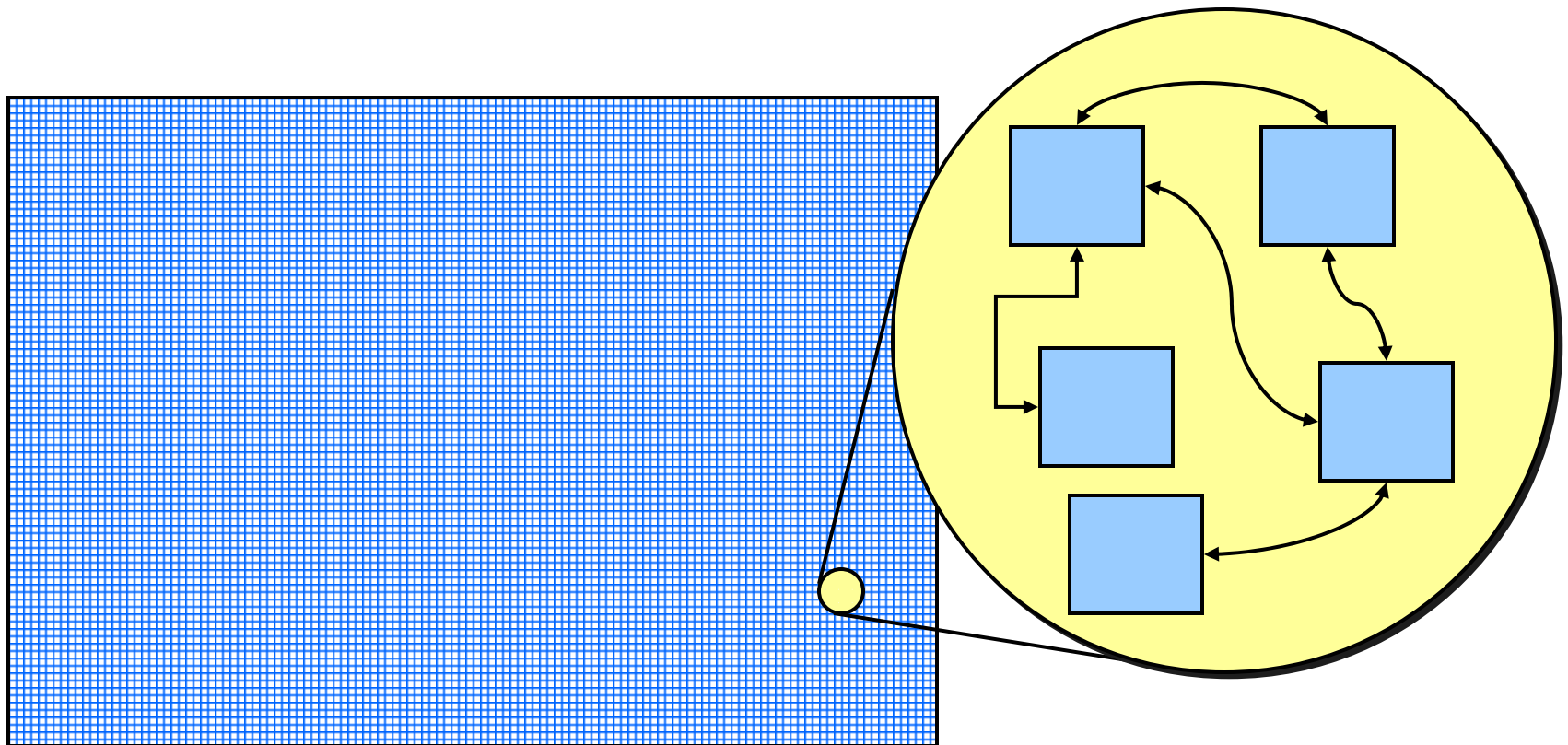


Microkernel Construction

IPC Implementation



IPC Importance



General IPC Algorithm

- Validate parameters
- Locate target thread
 - if unavailable, deal with it
- Transfer message
 - untyped - short IPC
 - typed message - long IPC
- Schedule target thread
 - switch address space as necessary
- Wait for IPC



IPC - Implementation

Short IPC



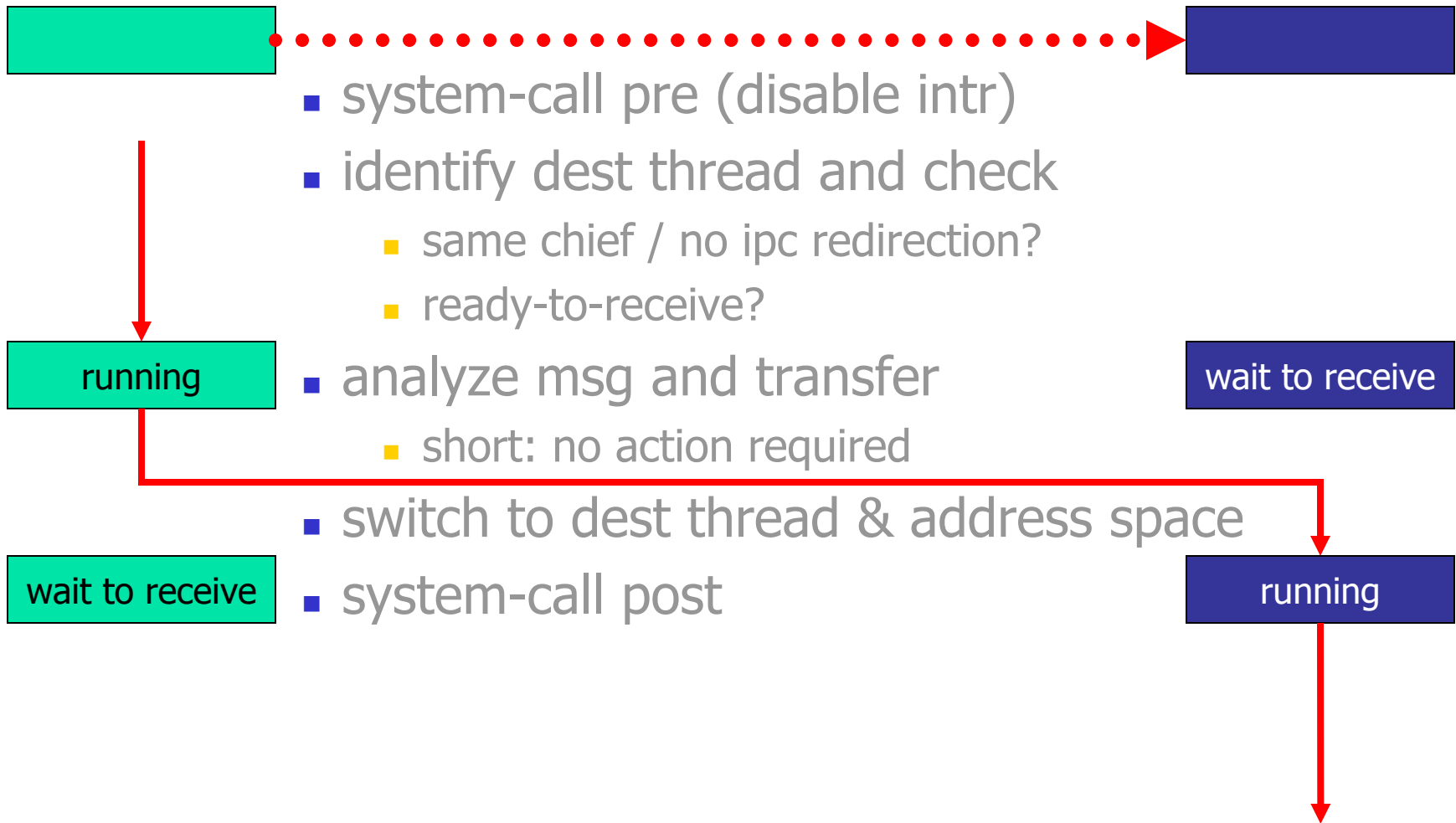
Short IPC (uniprocessor)

- system-call preamble (disable intr)
- identify dest thread and check
 - same chief / no ipc redirection?
 - ready-to-receive?
- analyze msg and transfer
 - short: no action required
- switch to dest thread & address space
- system-call postamble

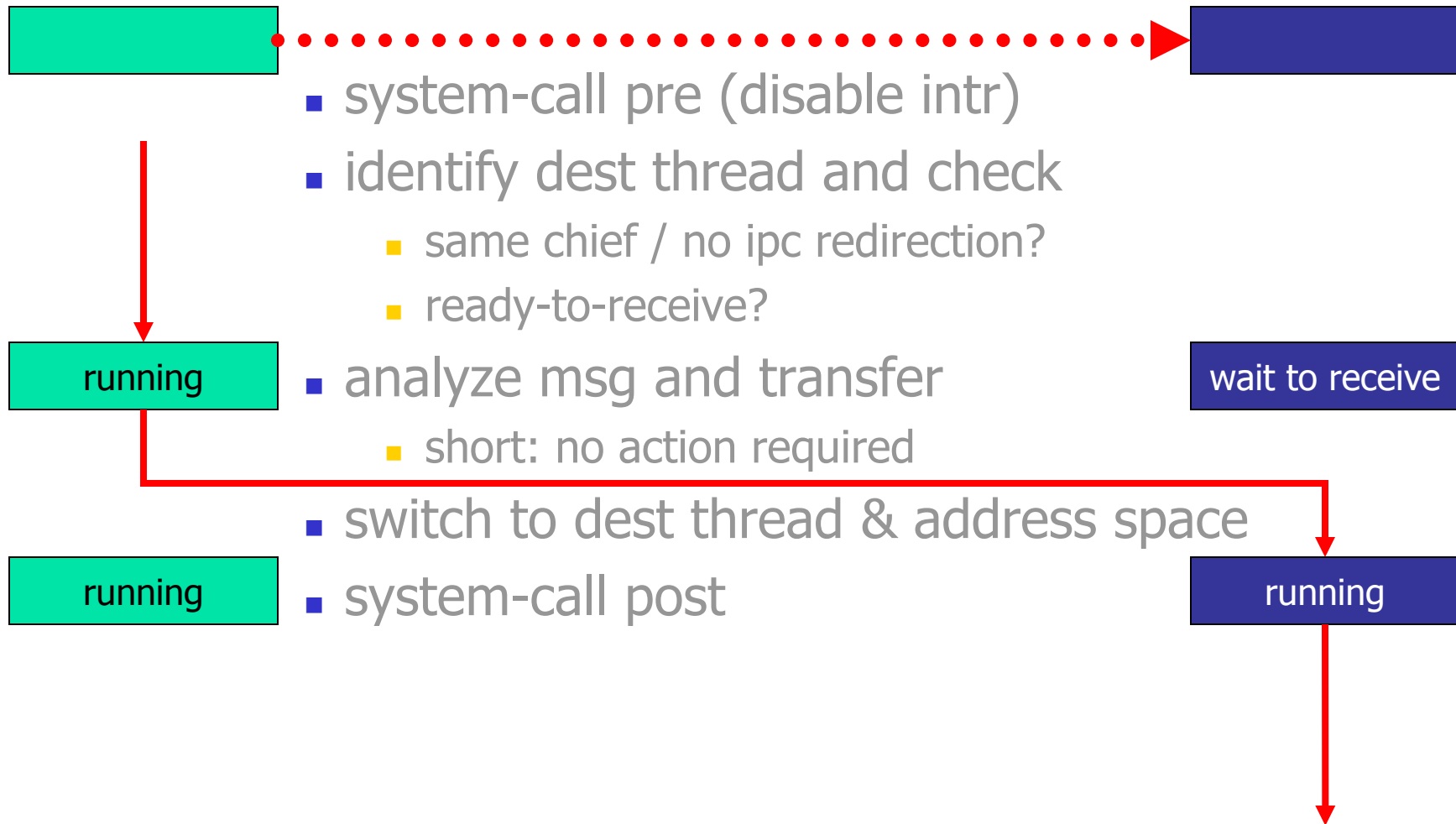
The critical path



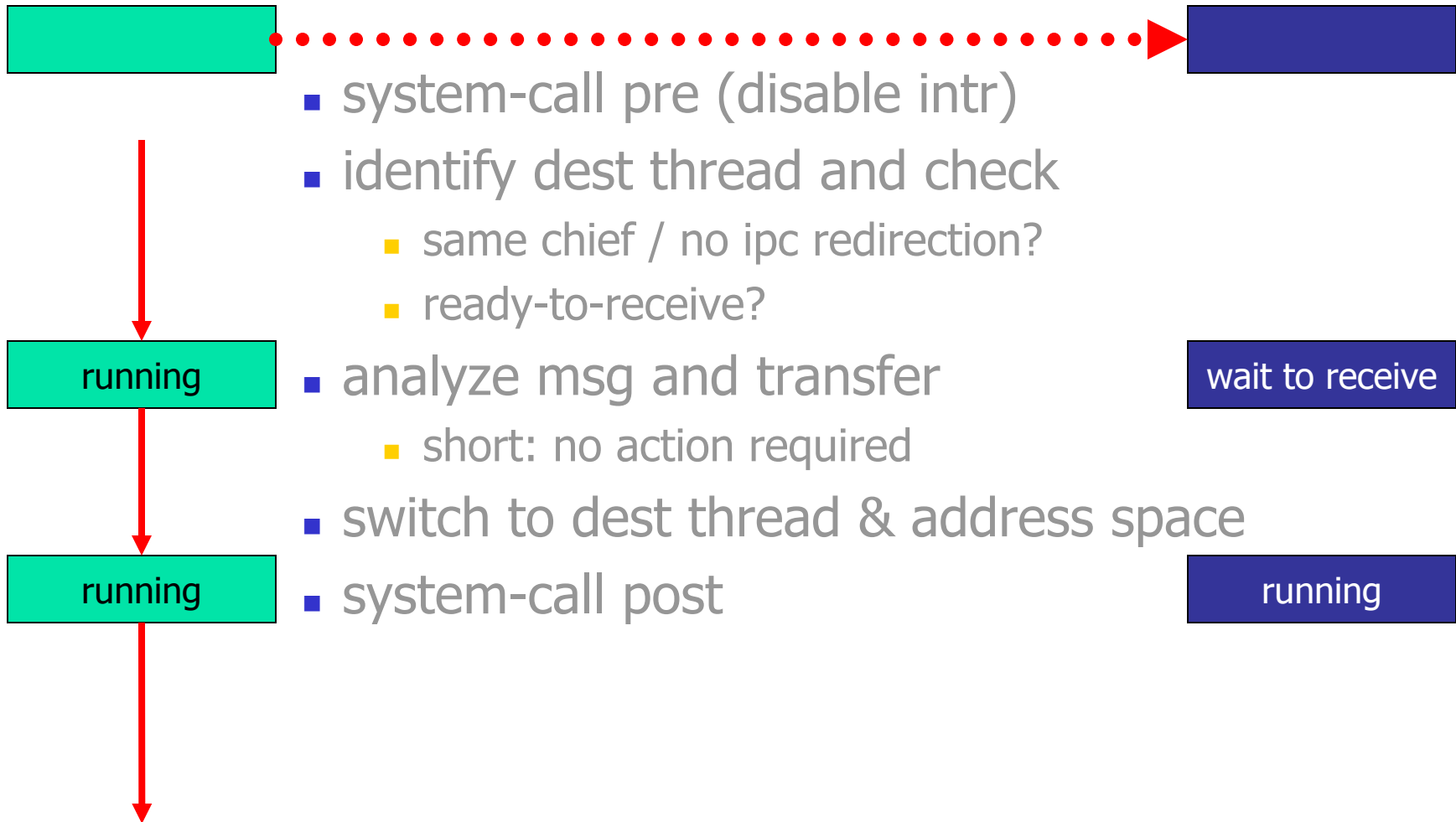
Short IPC (uniprocessor) "call"



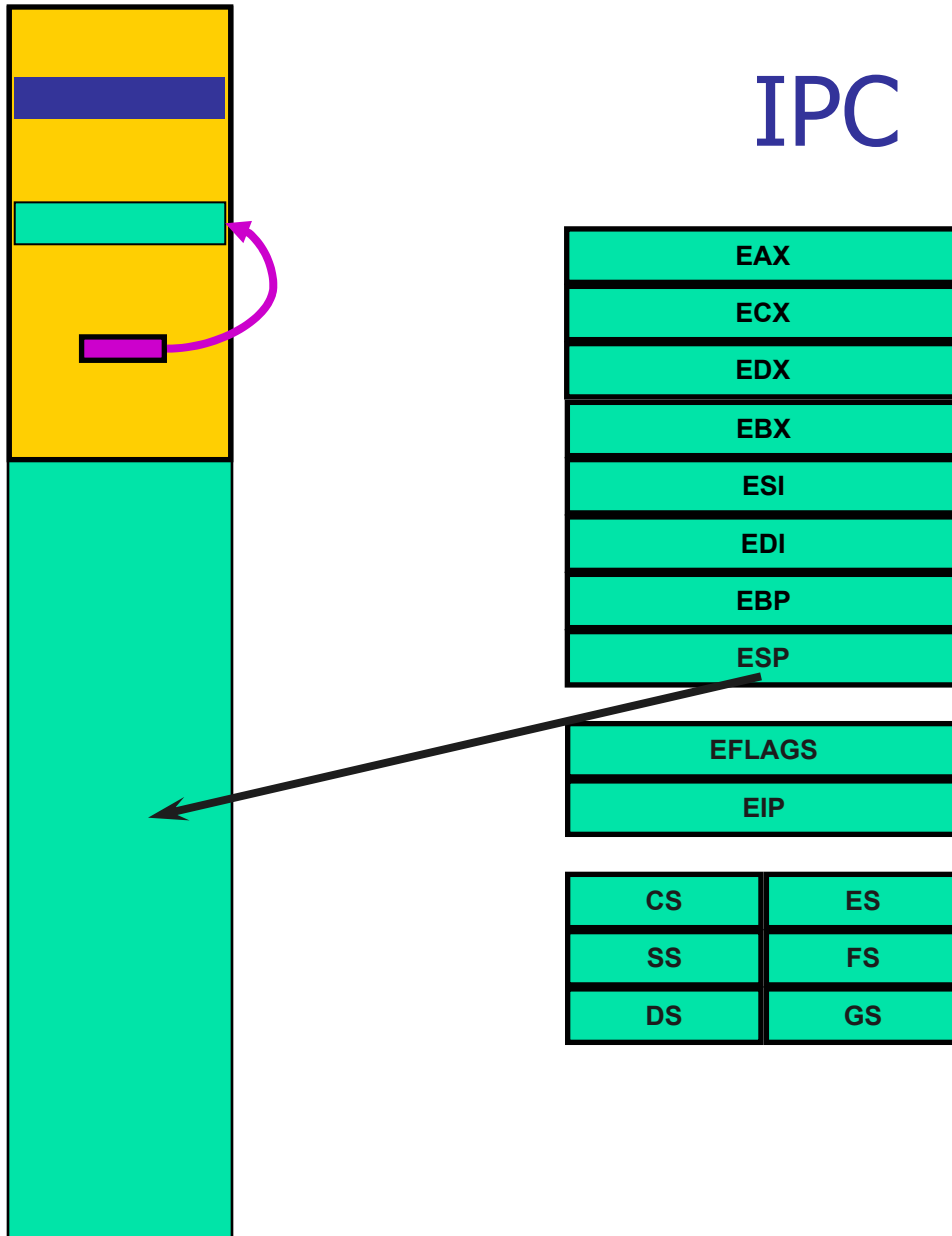
Short IPC (uniprocessor) **"send"** (eagerly)



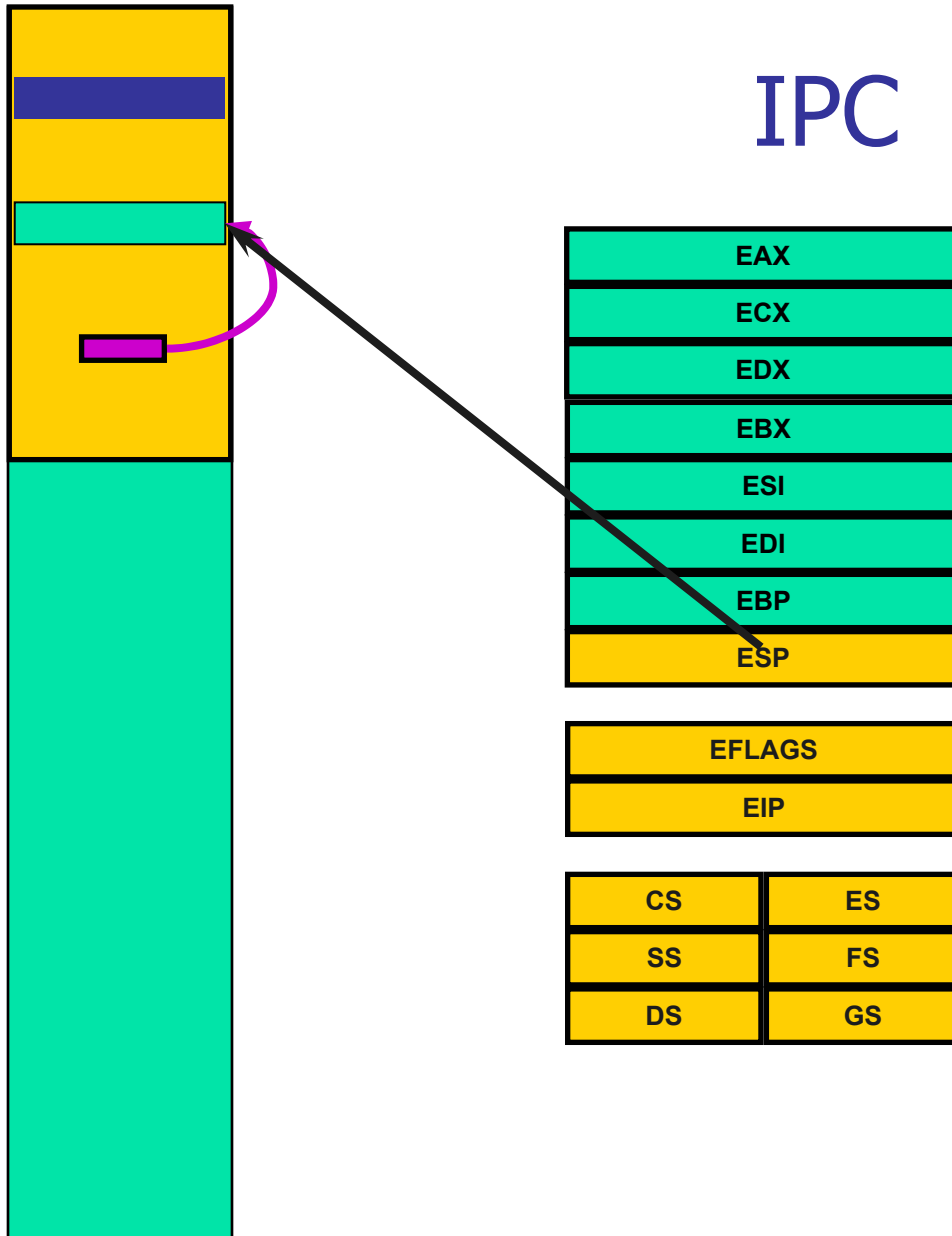
Short IPC (uniprocessor) "send" (lazily)



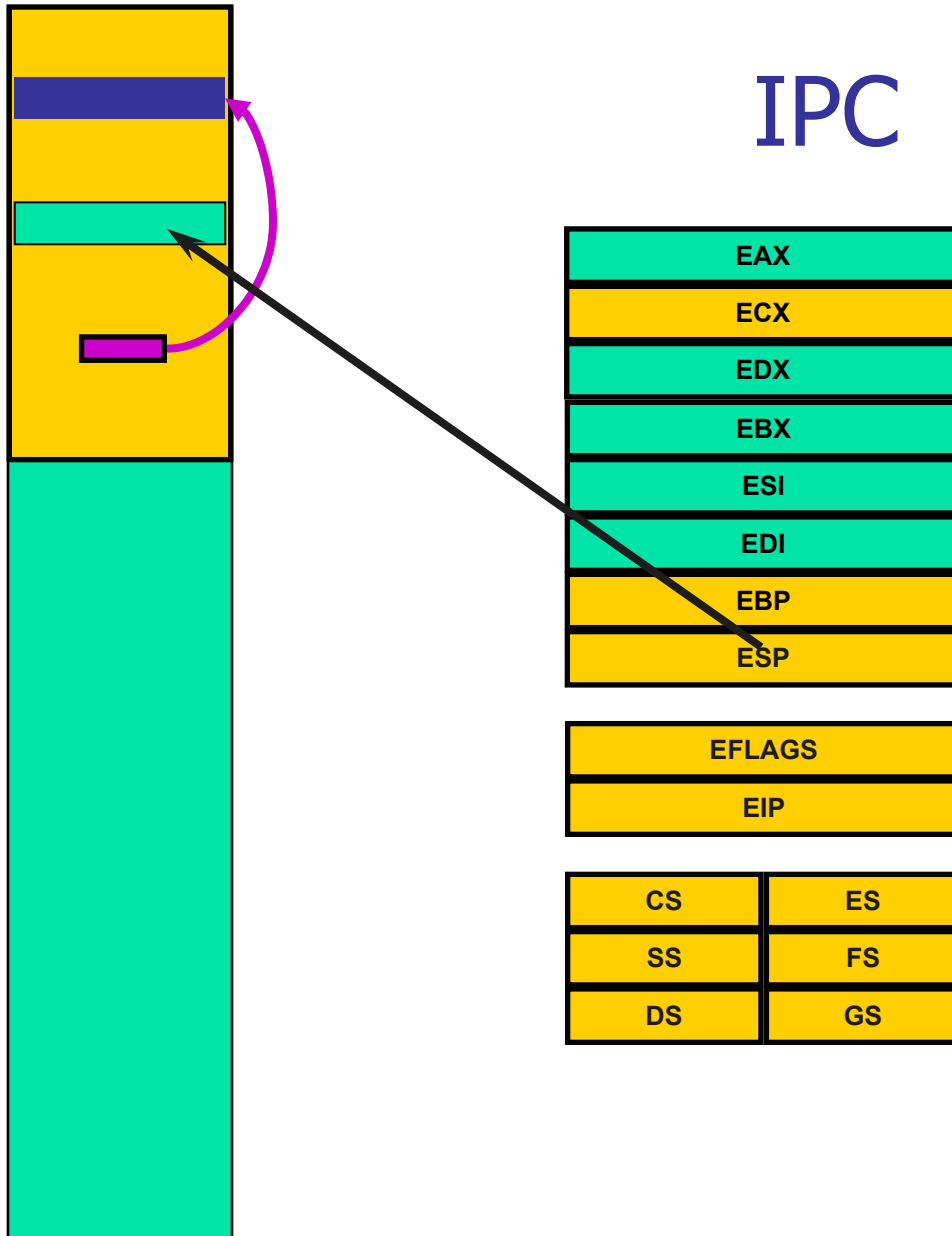
IPC



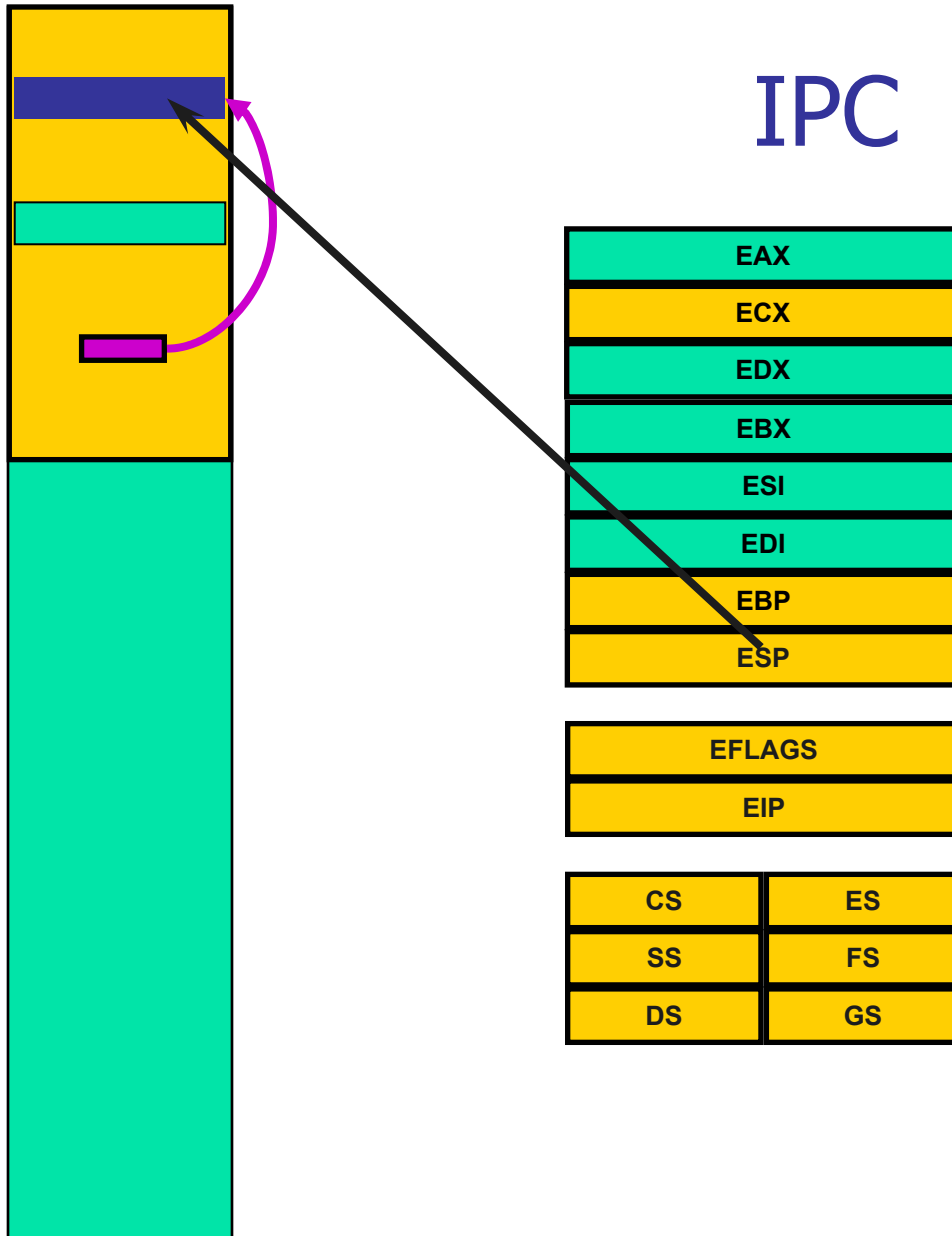
IPC



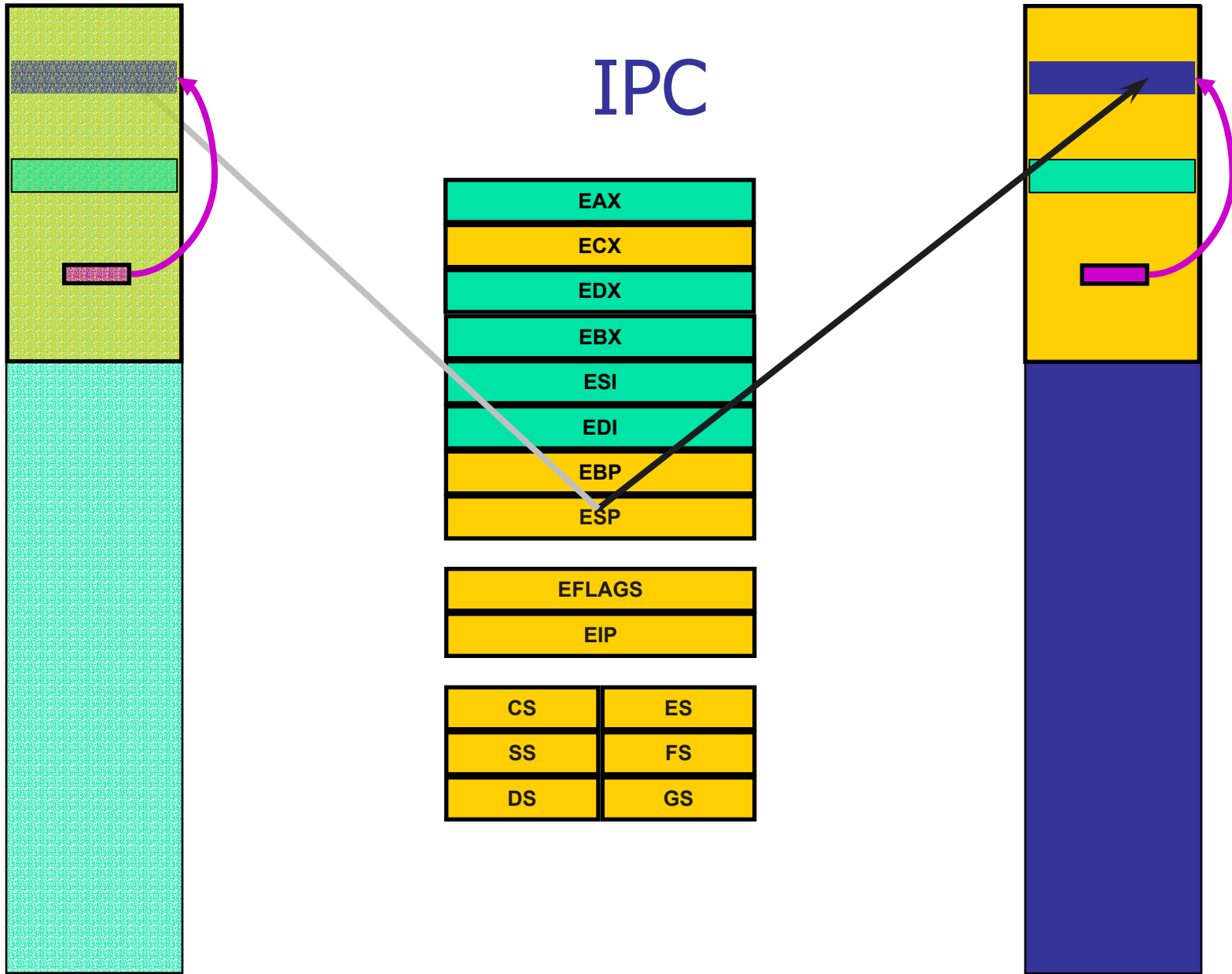
IPC



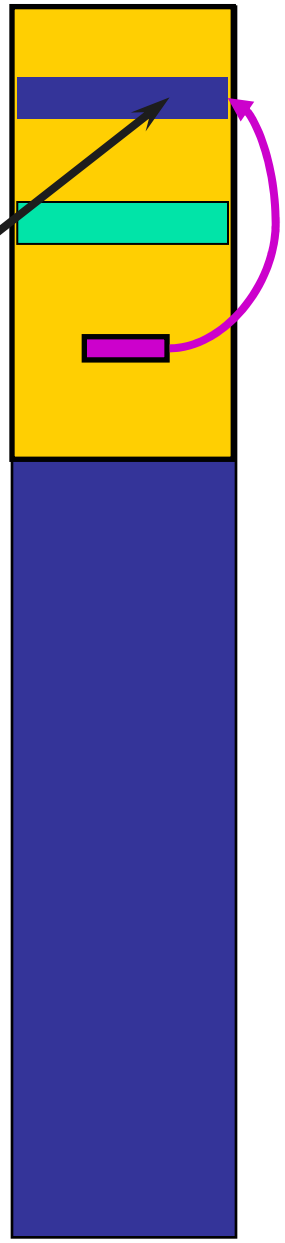
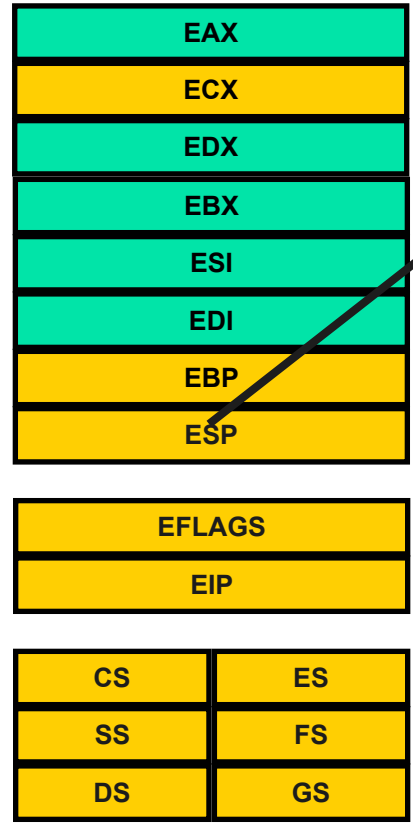
IPC



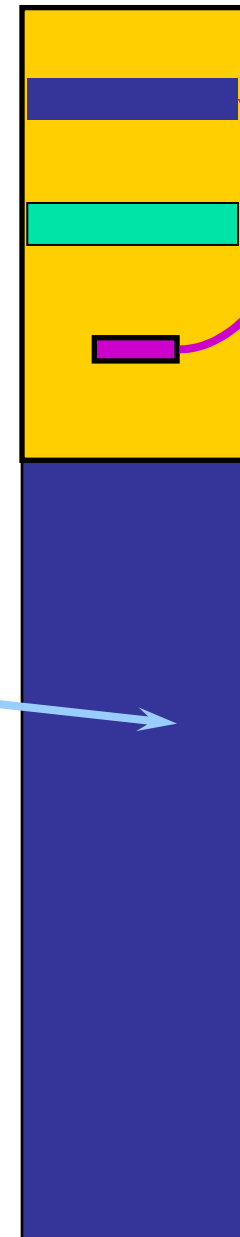
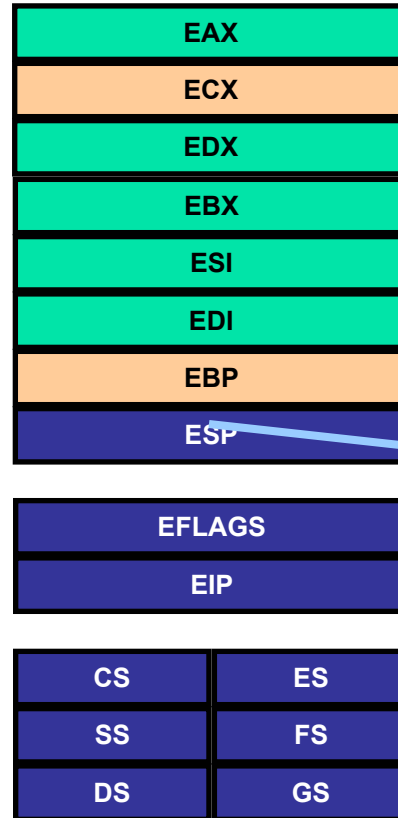
IPC



IPC



IPC



Implementation Goal

- Most frequent kernel op: short IPC
 - thousands of invocations per second
- **Performance** is critical:
 - structure IPC for speed
 - **structure entire kernel to support fast IPC**
- What affects performance?
 - cache line misses
 - TLB misses
 - memory references
 - pipe stalls and flushes
 - instruction scheduling



Fast Path

- Optimize for common cases
 - write in assembler
 - non-critical paths written in C++
 - but still fast as possible
- Avoid high-level language overhead:
 - function call state preservation
 - incompatible code optimizations
- We want every cycle possible!

IPC Attributes for Fast Path

- untyped message
- single runnable thread after IPC
 - must be valid IPC call
 - switch threads, originator blocks
 - send phase:
 - the target is waiting
 - receive phase:
 - the sender is not ready to couple, causing us to block
- no receive timeout

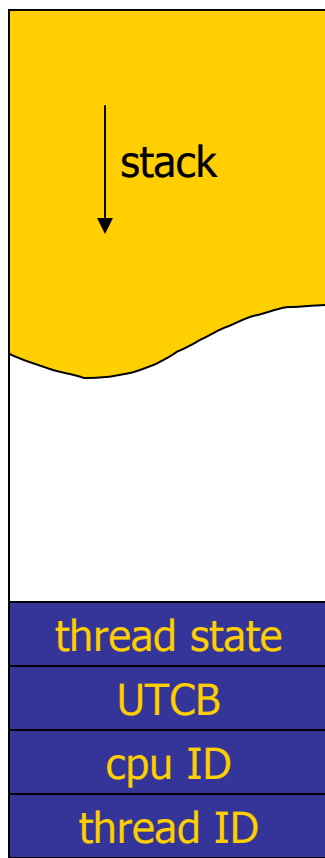


Avoid Memory References!!!

- Memory references are slow
 - avoid in IPC:
 - ex: use lazy scheduling
 - avoid in common case:
 - ex: timeouts
- Microkernel should minimize indirect costs
 - cache pollution
 - TLB pollution
 - memory bus



Optimized Memory

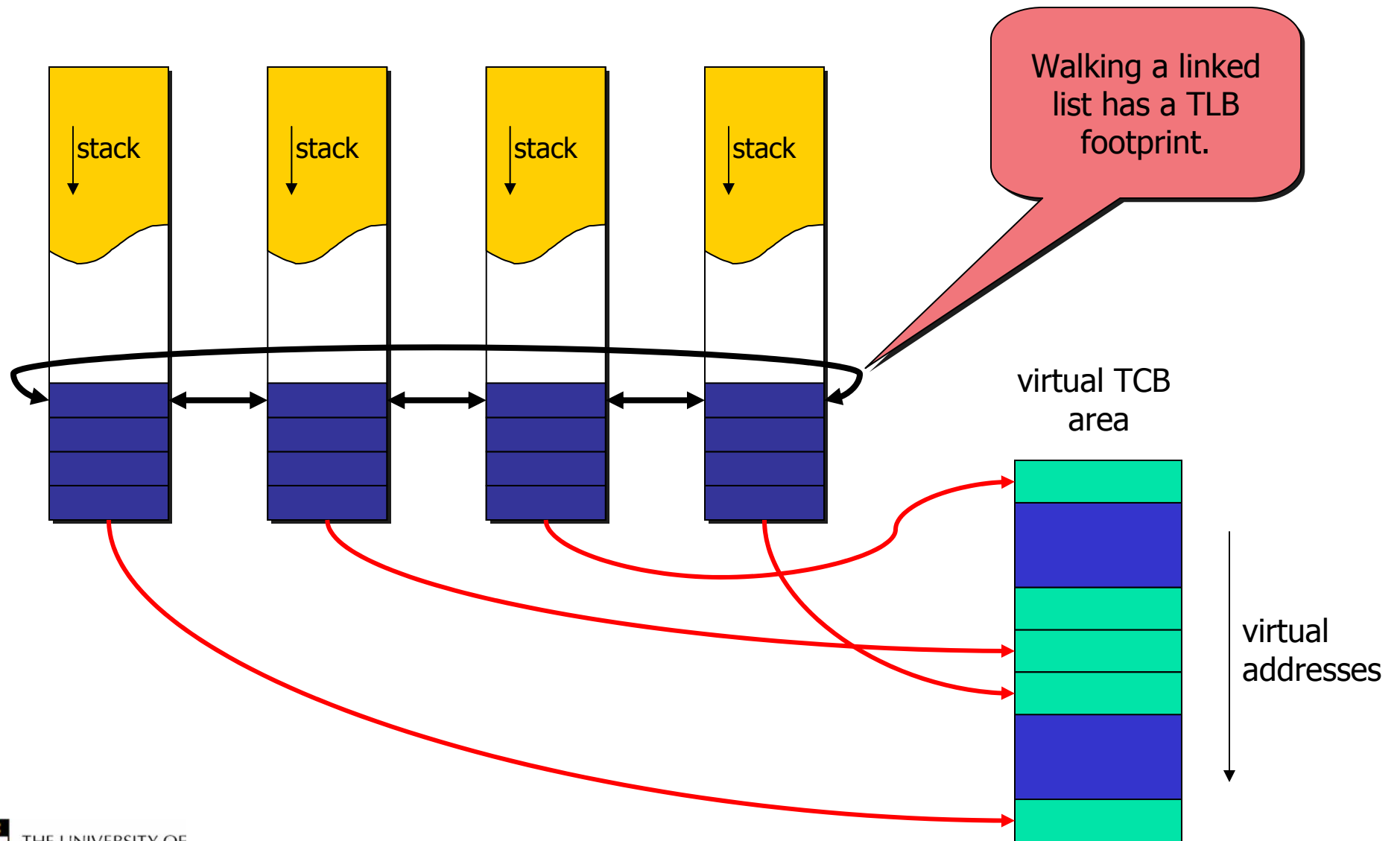


} TCB state,
grouped by
cache lines.

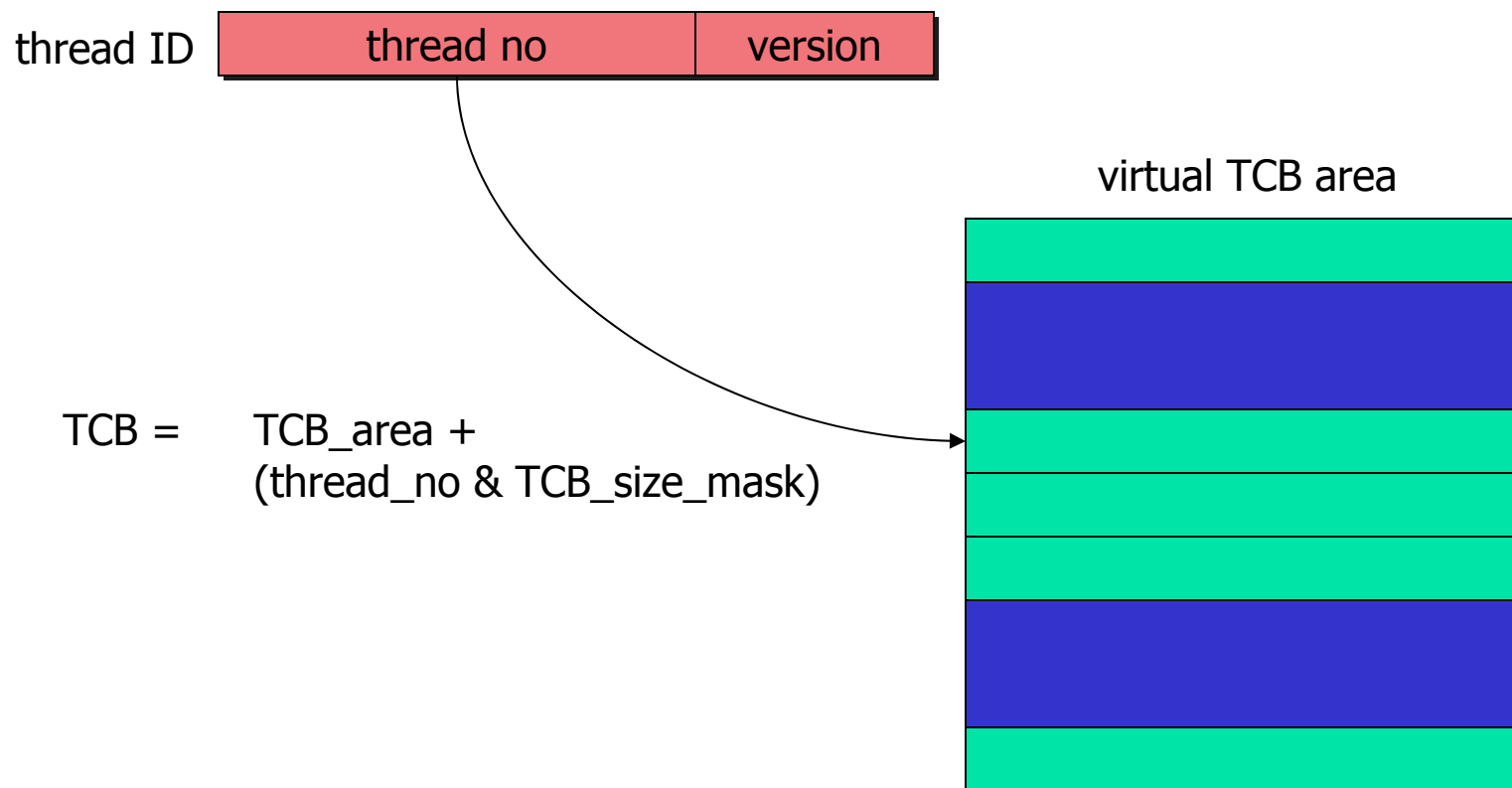
} Single TLB entry.

Also: hard-wire TLB entries for kernel code and data.

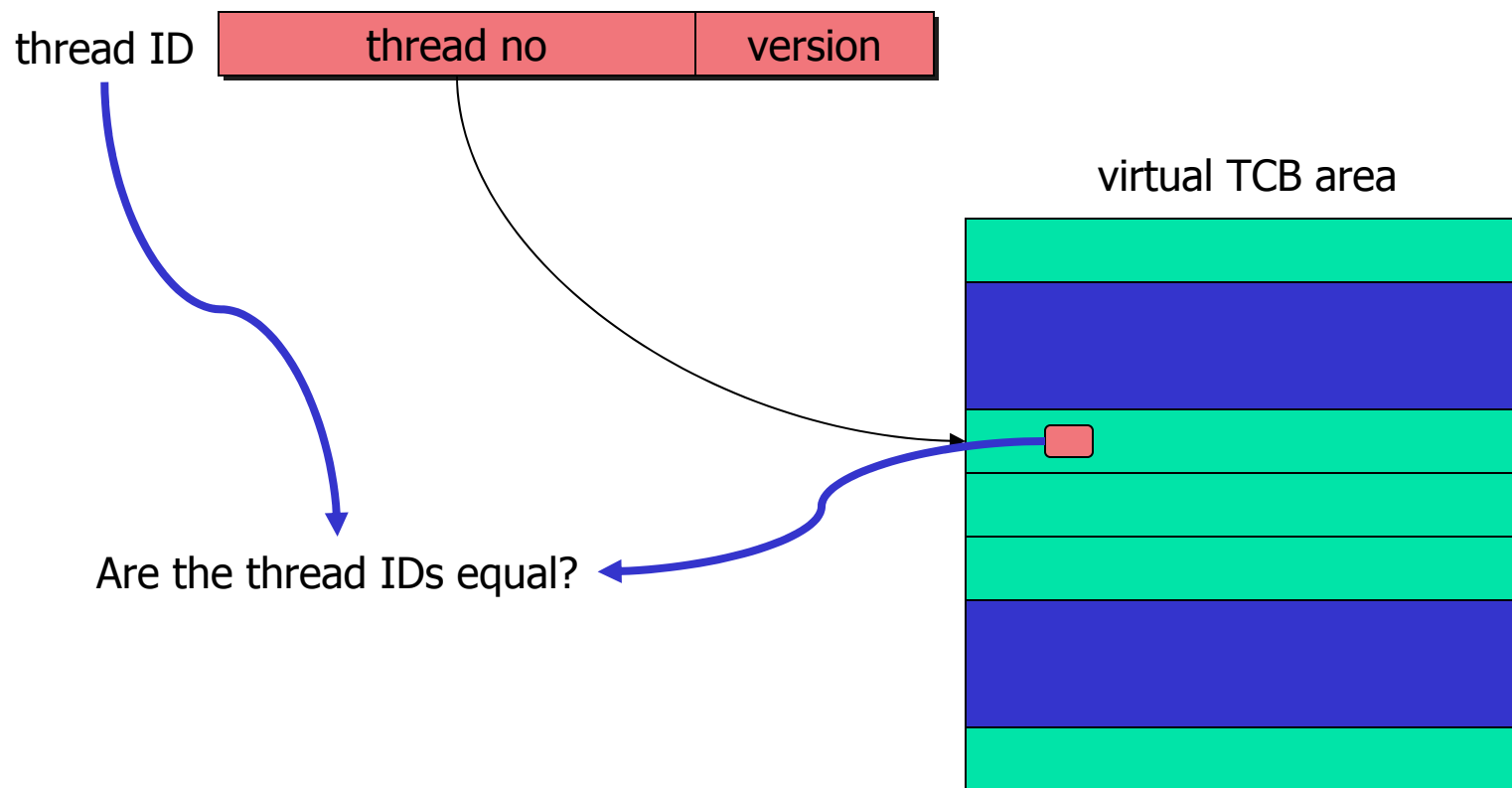
TLB Problem



Avoid Table Lookups



Validate Thread ID



Branch Elimination

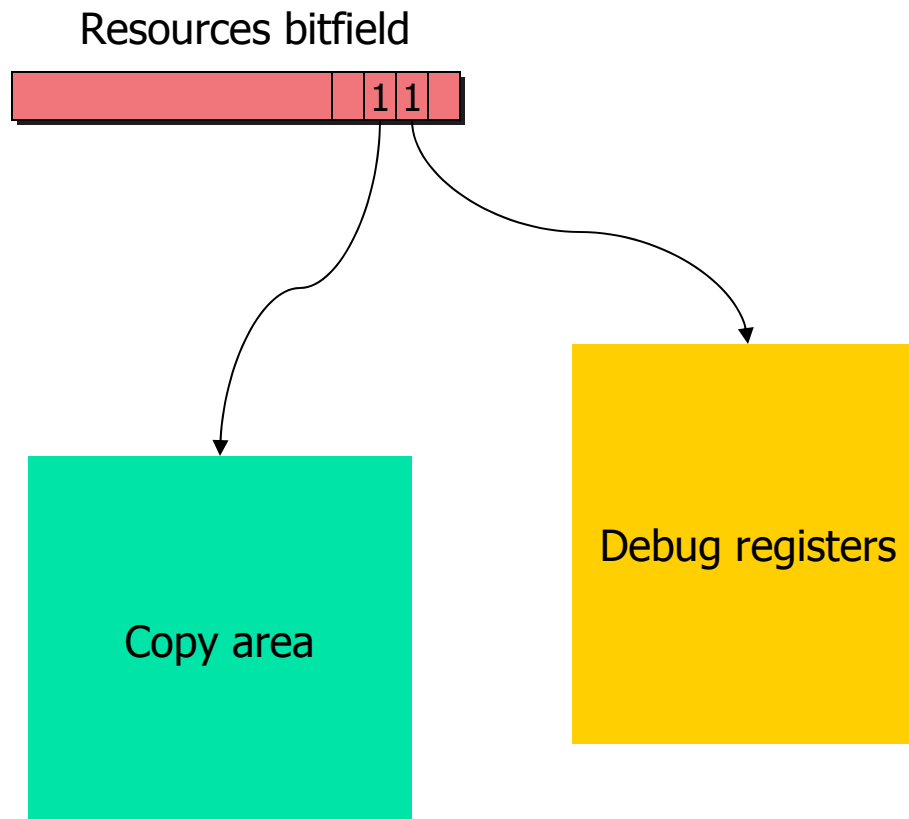
```
slow = ~receiver->thread_state +  
      (timeouts & 0xffff) +  
      sender->resources +  
      receiver->resources;  
  
if( slow )  
    enter_slow_path()
```

Common case: -1

Common case: 0

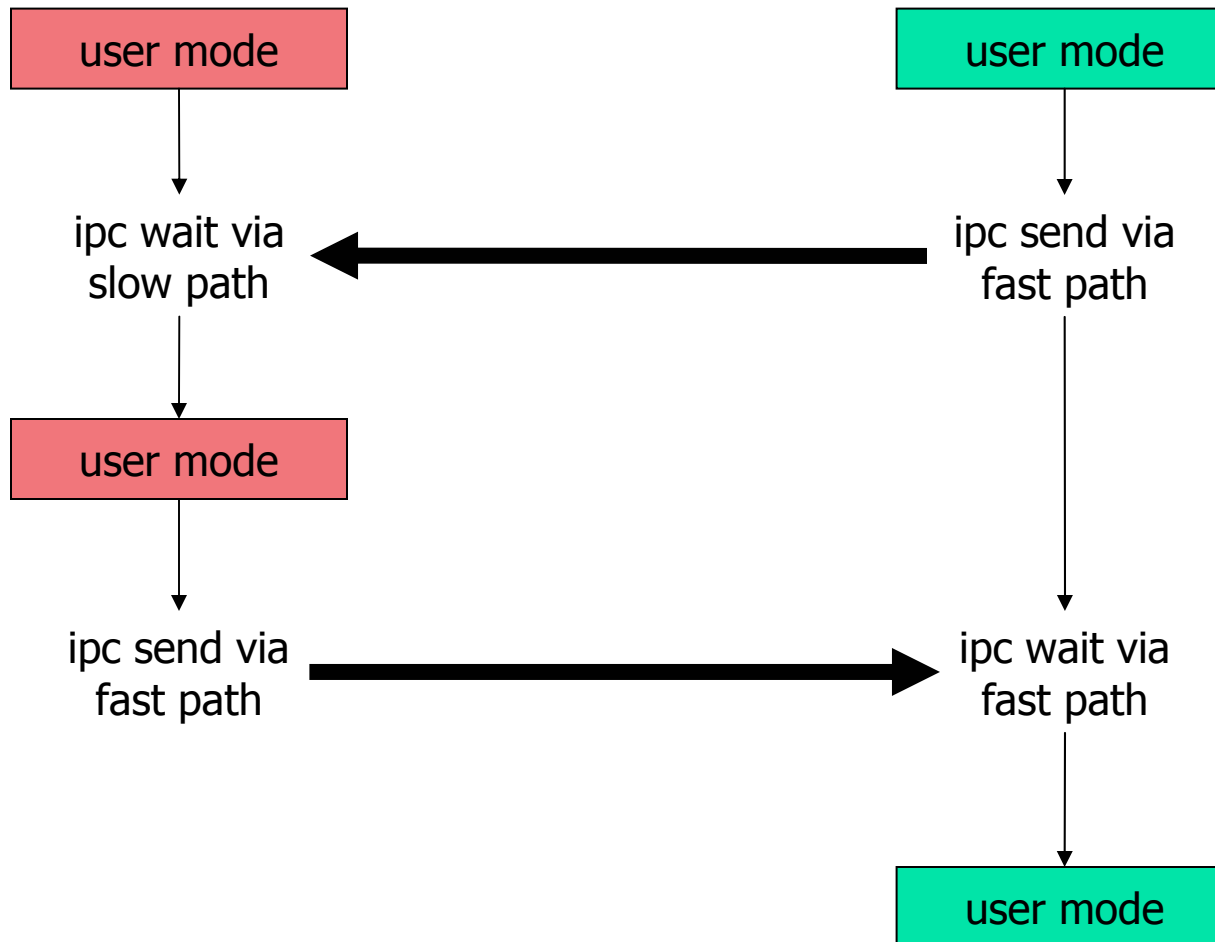
- Reduces branch prediction foot print.
- Avoids mispredicts & stalls & flushes.
 - Increases latency for slow path

TCB Resources



- One bit per resource
- Fast path checks entire word
 - if not 0, jump to resource handlers

Slow & Fast



Consistent State

- Cooperative thread scheduling in kernel
- TCB in consistent state for IPC wait
 - IPC restores user mode context
 - avoids cycles for restoring kernel context
 - fast path can activate slow path TCB



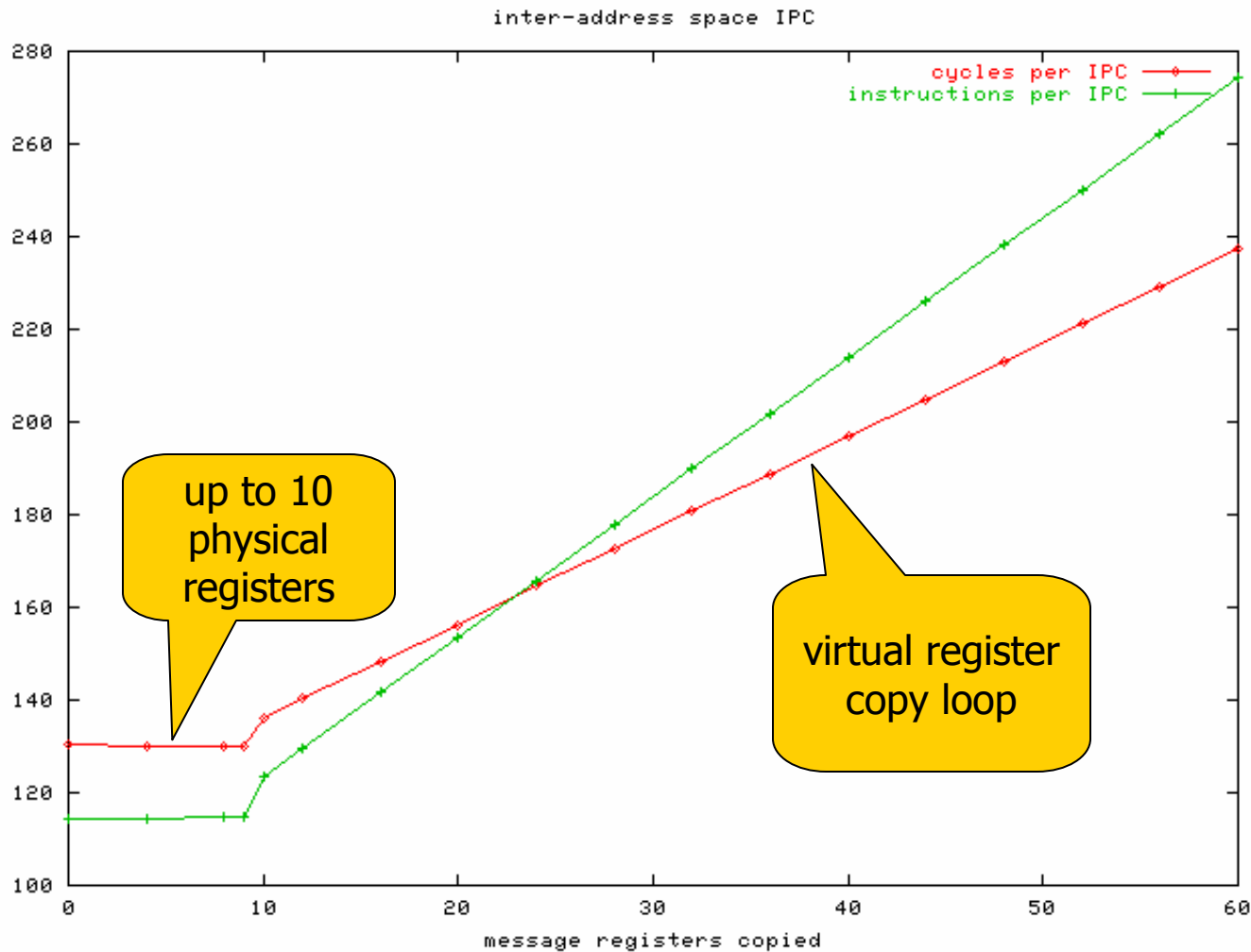
Problem?

Can't use fast path for **kernel threads**.

How often do kernel threads use IPC?

How to efficiently detect kernel threads?

Message Transfer

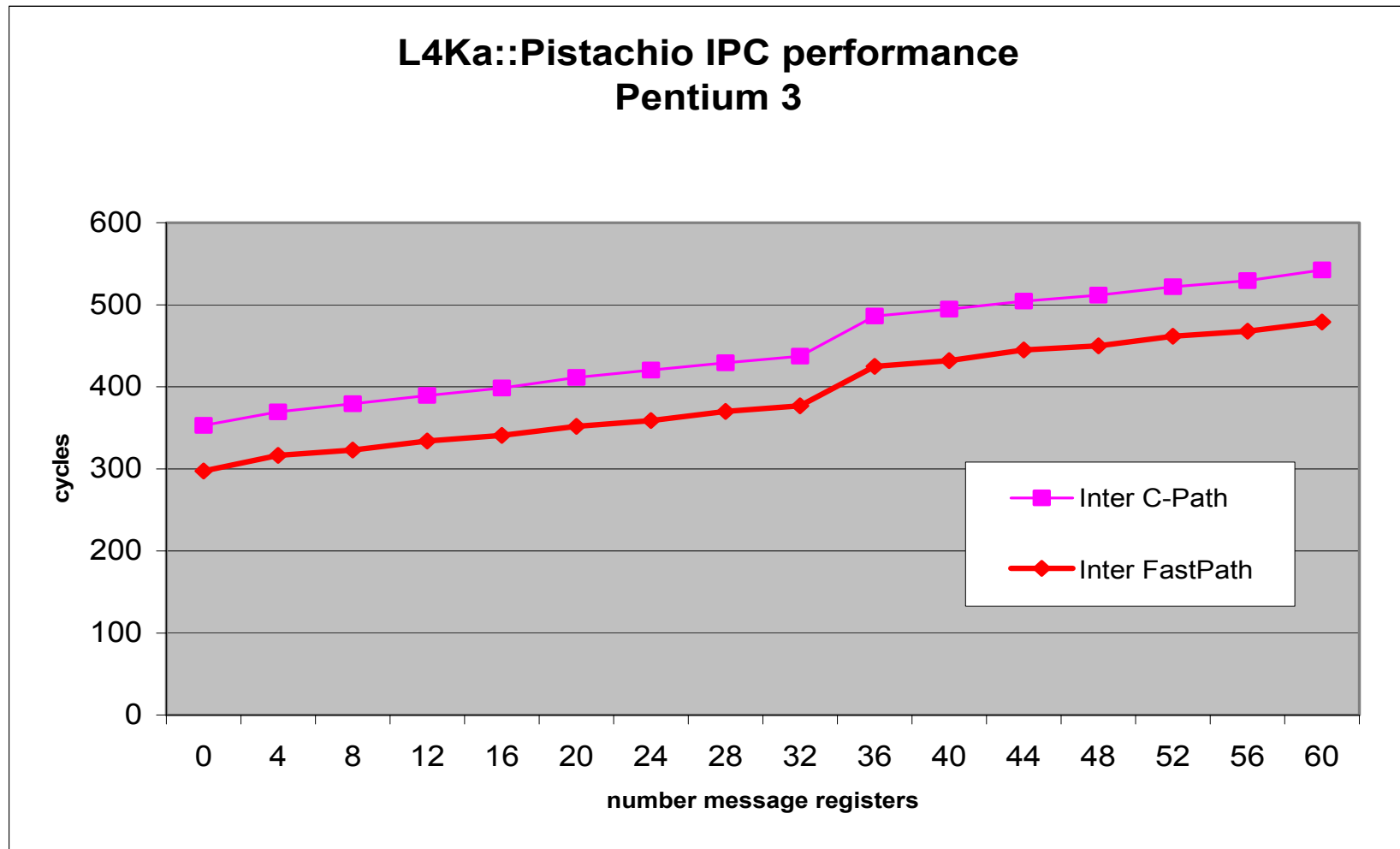


IBM PowerPC 750,
500 MHz,
32 registers

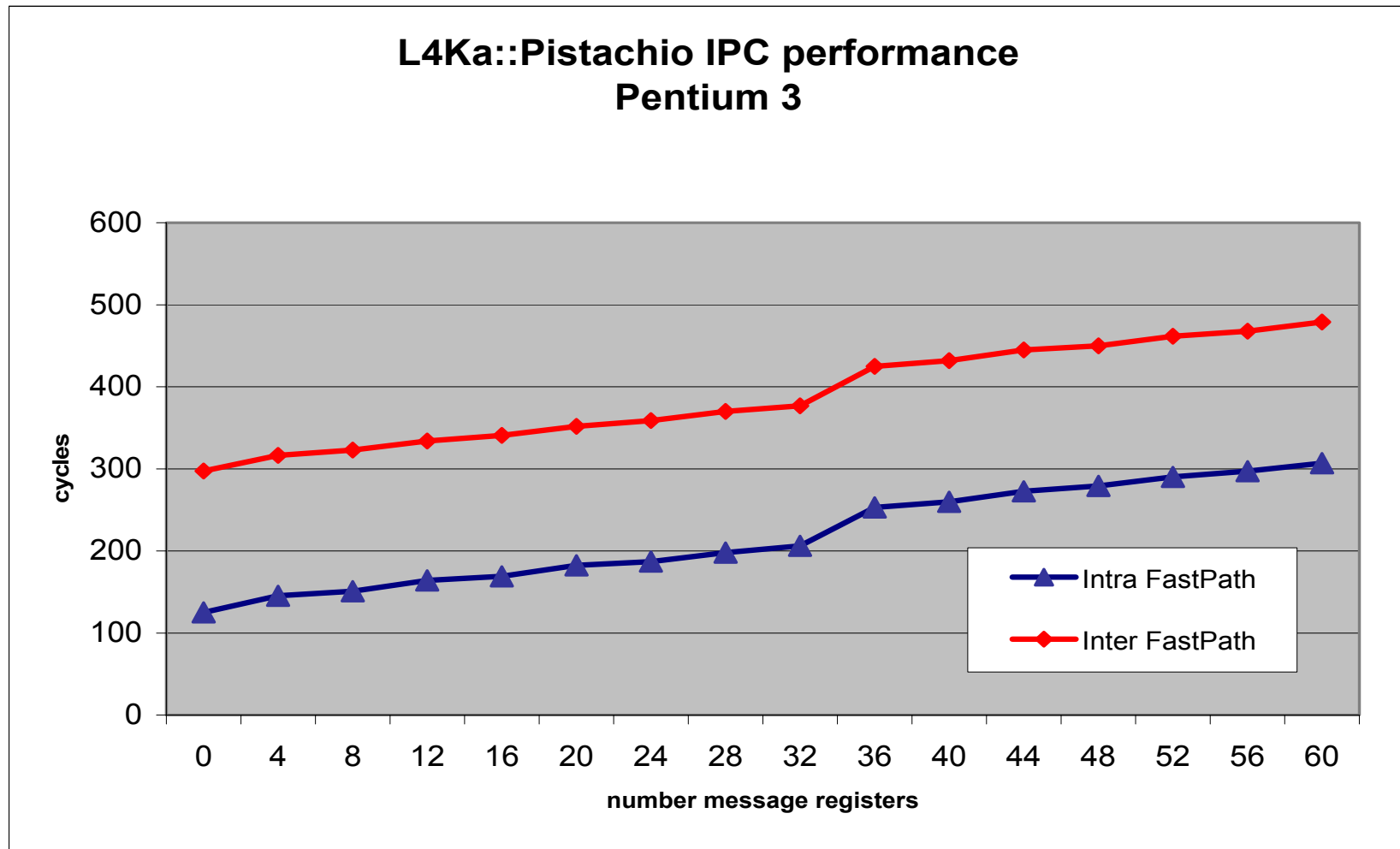
Many cycles
wasted on pipe
flushes for
privileged
instructions.



Slow Path vs. Fast Path



Inter vs. Intra Address Space



IPC - Implementation

Long IPC



Long IPC (uniprocessor)

- system-call preamble (disable intr)
- identify dest thread and check
 - same chief
 - ready-to-receive?
- analyze msg and transfer
 - long/map:

Preemptions possible!
(end of timeslice, device interrupt...)

Pagefaults possible!
(in source and dest address space)

■ ***– transfer message –***

- switch to dest thread & address space
- system-call postamble



Long IPC (uniprocessor)

- system-call pre (disable intr)
- identify dest thread and check
 - same chief
 - ready-to-receive?
- analyze msg and transfer
 - long/map:
 - **lock both partners**
 - **– transfer message –**
 - **unlock both partners**
- switch to dest thread & address space
- system-call post

Preemptions possible!
(end of timeslice, device interrupt...)

Pagefaults possible!
(in source and dest address space)



Long IPC (uniprocessor)

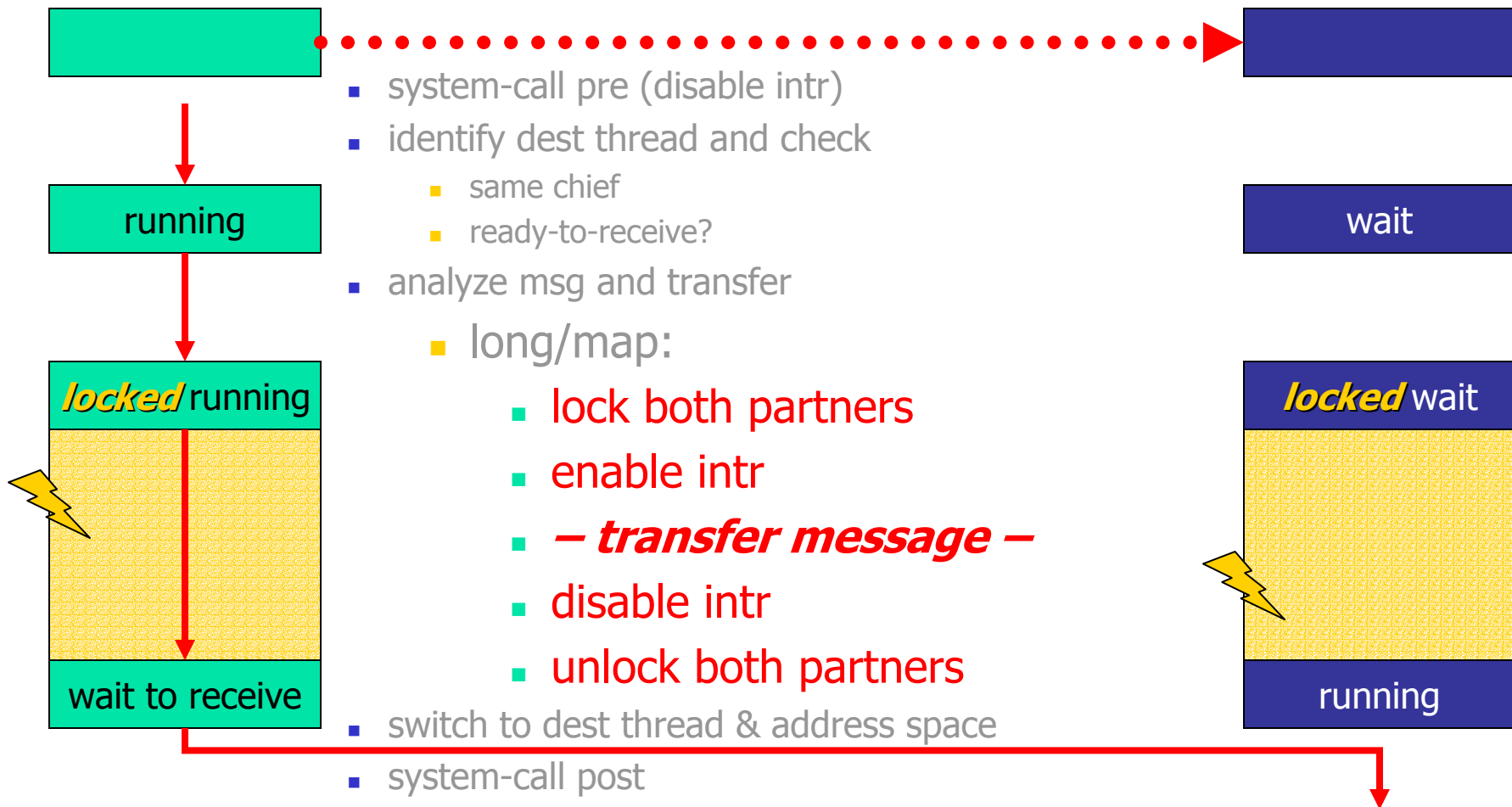
- system-call pre (disable intr)
- identify dest thread and check
 - same chief
 - ready-to-receive?
- analyze msg and transfer
 - long/map:
 - lock both partners
 - enable intr
 - **– transfer message –**
 - disable intr
 - unlock both partners
- switch to dest thread & address space
- system-call post

Preemptions possible!
(end of timeslice, device interrupt...)

Pagefaults possible!
(in source and dest address space)

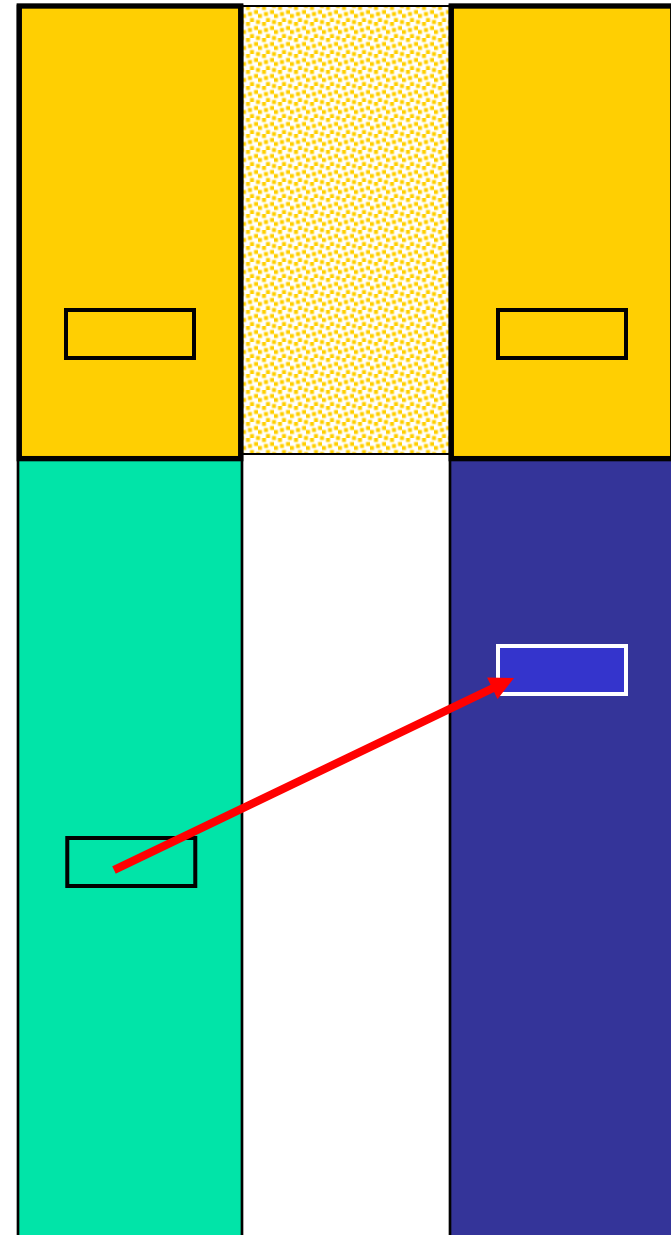


Long IPC (uniprocessor)



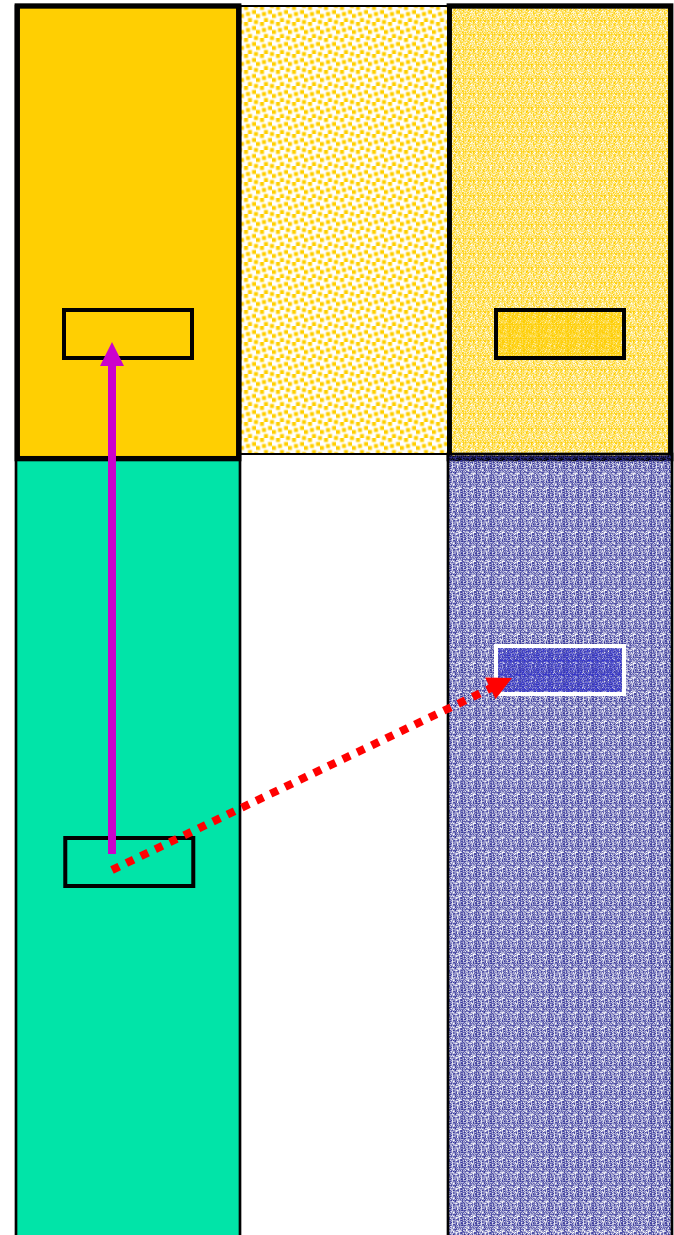
IPC - mem copy

- Why is it needed? Why not share?
 - Trust
 - Need own copy
 - Granularity
 - Object small than a page or not aligned



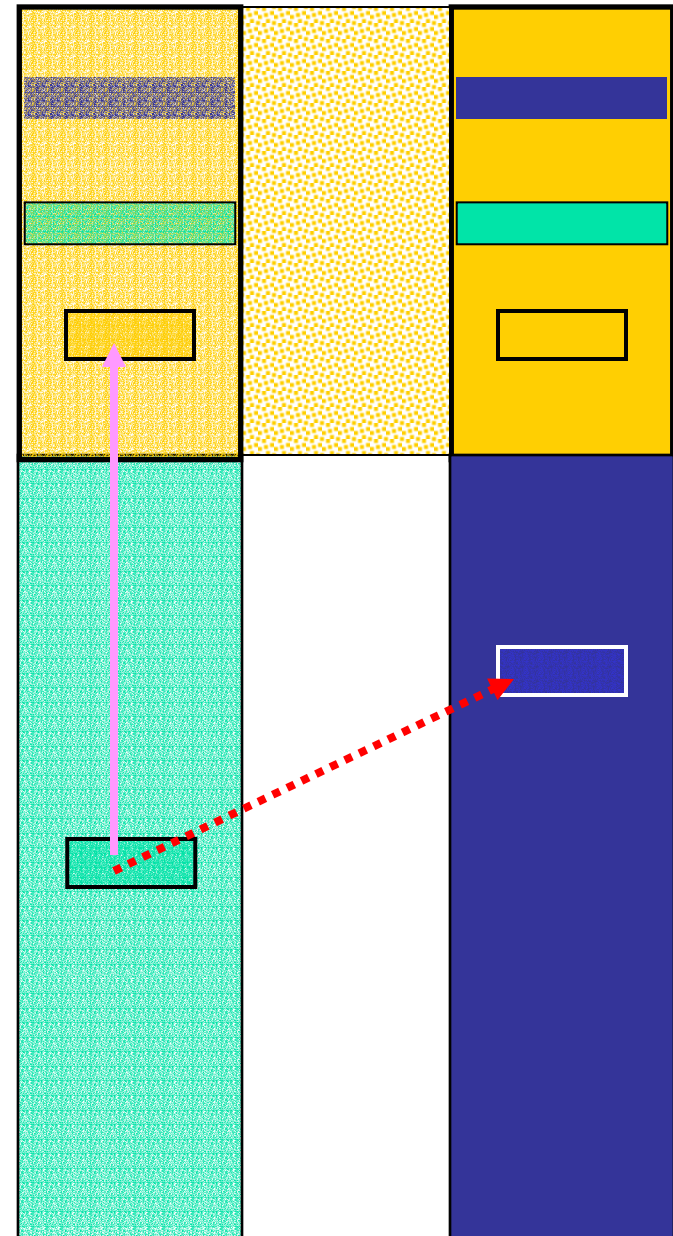
copy in - copy out

- copy into kernel buffer



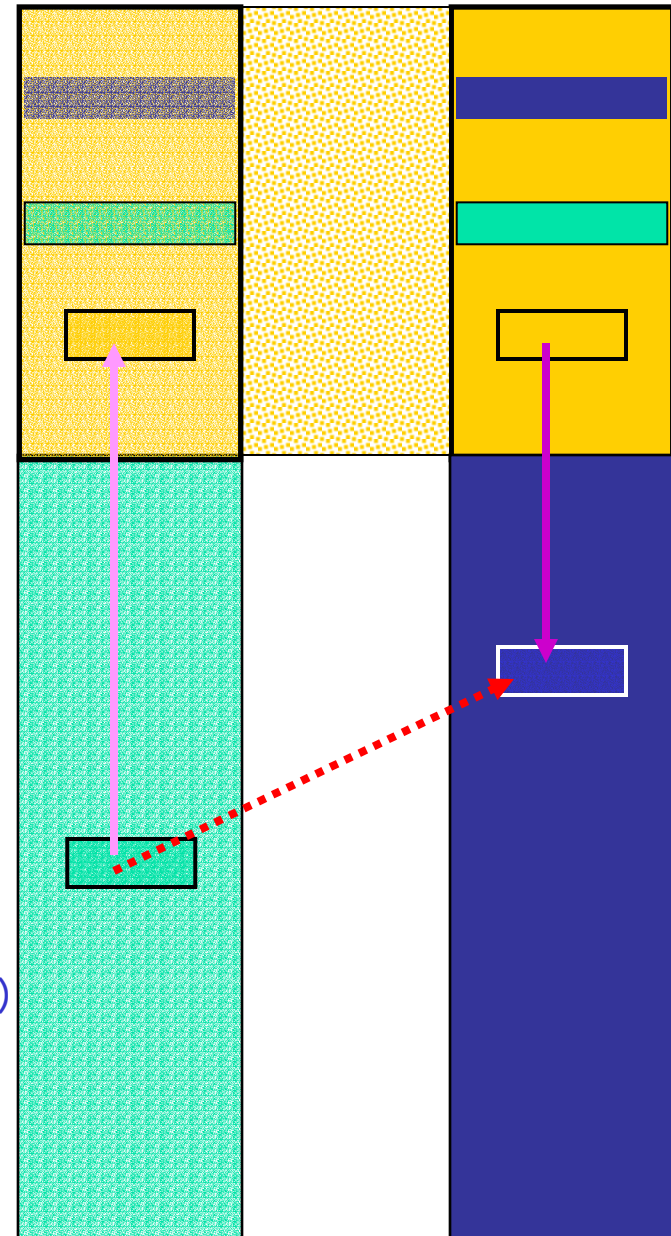
copy in - copy out

- copy into kernel buffer
- switch spaces

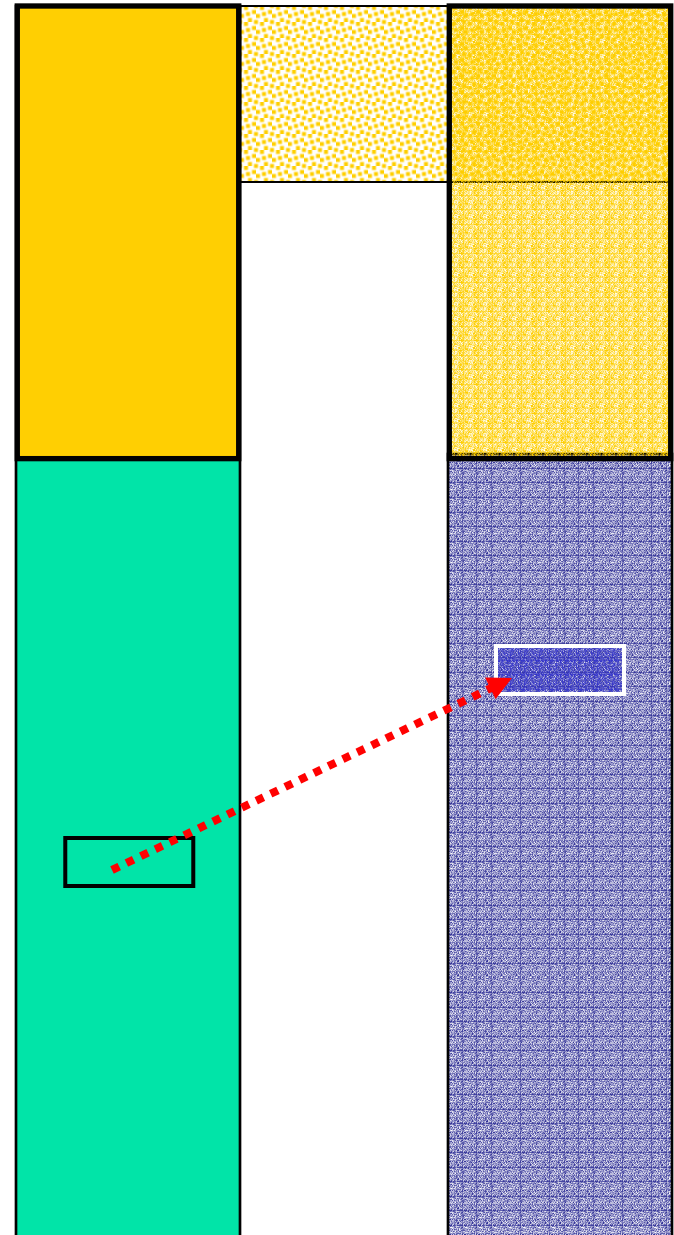


copy in - copy out

- copy into kernel buffer
 - switch spaces
 - copy out of kernel buffer
-
- costs for n words
 - $2 \times 2n$ r/w operations
 - $3 \times n/8$ cache lines
 - $1 \times n/8$ overhead cache misses (small n)
 - $4 \times n/8$ cache misses (large n)

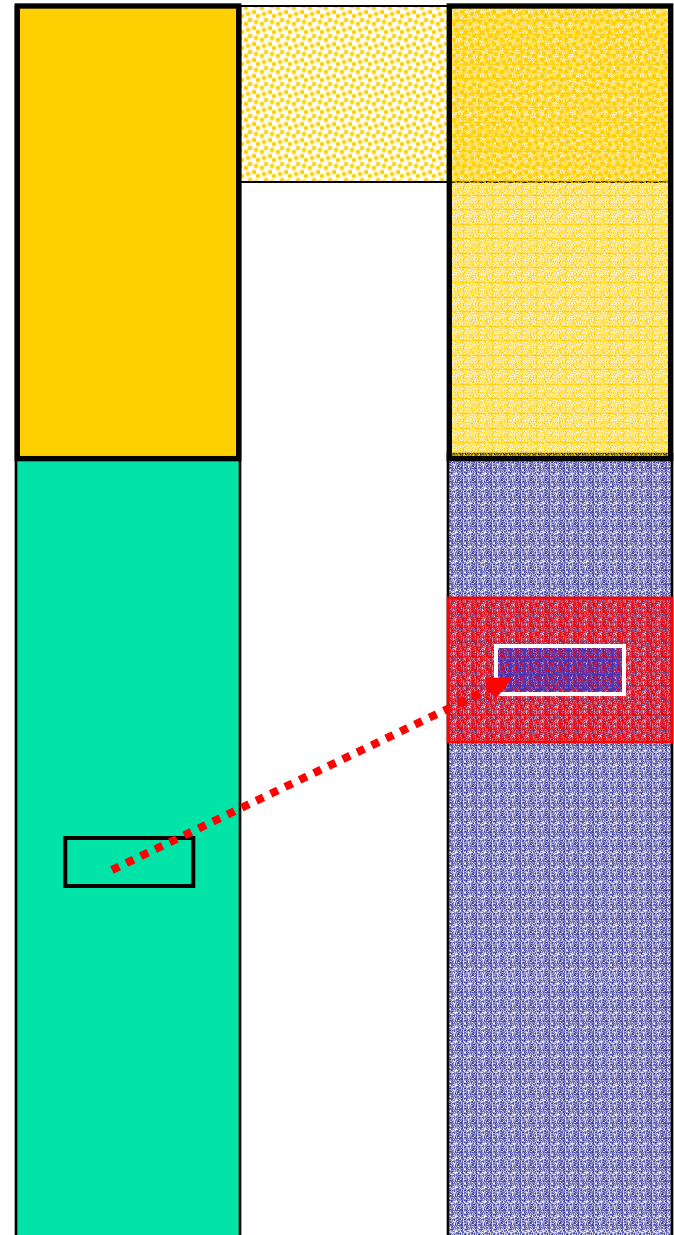


temporary mapping



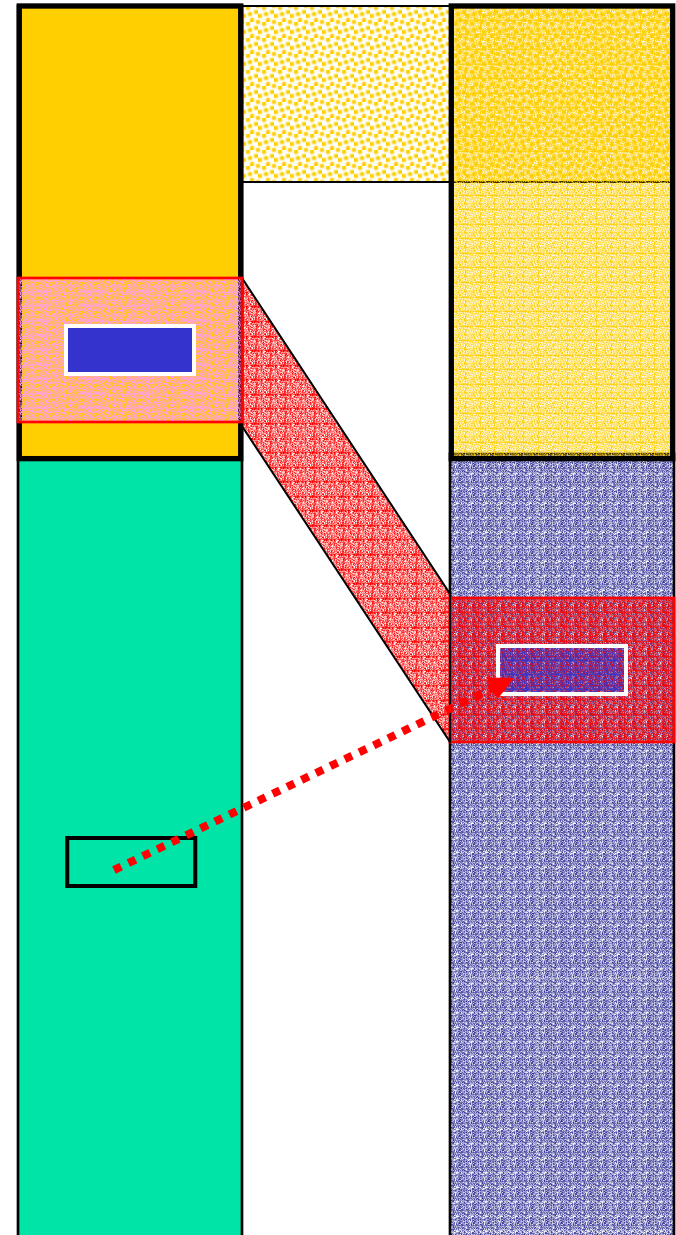
temporary mapping

- select dest area (4+4 M)



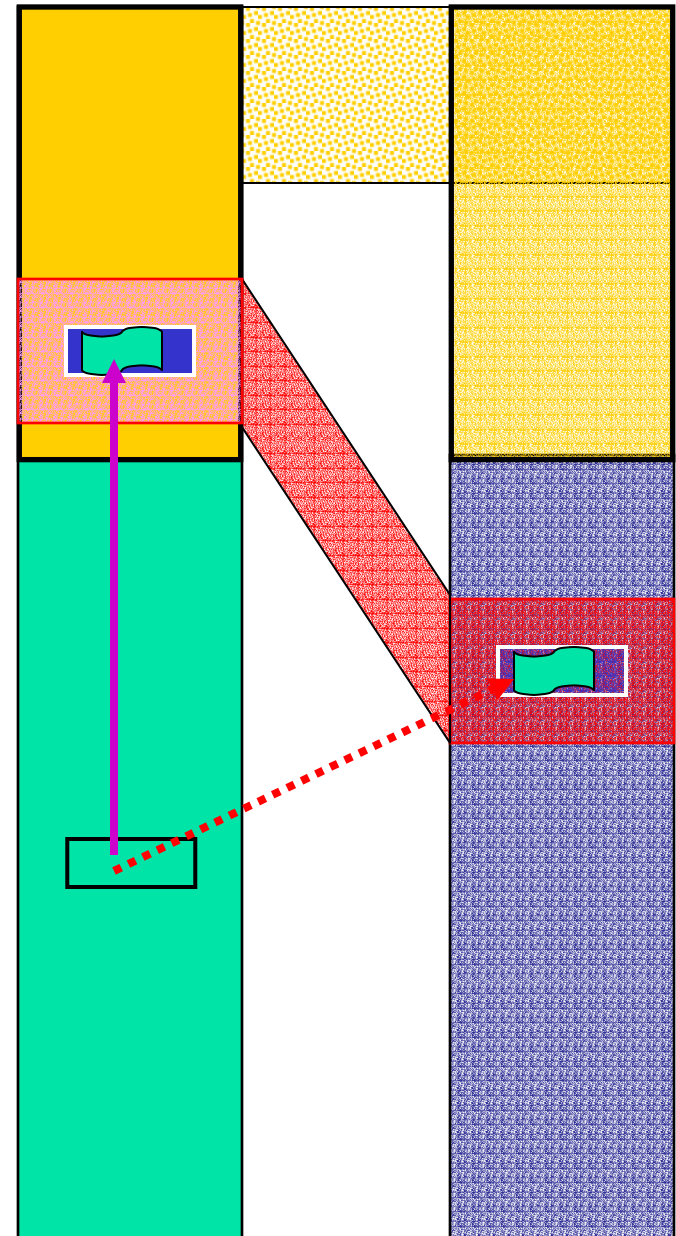
temporary mapping

- select dest area (4+4 M)
- map into source AS (kernel)



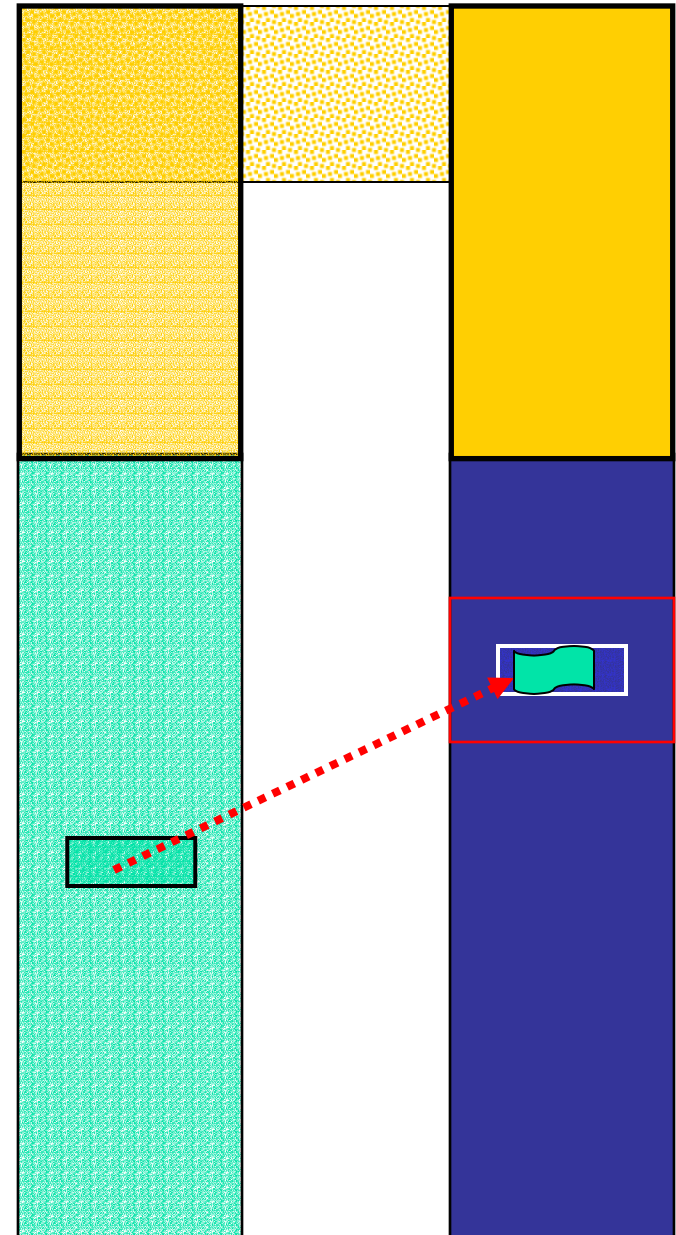
temporary mapping

- select dest area (4+4 M)
- map into source AS (kernel)
- copy data

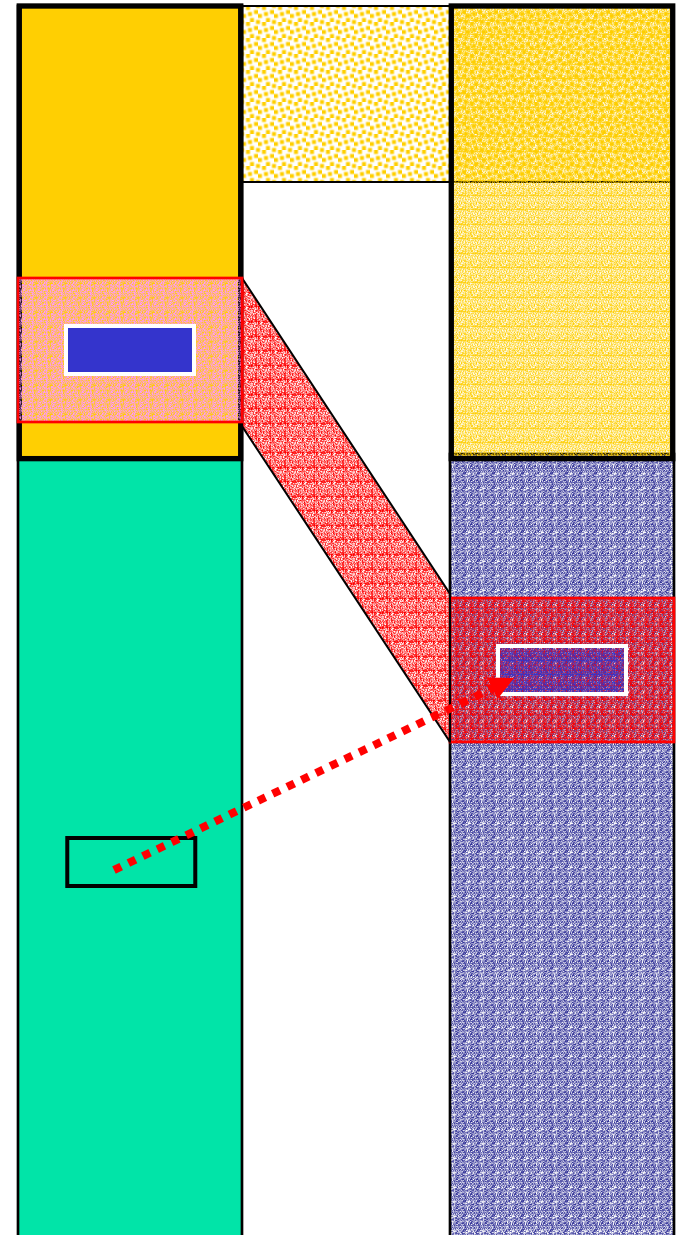
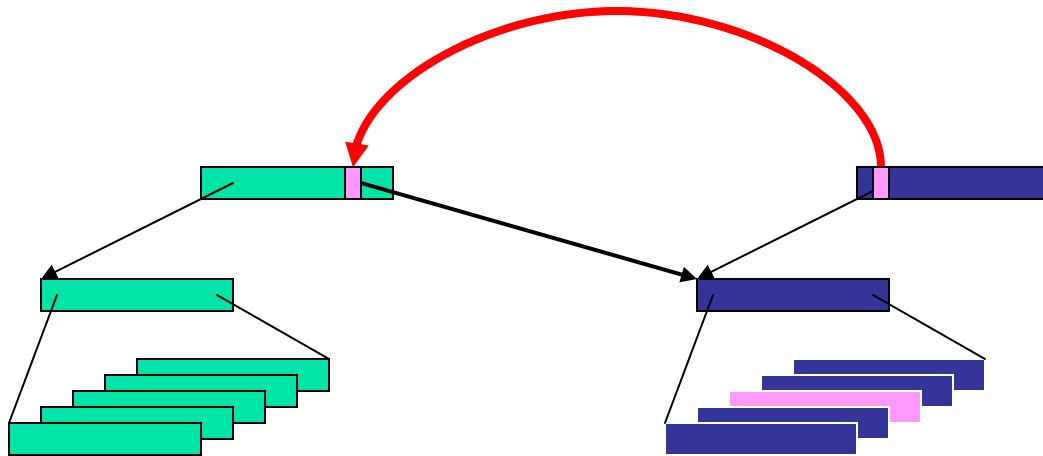


temporary mapping

- select dest area (4+4 M)
- map into source AS (kernel)
- copy data
- switch to dest space

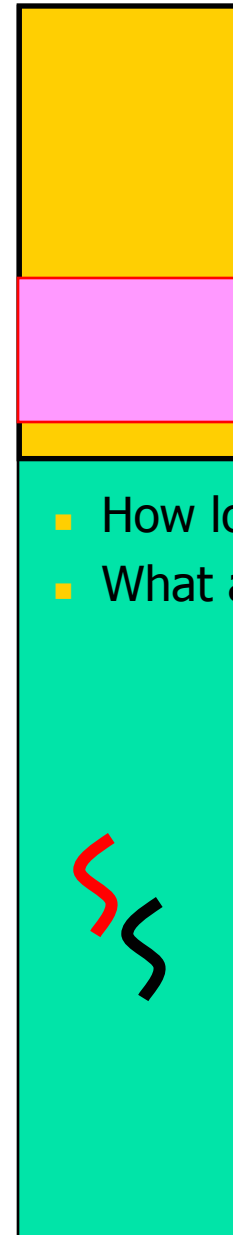
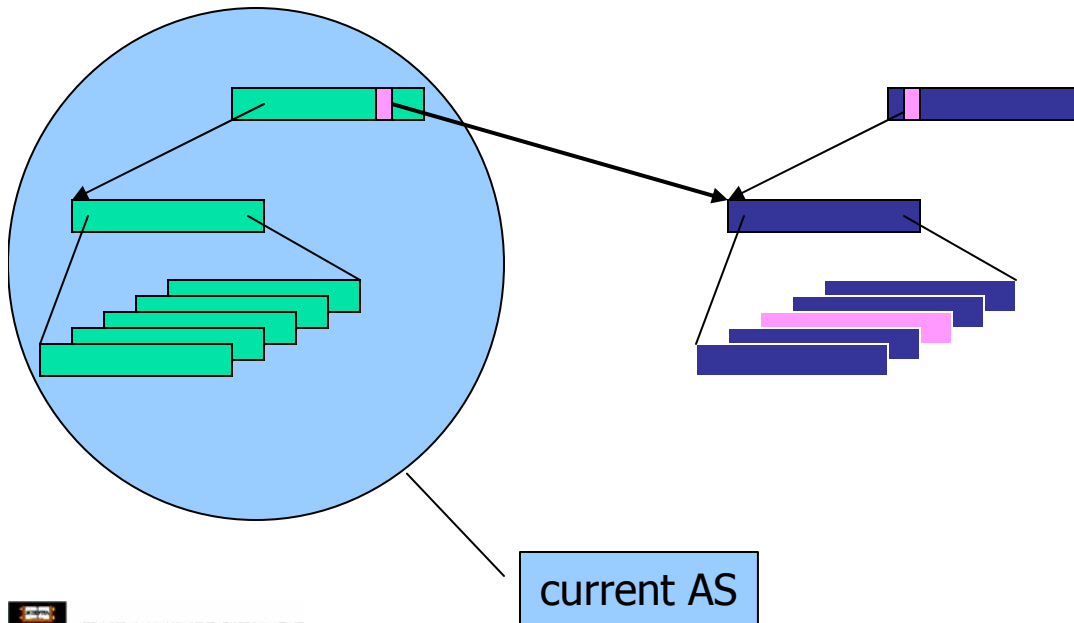


temporary mapping



temporary mapping

- problems
 - multiple threads per AS
 - mappings might change while message is copied

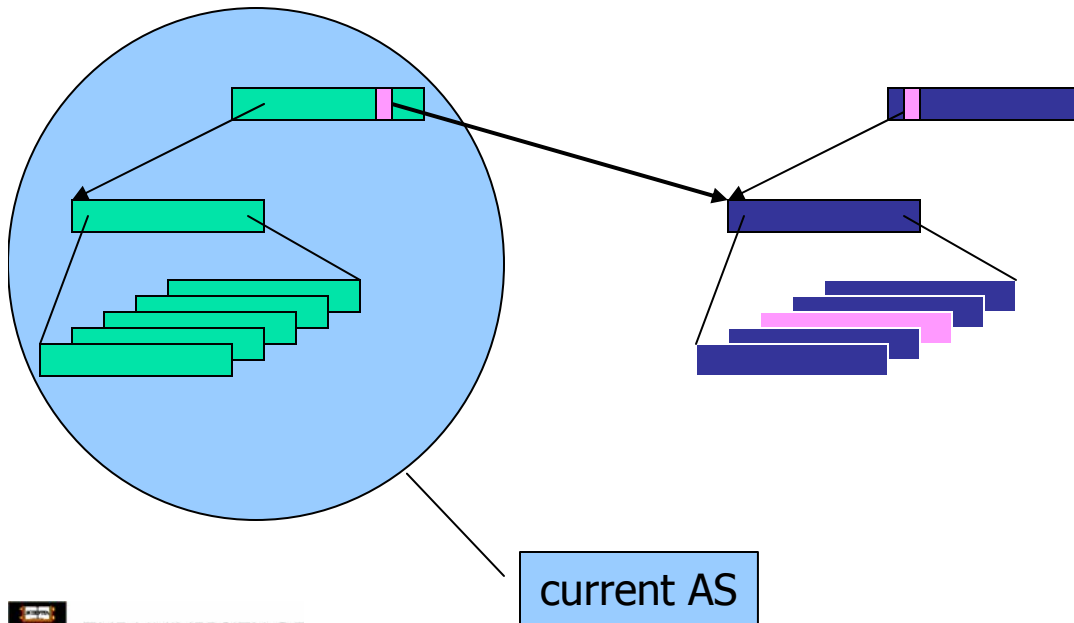


- How long to keep PTE?
- What about TLB?

temporary mapping

- invalidate PTE
- flush TLB

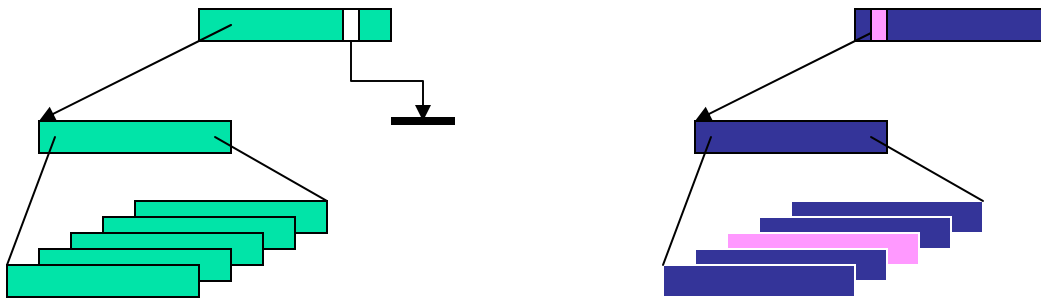
- when leaving curr thread *during* ipc?



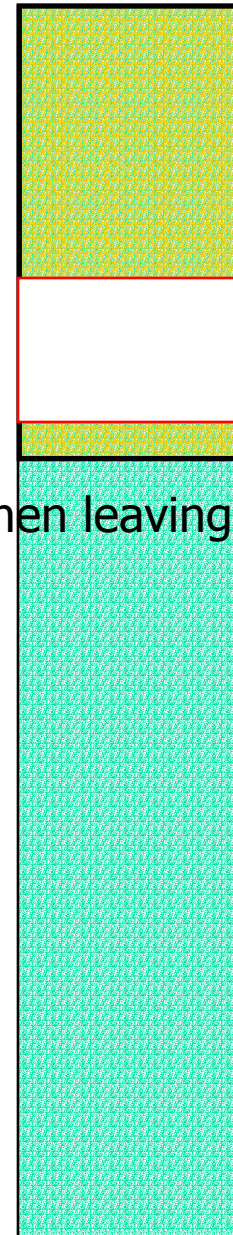
temporary mapping

- invalidate PTE
- flush TLB

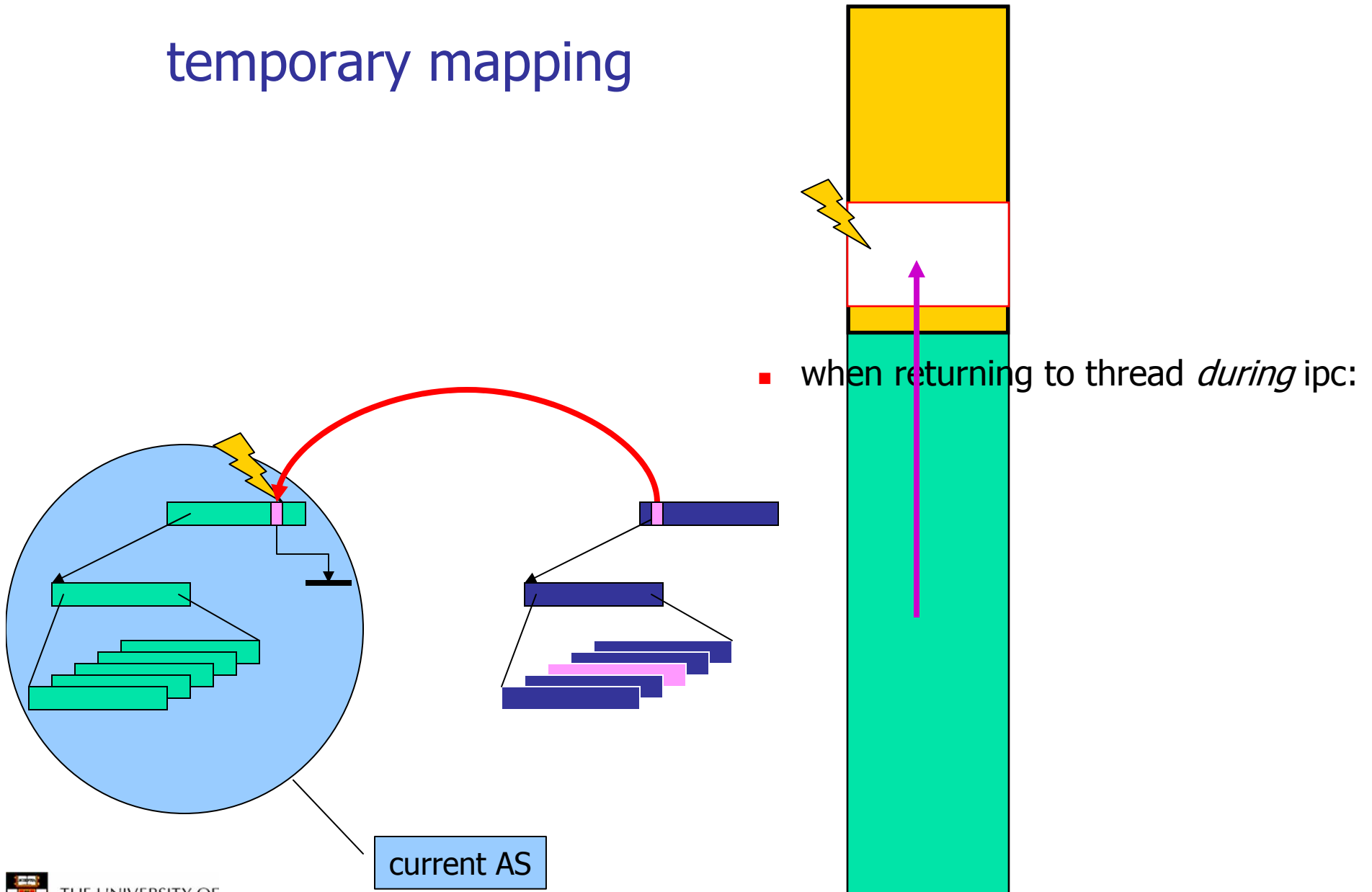
- when leaving curr thread *during* ipc:



current AS



temporary mapping

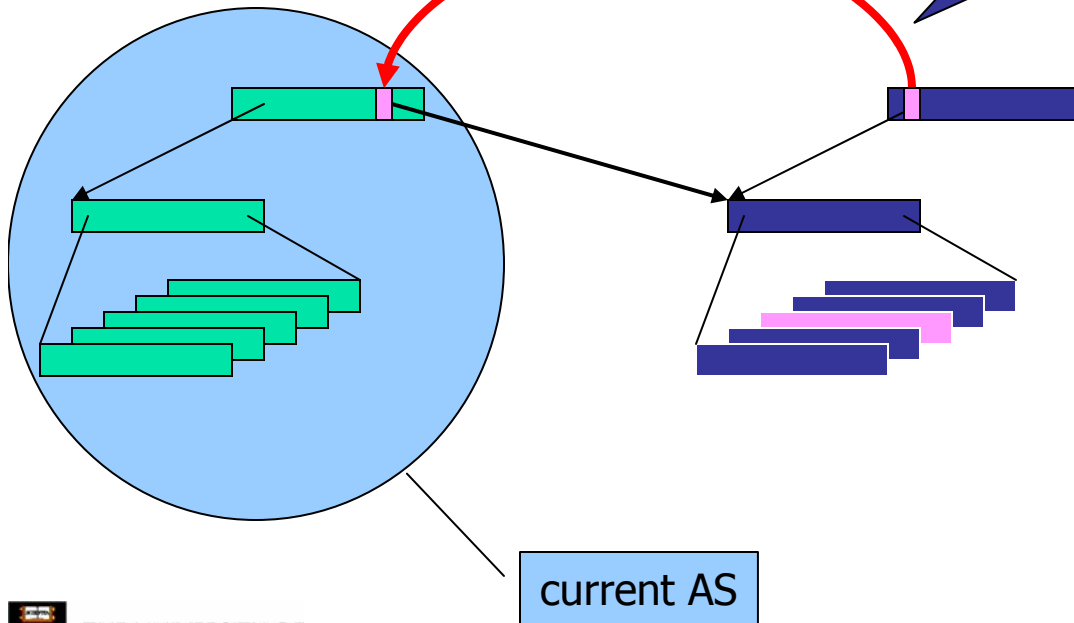


temporary mapping

Reestablishing temp mapping requires to store **partner id** and **dest area address** in the sender's tcb.

Note: receiver's page mappings might have changed !

when returning to thread *during* ipc:



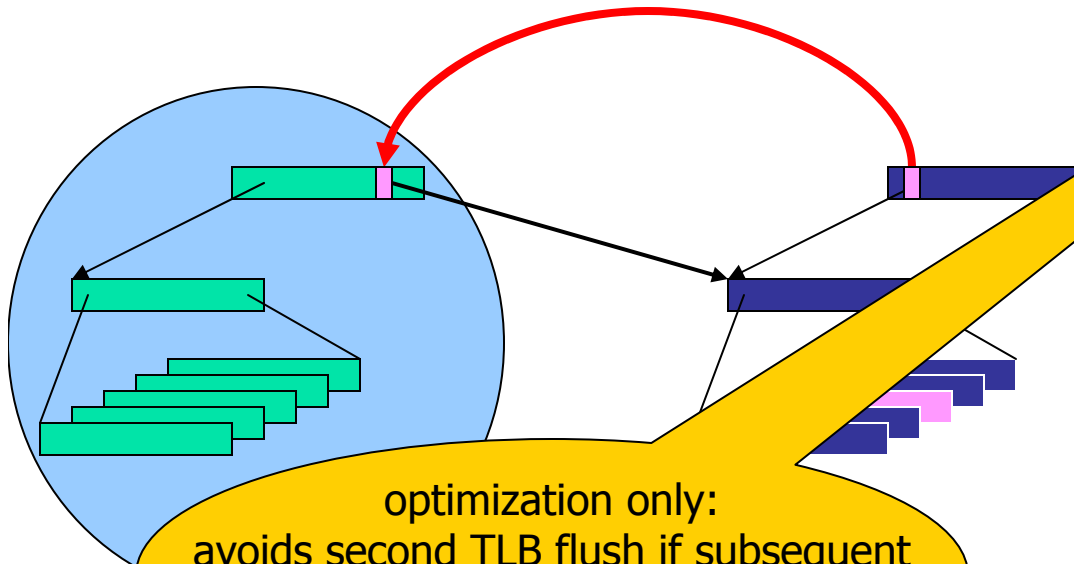
temporary mapping

Start temp mapping:

```
mytcb.partner := partner ;  
mytcb.waddr := dest 8M area base ;  
myPDE.TMarea := destPDE.destarea .
```

Leave thread:

```
if mytcb.waddr ≠ nil then  
  myPDE.TMarea := nil ;  
  if dest AS = my AS then  
    flush TLB  
  fi  
fi
```



optimization only:
avoids second TLB flush if subsequent
thread switch would flush TLB
anyhow

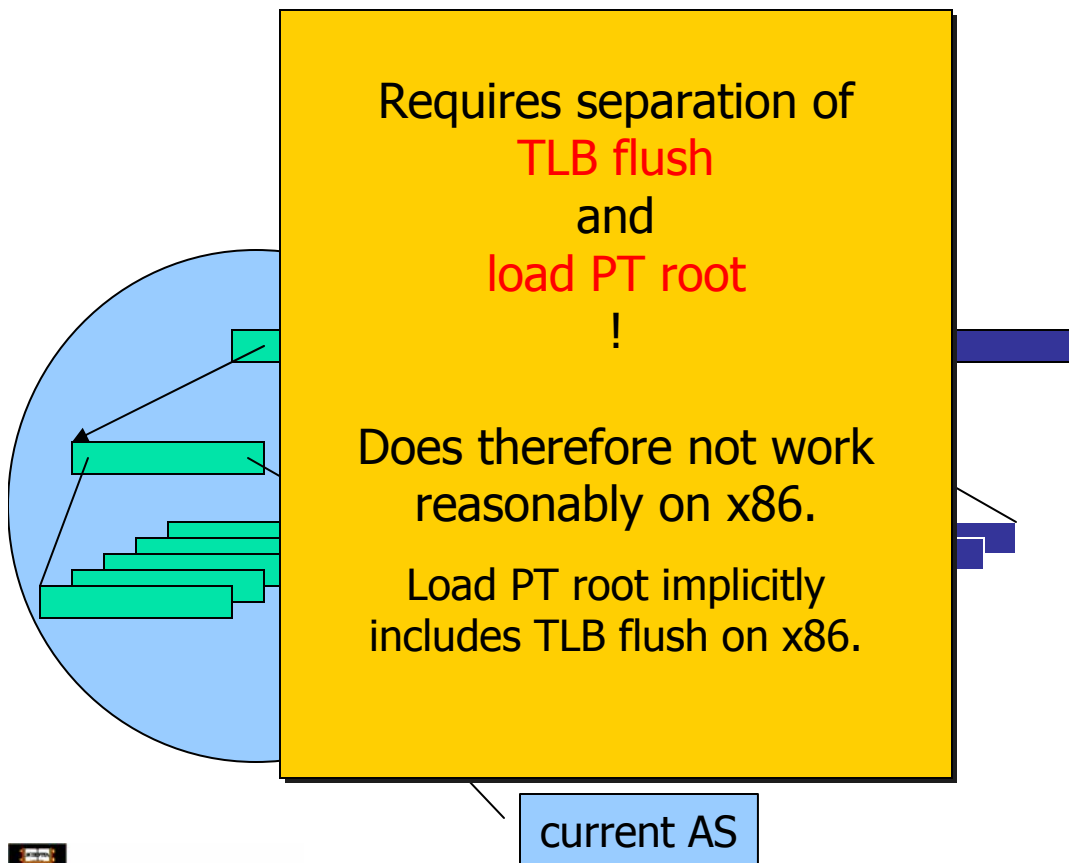
Close temp mapping:

```
mytcb.waddr := nil .
```

```
myPDE.TMarea := nil ??
```

temporary mapping

■ Alternative method:



Leave thread:

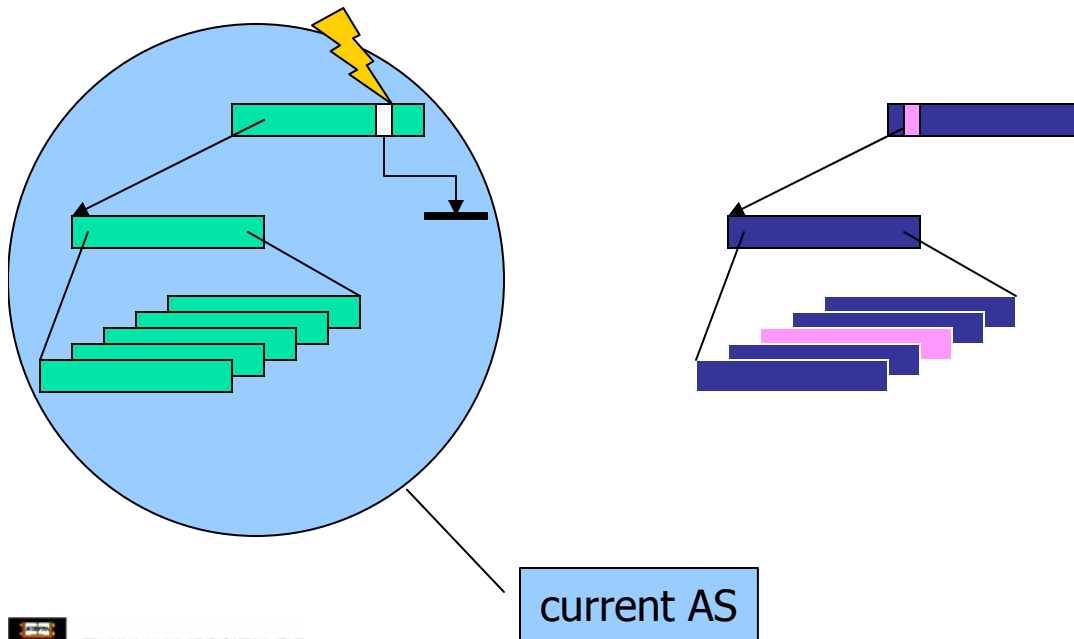
```
if mytcb.waddr  $\neq$  nil then  
    myPDE.TMarea := nil ;  
    flush TLB ;  
    TLB flushed := true  
fi .
```

Thread switch :

```
...  
if TLB just flushed  
    then TLB flushed := false  
    else flush TLB  
fi ;  
PT root := ...
```

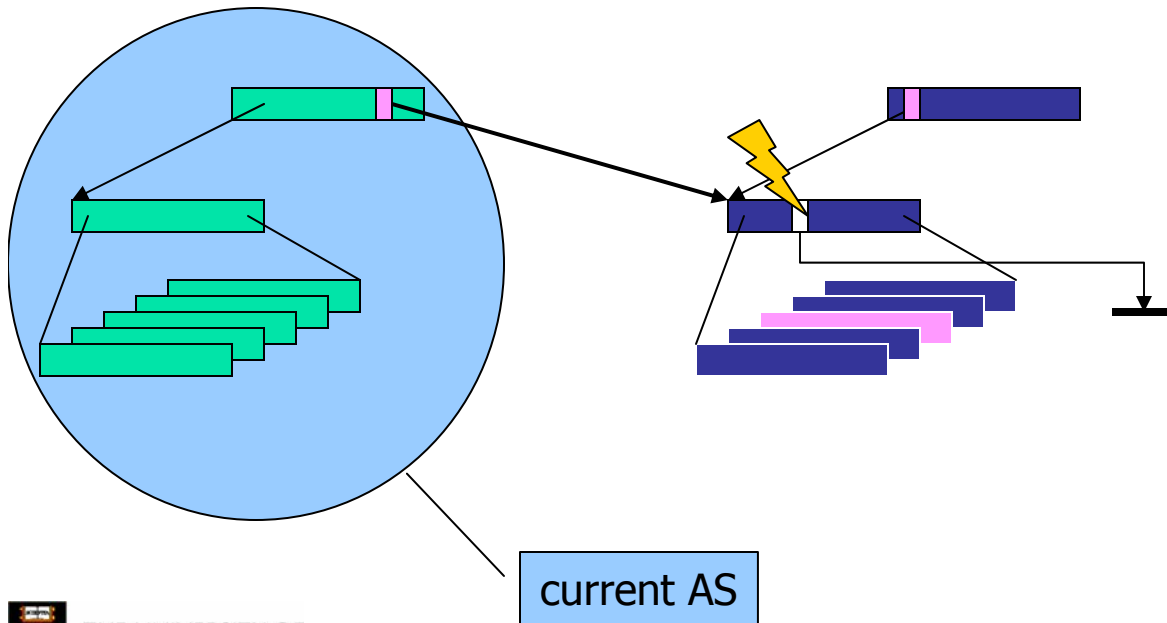
temporary mapping

- **Page Fault Resolution:**



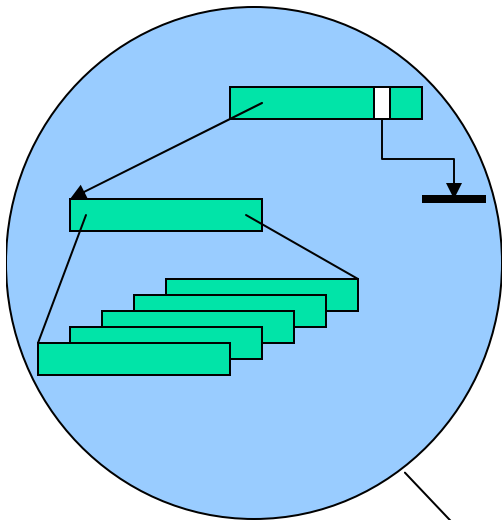
temporary mapping

- **Page Fault Resolution:**

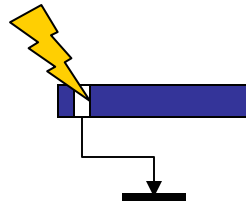


temporary mapping

- **Page Fault Resolution:**



current AS



temporary mapping

■ Page Fault Resolution:

TM area PF:

if myPDE.TMarea = destPDE.destarea **then**

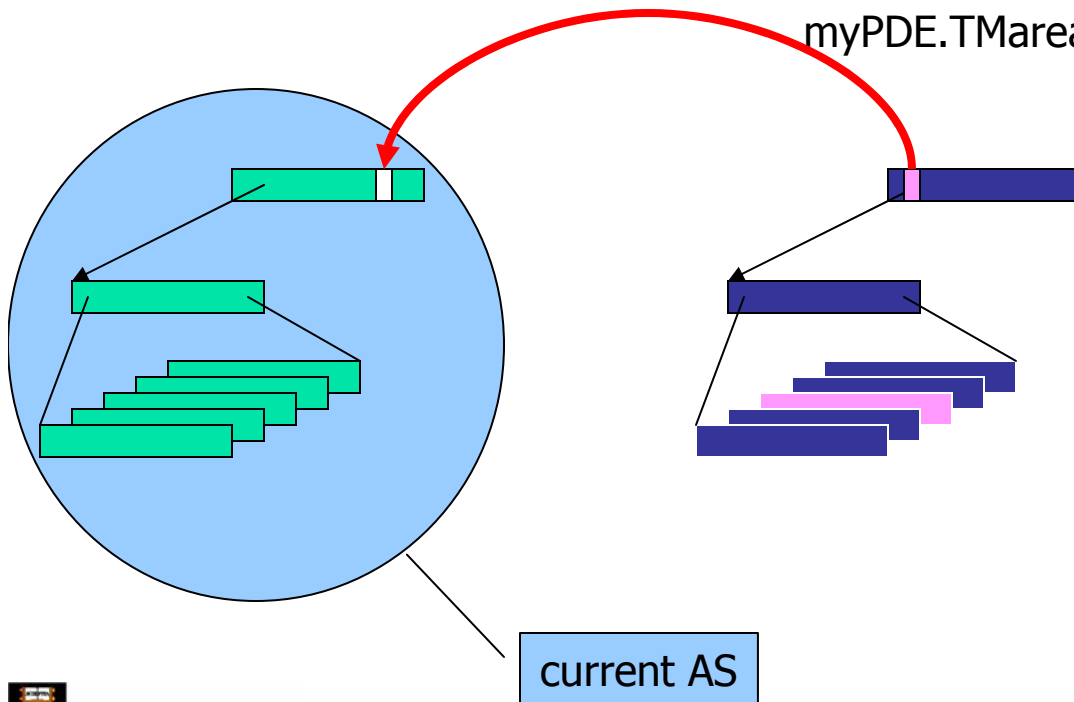
tunnel to (partner) ;

access dest area ;

tunnel to (my)

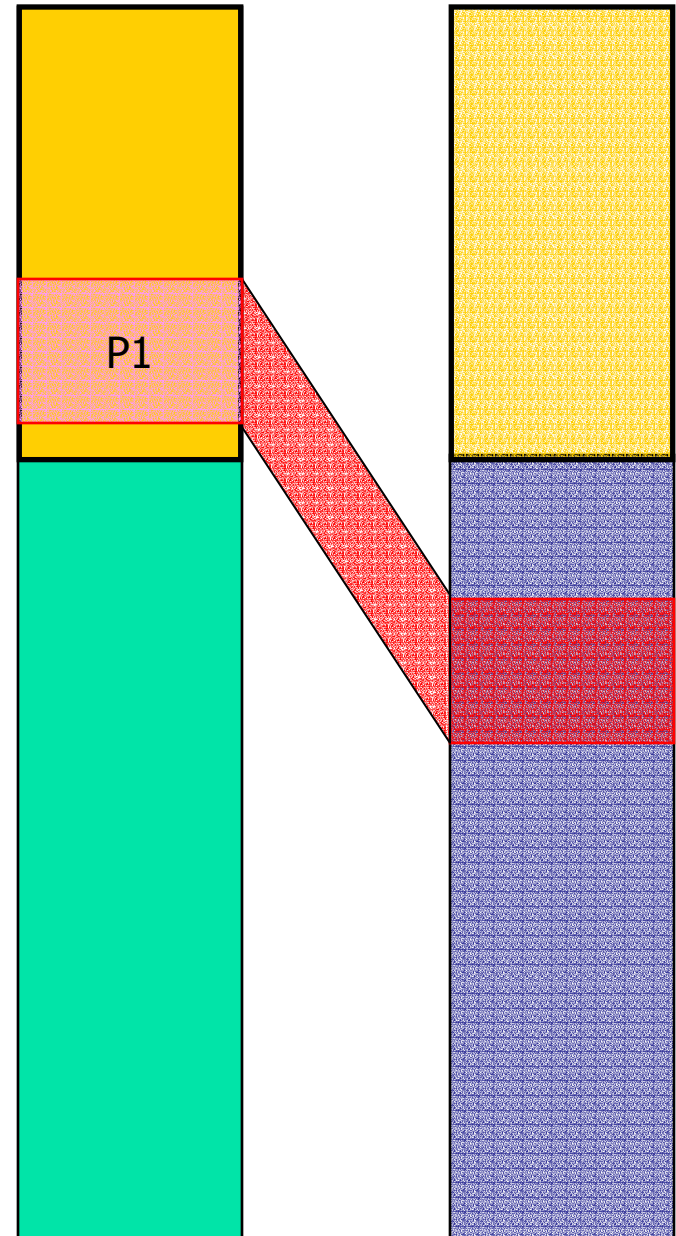
fi ;

myPDE.TMarea := destPDE.destarea .



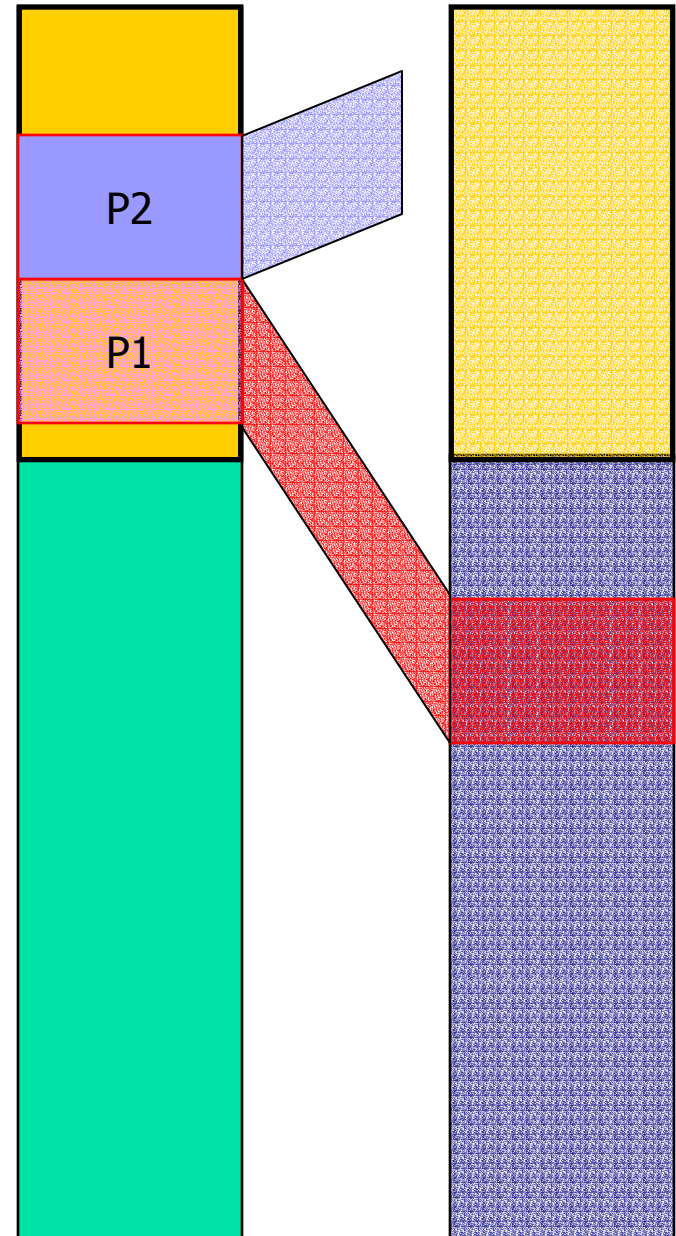
temporary mapping

- SMP



temporary mapping

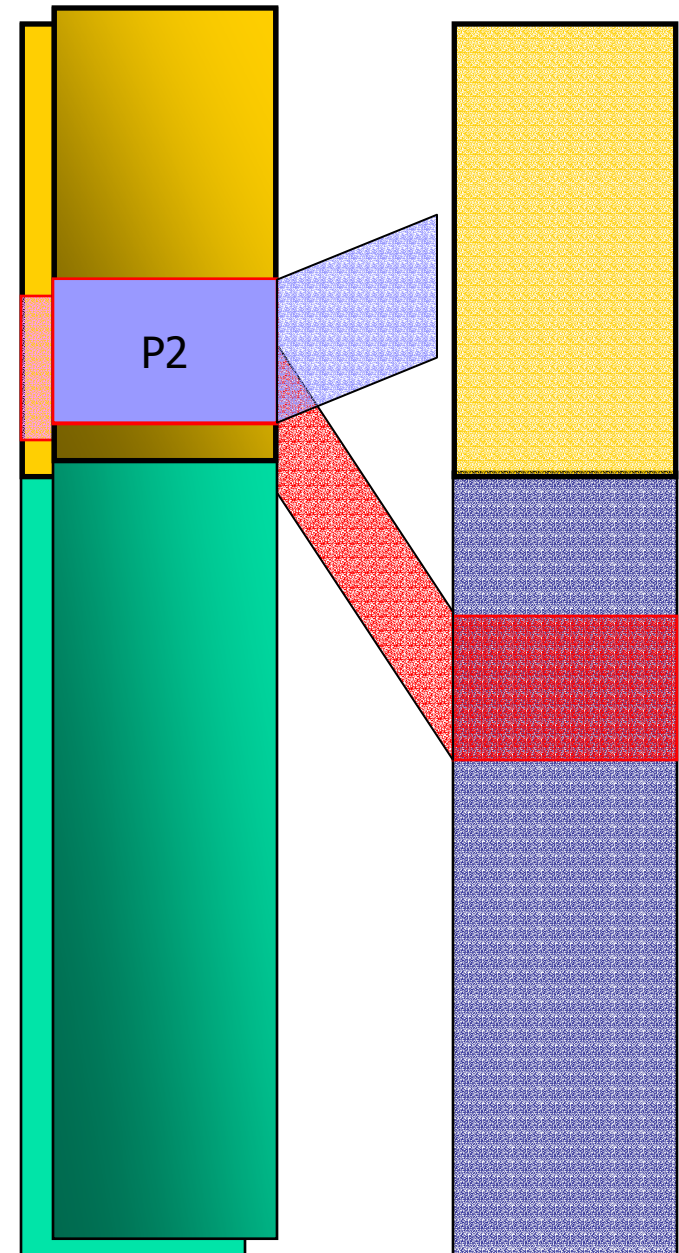
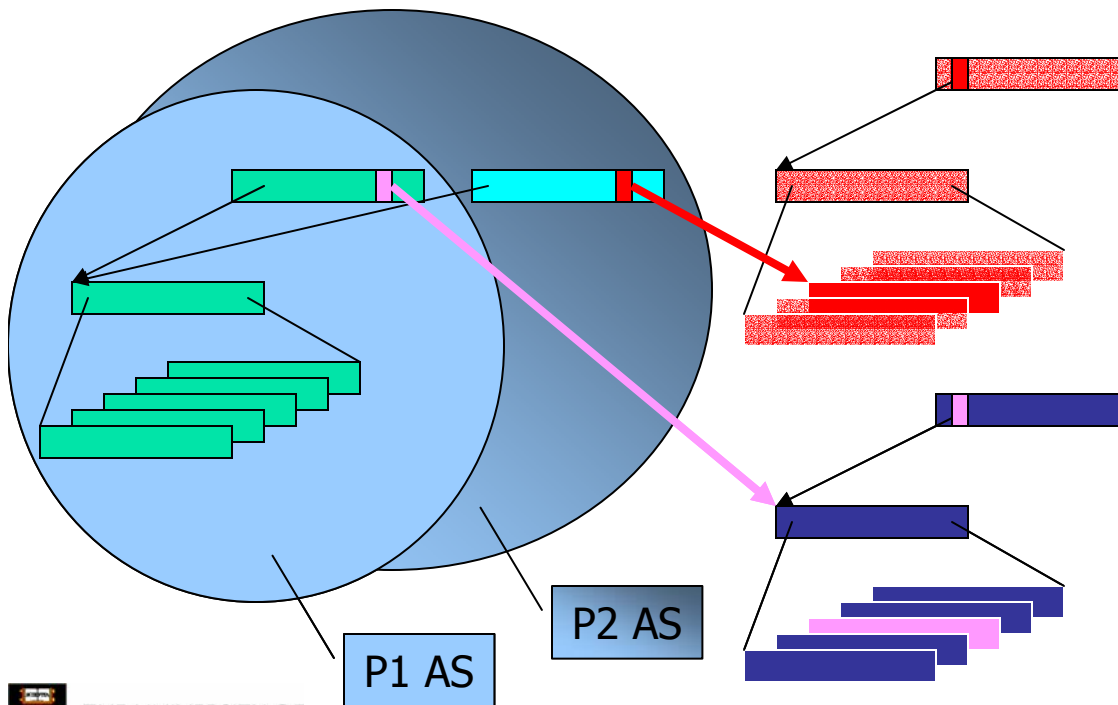
- SMP
 - TM area per processor



temporary mapping

- SMP

- TM area per processor
- Page table per processor

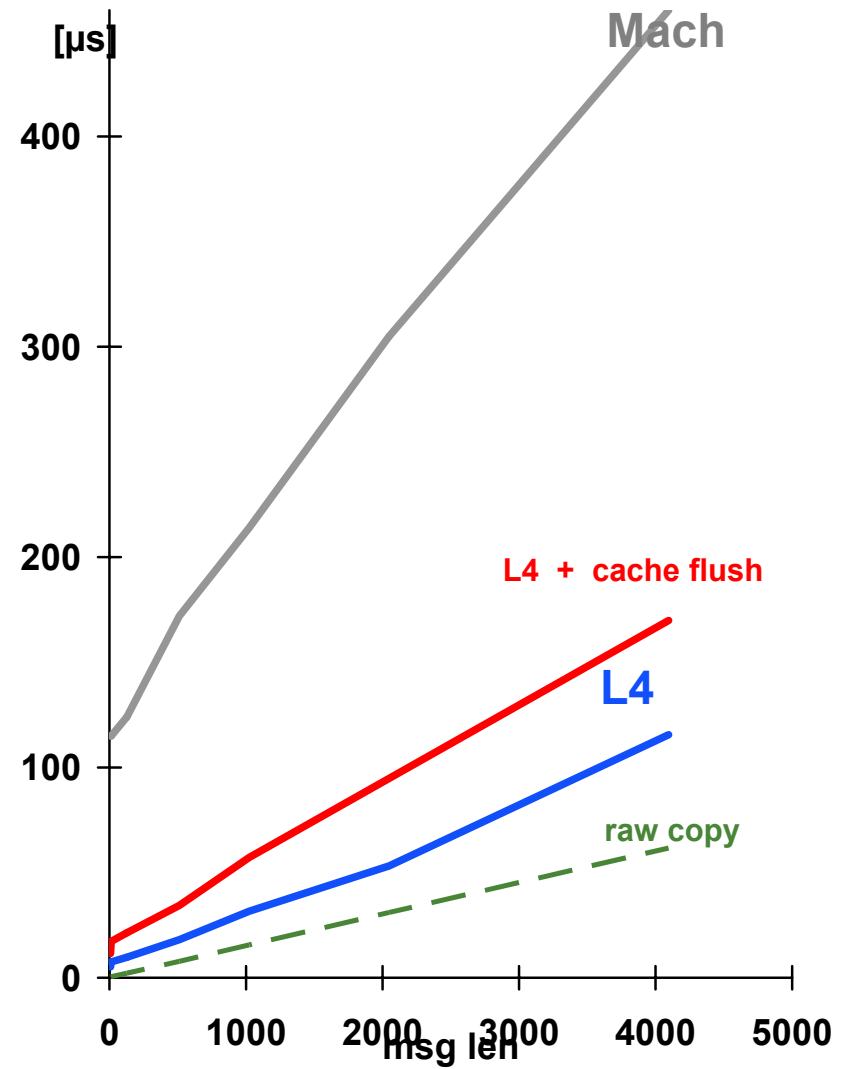


Cost estimates

	Copy in - copy out	Temporary mapping
<i>R/W operations</i>	$2 \times 2n$	$2n$
<i>Cache lines</i>	$3 \times n/8$	$2 \times n/8$
<i>Small n overhead cache misses</i>	$n/8$	0
<i>Large n cache misses</i>	$5 \times n/8$	$3 \times n/8$
<i>Overhead TLB misses</i>	0	$n / \text{words per page}$
<i>Startup instructions</i>	0	50

486 IPC costs

- Mach: copy in/out
- L4: temp mapping



Dispatching



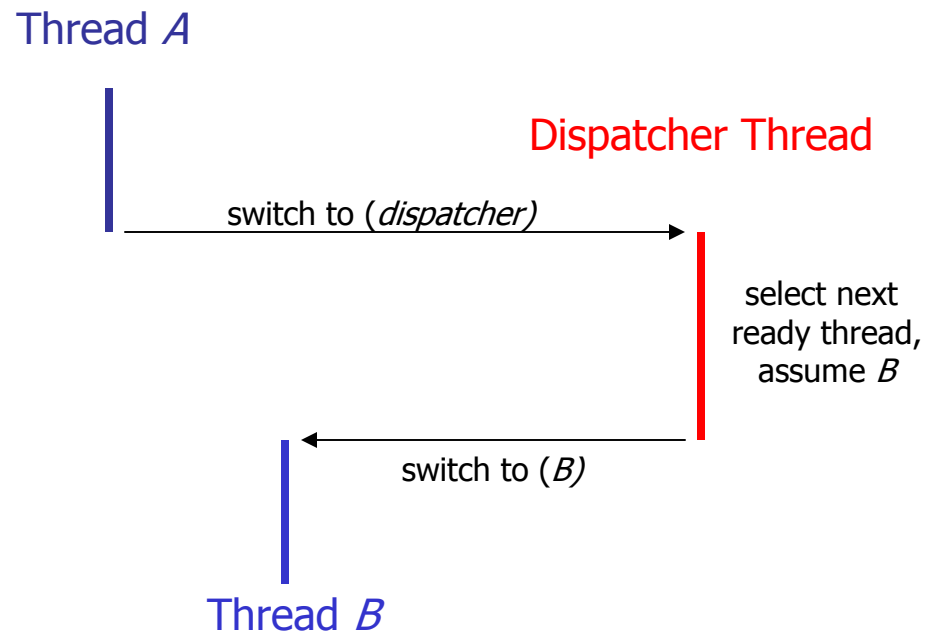
Dispatching topics:

- thread switch
 - (to a specific thread)
 - to next thread to be scheduled
 - (to nil)
 - implicitly, when ipc blocks
- priorities
- preemption
 - time slices
 - wakeups, interruptions
- timeouts and wake-ups
 - time

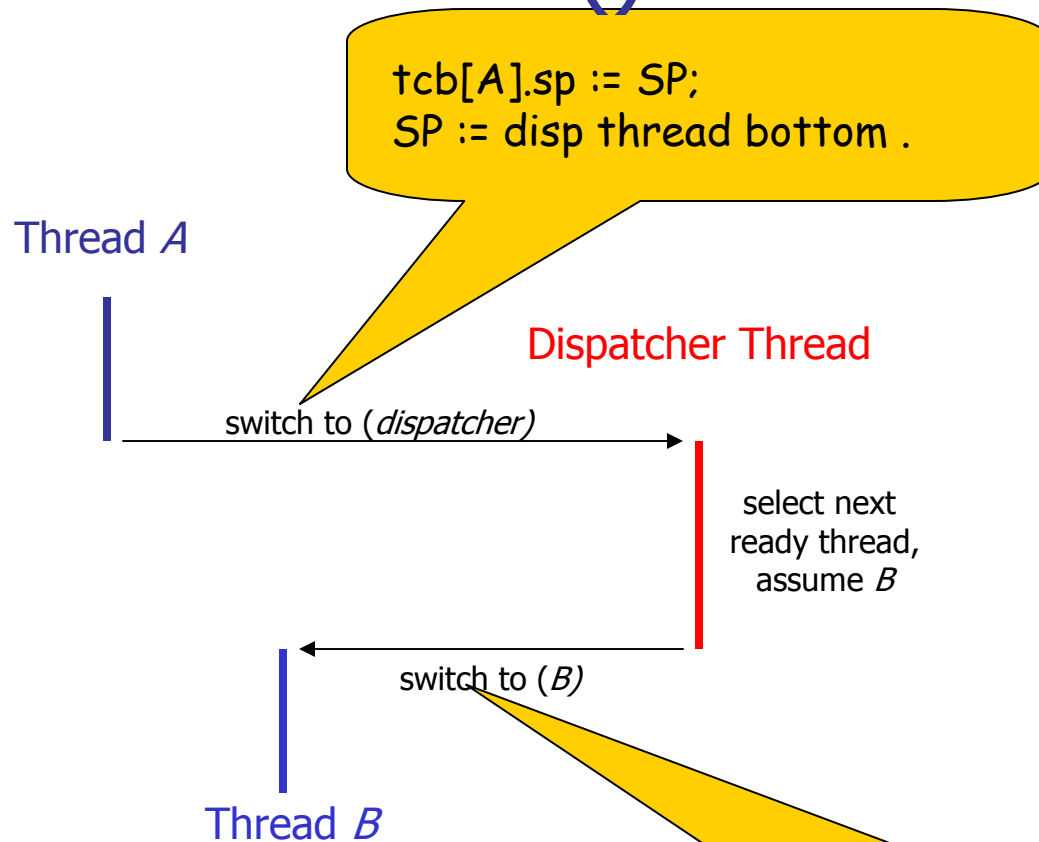


Switch to ():

- Smaller stack per thread
- Dispatcher is preemptable
 - Improved interrupt latency if dispatching is time consuming



Switch to ():



`tcb[A].sp := SP;`
`SP := disp thread bottom .`

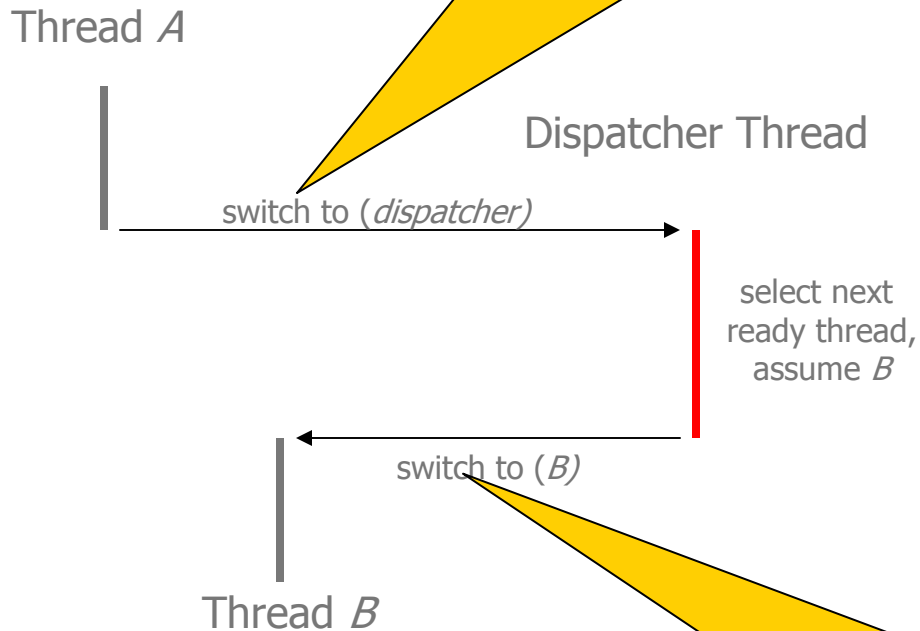
`SP := tcb[A].sp ;`
`if B ≠ A then`
 `switch from A to B`
`else return`
`fi .`

Why ??

- Optimizations :
 - disp thread is special
 - no user mode,
 - no own AS required
 - Can avoid AS switch
 - no id required
 - Freedom from tcb layout conventions
 - almost stateless (see priorities)
 - No need to preserve internal state between invocations
 - External state must be consistent
- $costs(A \rightarrow B)$
- $\approx costs(A \rightarrow disp \rightarrow B)$
 - $costs(select\ next)$
- $costs(A \rightarrow disp \rightarrow A)$ are low

Switch to ():

tcb[A].sp := SP;
SP := disp thread bottom .



Issue:
If preempted, thread A is not in a "good" state \Rightarrow
whenever disp thread is left, stack has to be discarded!
even if with intr or timer

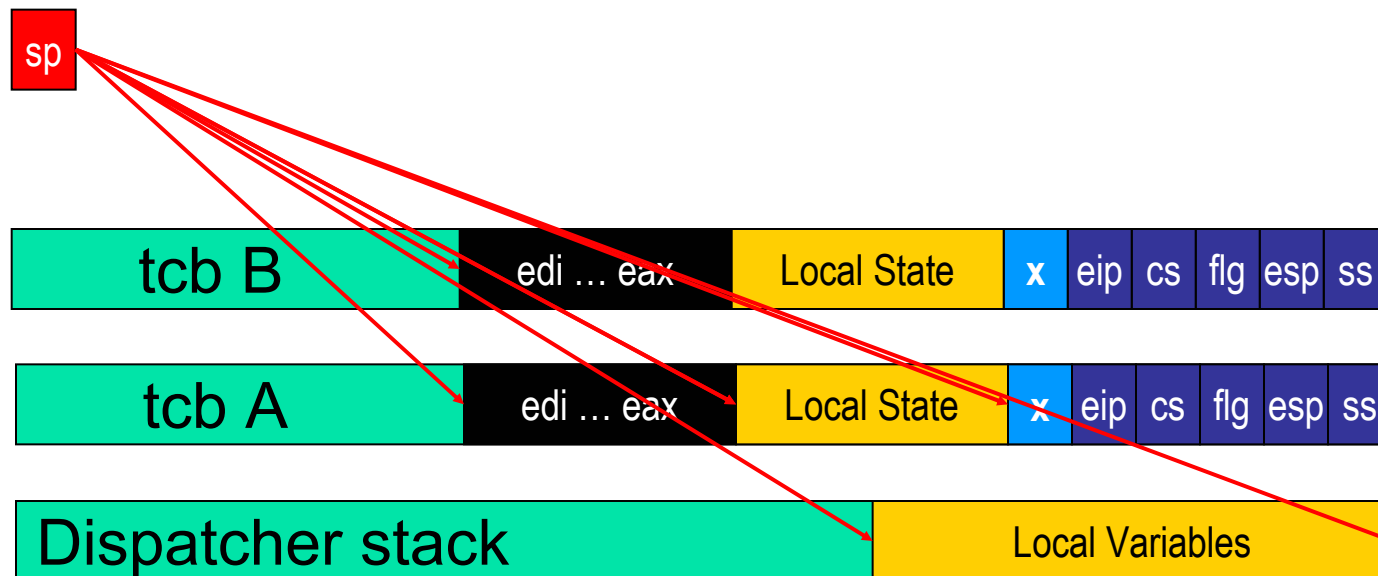
Why does this always work?

```
SP := tcb[A].sp ;  
if B  $\neq$  A then  
    switch from A to B  
else return  
fi .
```

Example: Simple Dispatch



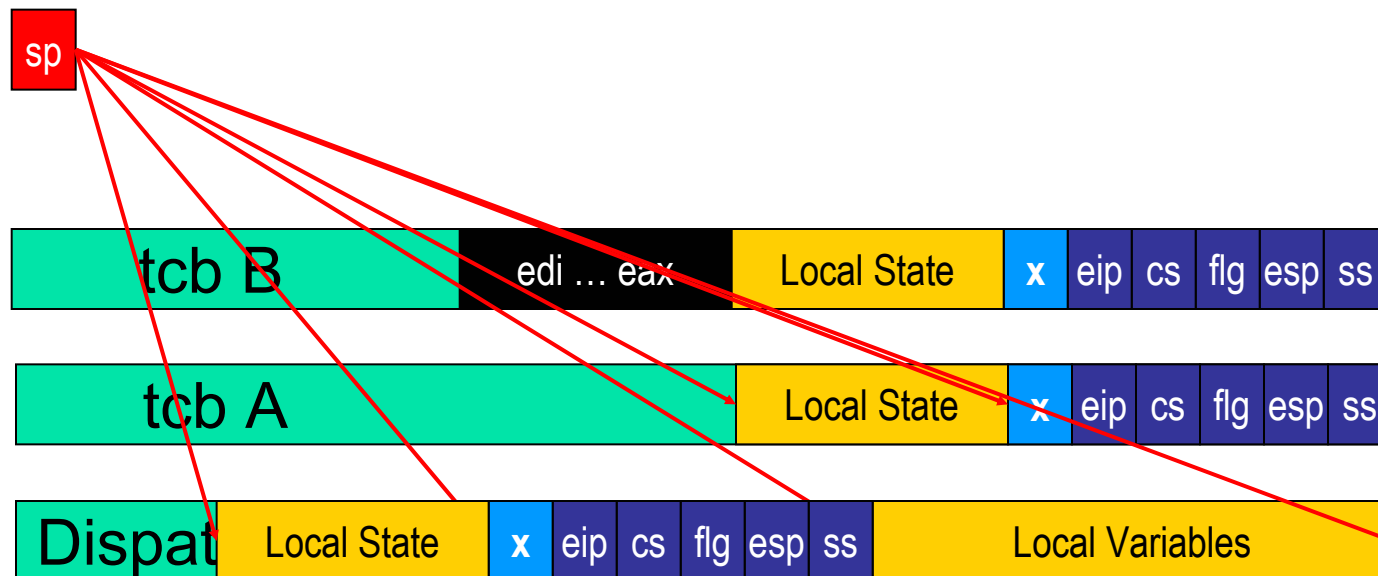
Example: Simple Dispatch



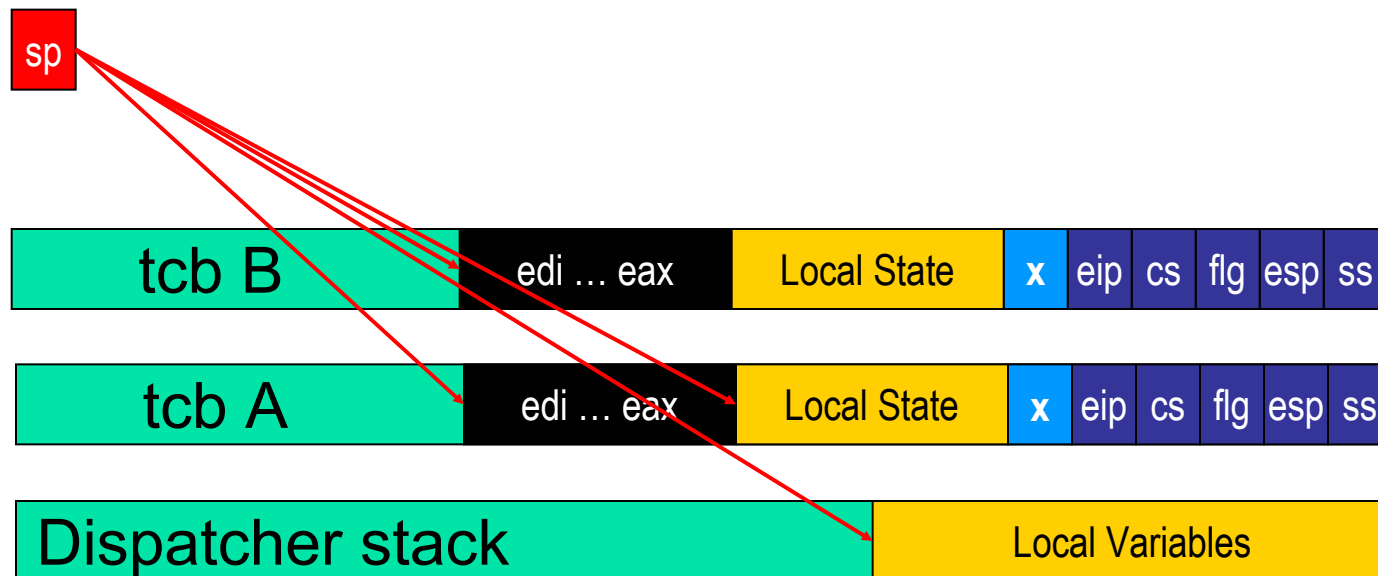
Example: Dispatch with 'Tick'



Example: Dispatch with 'Tick'



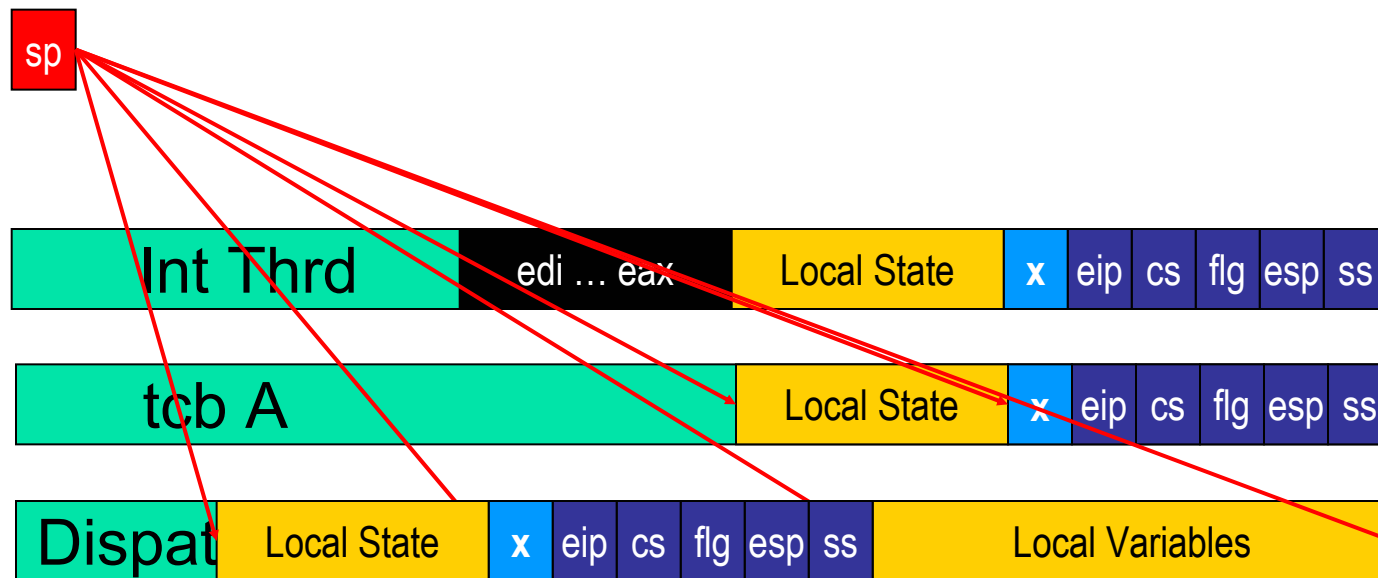
Example: Dispatch with 'Tick'



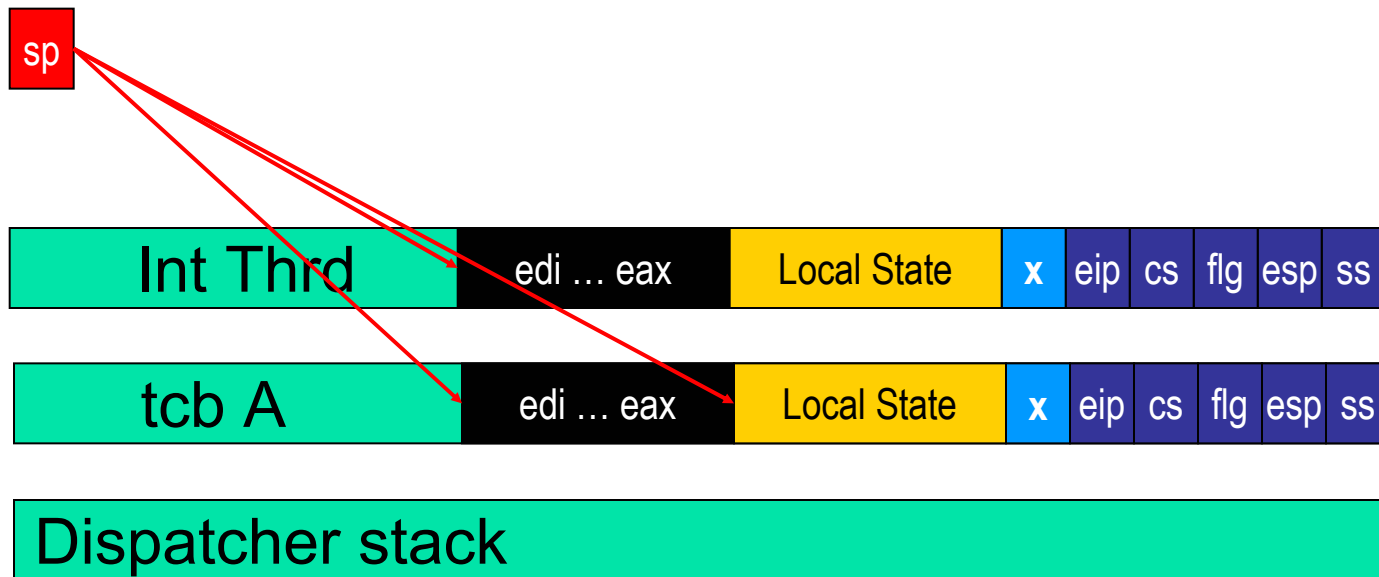
Example: Dispatch with Interrupt



Example: Dispatch with Interrupt

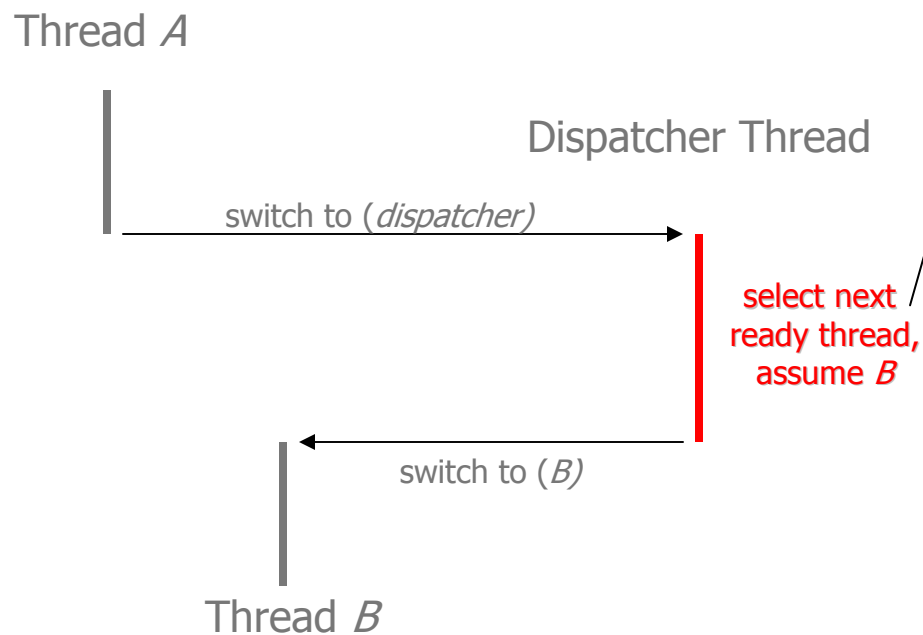


Example: Dispatch with Interrupt



Switch to ():

- dispatcher thread is also
 - *idle thread*

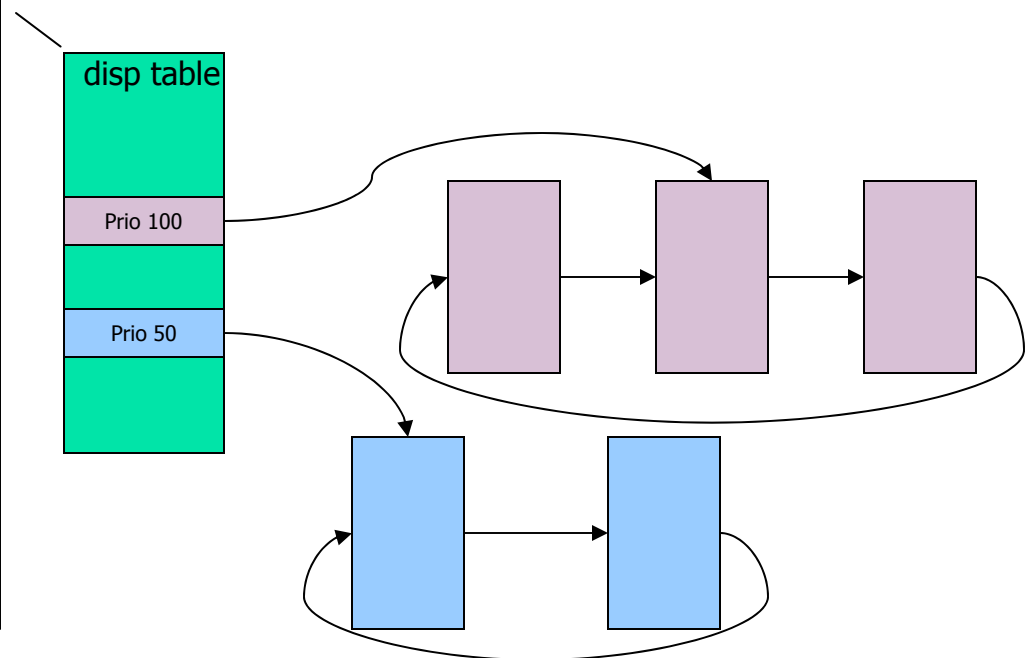


```
B := A ;  
do  
  B := next ready (B) ;  
  if B ≠ nil  
    then return  
  fi ;  
  idle  
od .
```

Priorities

- 0 (lowest) ... 255
- hard priorities
- round robin per prio
- dynamically changeable
- ready tcb list per prio
- 'current tcb' per list

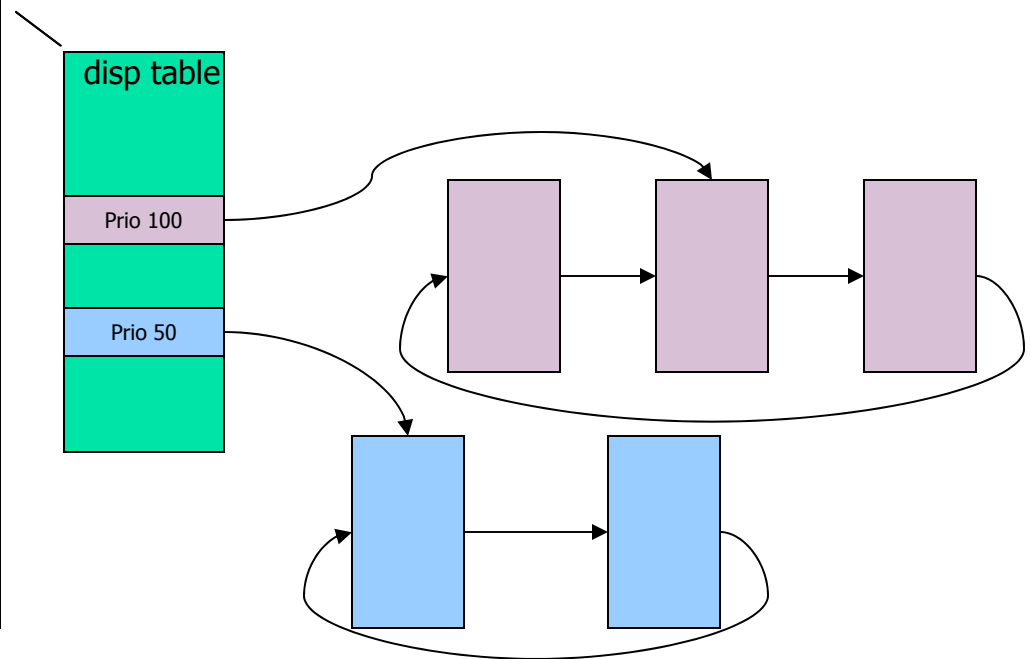
```
do
  p := 255;
  do
    if current[p] ≠ nil
      then B := current[p] ;
        return
      fi ;
    p -= 1
  until p < 0 od ;
  idle
od .
```



Priorities

- Optimization
 - keep highest active prio

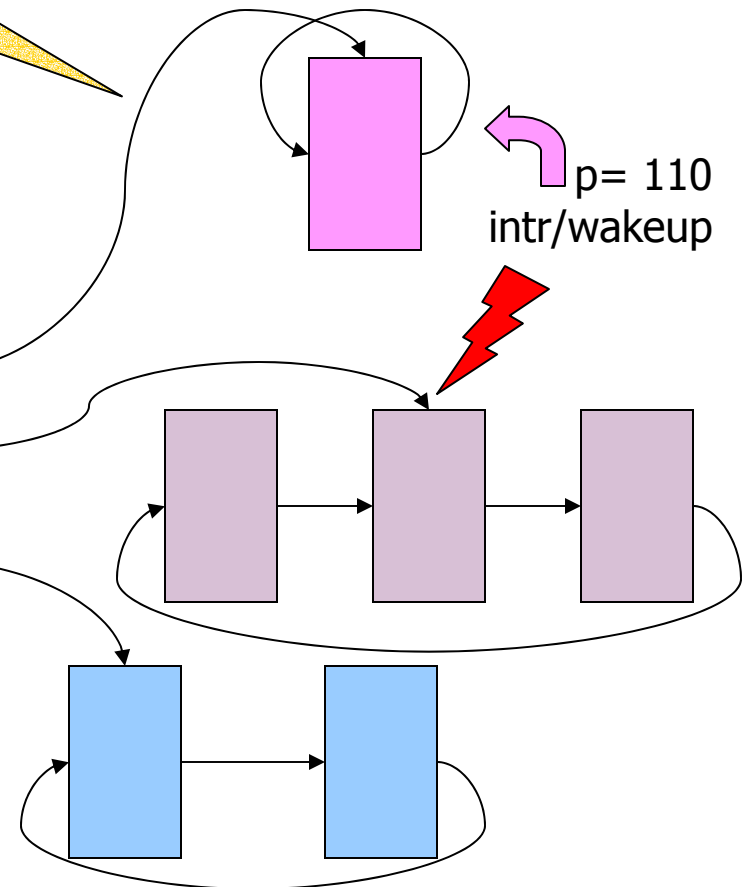
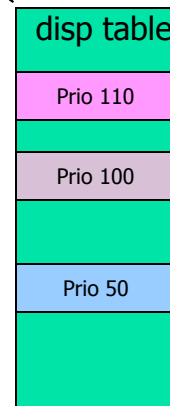
```
do
  if current[highest active p] ≠ nil
    then B := current[highest active p];
    return
  elif highest active p > 0
    then highest active p -= 1
  else
    idle
  fi
od .
```



Priorities, Preemption

$\text{highest active } p := \max(\text{new } p, \text{highest active } p) .$

```
do
  if current[highest active p] ≠ nil
    then B := current[highest active p];
      return
    elif highest active p > 0
      then highest active p -= 1
    else
      idle
    fi
od .
```

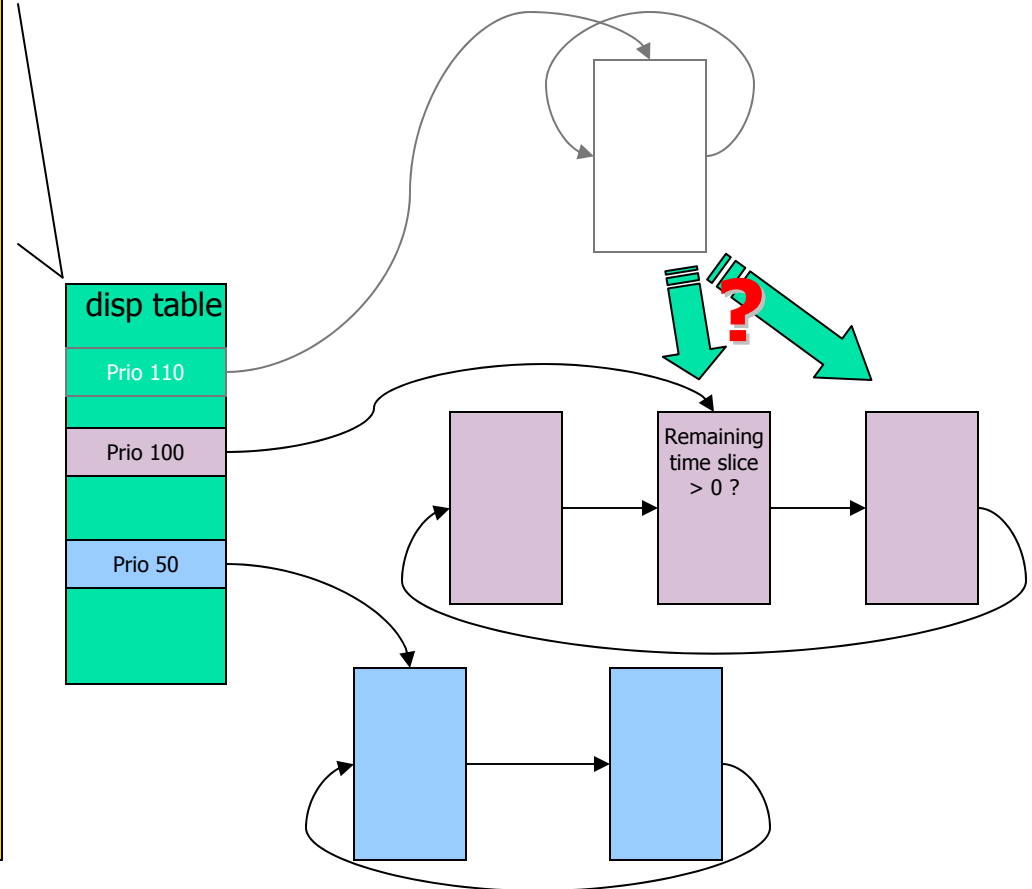


Priorities, Preemption

- What happens when a prio falls empty ?

```
do
  if current[highest active p] ≠ nil
    then round robin if necessary:
      B := current[highest active p];
      return
    elif highest active p > 0
      then highest active p -= 1
    else
      idle
  fi
od .

round robin if necessary:
  if curr[hi act p].rem ts = 0
    then curr[hi act p] := next ;
    current[hi act p].rem ts := new ts
  fi .
```

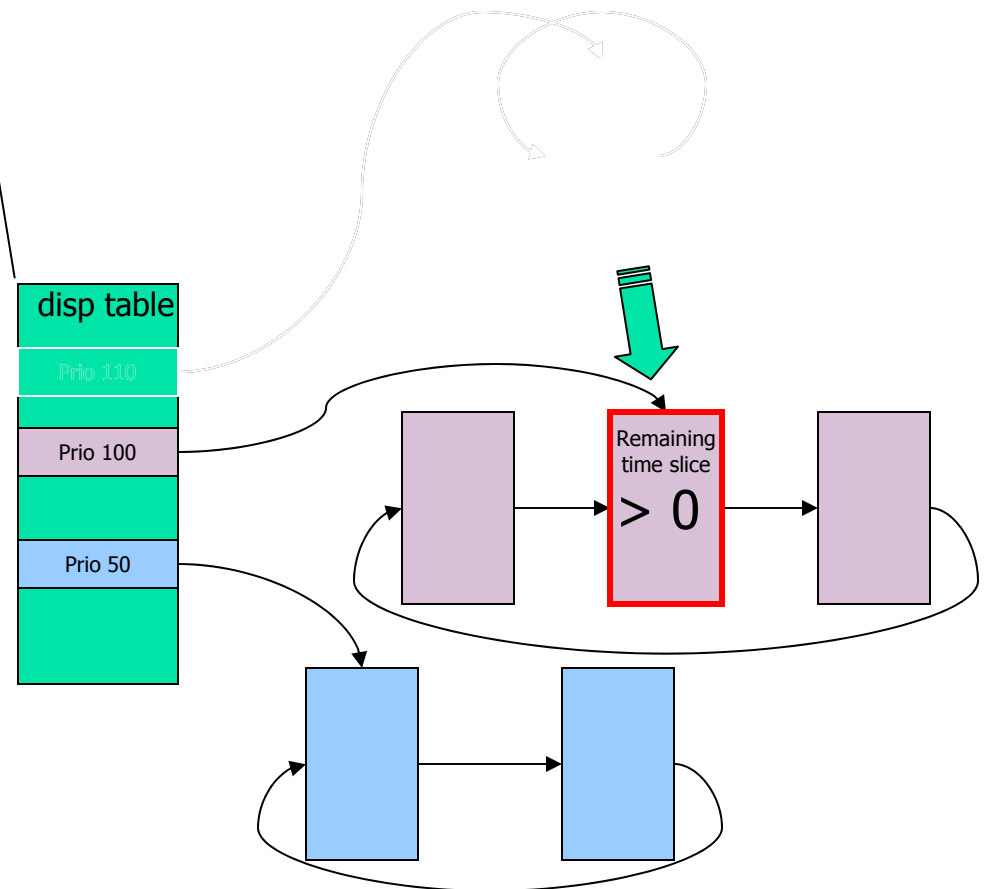


Priorities, Preemption

- What happens when a prio falls empty ?

```
do
  if current[highest active p] ≠ nil
    then round robin if necessary:
      B := current[highest active p];
      return
    elif highest active p > 0
      then highest active p -= 1
    else
      idle
  fi
od .

round robin if necessary:
  if curr[hi act p].rem ts = 0
    then curr[hi act p] := next ;
    current[hi act p].rem ts := new ts
  fi .
```

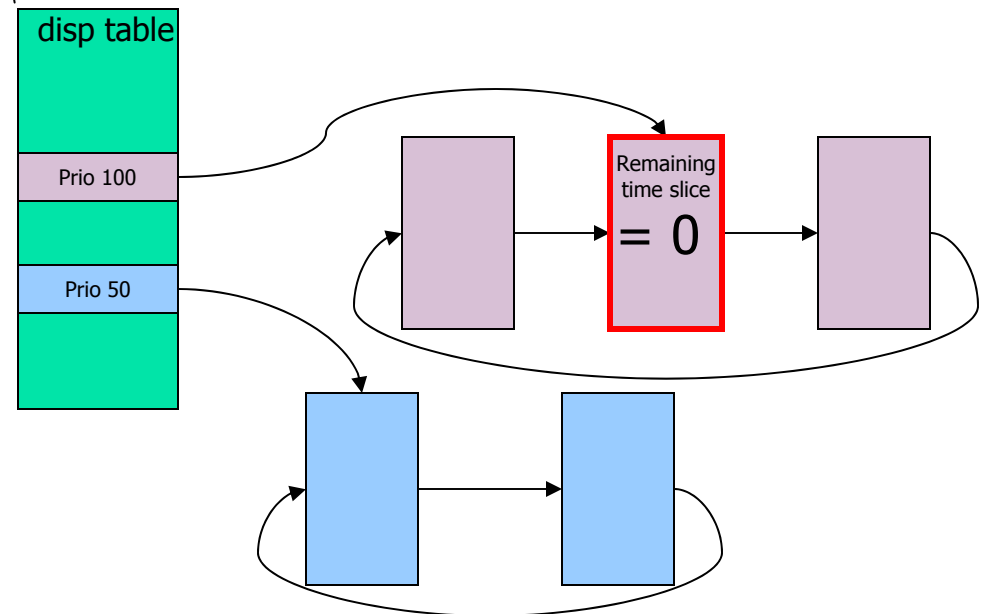


Preemption

- Preemption, time slice exhausted

```
do
  if current[highest active p] ≠ nil
    then round robin if necessary;
      B := current[highest active p];
      return
    elif highest active p > 0
      then highest active p -= 1
    else
      idle
  fi
od .

round robin if necessary:
  if curr[hi act p].rem ts = 0
    then curr[hi act p] := next ;
      current[hi act p].rem ts := new ts
  fi .
```

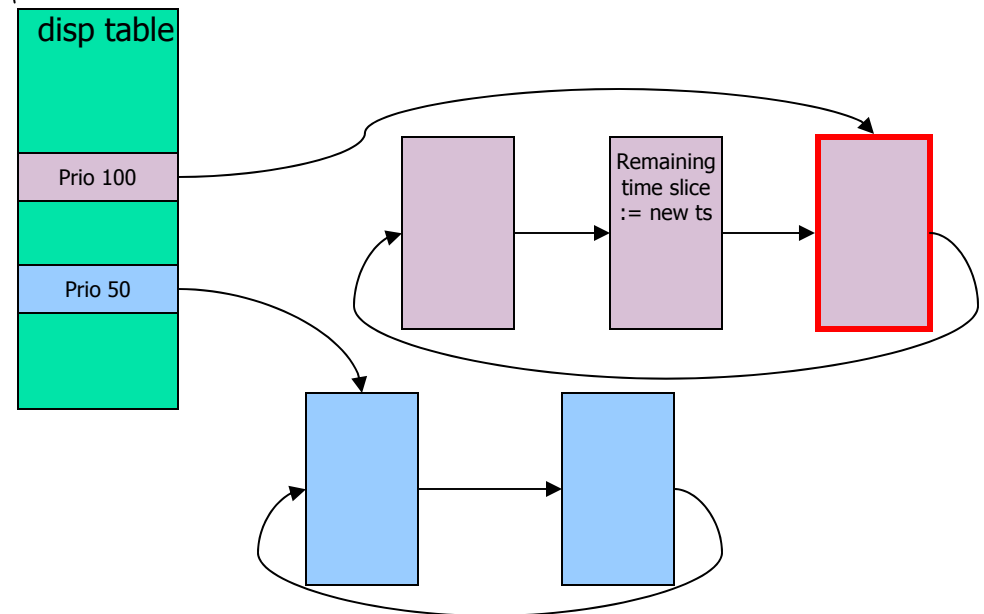


Preemption

- Preemption, time slice exhausted

```
do
  if current[highest active p] ≠ nil
    then round robin if necessary:
      B := current[highest active p];
      return
    elif highest active p > 0
      then highest active p -= 1
    else
      idle
  fi
od .

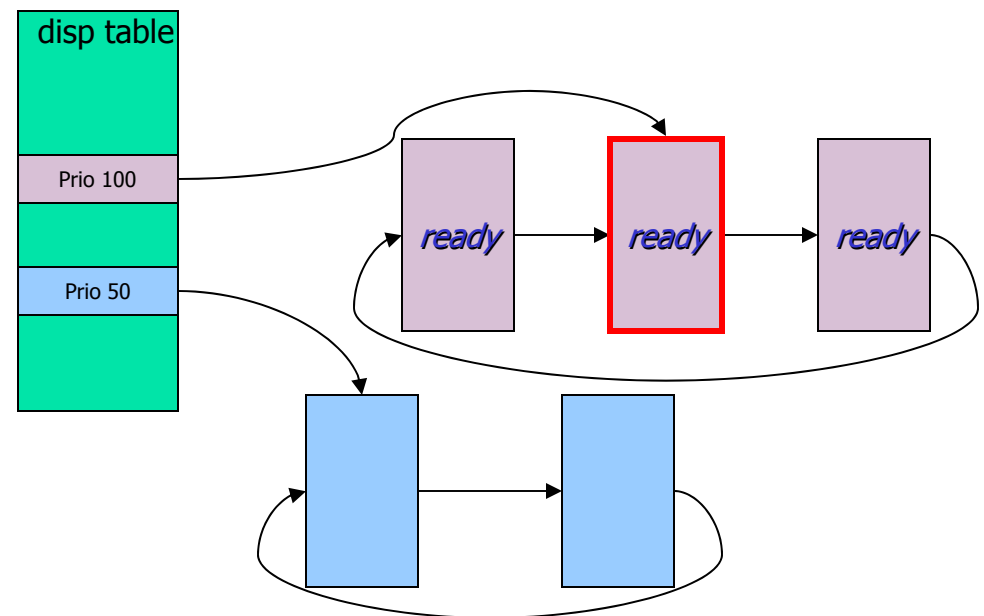
round robin if necessary:
  if current[hi act p].rem ts = 0
    then current[hi act p].rem ts := new ts ;
    current[hi act p] := next
  fi .
```



Lazy Dispatching

Thread state toggles frequently (per ipc)

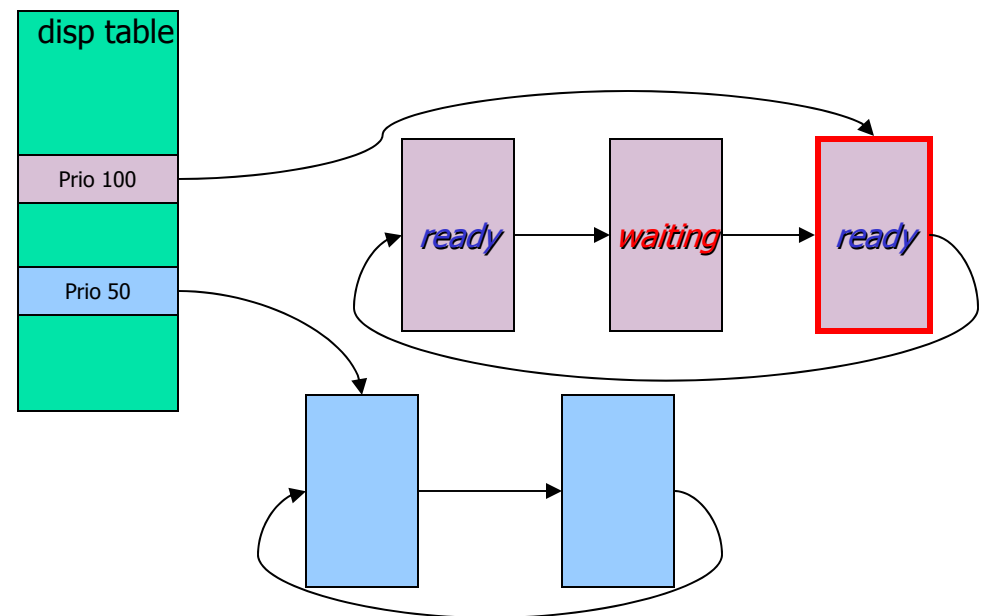
- *ready* ↔ *waiting*
 - delete/insert ready list is expensive
 - therefore: delete *lazily* from ready list



Lazy Dispatching

Thread state toggles frequently (per ipc)

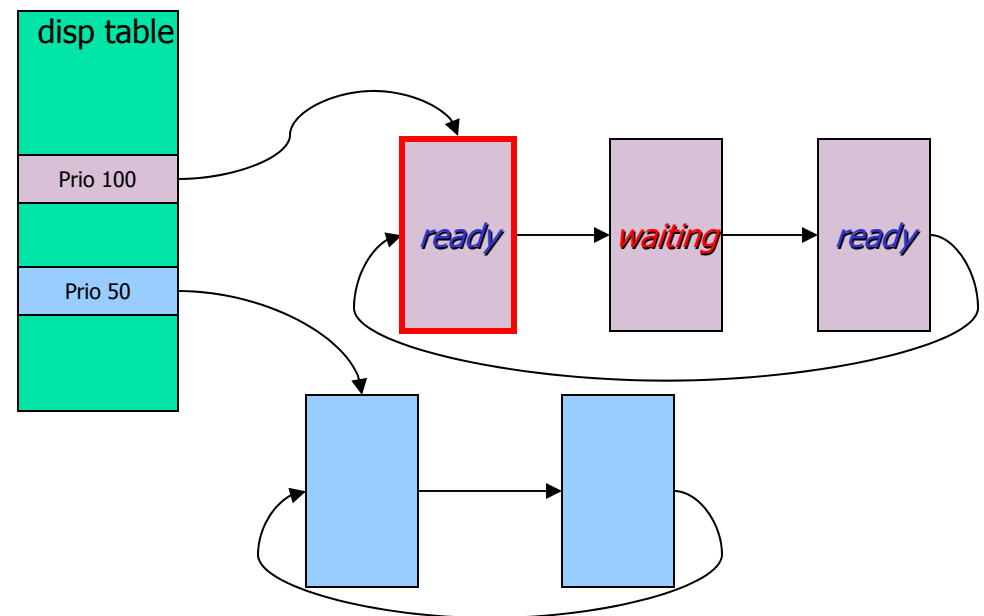
- *ready* ↔ *waiting*
 - delete/insert ready list is expensive
 - therefore: delete *lazily* from ready list



Lazy Dispatching

Thread state toggles frequently (per ipc)

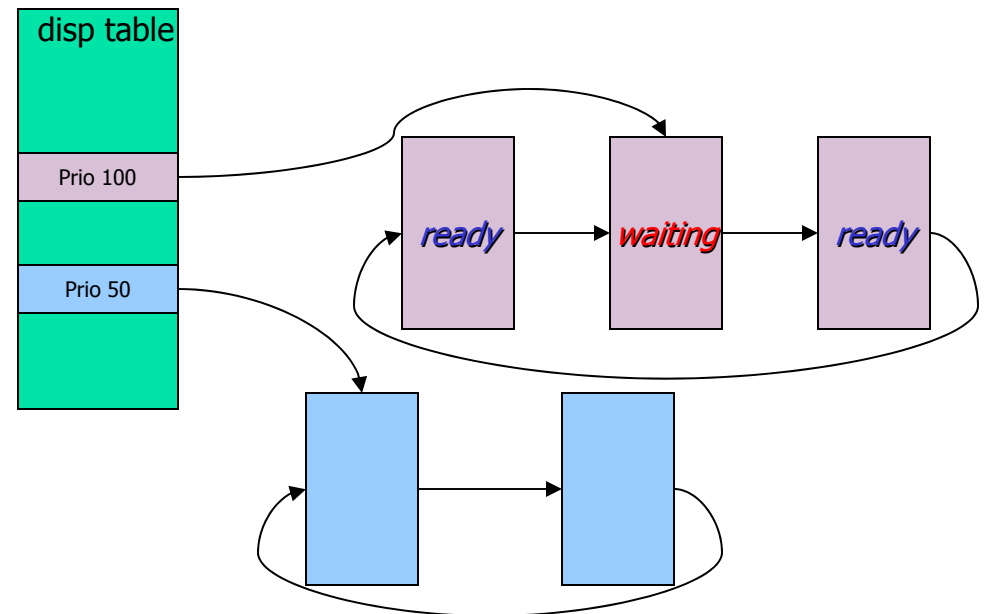
- *ready* ↔ *waiting*
 - delete/insert ready list is expensive
 - therefore: delete *lazily* from ready list



Lazy Dispatching

Thread state toggles frequently (per ipc)

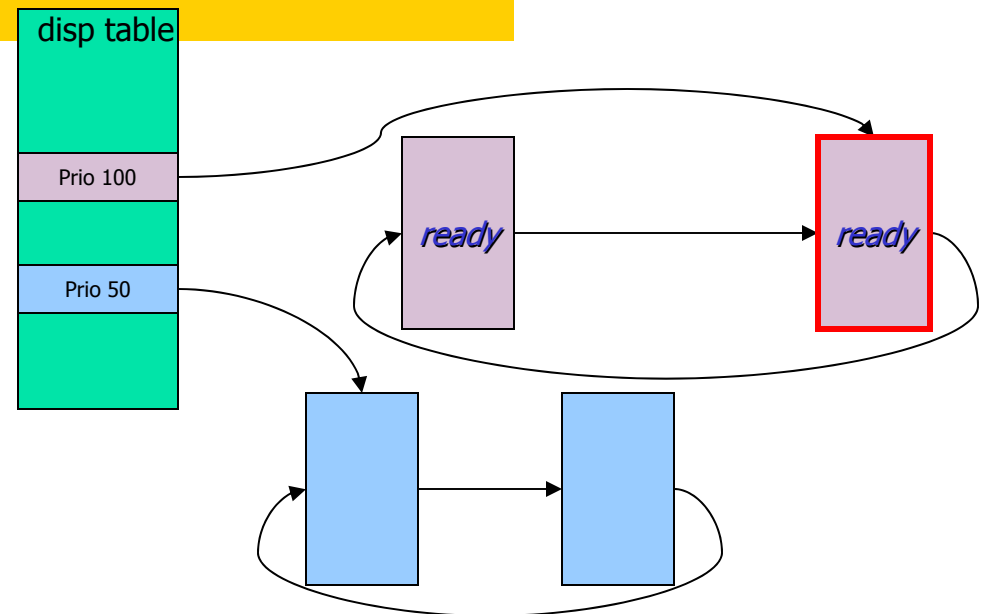
- *ready* ↔ *waiting*
 - delete/insert ready list is expensive
 - therefore: delete *lazily* from ready list



Lazy Dispatching

Thread state toggles frequently (per ipc)

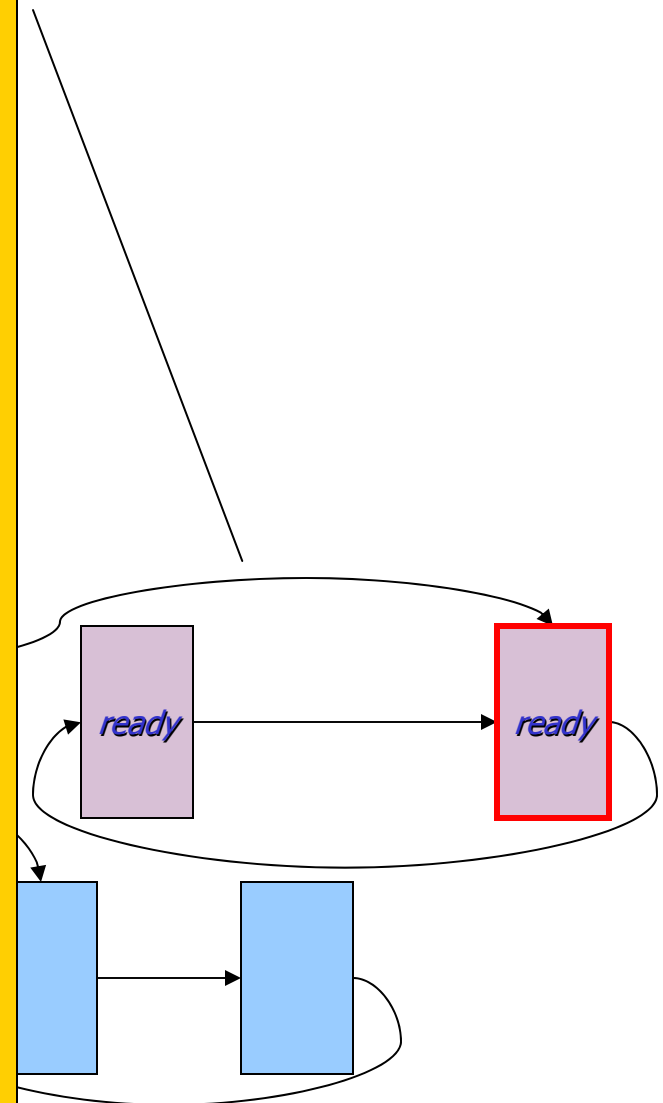
- *ready* ↔ *waiting*
 - delete/insert ready list is expensive
 - therefore: delete *lazily* from ready list
- Whenever reaching a non-ready thread,
 - delete it from list
 - proceed with next



Lazy Dispatching

```
do
  round robin if necessary;
  if current[highest active p] ≠ nil
    then B := current[highest active p]; return
  elif highest active p > 0
    then highest active p -= 1
  else
    idle
  fi
od .

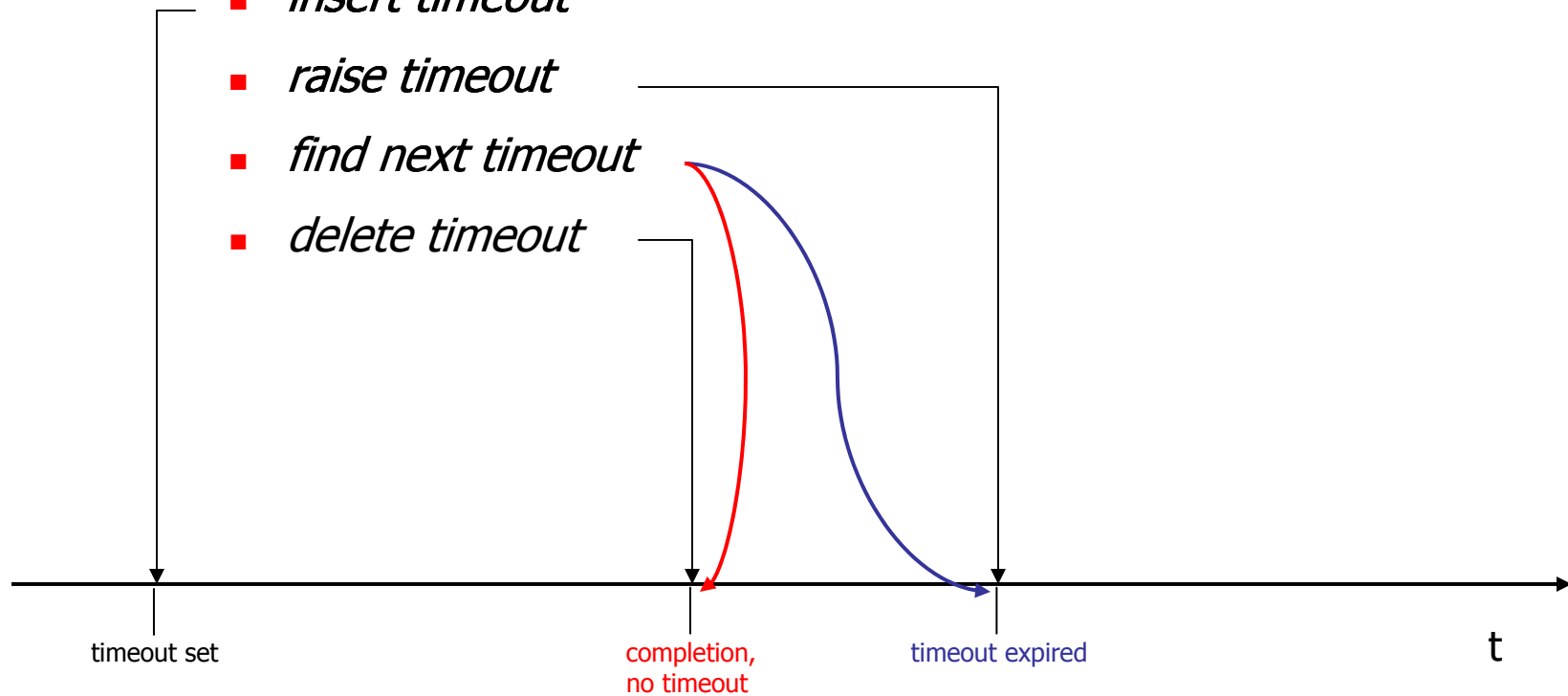
round robin if necessary:
  while curr[hi act p] ≠ nil do
    if curr[hi act p].state ≠ ready
      then delete from list (curr[hi act p])
    elif curr[hi act p].rem ts = 0
      then curr[hi act p].rem ts := new ts
    else leave round robin if necessary
    fi ;
    curr[hi act p] := next ;
  od .
```



Timeouts & Wakeups

- Operations:

- *insert timeout*
- *raise timeout*
- *find next timeout*
- *delete timeout*



- raised-timeout costs are uncritical
(occur only after timeout exp time)

- ***most timeouts are never raised !***



Timeouts & Wakeups

too expensive

Idea 1: *unsorted list*

- *insert timeout* costs:
 - search + insert entry 20..100 cycles
- *find next timeout* costs:
 - parse entire list $n \times 10..50$ cycles
- *raise timeout* costs:
 - delete found entry 20..100 cycles
- *delete timeout* costs:
 - delete entry 20..100 cycles

Timeouts & Wakeups

too expensive

Idea 2: *sorted list*

- *insert timeout* costs:
 - search + insert entry $n/2 \times 10..50 + 20..100$ cycles
- *find next timeout* costs:
 - find list head 10..50 cycles
- *raise timeout* costs:
 - delete head 20..100 cycles
- *delete timeout* costs:
 - delete entry 20..100 cycles

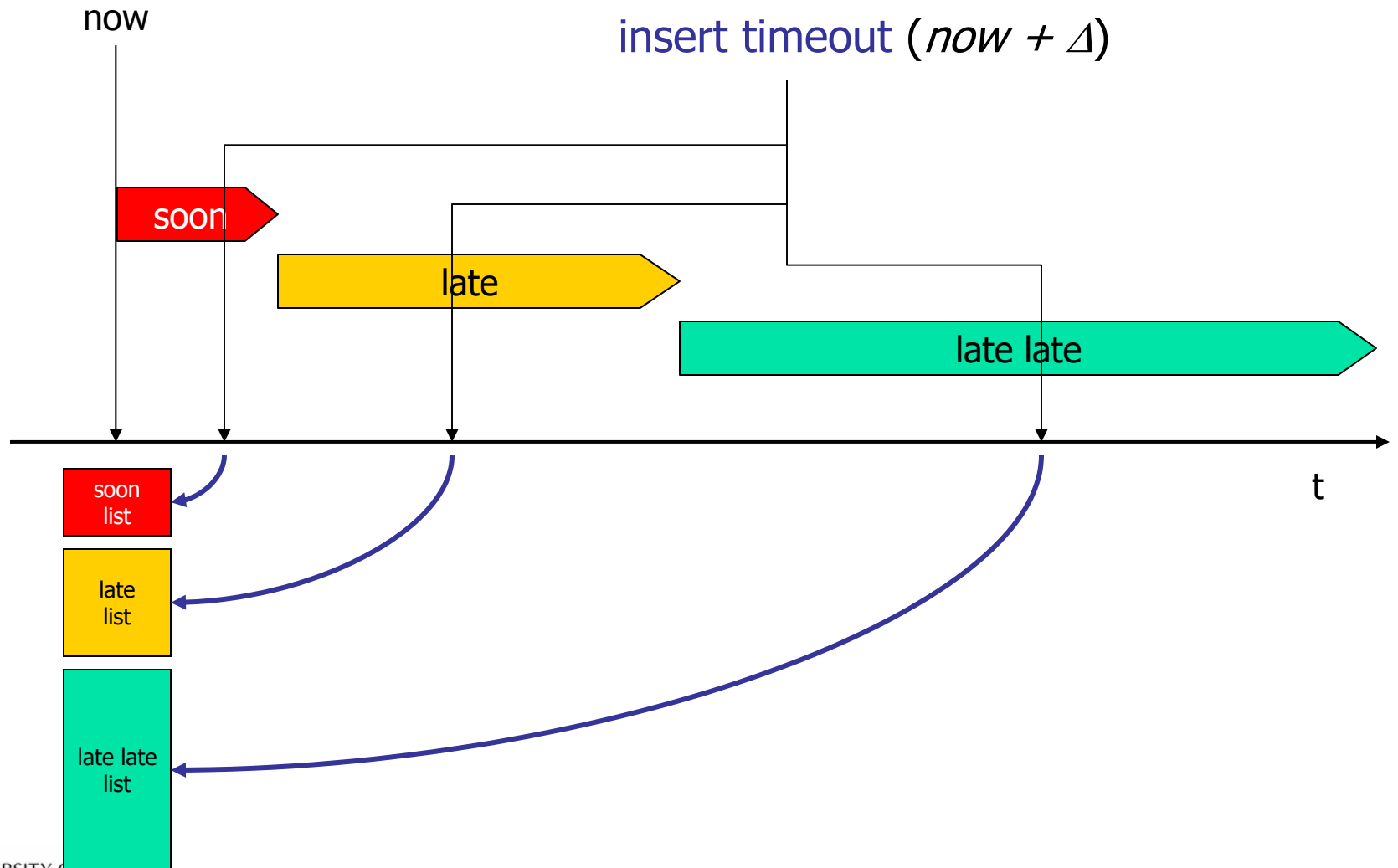
Timeouts & Wakeups

too expensive
too complicated

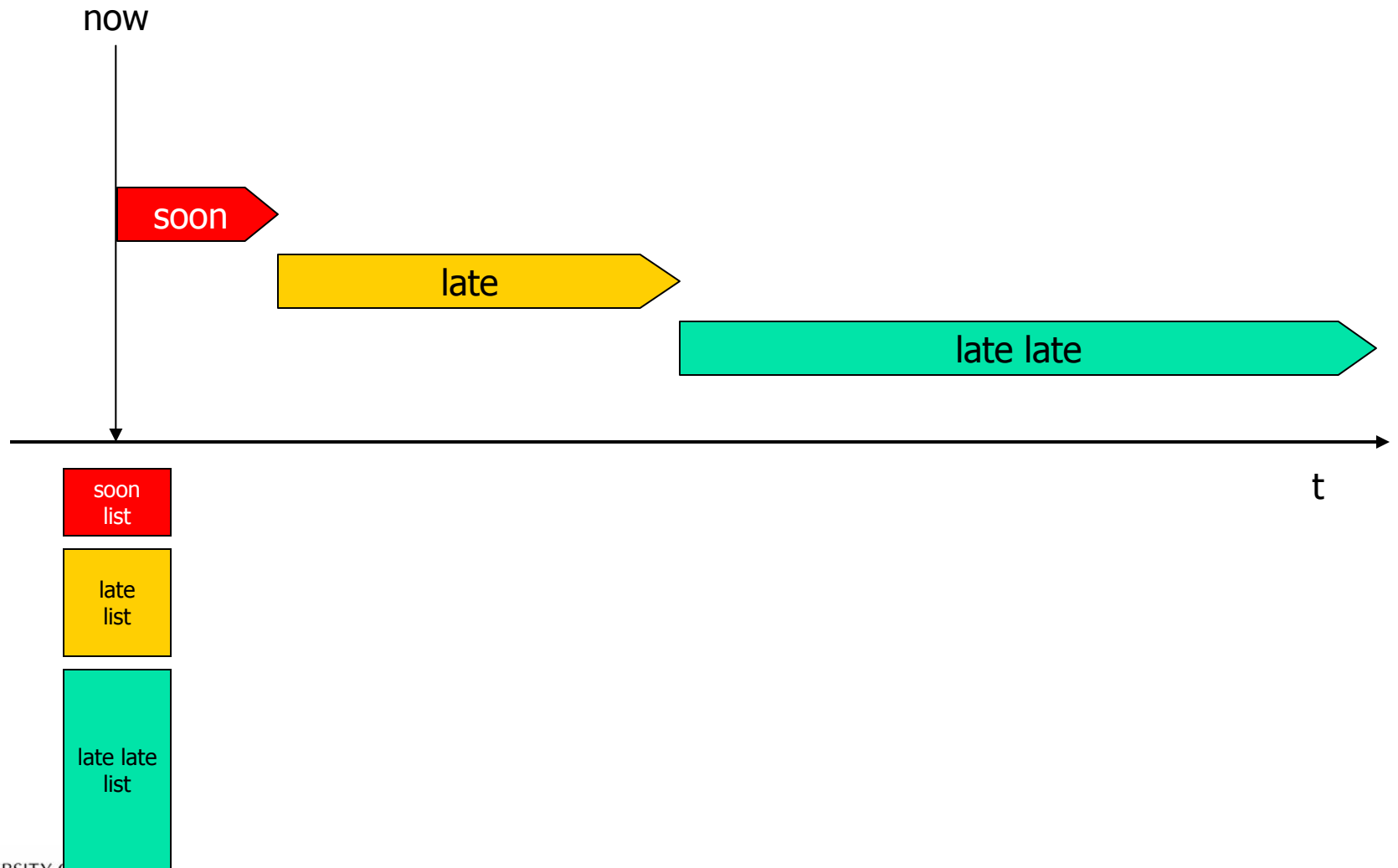
Idea 3: *sorted tree*

- *insert timeout* costs:
 - search + insert entry log n × 20..100 + 20..100 cycles
- *find next timeout* costs:
 - find list head 10..50 cycles
- *raise timeout* costs:
 - delete head 20..100 cycles
- *delete timeout* costs:
 - delete entry 20..100 cycles

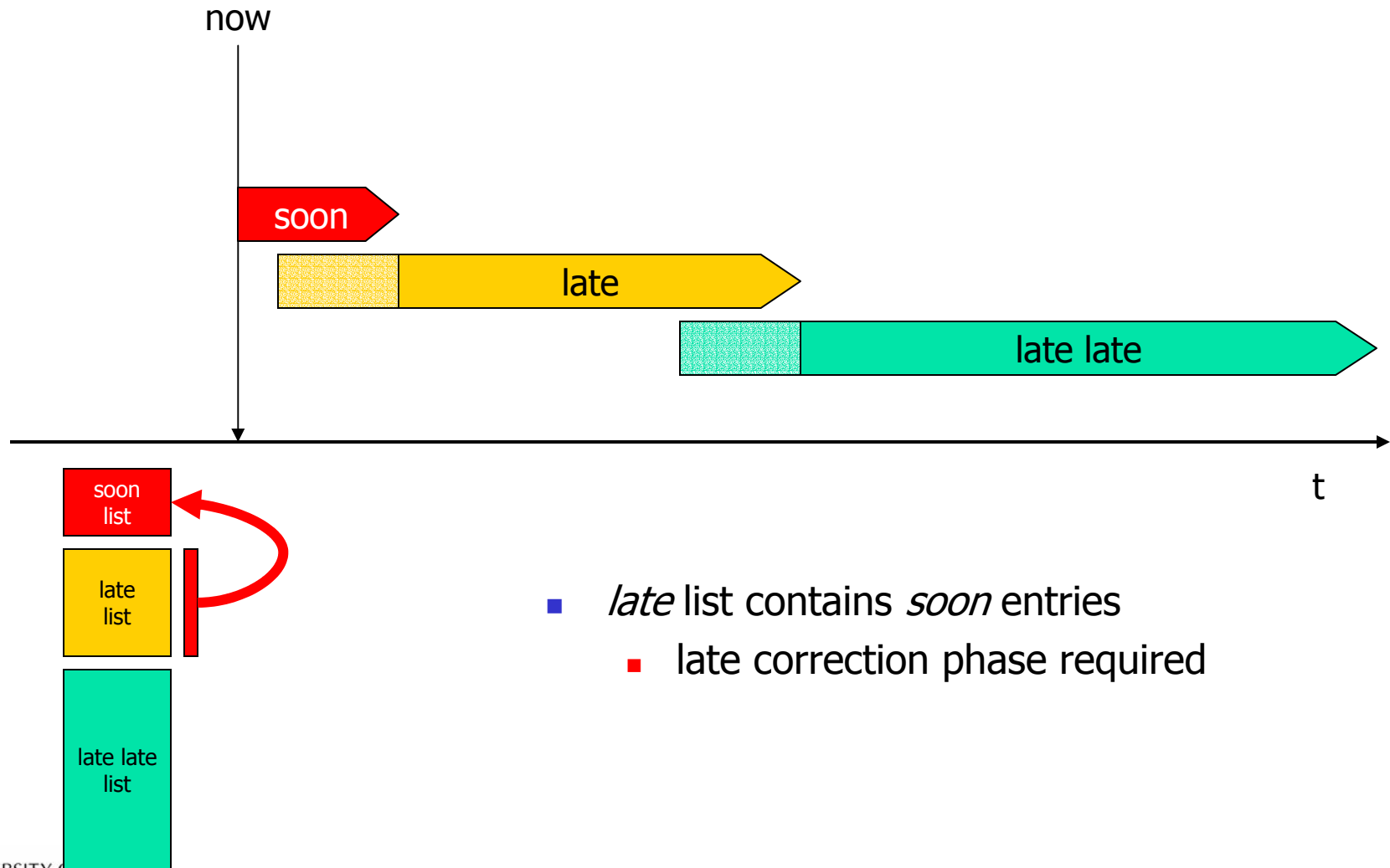
Wakeup Classes



Wakeup Classes

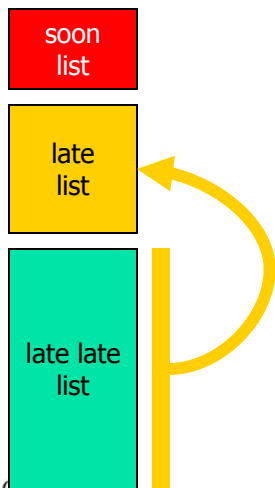
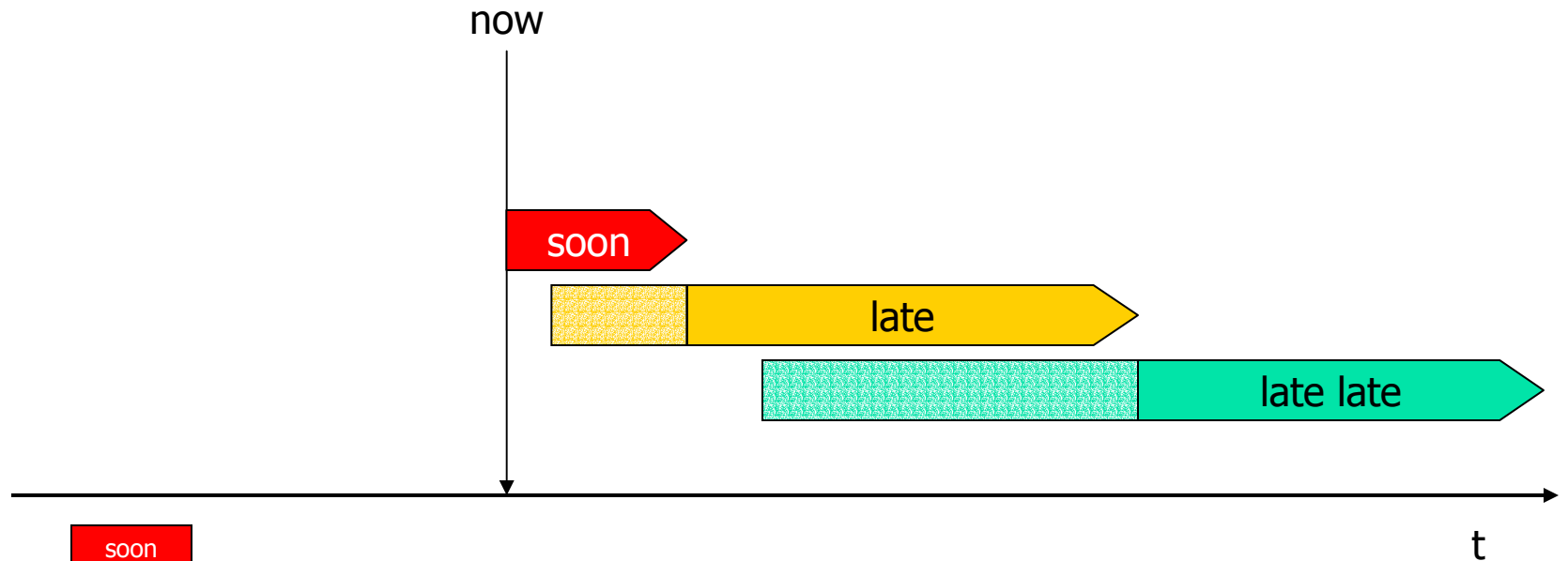


Wakeup Classes



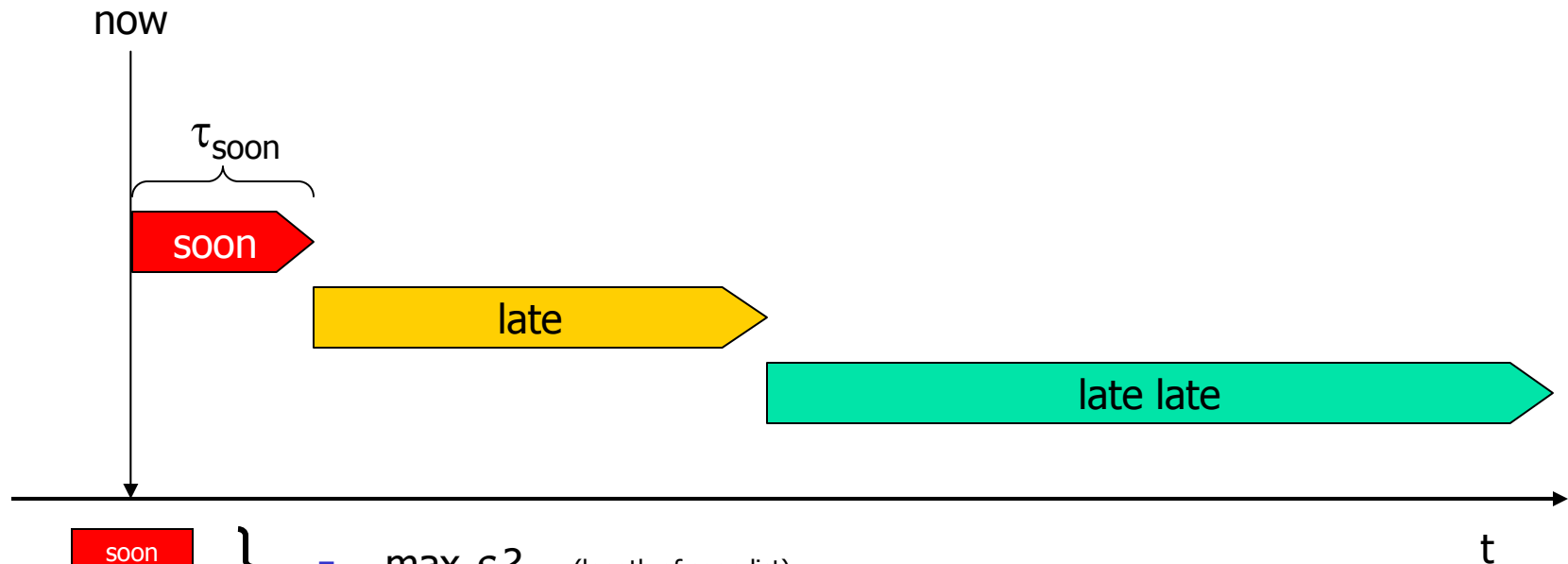
- *late* list contains *soon* entries
 - late correction phase required

Wakeup Classes



- *late late* list contains *late* entries
 - late late correction phase required

Wakeup Classes



soon list

late list

late late list

}

■ max s ? (length of soon list)

■ $s \leq$ timeouts to be raised in τ_{soon} + new timeouts in τ_{soon}

$\Rightarrow s$ is small if τ_{soon} is short enough



Timeouts & Wakeups

Idea 4: *unsorted wakeup classes*

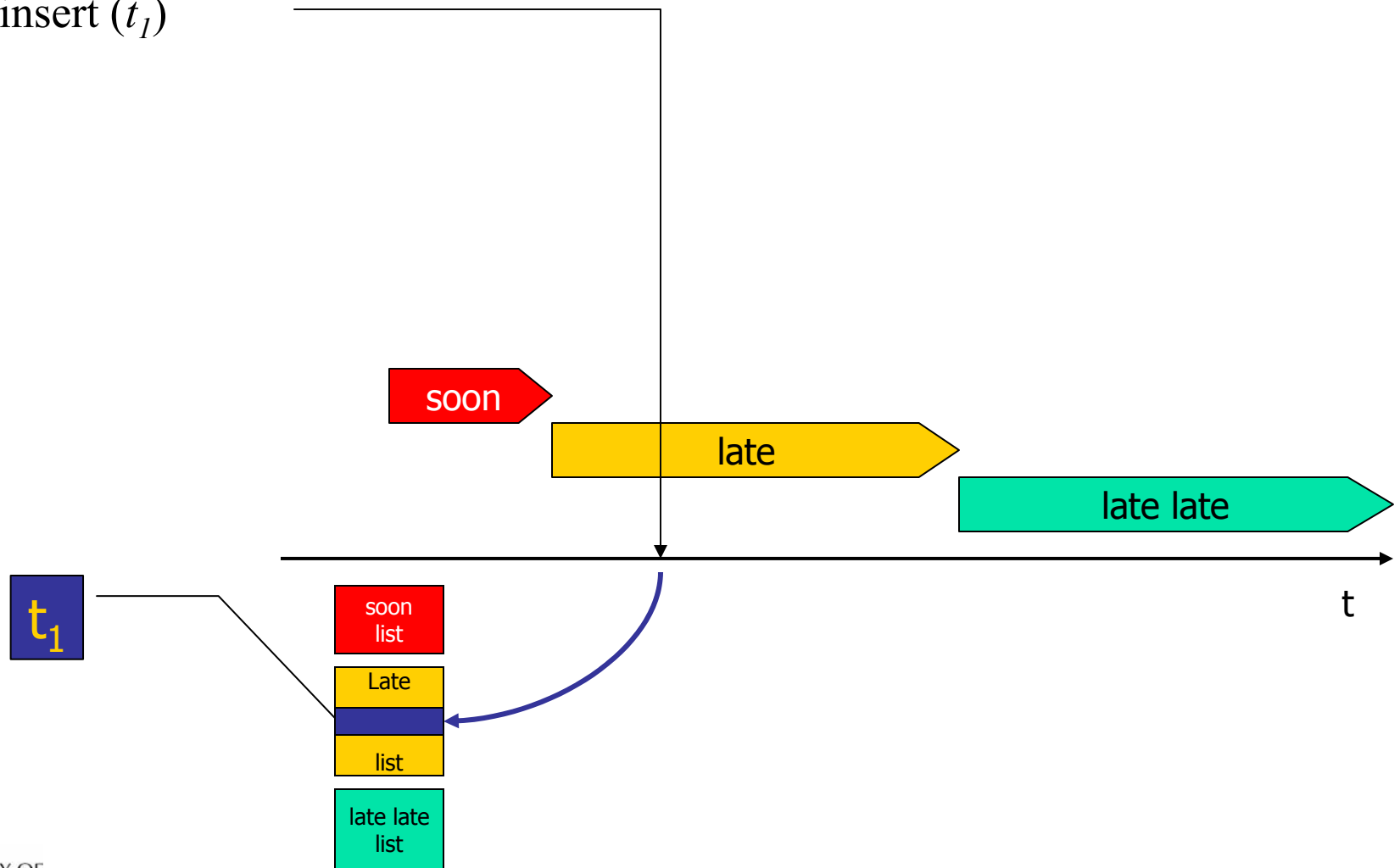
- *insert timeout* costs:
 - select class + add entry $10 + 20..100$ cycles
- *find next timeout* costs:
 - search *soon* class $s..n \times 10..50$
- *raise timeout* costs:
 - delete head $20..100$ cycles
- *delete timeout* costs:
 - delete entry $20..100$ cycles

still
too expensive

- raised-timeout costs are uncritical
(occur only after timeout exp time)
- ***BUT most timeouts are never raised !***

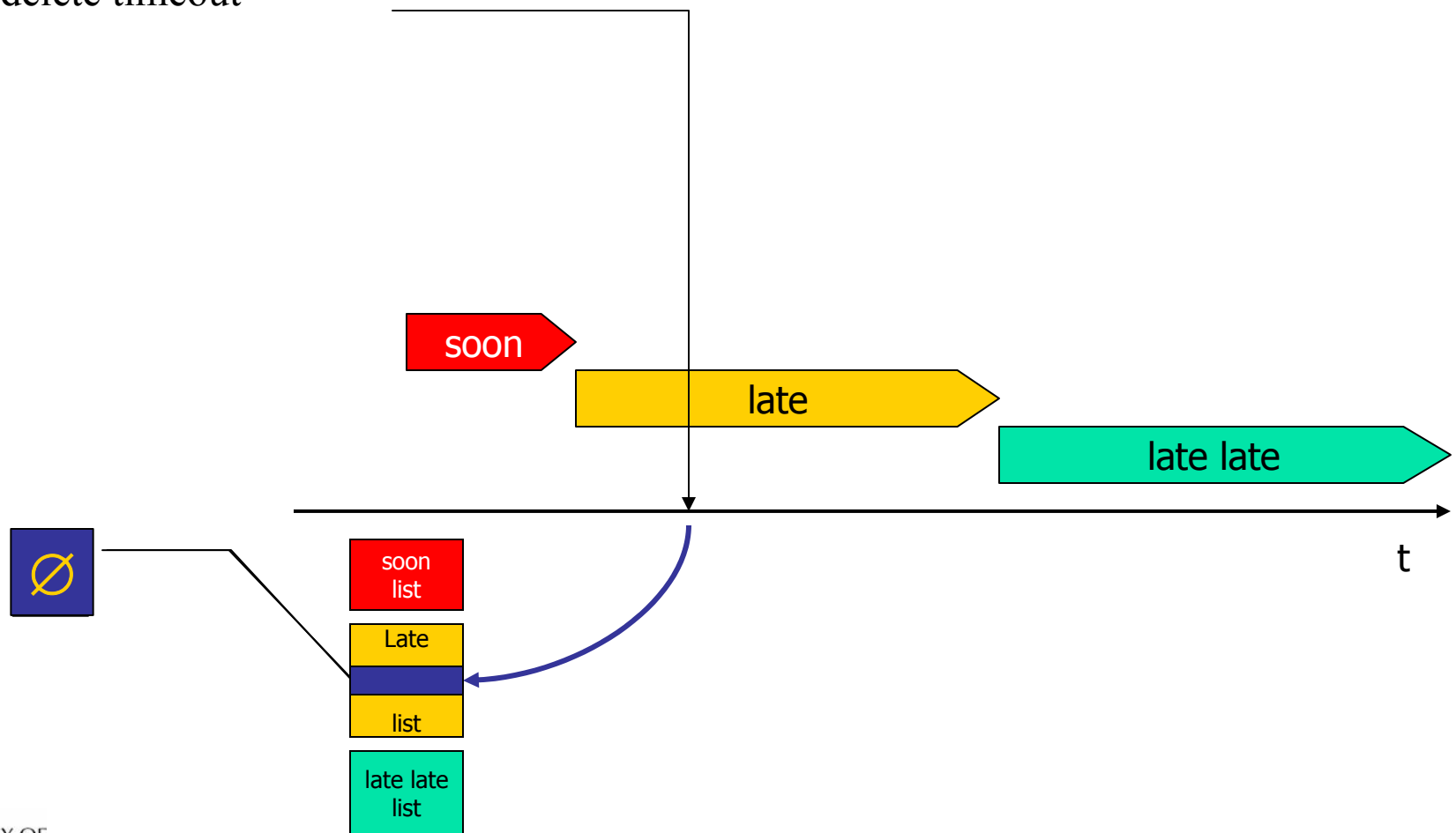
Lazy Timeouts

insert (t_1)



Lazy Timeouts

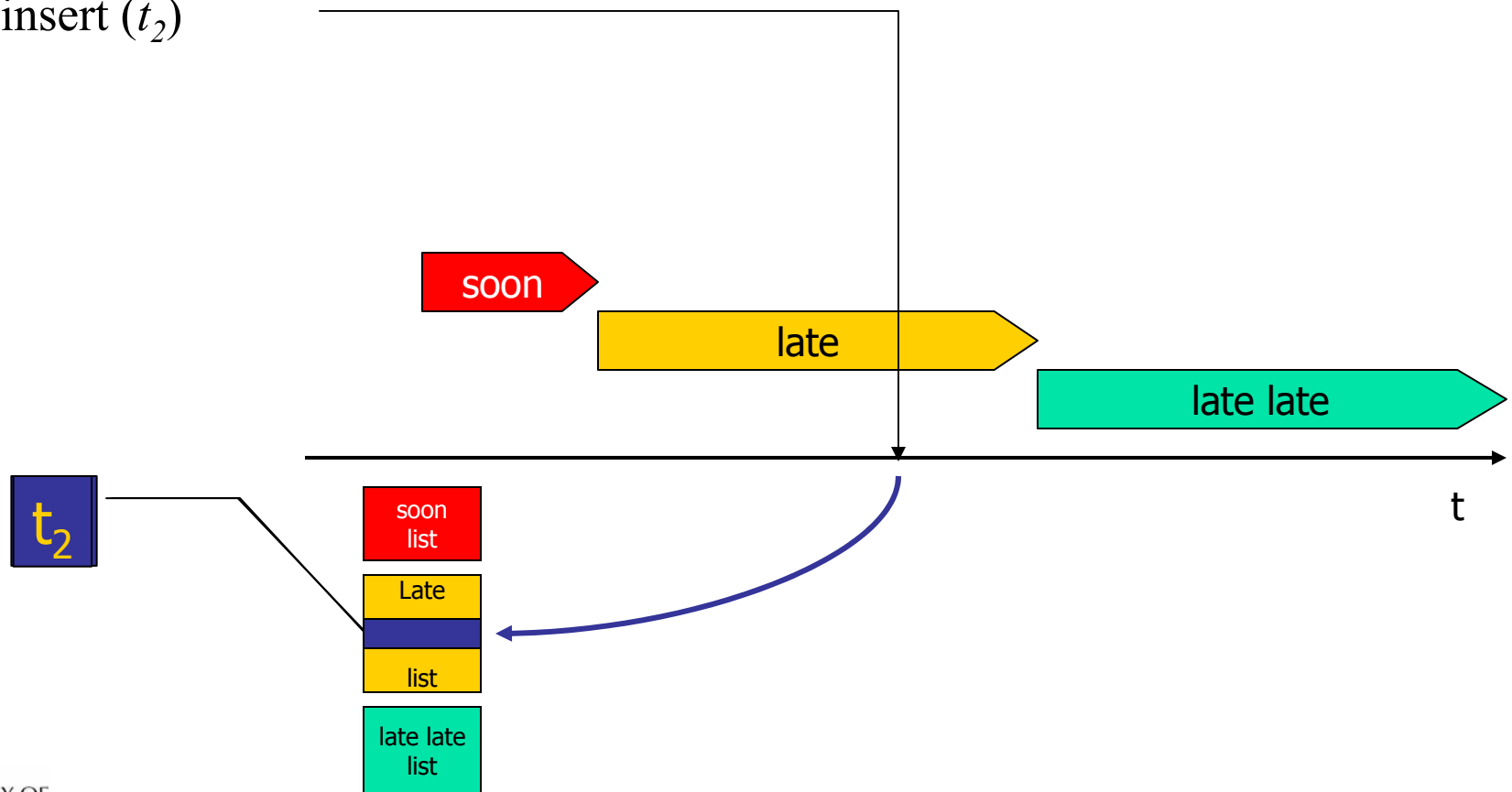
insert (t_1)
delete timeout



Lazy Timeouts

insert (t_1)
delete timeout

insert (t_2)



Lazy Sorting

- Keep a sorted list for fast lookup
- Don't sort on insert
 - insert is common
 - but timeouts are uncommon
- Sort lazily:
 - sort when walking wakeup list
 - thus we sort only when necessary



Incremental Sorting

- Combine the cost of sorting with cost of finding first thread to wake
- Problem: every addition to list resets the sorted flag, and thus we must perform entire list walk. But we want to avoid this.
- Alternative: maintain sorted list, and unsorted list. Merge the two lists when necessary.
 - merge can be incremental bubble sort
 - iow: we keep a list of new additions, so that we can remove the additions, without requiring a resort



Security

Is your system secure?



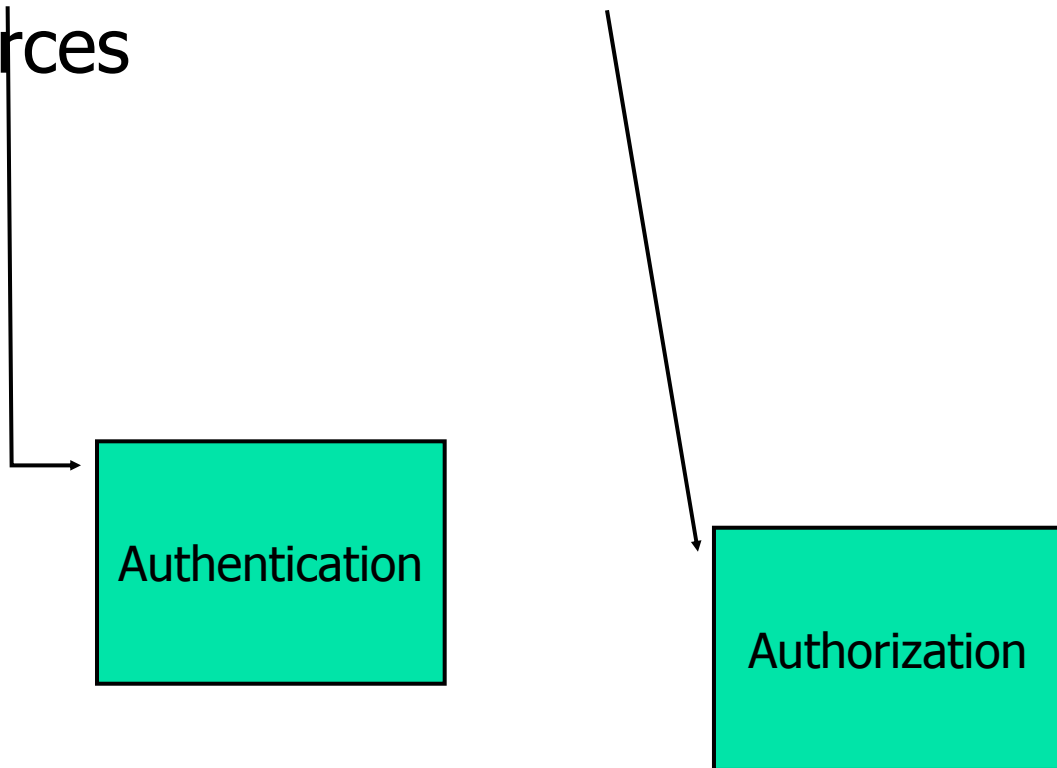
Security defined by policy

- Examples
 - All users have access to all objects
 - Physical access to servers is forbidden
 - Users only have access to their own files
 - Users have access to their own files, group access files, and public files (UNIX)

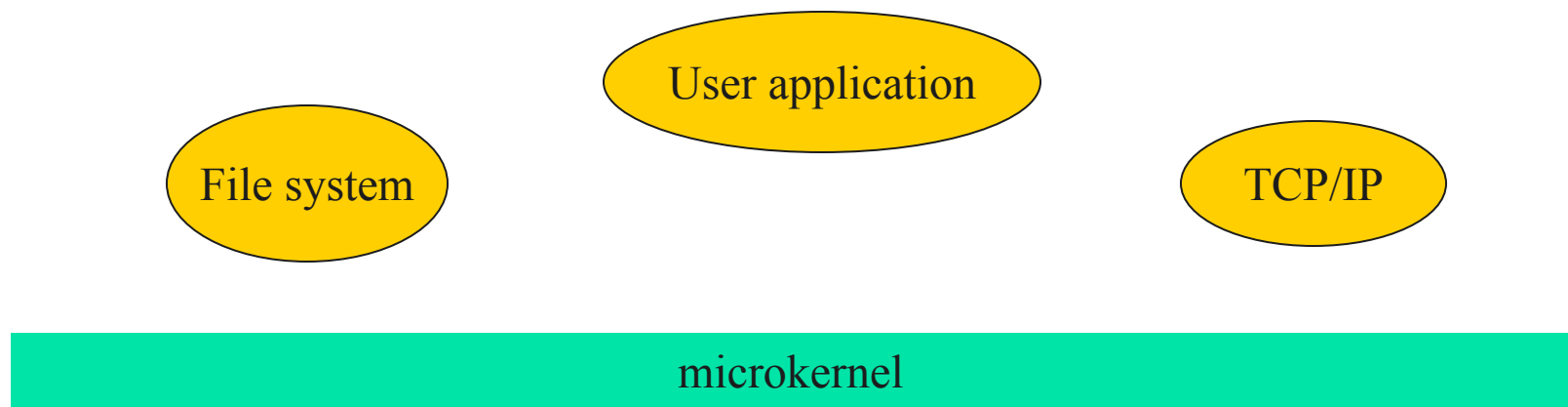


Security policy

- Specifies who has what type of access to which resources



The Microkernel Approach



All access is via IPC

- What microkernel mechanisms are needed for security?
 - How do we authenticate?
 - How do we perform authorization?
 - How do we implement arbitrary security policies?
 - How do we *enforce* arbitrary security policies?



Authentication

- Unforgeable thread identifiers
 - Thread identifiers can be mapped to
 - Tasks
 - Users
 - Groups
 - Machines
 - Domains
 - Authentication is outside the microkernel, any policy can be implemented.



Authorization

- Servers implement objects; clients access objects via IPC.
- Servers receive unforgeable client identities from the IPC mechanism
 - Servers can implement arbitrary access control policy
- No special mechanisms needed in the microkernel

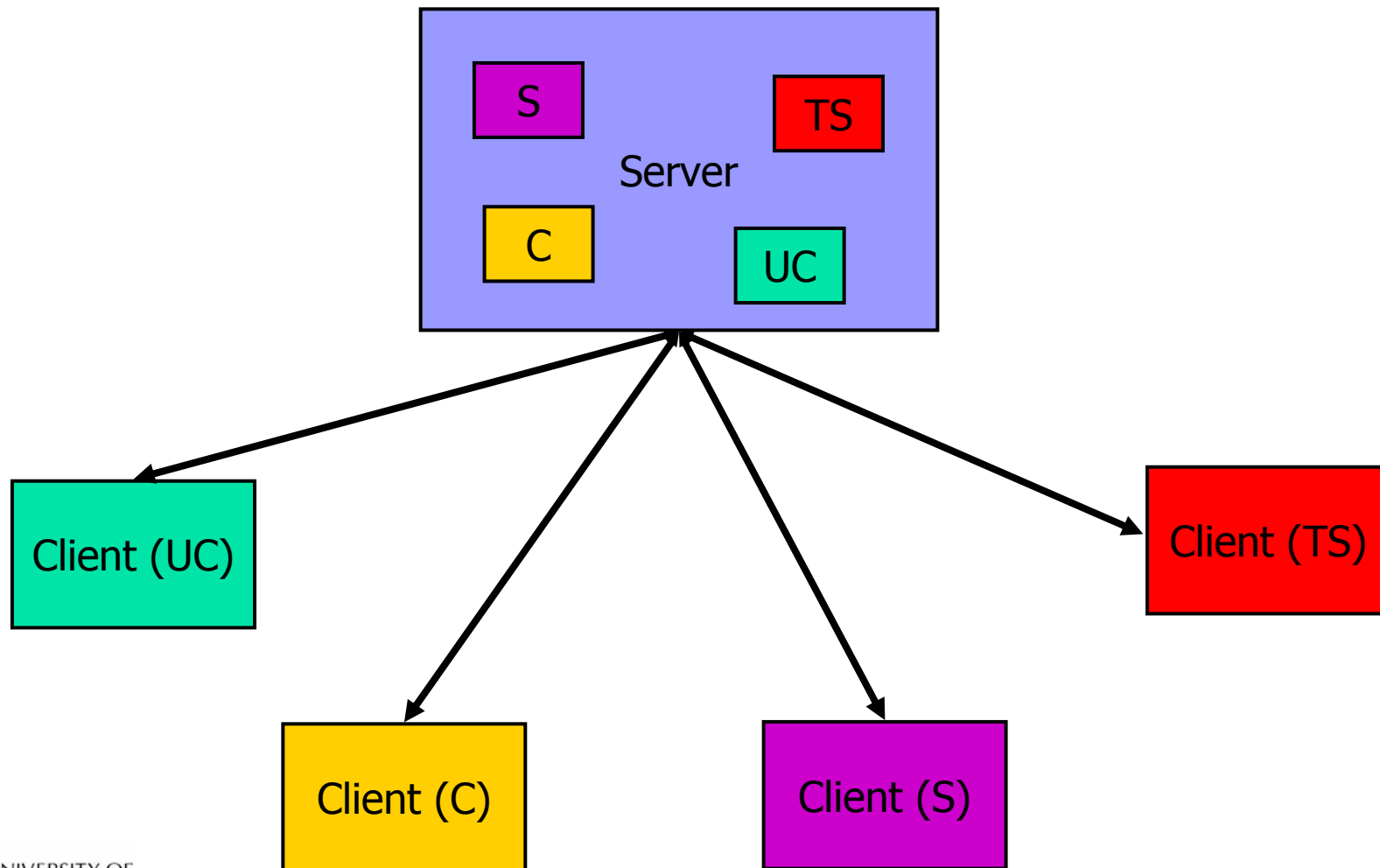
Is this really true???



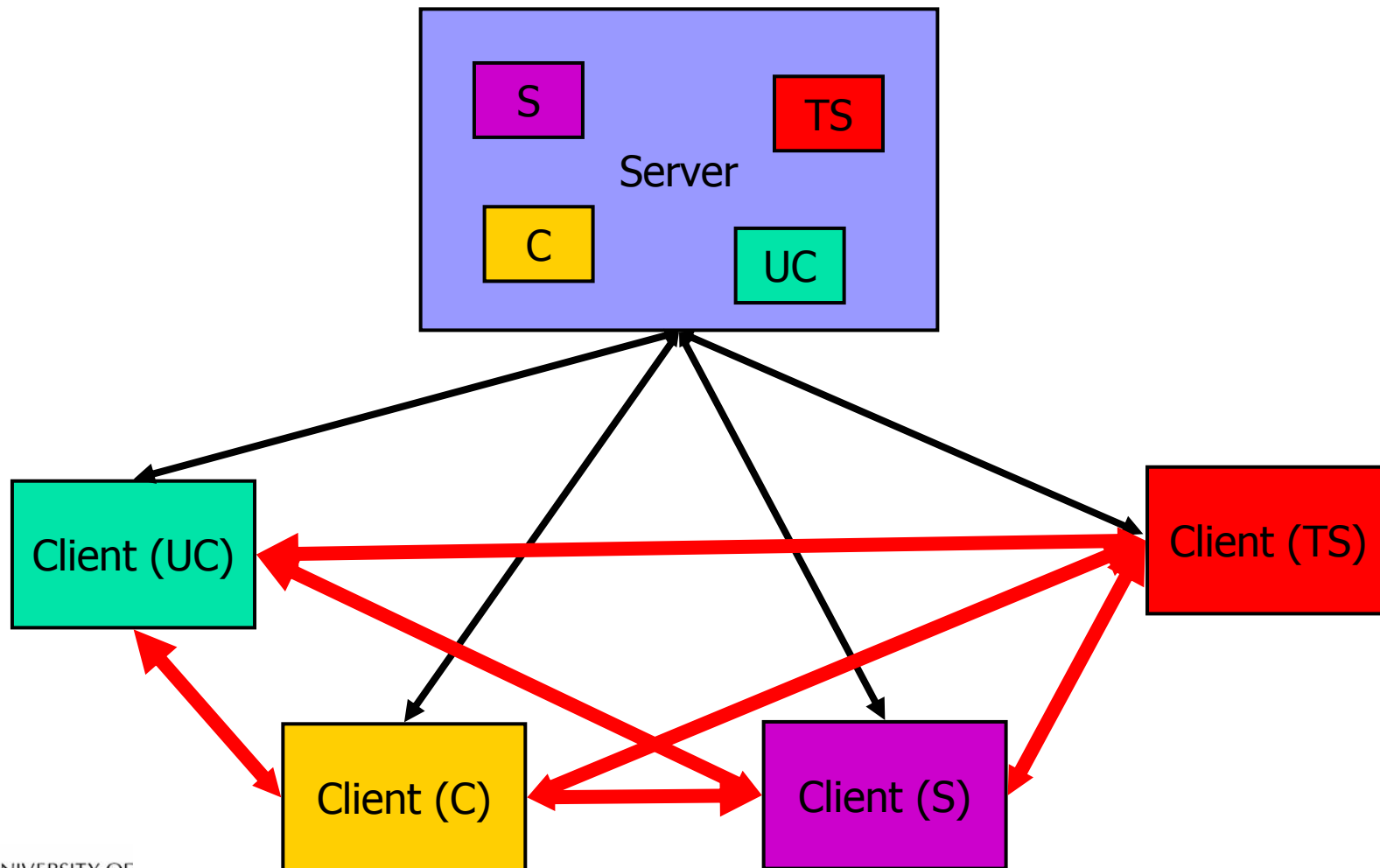
Example Policy: Mandatory Access Control

- Objects assigned security levels
 - Top Secret, Secret, Classified, Unclassified
 - $TS > S > C > UC$
- Subjects (users) assigned security levels
 - Top Secret, Secret, Classified, Unclassified
- A subject (S) can read an object (O) iff
 - $level(S) \geq level(O)$
- A subject (S) can write an object (O) iff
 - $level(S) \leq level(O)$

Secure System



Problem



Conclusion

To control information flow we must control communication

- We need mechanisms to not only implement a policy – we must also be able to *enforce* a policy!!!
- Mechanism should be flexible enough to implement and enforce all relevant security policies.



Clans & Chiefs



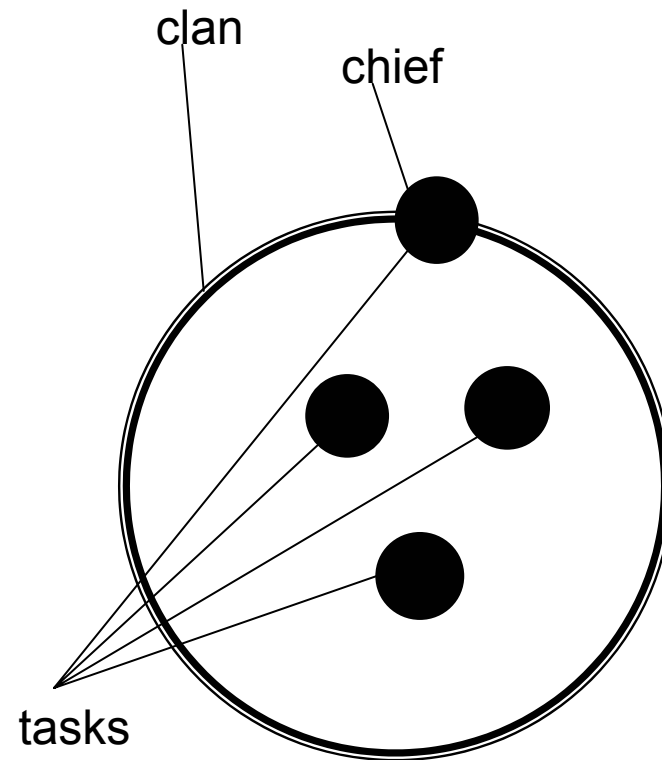
Clans & Chiefs

Within all system based on direct message transfer, protection is essentially a matter of message control. Using access control lists can be done at the server level, but maintenance of large distributed access control lists becomes hard when access rights change rapidly. The clan concept permits to complement the mentioned passive entity protection by active protection based on intercepting all communication of suspicious subjects.

A *clan* is a set of tasks headed by a *chief* task. Inside the clan all messages are transferred freely and the kernel guarantees message integrity. But whenever a message tries to cross a clan's borderline, regardless of whether it is outgoing or incoming, it is redirected to the clan's chief. This chief may inspect the message (including the sender and receiver ids as well as the contents) and decide whether or not it should be passed to the destination to which it was addressed. Obviously subject restriction and local reference monitors can be implemented outside the kernel by means of clans. Since chief are tasks at user level, the clan concept allows more sophisticated and user definable checks as well as active control.

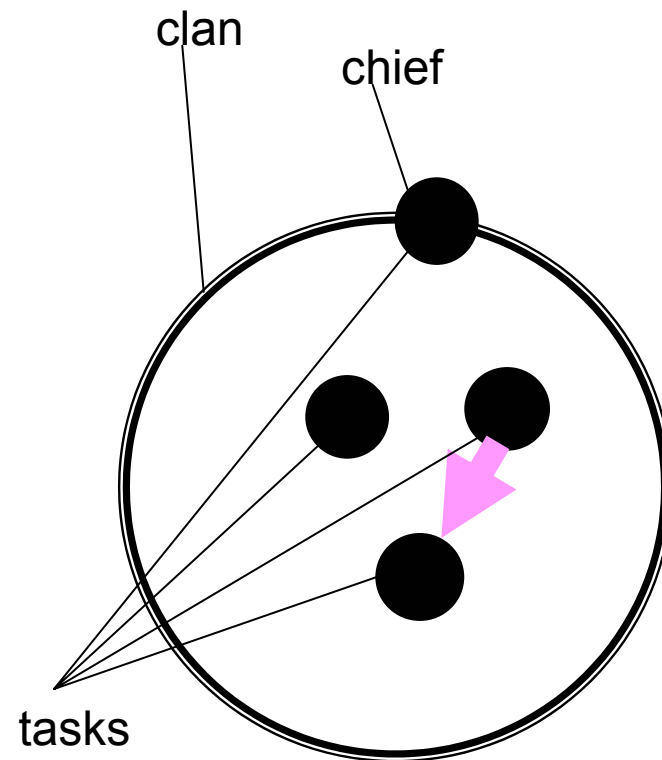


Clans & Chiefs



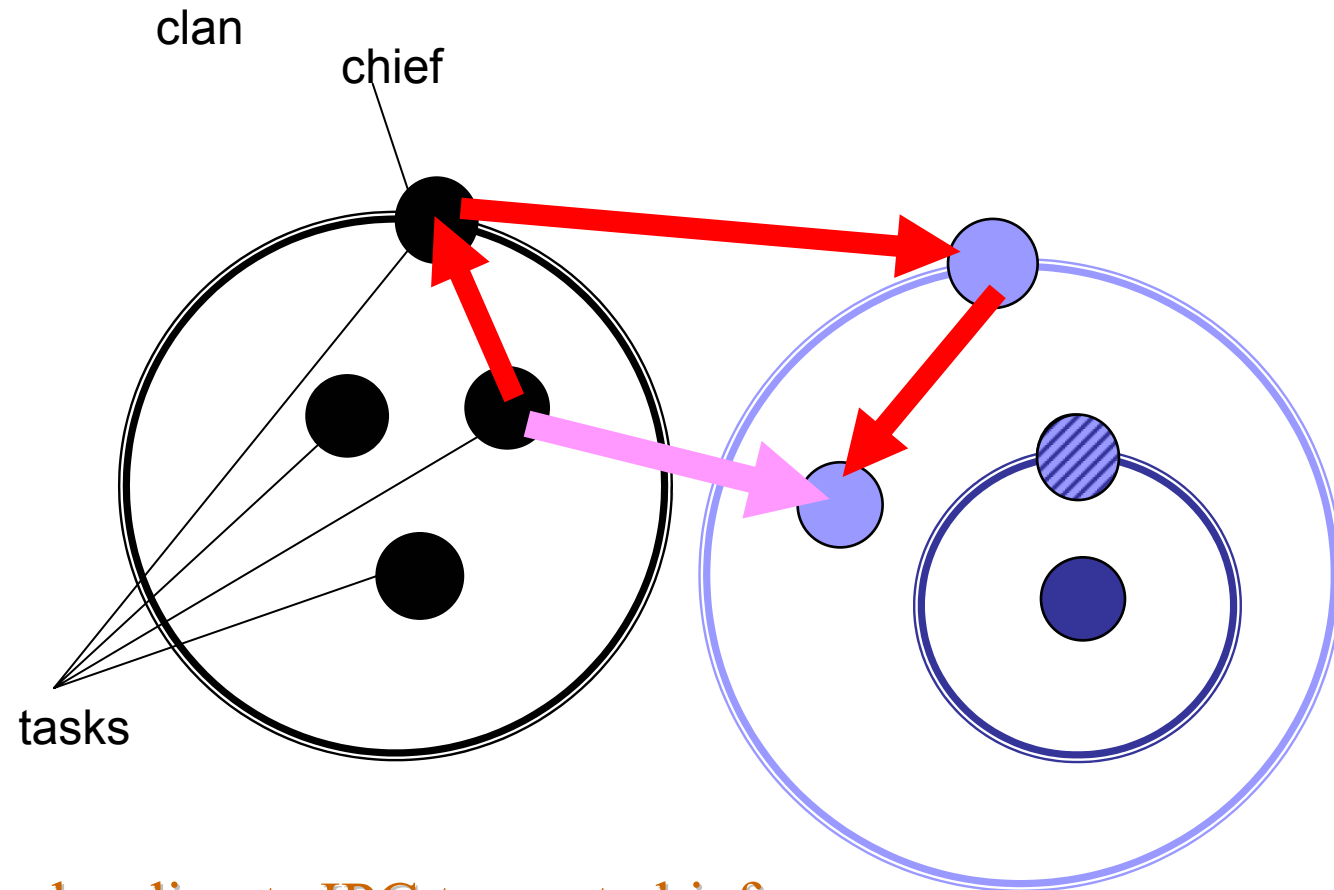
- A *clan* is a set of tasks headed by a chief task

Intra-Clan IPC



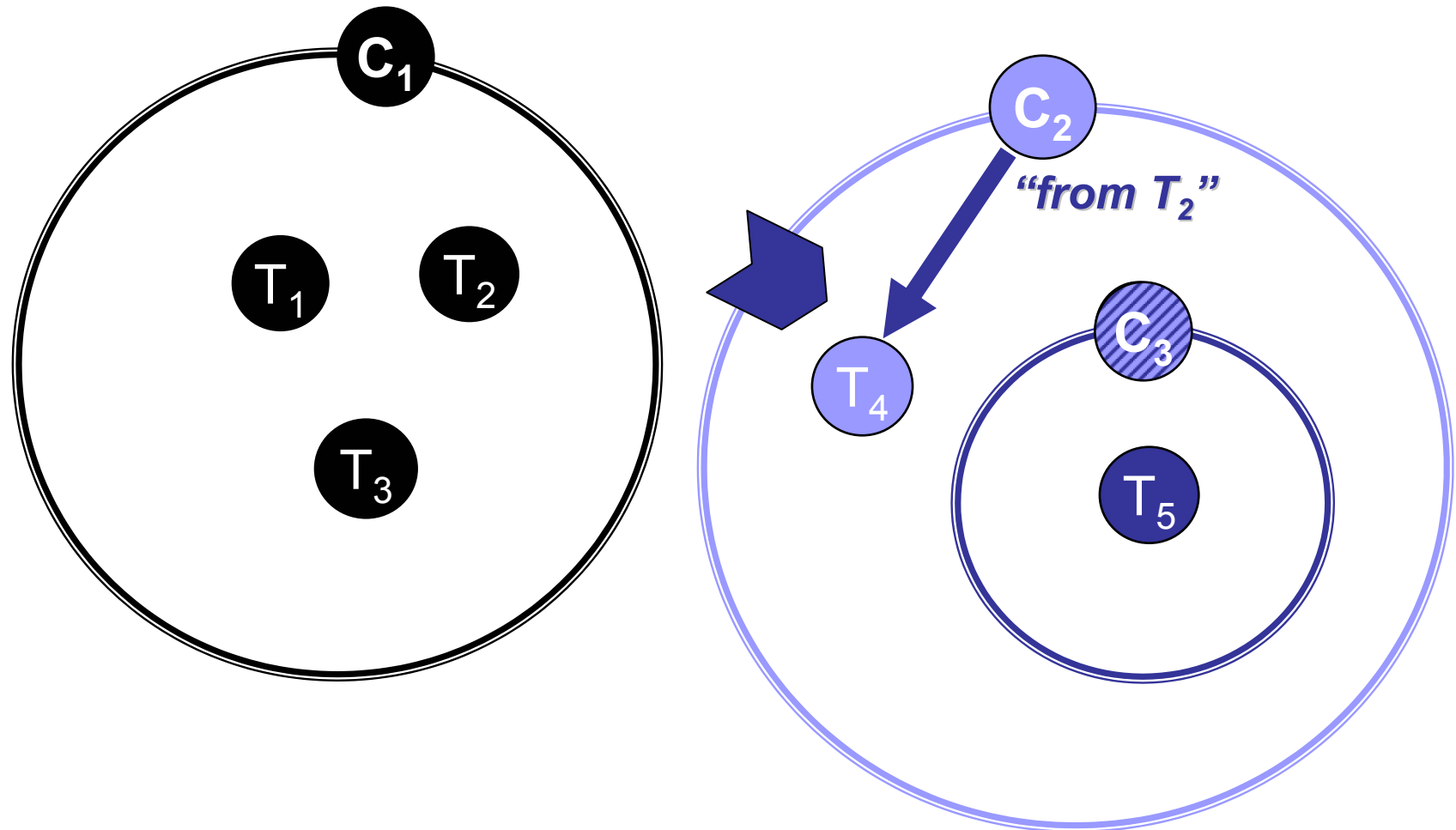
- Direct IPC by microkernel

Inter-Clan IPC

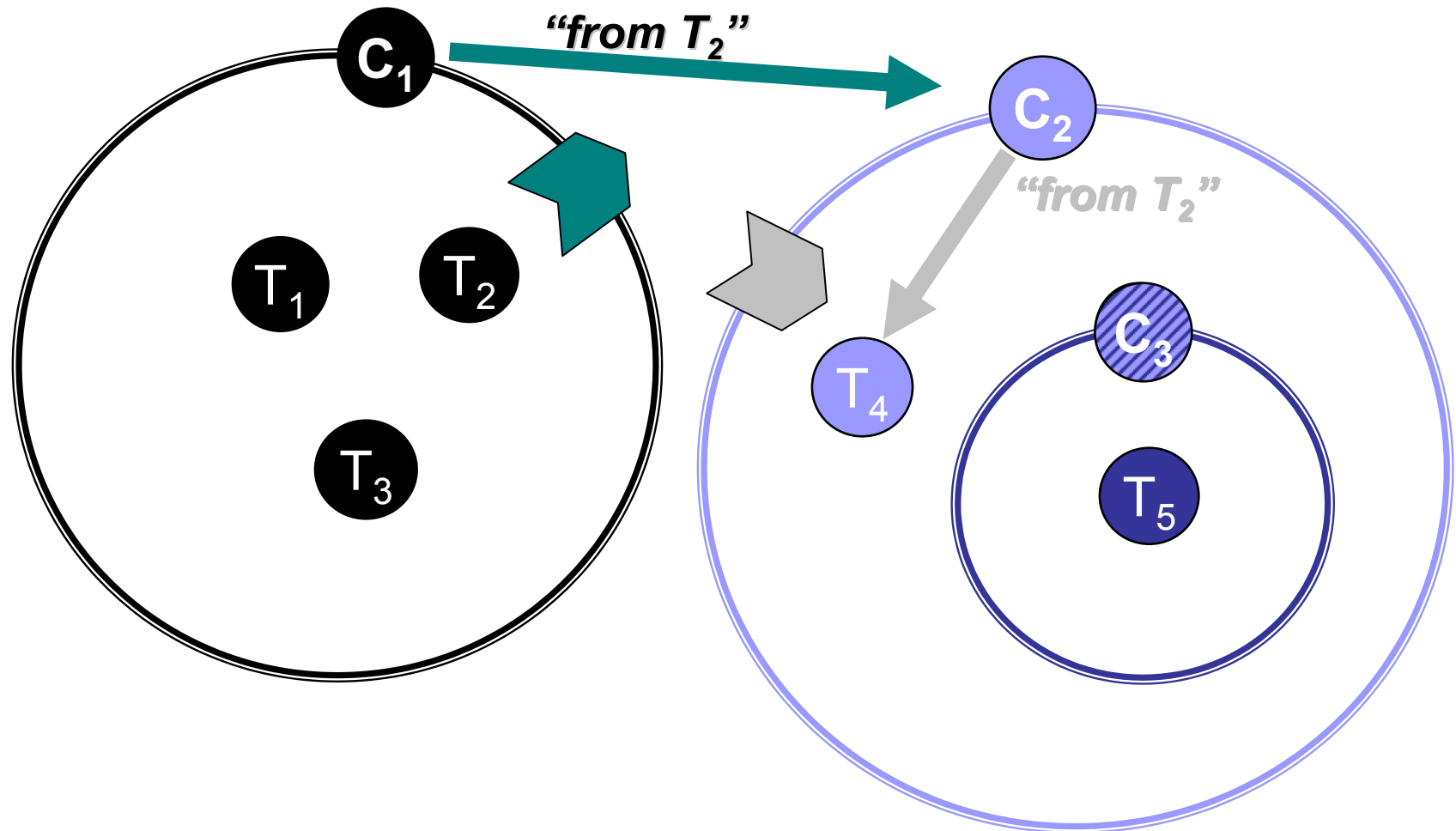


- Microkernel redirects IPC to next chief
- Chief (user task) *can* forward IPC or modify or ...

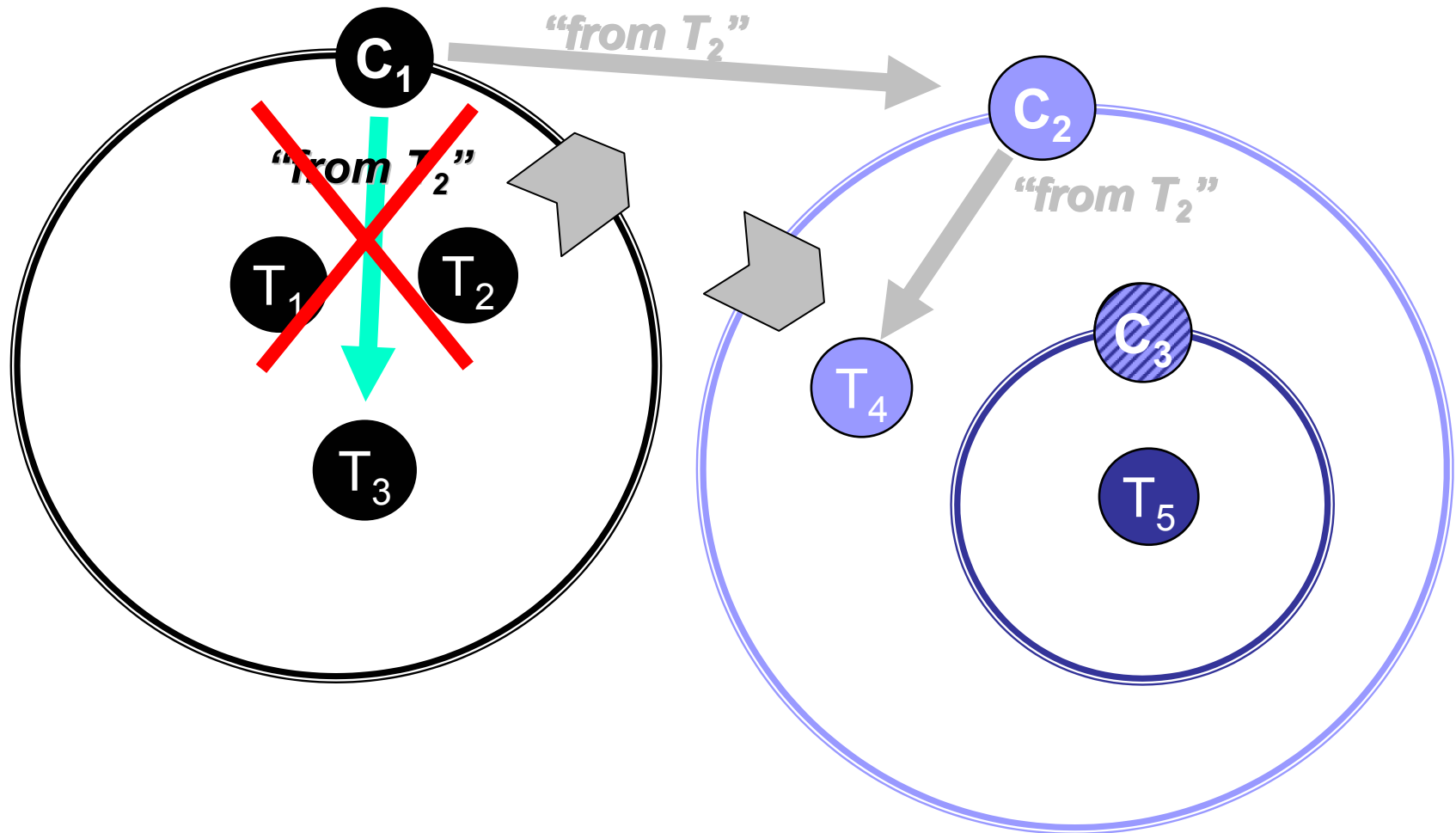
Direction-Preserving Deceiving



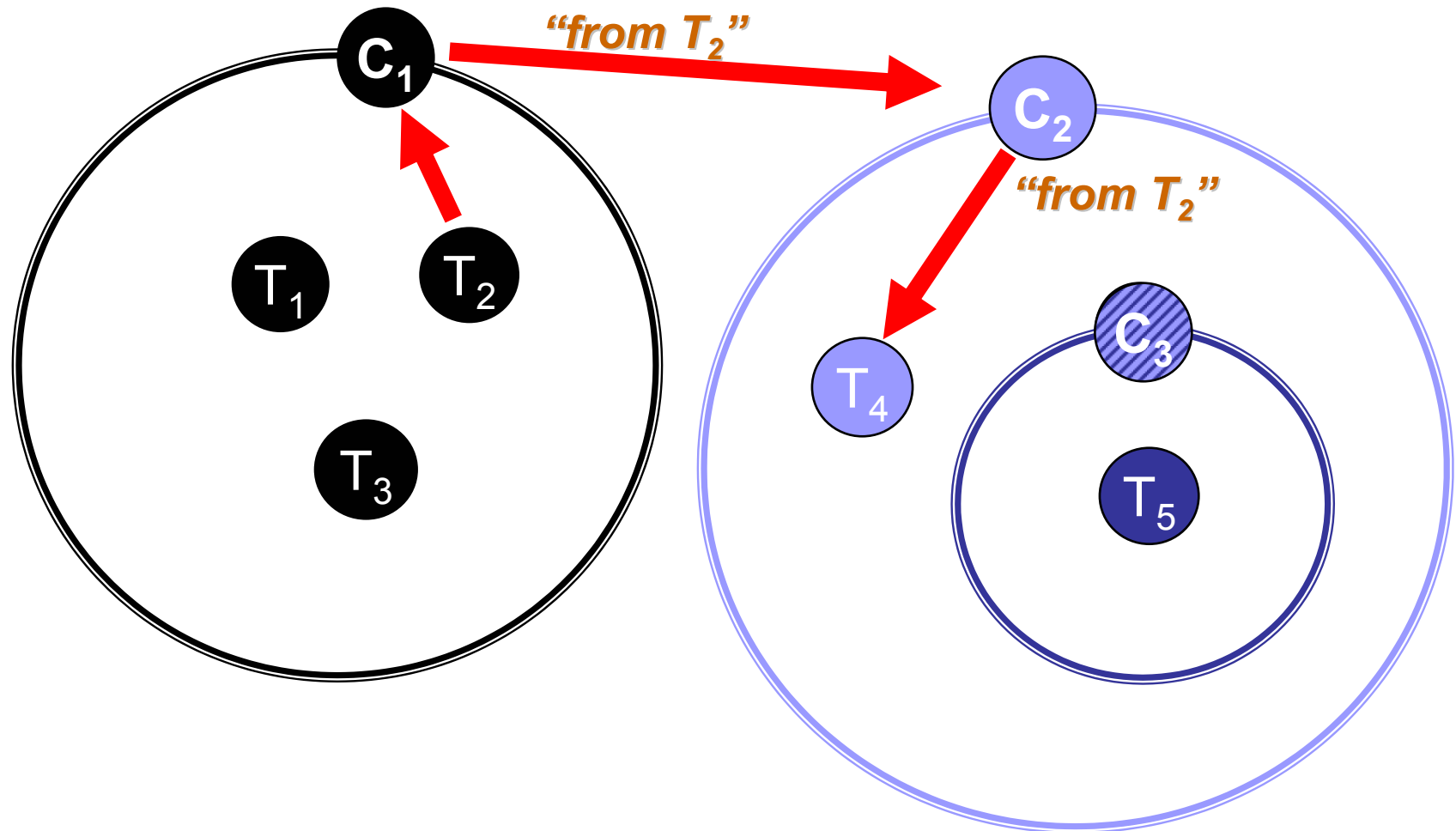
Direction-Preserving Deceiving



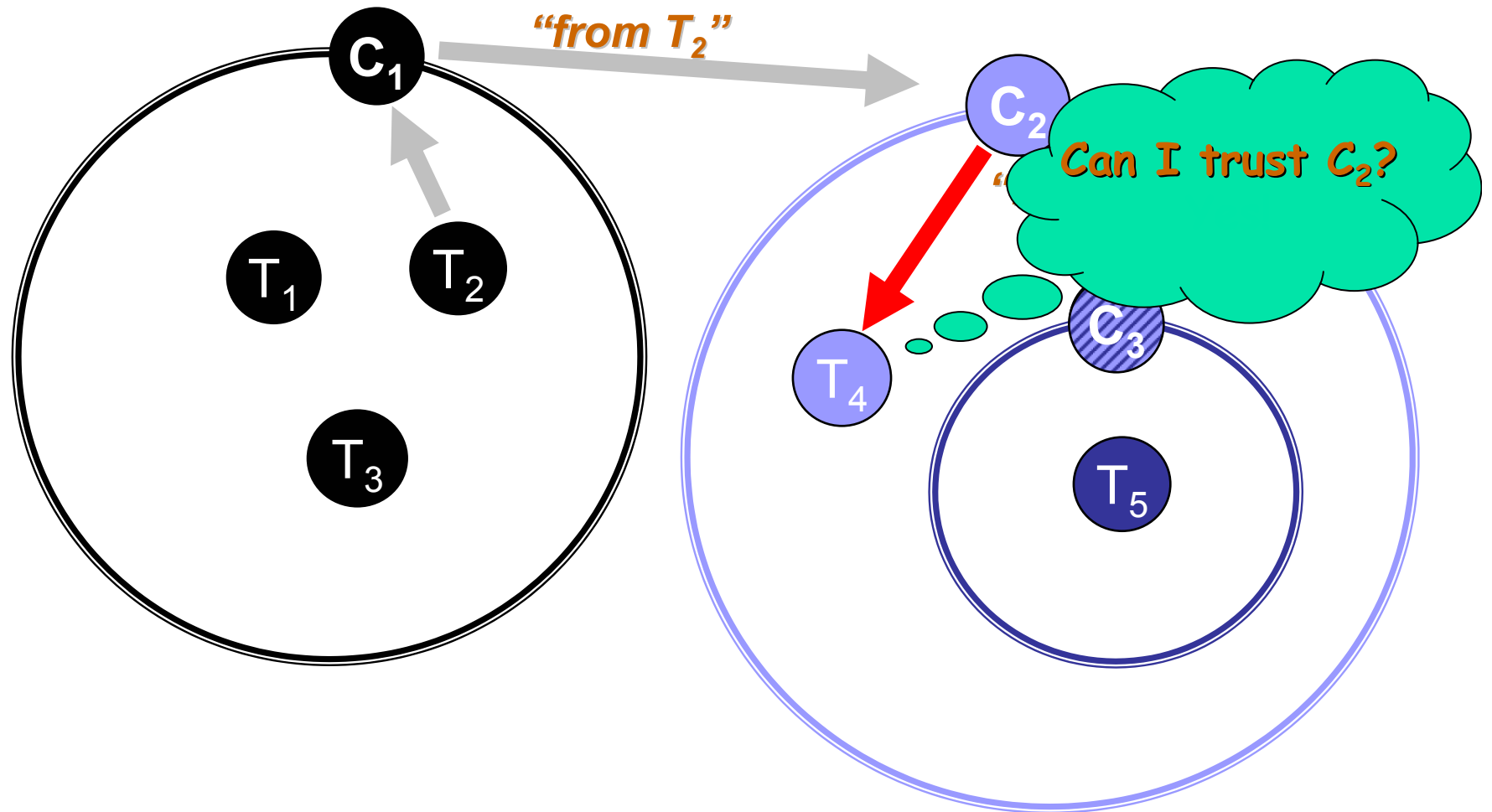
Direction-Preserving Deceiving



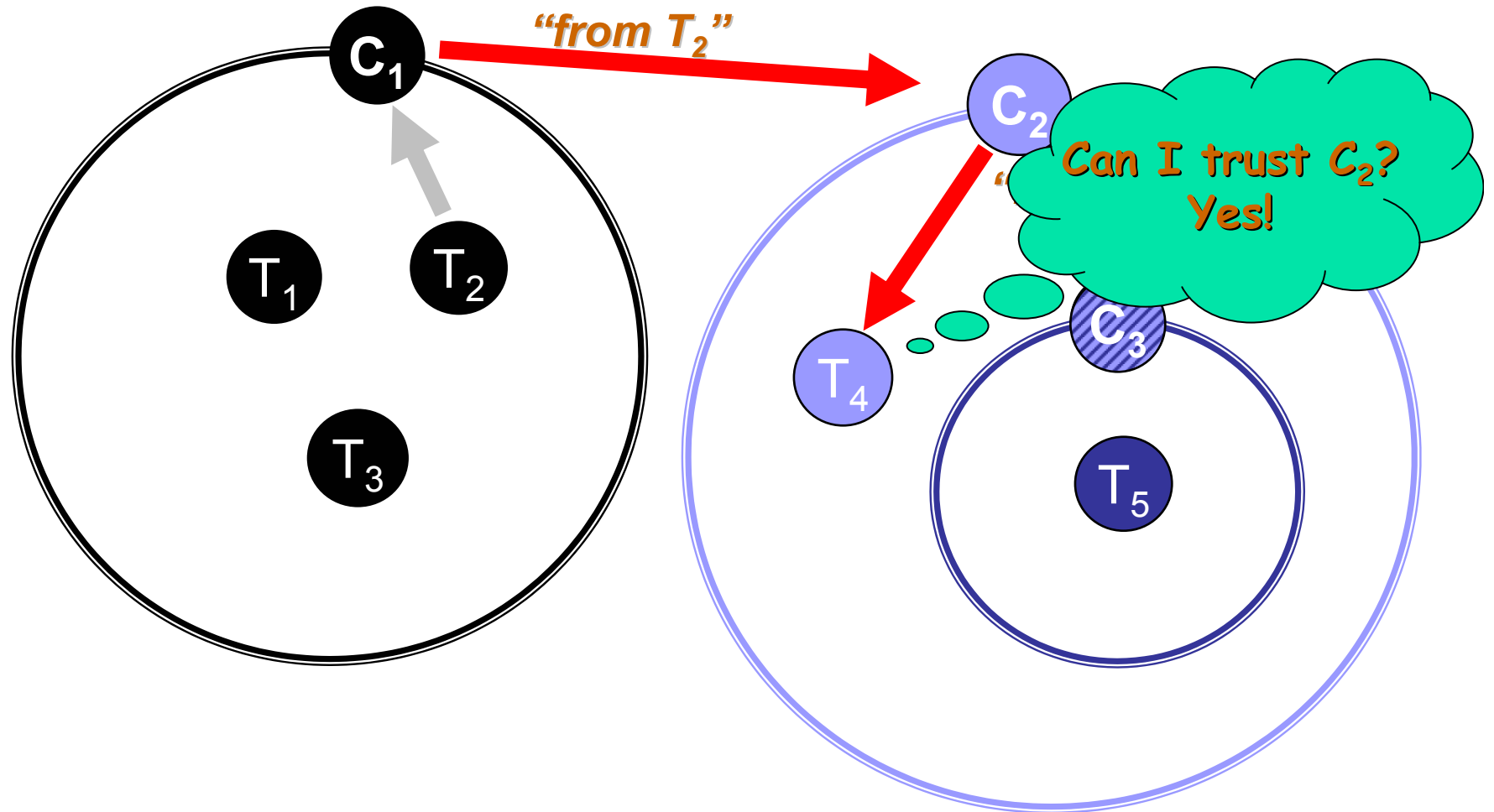
Direction-Preserving Deceiving



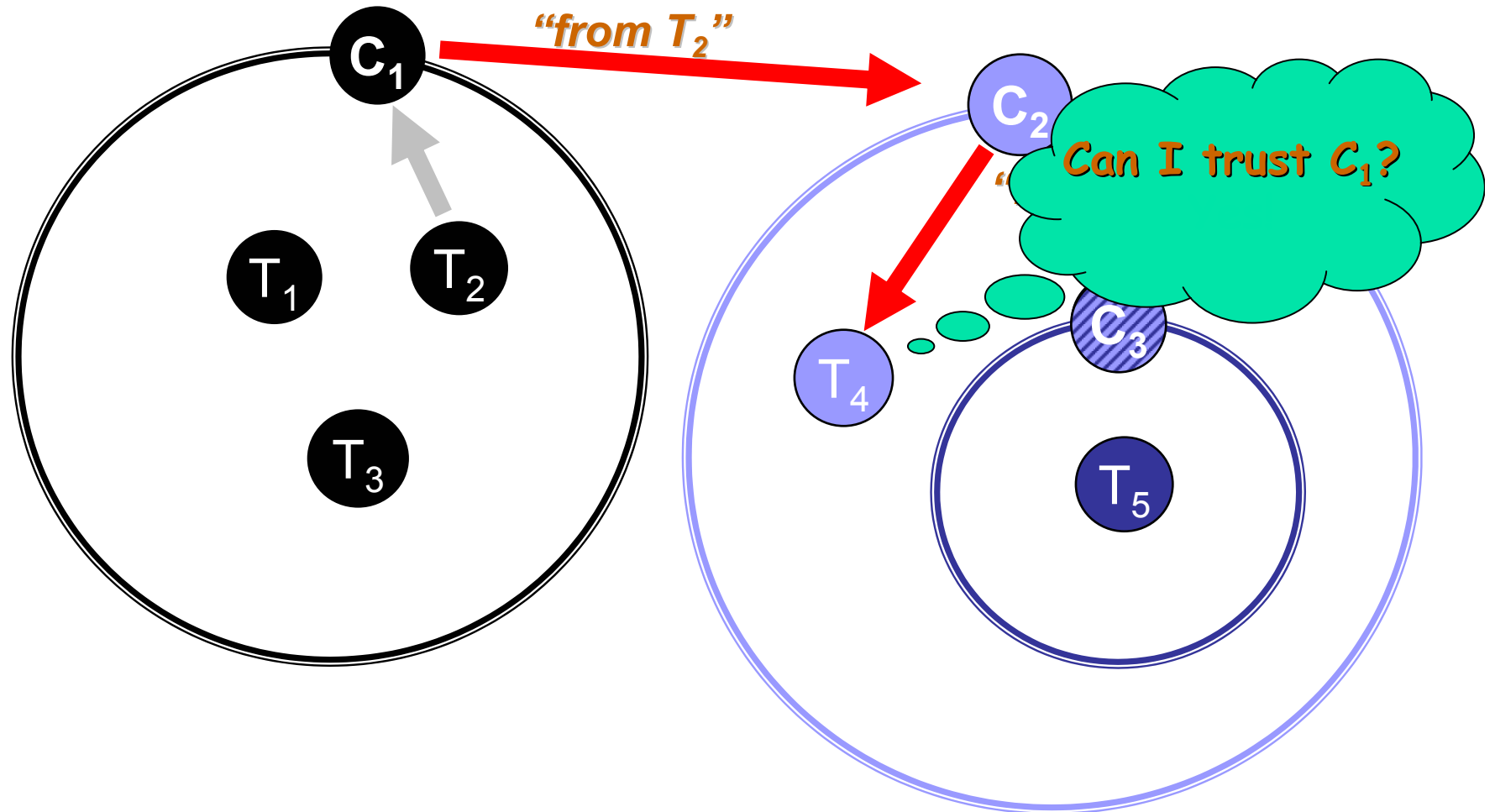
Direction-Preserving Deceiving



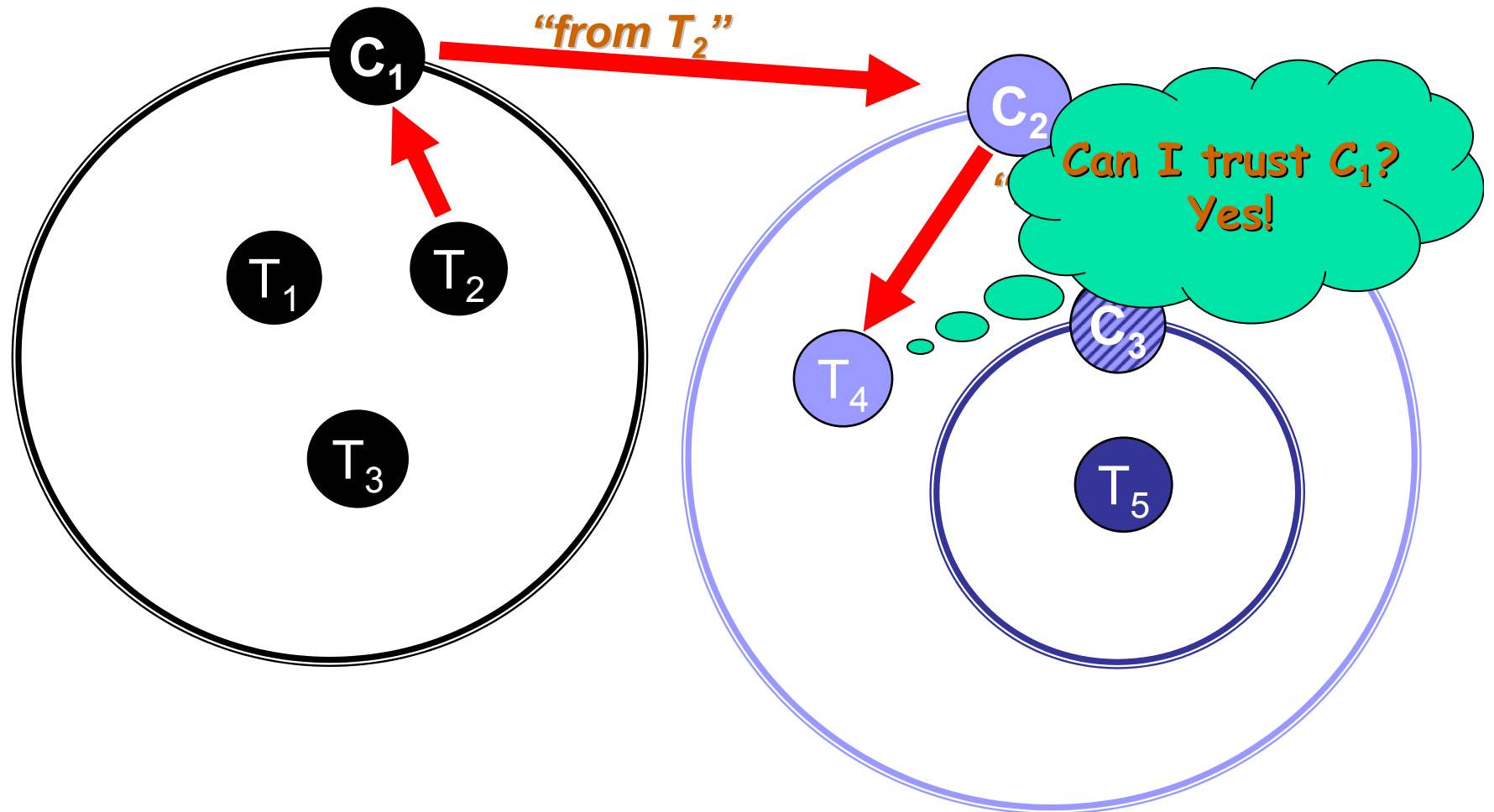
Direction-Preserving Deceiving



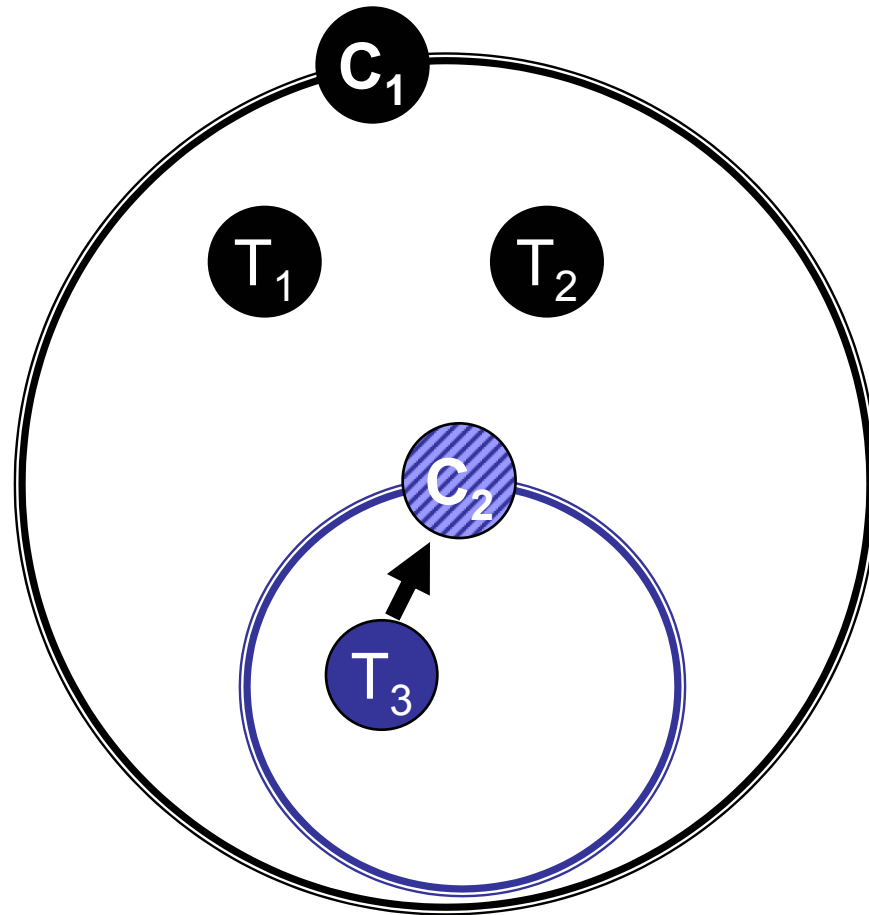
Direction-Preserving Deceiving



Direction-Preserving Deceiving



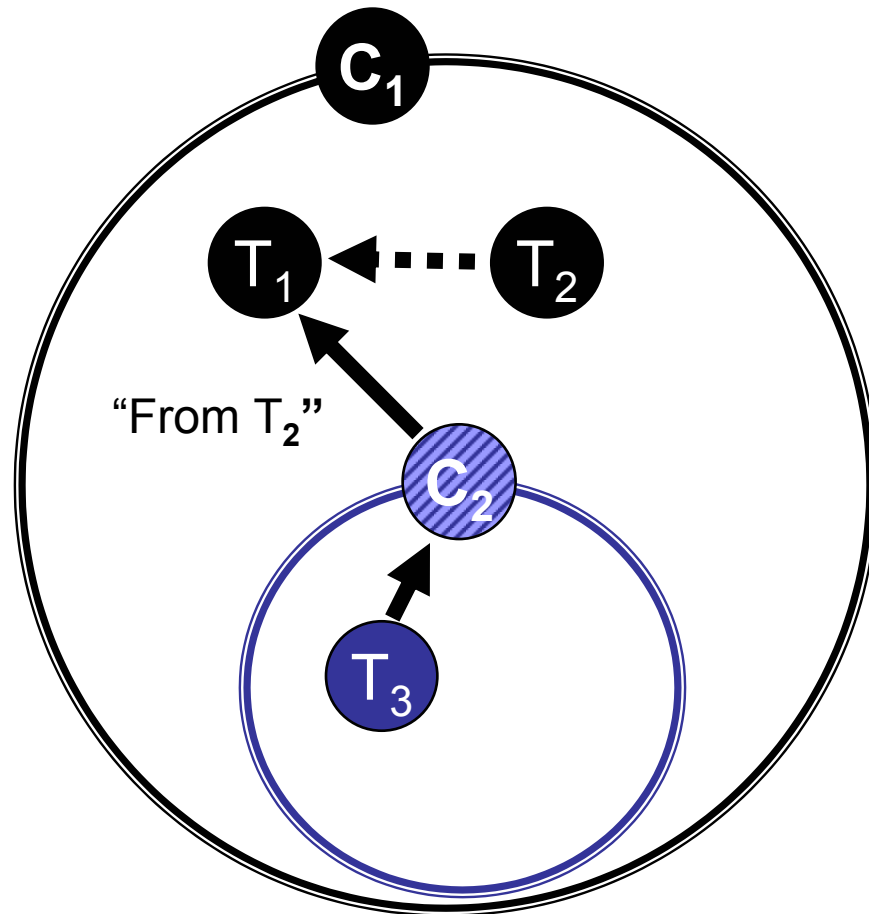
Example



Direct-Preserving-Deceiving (DPD) is a simple mechanism to realize security.

Imagine the blue task is a tool you have from the Internet. Without DPD there is no relevant security. The blue thread T_3 wants to get some private information from T_1 .

Example

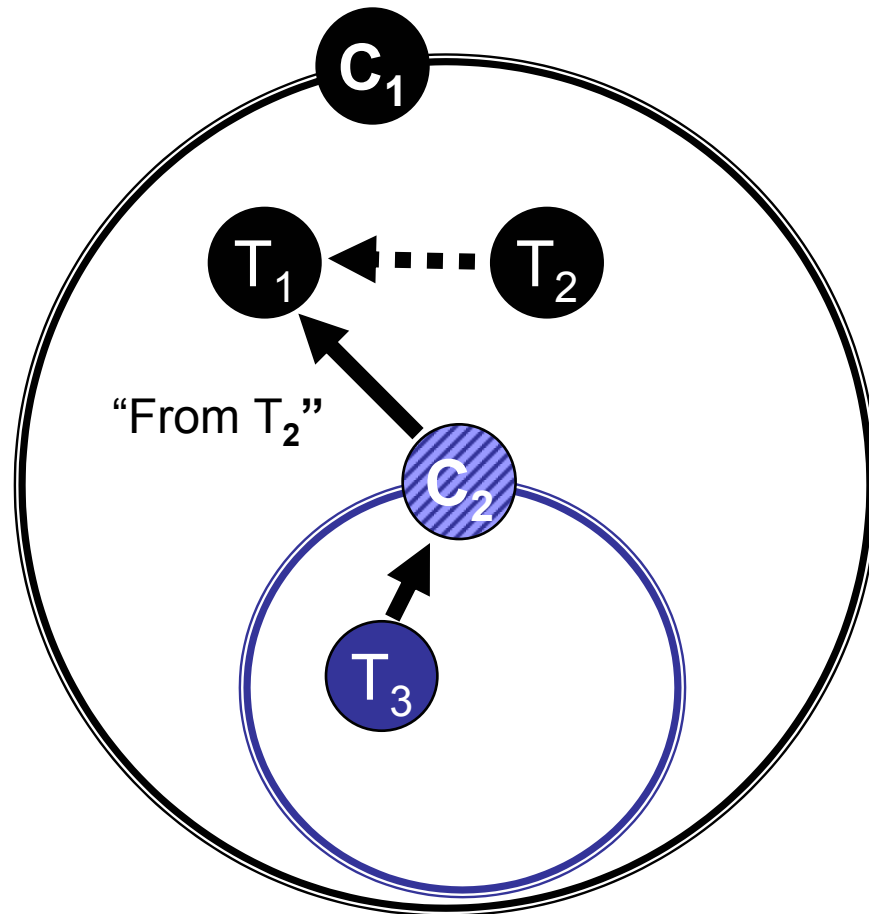


Direct-Preserving-Deceiving (DPD) is a simple mechanism to realize security.

Imagine the blue task is a tool you have from the Internet. Without DPD there is no relevant security. The blue thread T_3 want to get some private information from T_1 .

The chief C_2 can send an IPC to T_1 so it appears that it came from T_2 .

Example



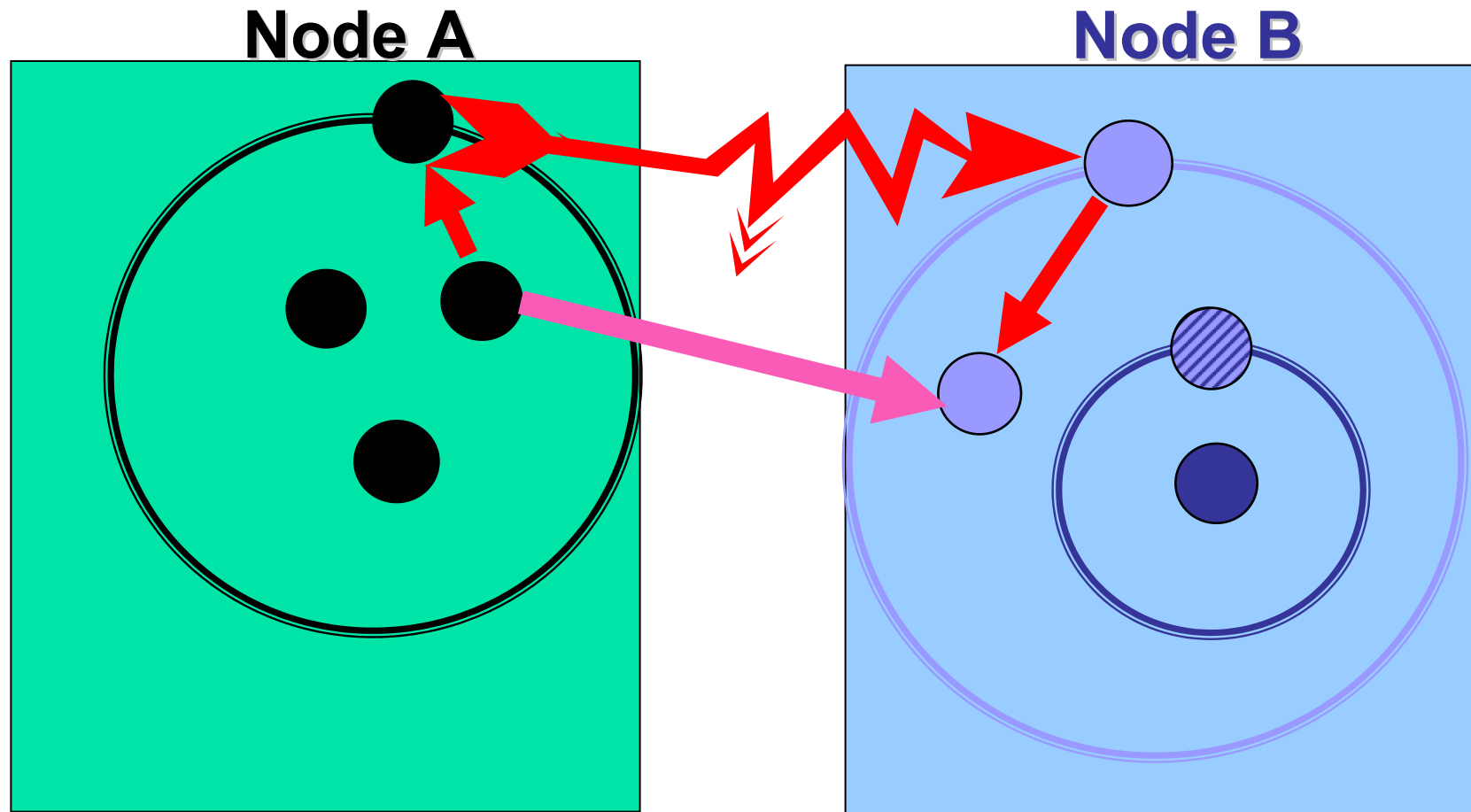
Direct-Preserving-Deceiving (DPD) is a simple mechanism to realize security.

Imagine the blue task is a tool you have from the Internet. Without DPD there is no relevant security. The blue thread T_3 want to get some private information from T_1 .

The chief C_2 can send an IPC to T_1 so it appears that it came from T_2 .

The important fact is that with DPD when T_1 gets an IPC from C_2 then he definitely knows that the message came from inside the clan C_2 . Vice versa is the same.

Remote IPC

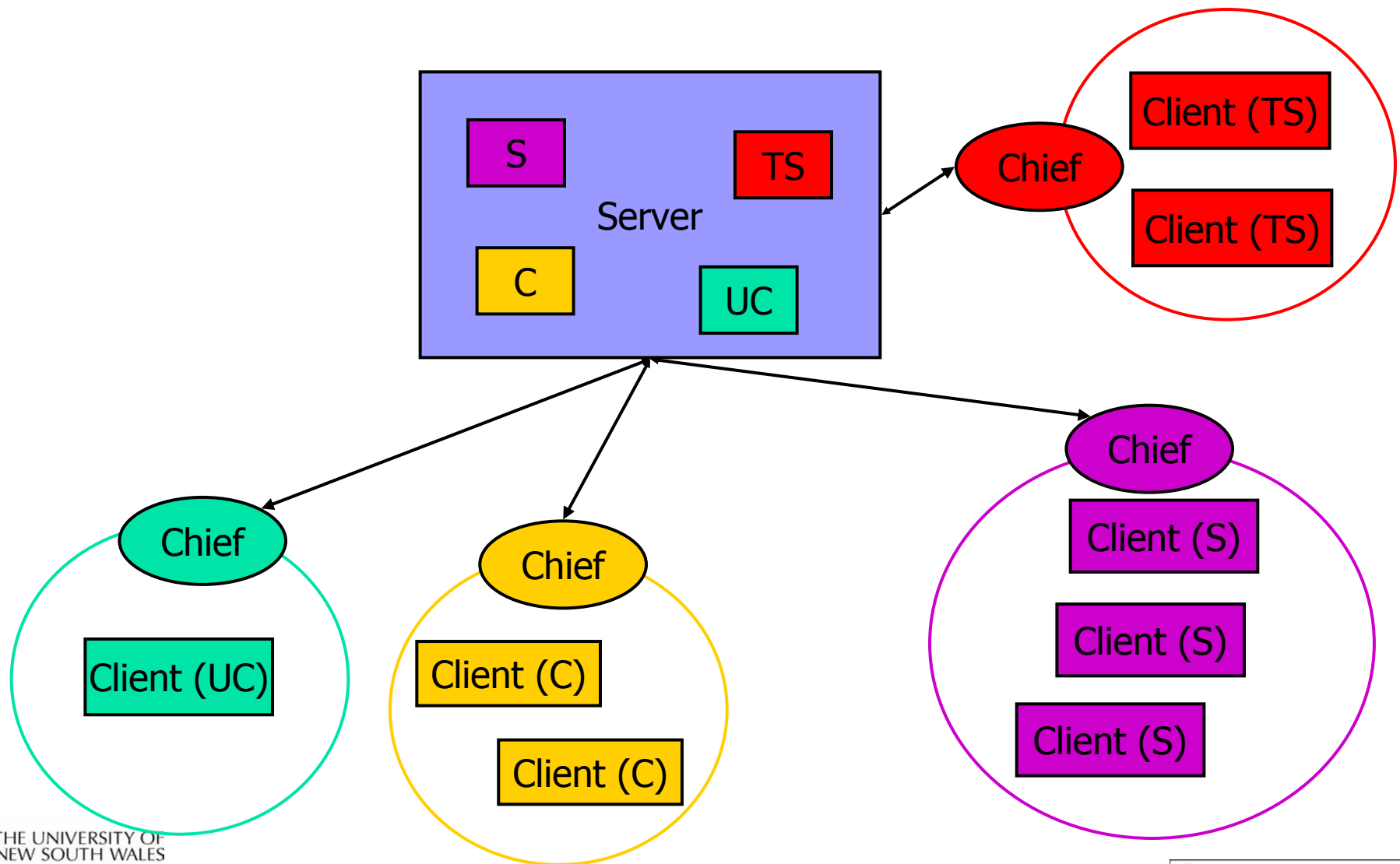


Clans & Chiefs

- Remote IPC
- Multi-level security
- Debugging
- Heterogeneity



Secure System using Clans & Chiefs



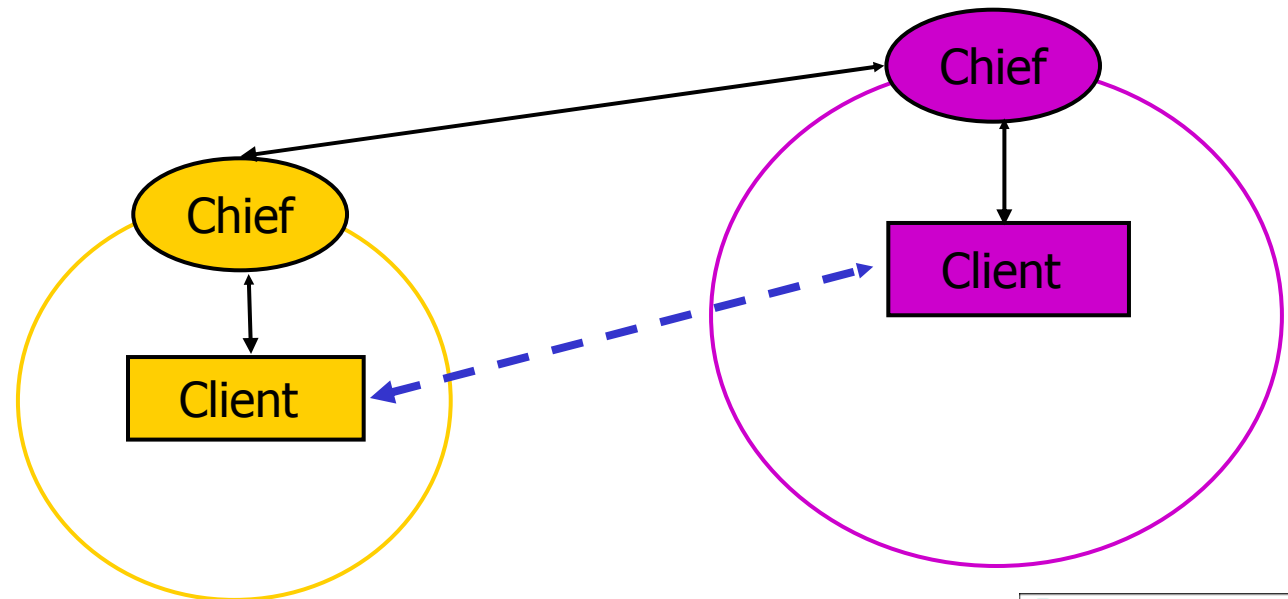
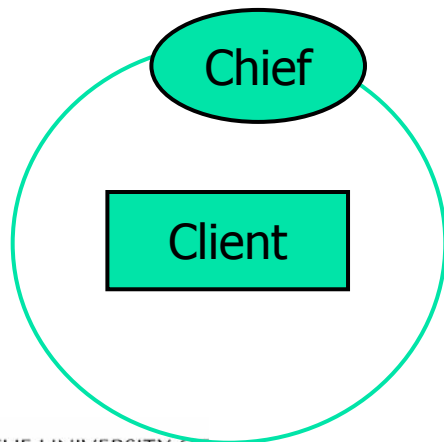
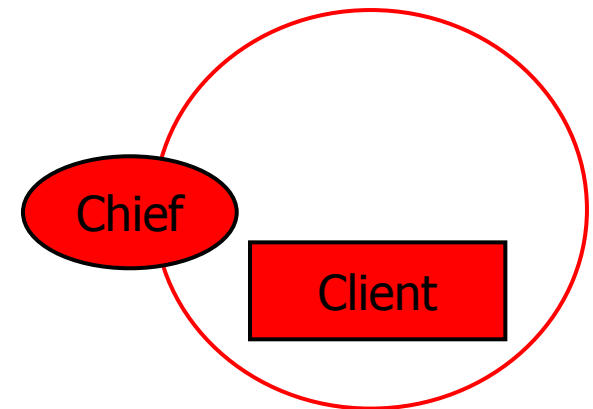
Problems with Clans & Chiefs

- Static
 - A chief is assigned when task is started
 - If we *might* want to control IPC, we must always assign a chief
- General case requires many more IPCs
 - Every task has its own chief



The most general system configuration

- If a pair could communicate freely we still require 3 IPCs where one would suffice

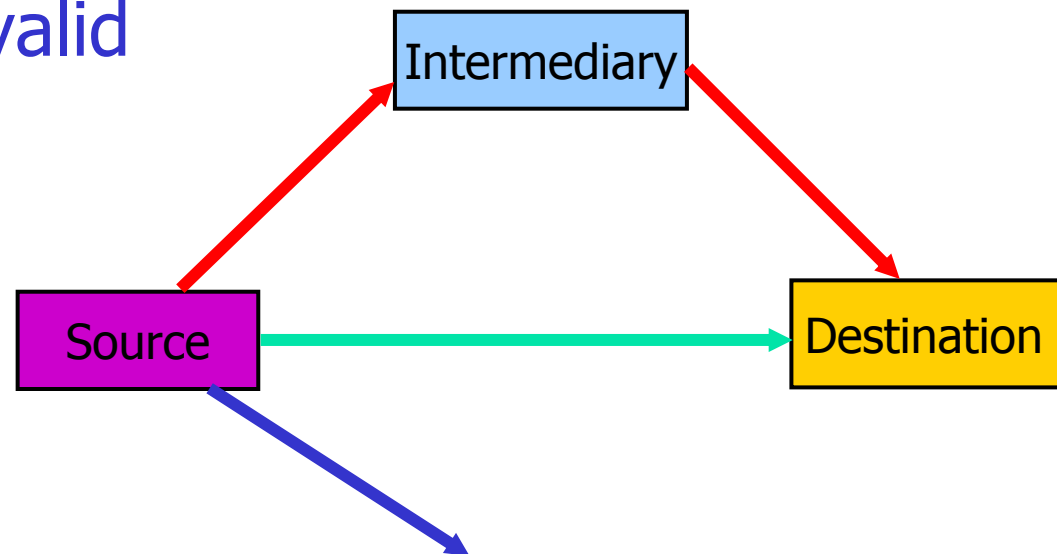


IPC Redirection



IPC Redirection

- For each source and destination we actually deliver to X , where X is one of:
 - Destination
 - Intermediary
 - Invalid



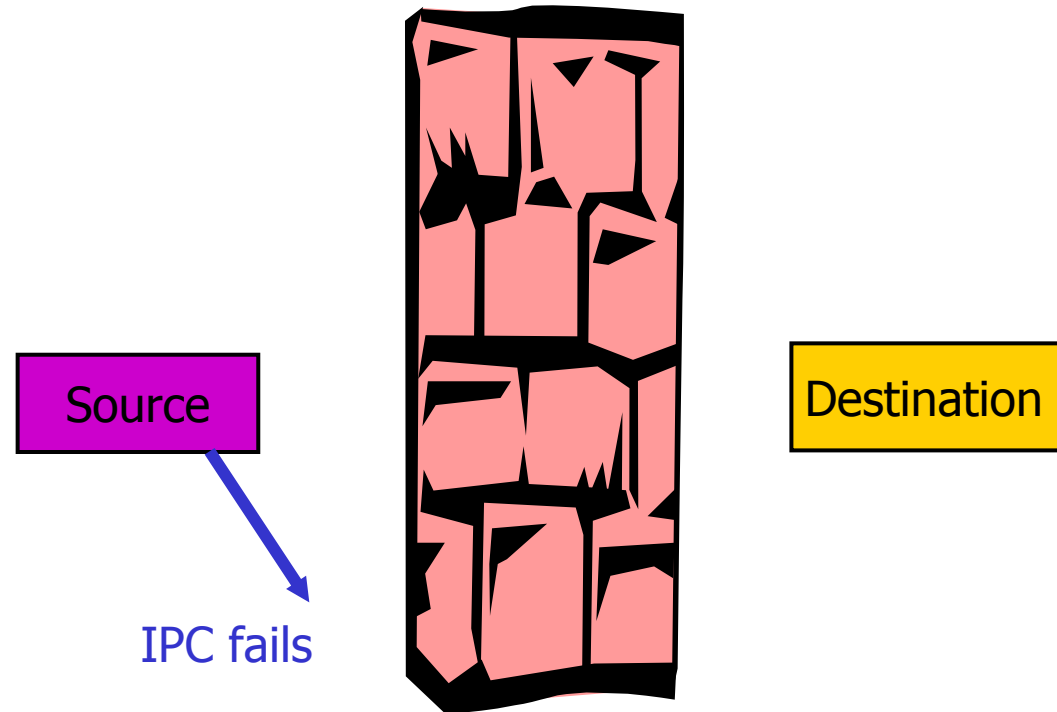
IPC Redirection

- If X is
 - Destination
 - We have a fast path when source and destination can communicate freely



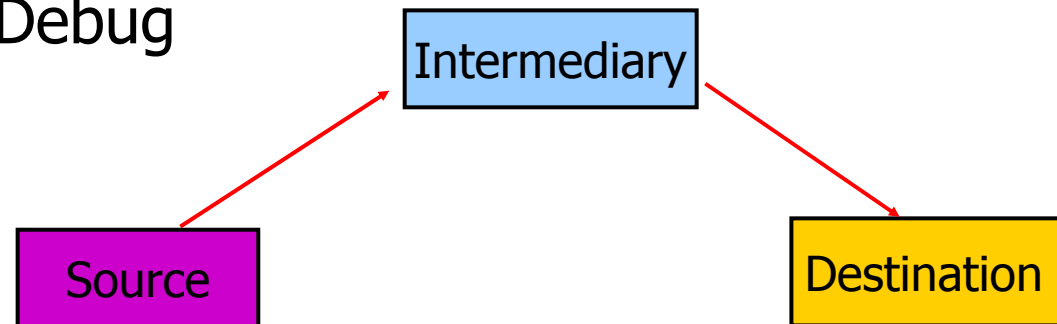
IPC Redirection

- If X is
 - Invalid
 - We have a barrier that prevents communication completely



IPC Redirection

- If X is
 - Intermediary
 - Enforce security policy
 - Monitor, analyze, reject, modify each IPC
 - Audit communication
 - Debug



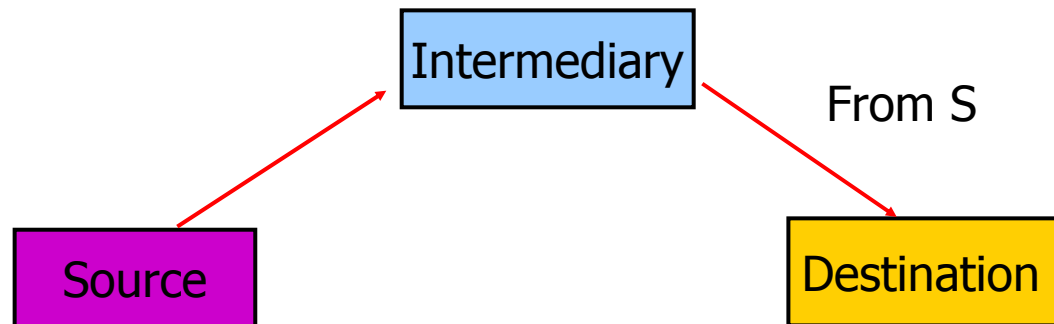
Deception

- To be able to transparently insert an intermediary, intermediaries must be able to deceive the destination into believing the intermediary is the source.
- An intermediary (I) can impersonate a source (S) in IPC to a destination (D)
 - $I [S] \Rightarrow D$
 - Iff $R(S,D) = I$ or
 - $R(S,D) = x$ and $I[x] \Rightarrow D$



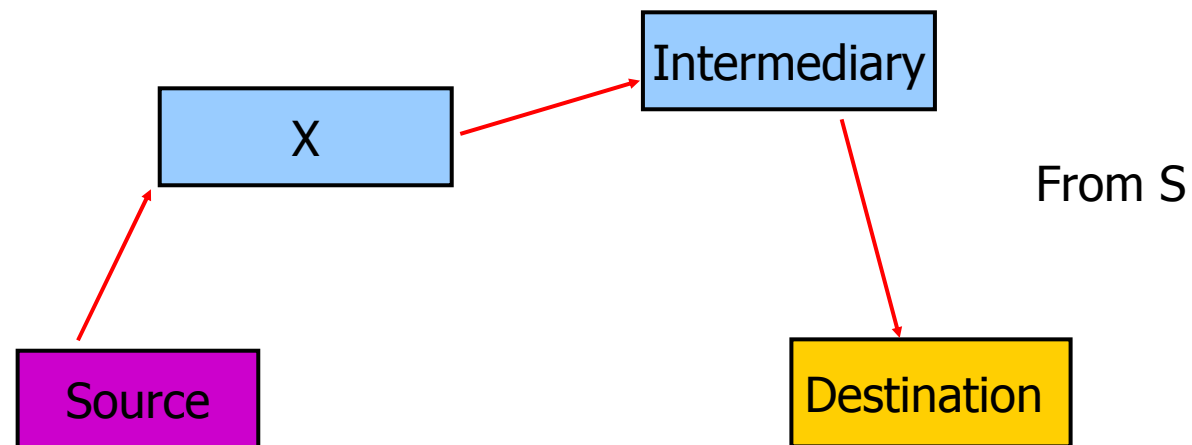
Case 1

- $I[S] \Rightarrow D$ if $R(S,D) = I$

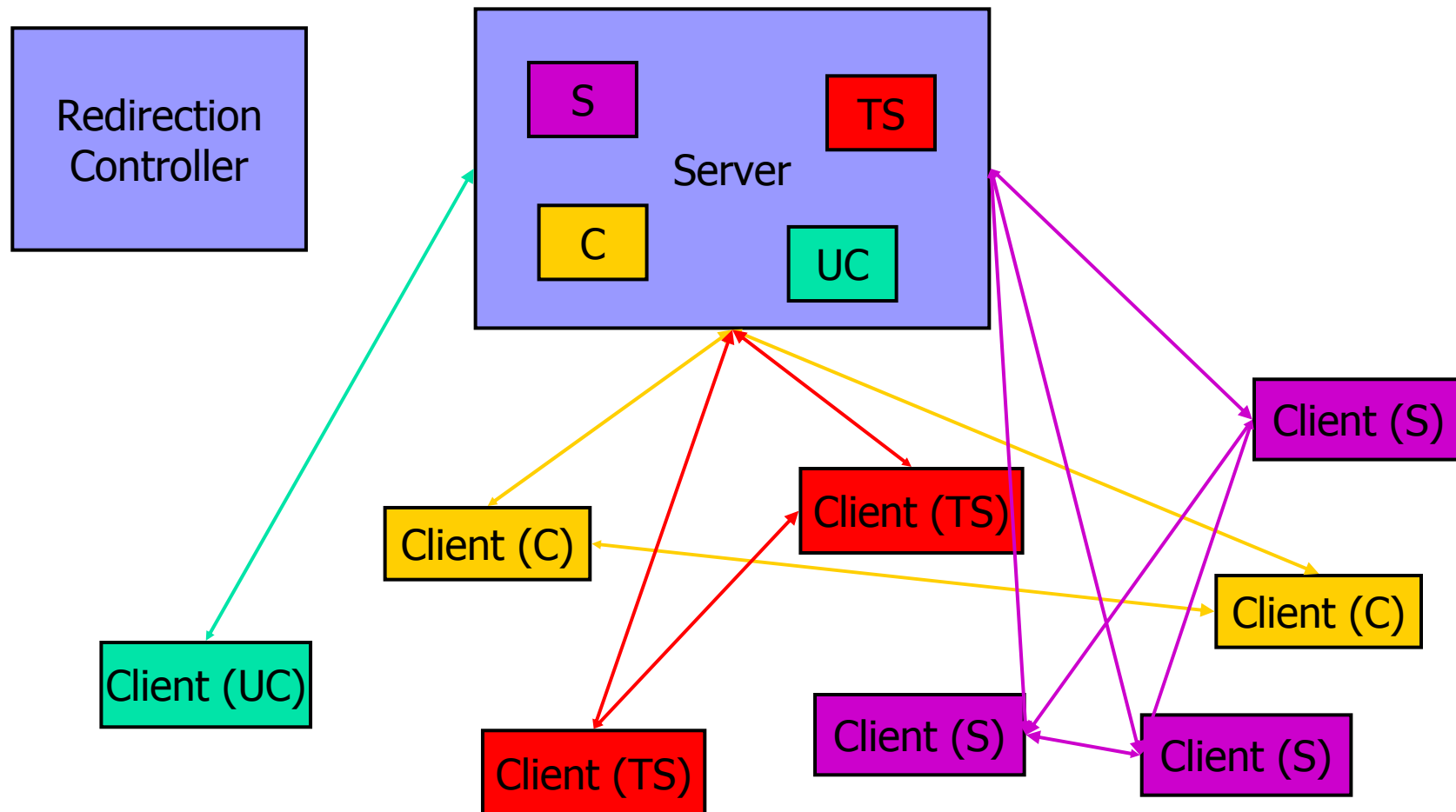


Case 2

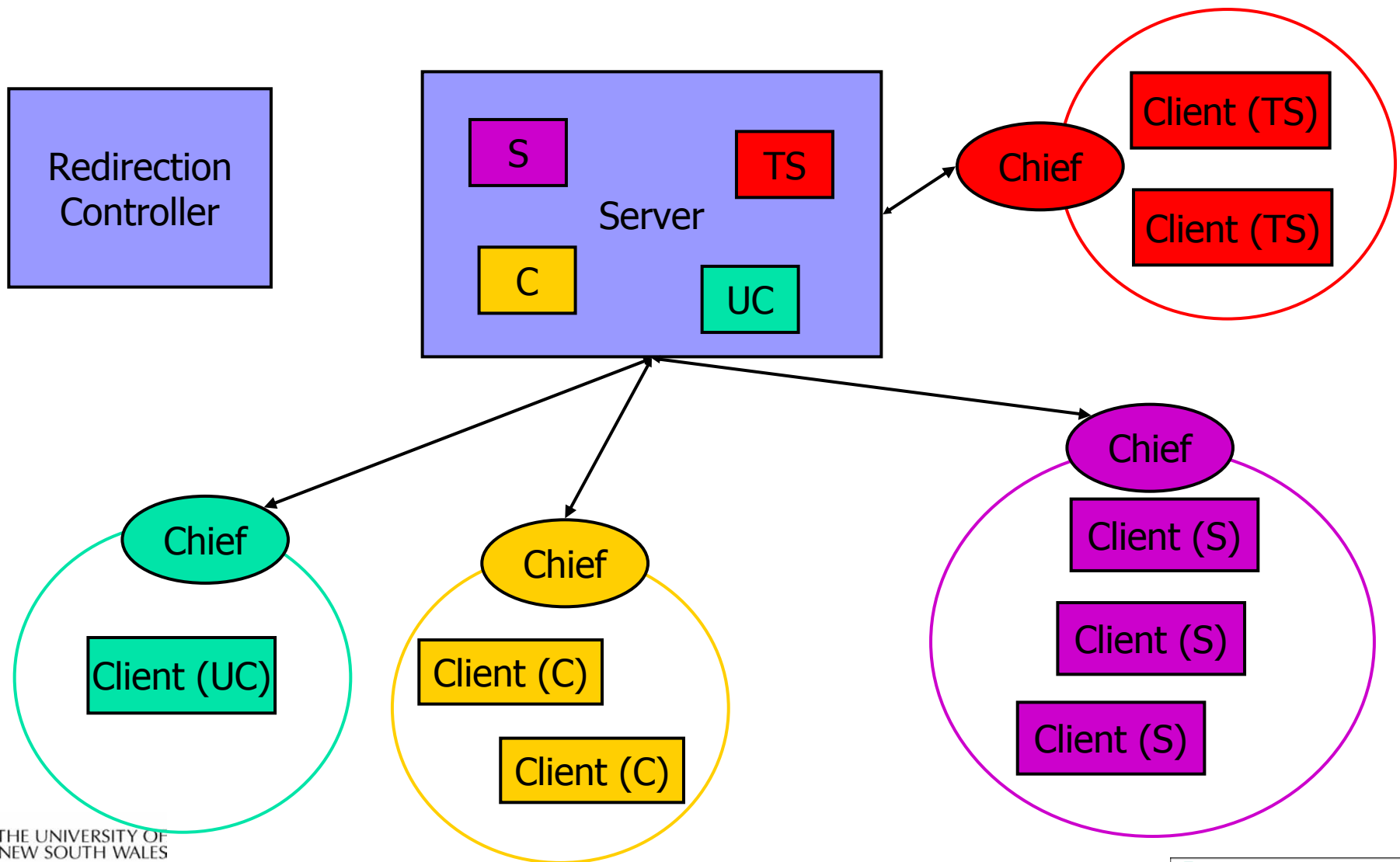
- $I[S] \Rightarrow D$ if $R(S, D) = x$, and $I[x] \Rightarrow D$



Secure System using IPC Redirection



IPC Redirection can implement Clans & Chiefs



Disadvantages and Issues

- The check for if impersonation is permitted if defined recursively
 - Could be expensive to validate
- Dynamic insertion of transparent intermediaries is easy, removal is hard.
 - There might be "state" along a path of intermediaries, redirection controller cannot know unless it has detailed knowledge and/or coordination with intermediaries.
- Cannot determine IPC path of an impersonated message as path may not exist after message arrives
- Centralized redirection controller

Summary

- In microkernel based systems information flow is via communication
 - Communication control is necessary to enforce security policy.
- Any mechanism for communication control must be flexible enough to implement arbitrary security policies.
- We examined two “policy-free” mechanisms to provide communication control
 - Clans & Chiefs
 - Redirection
 - Neither is perfect
- Current research: Virtual Threads

