µ-Kernel Construction (9)

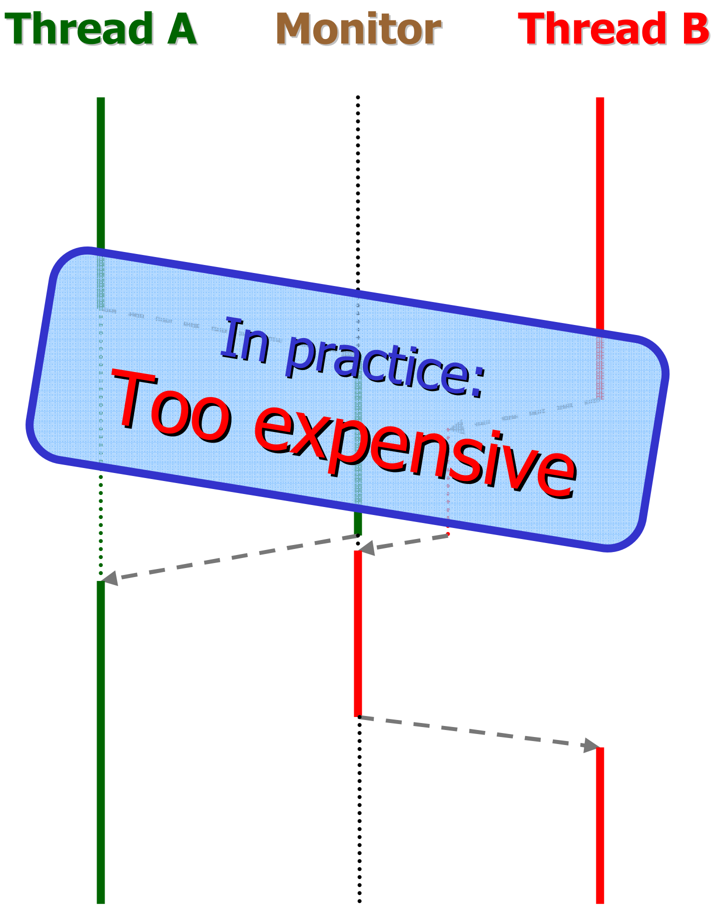
Local IPC

Optimization for Multi-Threaded Applications

Synchronization via IPC

Thread A **Monitor** **Thread B**

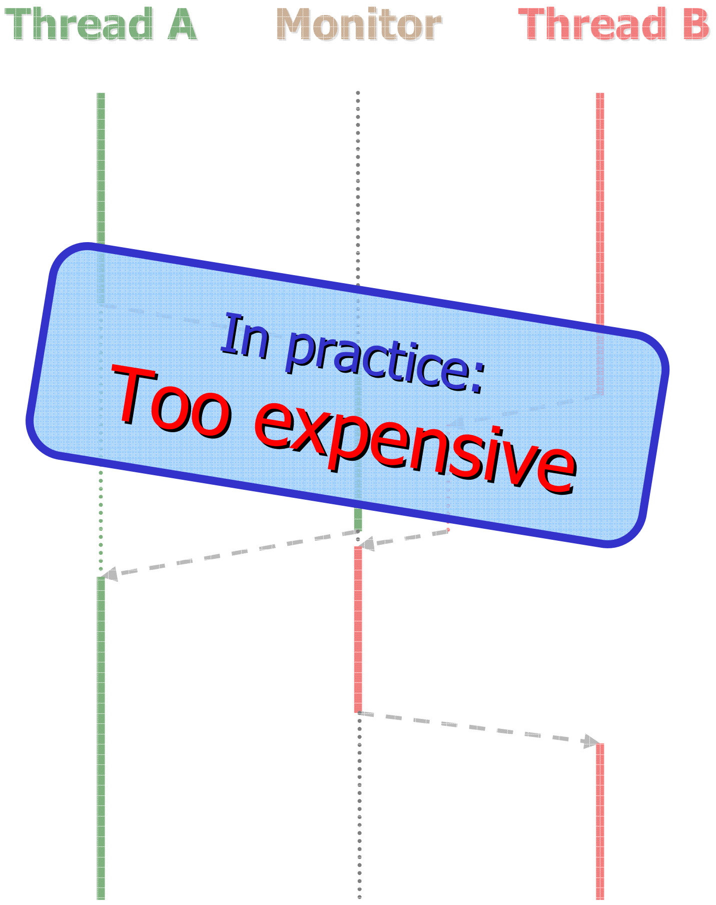
Synchronization via IPC



In practice:
Too expensive

Synchronization via IPC

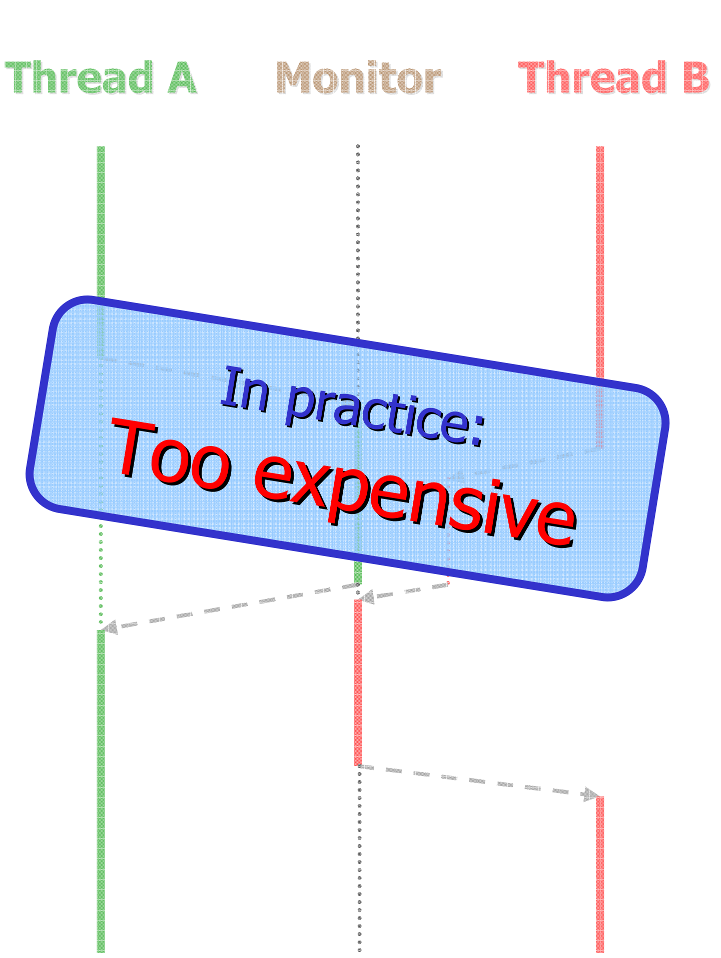
Load Distribution via IPC



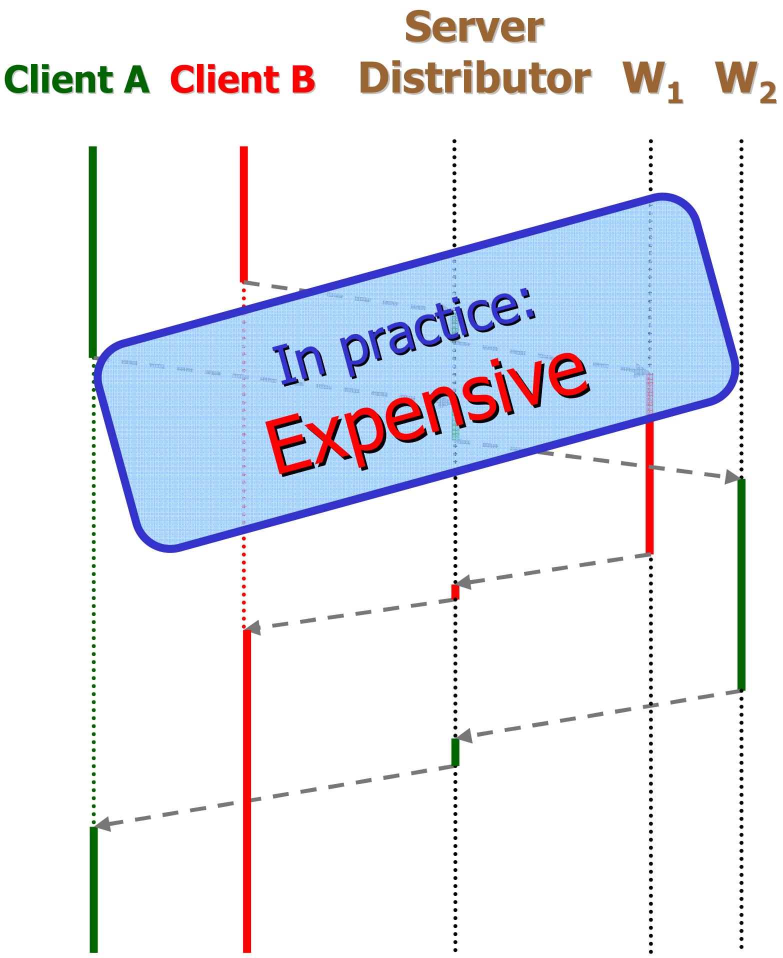
In practice:
Too expensive

Synchronization via IPC

Load Distribution via IPC



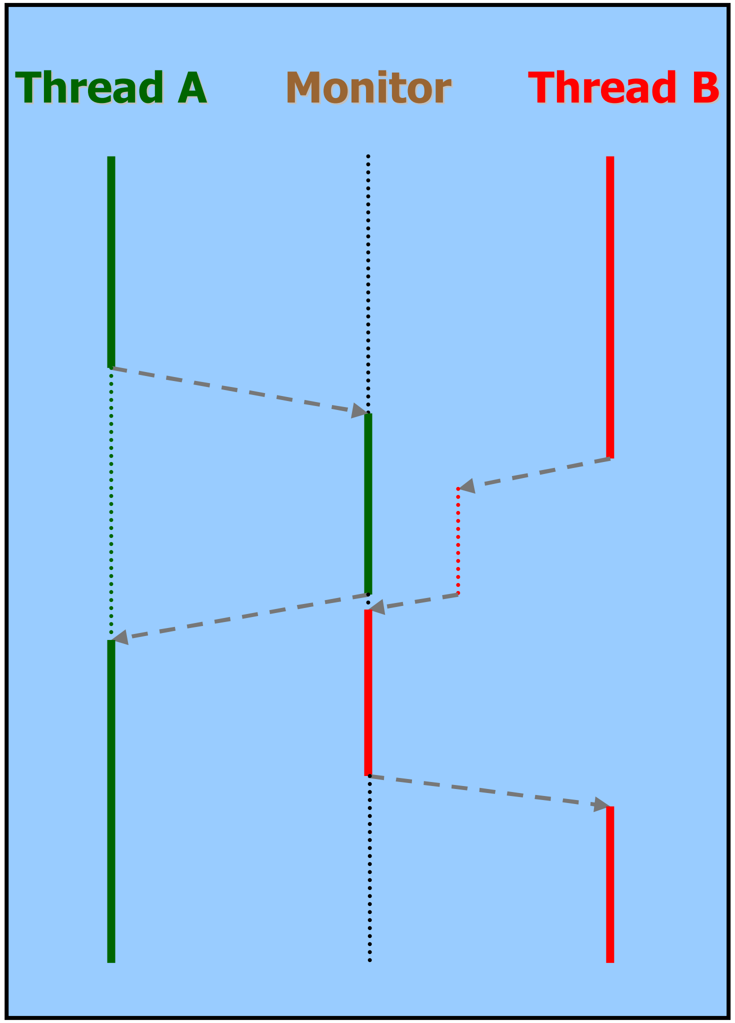
In practice:
Too expensive

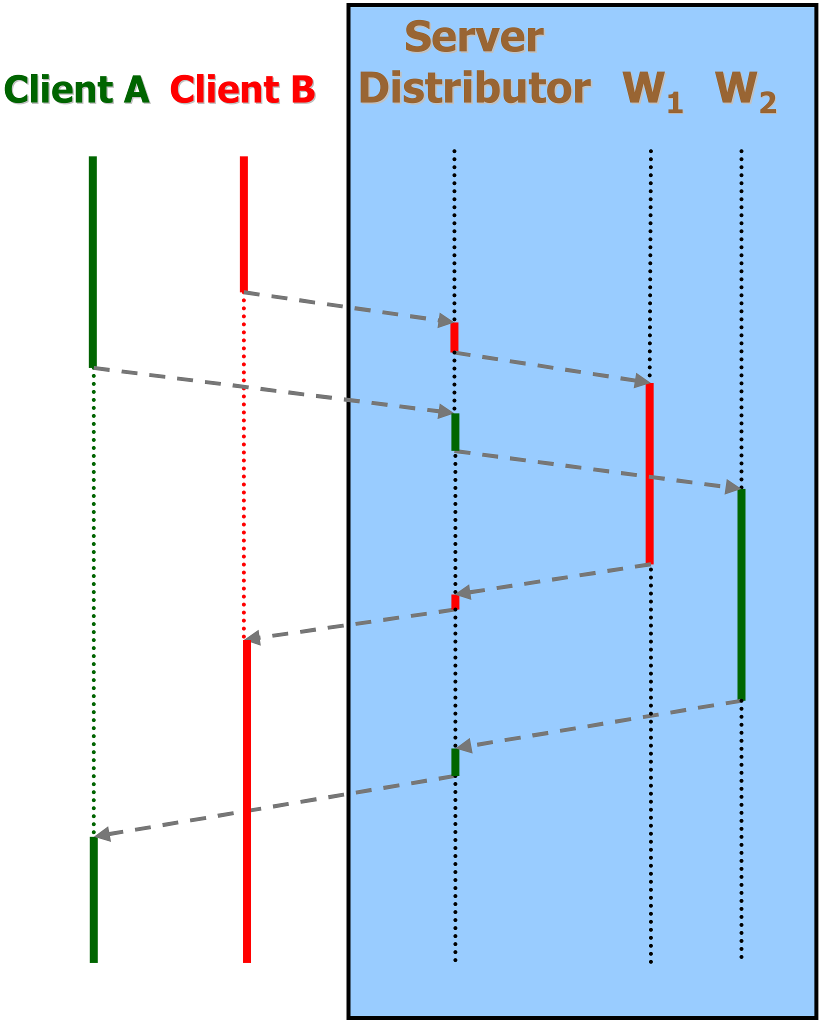


In practice:
Expensive

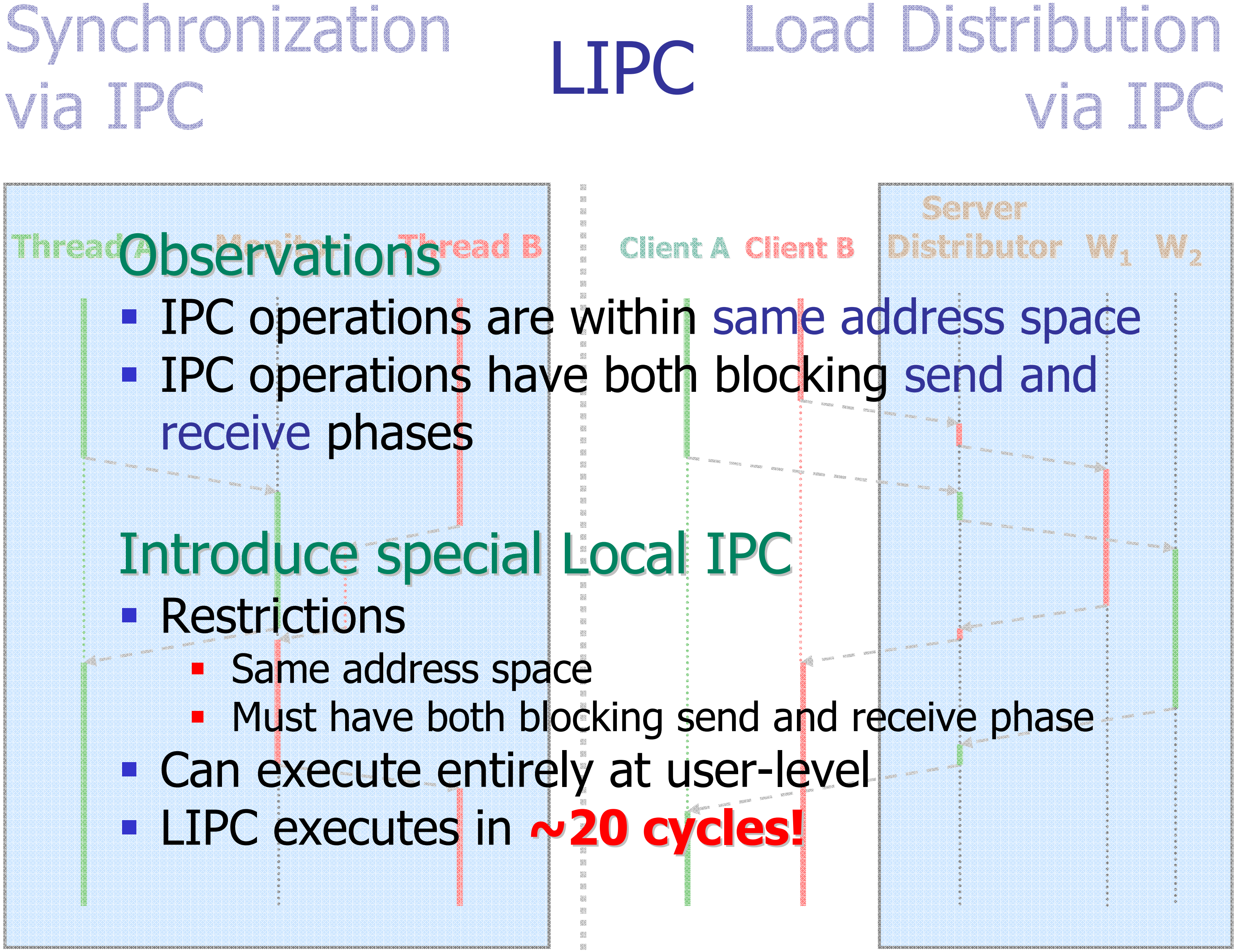
Synchronization via IPC

Load Distribution via IPC





LIPC



Observations

- IPC operations are within same address space
- IPC operations have both blocking send and receive phases

Introduce special Local IPC

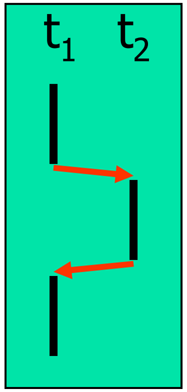
- Restrictions
  - Same address space
  - Must have both blocking send and receive phase
- Can execute entirely at user-level
- LIPC executes in **~20 cycles!**

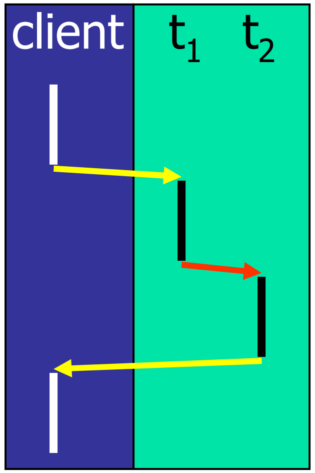
User-Level Threads?

- Would achieve required speed
- But …
  - Not known to the kernel
  - Execute in a single thread’s context
  - Making them kernel-schedulable does not pay
  - Two concepts – inelegant, contradicts minimality
- We want …
  - Kernel-level threads
  - The speed of user-level threads

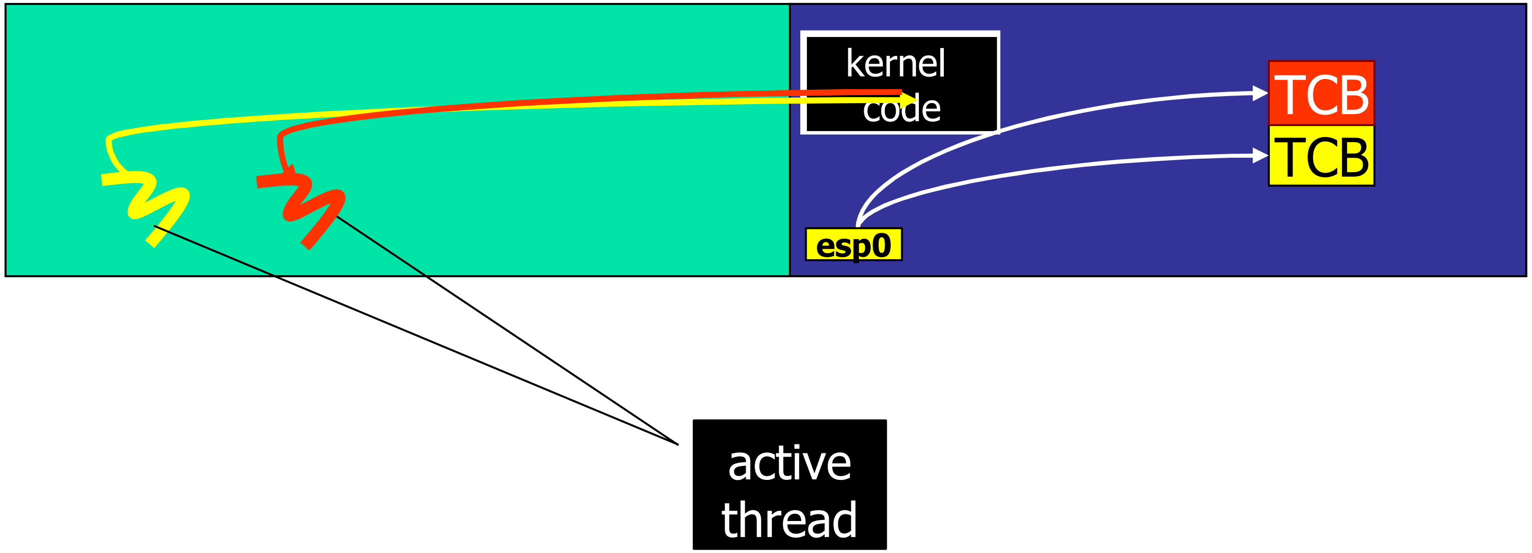
Basic Idea

- Assume IPC $t_1 \rightarrow t_2$, same address space
- Let t_1 execute t_2-code
- Postpone real switch **until the kernel is activated**
- Pays if multiple lazy switches occur before first kernel activation, e.g.:
  - $t_1 \rightarrow t_2$, work, $t_2 \rightarrow t_1$
    - Costs 0 kernel-level IPC
  - client $\rightarrow t_1 \rightarrow t_2 \rightarrow$ client
    - Costs 2 kernel-level IPCs

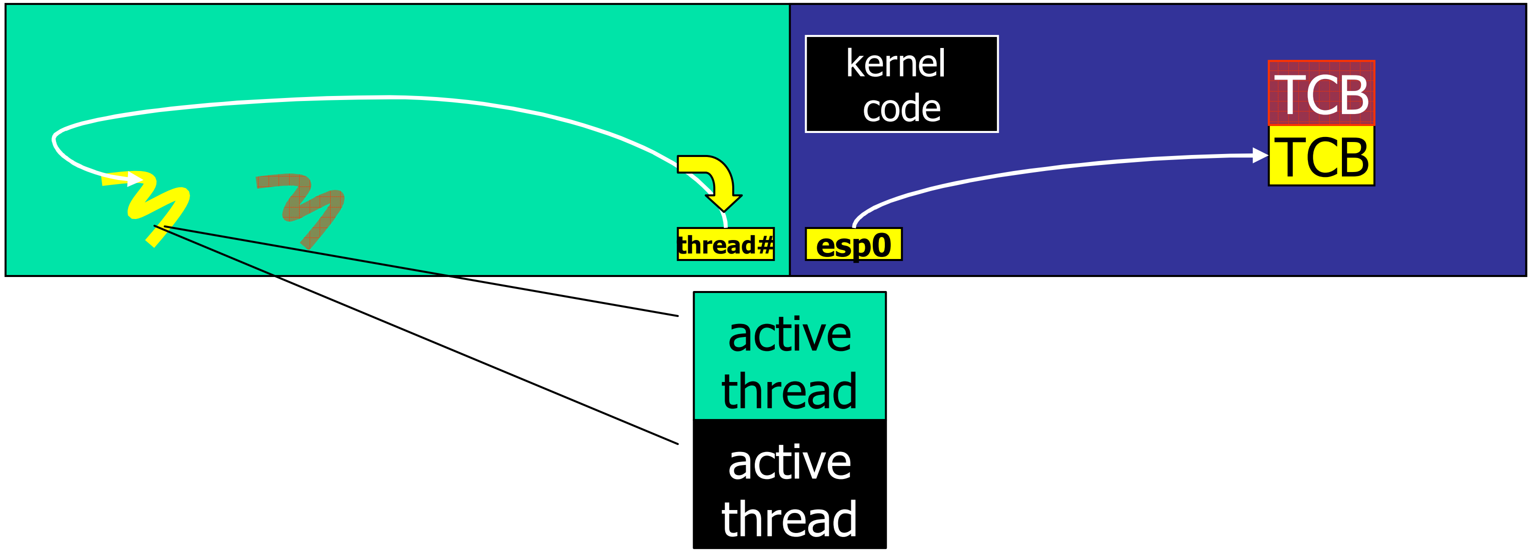




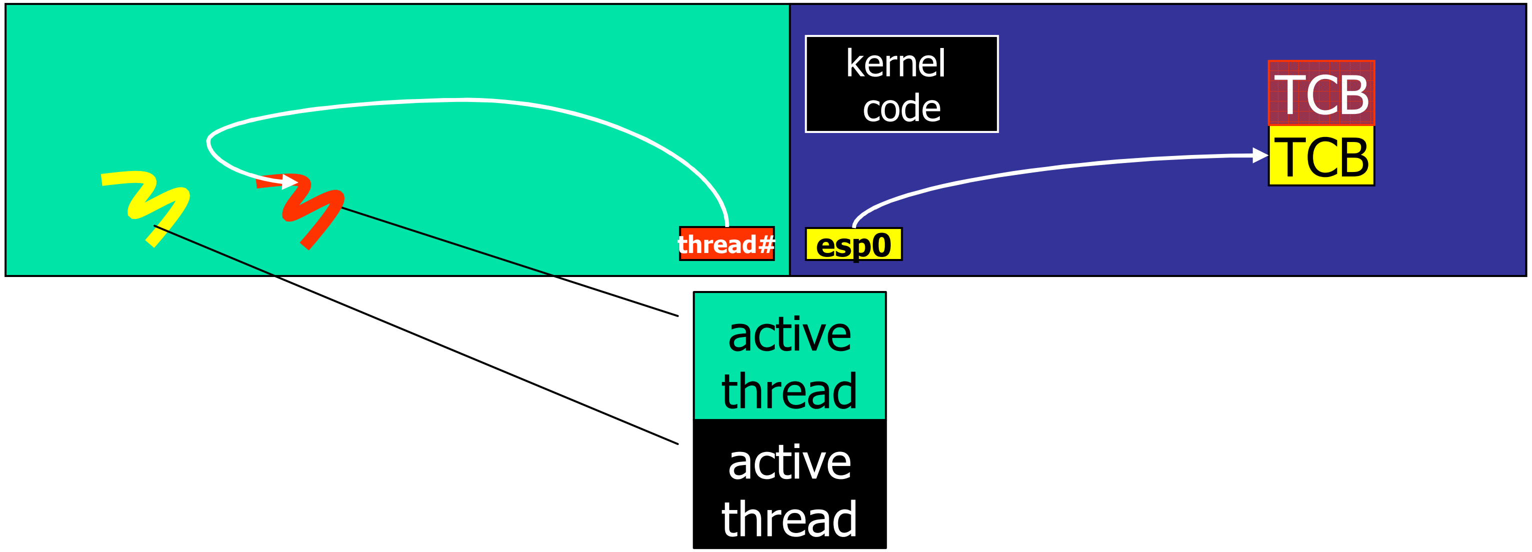
Strict Switching



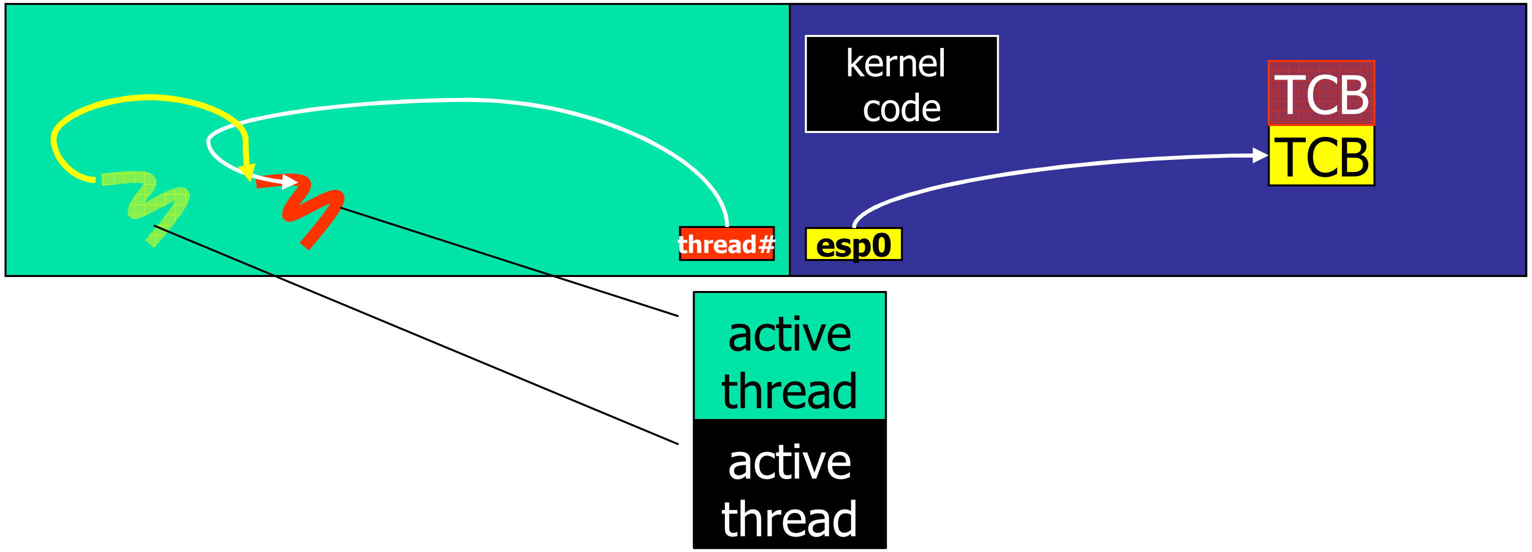
Lazy Switching



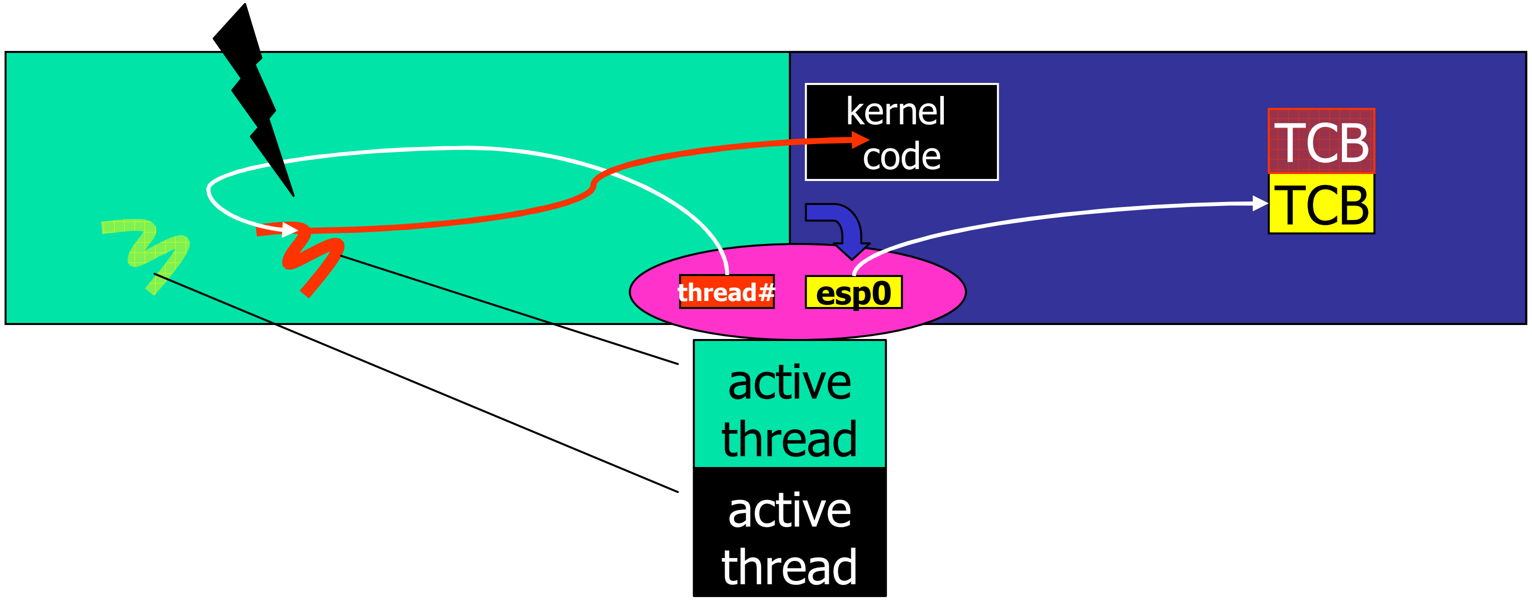
Lazy Switching



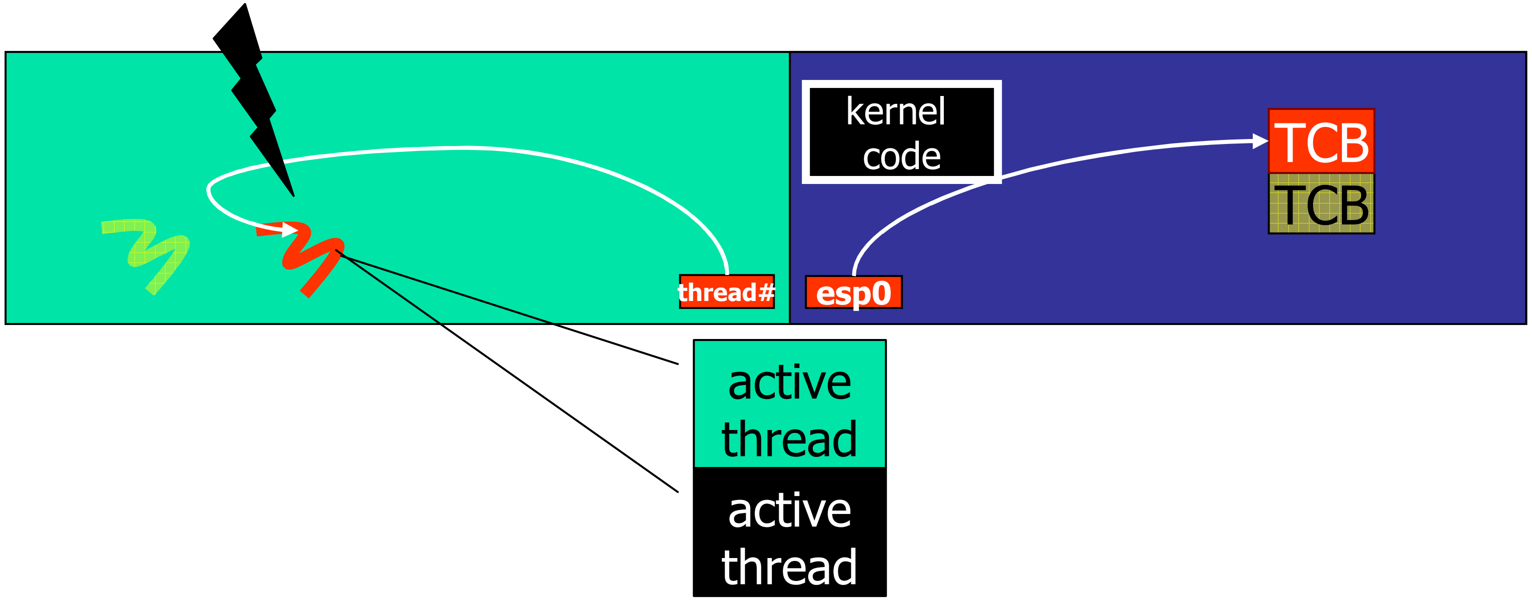
Lazy Switching



Lazy Switching



Lazy Switching



IPC Revisited

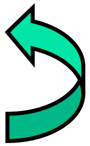
```
A → B: SendAndWaitForReply in user-mode
    call IPC function, i.e. push A's instruction pointer
    if B is valid thread id and thread B waits for thread A
    then
        save A's stack pointer
        set A's status to "wait for B"
        set B's status to "run"
        load B's stack pointer
        current thread := B
        return, i.e. pop B's instruction pointer
    else
        more complicated IPC handling
    endif
```

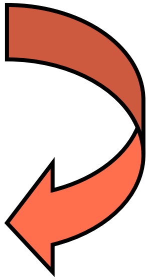
Atomicity?

Kernel Data?

Atomicity

```
A → B: SendAndWaitForReply in user-mode
    call IPC function, i.e. push A’s instruction pointer
    save A’s stack pointer
    – restart point –
    if B is valid thread id and thread B waits for thread A
    then
        – forward point –
        set A’s status to “wait for B”
        set B’s status to “run”
        load B’s stack pointer
        current thread := B
        – completion point –
        return, i.e. pop B’s instruction pointer
    else
        more complicated IPC handling
    endif
```



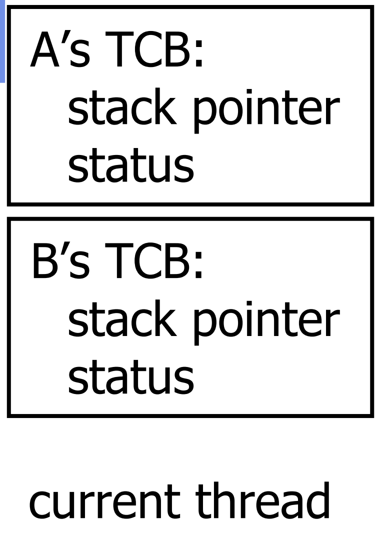


Atomicity (2)

Interruption between forward point and completion point:

```
if is page fault
then kill thread A
else
    set A’s status to “wait for B”
    set B’s status to “run”
    load B’s stack pointer
    current thread := B
    set interrupted instruction pointer to completion point
endif
```

Kernel Data



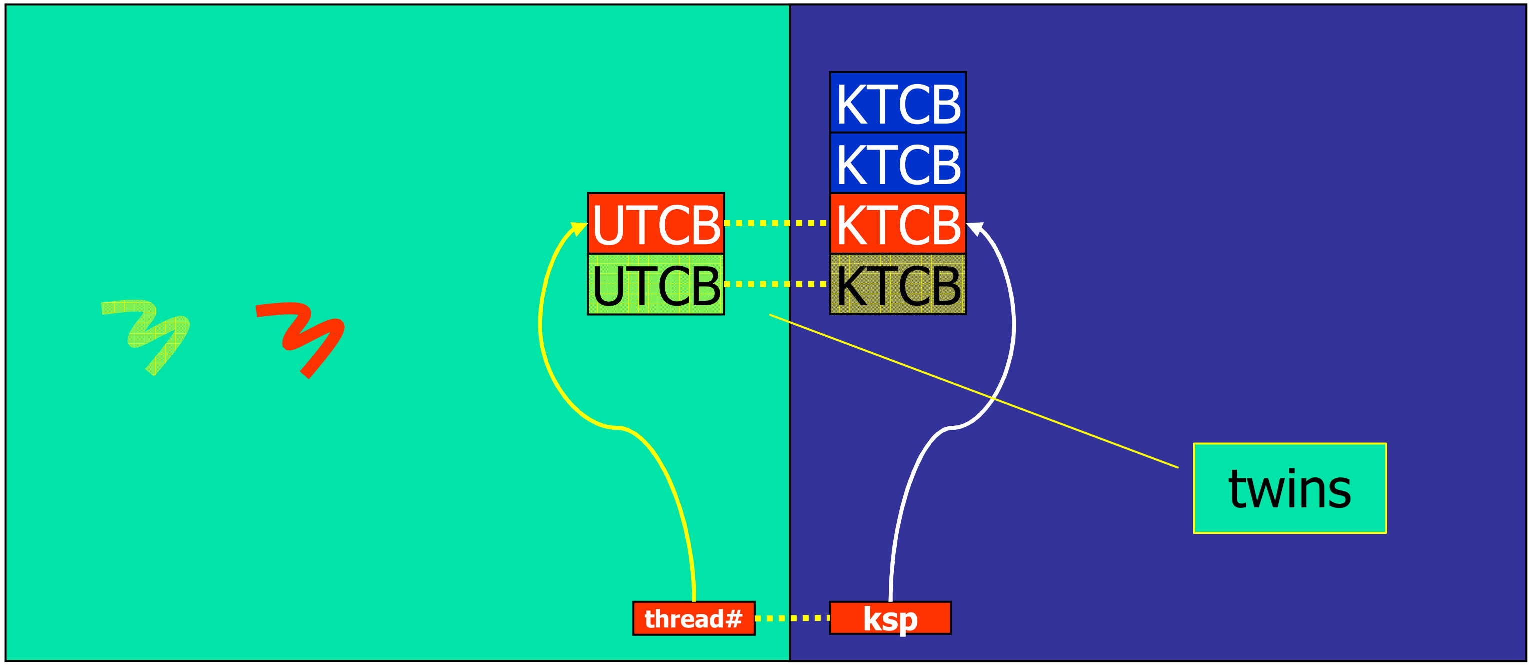
A’s TCB:
stack pointer
status

B’s TCB:
stack pointer
status

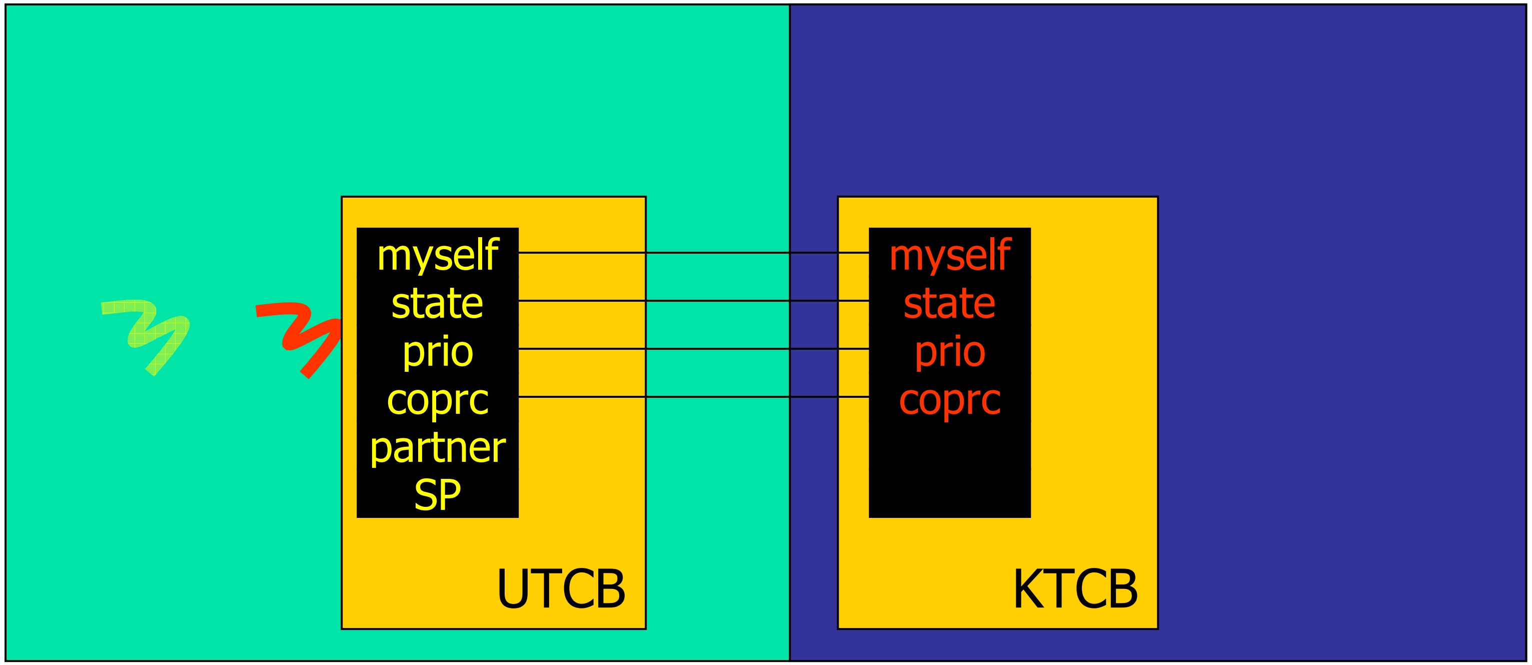
current thread

- Stack pointer
  - Can be user accessible
- Status
  - User-level effects
    - Local to A’s task can be ignored
    - Indirect effects on other tasks can be ignored
  - System-level effects
    - Must be avoided
    - Validate values or
    - Maintain twin variable in kernel

UTCB – KTCB



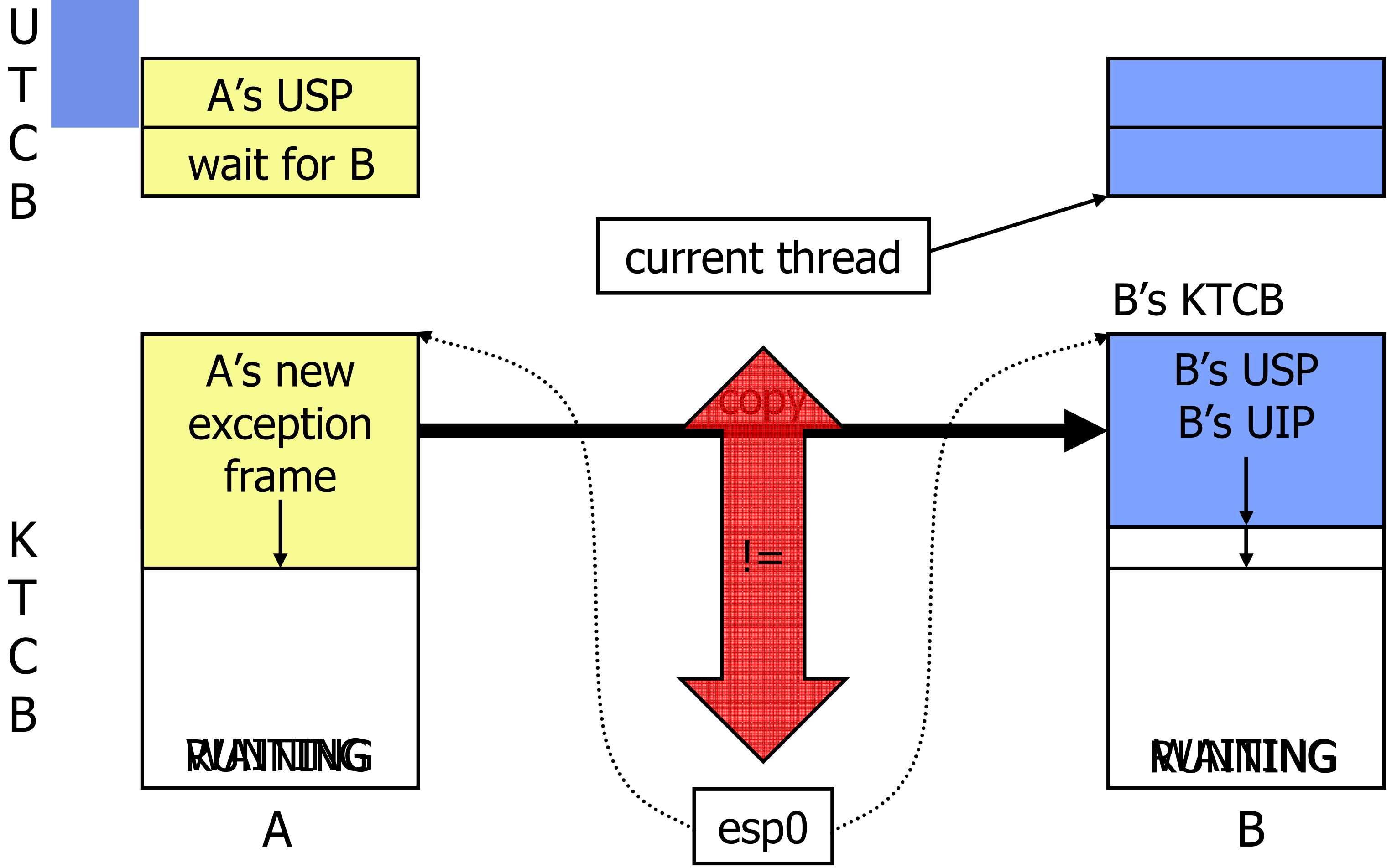
UTCB – KTCB


myself
state
prio
coprc
partner
SP
UTCB
myself
state
prio
coprc
KTCB

Current_thread Inconsistency

```
if CurrentUTCB is valid UTCB
then
    NewKTCB := CurrentUTCB.ktcb
    if  NewKTCB is valid KTCB and
        NewKTCB.space = CurrentKTCB.space and
        NewKTCB.utcb = CurrentUTCB
    then
        update kernel state
        CurrentKTCB := NewKTCB
        return
    endif
endif
kill thread(CurrentKTCB)
```

Kernel State Fixup – A → B



U T C B

A's USP

wait for B

current thread

B's KTCB

K T C B

A's new exception frame

copy

!=

B's USP
B's UIP

RUNNING
WAITING

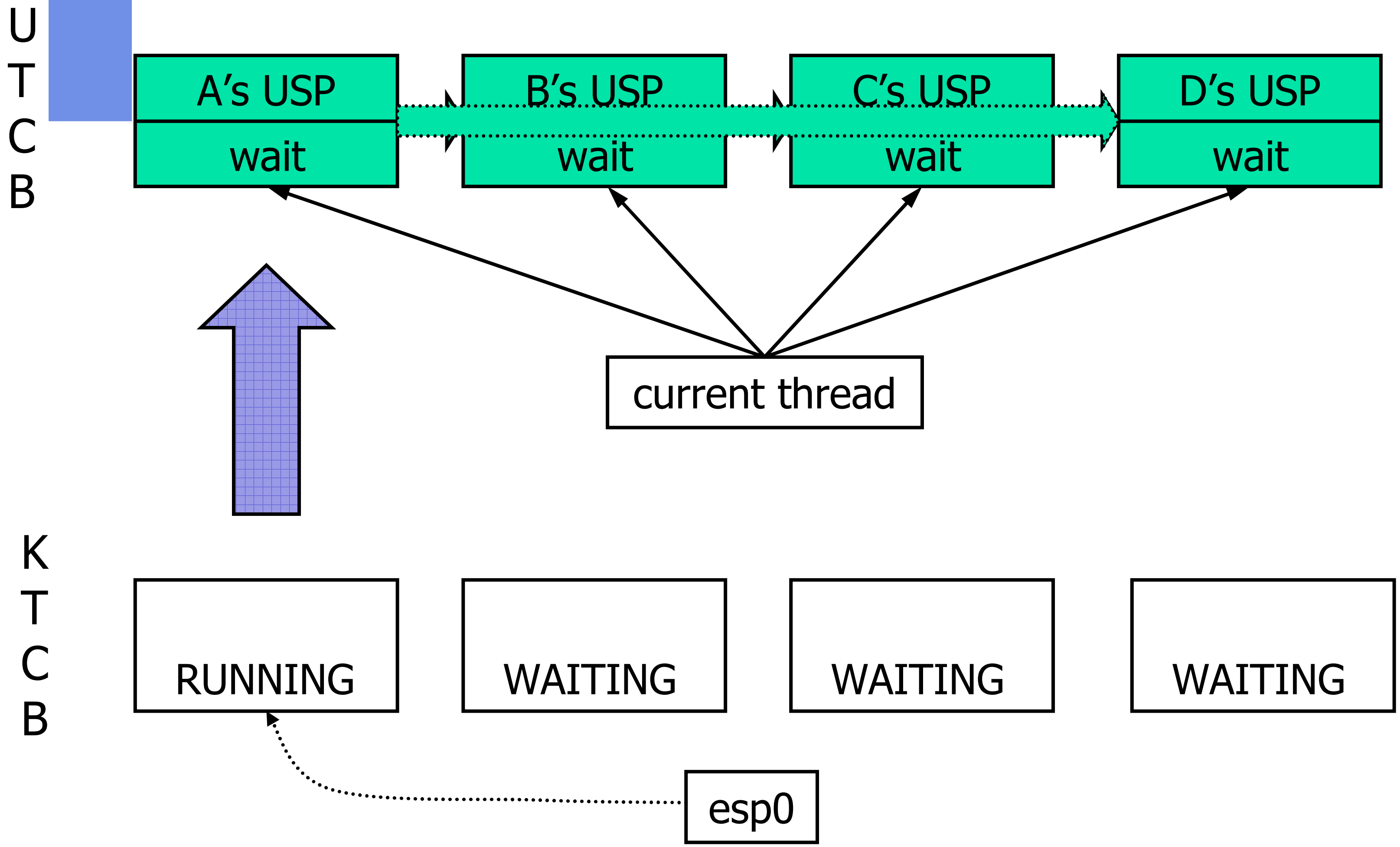
WAITING
RUNNING

A

esp0

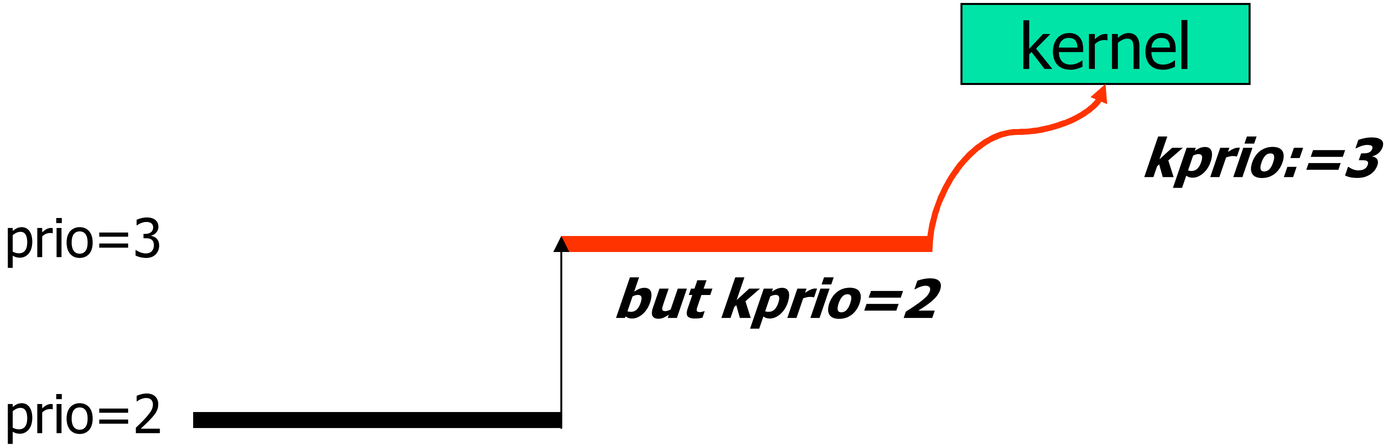
B

LIPC Chains



© 2009 Universität Karlsruhe, System Architecture Group

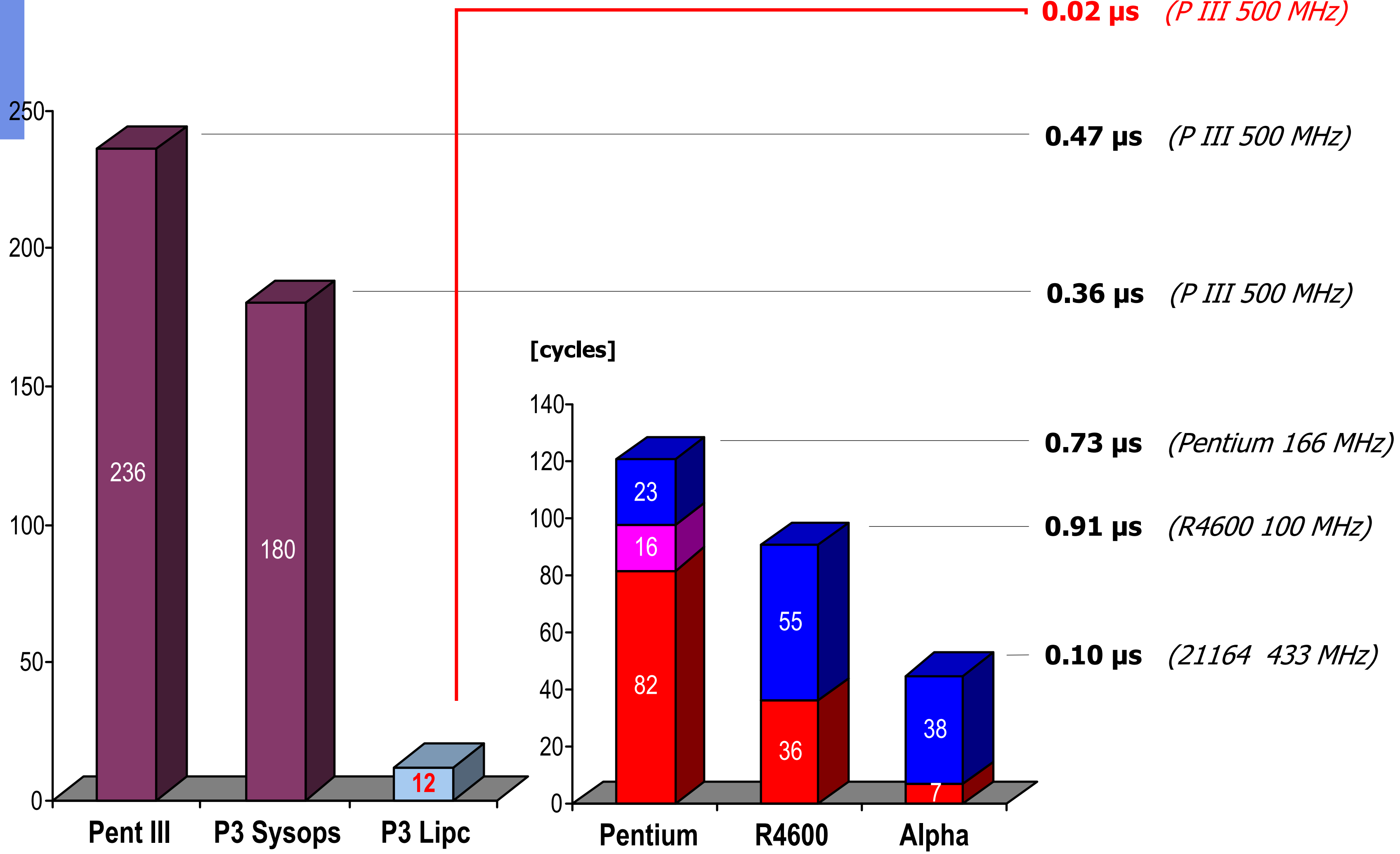
What About Priorities?



Safety & Security

- Threads can only destroy their **own** task.
  - Possible even without lazy switching.
- Threads can only cheat about their identity **within** their own task.
  - Possible even without lazy switching.
- Threads **cannot** modify their effective priority, uid, etc.

IPC Performance Promise – May 2001



IPC Performance – Prototype

- LIPC: 23 cycles
  - 1/15th of regular IPC (no sysops, no fastpath)
- Overhead on IPC due to LIPC extensions
  - 43 cycles intra-AS IPC
  - 146 cycles inter-AS IPC
    - UTCB synchronization

Too much for real-world systems:
P3 inter-AS IPC was only
180 cycles w/o LIPC support!

- Overhead due to kernel fixup
  - ???

Limitations of LIPC

- Intra address space only
- Register-only IPC, no map/grant/string
- Always send and receive phase
- Infinite receive timeout

- Tricky
  - Change from Wait_for_X to Wait_for_Any