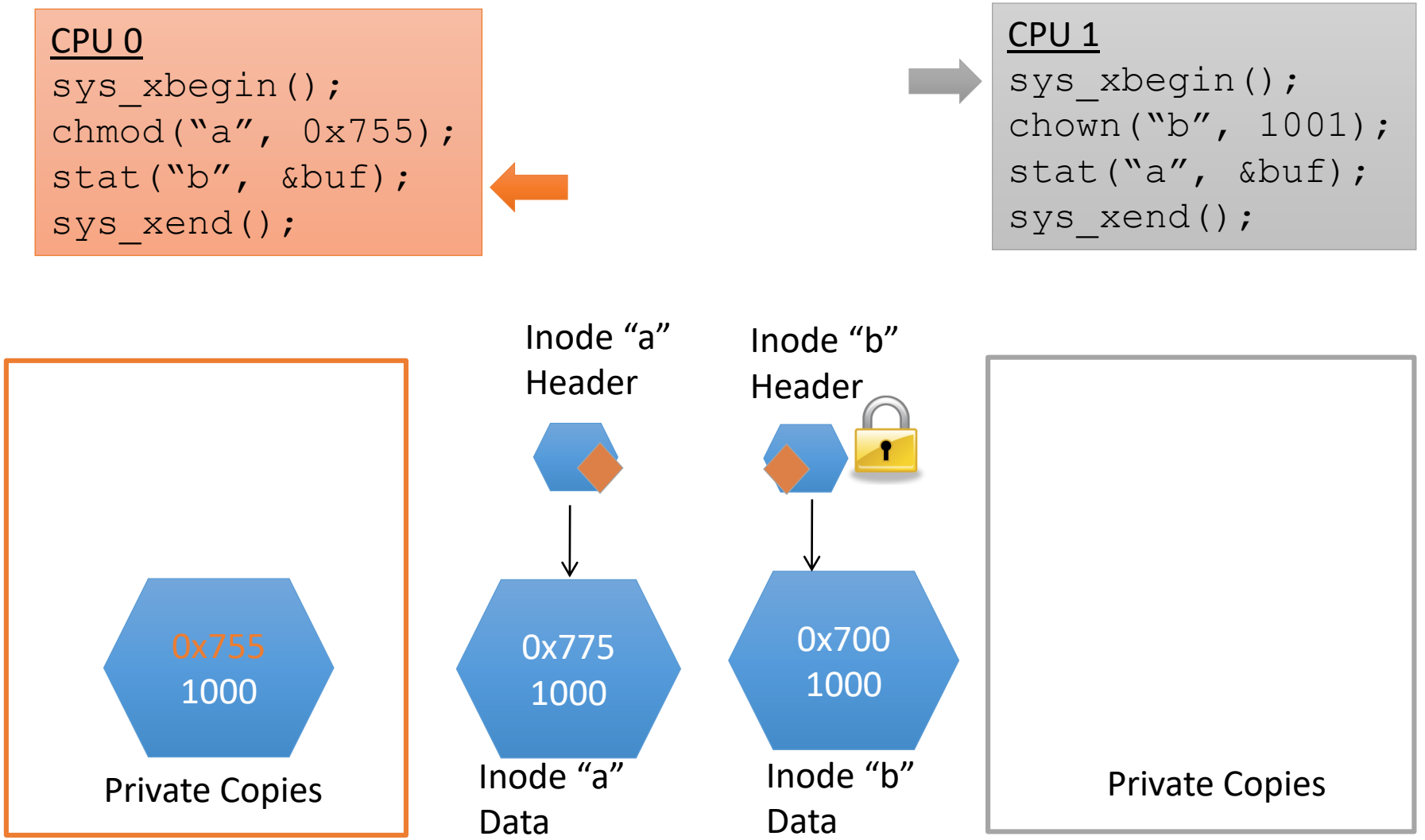
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
Private Copies

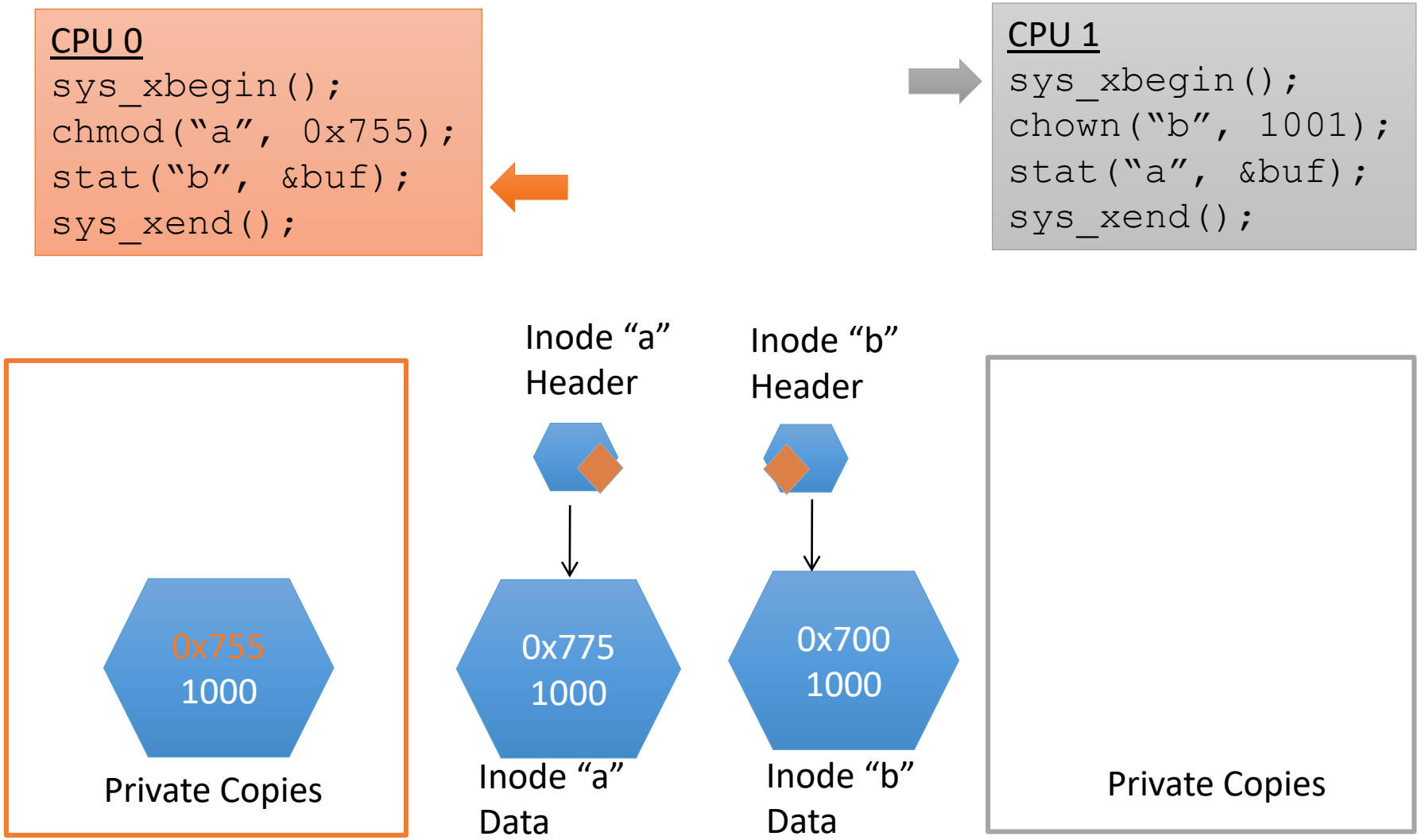
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
Private Copies

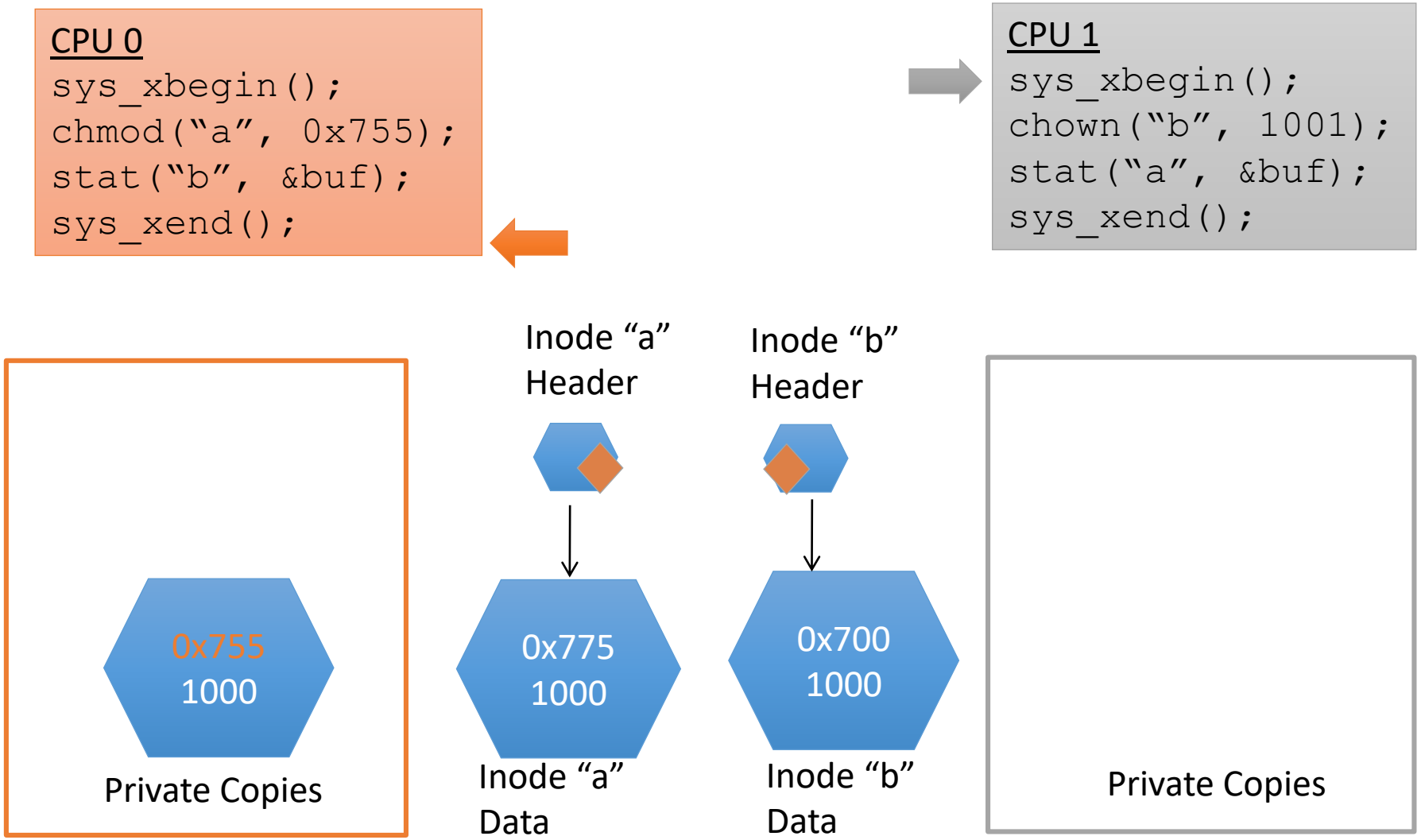
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
Private Copies

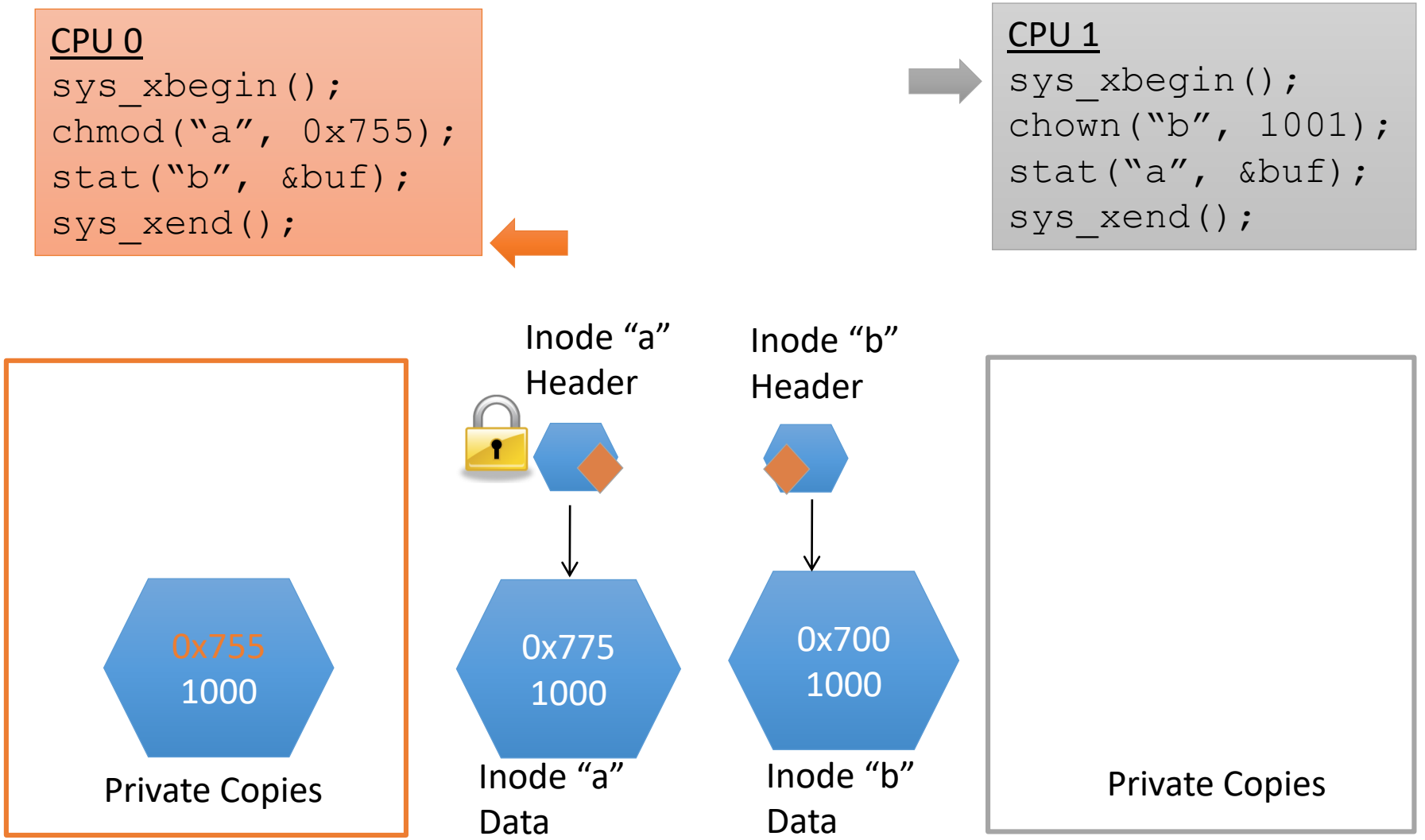
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
Private Copies

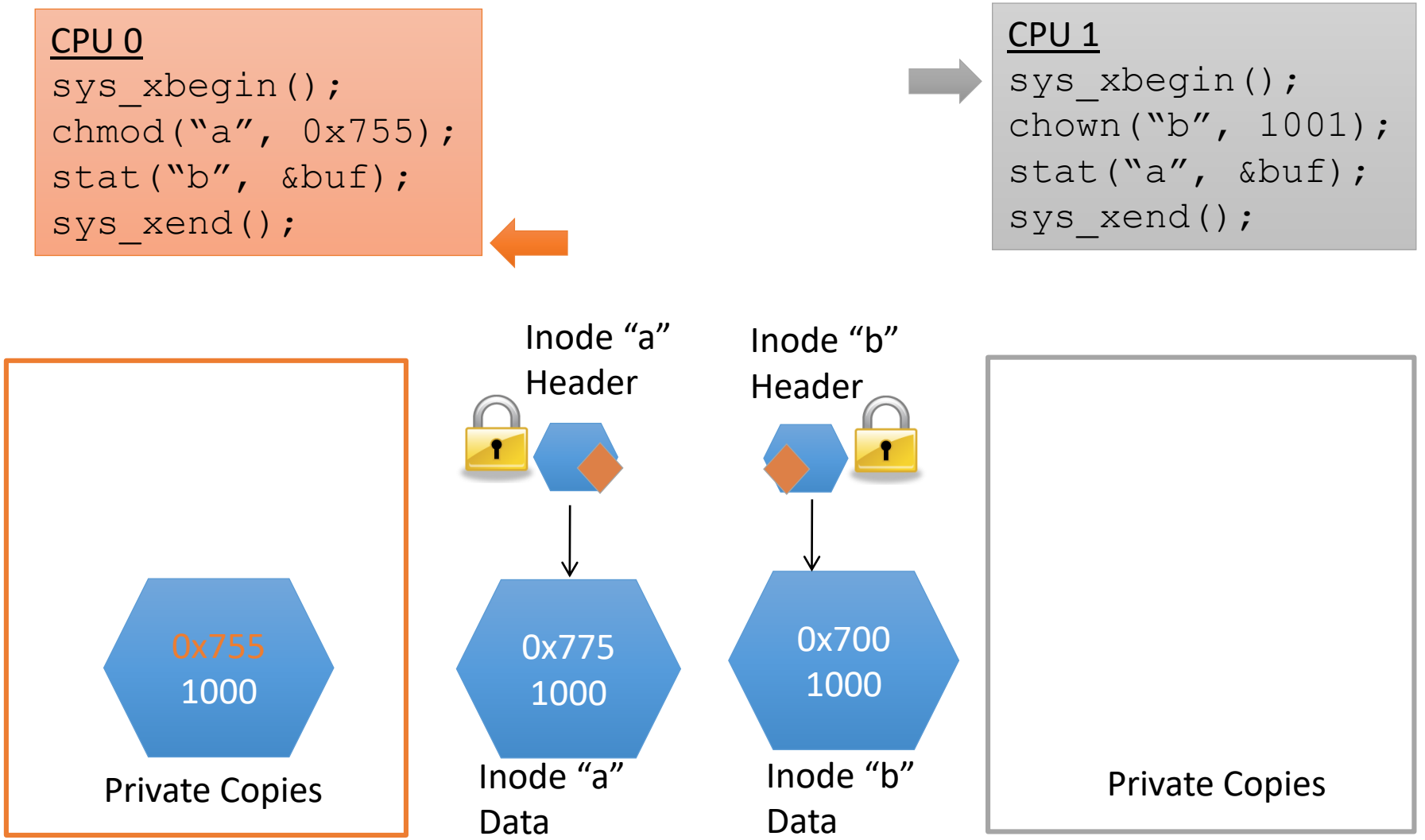
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
Private Copies

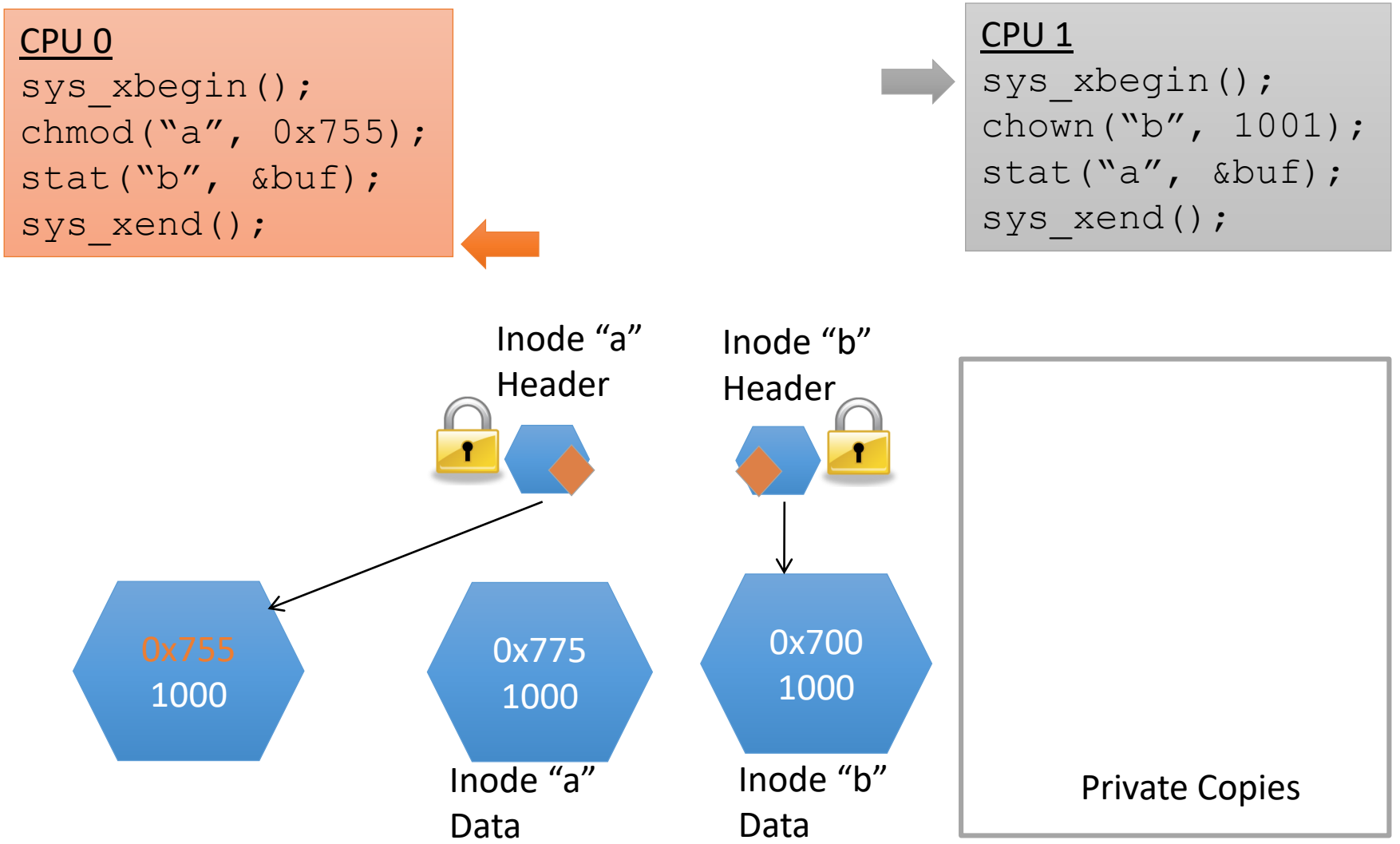
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a"
Header
Inode "b"
Header
0x755
1000
Private Copies
0x775
1000
Inode "a"
Data
0x700
1000
Inode "b"
Data
Private Copies

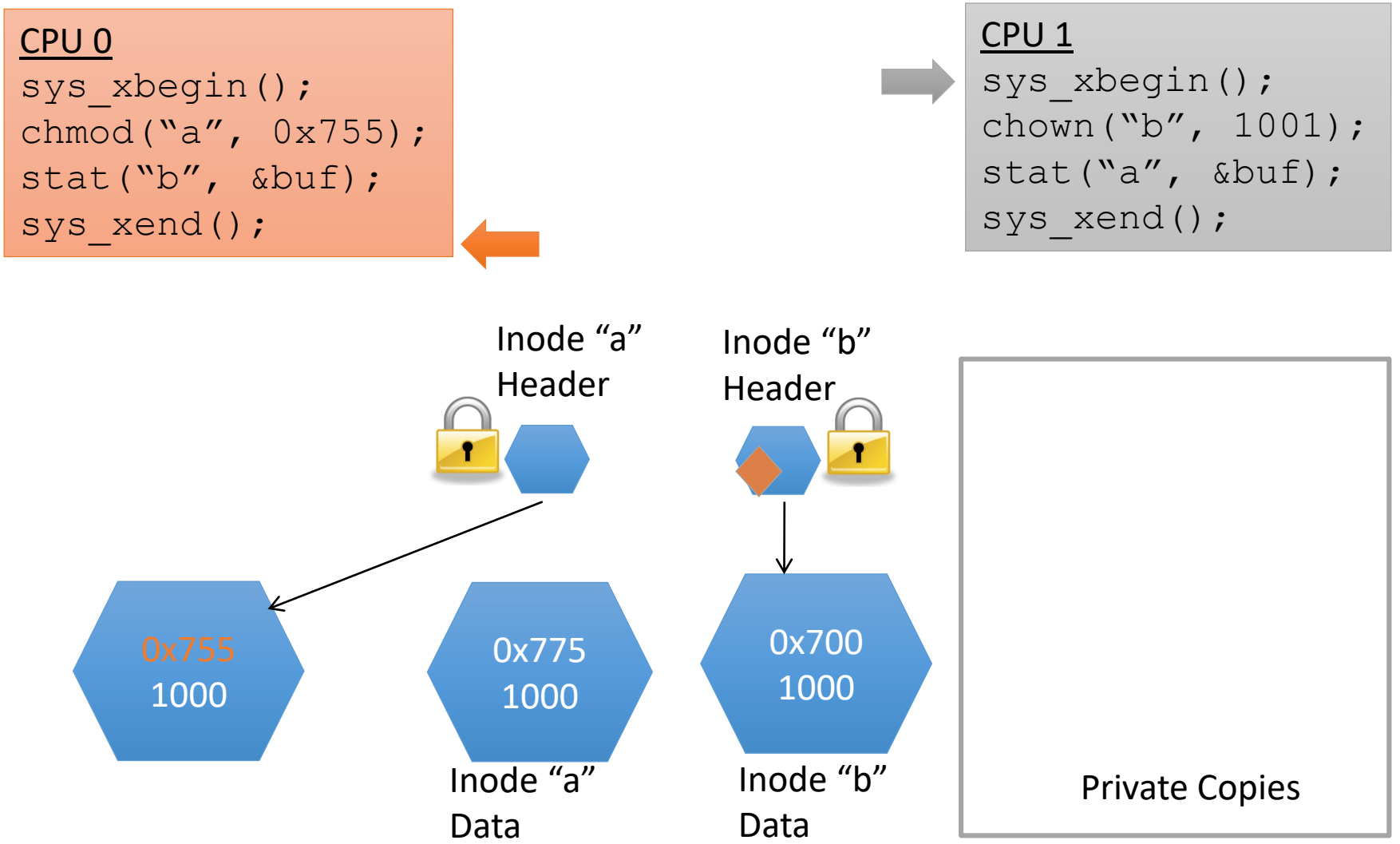
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a"
Header
Inode "b"
Header
0x755
1000
Private Copies
0x775
1000
Inode "a"
Data
0x700
1000
Inode "b"
Data
Private Copies

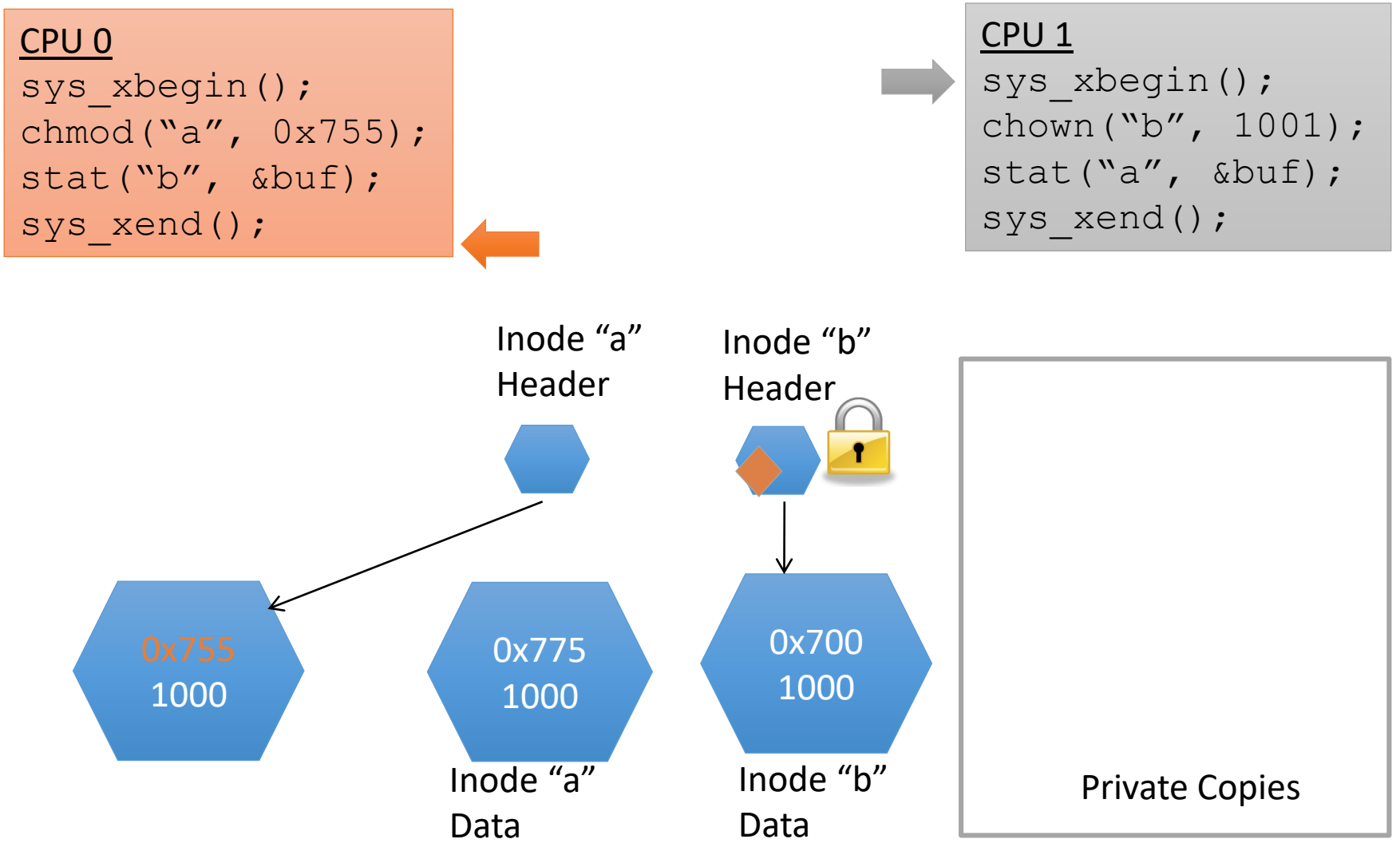
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a"
Header
Inode "b"
Header
0x755
1000
0x775
1000
0x700
1000
Inode "a"
Data
Inode "b"
Data
Private Copies

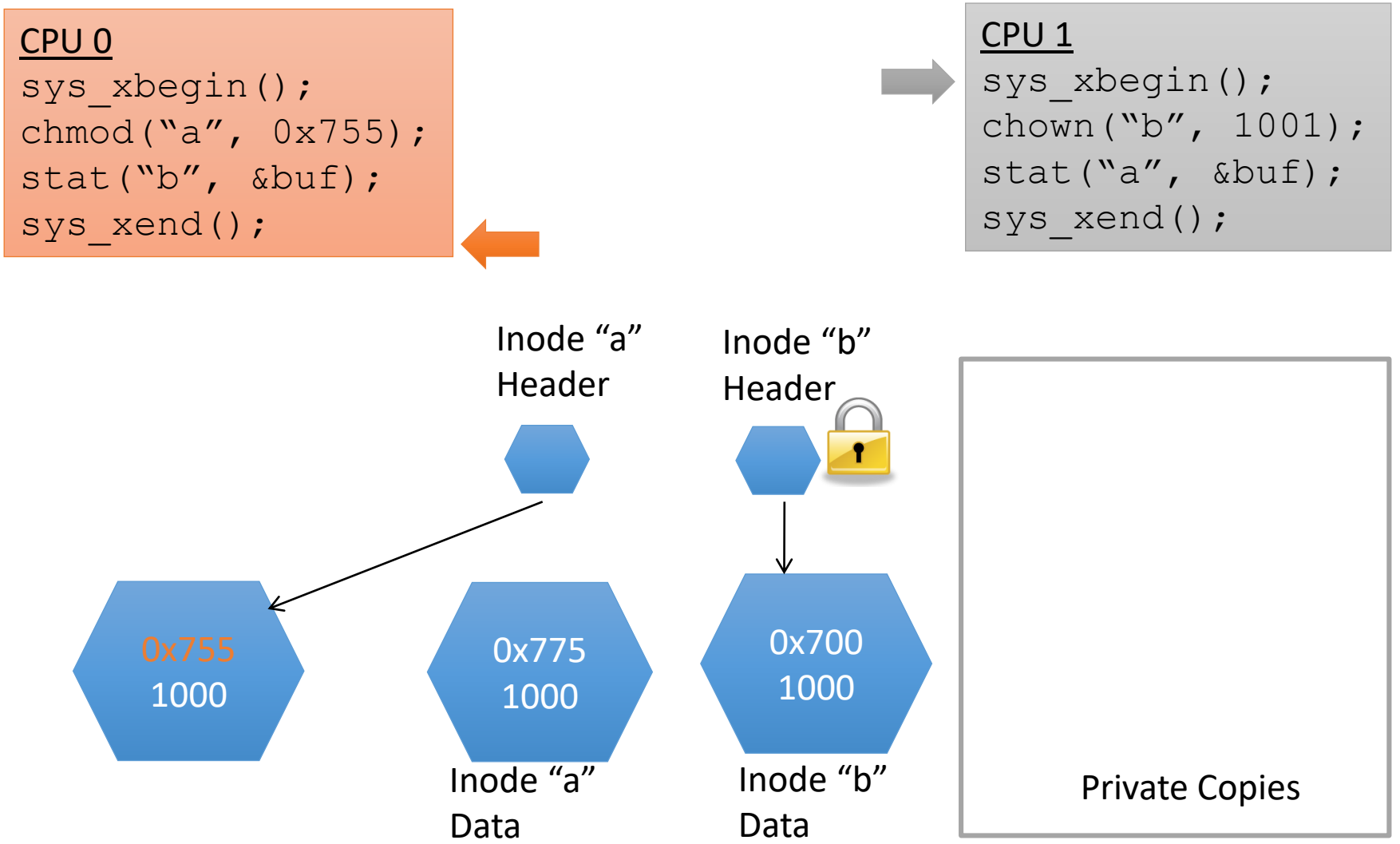
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a"
Header
Inode "b"
Header
0x755
1000
0x775
1000
0x700
1000
Inode "a"
Data
Inode "b"
Data
Private Copies

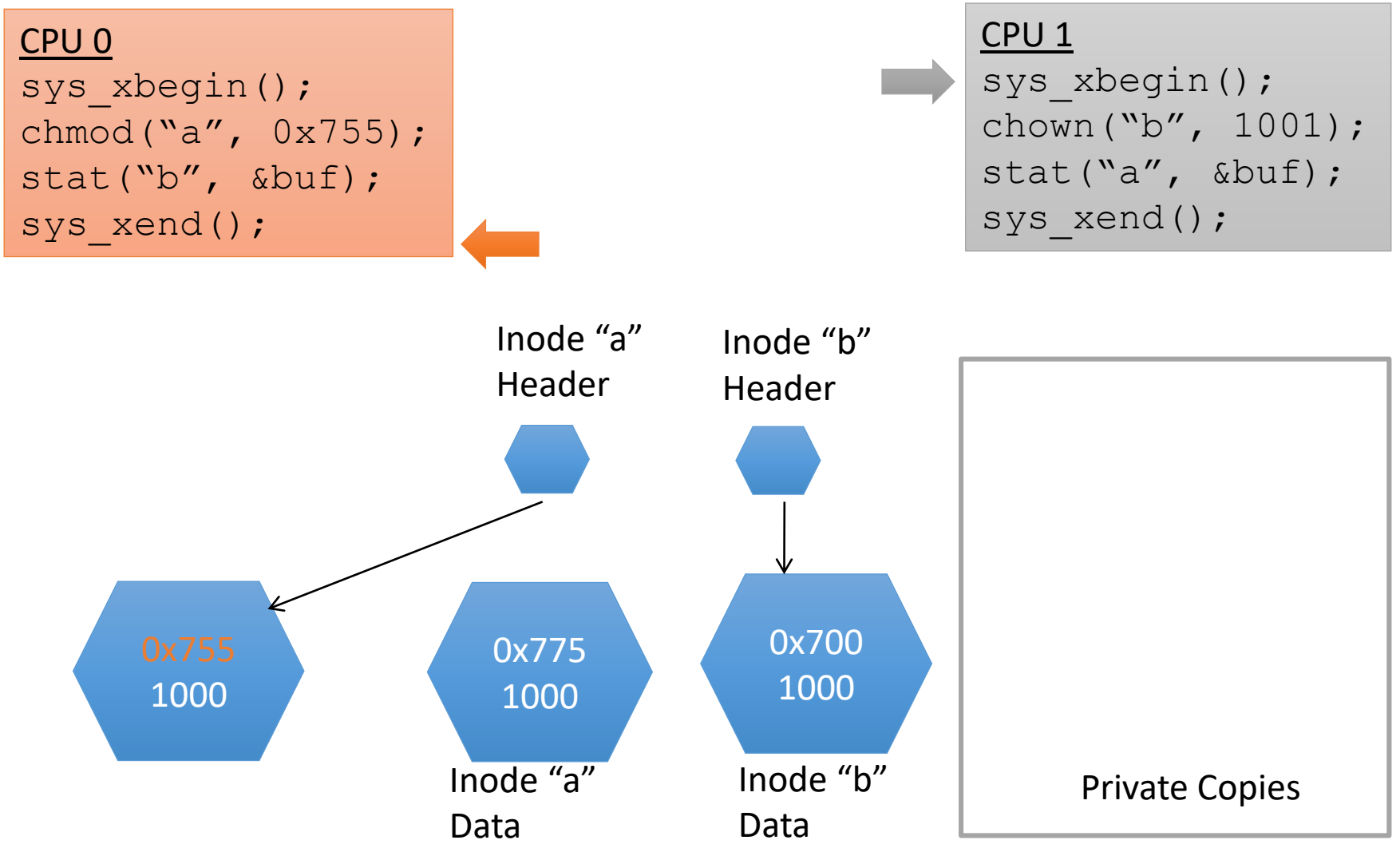
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
0x775
1000
0x700
1000
Inode "a" Data
Inode "b" Data
Private Copies

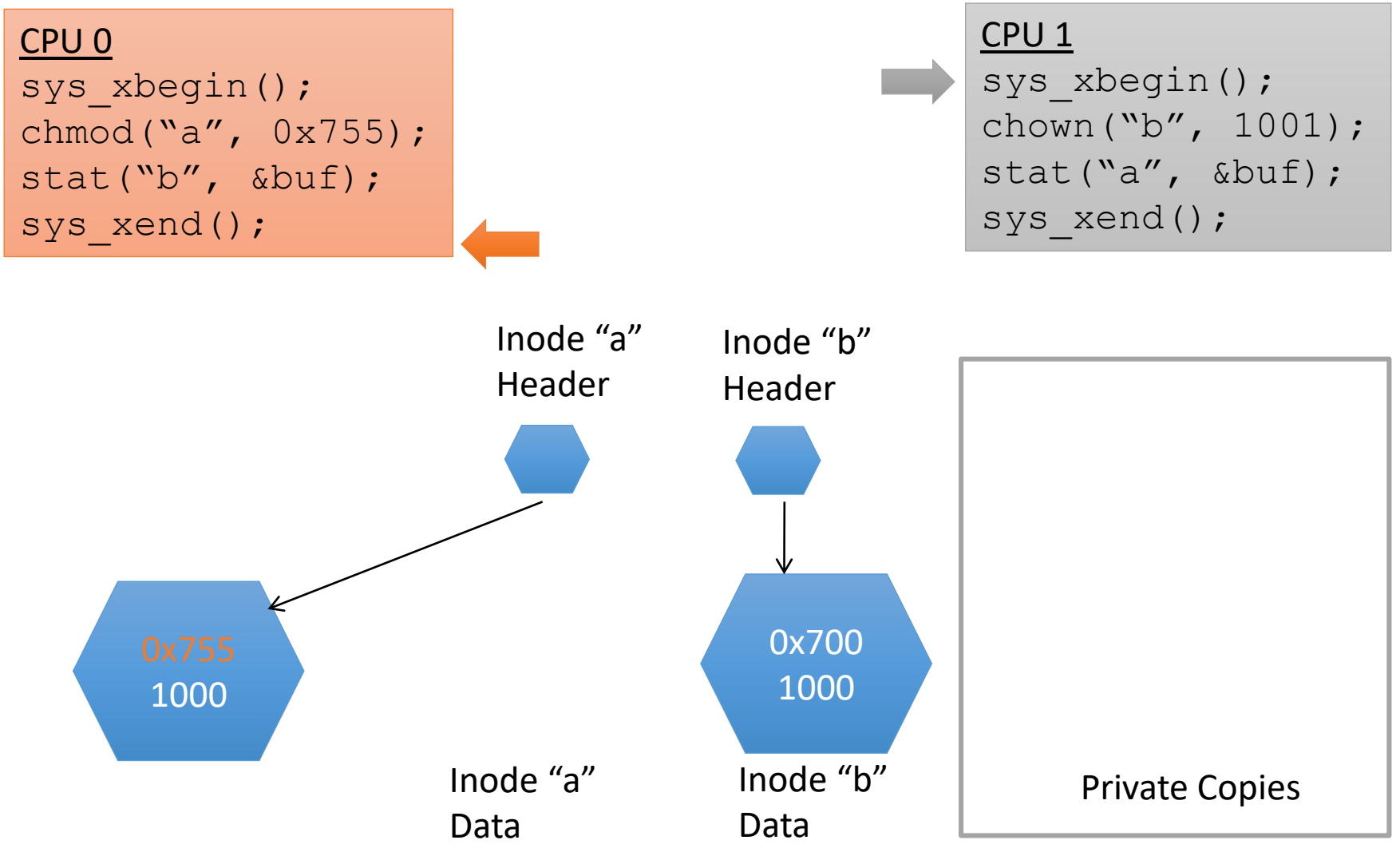
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a"
Header
Inode "b"
Header
0x755
1000
0x775
1000
0x700
1000
Inode "a"
Data
Inode "b"
Data
Private Copies

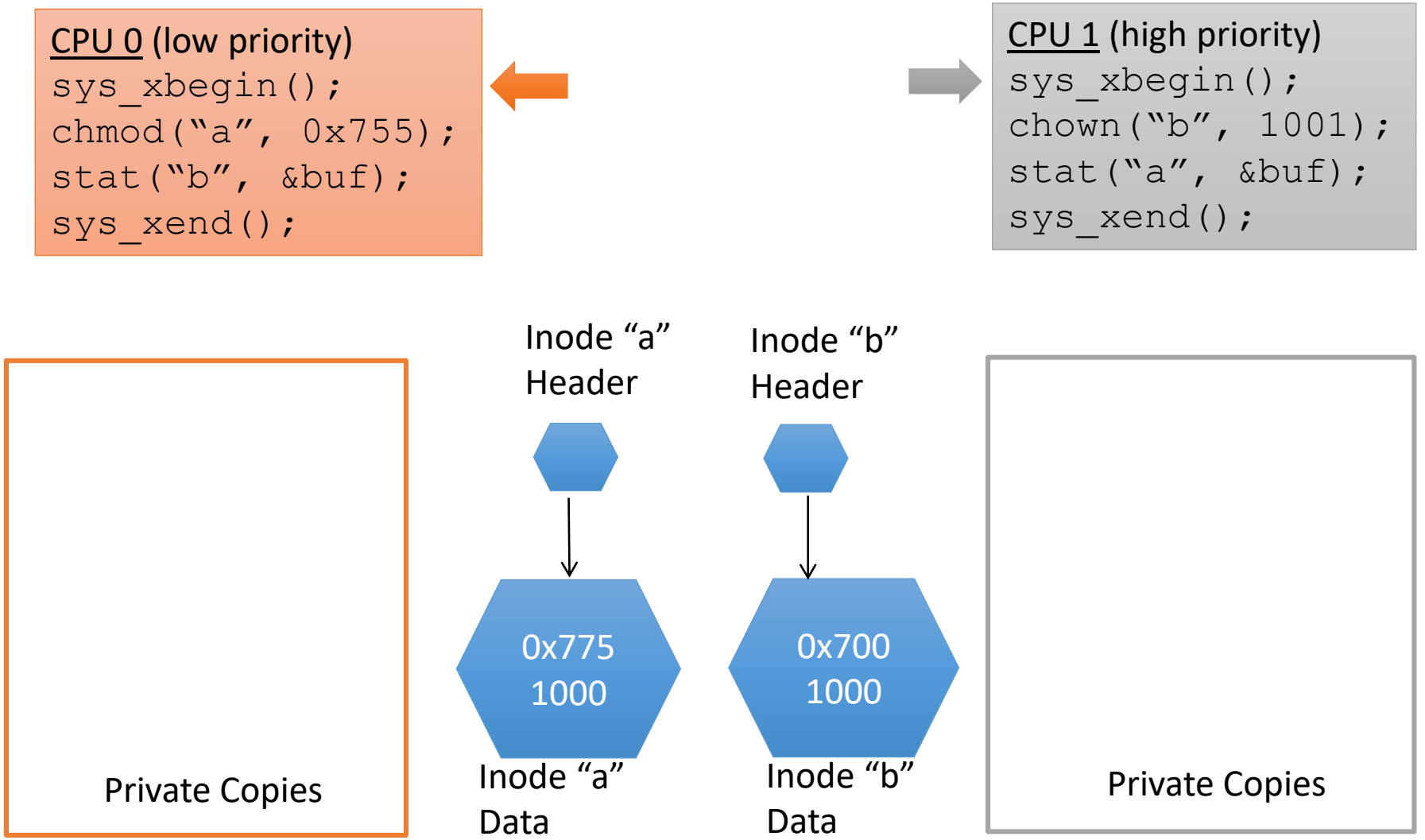
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
0x775
1000
0x700
1000
Inode "a" Data
Inode "b" Data
Private Copies

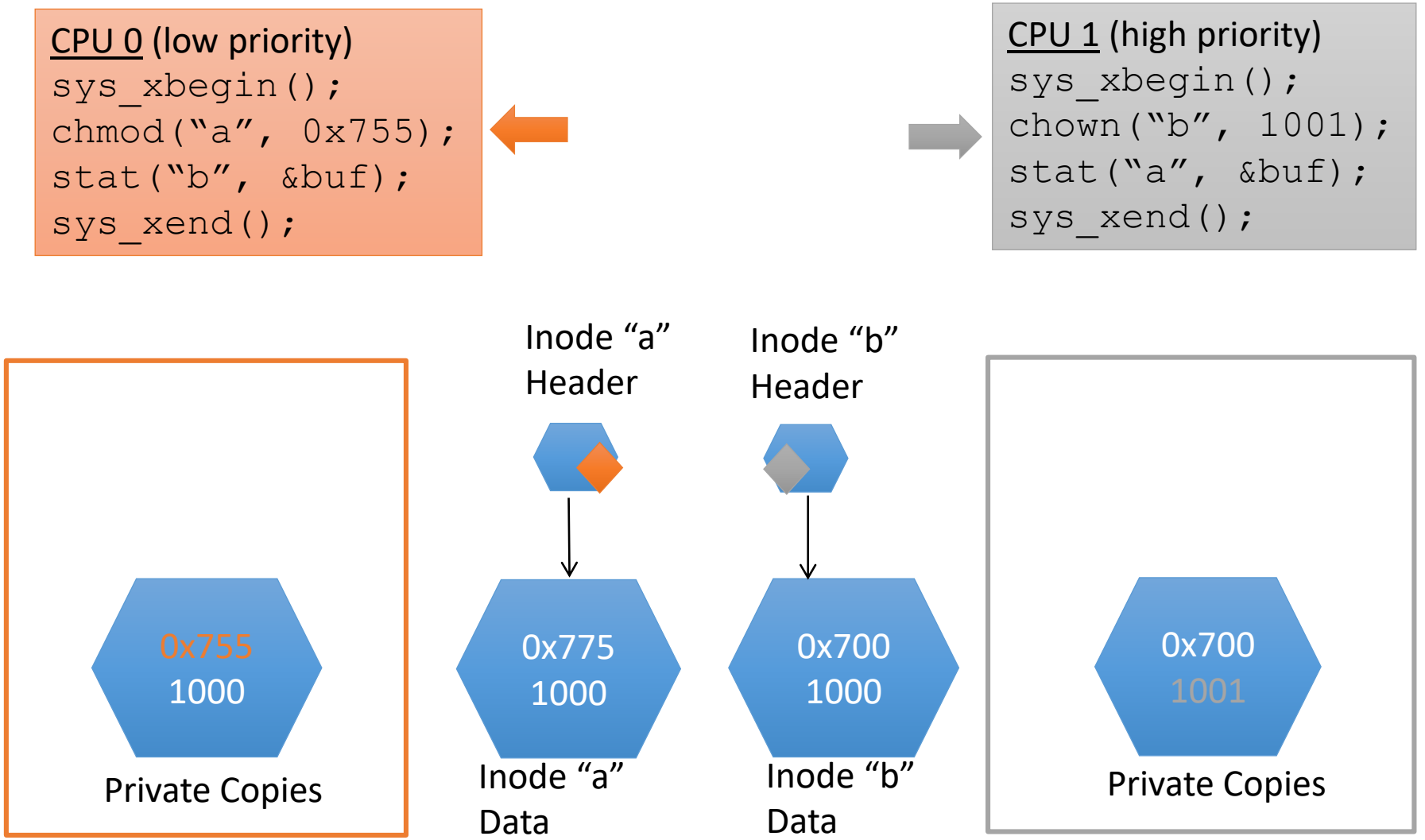
TxOS design


CPU 0
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
0x700
1000
Inode "a" Data
Inode "b" Data
Private Copies

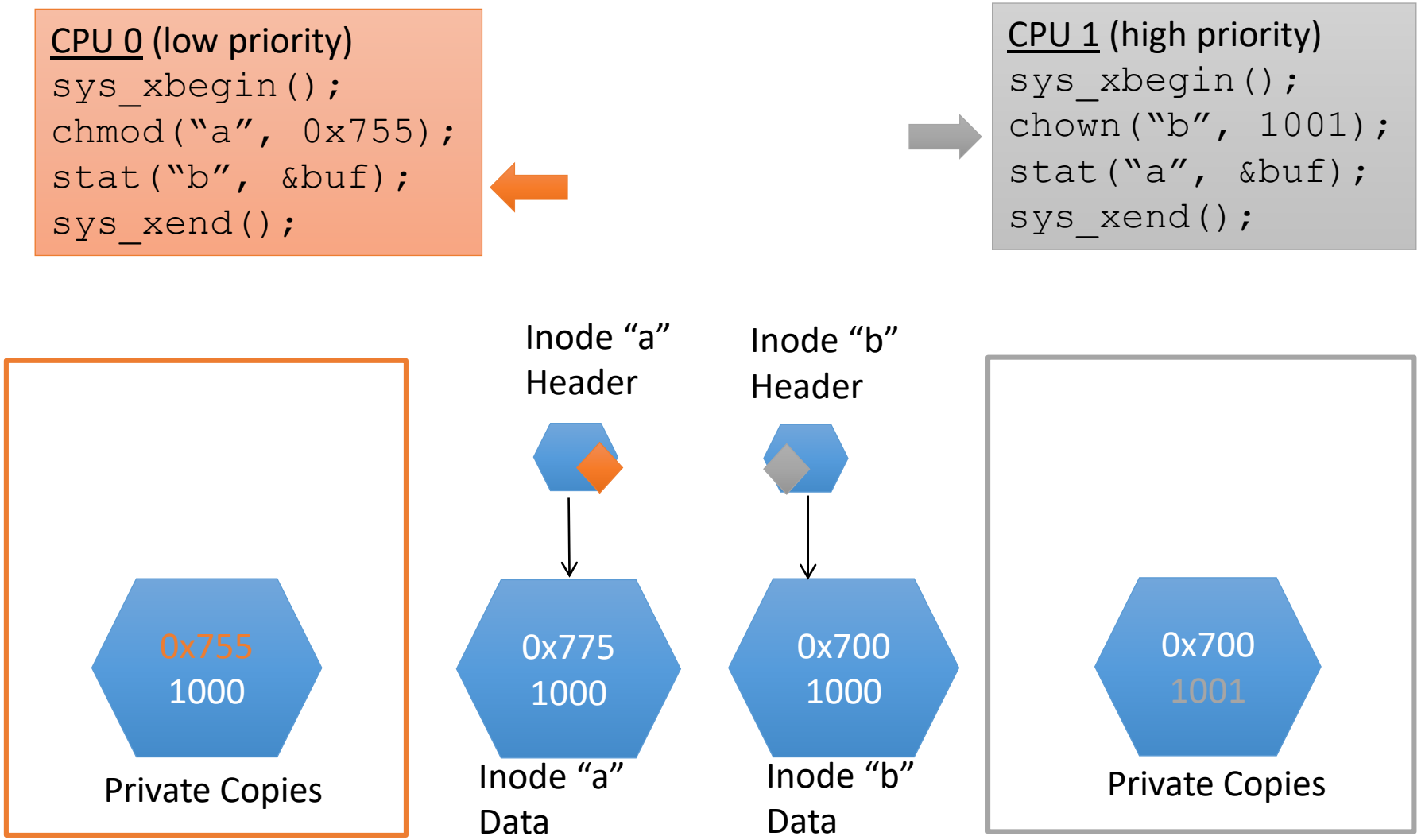
TxOS design


CPU 0 (low priority)
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x775 1000
0x700 1000
Inode "a" Data
Inode "b" Data
Private Copies
Private Copies

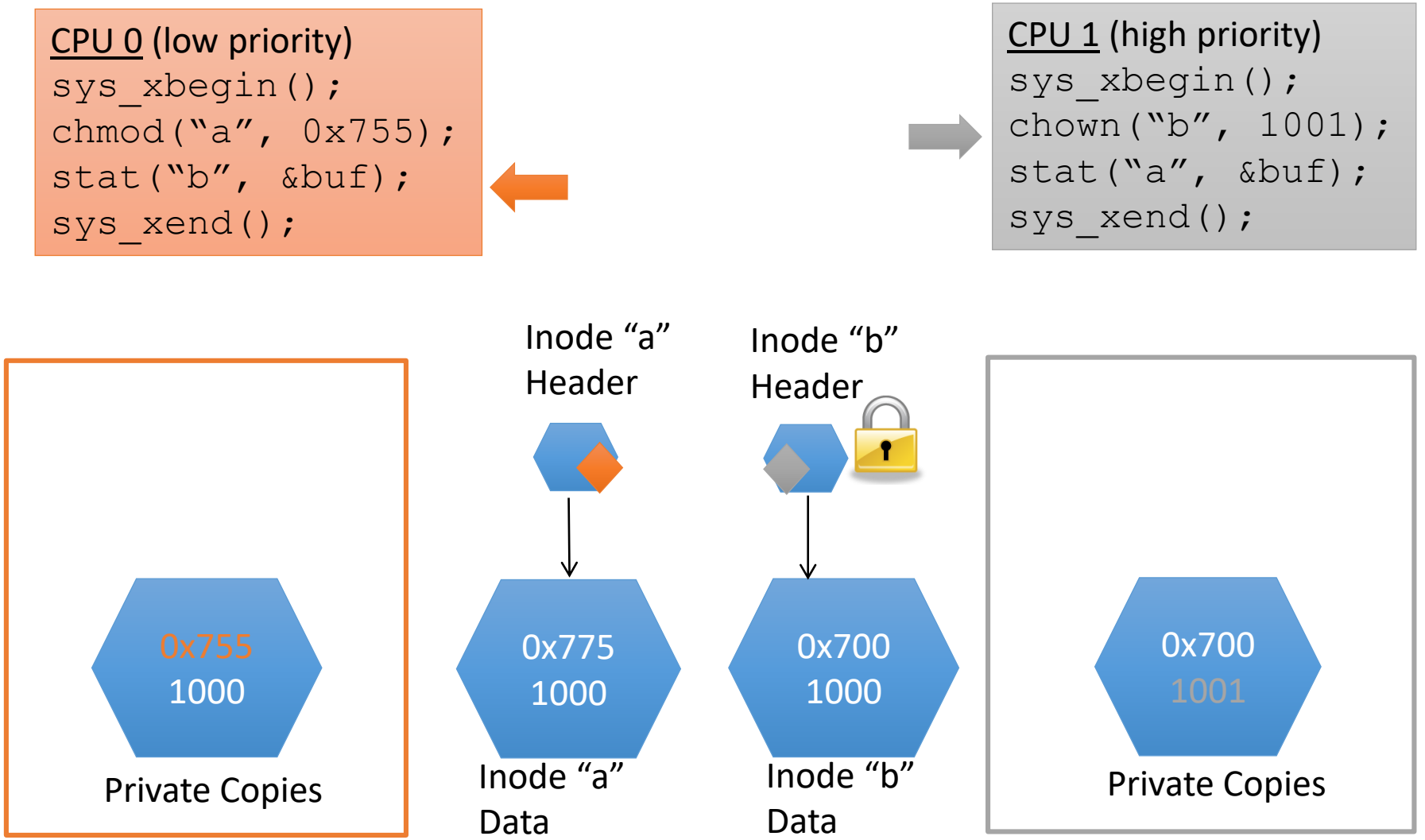
TxOS design


CPU 0 (low priority)
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
0x700
1001
Private Copies

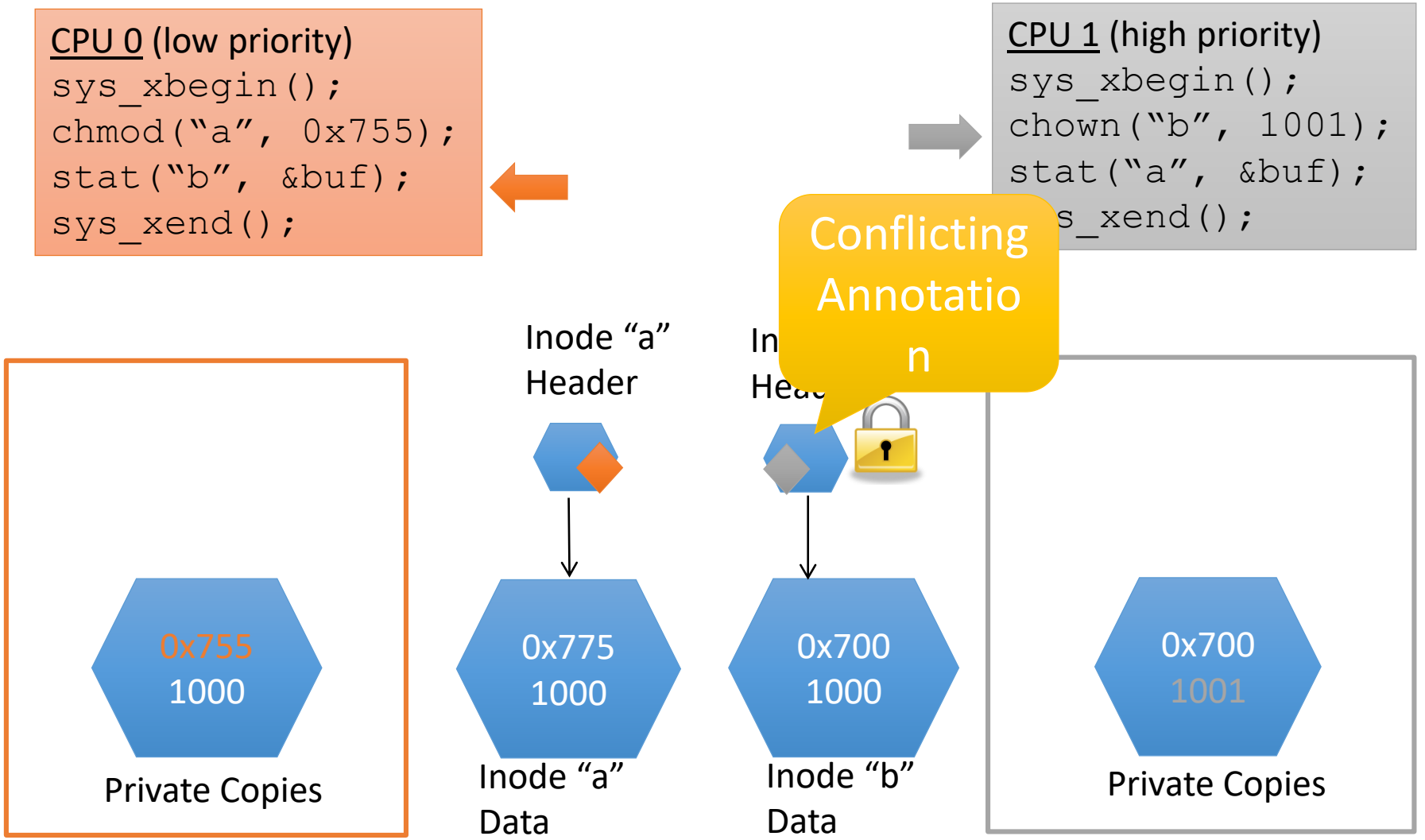
TxOS design


CPU 0 (low priority)
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
0x700
1001
Private Copies

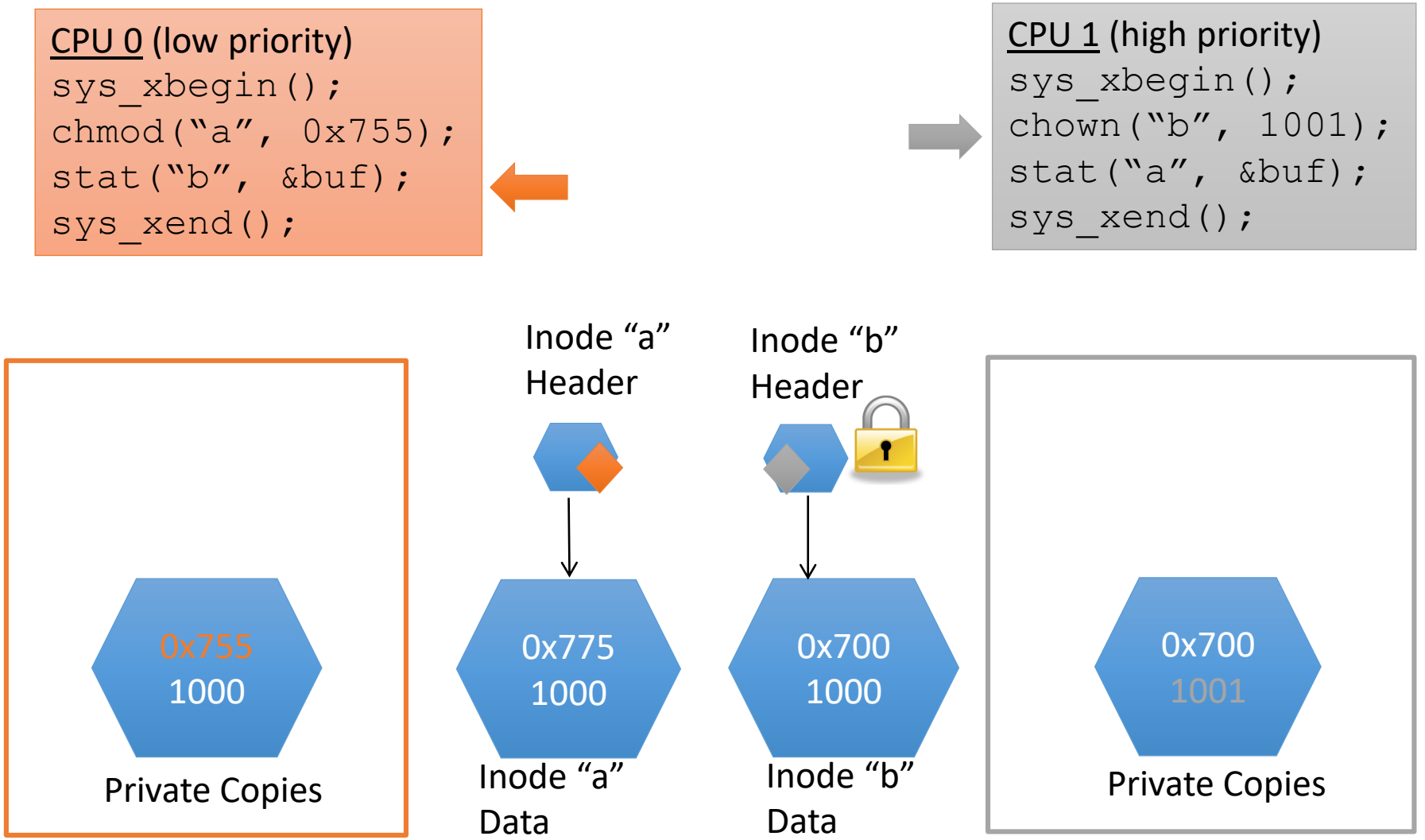
TxOS design


CPU 0 (low priority)
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
0x700
1001
Private Copies

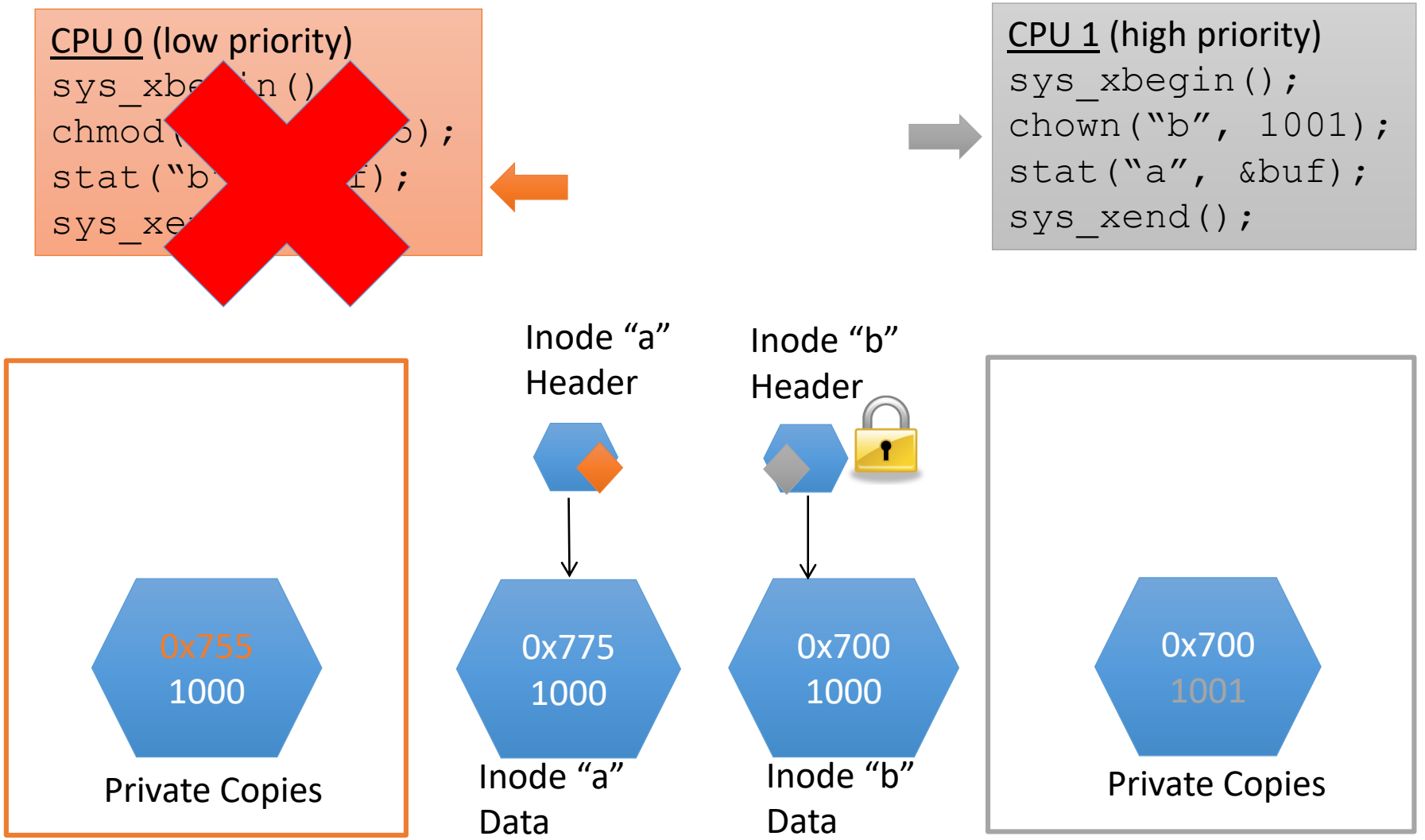
TxOS design


CPU 0 (low priority)
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
s_xend();
Conflicting Annotation
Inode "a" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
0x700
1001
Private Copies

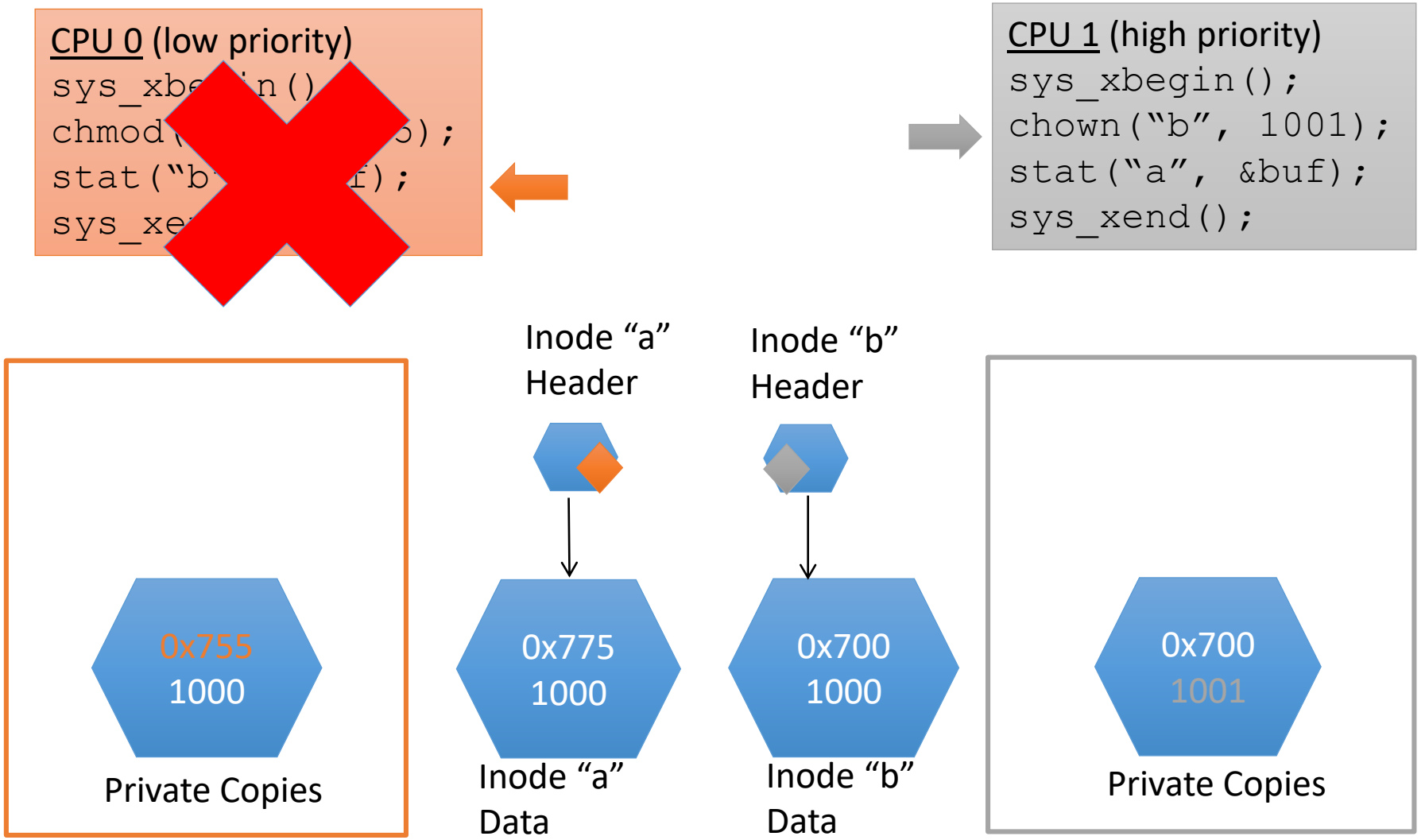
TxOS design


CPU 0 (low priority)
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
0x700
1001
Private Copies

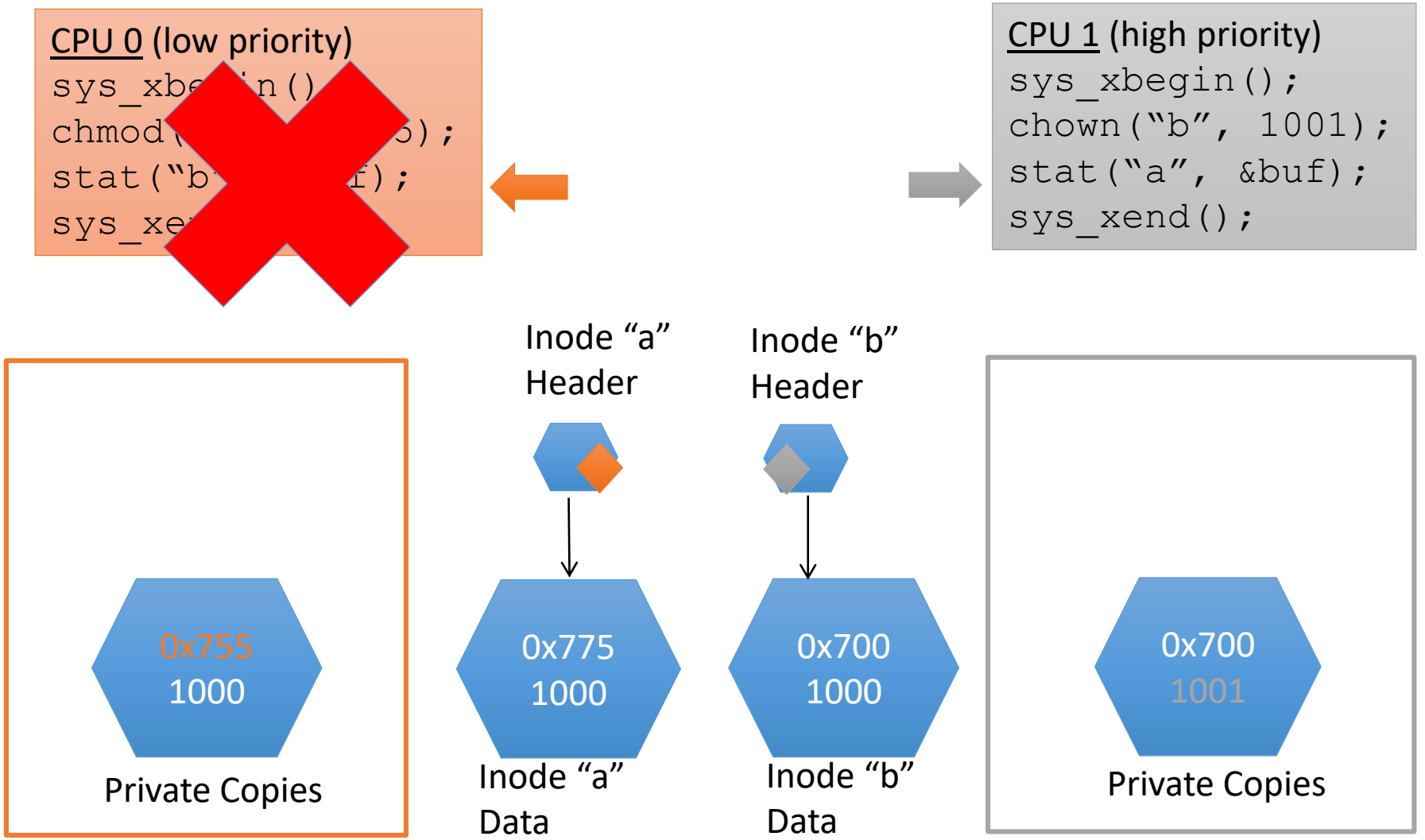
TxOS design


CPU 0 (low priority)
sys_xbe n()
chmod(5);
stat("b f);
sys_xe
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a"
Header
Inode "b"
Header
0x755
1000
Private Copies
0x775
1000
Inode "a"
Data
0x700
1000
Inode "b"
Data
0x700
1001
Private Copies

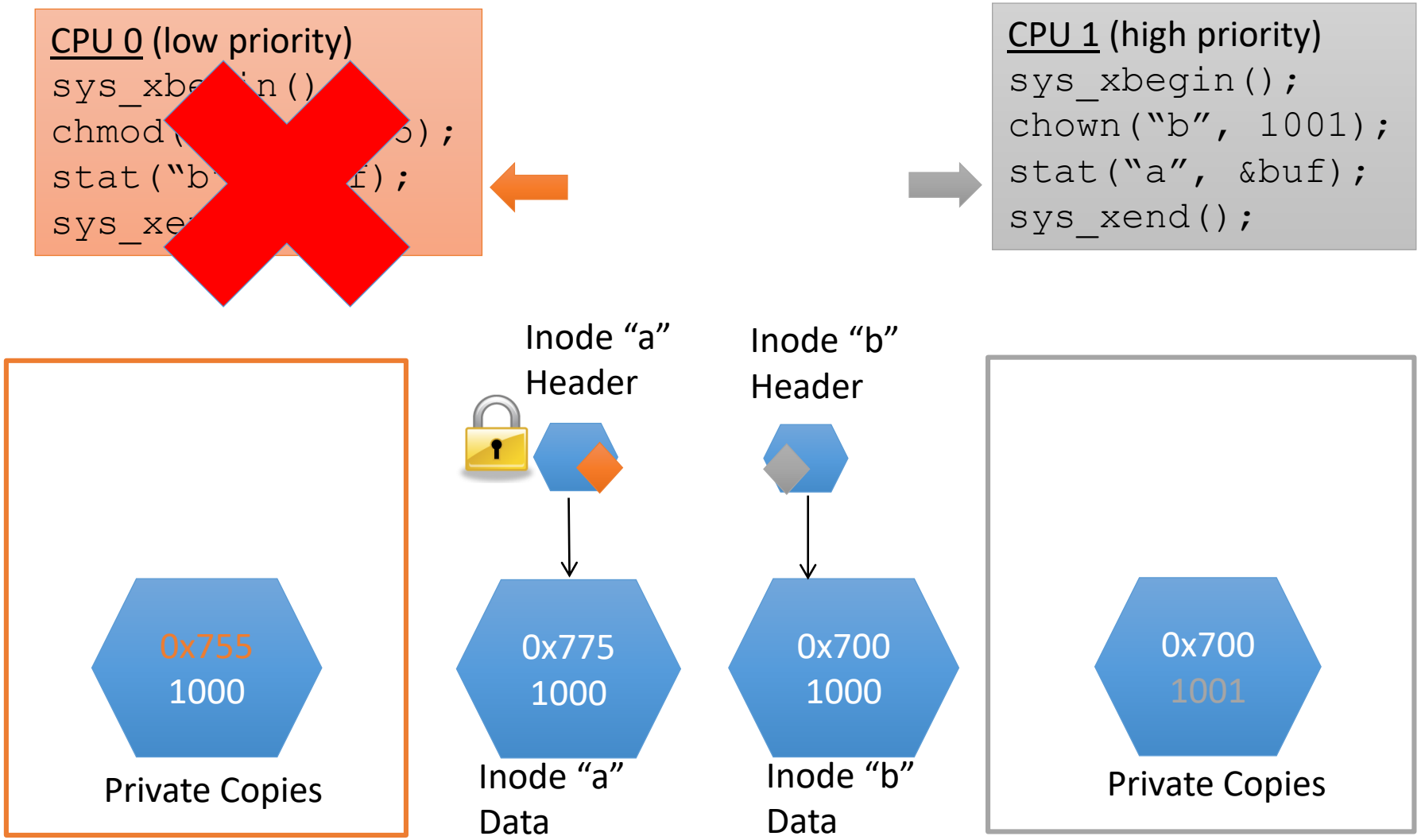
TxOS design


CPU 0 (low priority)
sys_xbe n()
chmod(5);
stat("b f);
sys_xe
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
0x700
1001
Private Copies

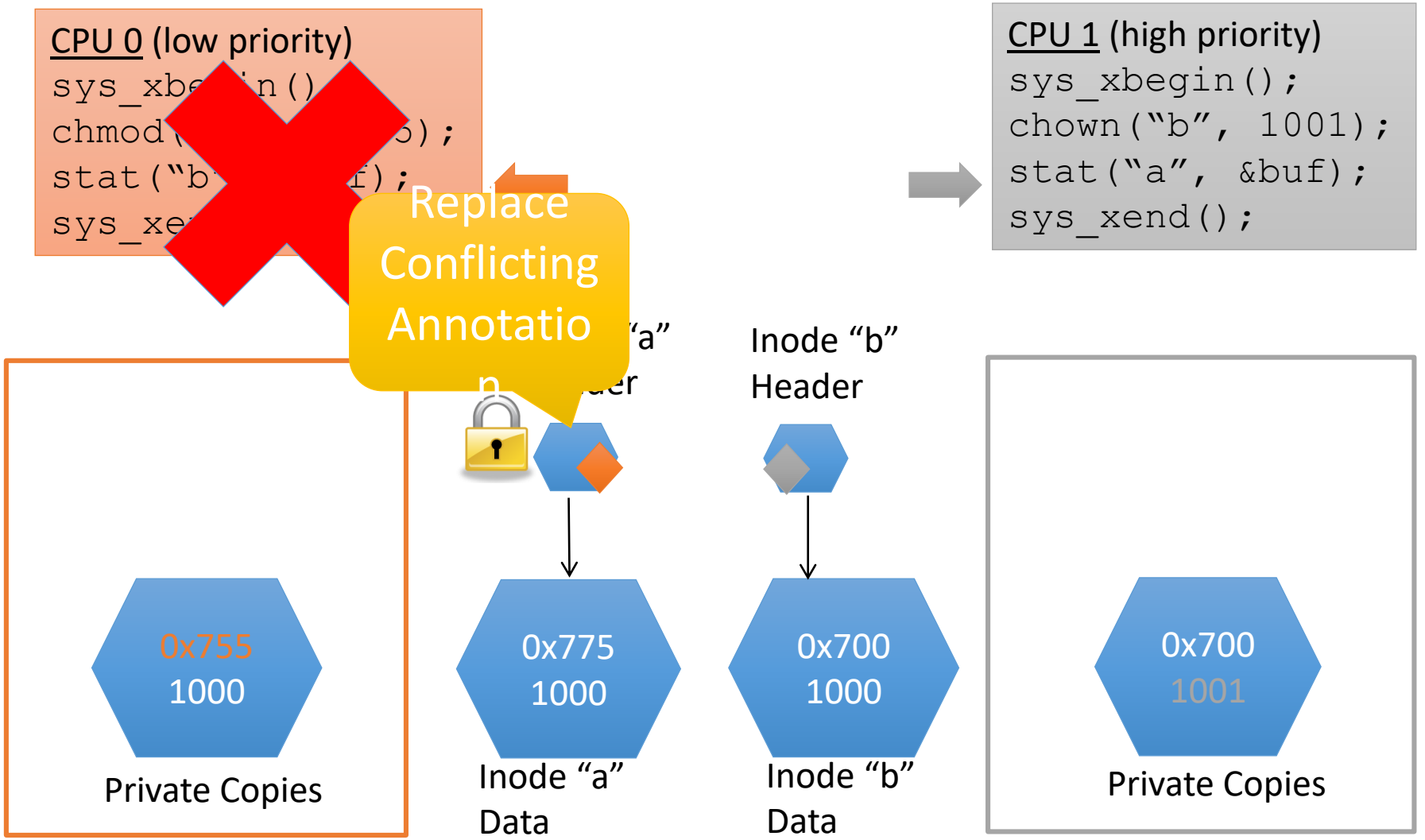
TxOS design


CPU 0 (low priority)
sys_xbe n()
chmod(5);
stat("b f);
sys_xe
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
0x700
1001
Private Copies

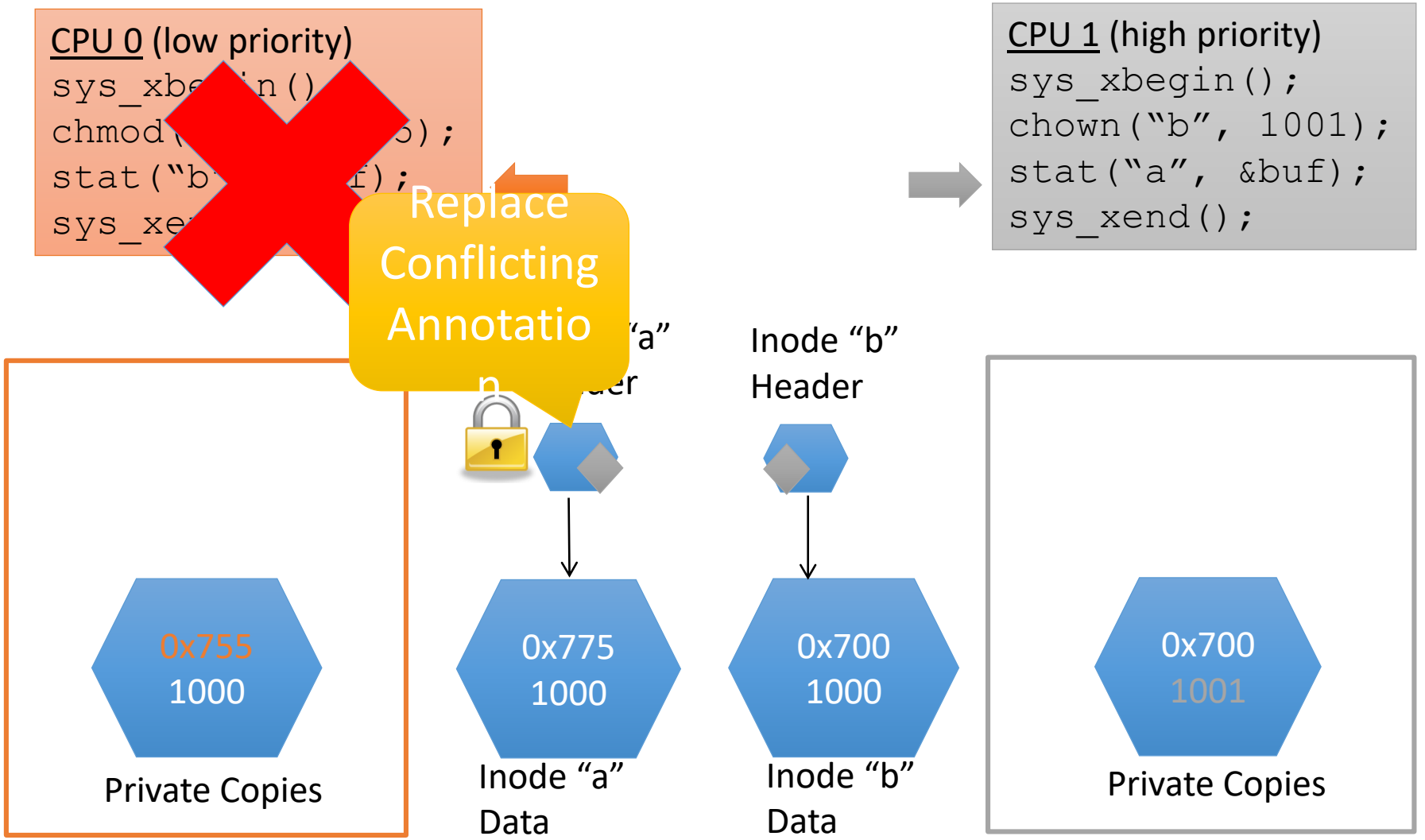
TxOS design


CPU 0 (low priority)
sys_xbe n()
chmod(5);
stat("b f);
sys_xe
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a"
Header
Inode "b"
Header
0x755
1000
Private Copies
0x775
1000
Inode "a"
Data
0x700
1000
Inode "b"
Data
0x700
1001
Private Copies

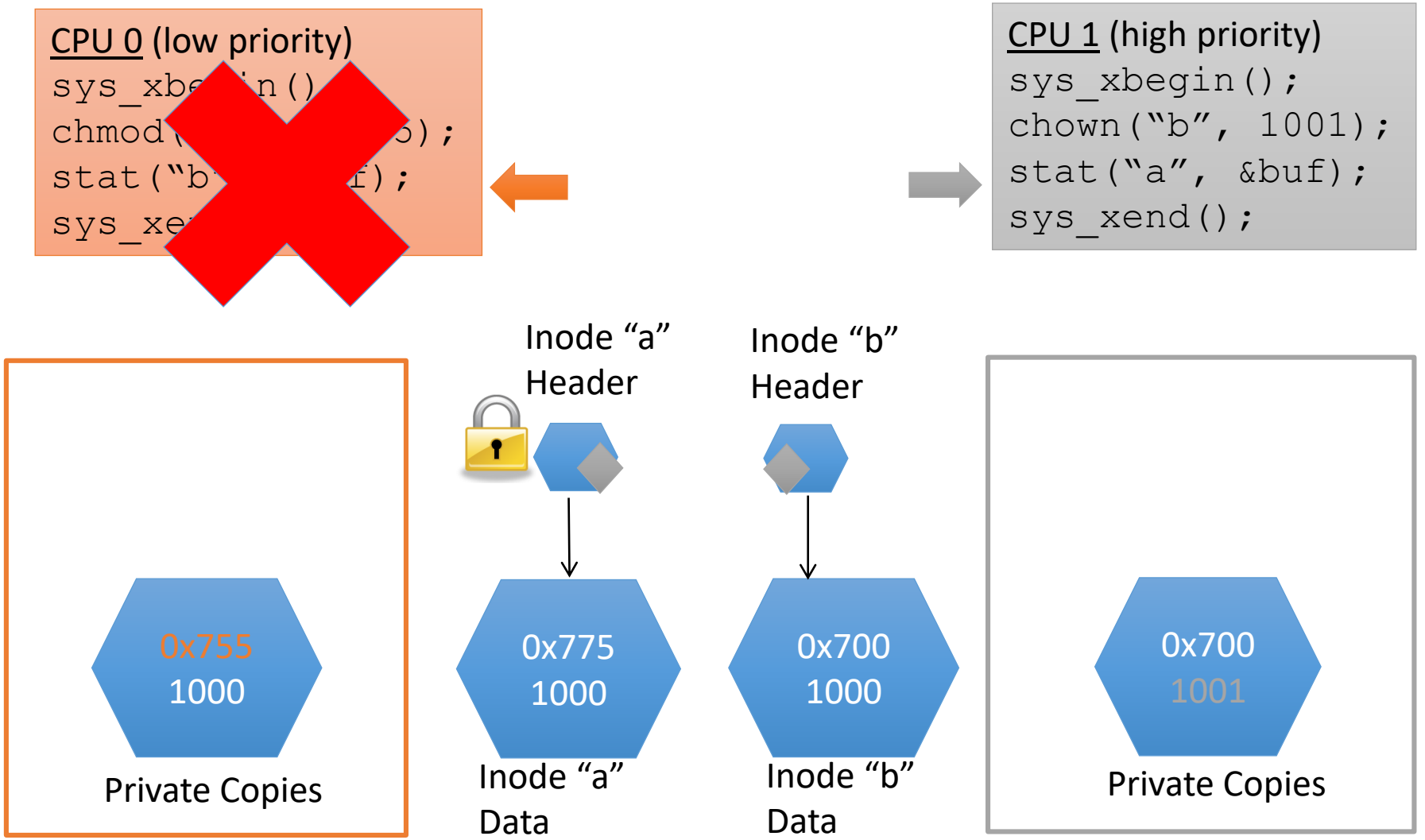
TxOS design


CPU 0 (low priority)
sys_xbe n()
chmod(5);
stat("b f);
sys_xe
Replace Conflicting Annotatio
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "b" Header
0x755 1000
Private Copies
0x775 1000
Inode "a" Data
0x700 1000
Inode "b" Data
0x700 1001
Private Copies

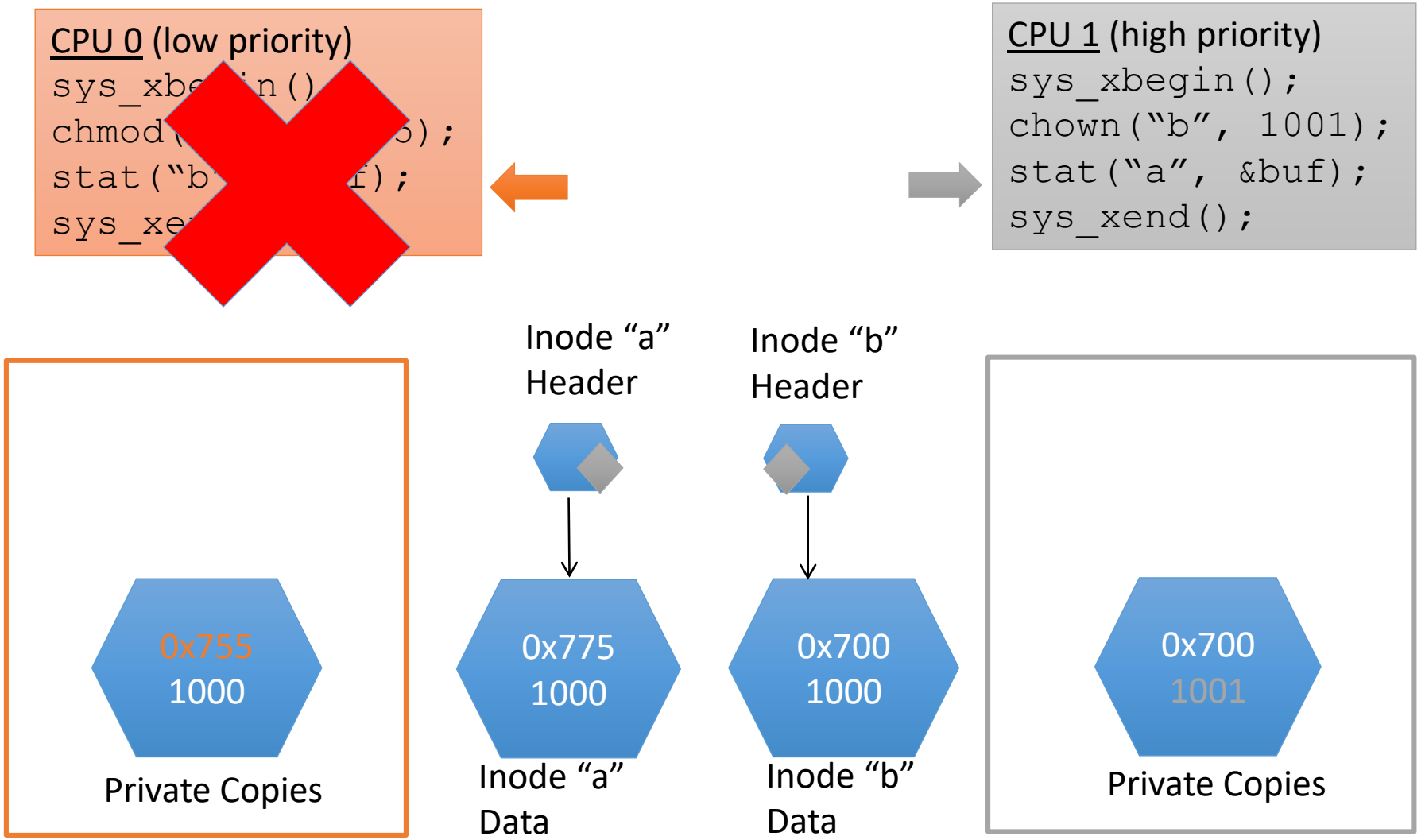
TxOS design


CPU 0 (low priority)
sys_xbe n()
chmod(5);
stat("b f);
sys_xe
Replace Conflicting Annotatio
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
0x700
1001
Private Copies

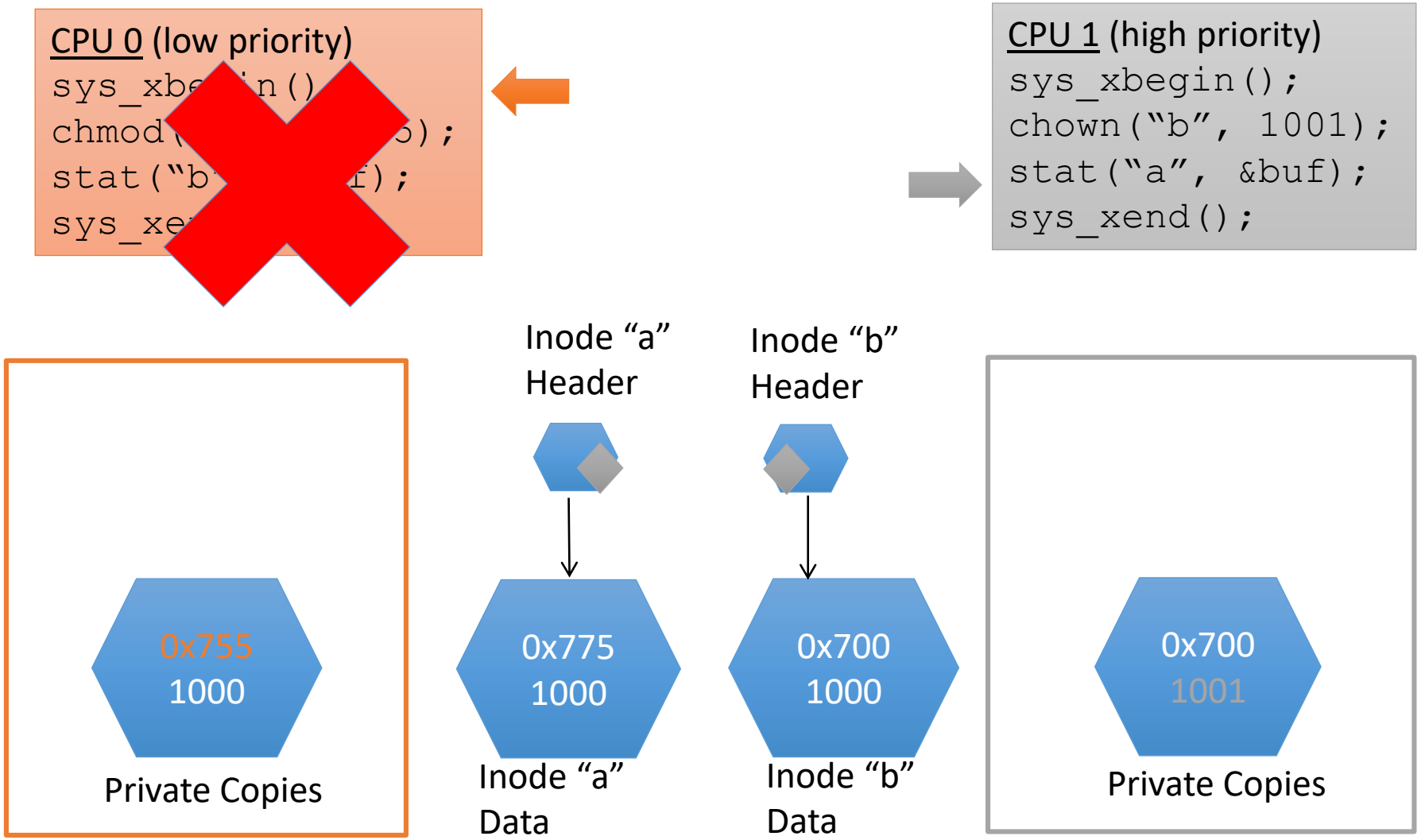
TxOS design


CPU 0 (low priority)
sys_xbe n()
chmod(5);
stat("b f);
sys_xe
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
0x700
1001
Private Copies

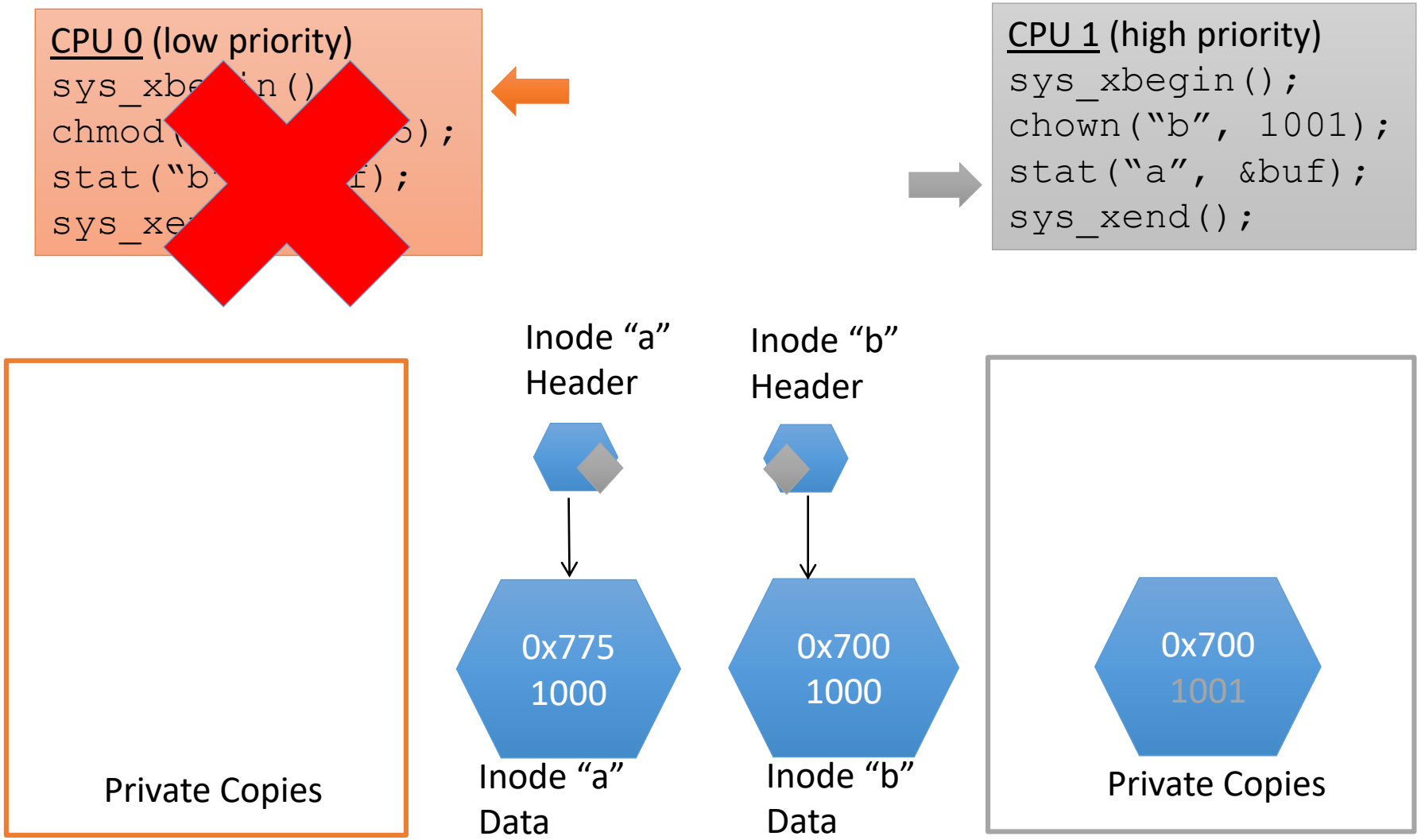
TxOS design


CPU 0 (low priority)
sys_xbe n()
chmod(5);
stat("b f);
sys_xe
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
0x700
1001
Private Copies

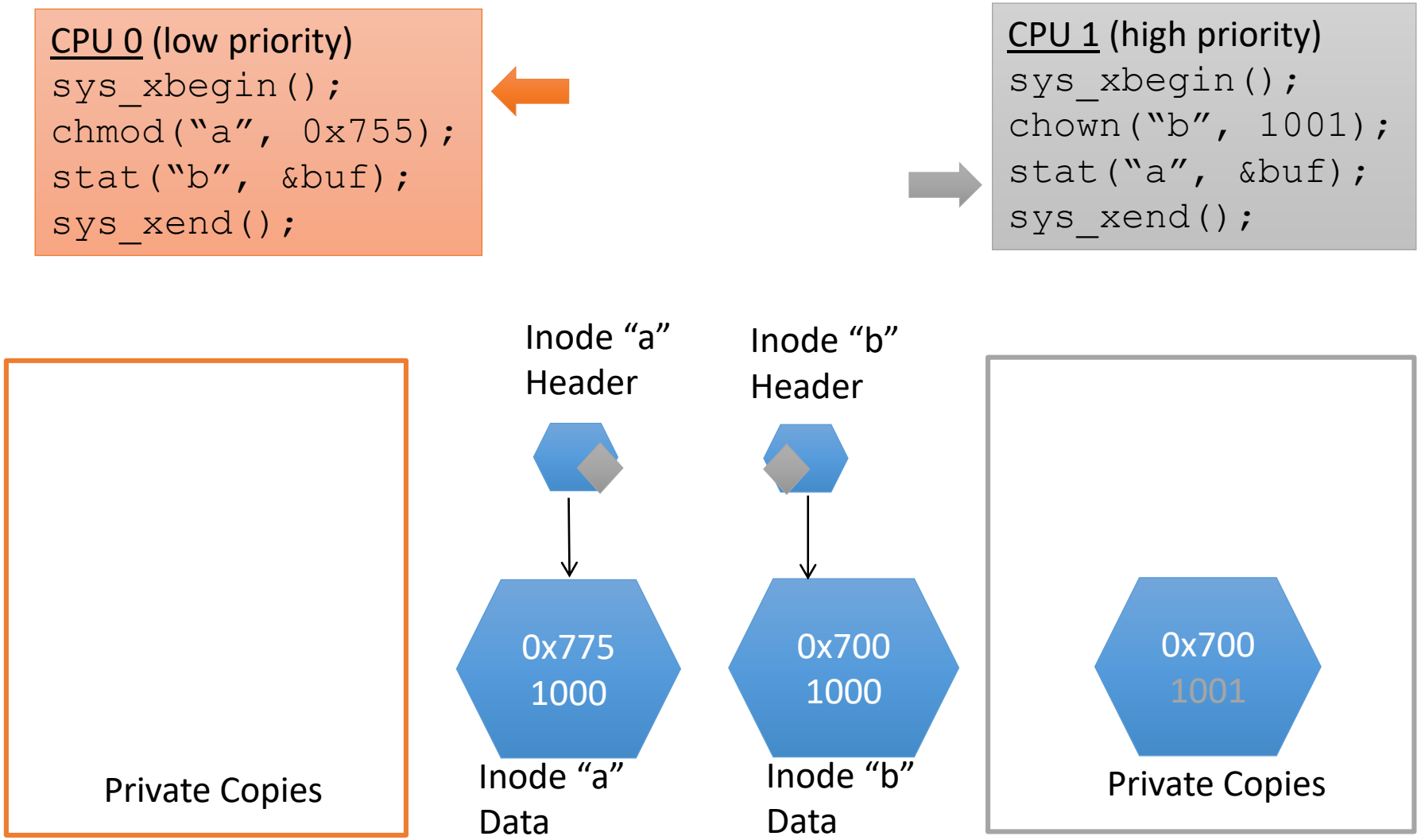
TxOS design


CPU 0 (low priority)
sys_xbe n()
chmod(5);
stat("b f);
sys_xe
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x755
1000
Private Copies
0x775
1000
Inode "a" Data
0x700
1000
Inode "b" Data
0x700
1001
Private Copies

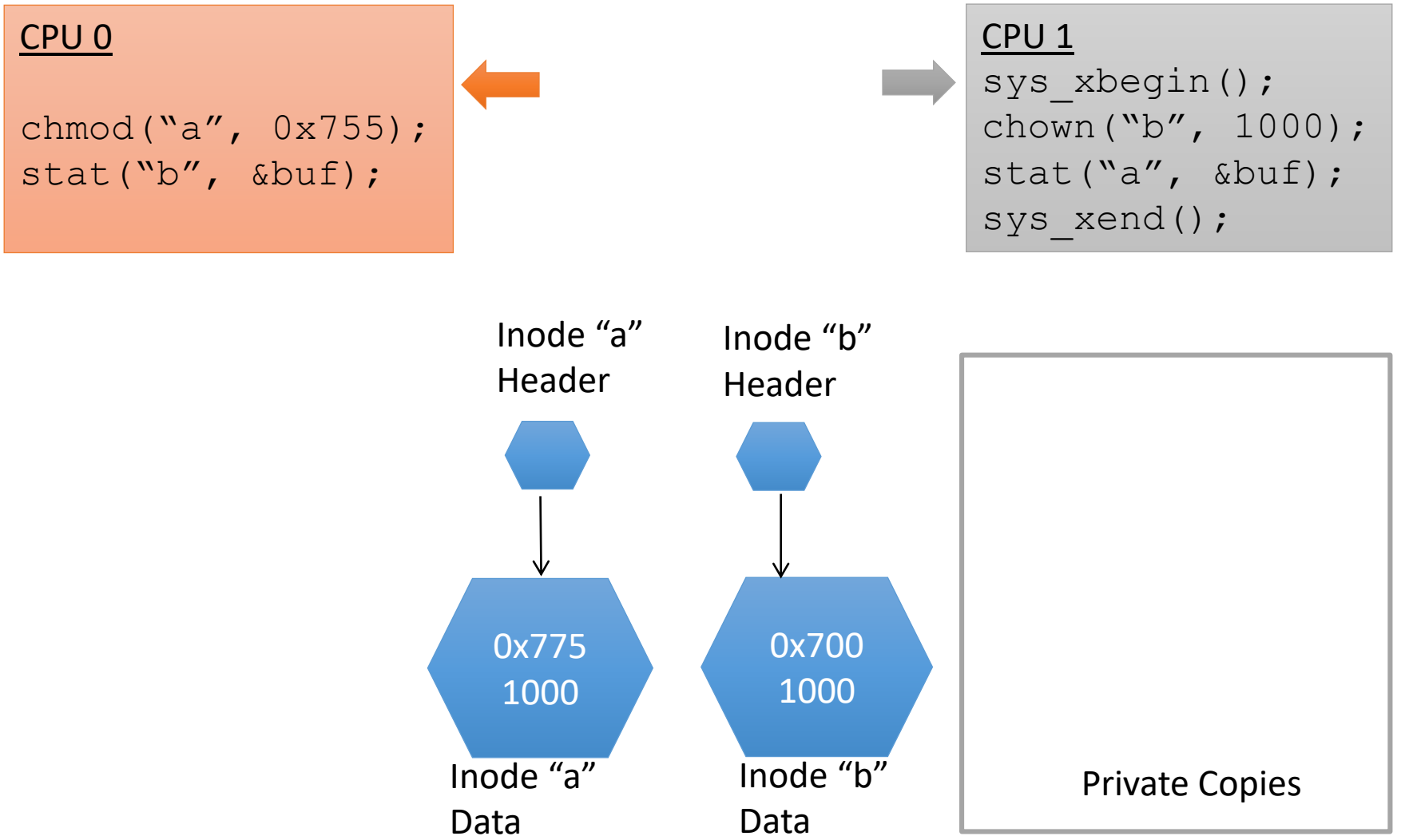
TxOS design


CPU 0 (low priority)
sys_xbe n()
chmod(5);
stat("b f);
sys_xe
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x775 1000
0x700 1000
0x700 1001
Inode "a" Data
Inode "b" Data
Private Copies
Private Copies

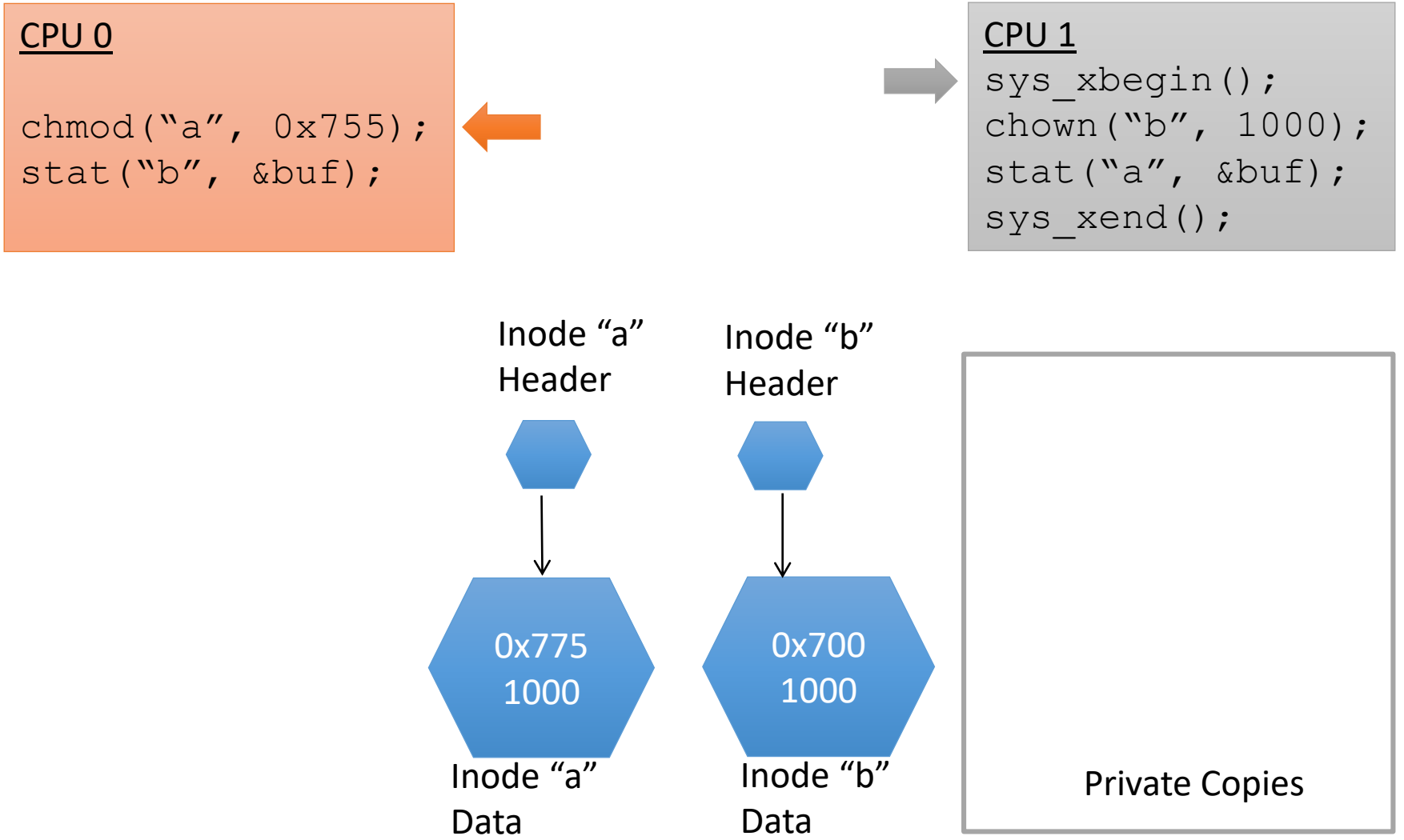
TxOS design


CPU 0 (low priority)
sys_xbegin();
chmod("a", 0x755);
stat("b", &buf);
sys_xend();
CPU 1 (high priority)
sys_xbegin();
chown("b", 1001);
stat("a", &buf);
sys_xend();
Inode "a" Header
Inode "b" Header
0x775 1000
0x700 1000
0x700 1001
Private Copies
Inode "a" Data
Inode "b" Data
Private Copies

Strong isolation in TxOS



Strong isolation in TxOS



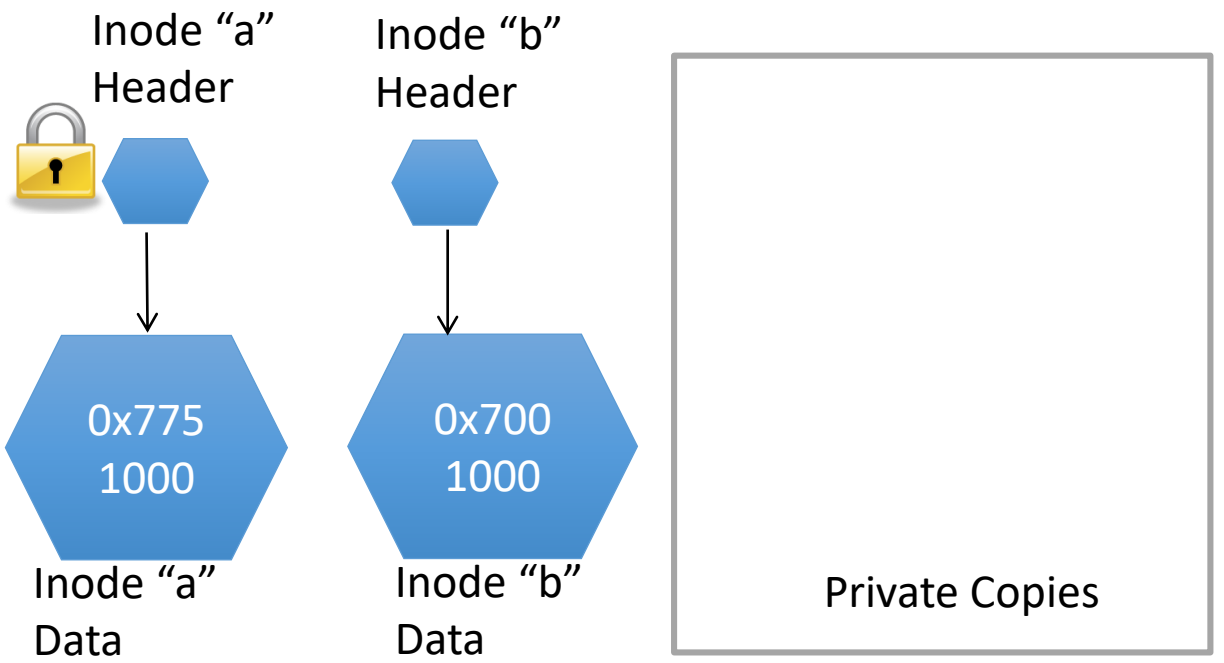
Strong isolation in TxOS

CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```



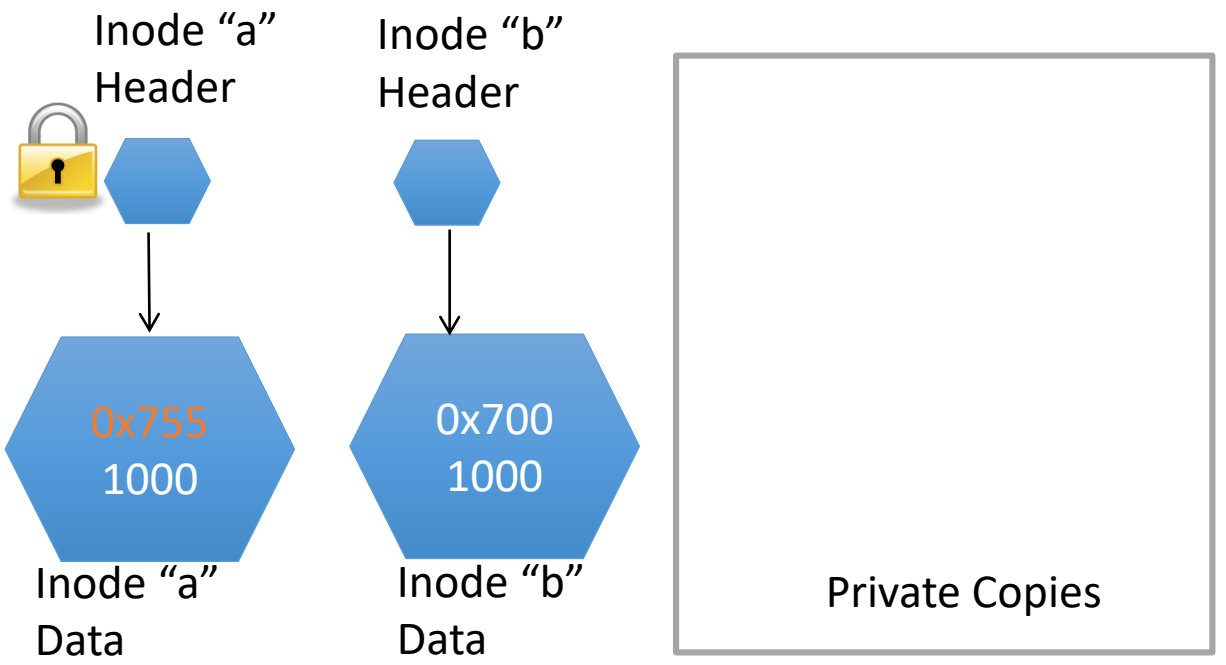
Strong isolation in TxOS

CPU 0

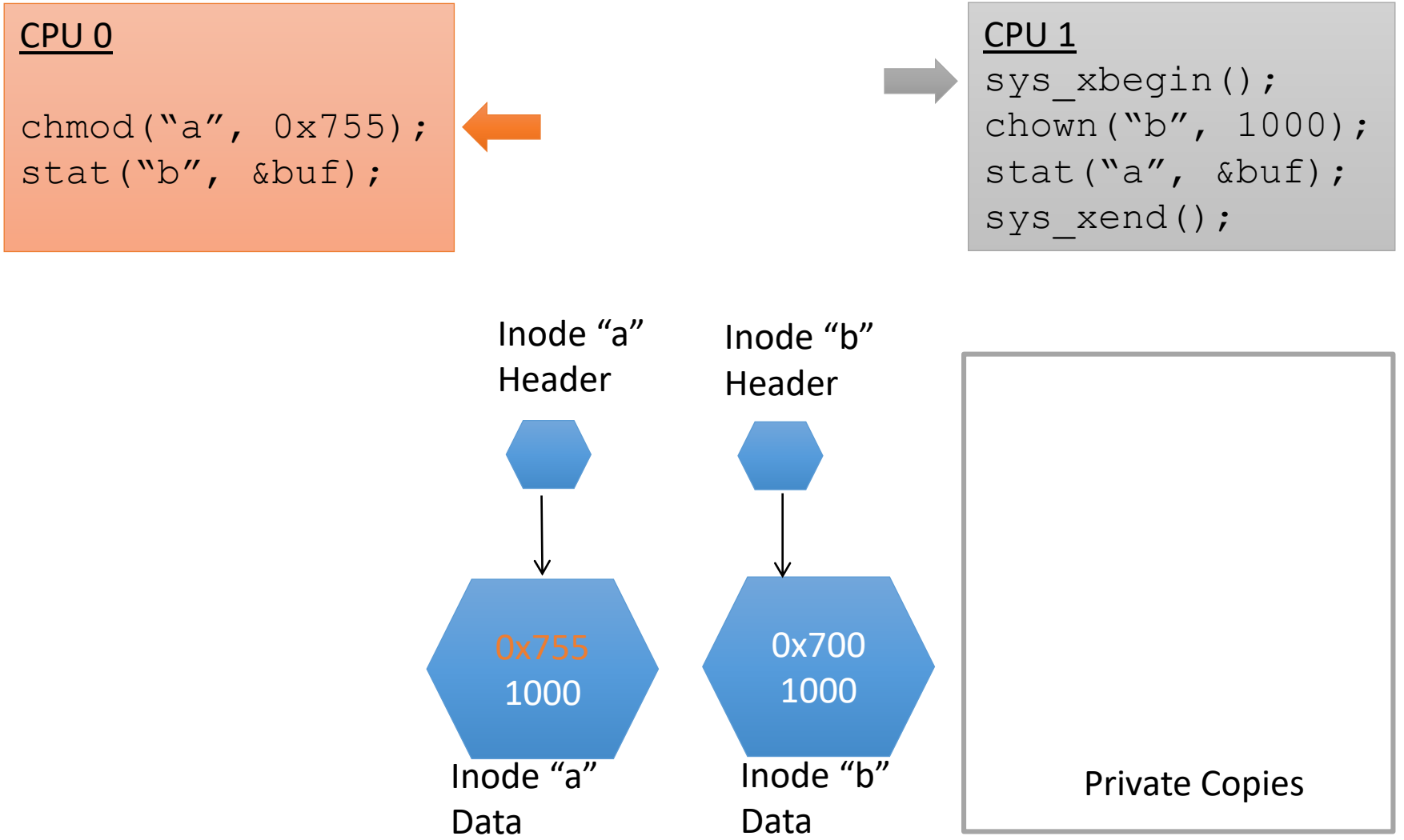
```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

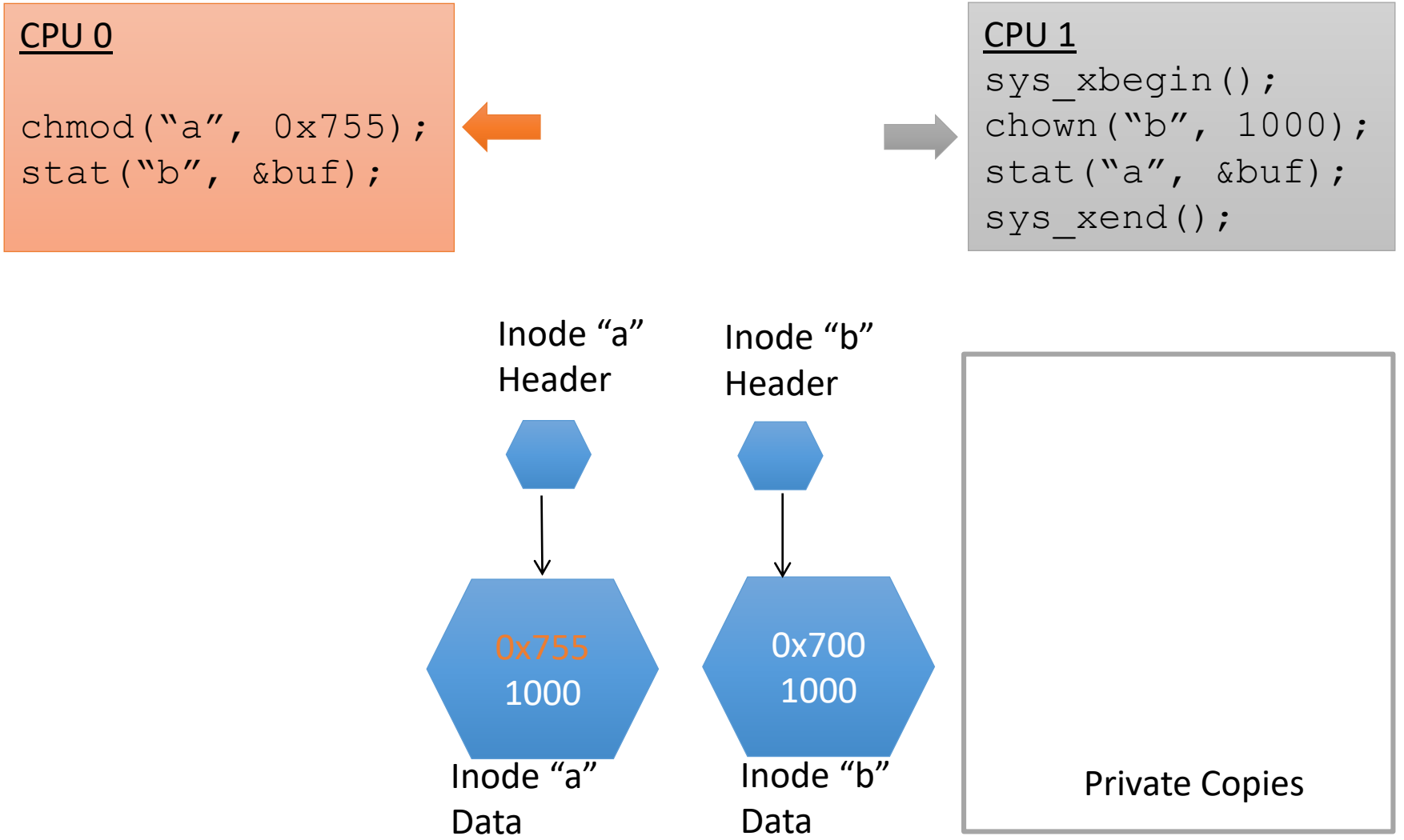
```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```



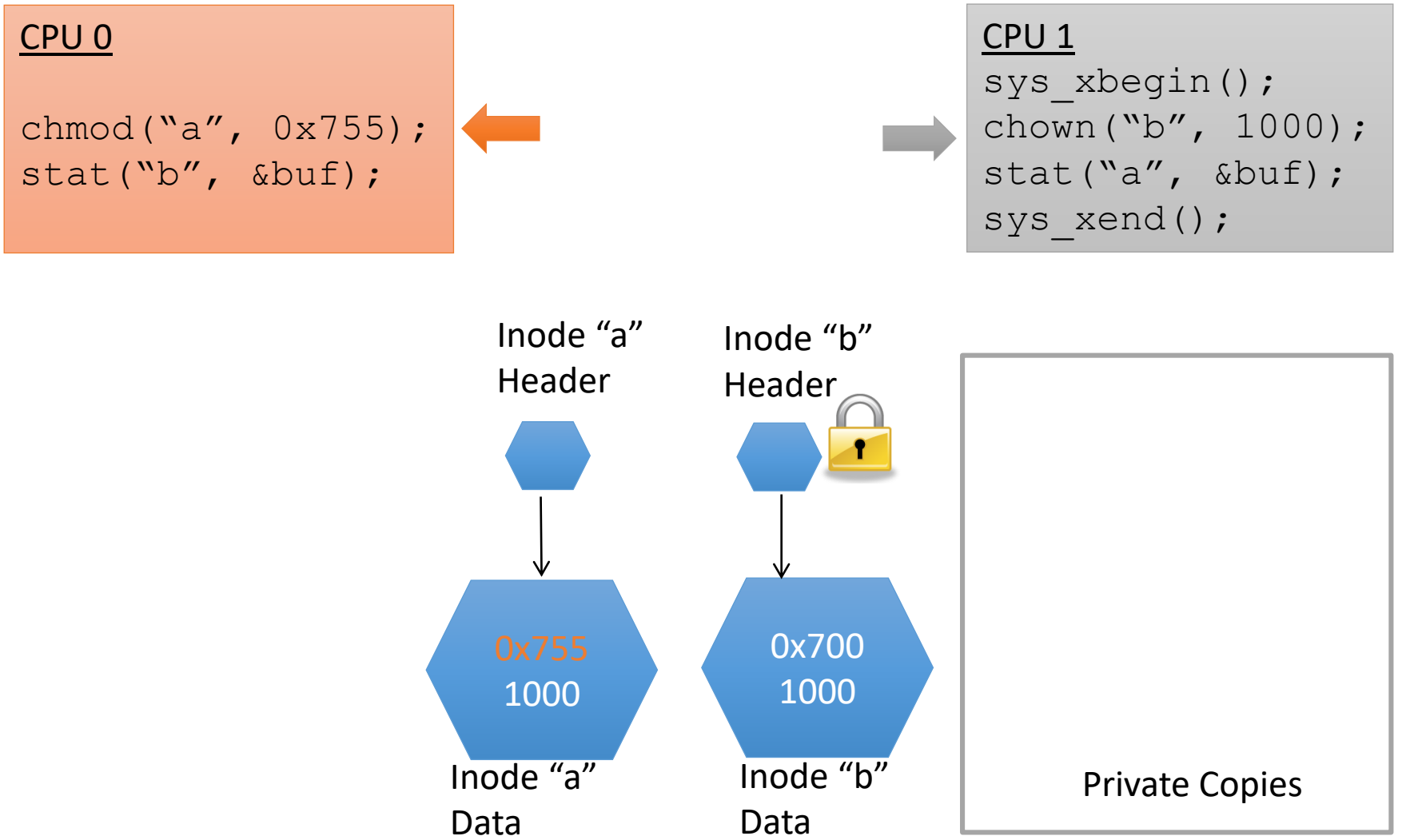
Strong isolation in TxOS



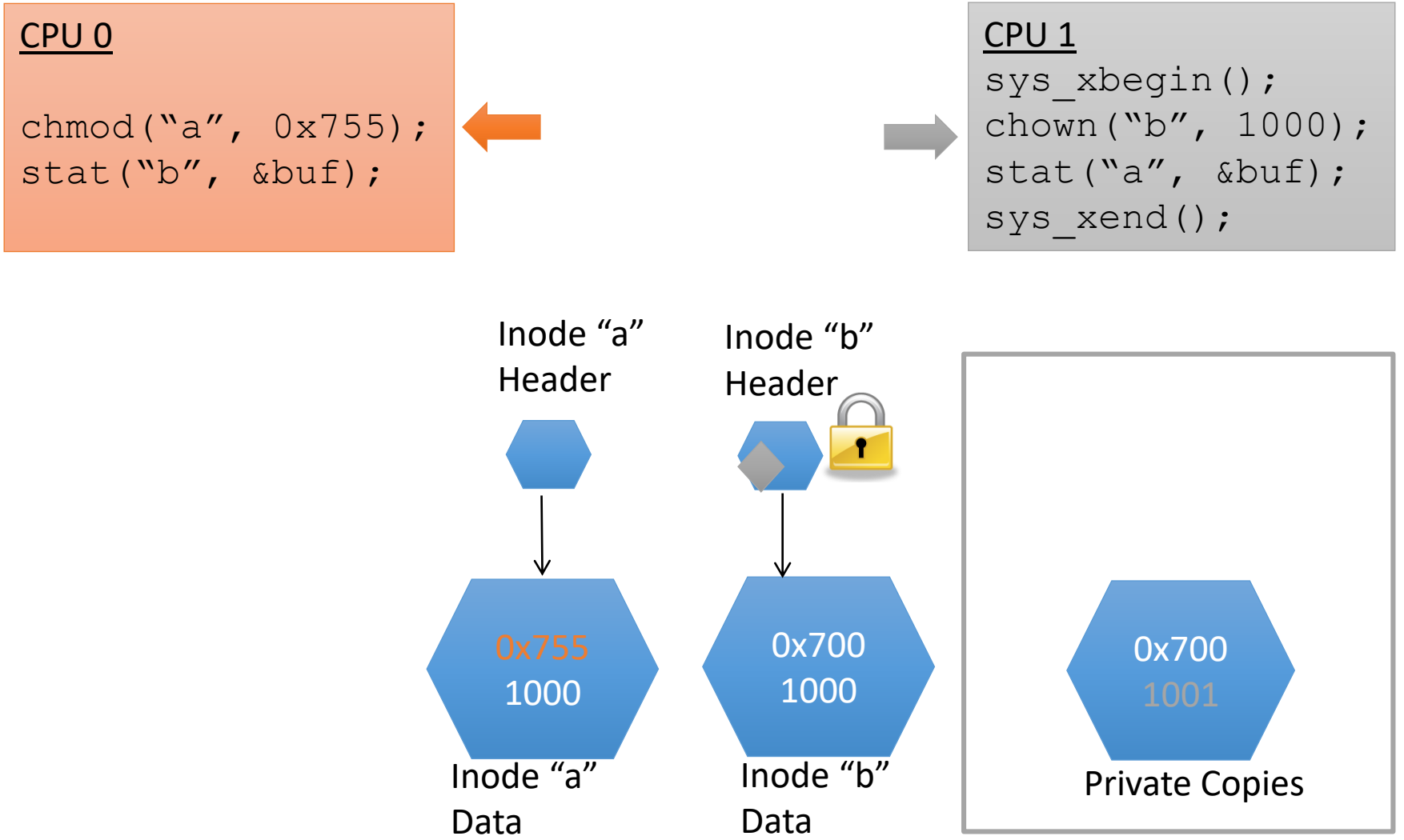
Strong isolation in TxOS



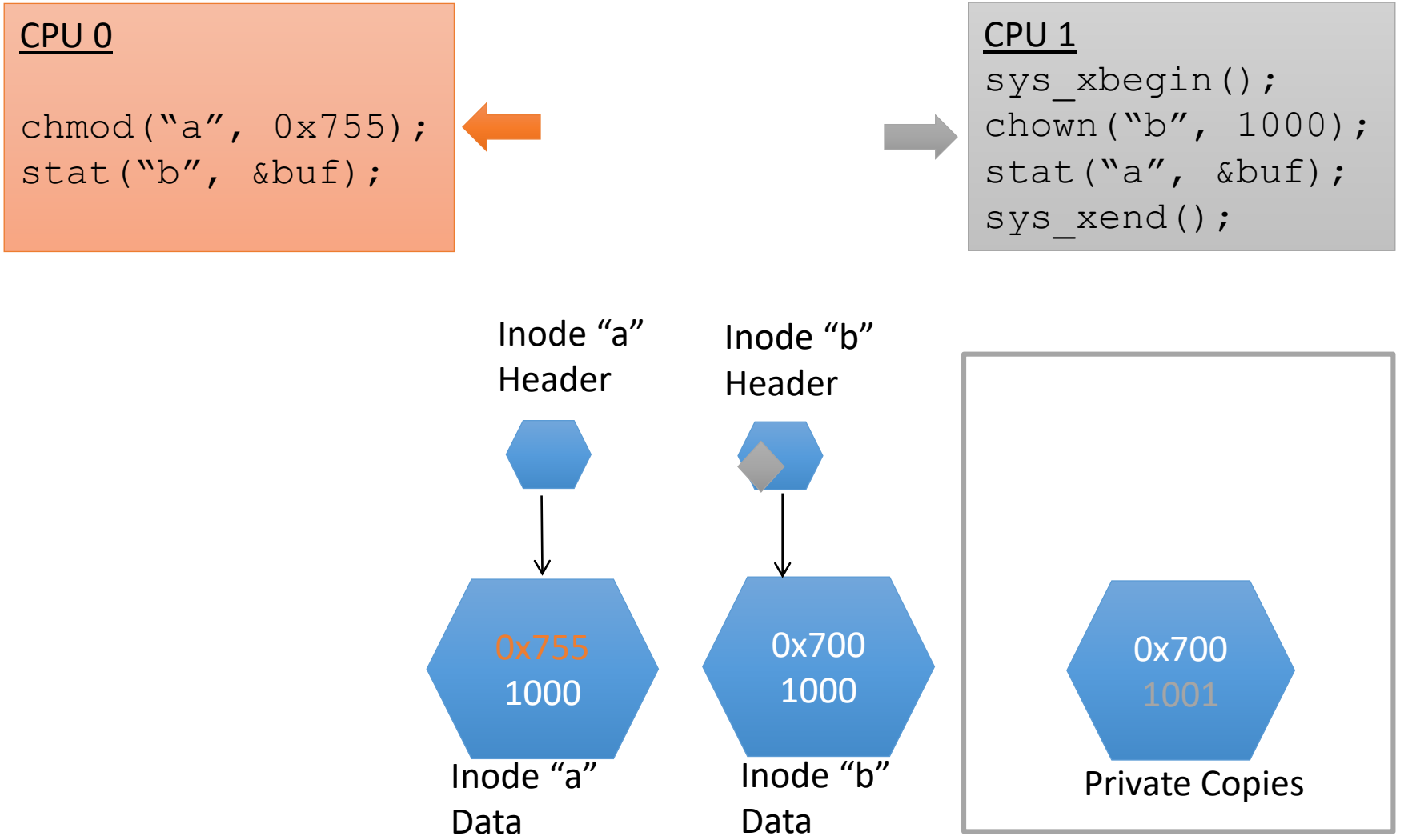
Strong isolation in TxOS



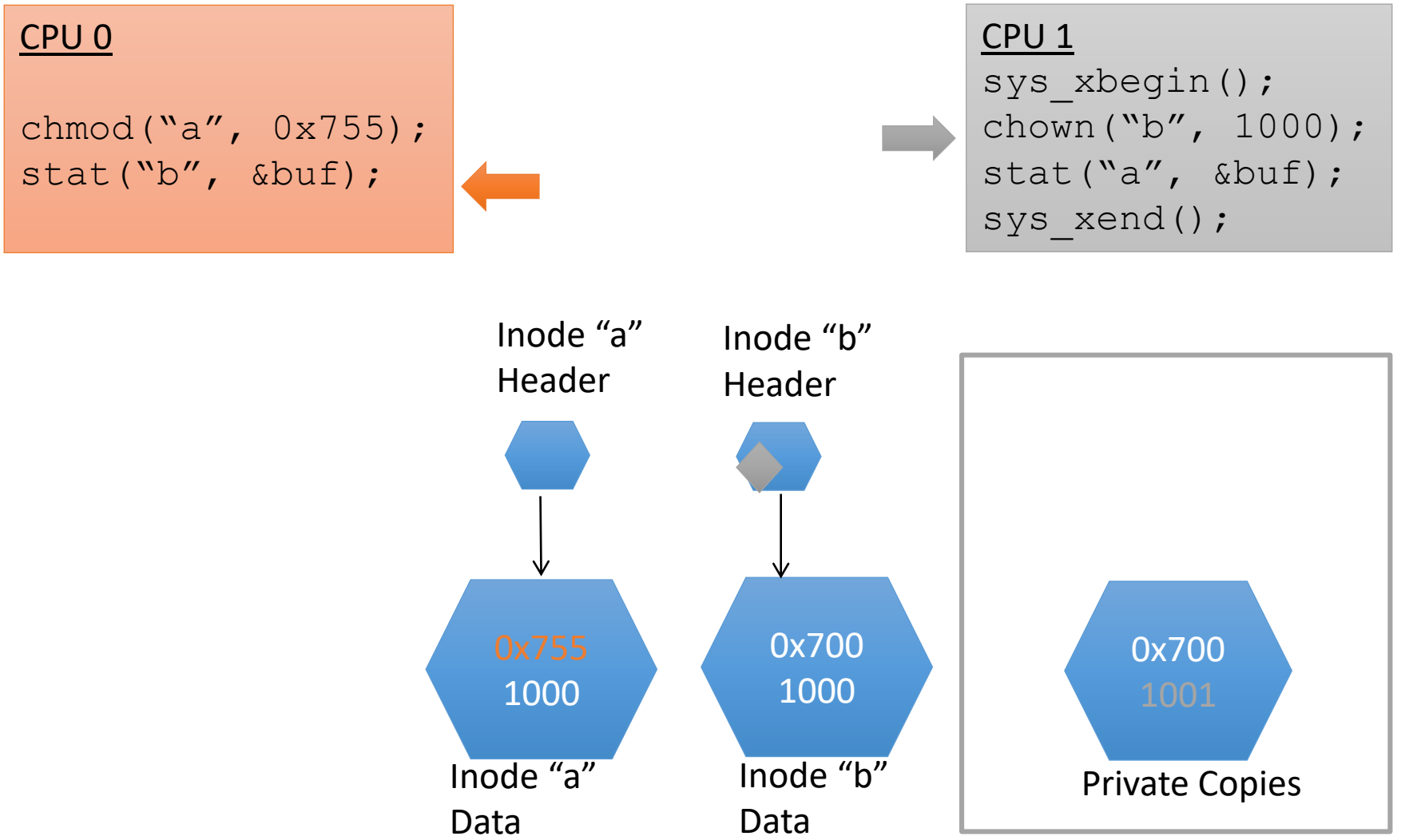
Strong isolation in TxOS



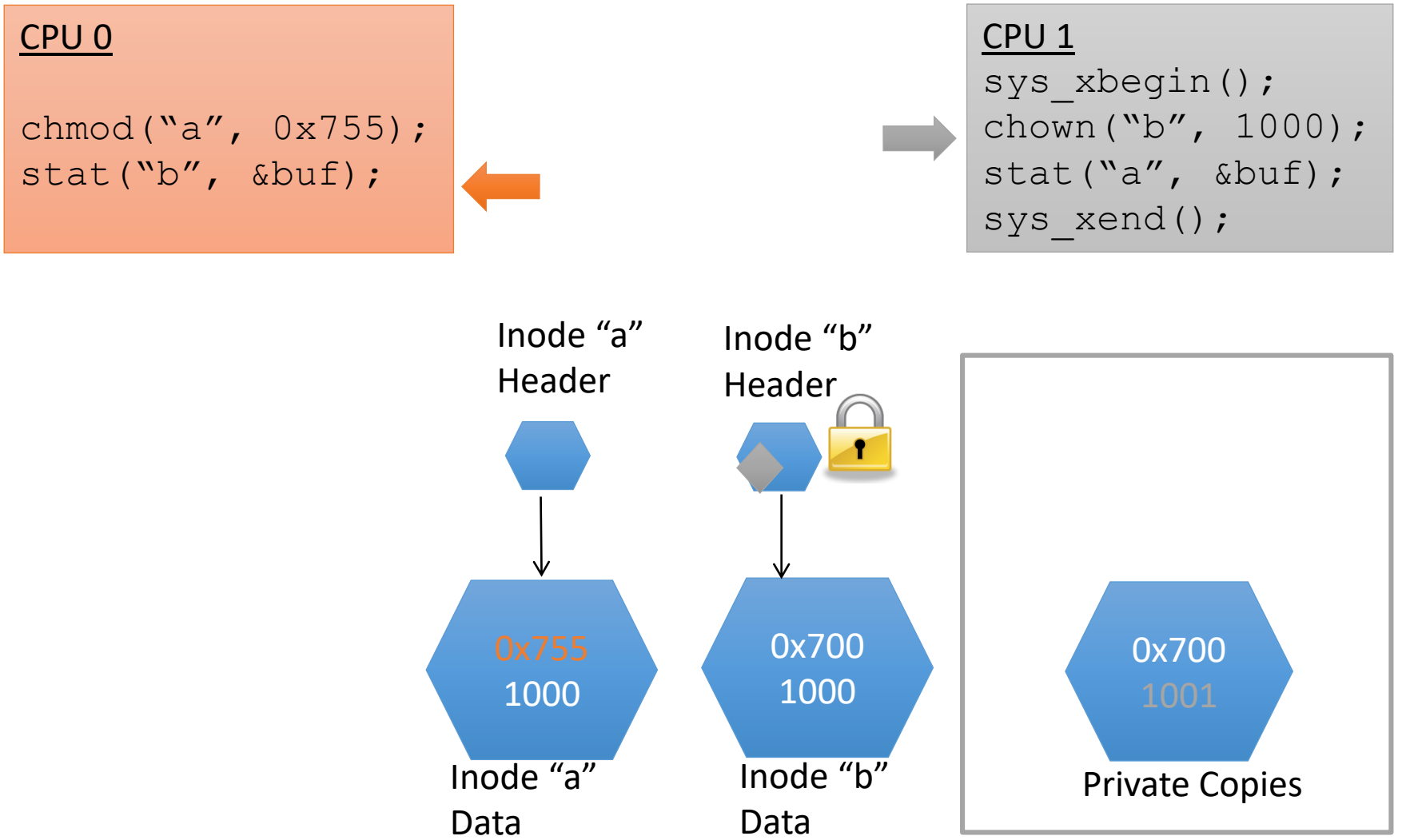
Strong isolation in TxOS



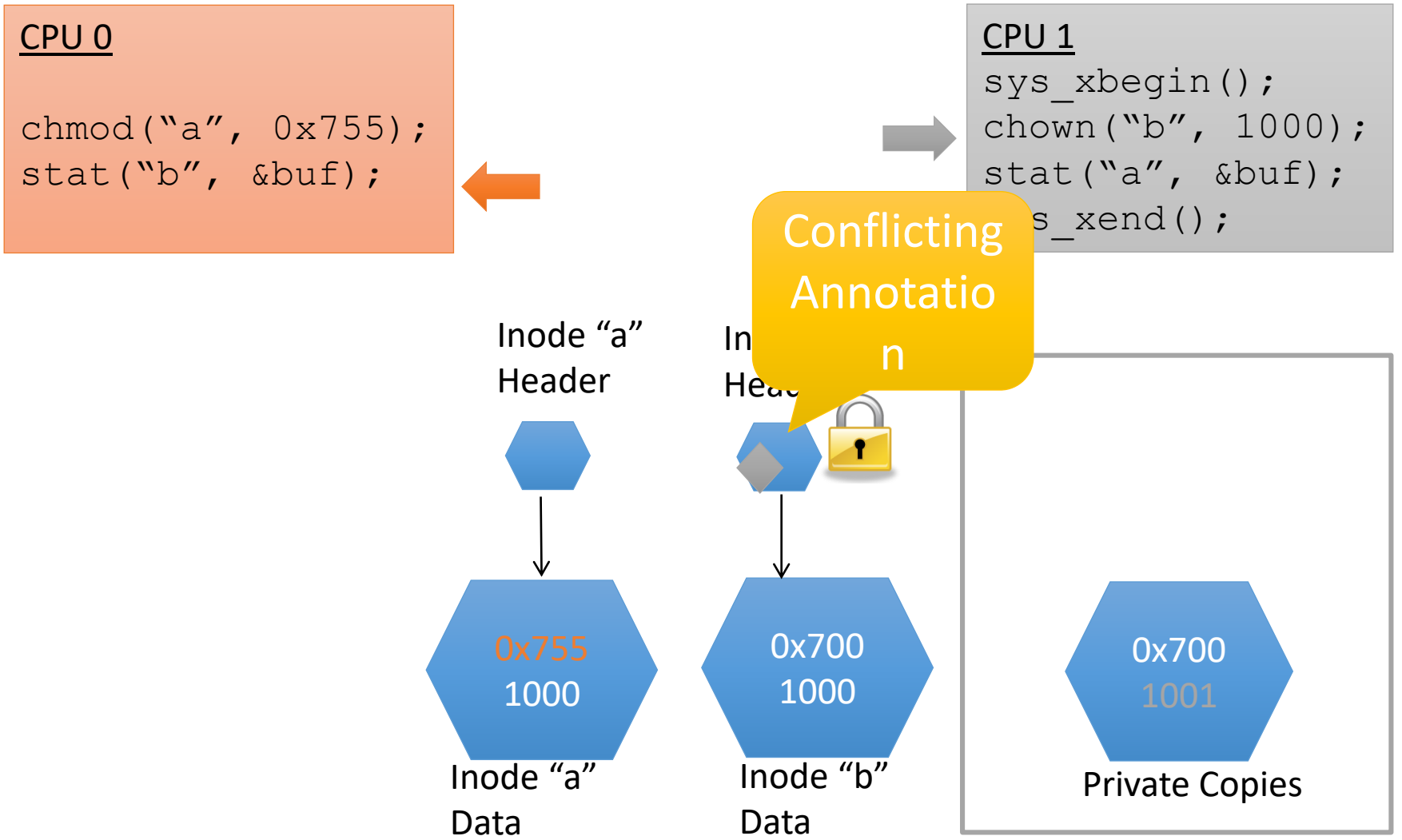
Strong isolation in TxOS



Strong isolation in TxOS



Strong isolation in TxOS



Strong isolation in TxOS

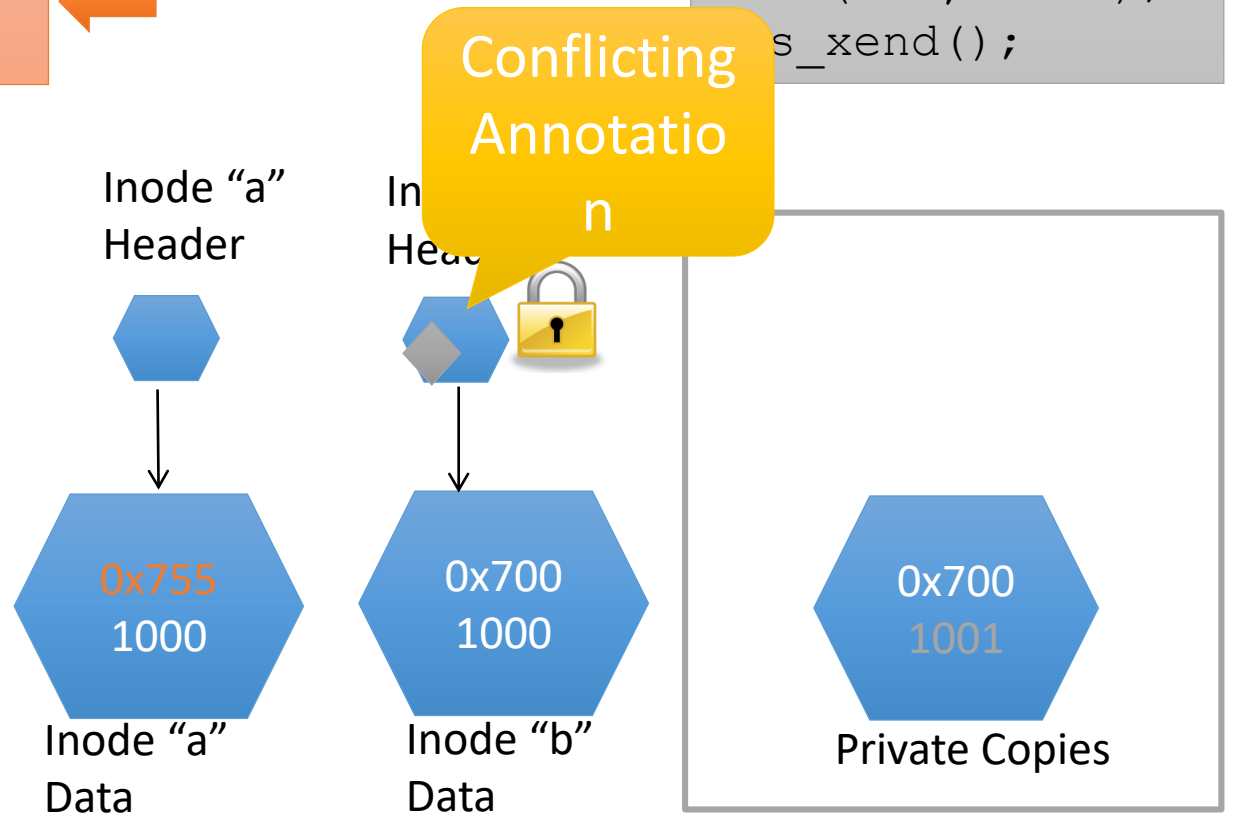
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
s_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS

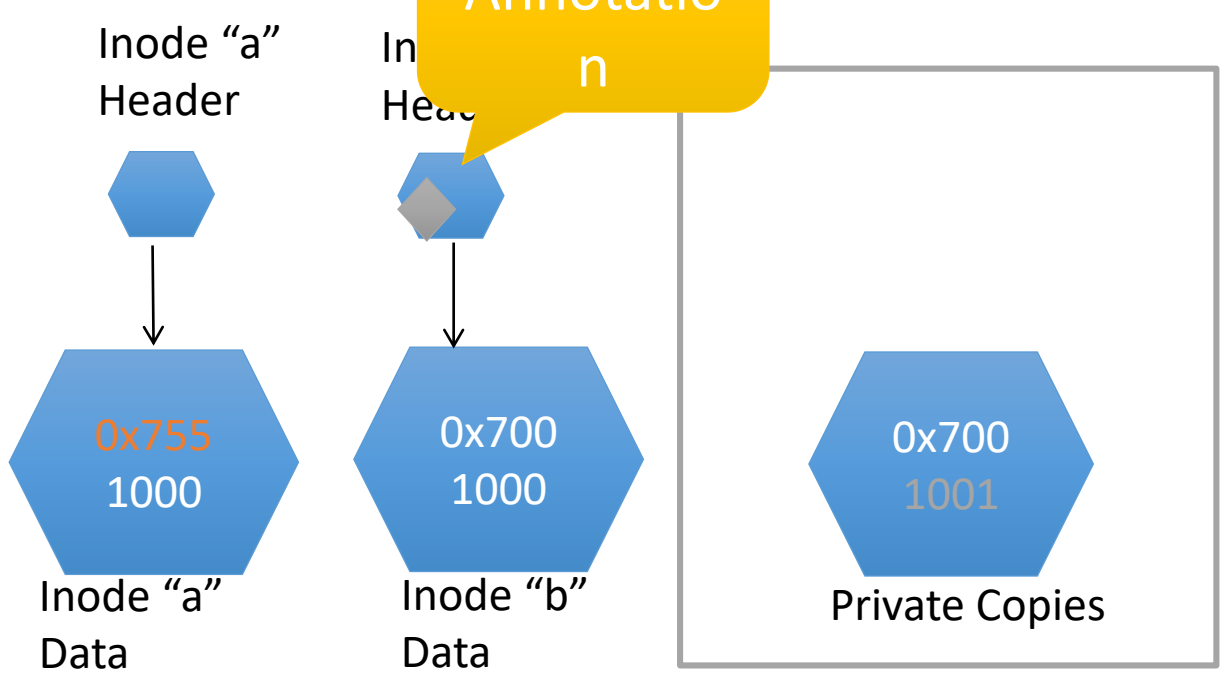
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

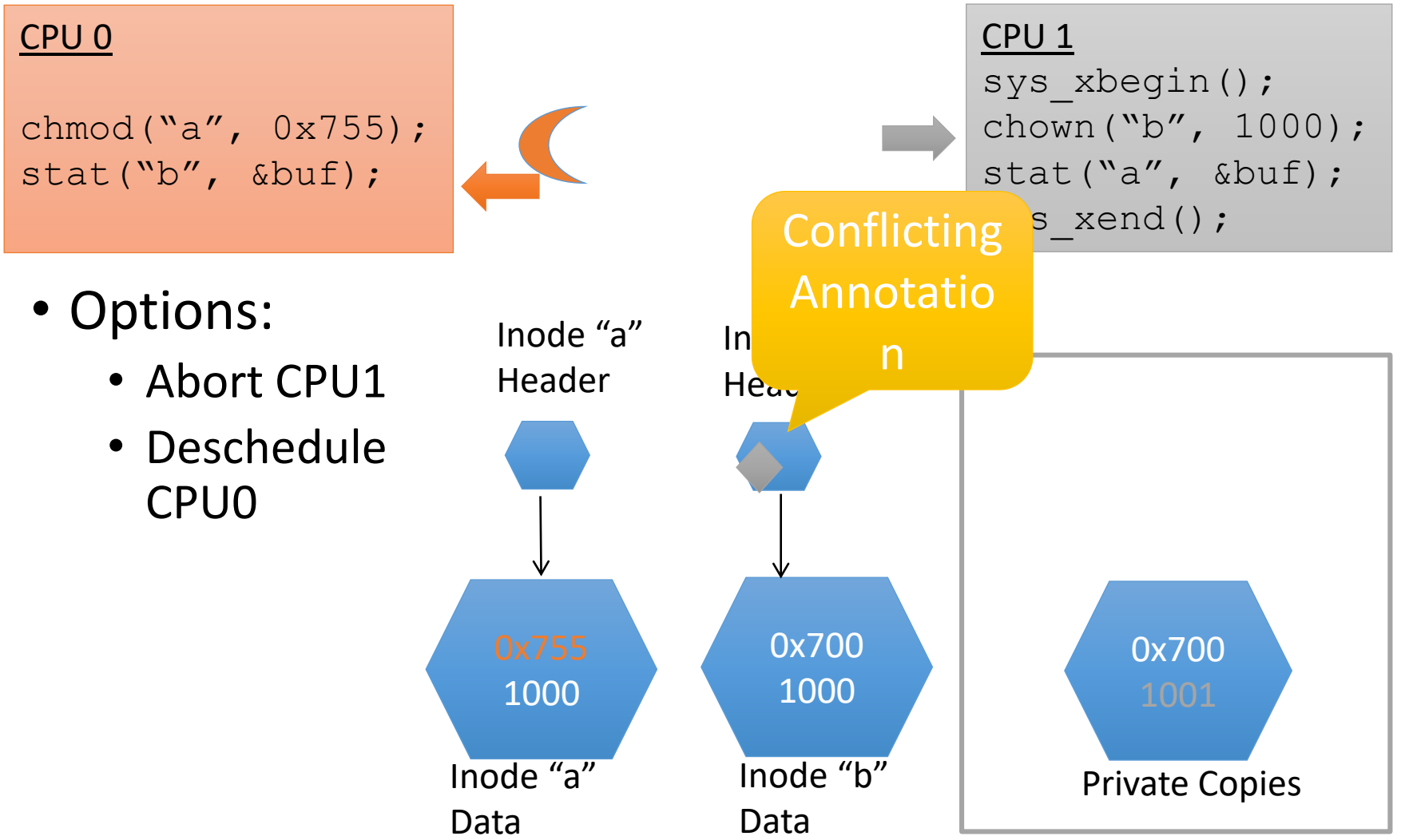
CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
s_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS



- Options:
  - Abort CPU1
  - Deschedule CPU0

Strong isolation in TxOS

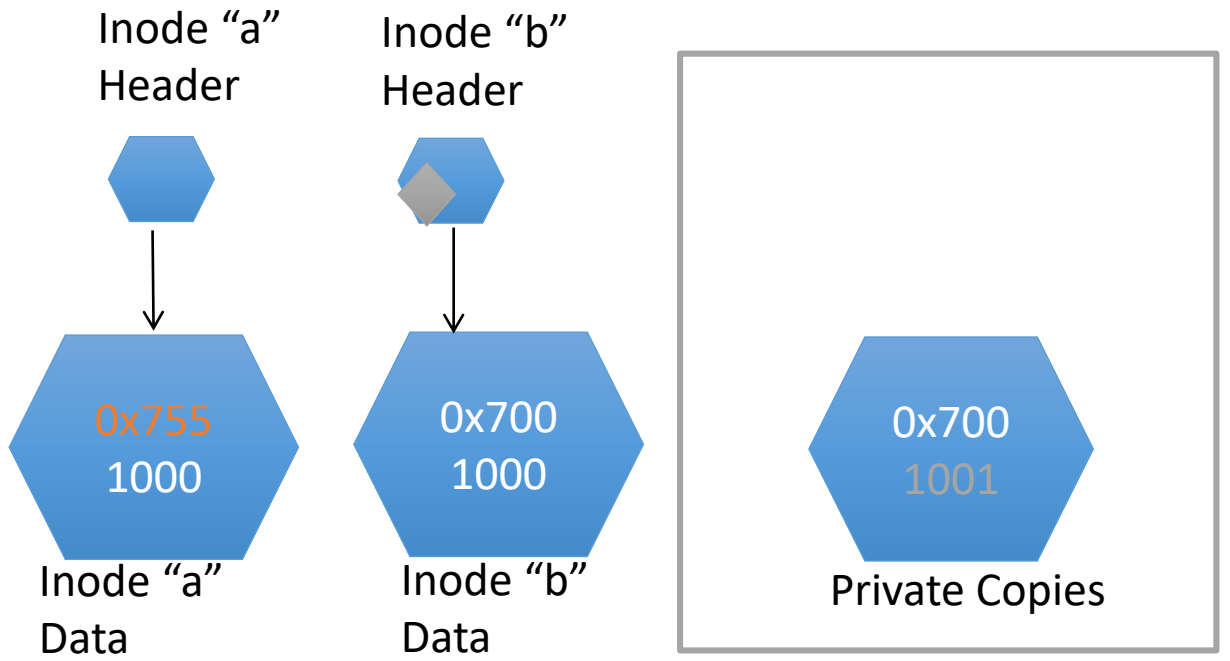
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS

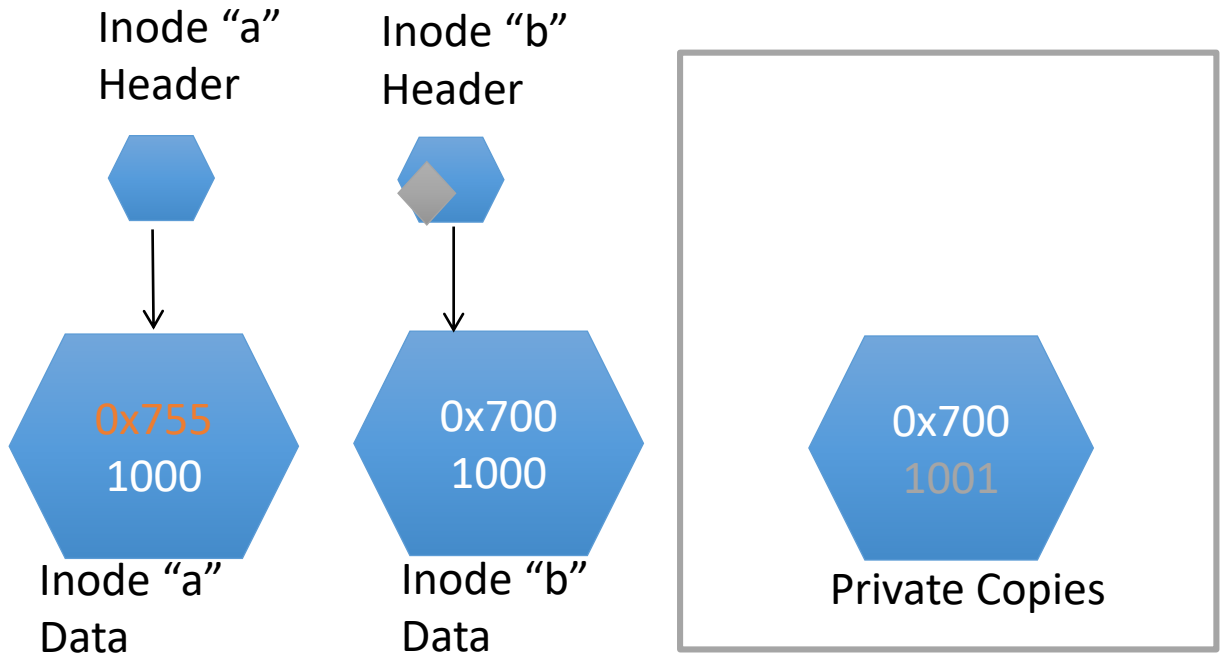
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS

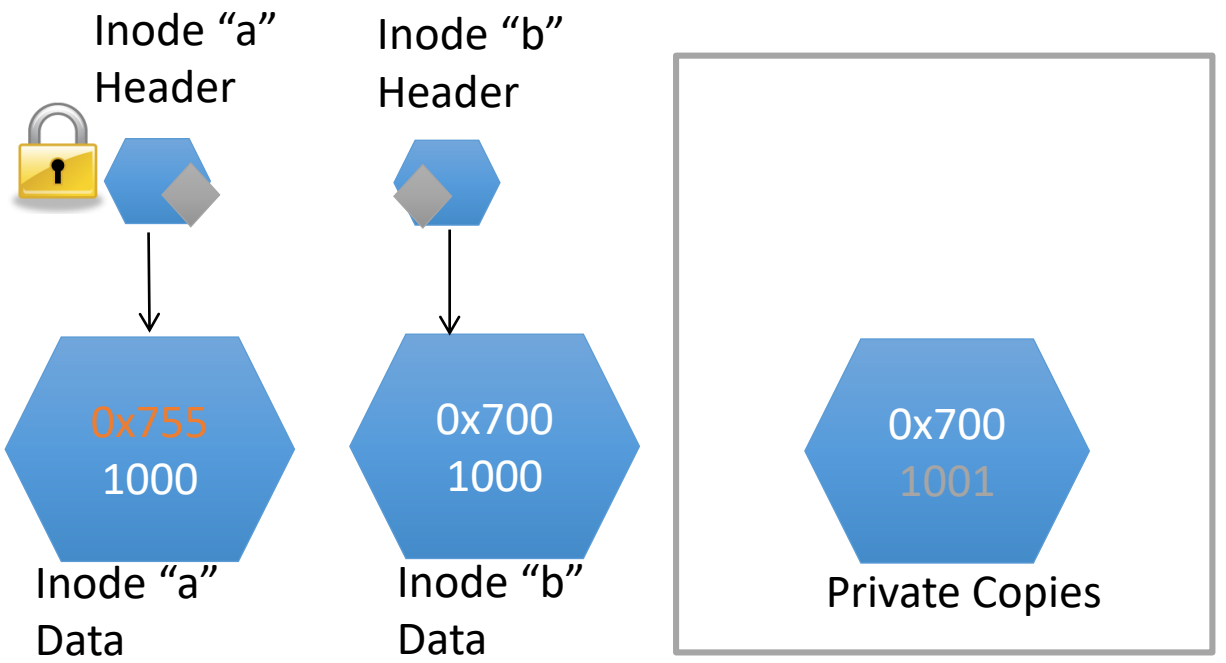
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS

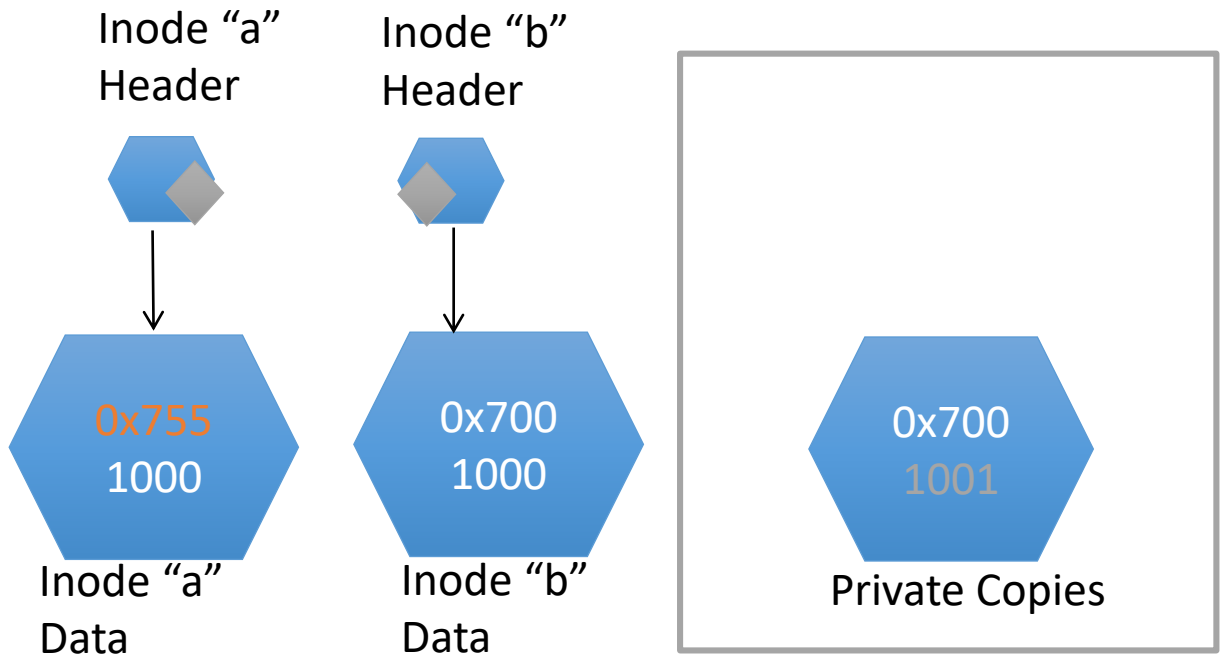
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS

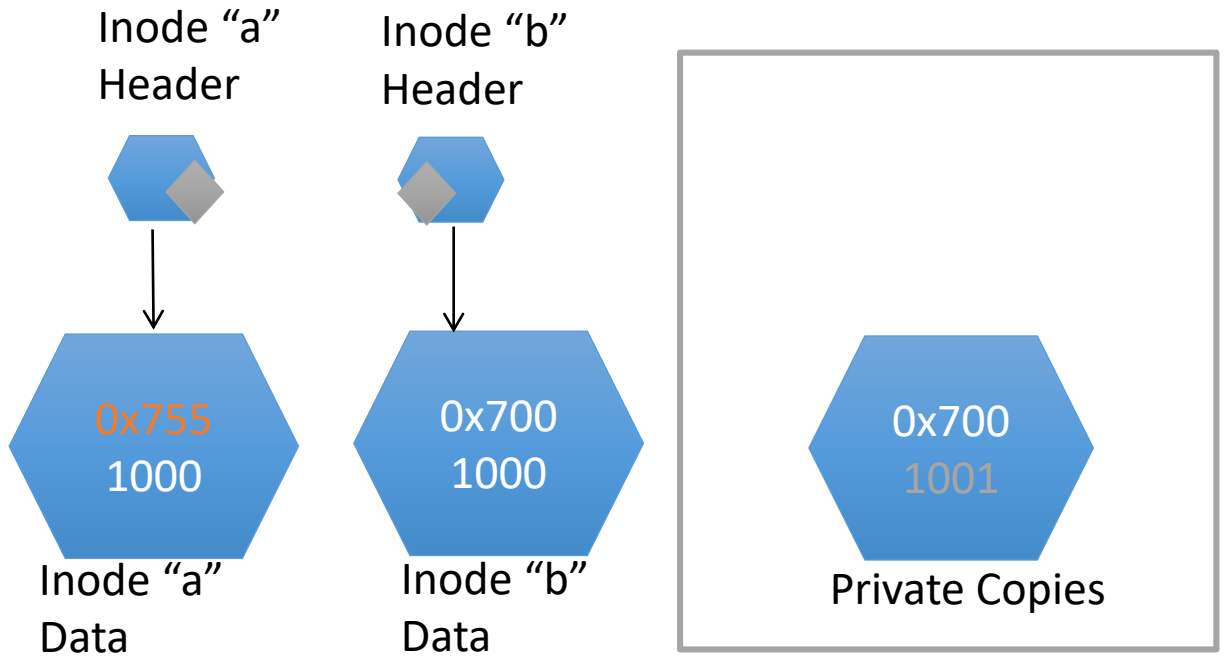
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS

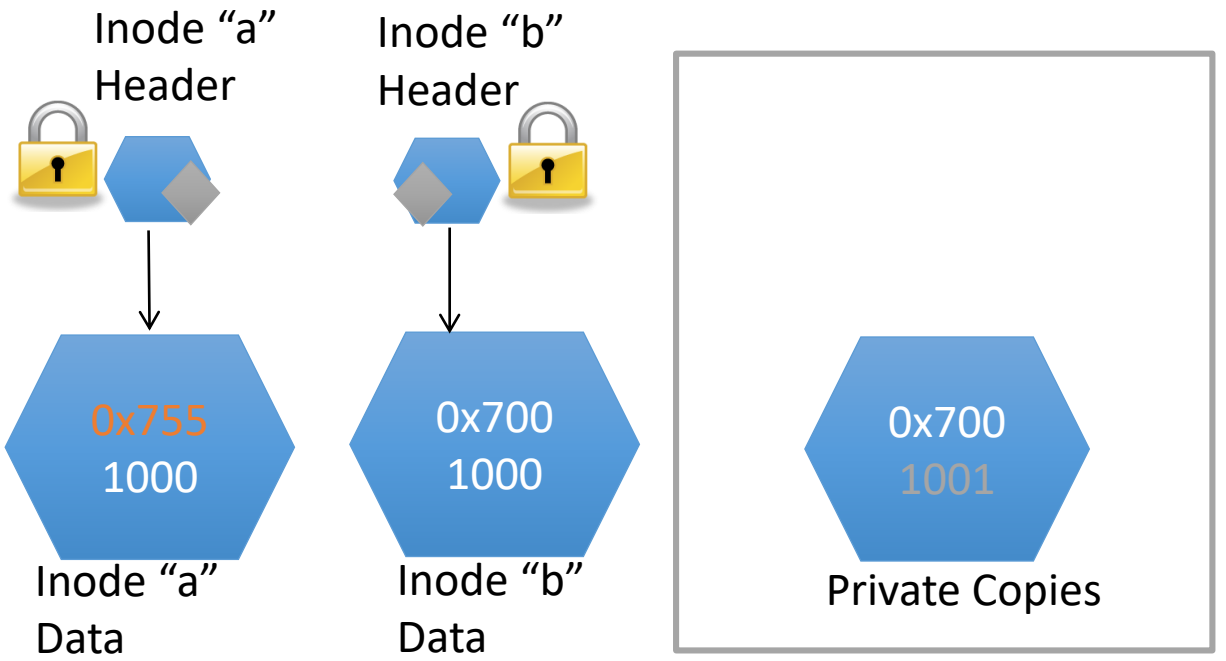
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS

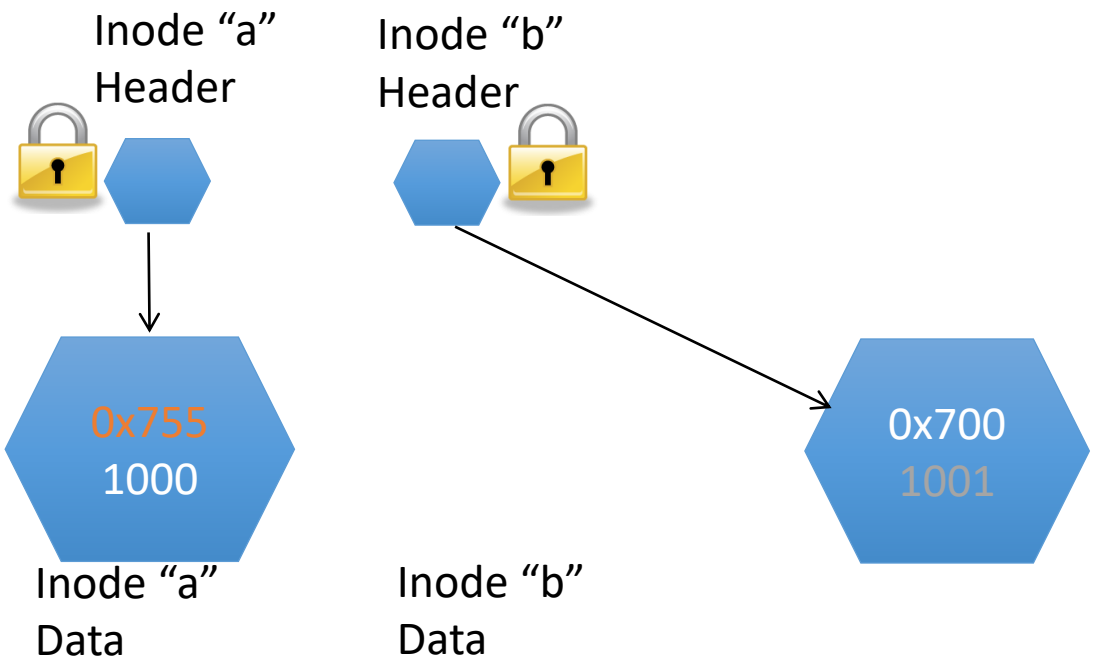
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS

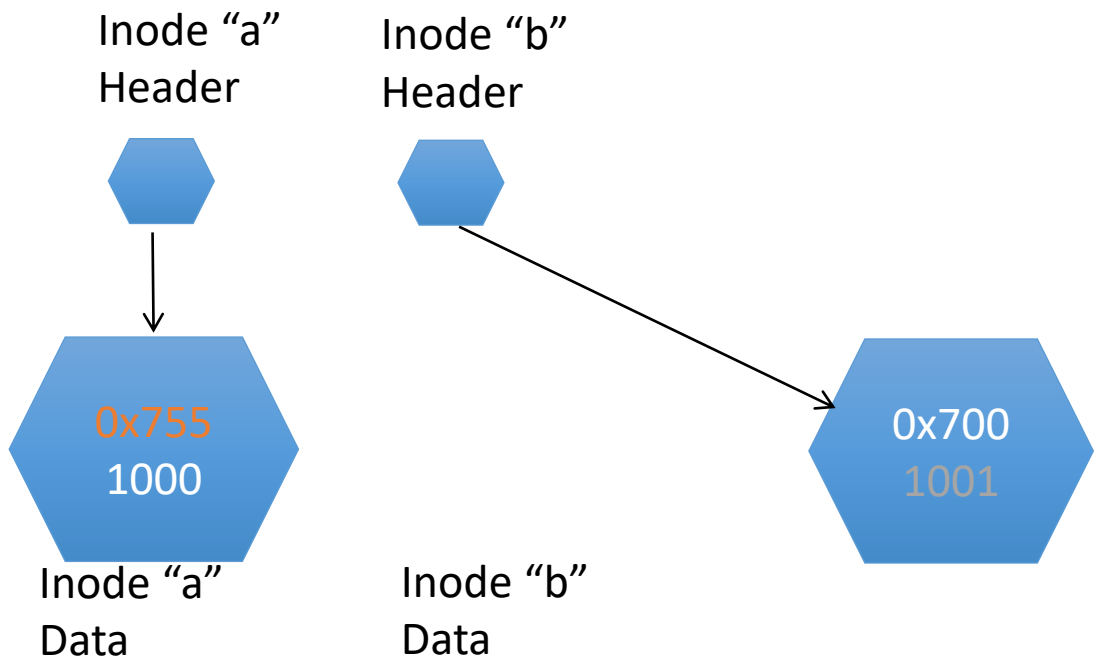
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS

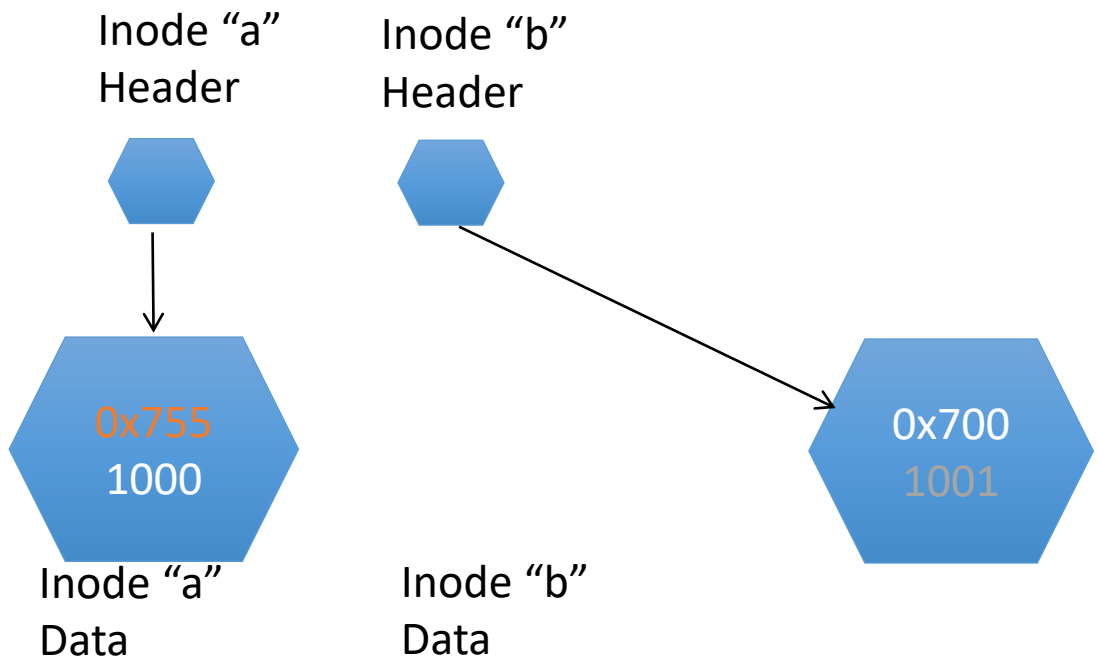
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS

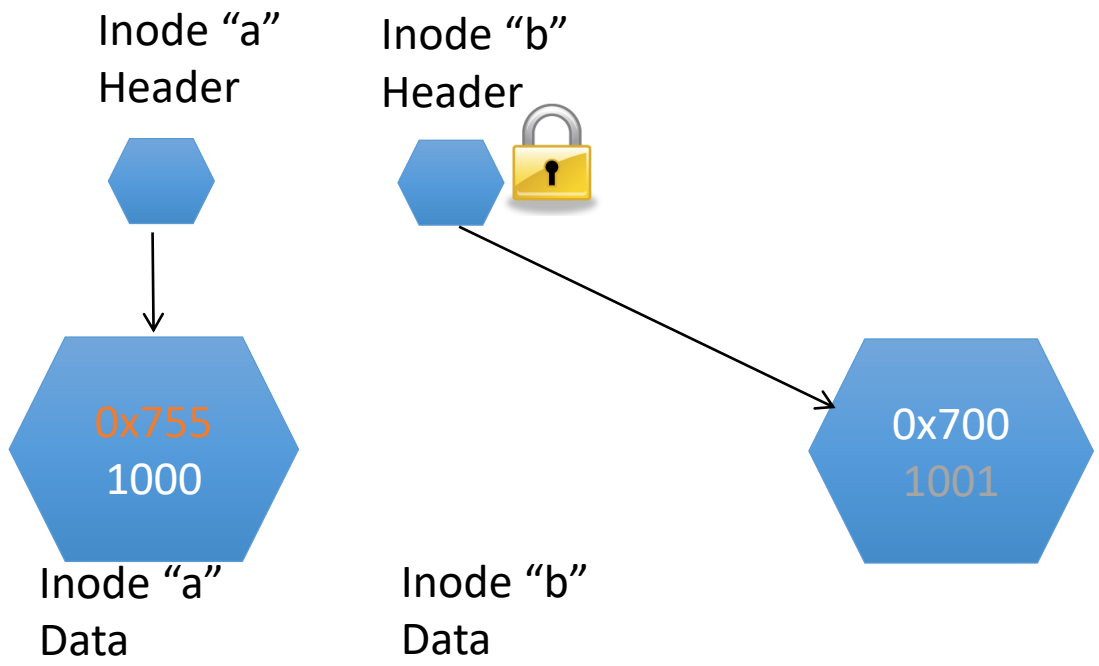
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS

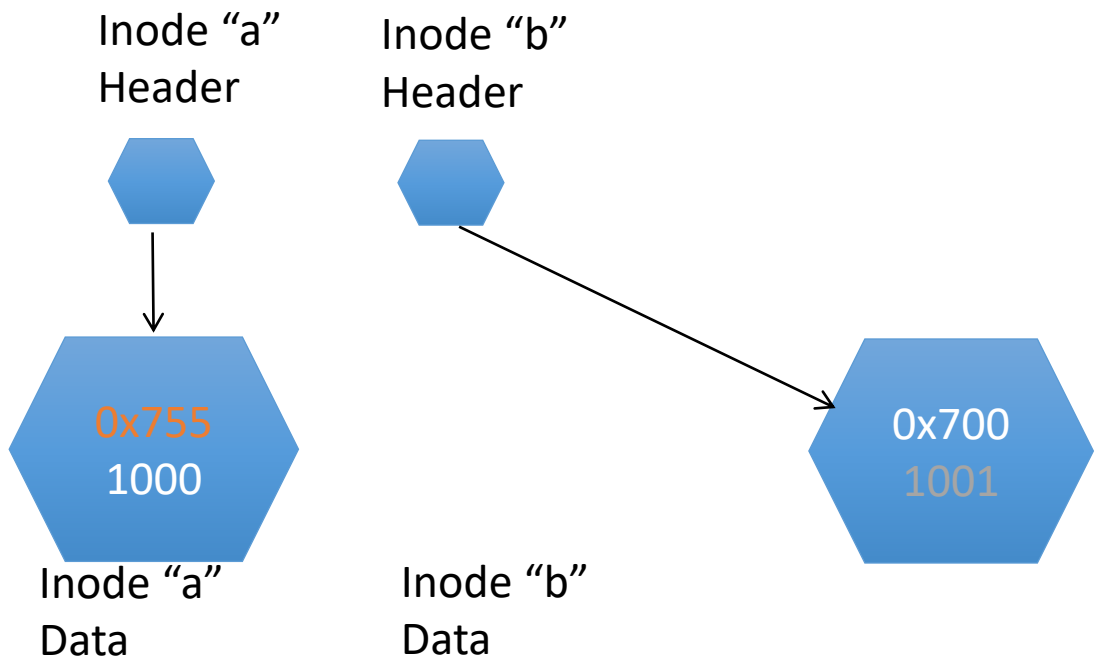
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```

- Options:
  - Abort CPU1
  - Deschedule CPU0



Strong isolation in TxOS

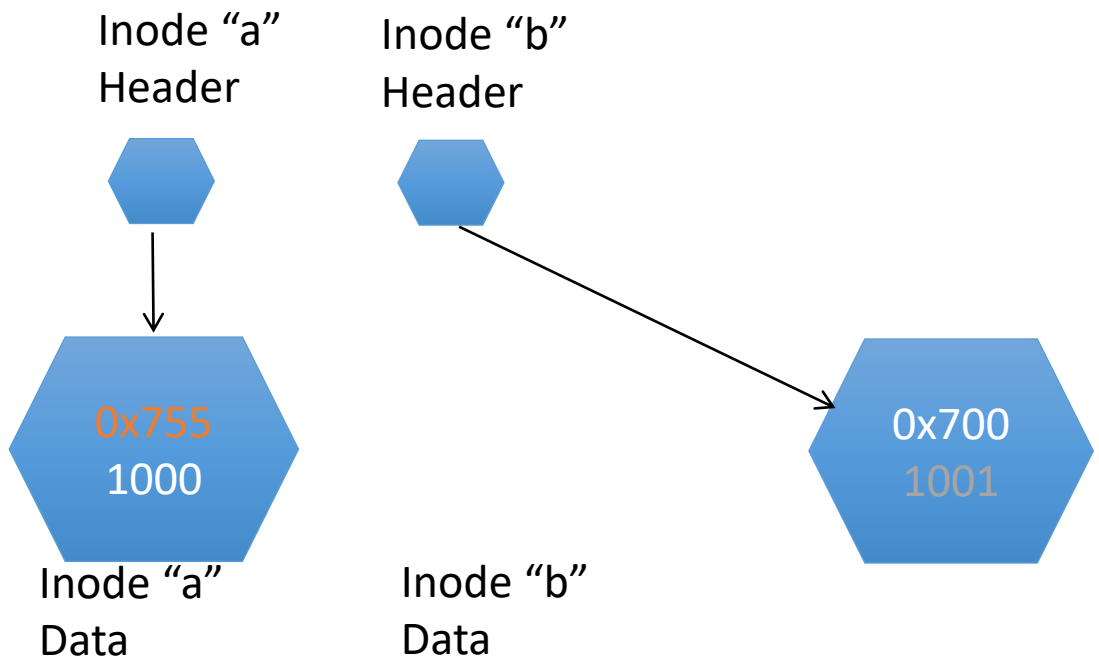
CPU 0

```
chmod("a", 0x755);
stat("b", &buf);
```

CPU 1

```
sys_xbegin();
chown("b", 1000);
stat("a", &buf);
sys_xend();
```

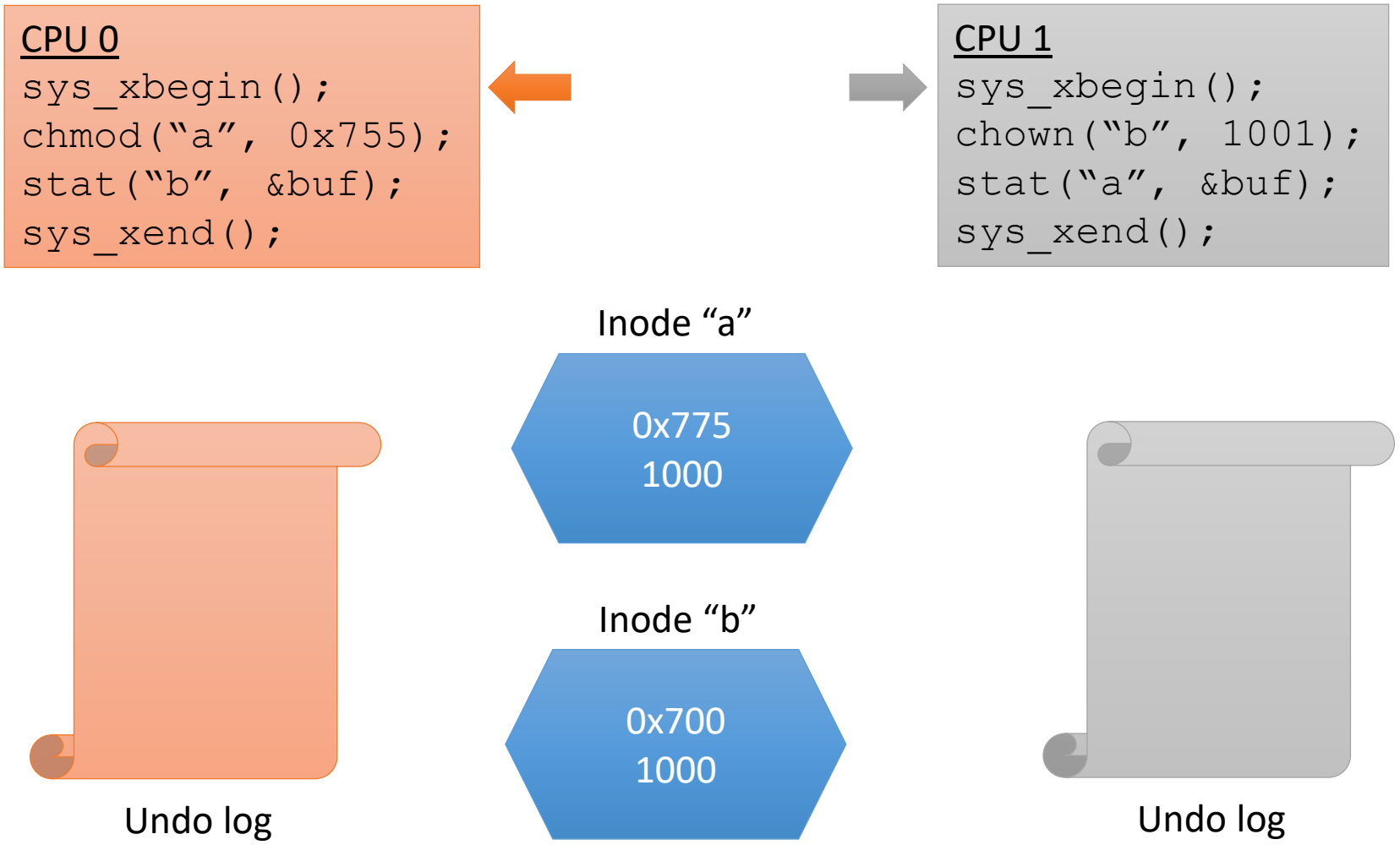
- Options:
  - Abort CPU1
  - Deschedule CPU0



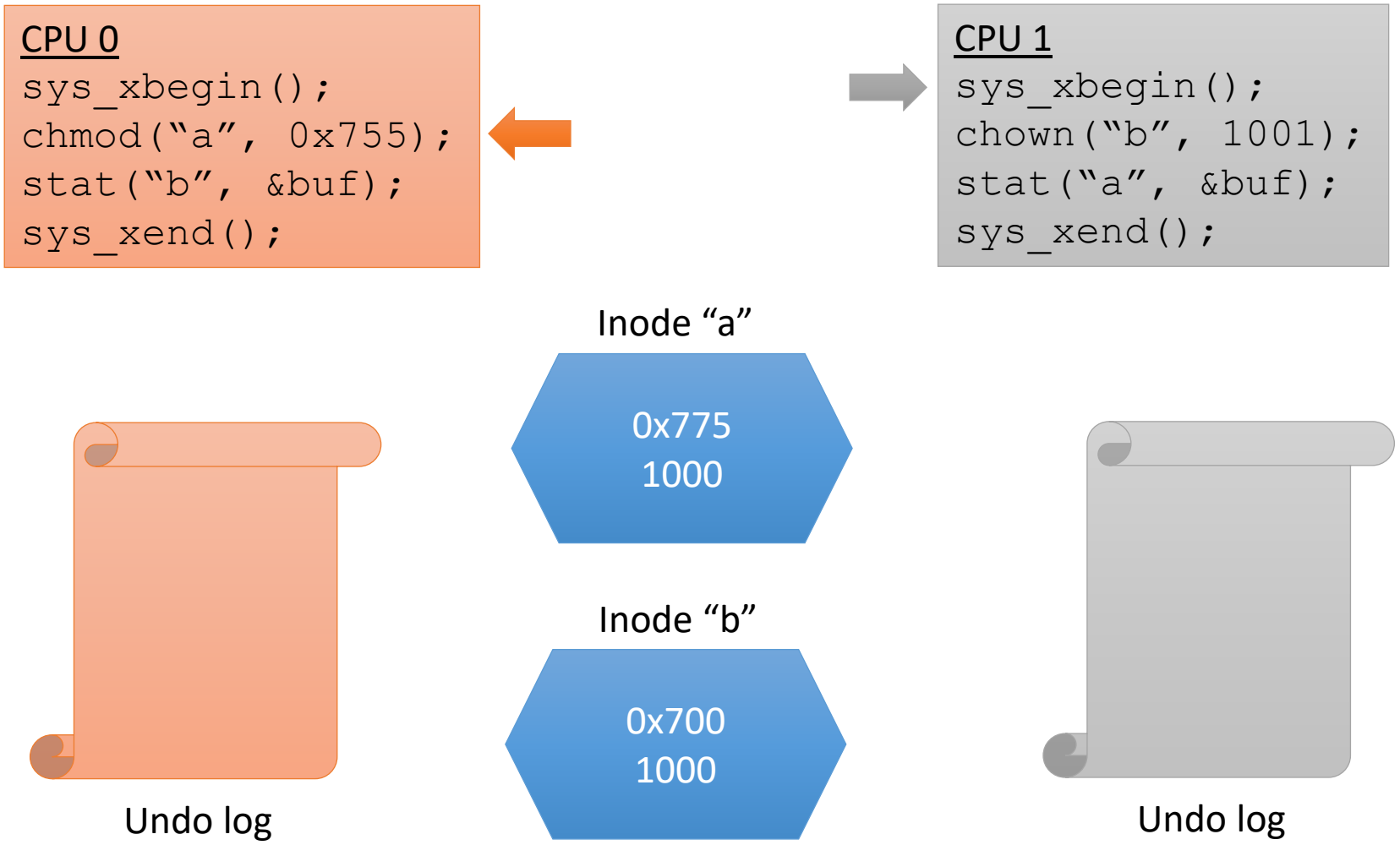
Traditional design take-aways

- In-place updates and undo log
  - Fast commit
  - Aborts stall both winning and losing transaction
  - Slow aborts can undermine scheduling policy
- Two-phase locking
  - Deadlock-prone
  - Detect with timeout or cycle detection

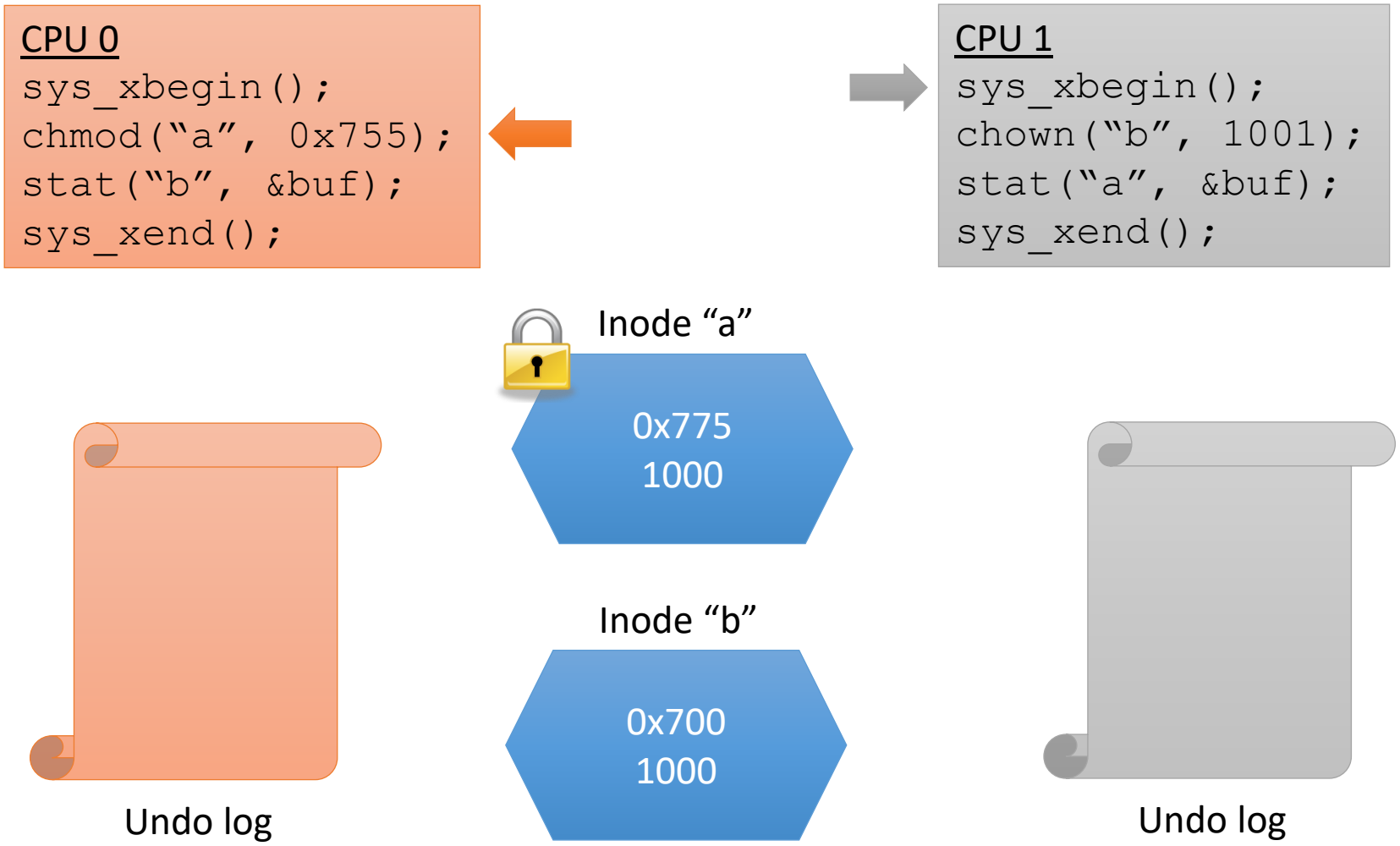
Traditional transaction design



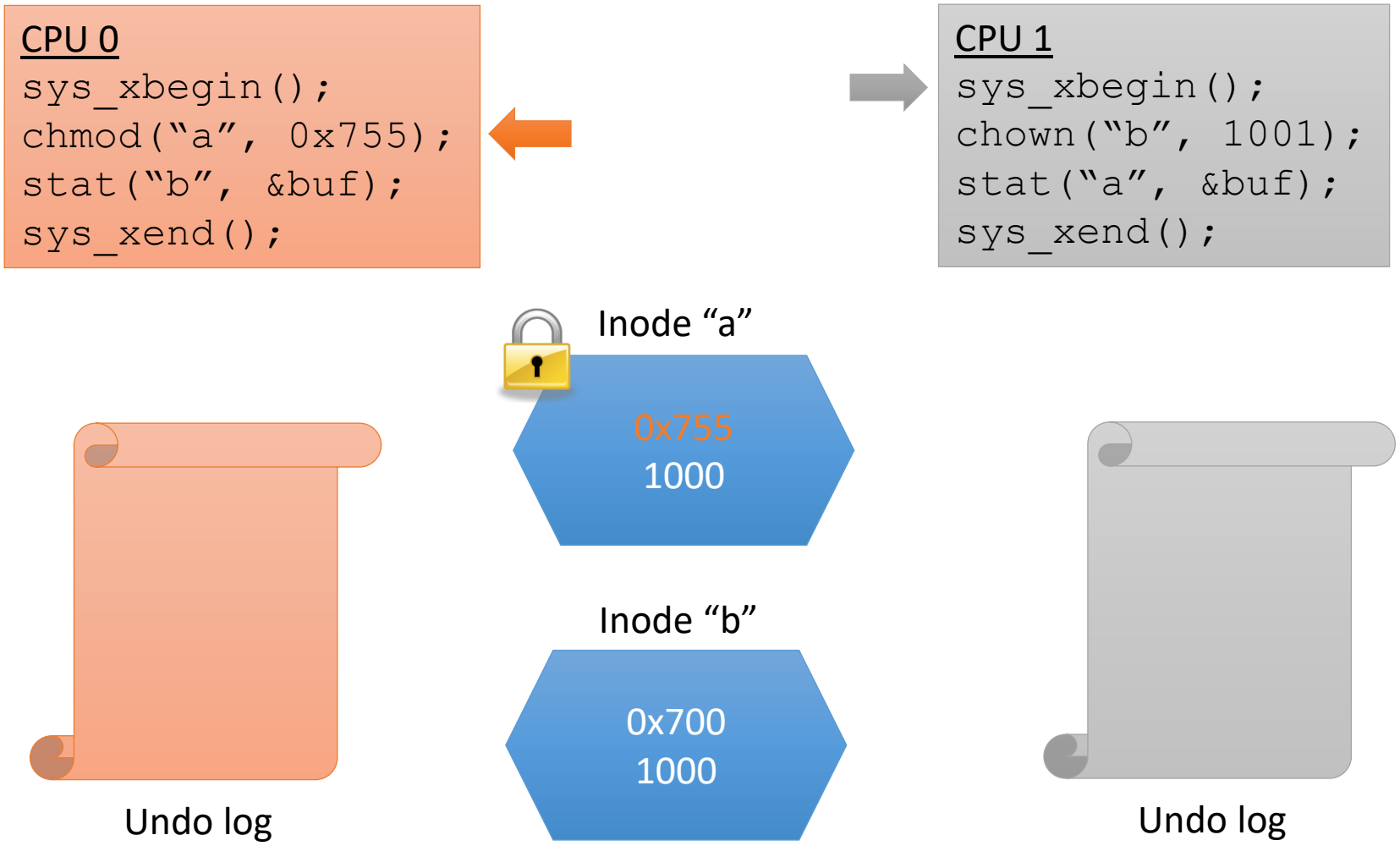
Traditional transaction design



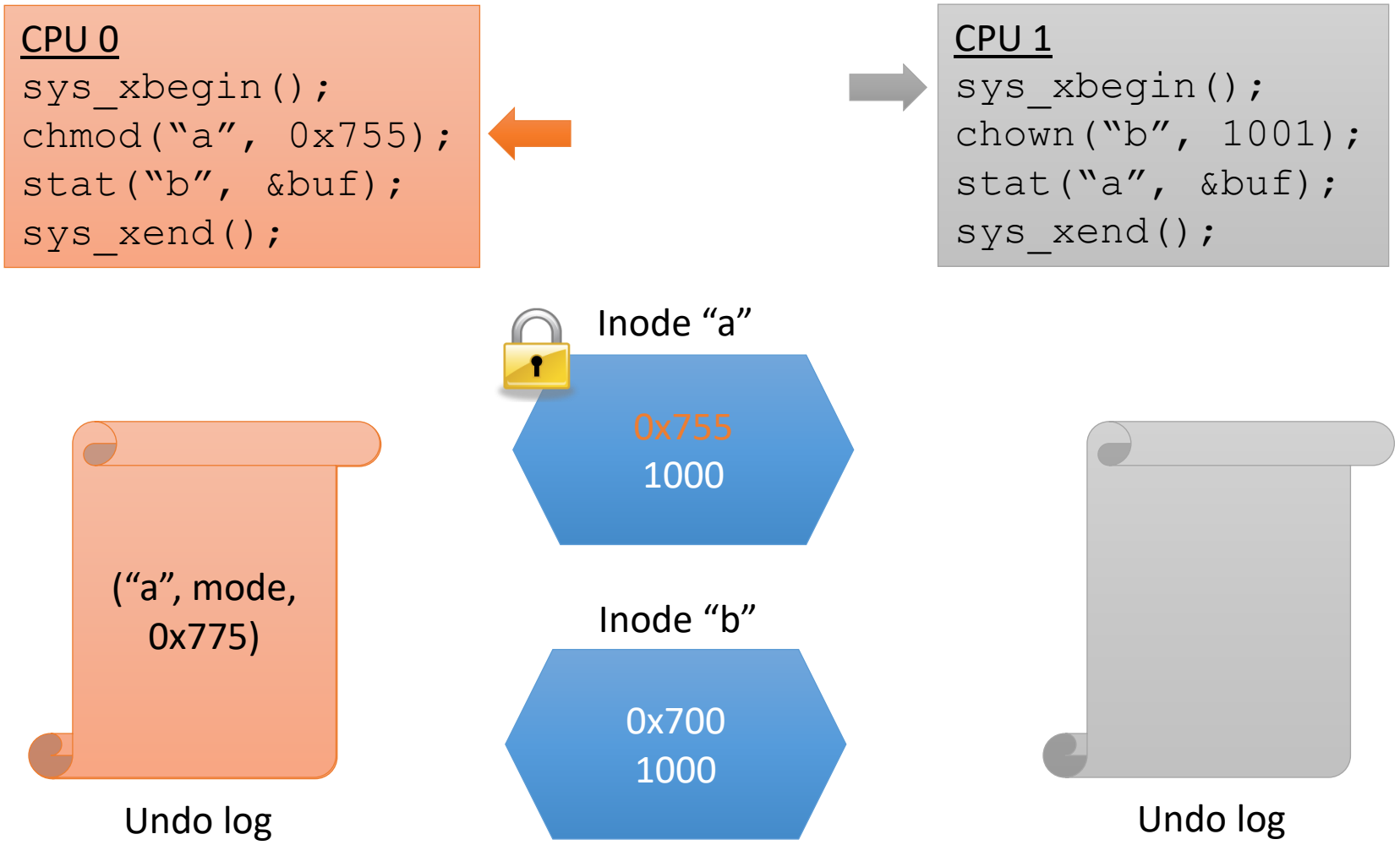
Traditional transaction design



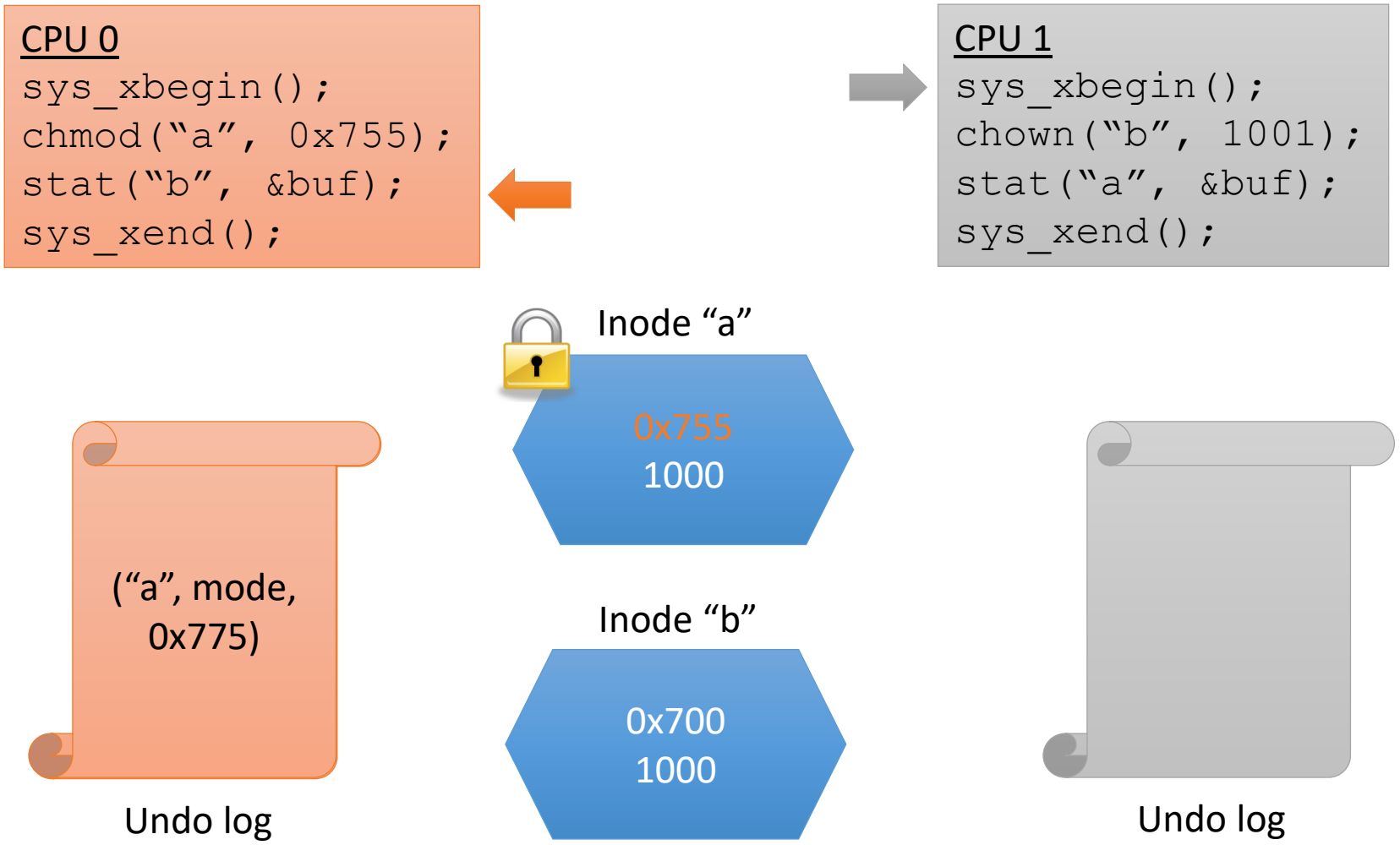
Traditional transaction design



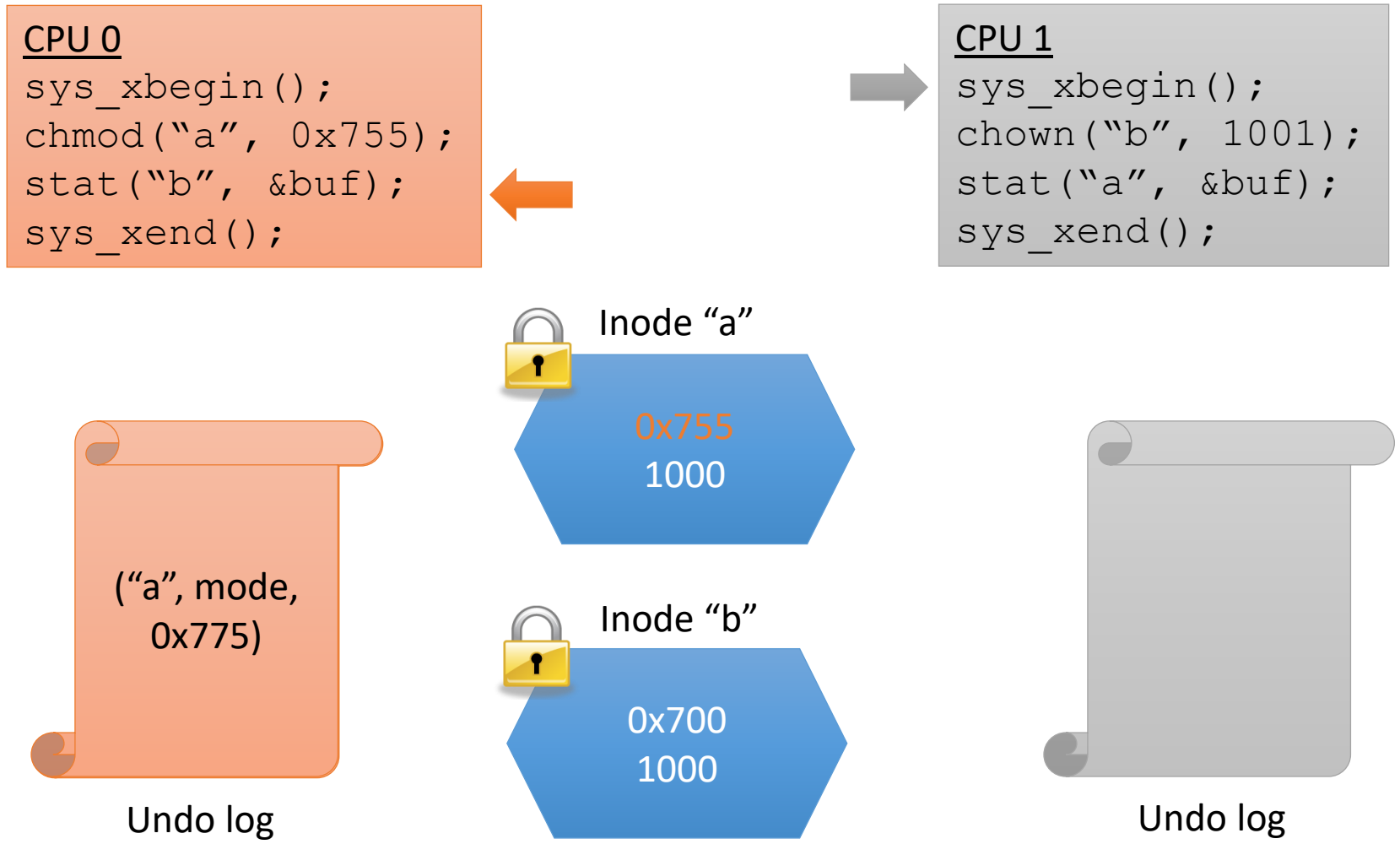
Traditional transaction design



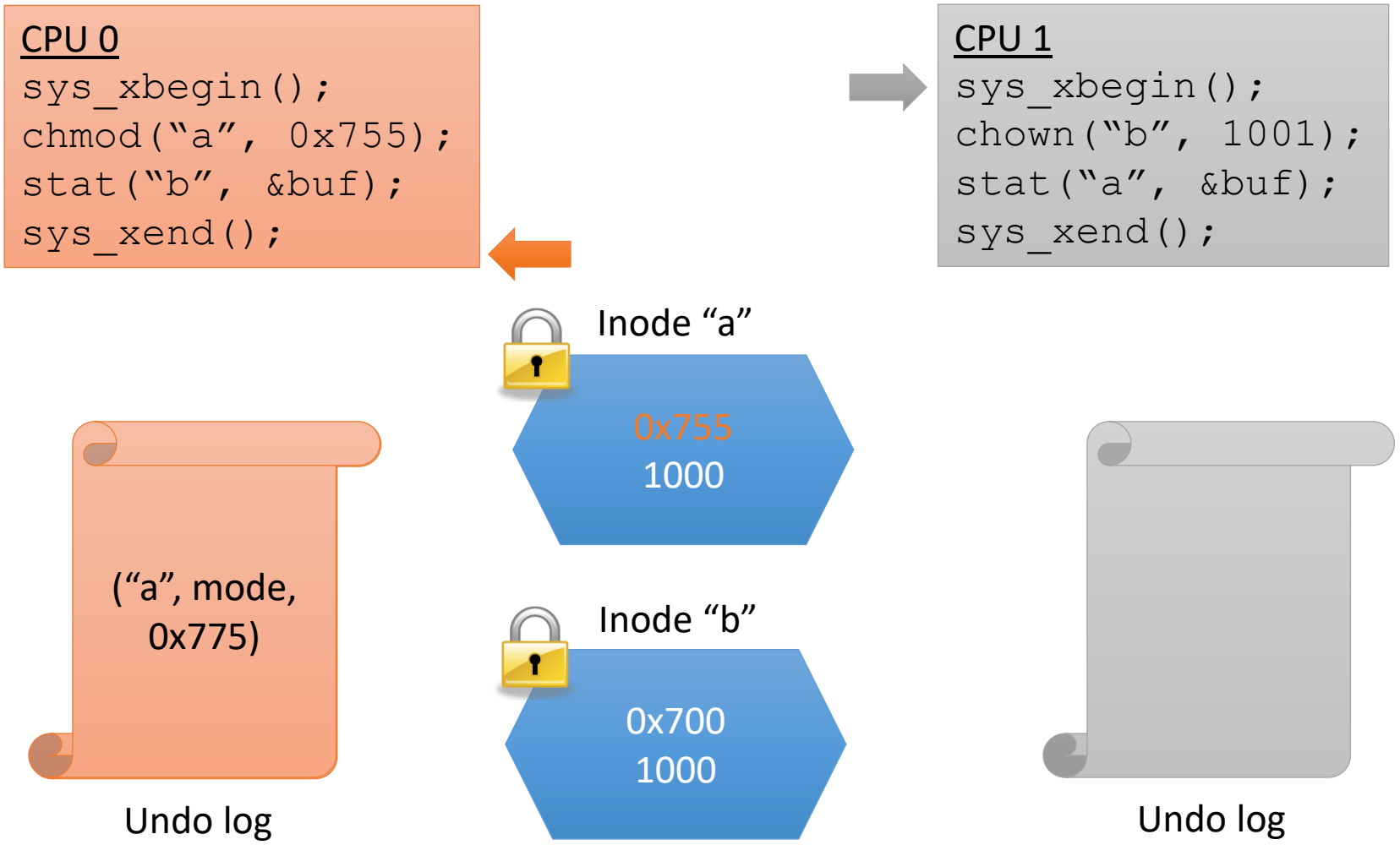
Traditional transaction design



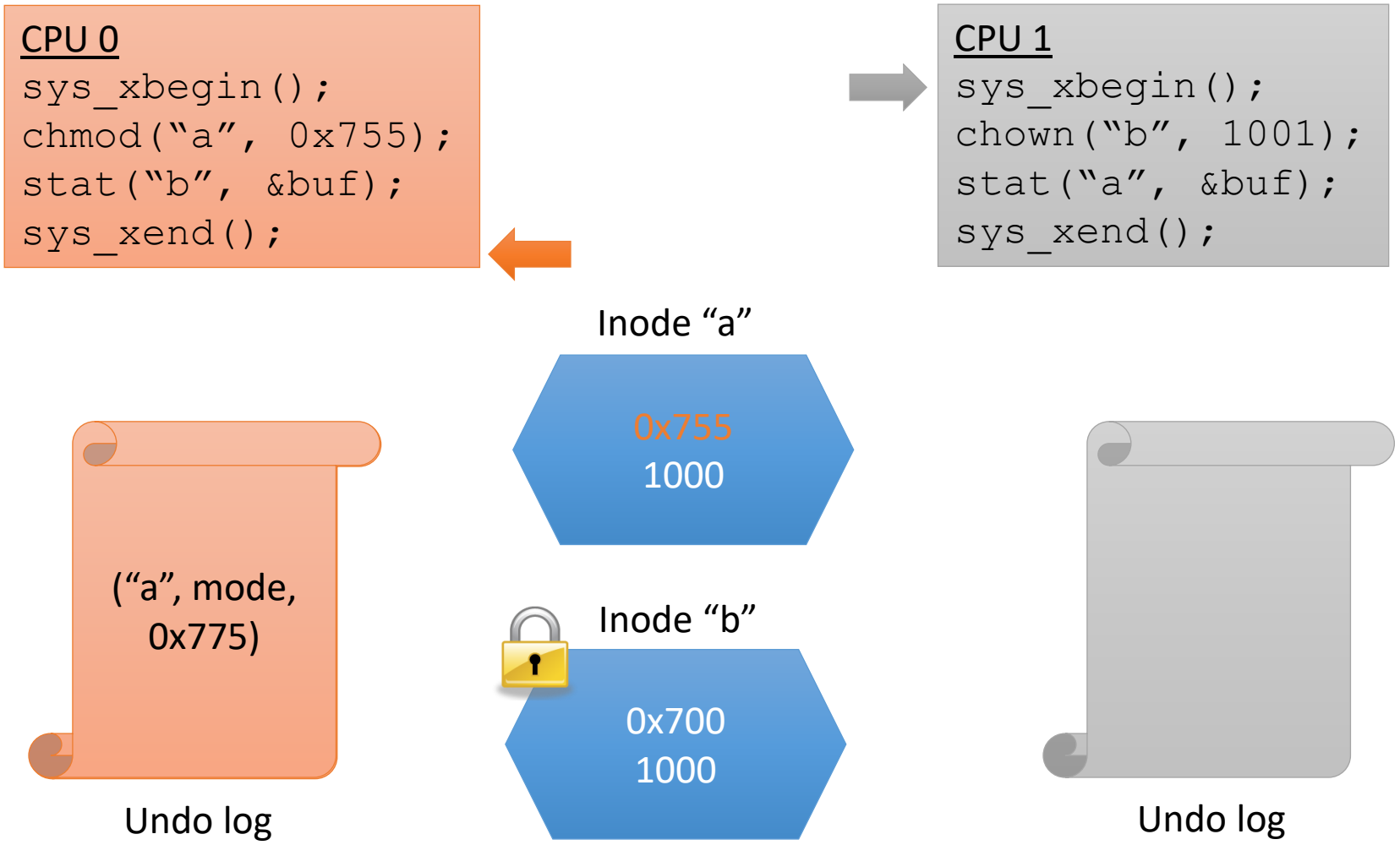
Traditional transaction design



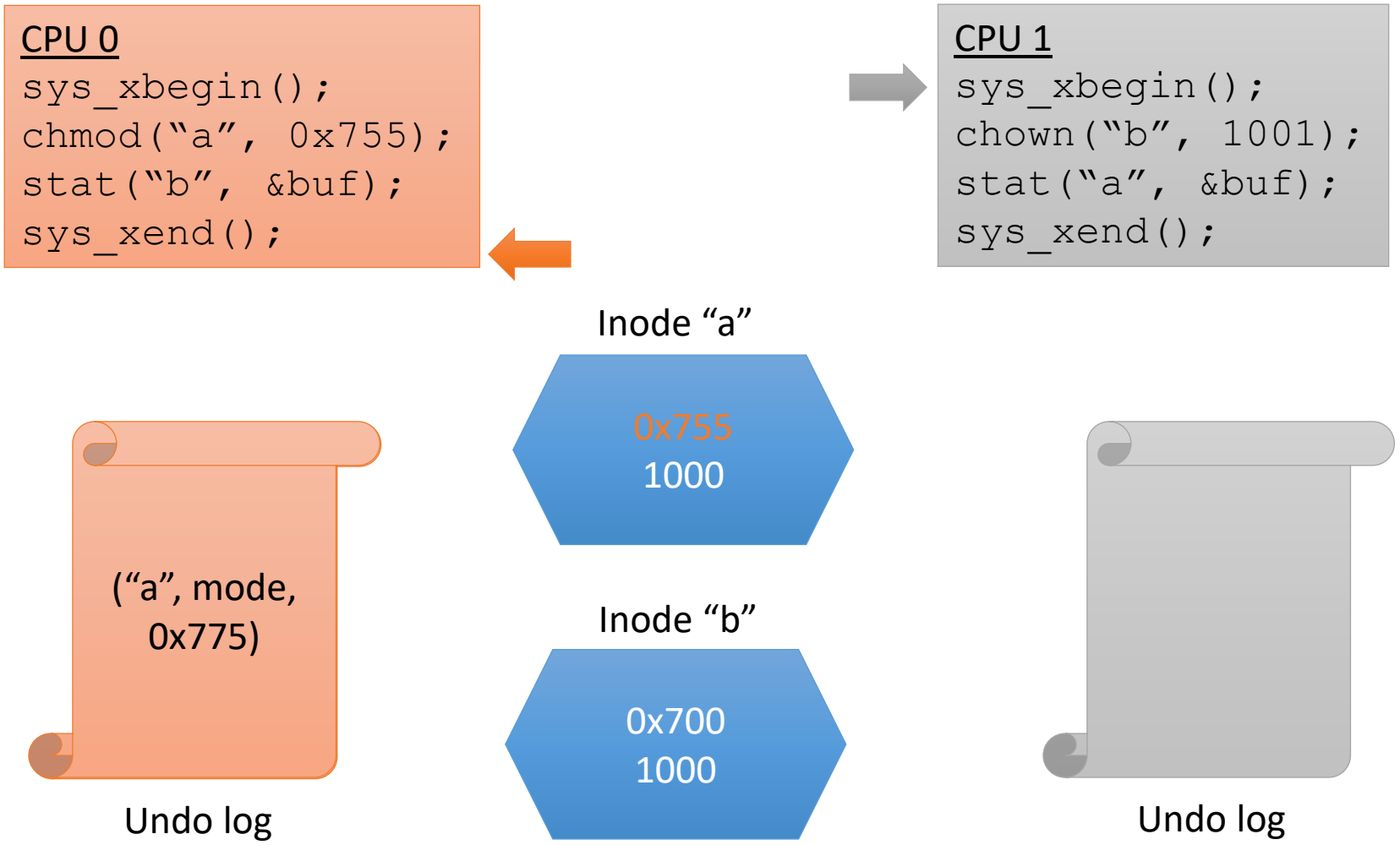
Traditional transaction design



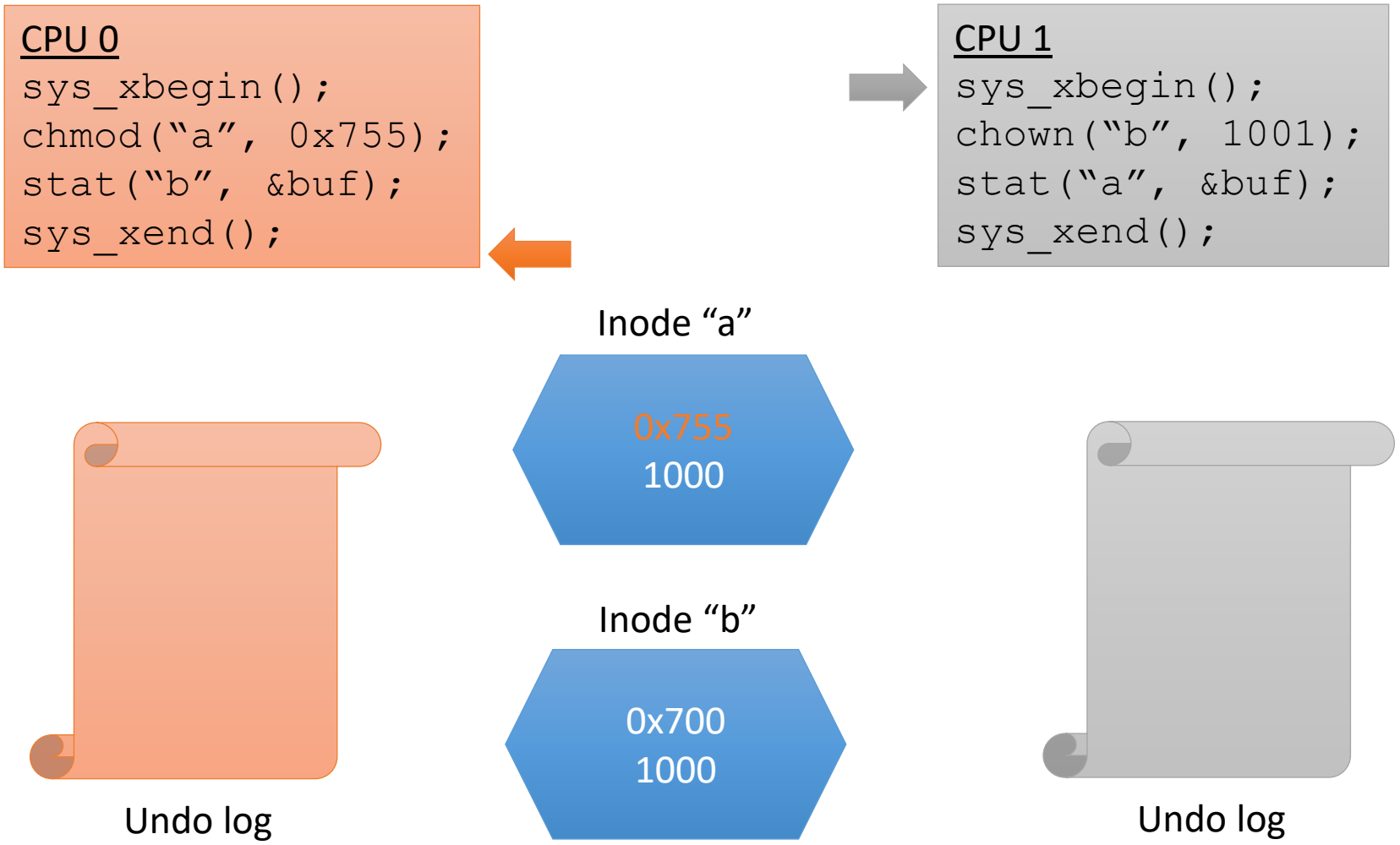
Traditional transaction design



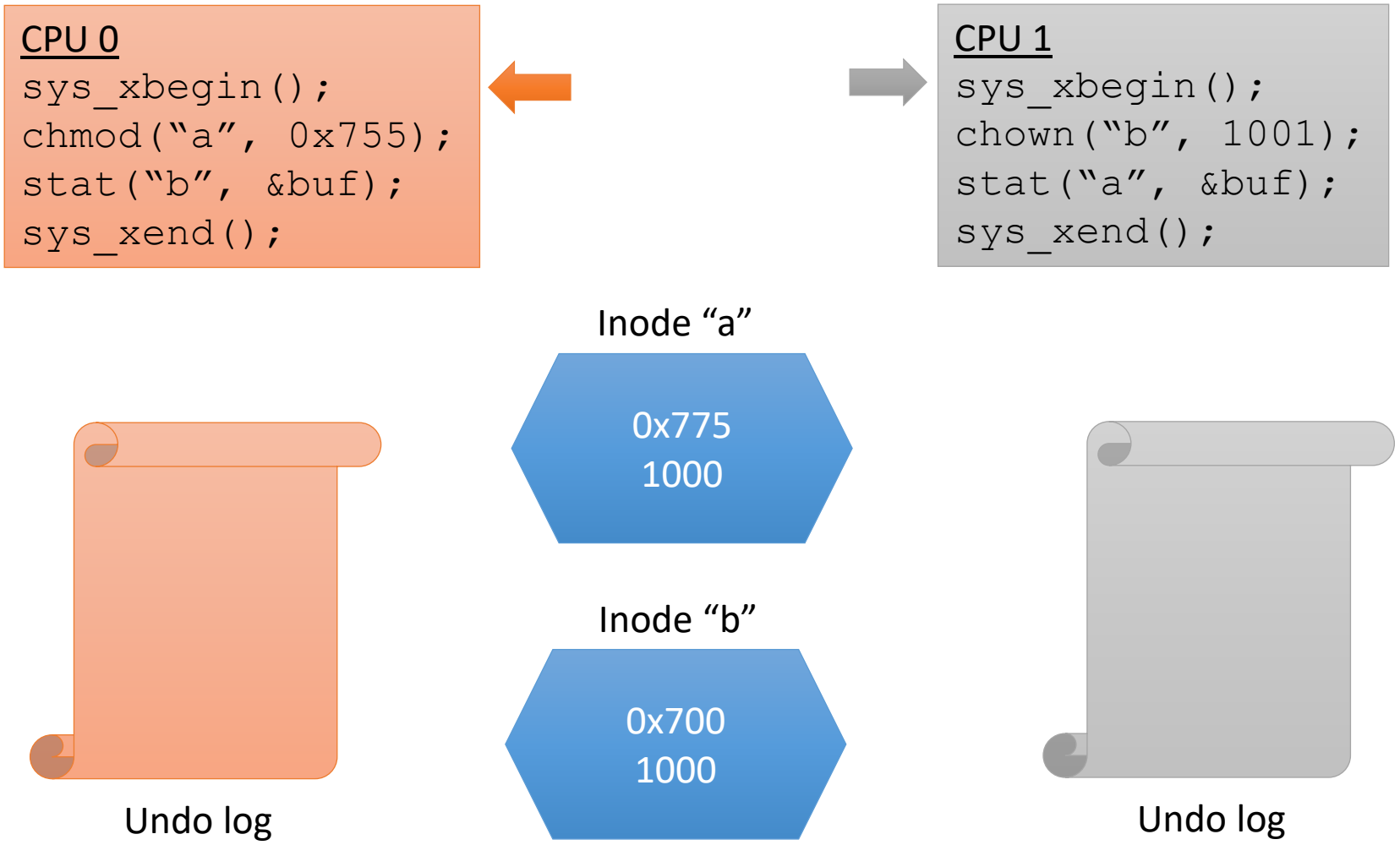
Traditional transaction design



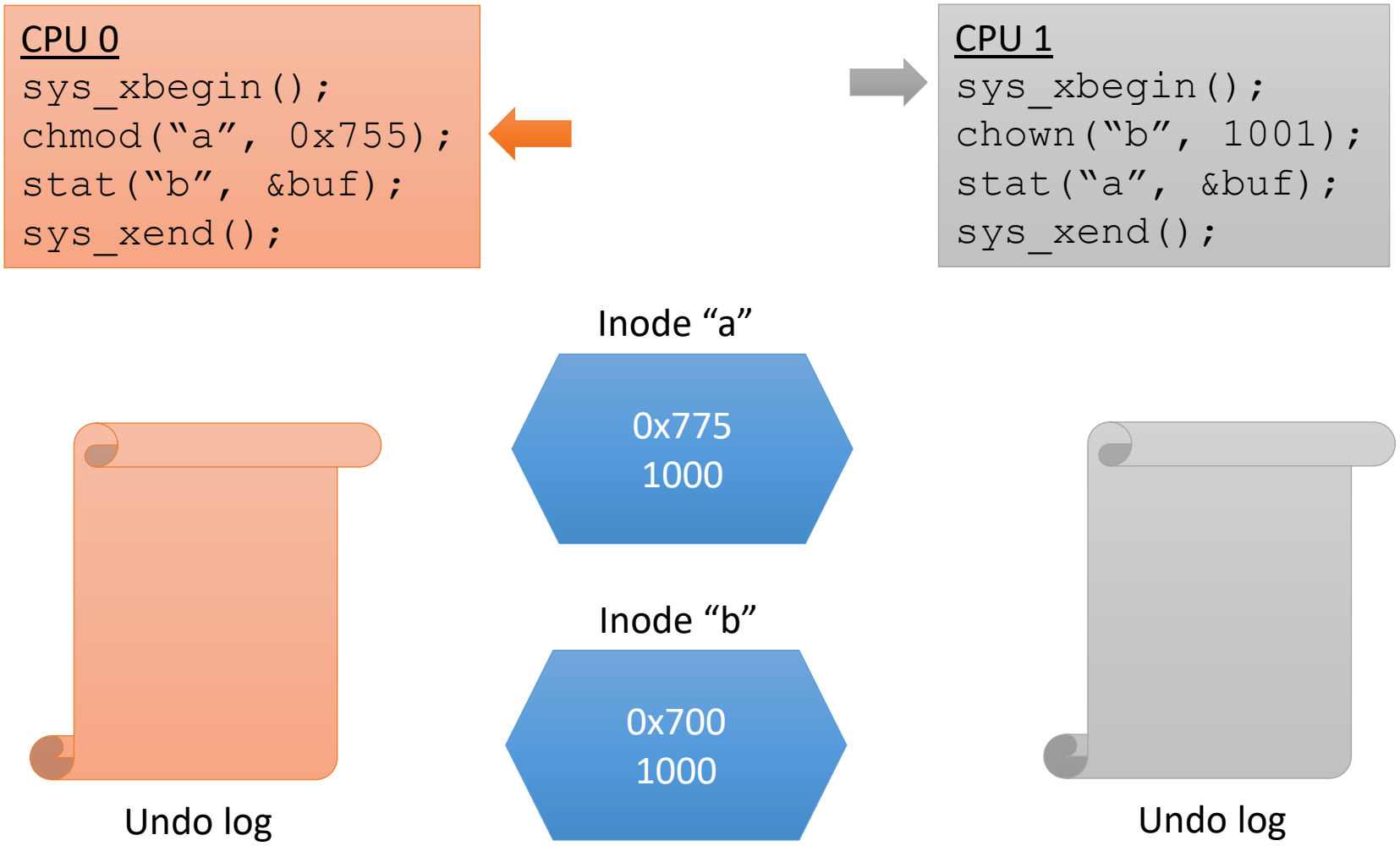
Traditional transaction design



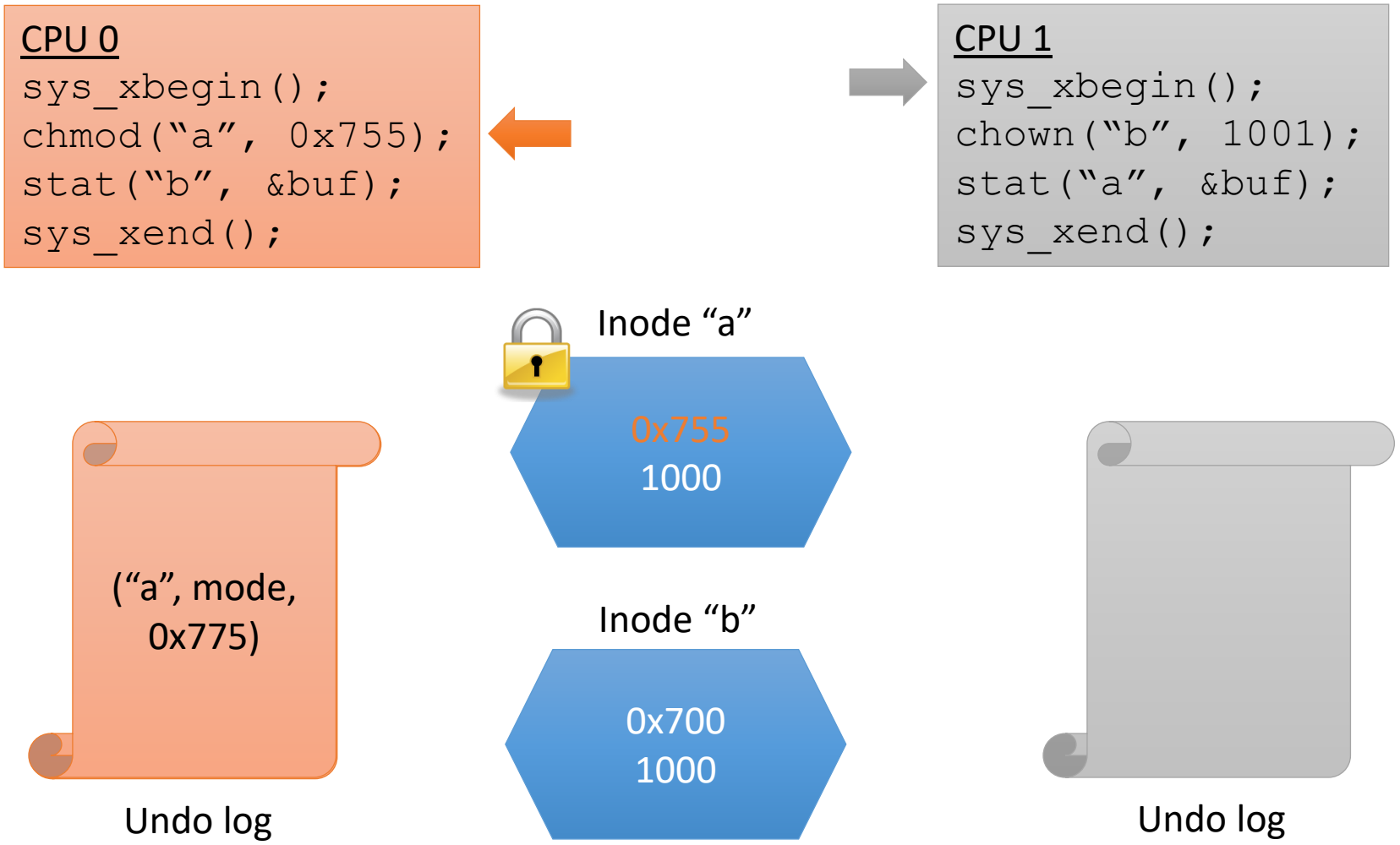
Traditional transaction design



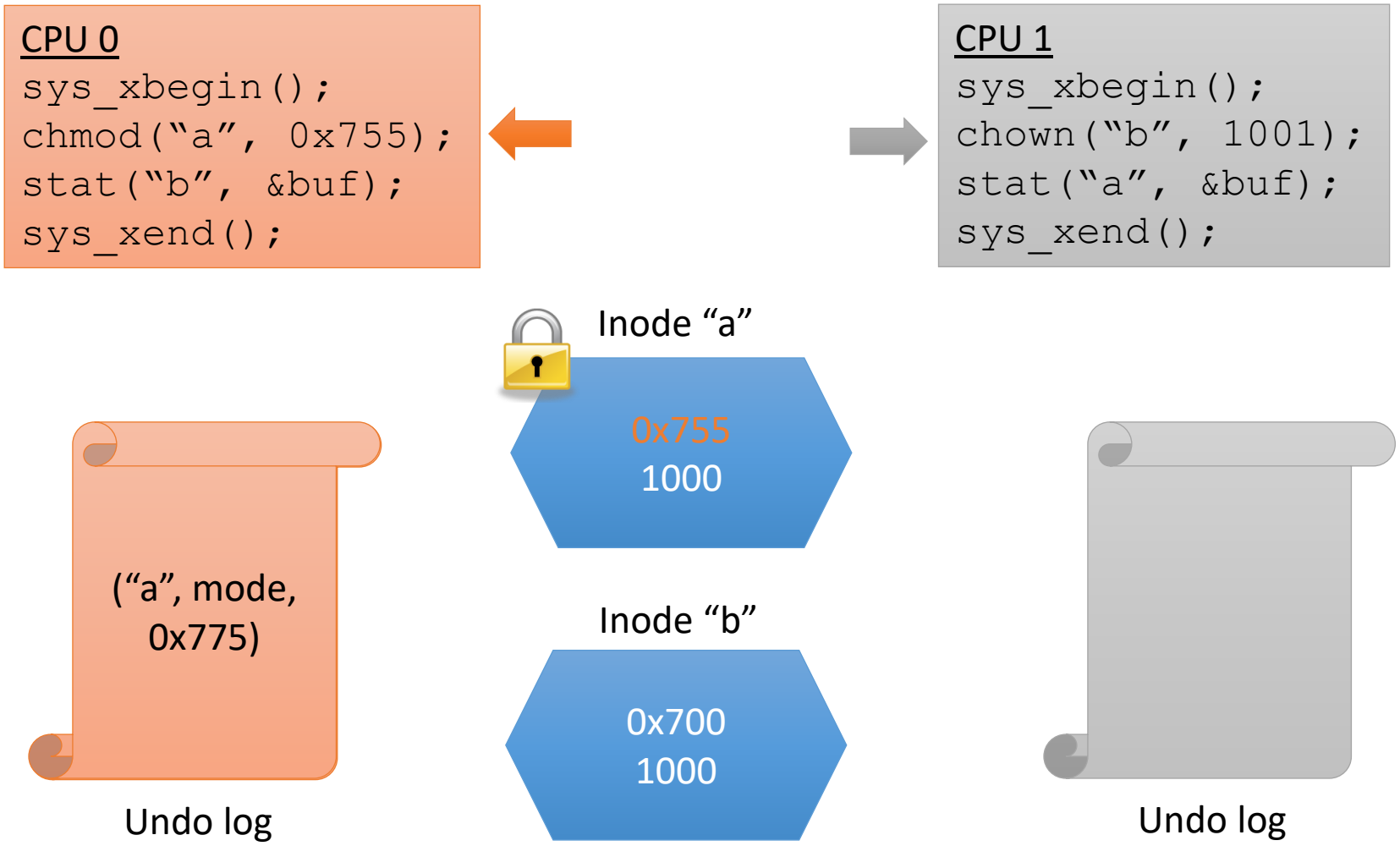
Traditional transaction design



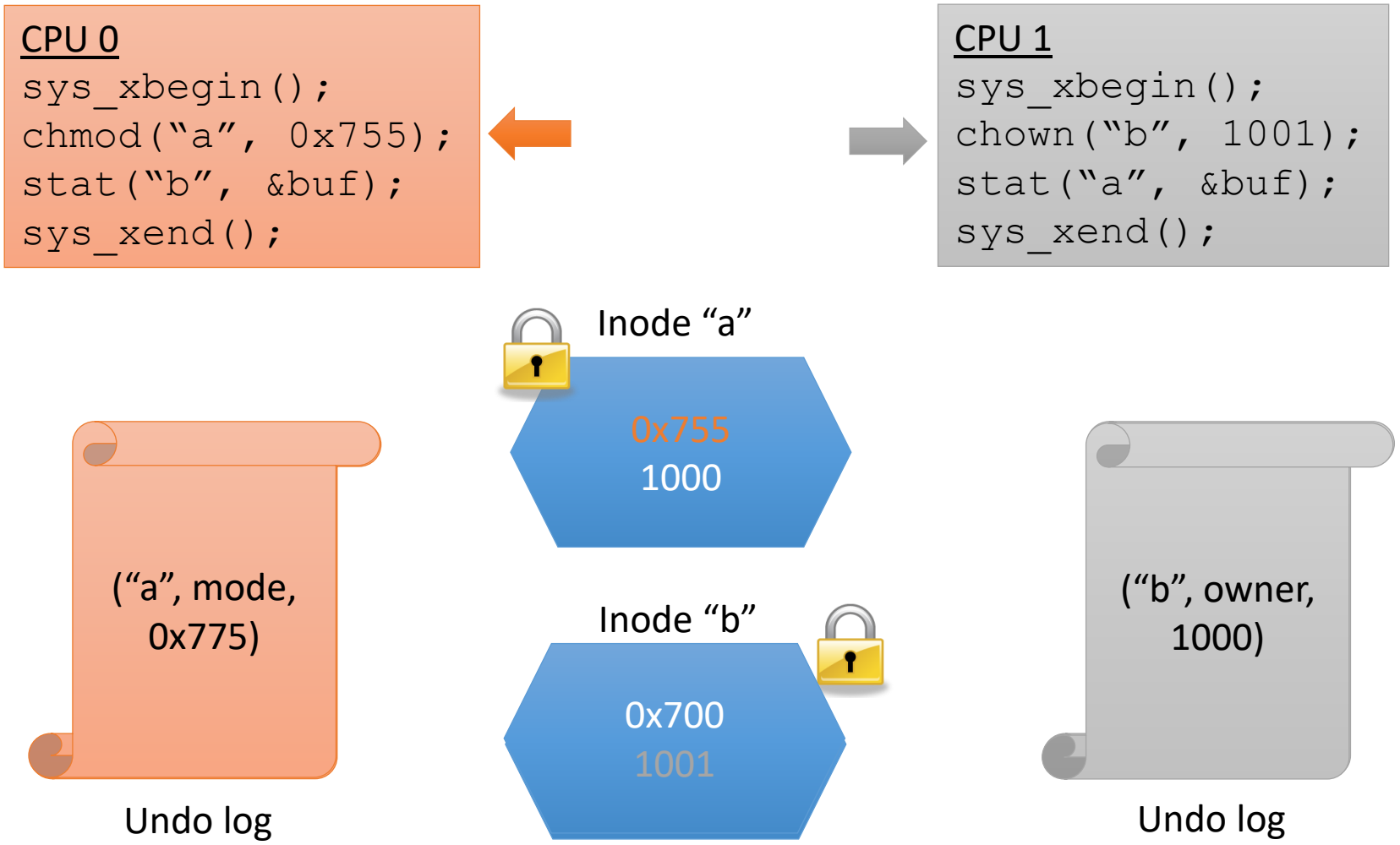
Traditional transaction design



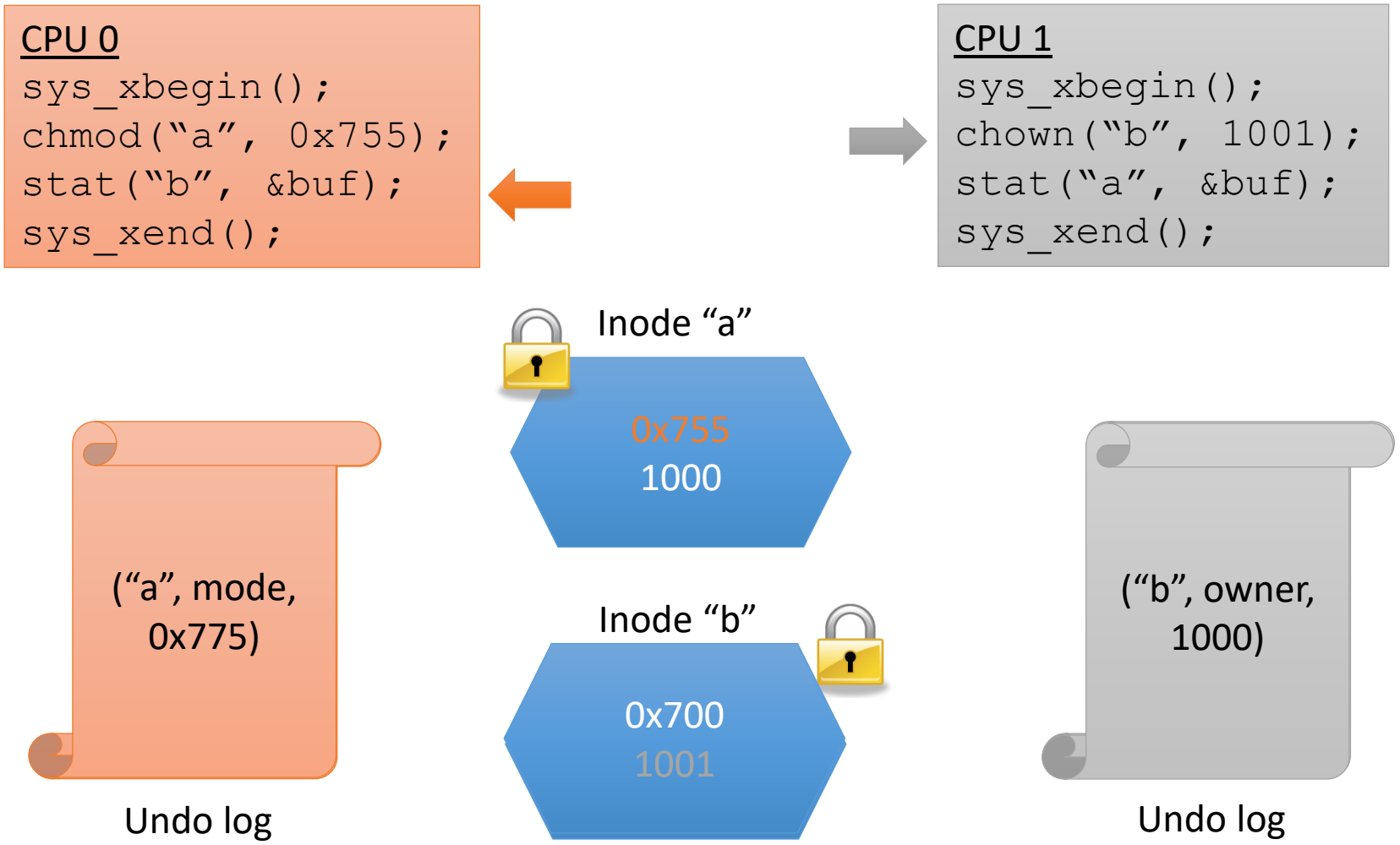
Traditional transaction design



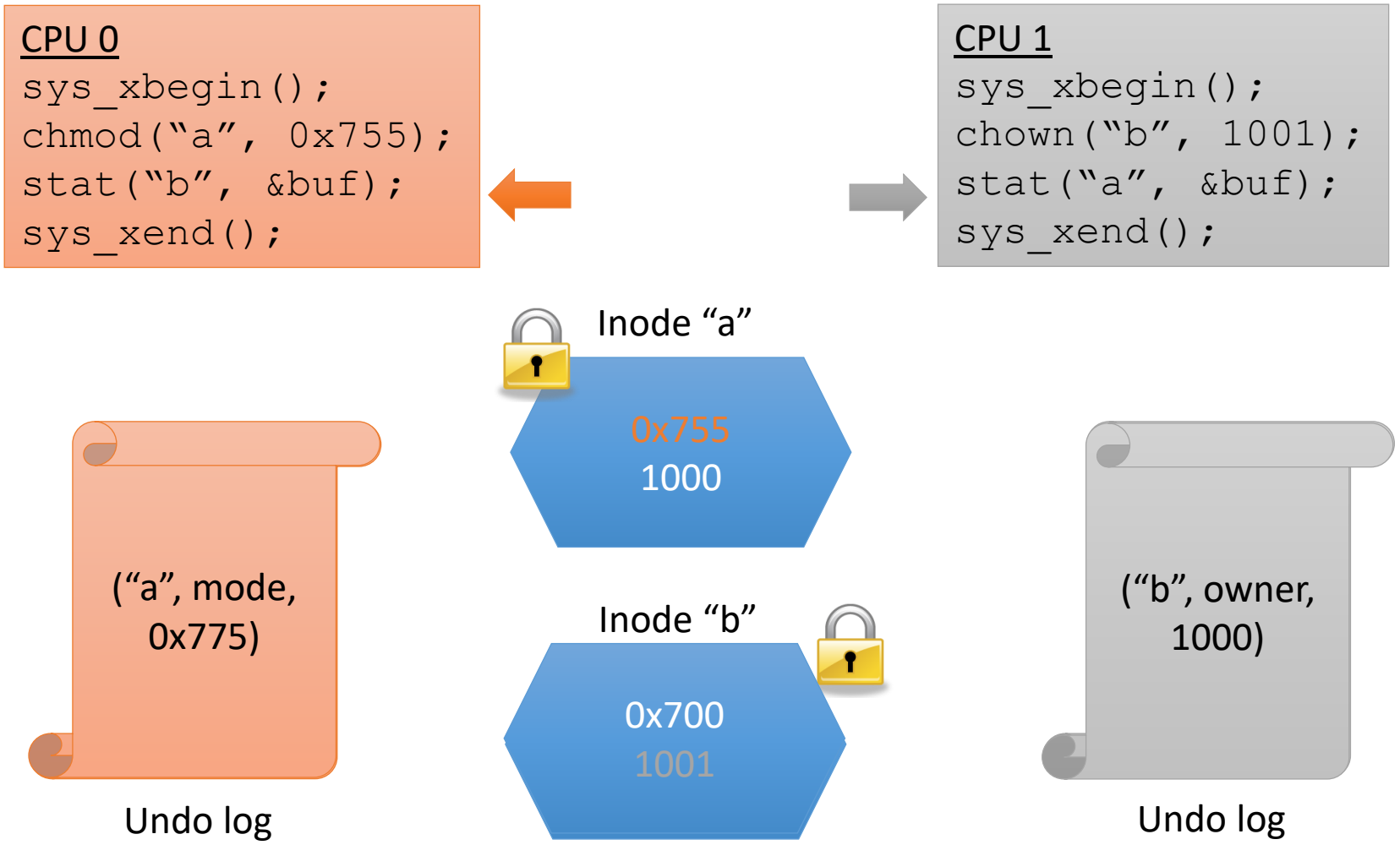
Traditional transaction design



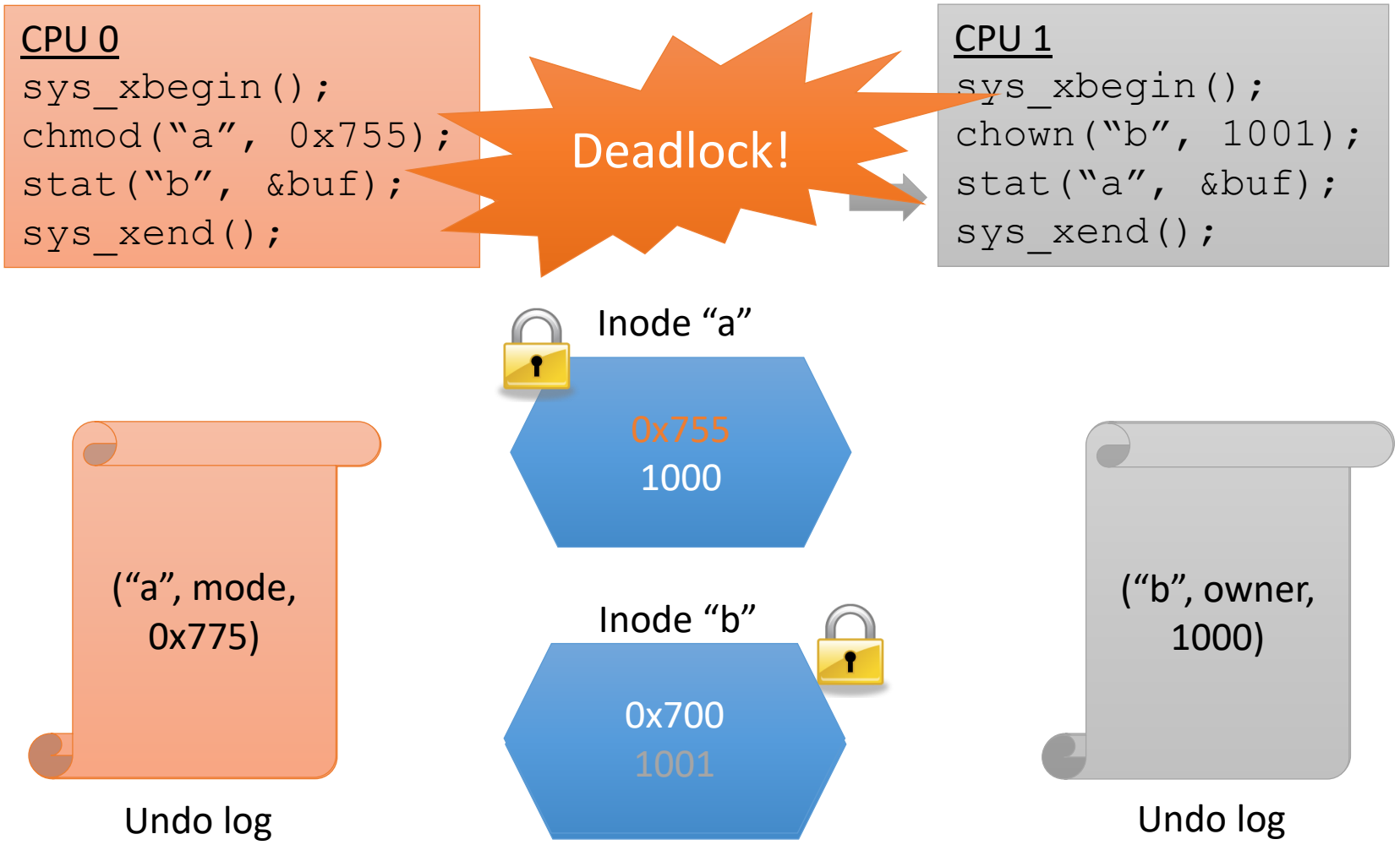
Traditional transaction design



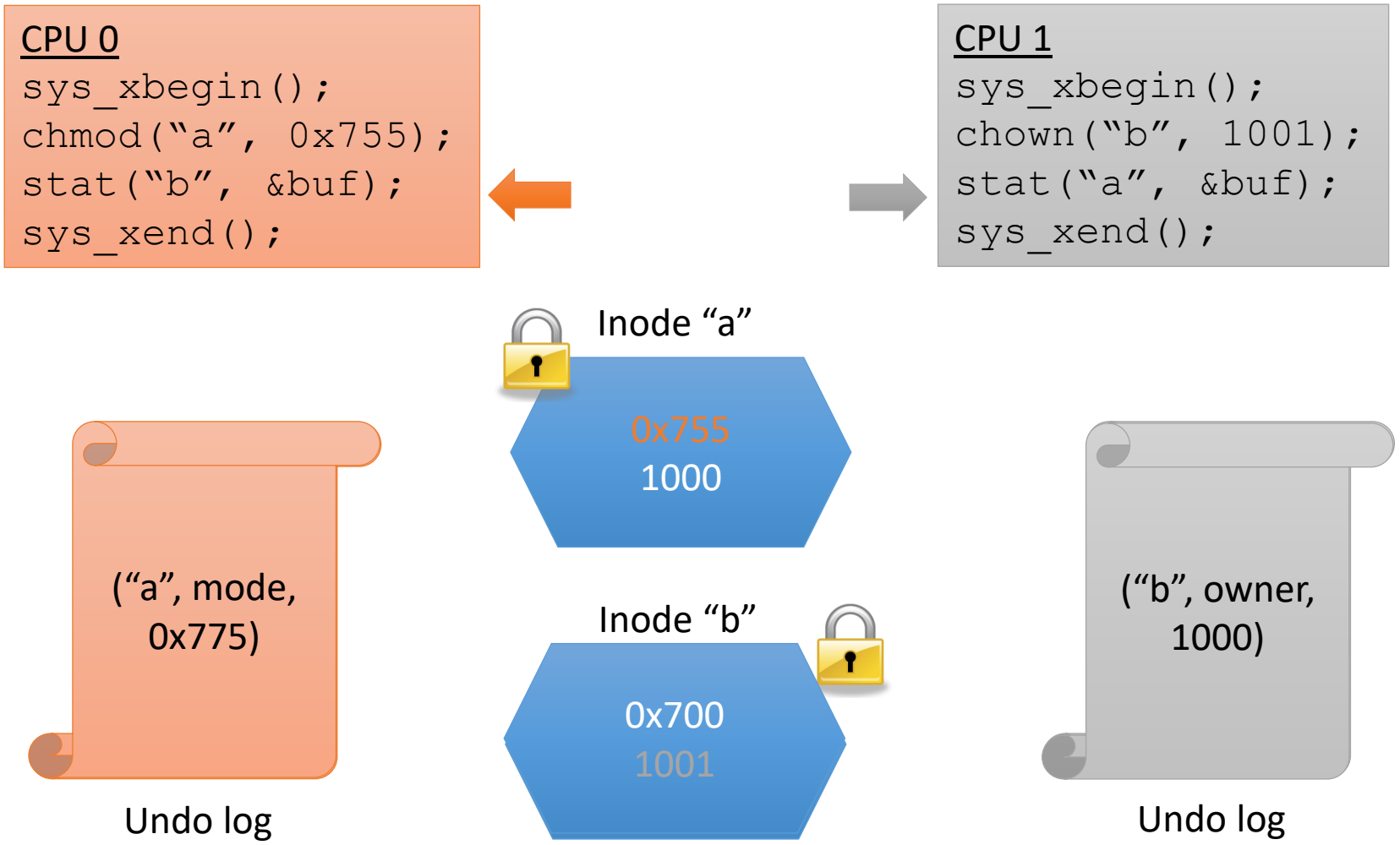
Traditional transaction design



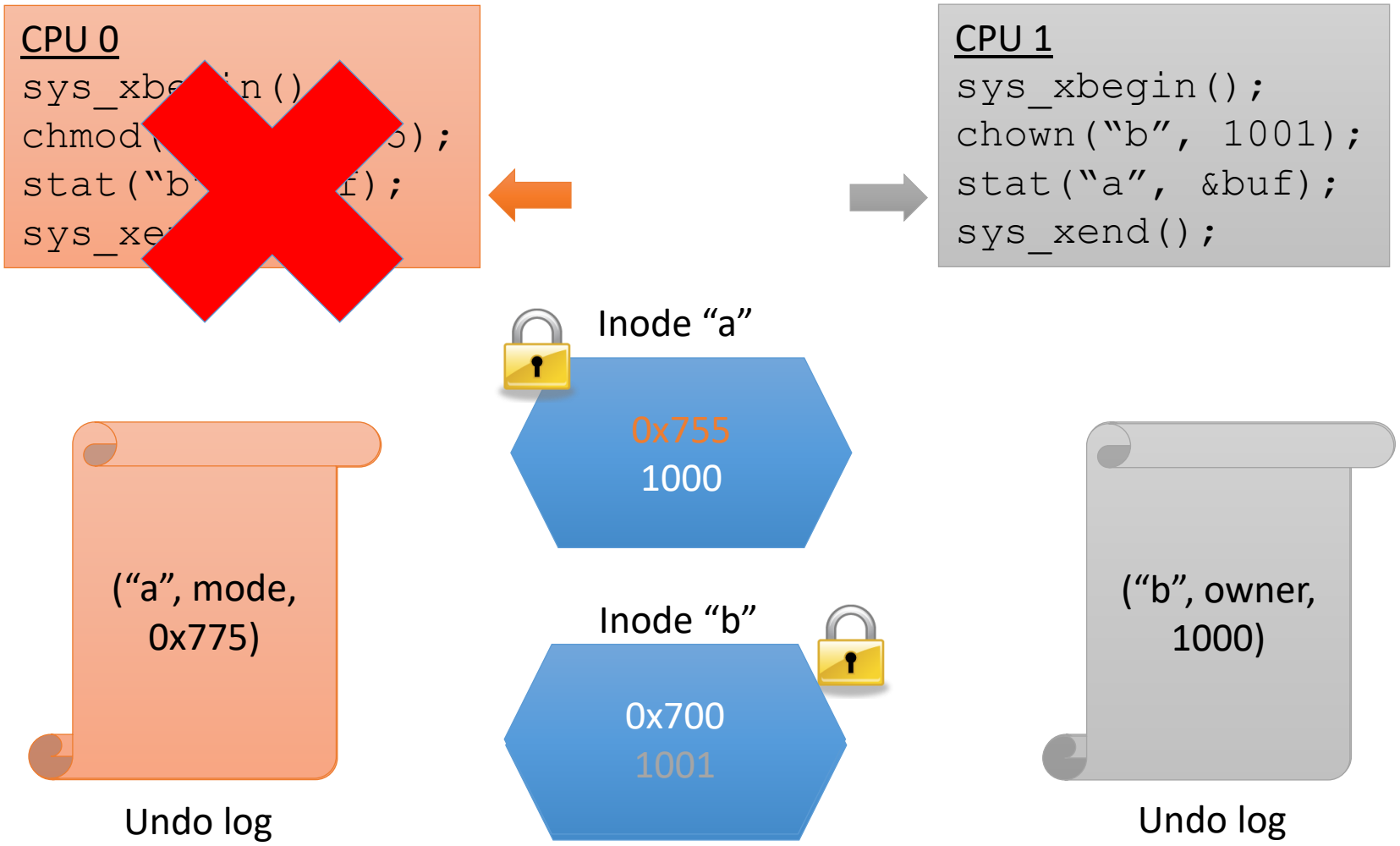
Traditional transaction design



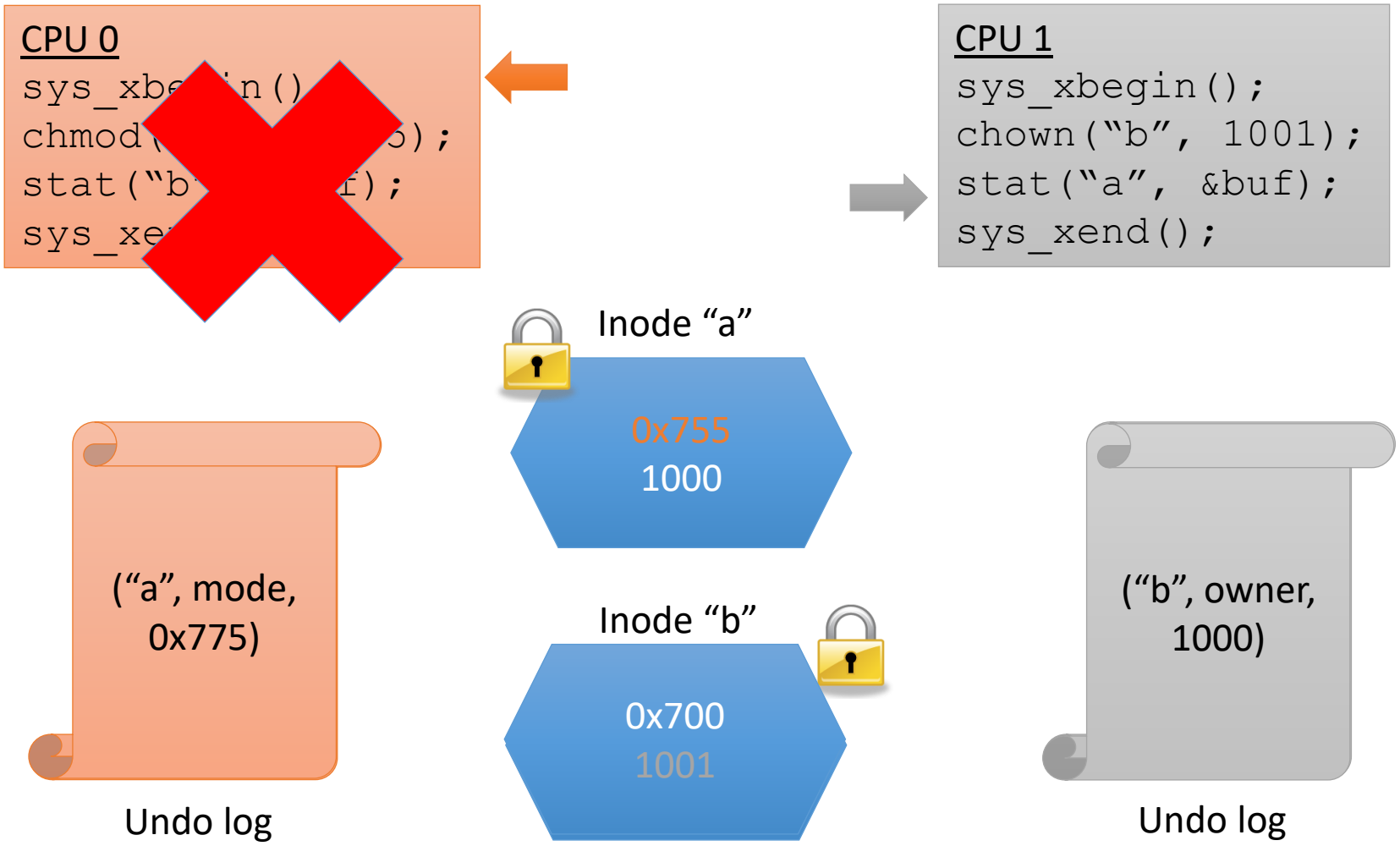
Traditional transaction design



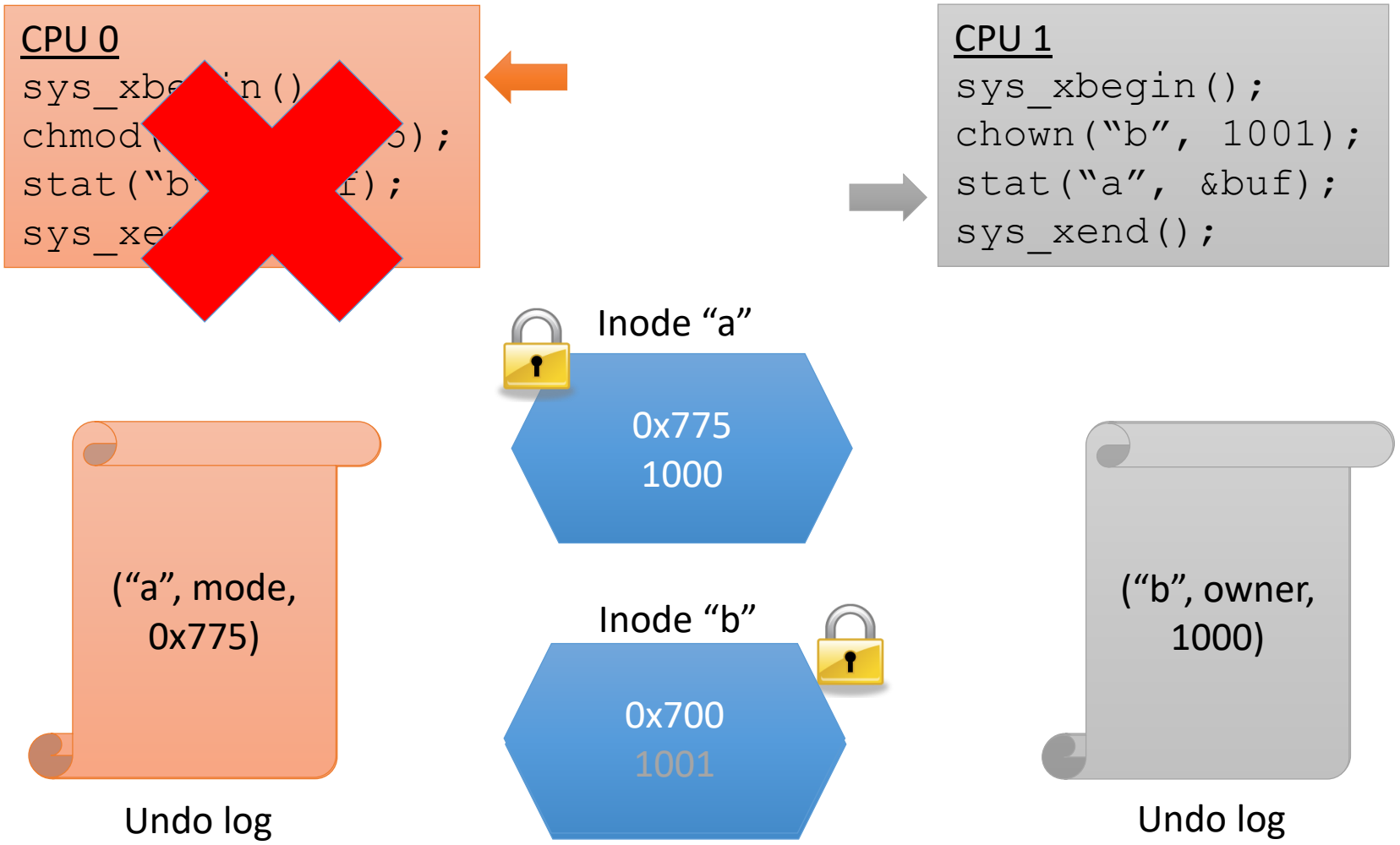
Traditional transaction design



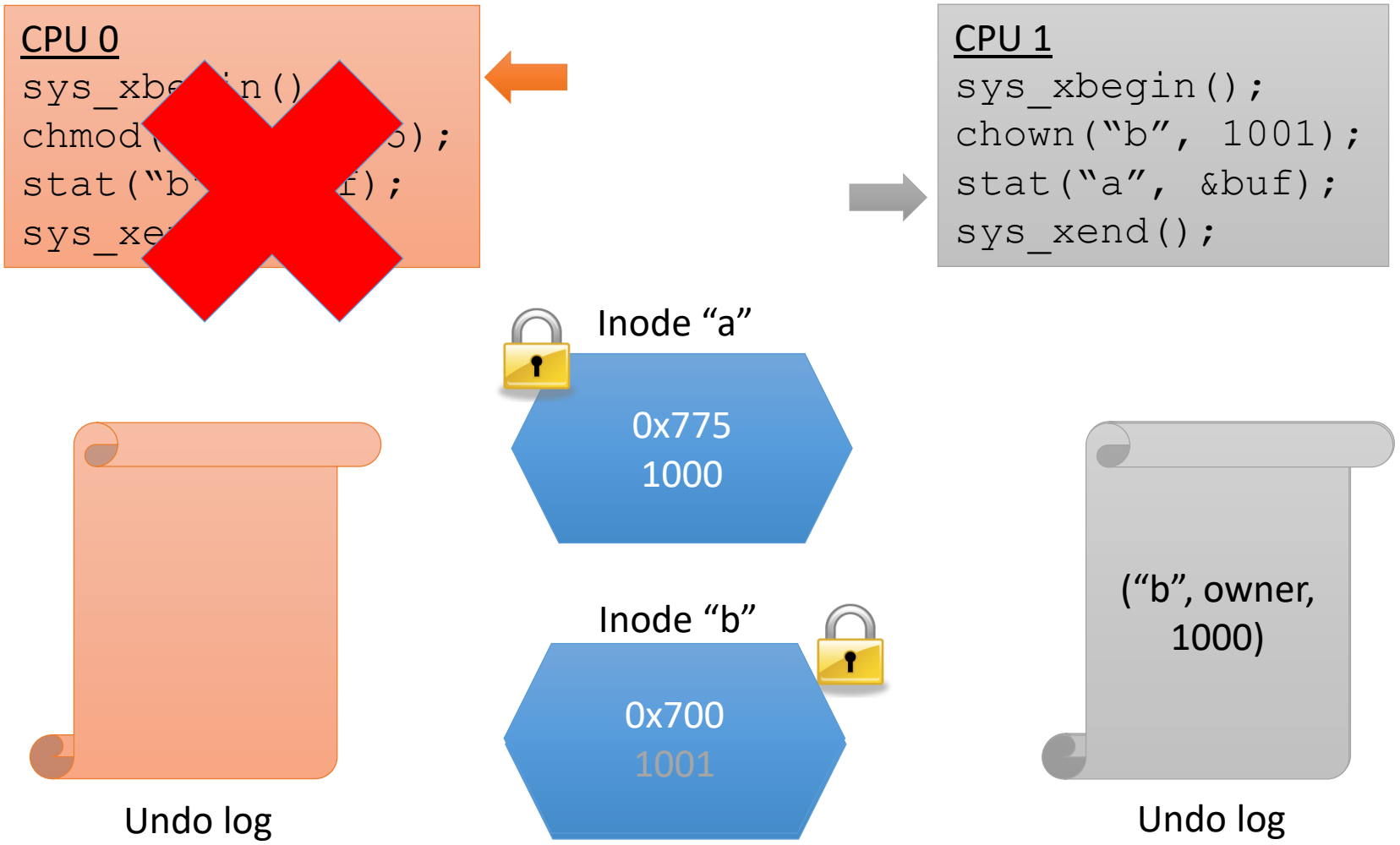
Traditional transaction design



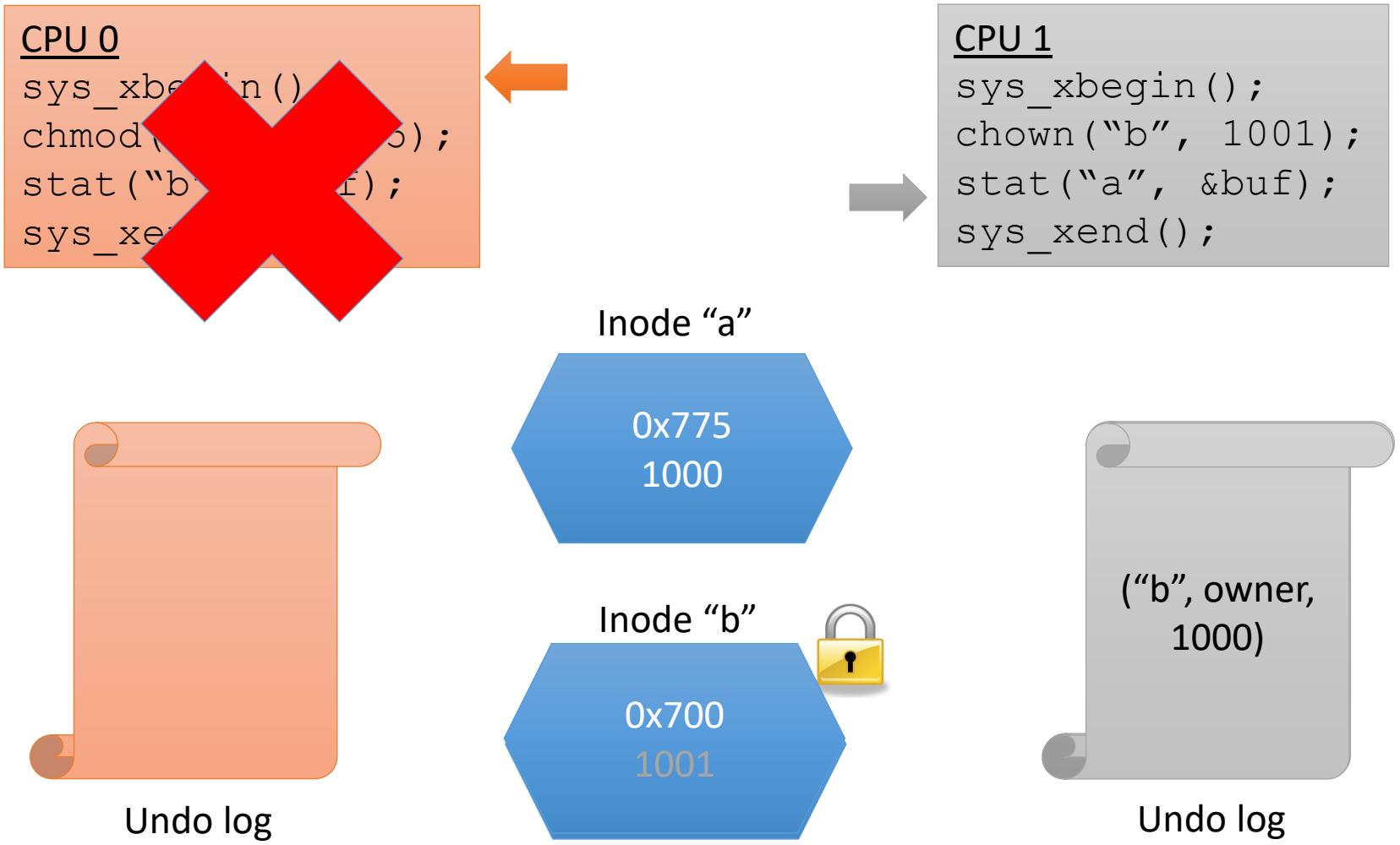
Traditional transaction design



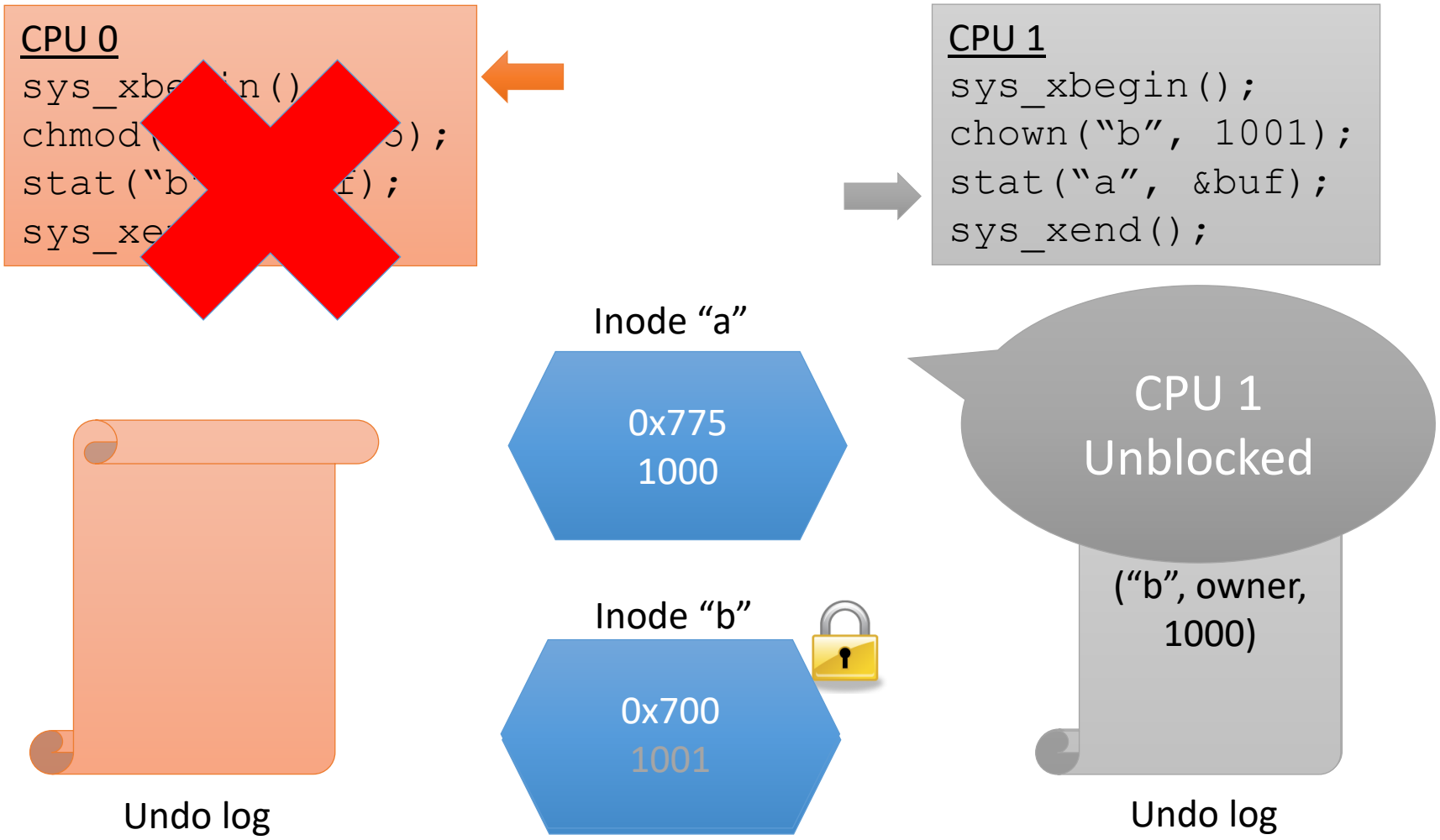
Traditional transaction design



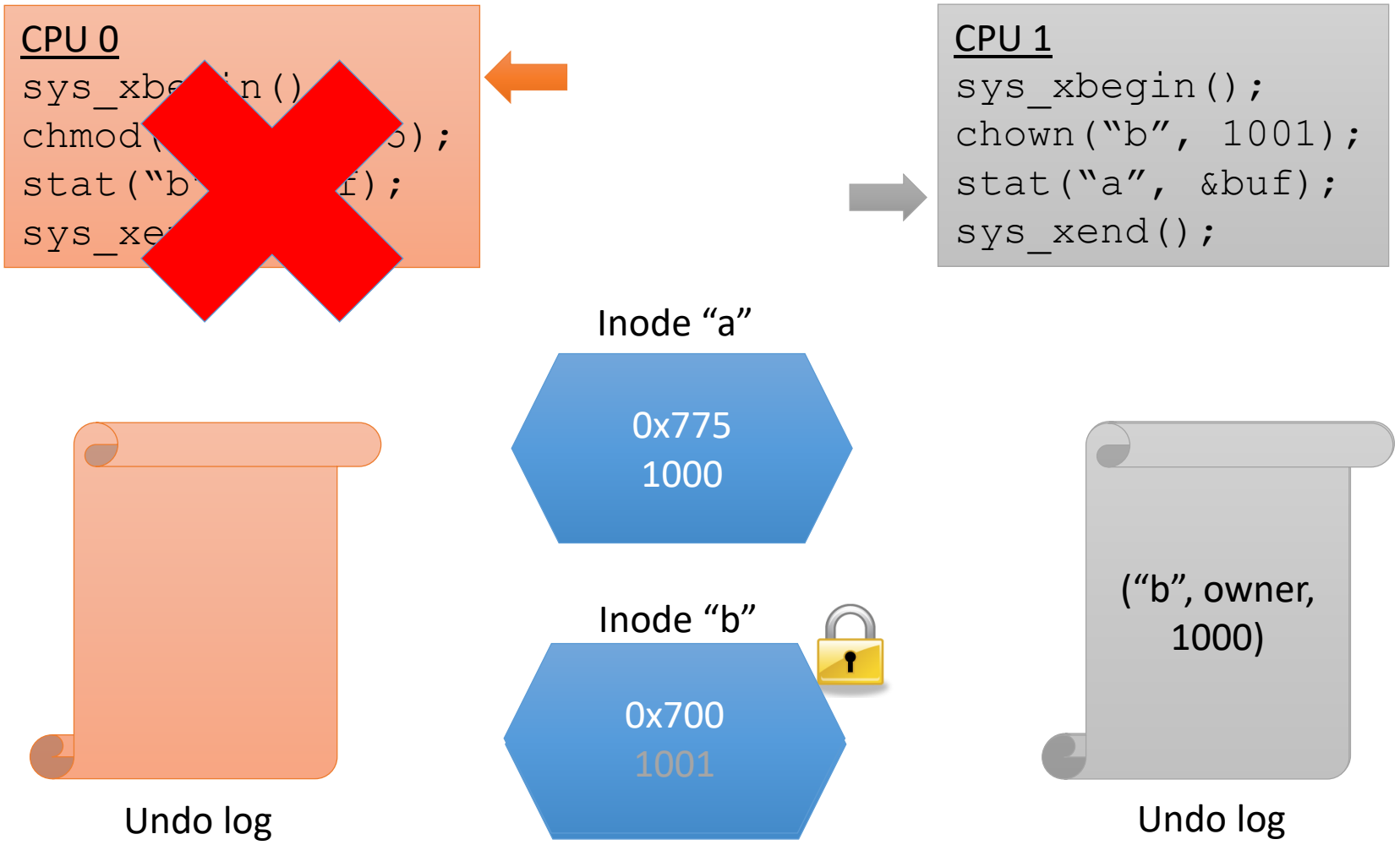
Traditional transaction design



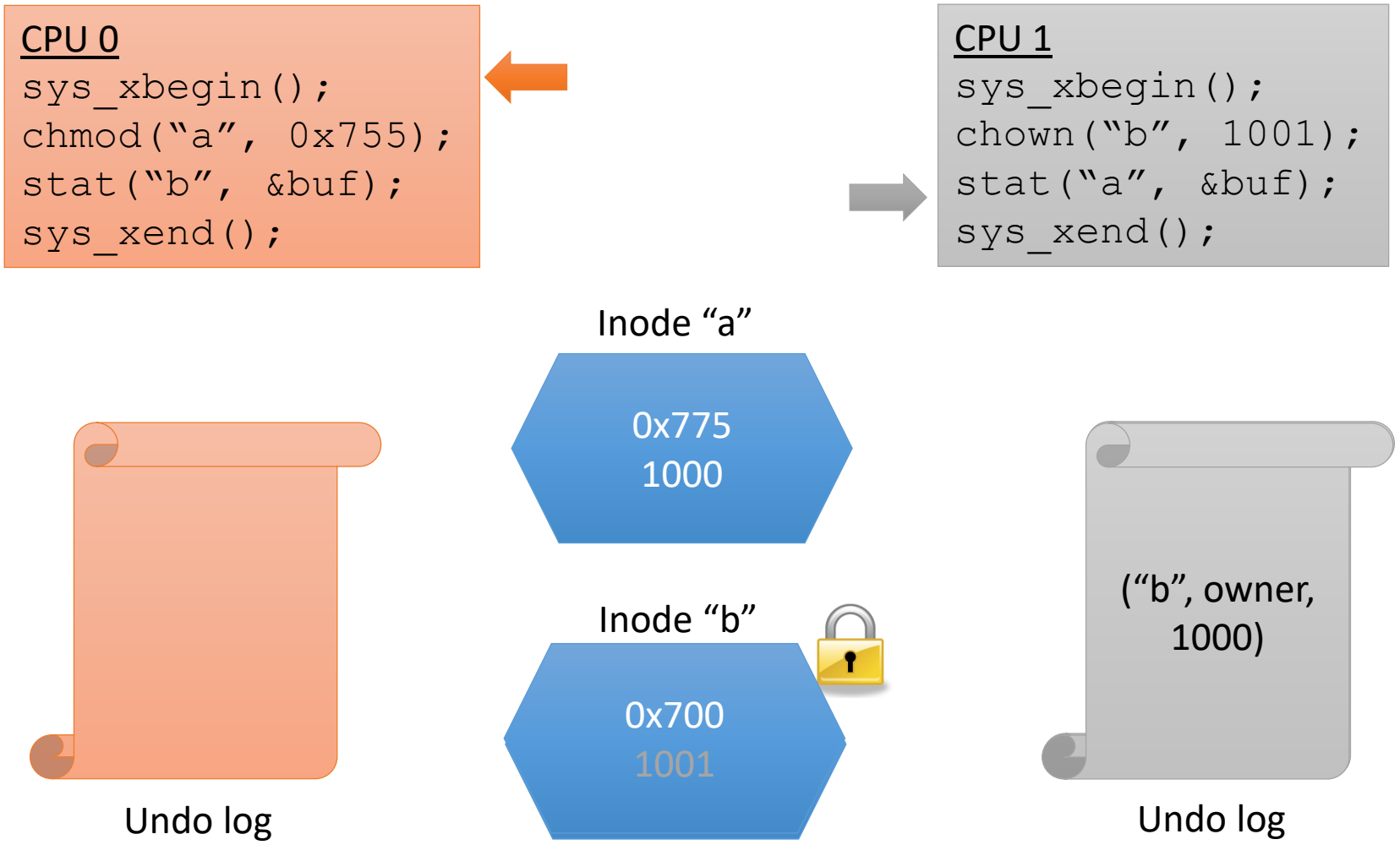
Traditional transaction design



Traditional transaction design



Traditional transaction design



TxOS design

- Inspired by optimistic concurrency control and object-based STM's
- Transactions operate on private copies of data
  - Commit by replacing previous stable version
  - Abort by discarding private copies
  - Similar to redo log, but more efficient
- Split objects into header and data
  - Data object contains fields which may be modified in tx
  - Header provides layer of indirection for pointers
  - Commit by replacing pointer in header to data object

Traditional transaction design

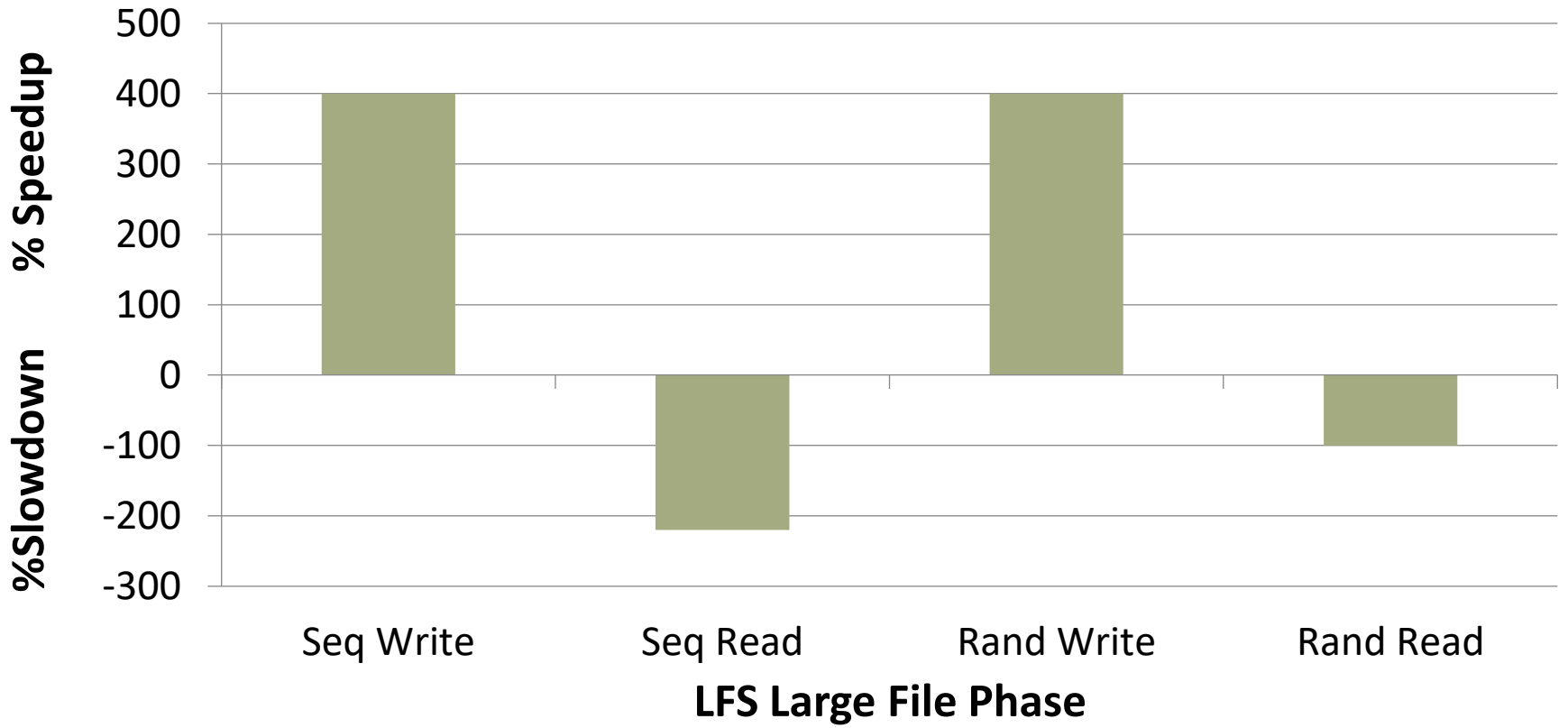
- Adopted by many databases, QuickSilver, LOCUS
- In-place updates and undo log
- Two-phase locking
  - Locks acquired as data accessed, held until commit

Object Splitting

- Solves incoming pointers problem
- Avoids conflicts
  - Kernel bookkeeping in header (e.g., reference counters)
  - Can support multiple data objects for unrelated data
- RCU allows transactions to read share data objects

Acceptable Performance

- 40% overhead for dpkg install



- Speedup compared to unmodified Linux

Example: browser plug-in upgrade

```
write new plug-in binary
```

```
exec post-install script
   (updates browser config)
```

Example: browser plug-in upgrade

```
write new plug-in binary
start browser, old config,
     old plug-in arguments

exec post-install script
   (updates browser config)
```

Example: browser plug-in upgrade

```
write new plug-in binary
start browser, old config,
    old plug-in arguments
   corrupt data files
exec post-install script
   (updates browser config)
```

Example: browser plug-in upgrade

```
write new plug-in binary
start browser, old config,
    old plug-in arguments
   corrupt data files
exec post-install script
   (updates browser config)
```

- API can't ensure consistent updates to OS resources
- Concurrency and crashes cause subtle inconsistencies

Transactions solve important problems

Transactions solve important problems

- Applications

Transactions solve important problems

- Applications
  - Replace databases for simple synchronization

Transactions solve important problems

- Applications
  - Replace databases for simple synchronization
  - Support system calls in transactional memory apps

Transactions solve important problems

- Applications
  - Replace databases for simple synchronization
  - Support system calls in transactional memory apps
  - Tolerate faults in untrusted software modules

Transactions solve important problems

- Applications
  - Replace databases for simple synchronization
  - Support system calls in transactional memory apps
  - Tolerate faults in untrusted software modules
  - Atomically update file contents and access control list

Transactions solve important problems

- Applications
  - Replace databases for simple synchronization
  - Support system calls in transactional memory apps
  - Tolerate faults in untrusted software modules
  - Atomically update file contents and access control list
- Easier to write OS extensions

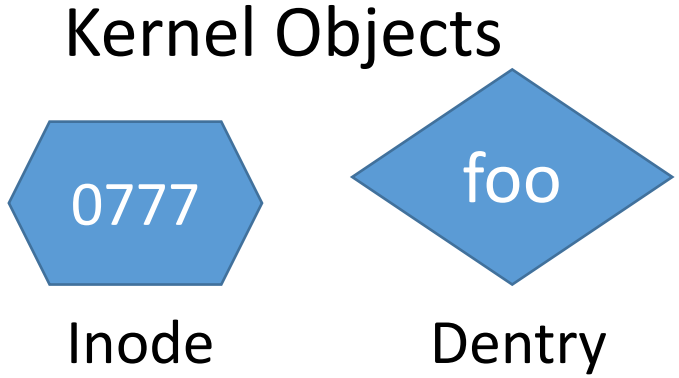
Transactions solve important problems

- Applications
  - Replace databases for simple synchronization
  - Support system calls in transactional memory apps
  - Tolerate faults in untrusted software modules
  - Atomically update file contents and access control list
- Easier to write OS extensions
  - System Tx + Journal = Tx Filesystem

Related Systems

- Similar interface, different implementation
  - QuickSilver [SOSP ‘91], TABS [SOSP ‘85]
    - Weaker guarantees
  - TxF, Valor [FAST ‘09]
    - Only file system transactions
- Different interface, similar implementation
  - Speculator [SOSP ’05, OSDI ‘06]
- Terms “transaction” and “OS” appear in paper title
  - TxLinux [SOSP ’07, ASPLOS ‘09]

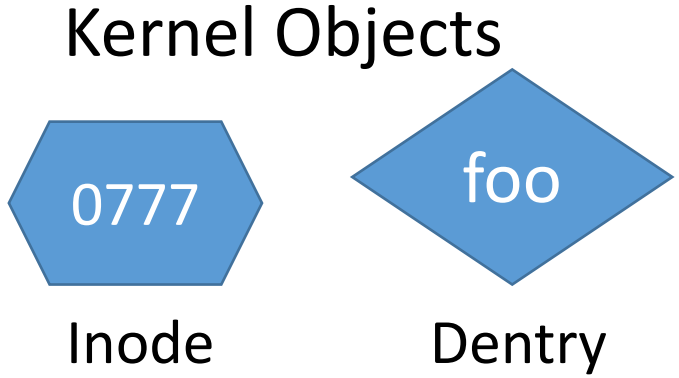
Lazy Version Management - Restart




```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```

Transaction Working Set

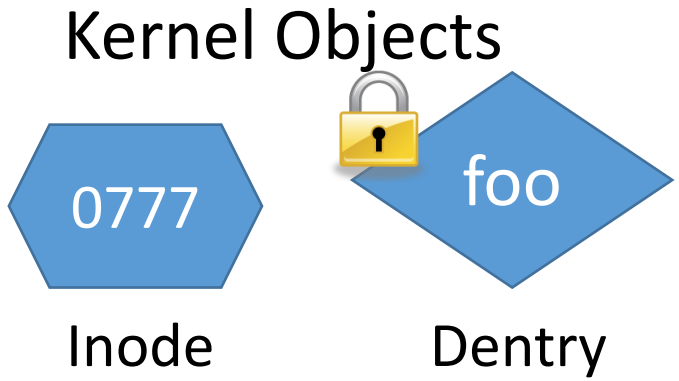
Lazy Version Management - Restart




```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```

Transaction Working Set

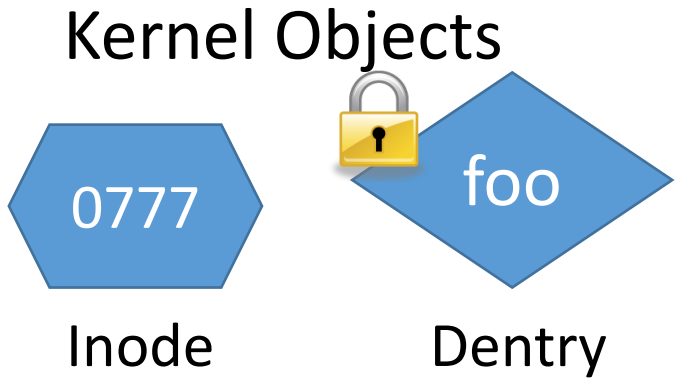
Lazy Version Management - Restart



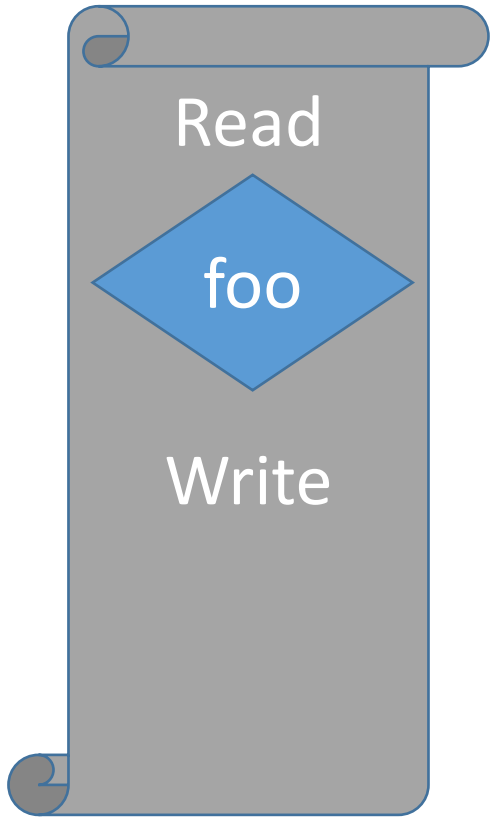

```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```

Transaction Working Set

Lazy Version Management - Restart

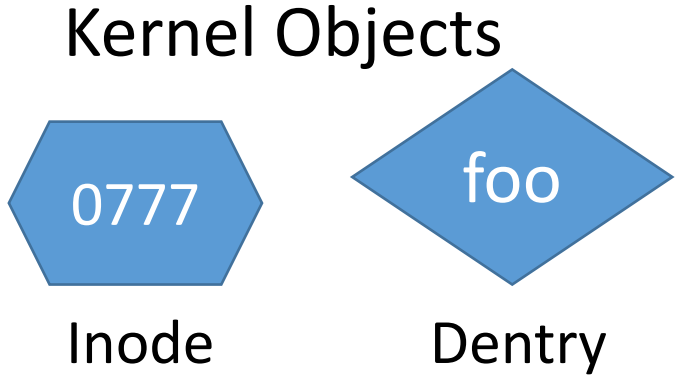



```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```

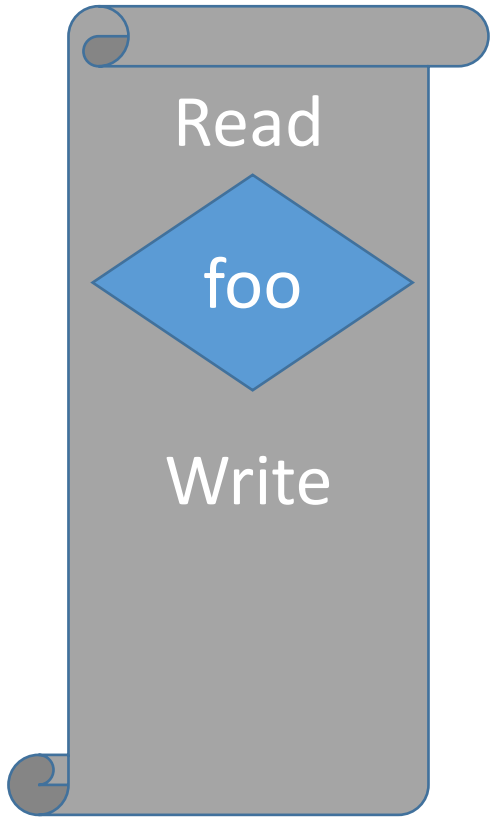


Transaction Working Set

Lazy Version Management - Restart

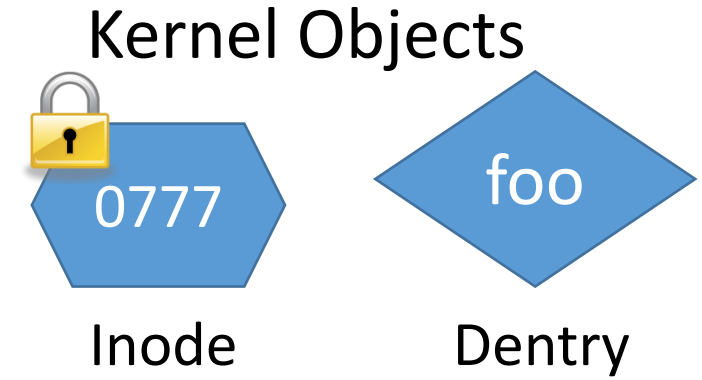



```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```

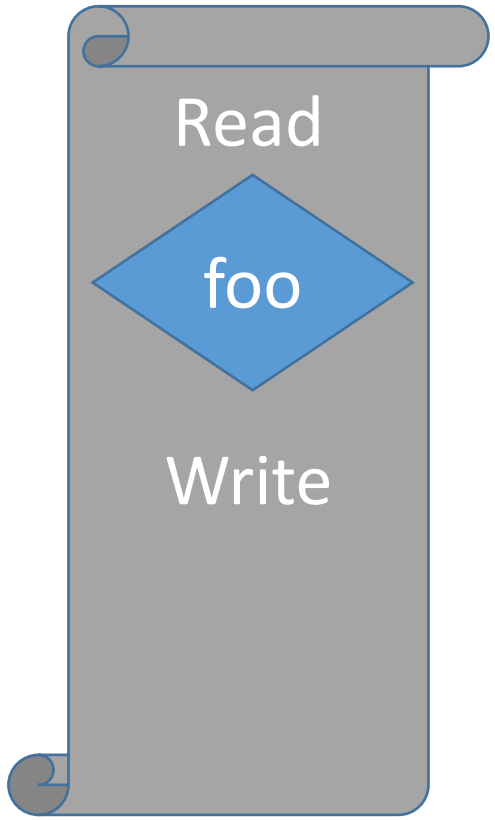


Transaction Working Set

Lazy Version Management - Restart

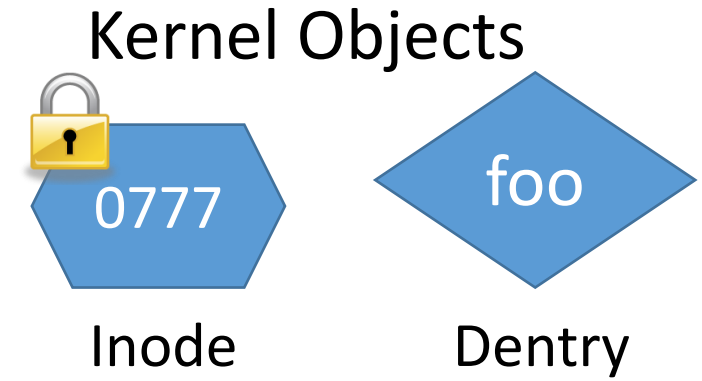



```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```



Transaction Working Set

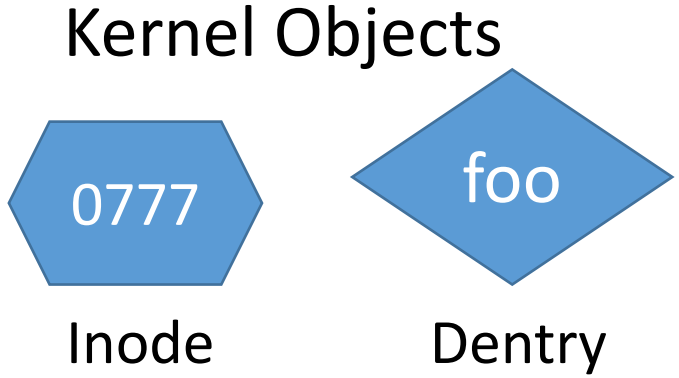
Lazy Version Management - Restart




```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```

Transaction Working Set

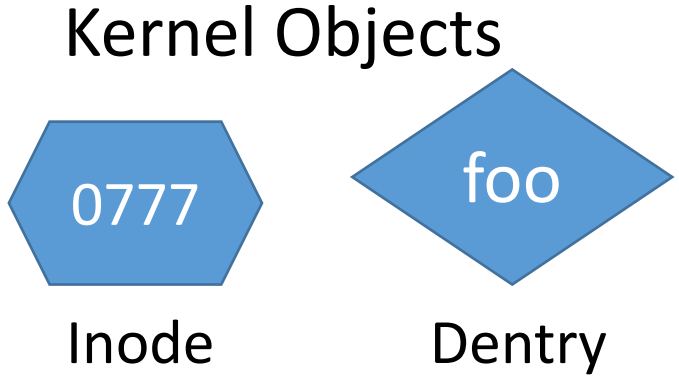
Lazy Version Management - Restart




```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```

Transaction Working Set

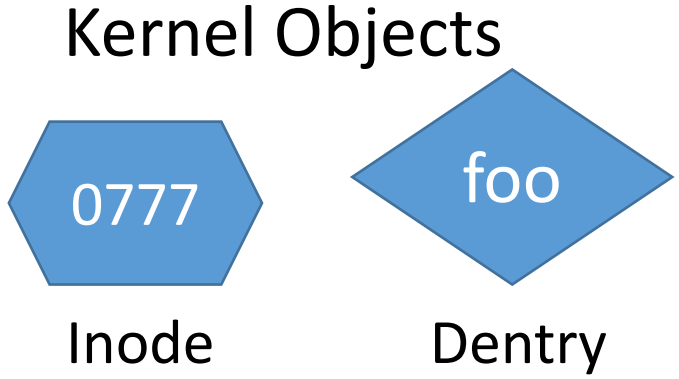
Lazy Version Management - Restart




```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```

Transaction Working Set

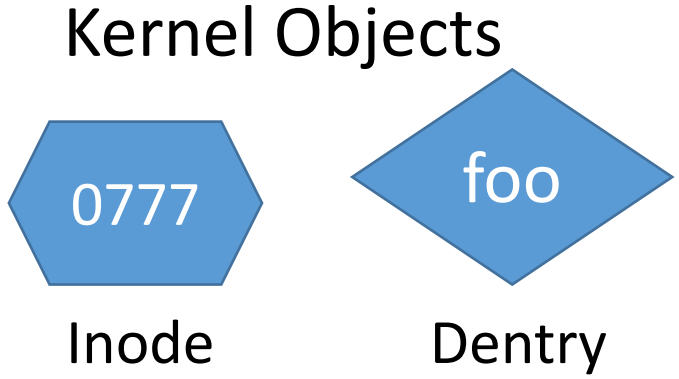
Lazy Version Management - Restart




```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```

Transaction Working Set

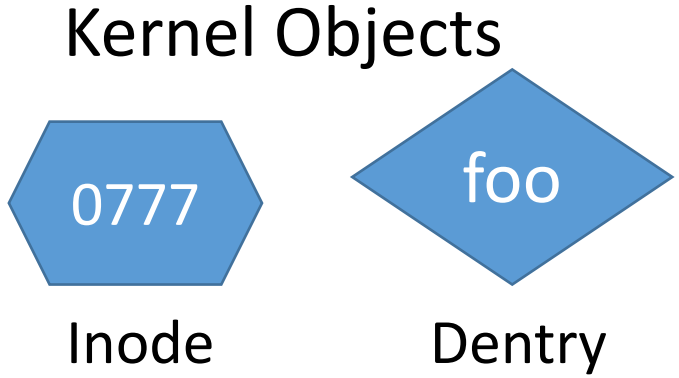
Lazy Version Management - Restart




```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```

Transaction Working Set

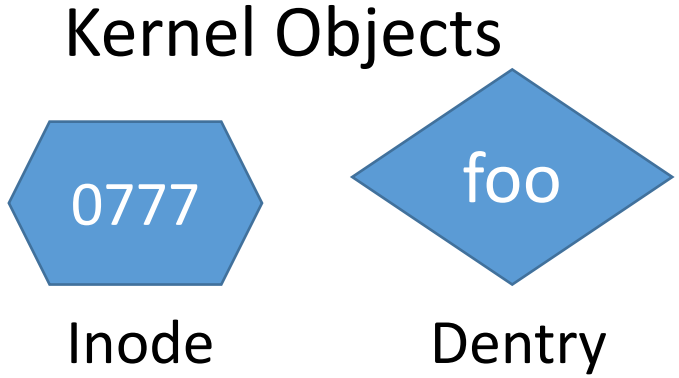
Lazy Version Management - Restart




```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```

Transaction Working Set

Lazy Version Management - Restart




```
sys_xbegin();
chmod("foo", 0755);
…
sys_xend();
```

Transaction Working Set

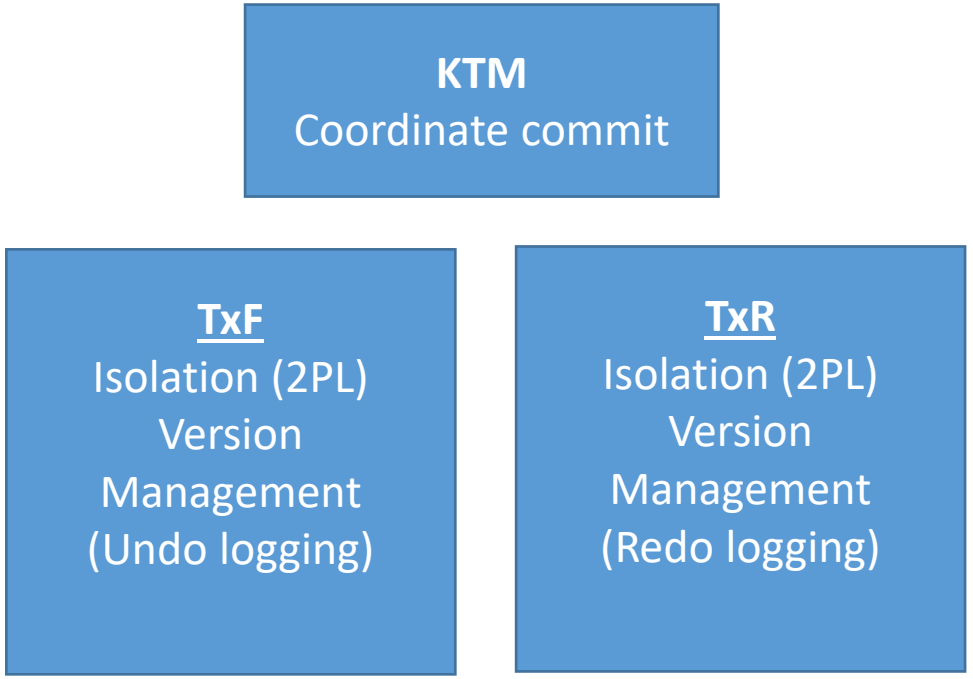
Related work: Transactional FSes

- Provide transactional interface to storage
  - Solves important problems (limited SW install)
  - Adopted by Windows Vista
- Limited by lack of integration with rest of system
  - Ex: Can’t execute a transactionally written binary
- TxOS makes transactions a first class abstraction

Transactions in the OS

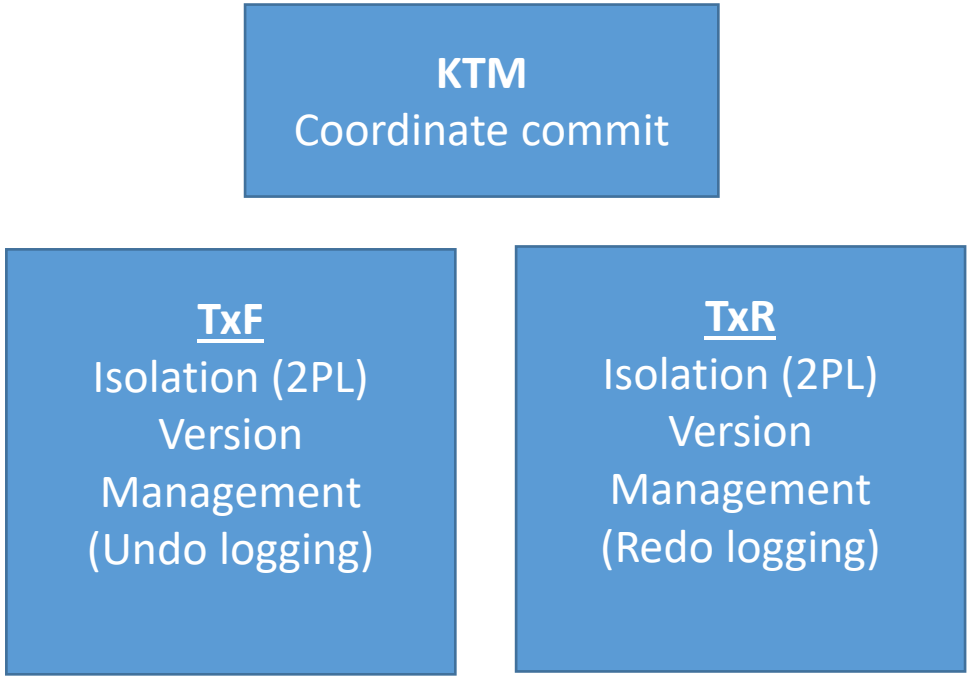
- Transactions work great in databases
  - Much potential for general purpose programming
- 2 key areas of related work
  - Transactional operating systems
  - Transactional file systems

Windows transaction architecture

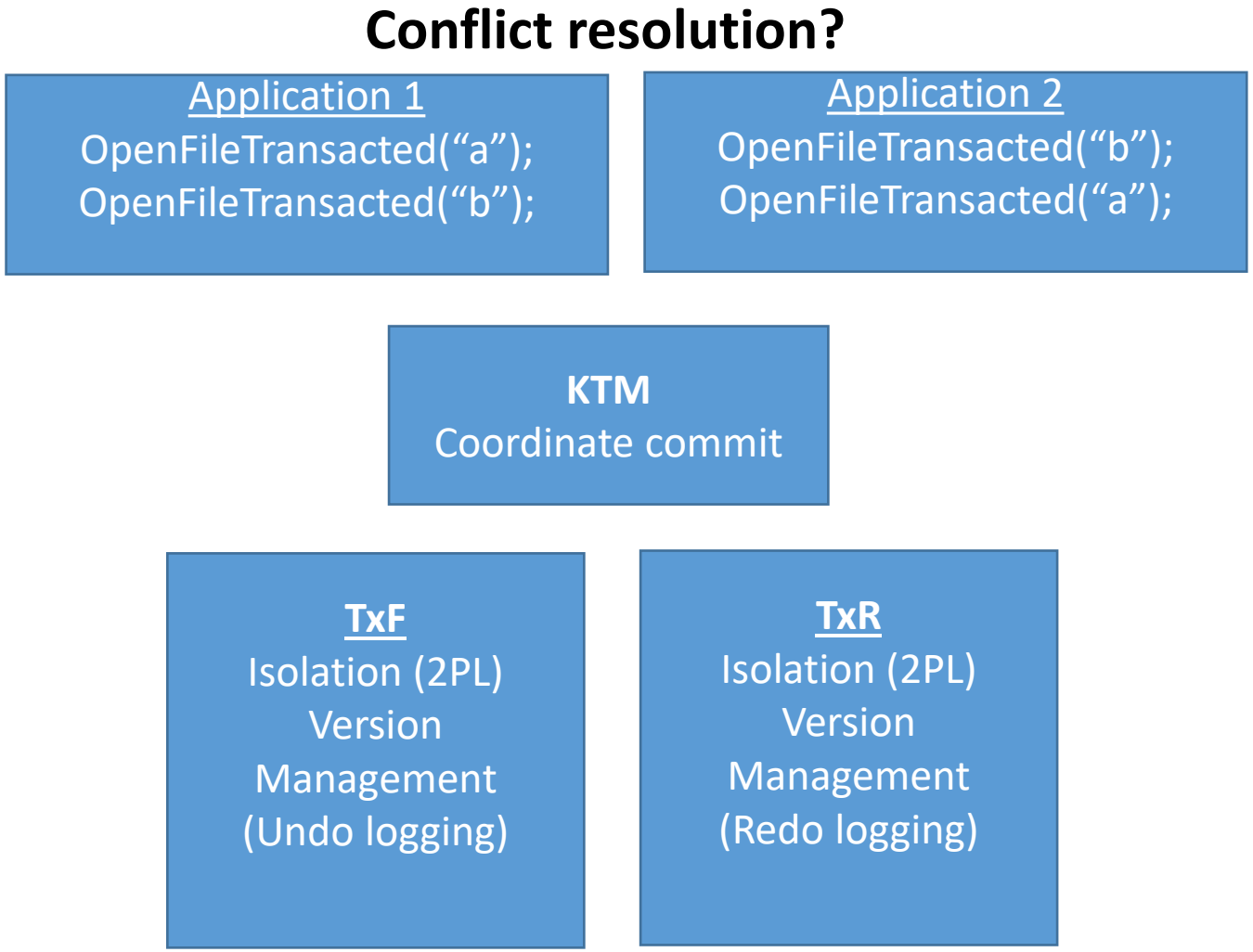


Windows transaction architecture

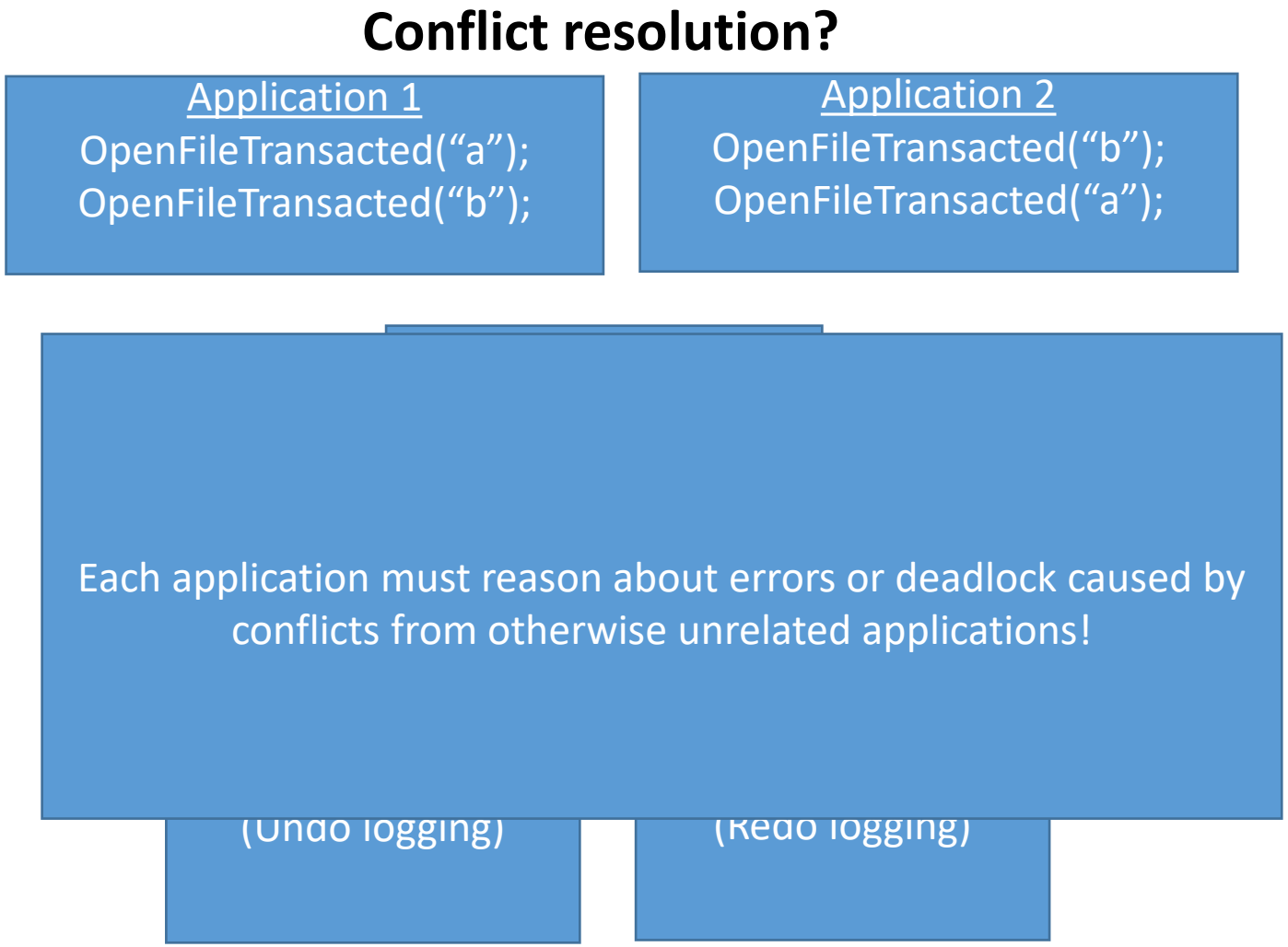
Conflict resolution?



Windows transaction architecture



Windows transaction architecture

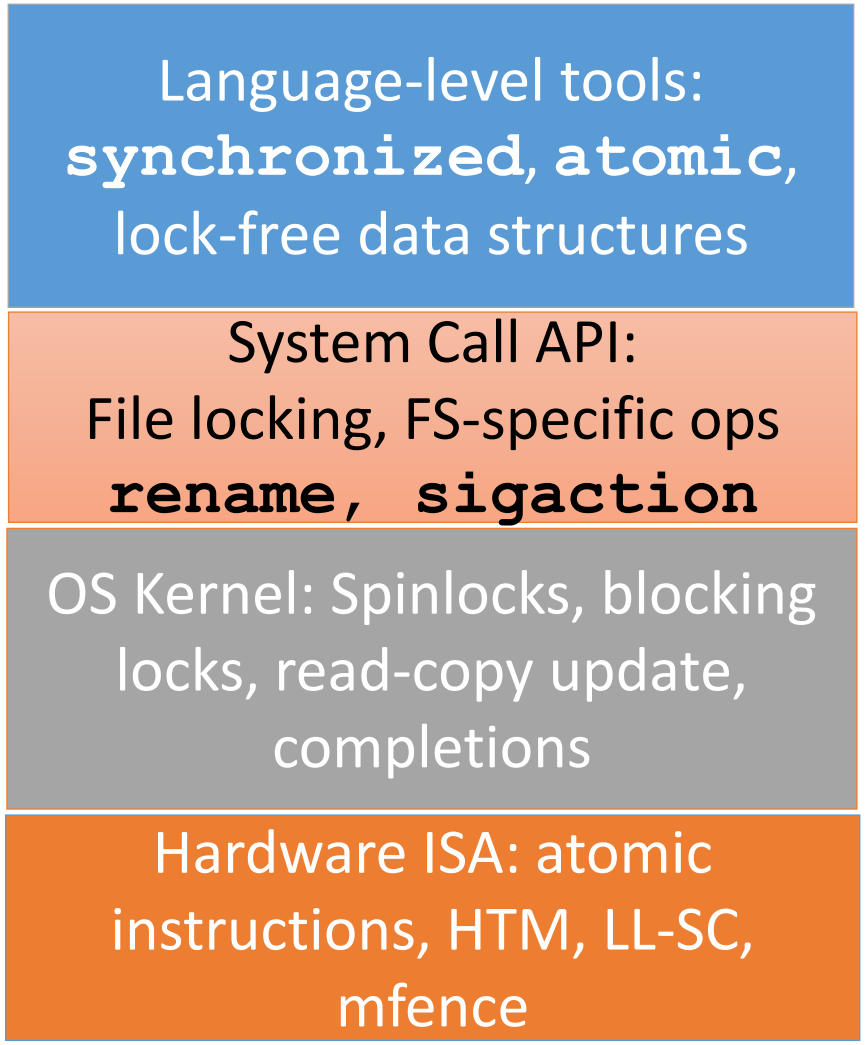


Transactional file systems

- Important step forward and enabling technology
- Not complete
  - E.g., many unsupported resources
- Not simple
  - E.g., must handle deadlock with unrelated applications

- Insight: Transactions must be a first-class OS abstraction

OS APIs don’t handle concurrency



- Most levels of stack provide complete, if not rich API
- Can’t make consistent updates to system resources across multiple system calls
- Race conditions for OS-managed resources with no simple work-around

Ex 2: transactional software install

```
sys_xbegin();
apt-get upgrade
sys_xend();
```

- A failed install is automatically rolled back
  - Rolls back volatile system state (new processes, signals)
  - Concurrent, unrelated operations are unaffected
- System crash: reboot to entire upgrade or none
- Transactional execution of utilities written during install