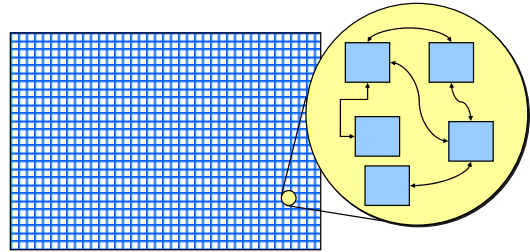


Microkernel Construction

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

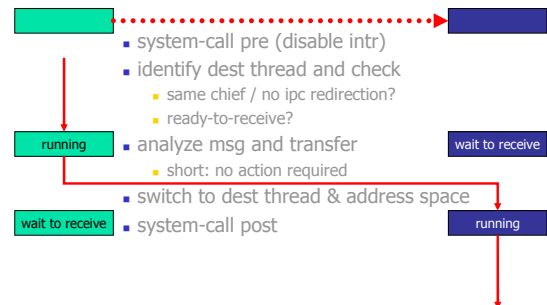
Short IPC

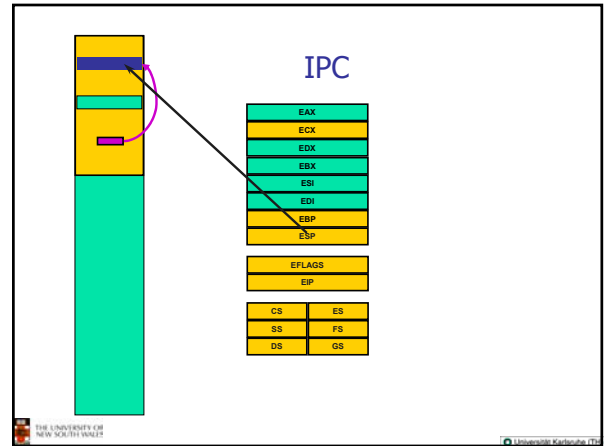
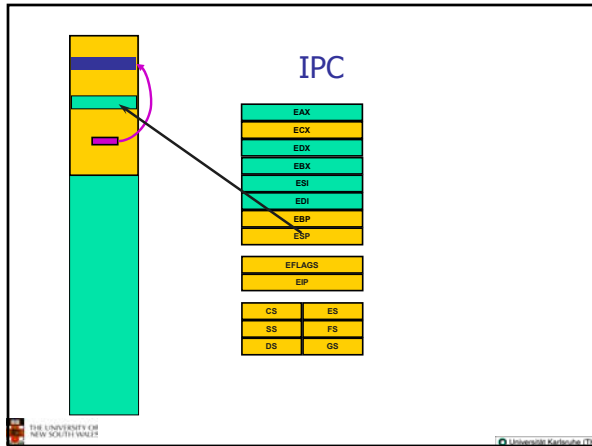
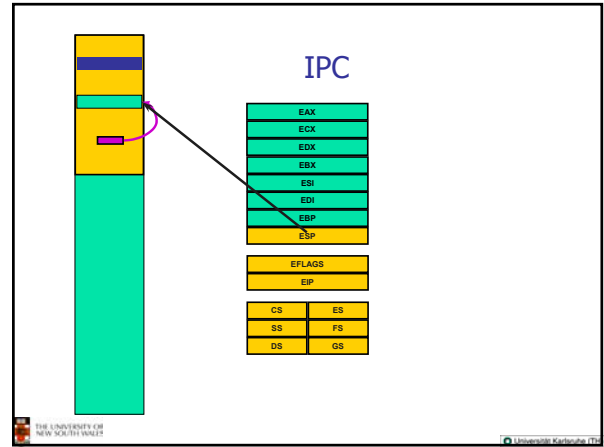
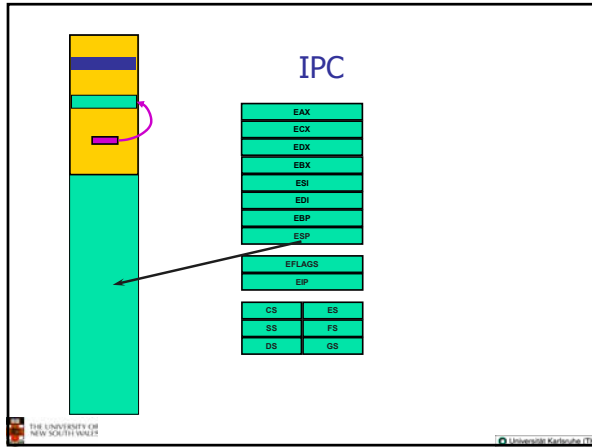
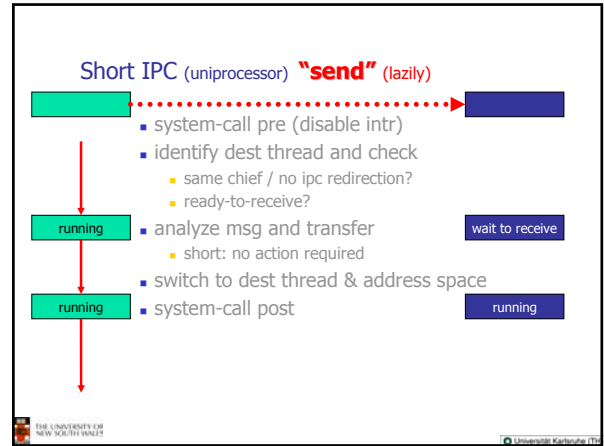
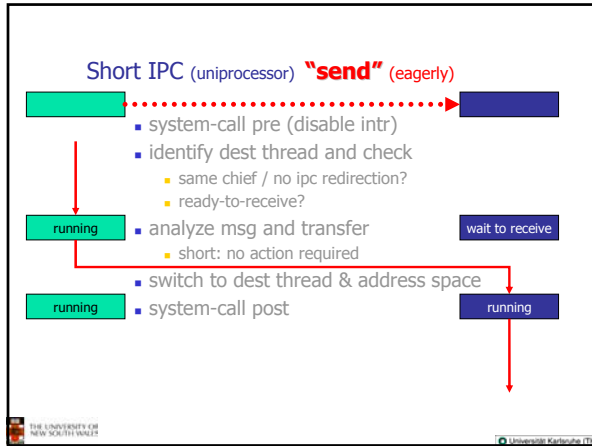
Short IPC (uniprocessor)

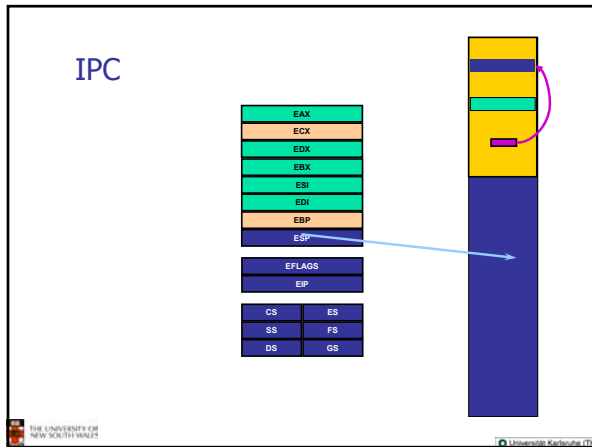
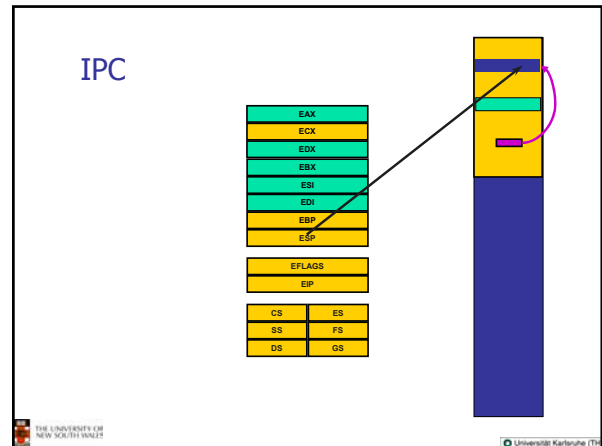
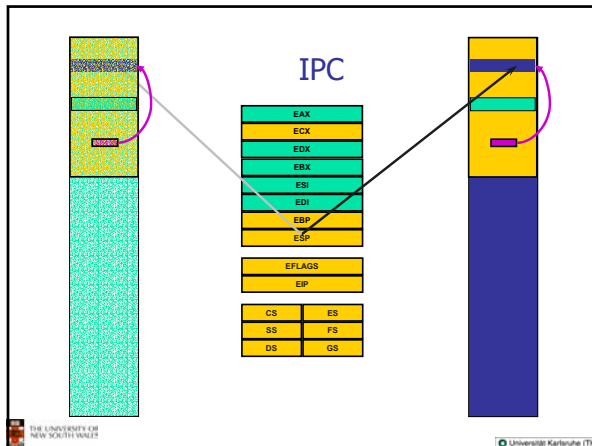
- system-call preamble (disable intr)
- identify dest thread and check
 - same chief / no ipc redirection?
 - ready-to-receive?
- analyze msg and transfer
 - short: no action required
- switch to dest thread & address space
- system-call postamble

The critical path

Short IPC (uniprocessor) "call"







- ### 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!

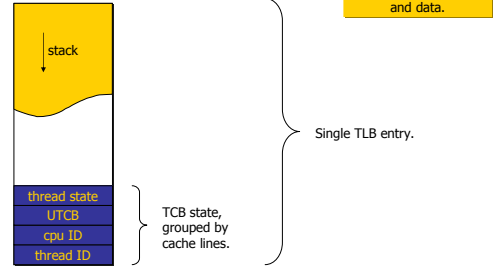
- ### IPC Attributes for Fast Path
- untyped message
 - single runnable thread after IPC
 - must be valid IPC call
 - switch threads, originator blocks
 - send phase:
 - the target is waiting
 - receive phase:
 - the sender is not ready to couple, causing us to block
 - no receive timeout

Avoid Memory References!!!

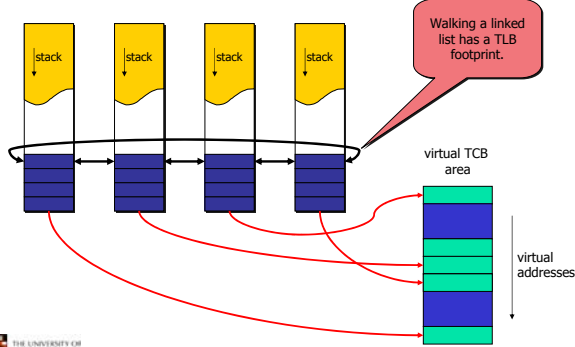
- Memory references are slow
 - avoid in IPC:
 - ex: use lazy scheduling
 - avoid in common case:
 - ex: timeouts
- Microkernel should minimize indirect costs
 - cache pollution
 - TLB pollution
 - memory bus



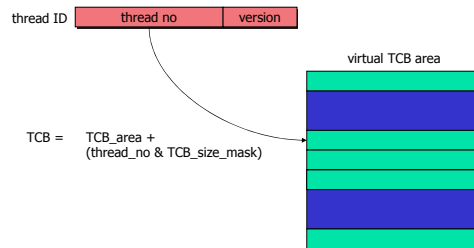
Optimized Memory



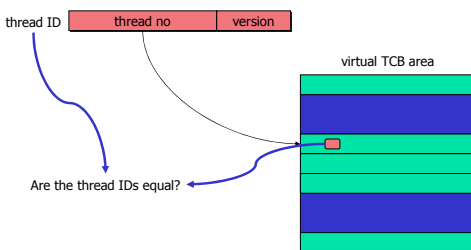
TLB Problem



Avoid Table Lookups



Validate Thread ID



Branch Elimination

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

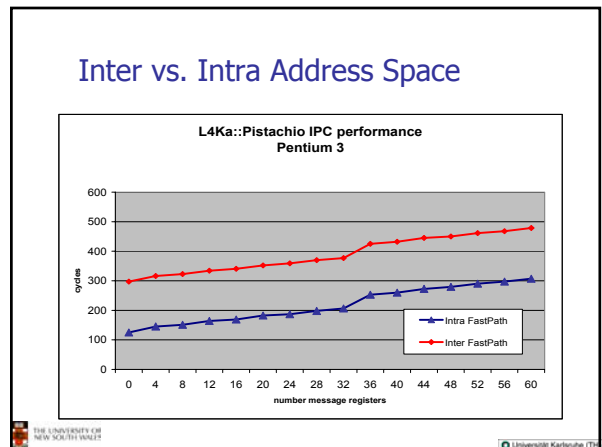
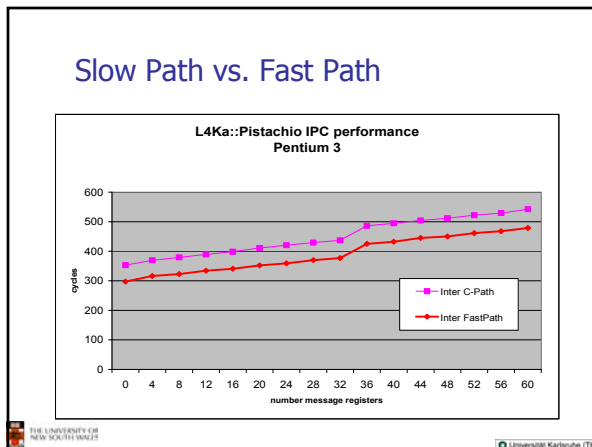
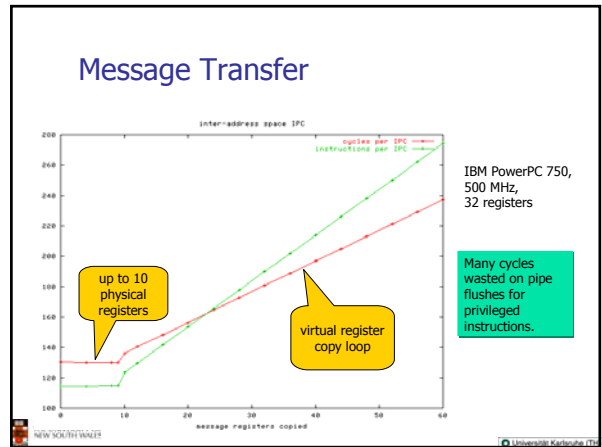
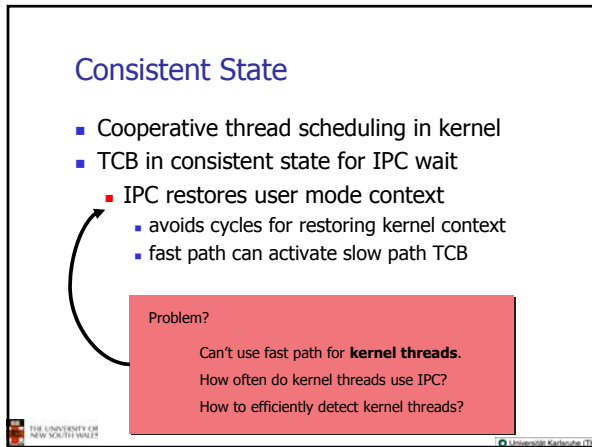
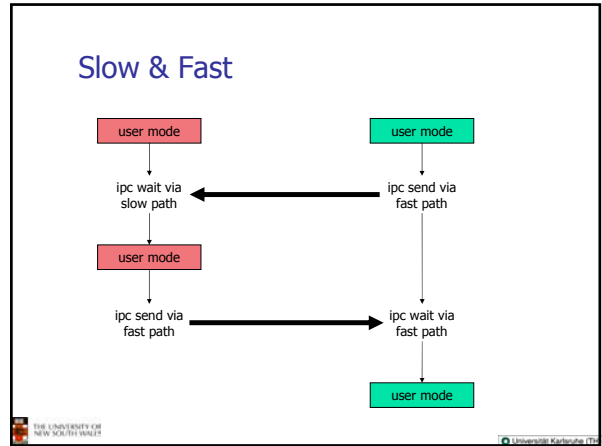
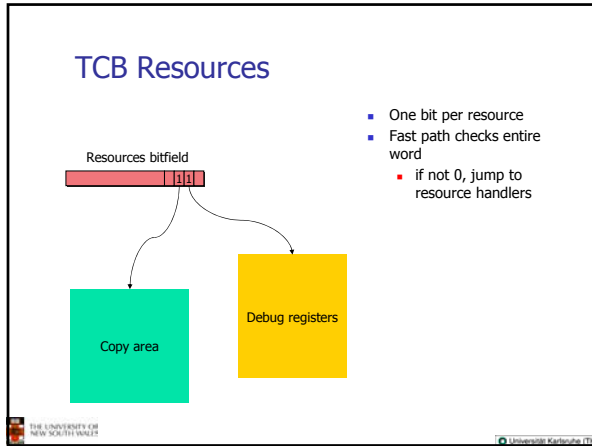
if( slow )
    enter_slow_path();
```

Common case: -1

Common case: 0

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





IPC - Implementation

Long IPC

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

Long IPC (uniprocessor)

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

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

Pagefaults possible!
(in source and dest address space)

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

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

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)

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

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)

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

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

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

IPC - mem copy

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

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

copy in - copy out

- copy into kernel buffer

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

copy in - copy out

- copy into kernel buffer
- switch spaces

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

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)

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

temporary mapping

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

temporary mapping

- select dest area (4+4 M)

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

temporary mapping

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

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

temporary mapping

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

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

temporary mapping

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

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

temporary mapping

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

temporary mapping

- problems
 - multiple threads per AS
 - mappings might change while message is copied
- How long to keep PTE?
- What about TLB?

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

temporary mapping

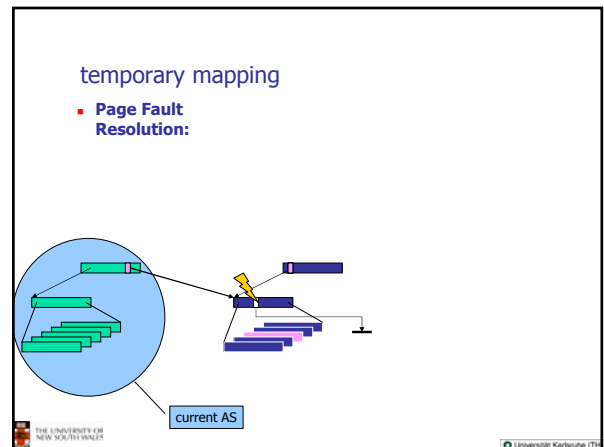
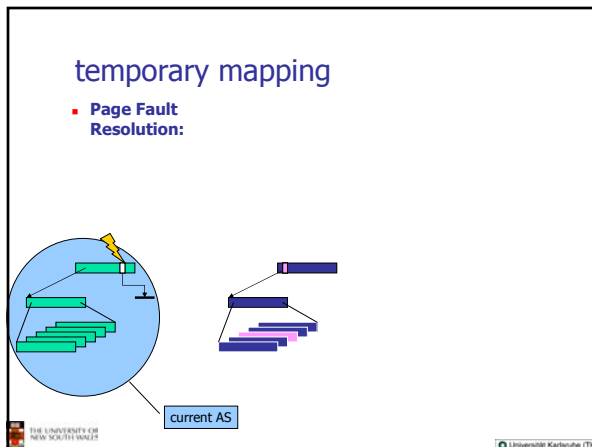
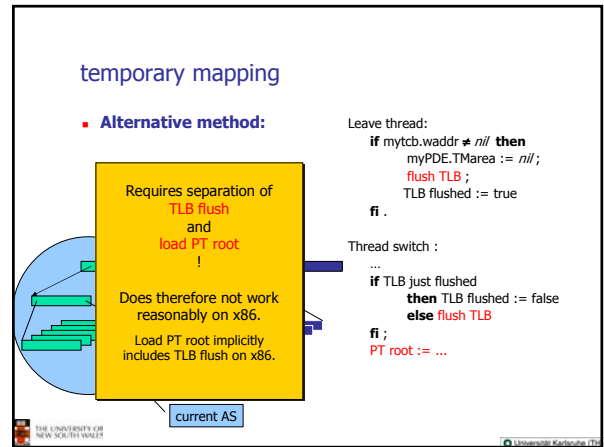
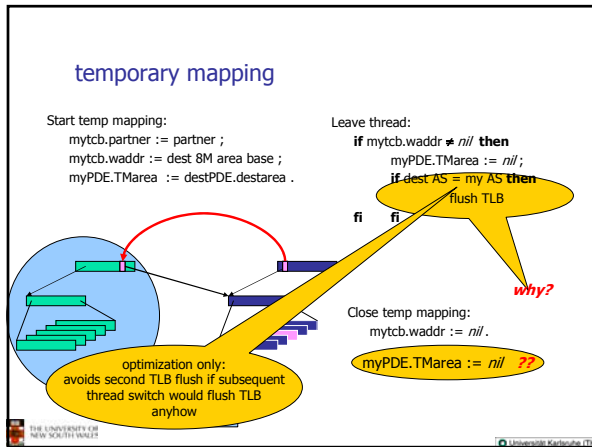
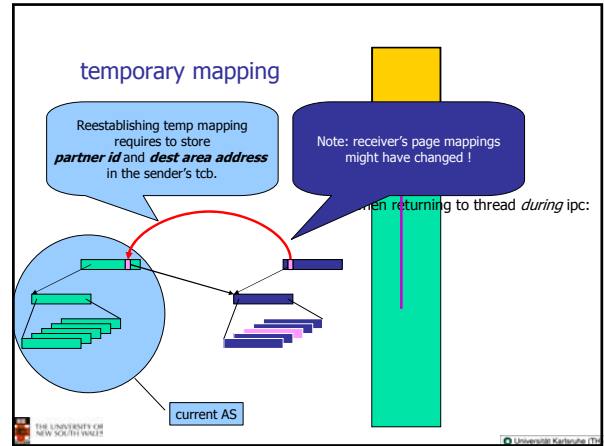
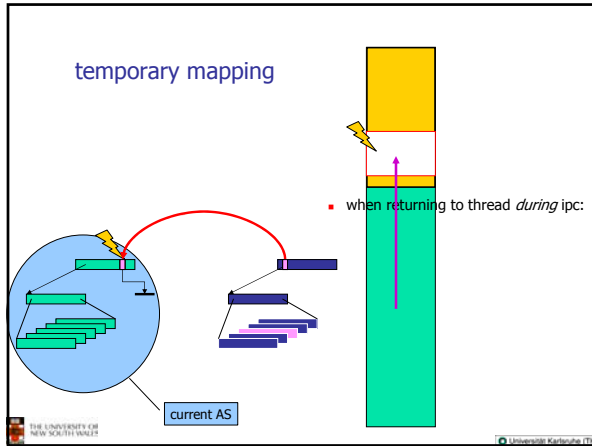
- invalidate PTE
- flush TLB
- when leaving curr thread *during* ipc?

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

temporary mapping

- invalidate PTE
- flush TLB
- when leaving curr thread *during* ipc:

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)



temporary mapping

- Page Fault Resolution:

current AS

THE UNIVERSITY OF NEW SOUTH WALES

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 .
    
```

current AS

THE UNIVERSITY OF NEW SOUTH WALES

temporary mapping

- SMP

THE UNIVERSITY OF NEW SOUTH WALES

temporary mapping

- SMP
 - TM area per processor

THE UNIVERSITY OF NEW SOUTH WALES

temporary mapping

- SMP
 - TM area per processor
 - Page table per processor

P1 AS

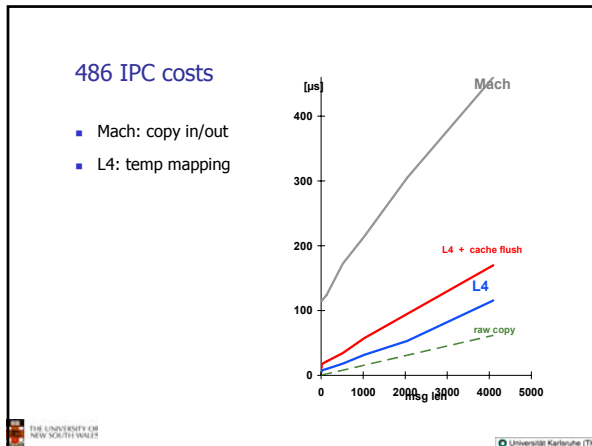
P2 AS

THE UNIVERSITY OF NEW SOUTH WALES

Cost estimates

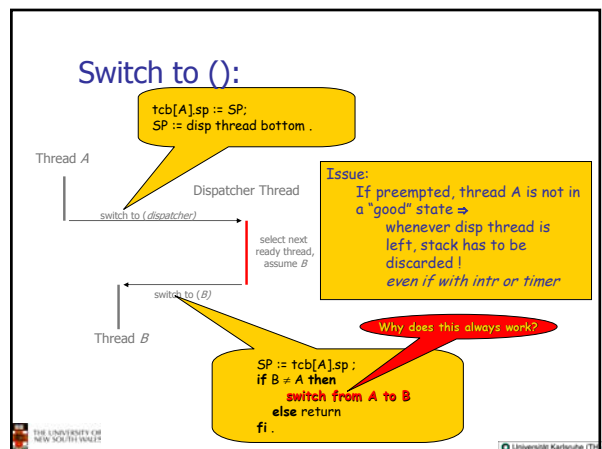
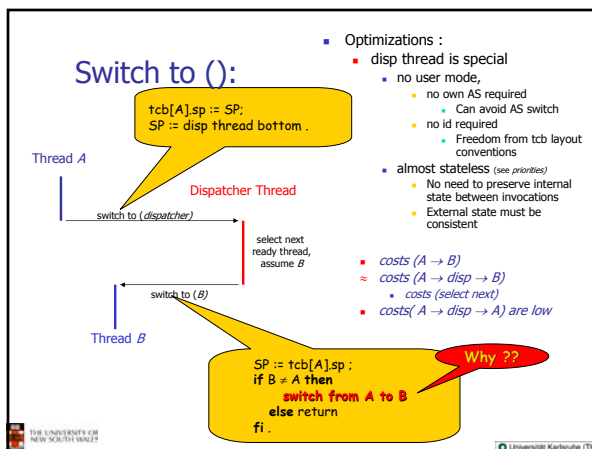
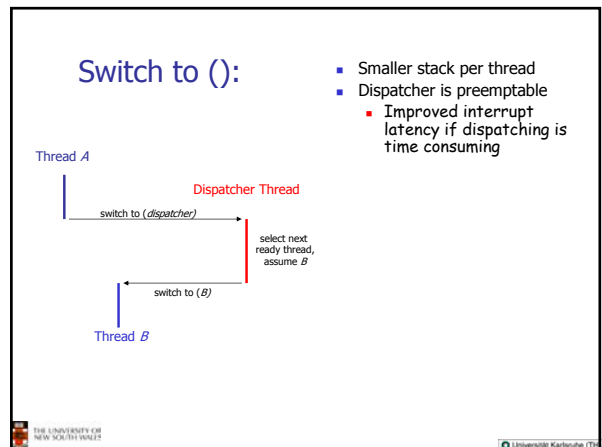
| | Copy in - copy out | Temporary mapping |
|--------------------------------------|--------------------|-----------------------------|
| <i>R/W operations</i> | $2 \times 2n$ | $2n$ |
| <i>Cache lines</i> | $3 \times n/8$ | $2 \times n/8$ |
| <i>Small n overhead cache misses</i> | $n/8$ | 0 |
| <i>Large 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 |

THE UNIVERSITY OF NEW SOUTH WALES



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



Example: Simple Dispatch

Example: Simple Dispatch



Example: Dispatch with 'Tick'

Example: Dispatch with 'Tick'

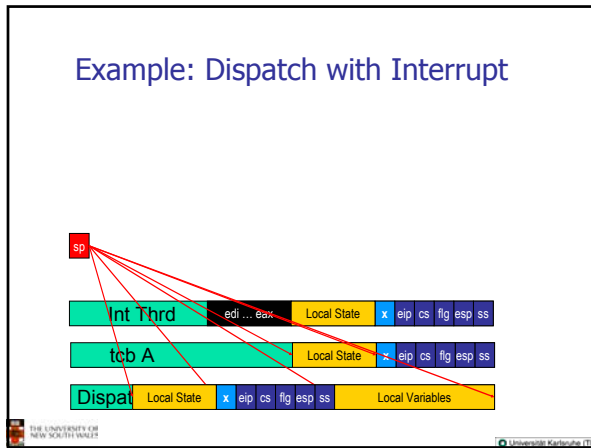


Example: Dispatch with 'Tick'

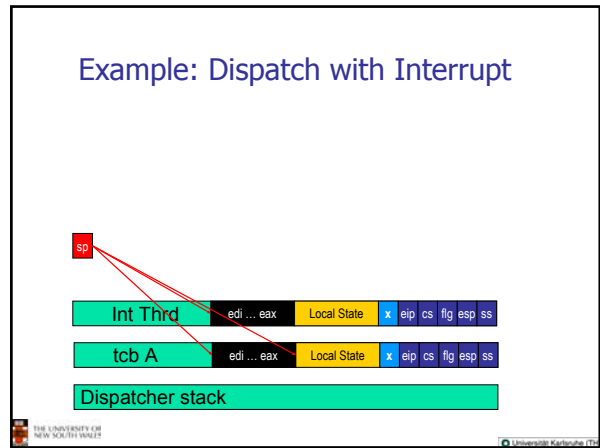
Example: Dispatch with Interrupt



Example: Dispatch with Interrupt

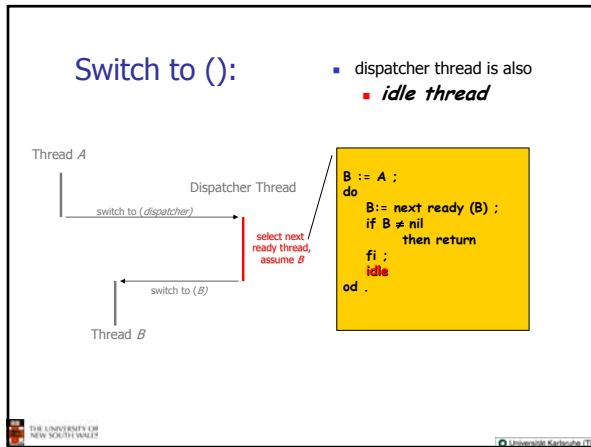


Example: Dispatch with Interrupt



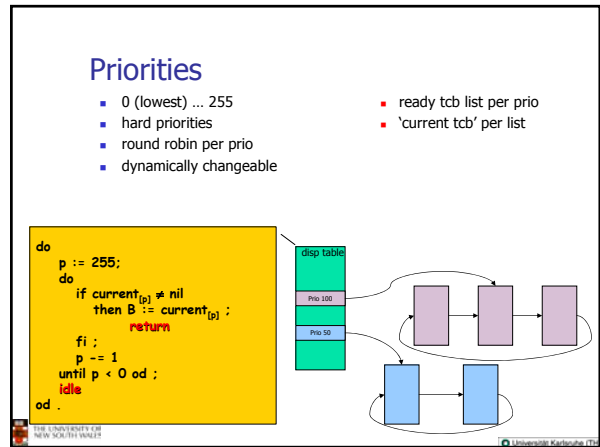
Switch to ():

- dispatcher thread is also *idle thread*



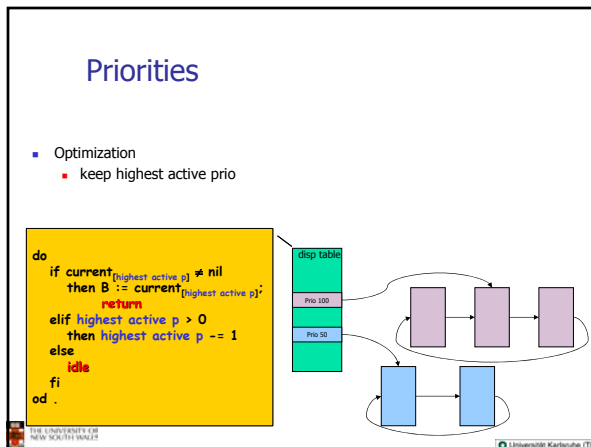
Priorities

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



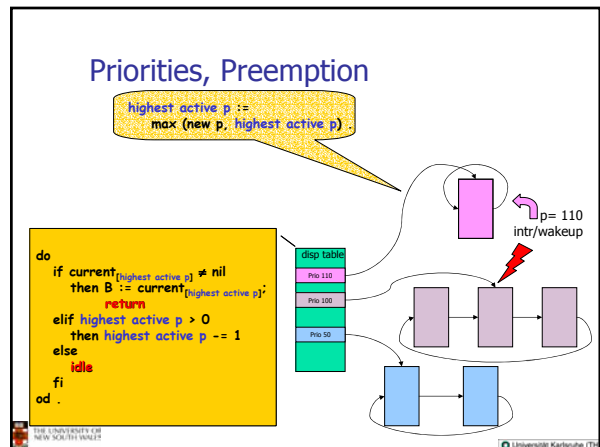
Priorities

- Optimization
- keep highest active prio



Priorities, Preemption

highest active p := max (new p, highest active p)



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 cur(prio set p).rem ts = 0
  then cur(prio set p) := next ;
  current(prio set 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 cur(prio set p).rem ts = 0
  then cur(prio set p) := next ;
  current(prio set 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 cur(prio set p).rem ts = 0
  then cur(prio set p) := next ;
  current(prio set 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 cur(prio set p).rem ts = 0
  then current(prio set p).rem ts := new ts ;
  current(prio set 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

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

Lazy Dispatching

Thread state toggles frequently (per ipc)

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

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

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

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

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

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

Timeouts & Wakeups

Operations:

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

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

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

Timeouts & Wakeups

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

too expensive

THE UNIVERSITY OF NEW SOUTH WALES
Universität Karlsruhe (TH)

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

THE UNIVERSITY OF NEW SOUTH WALES | Universität Karlsruhe (TH)

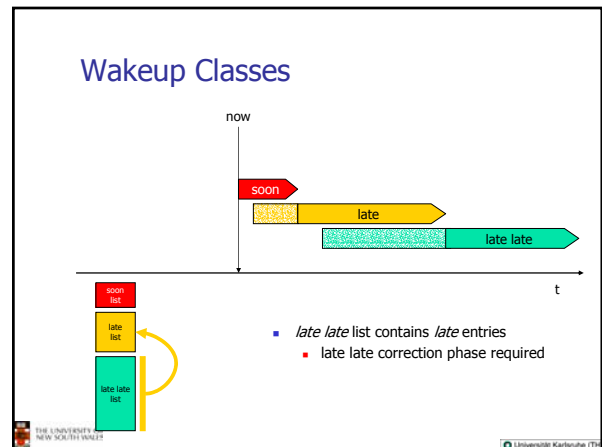
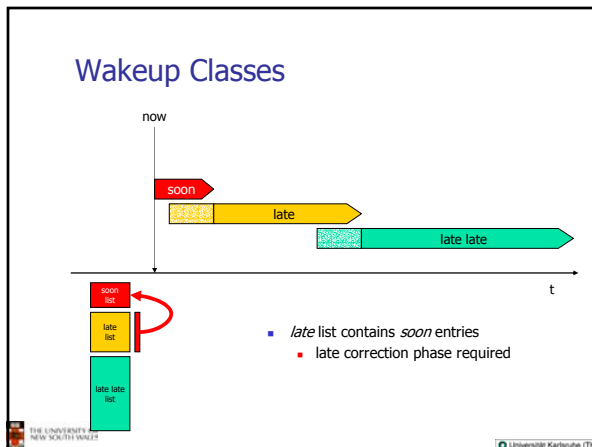
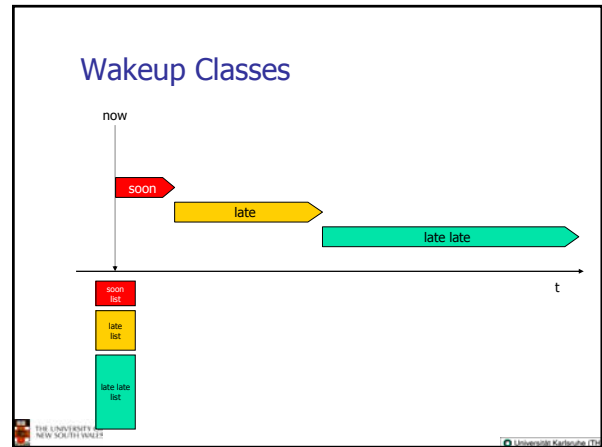
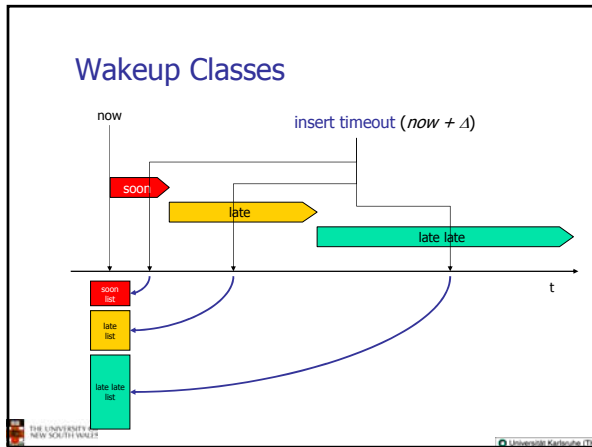
Timeouts & Wakeups

too expensive
too complicated

Idea 3: *sorted tree*

- *insert timeout costs:*
 - search + insert entry $\log n \times 20..100 + 20..100$ 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

THE UNIVERSITY OF NEW SOUTH WALES | Universität Karlsruhe (TH)



Wakeup Classes

now

τ_{soon}

soon

late

late late

t

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 & Wakeup

Idea 4: *unsorted wakeup classes*

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

t

soon list

late list

late late list

Lazy Timeouts

insert (t_1)

delete timeout

t

soon list

late list

late late list

Lazy Timeouts

insert (t_1)

delete timeout

insert (t_2)

t

soon list

late list

late late list

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?



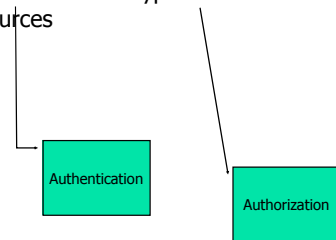
Security defined by policy

- 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)

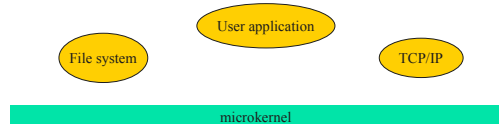


Security policy

- 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???

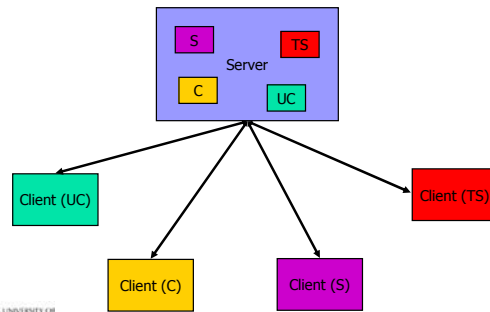


Example Policy: Mandatory Access Control

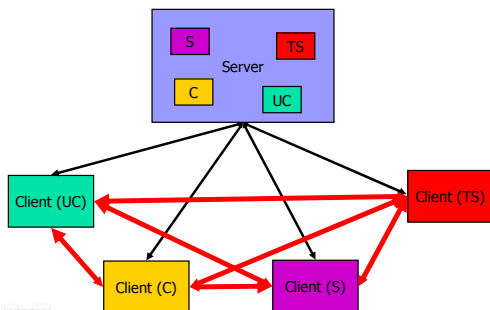
- Objects assigned security levels
 - Top Secret, Secret, Classified, Unclassified
 - $TS > S > C > UC$
- Subjects (users) assigned security levels
 - Top Secret, Secret, Classified, Unclassified
- A subject (S) can read an object (O) iff
 - $level(S) \geq level(O)$
- A subject (S) can write an object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

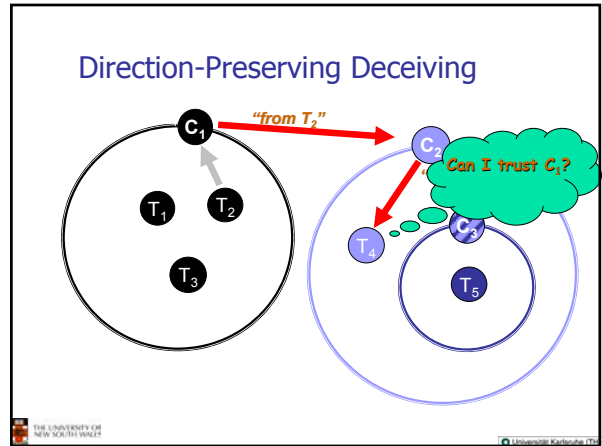
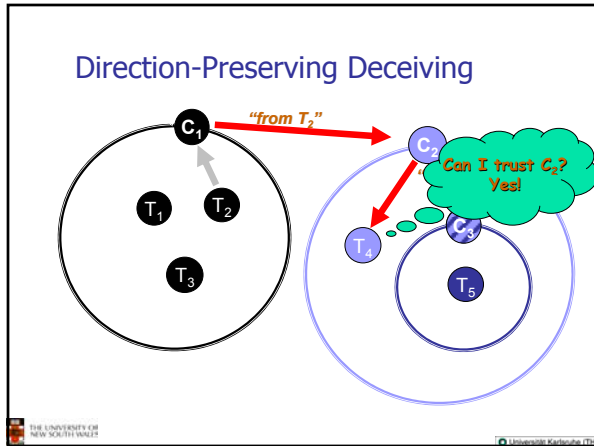
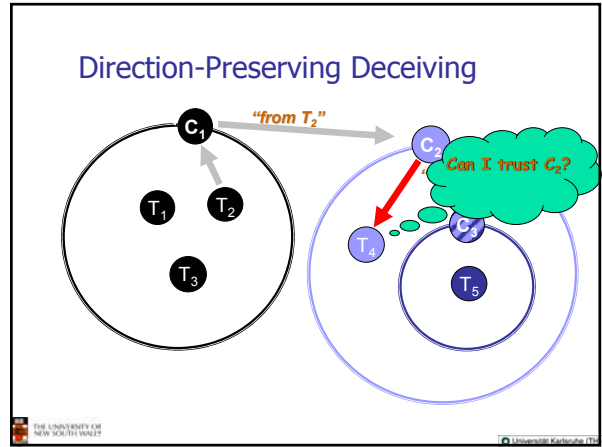
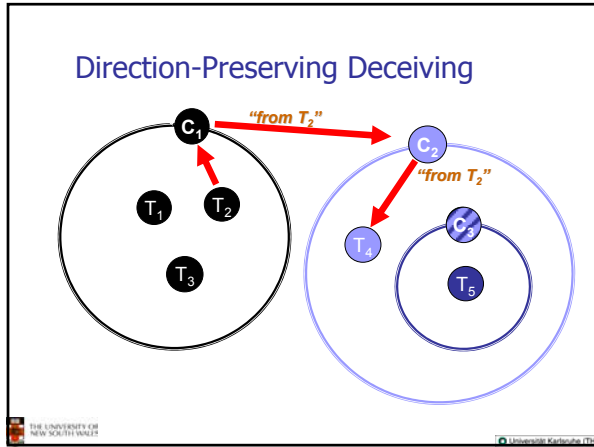
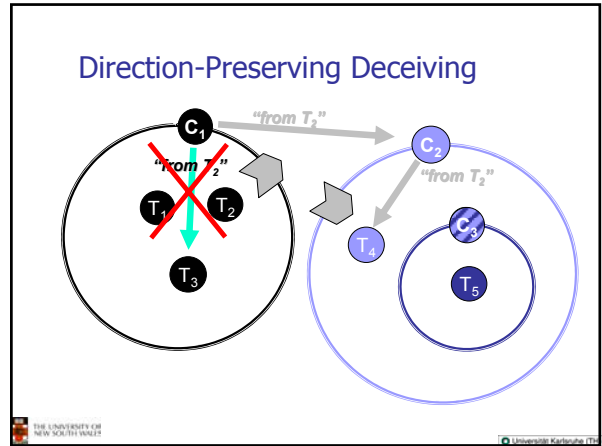
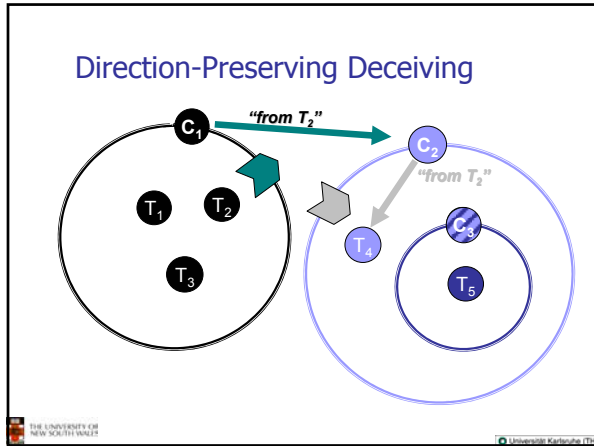
Intra-Clan IPC

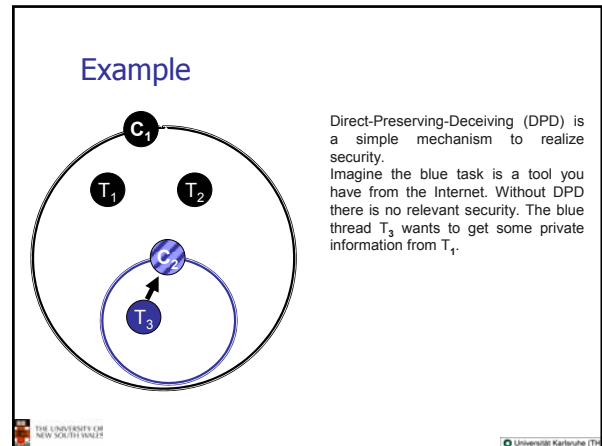
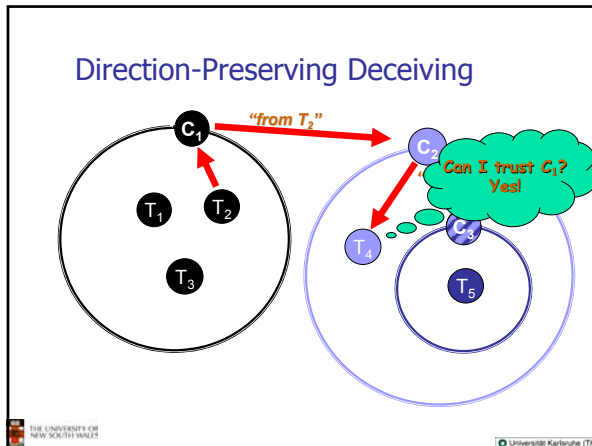
- Direct IPC by microkernel

Inter-Clan IPC

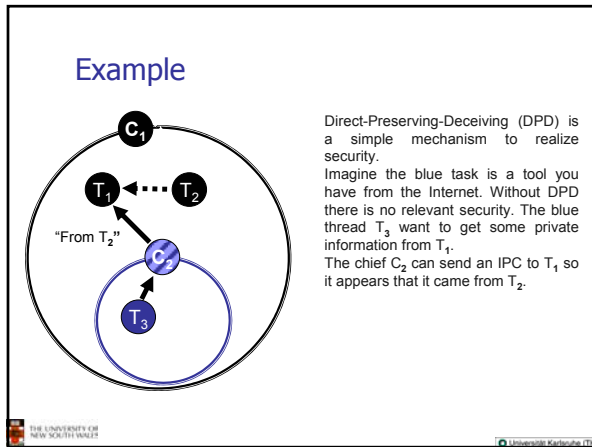
- Microkernel redirects IPC to next chief
- Chief (user task) *can* forward IPC or modify or ...

Direction-Preserving Deceiving

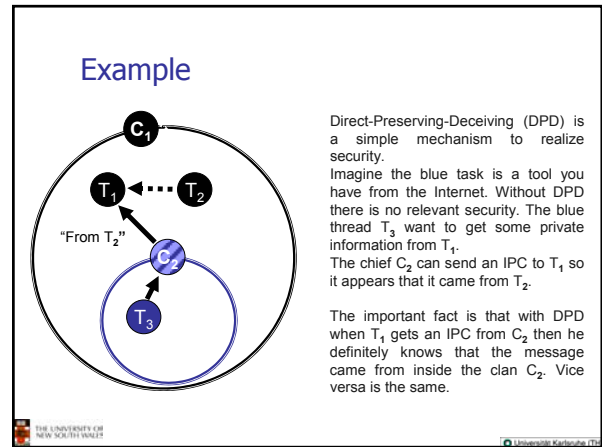




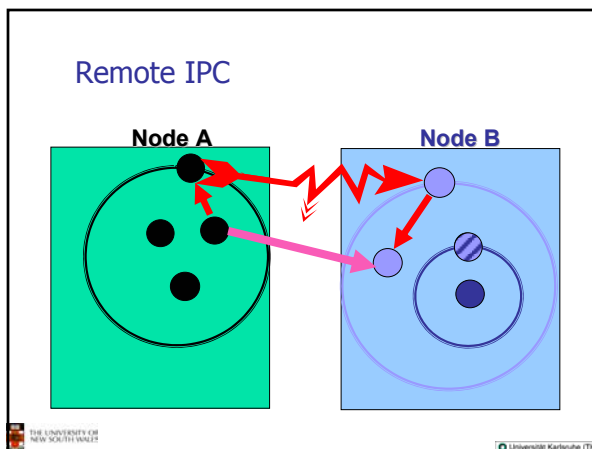
Direct-Preserving-Deceiving (DPD) is a simple mechanism to realize security. Imagine the blue task is a tool you have from the Internet. Without DPD there is no relevant security. The blue thread T₃ wants to get some private information from T₄.



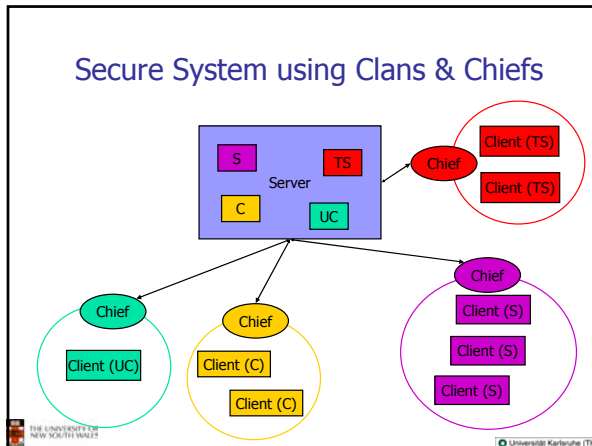
Direct-Preserving-Deceiving (DPD) is a simple mechanism to realize security. Imagine the blue task is a tool you have from the Internet. Without DPD there is no relevant security. The blue thread T₃ want to get some private information from T₁. The chief C₂ can send an IPC to T₁ so it appears that it came from T₂.



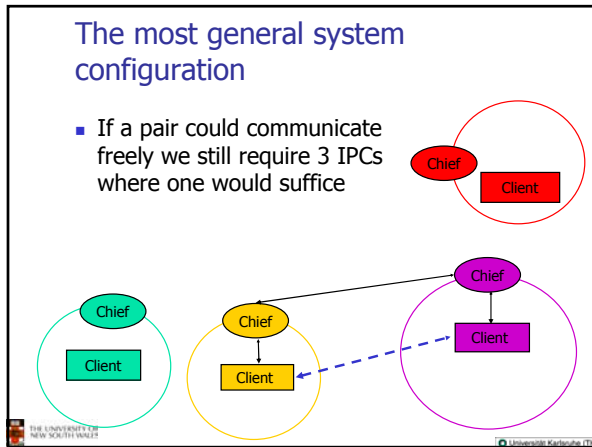
Direct-Preserving-Deceiving (DPD) is a simple mechanism to realize security. Imagine the blue task is a tool you have from the Internet. Without DPD there is no relevant security. The blue thread T₃ want to get some private information from T₁. The chief C₂ can send an IPC to T₁ so it appears that it came from T₂. The important fact is that with DPD when T₁ gets an IPC from C₂ then he definitely knows that the message came from inside the clan C₂. Vice versa is the same.



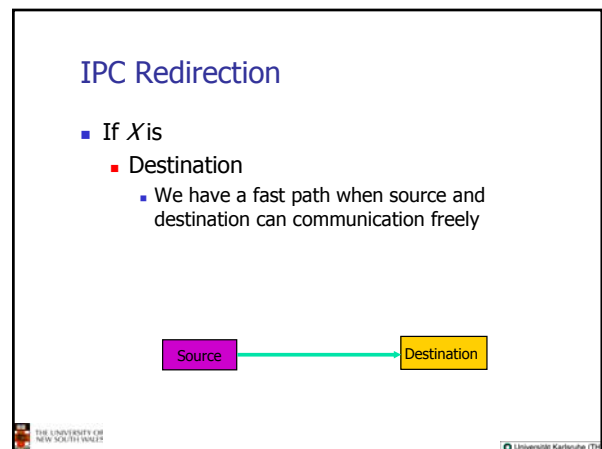
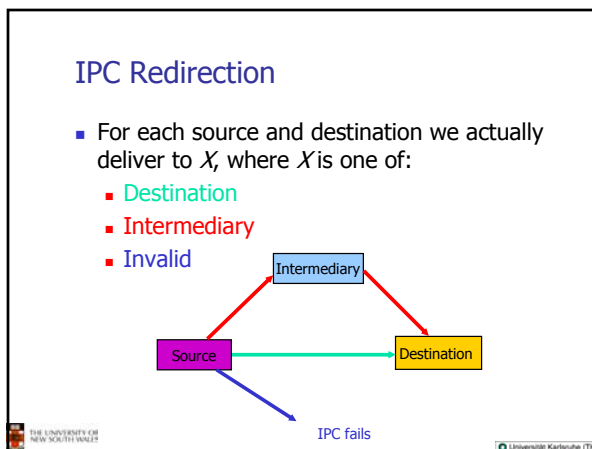
- ### Clans & Chiefs
- Remote IPC
 - Multi-level security
 - Debugging
 - Heterogeneity



- ### 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



IPC Redirection



IPC Redirection

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

Source → IPC fails → Destination

IPC Redirection

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

Source → Intermediary → Destination

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 $R(S,D) = I$

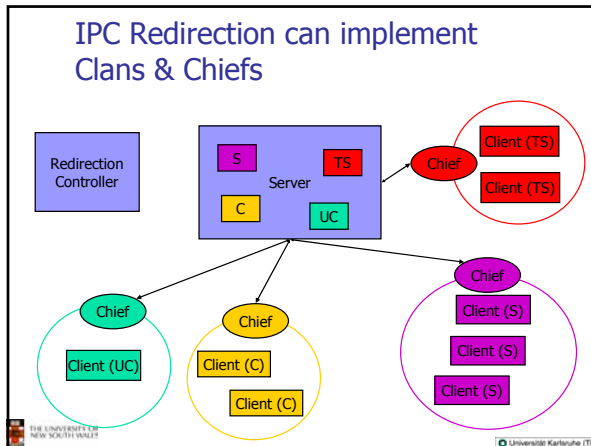
Source → Intermediary → Destination (From S)

Case 2

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

Source → X → Intermediary → Destination (From S)

Secure System using IPC Redirection



- ### 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