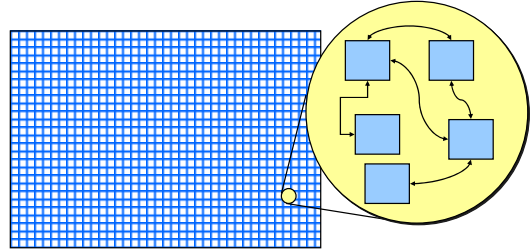# Microkernel Construction

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
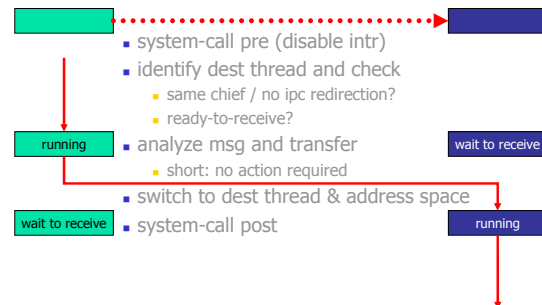- Wait for IPC
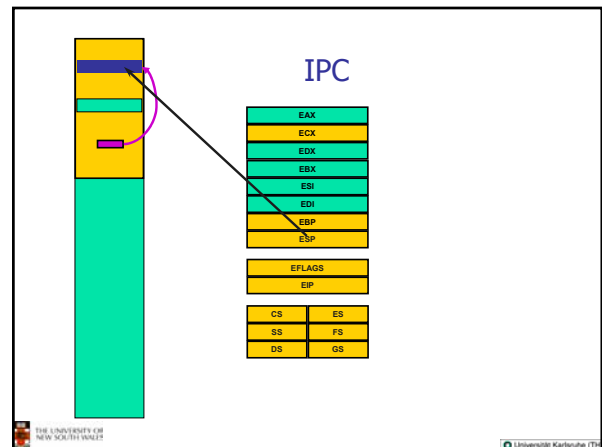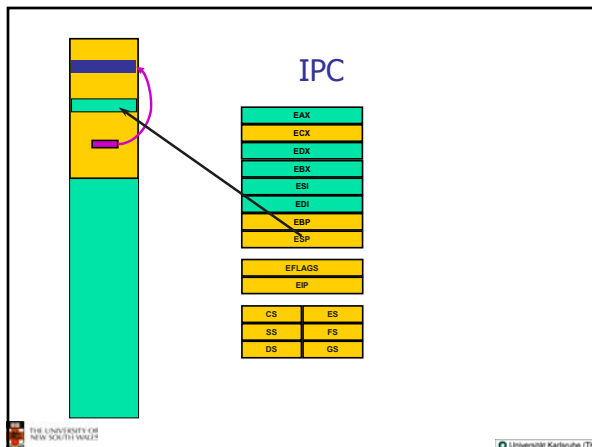

IPC - Implementation

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
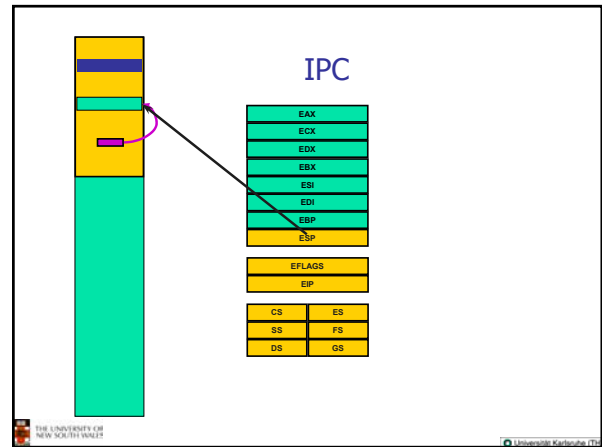 - short: no action required
- switch to dest thread & address space
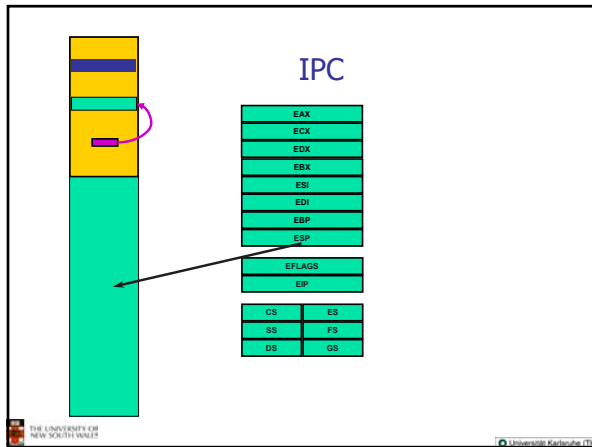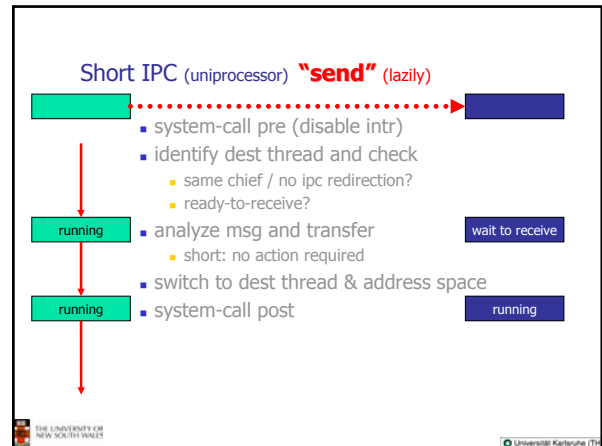- system-call postamble

The critical path

Short IPC (uniprocessor) **"call"**

- system-call pre (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 post

running

wait to receive

wait to receive

running

Short IPC (uniprocessor) **"send"** (eagerly)
- system-call pre (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
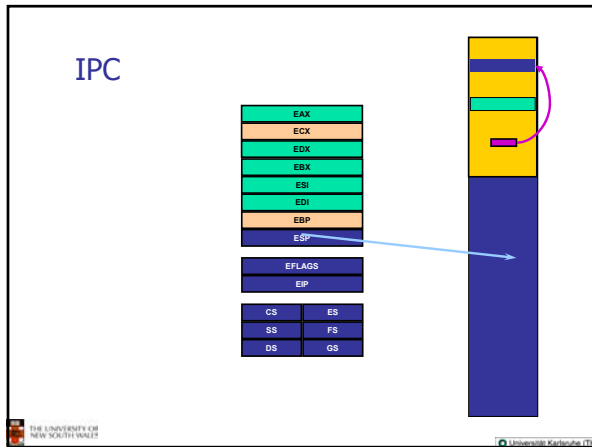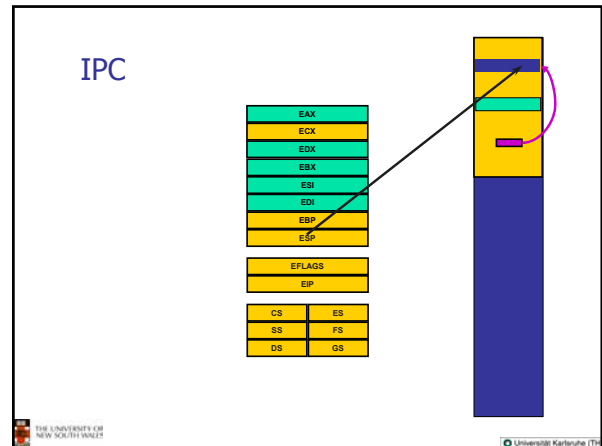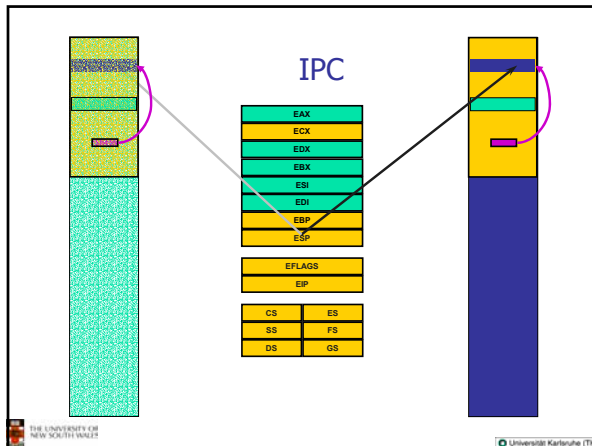- system-call post

Short IPC (uniprocessor) **"send"** (lazily)
- system-call pre (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 post

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
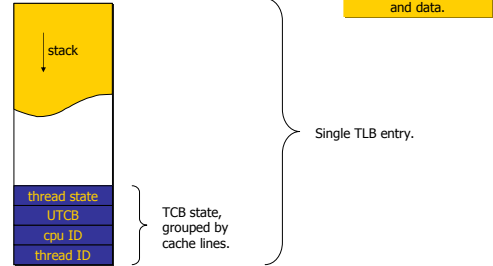 - incompatible code optimizations

- 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
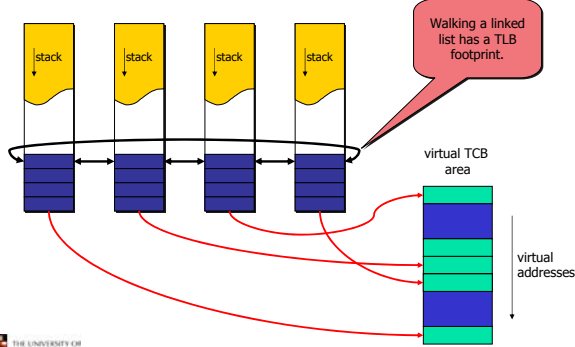
Avoid Memory References!!!

- Memory references are slow
  - avoid in IPC:
    - ex: use lazy scheduling
  - avoid in common case:
    - ex: timeouts

- Microkernel should minimize indirect costs
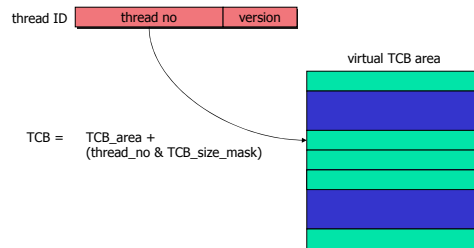 - cache pollution
  - TLB pollution
  - memory bus

Optimized Memory

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

stack

Single TLB entry.

thread state
UTCB
cpu ID
thread ID

TCB state, grouped by cache lines.

TLB Problem

stack stack stack stack

Walking a linked list has a TLB footprint.

virtual TCB area

virtual addresses

Avoid Table Lookups

thread ID | thread no | version

virtual TCB area

TCB = TCB_area + (thread_no & TCB_size_mask)

Validate Thread ID

thread ID | thread no | version

virtual TCB area

Are the thread IDs equal?
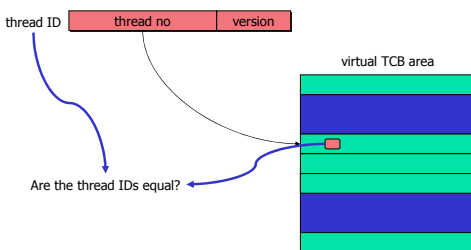
Branch Elimination

Common case: -1

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

if( slow )
         enter_slow_path()
```
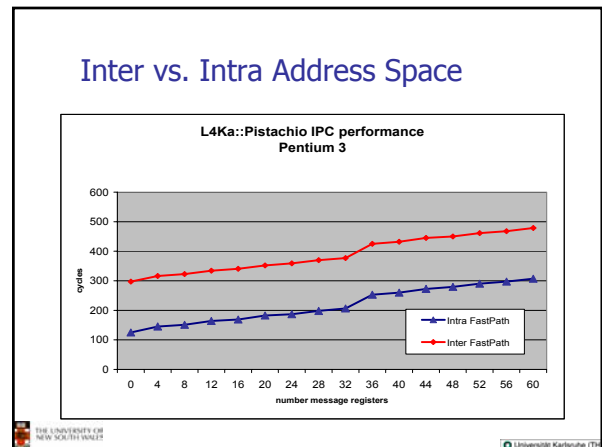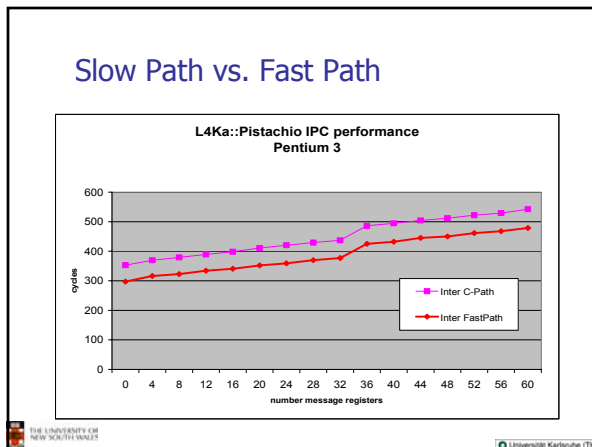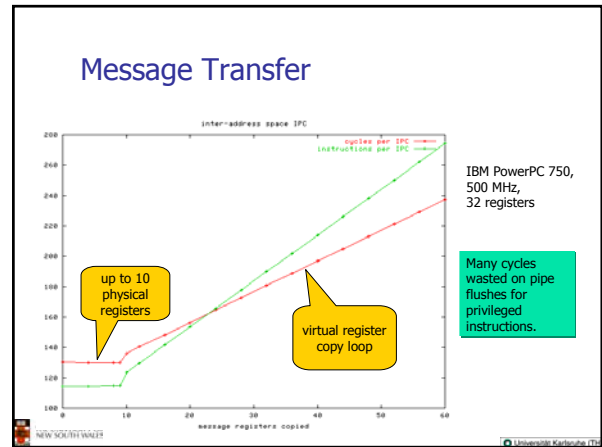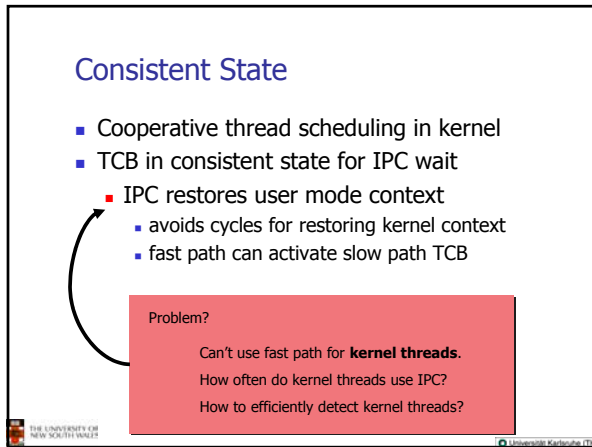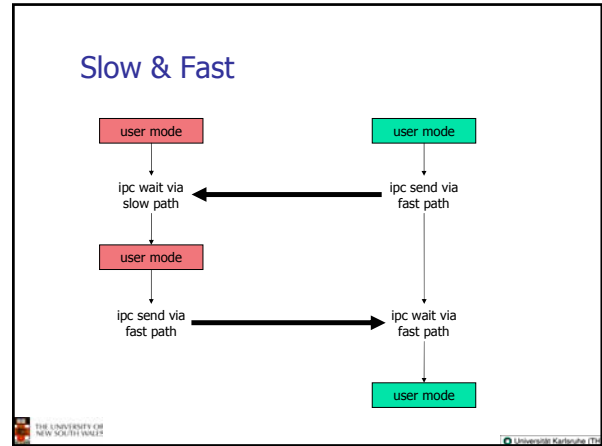
Common case: 0

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

TCB Resources

Resources bitfield

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

Copy area

Debug registers

Slow & Fast

user mode

ipc wait via slow path

user mode

ipc send via fast path

user mode

ipc send via fast path

ipc wait via fast path

user mode

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

inter-address space IPC

cycles per IPC
instructions per IPC

IBM PowerPC 750, 500 MHz, 32 registers

up to 10 physical registers

virtual register copy loop

Many cycles wasted on pipe flushes for privileged instructions.

Slow Path vs. Fast Path

**L4Ka::Pistachio IPC performance
Pentium 3**

Inter C-Path
Inter FastPath

Inter vs. Intra Address Space

**L4Ka::Pistachio IPC performance
Pentium 3**

Intra FastPath
Inter FastPath

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:

    - *– transfer message –*

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

> 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
    - *– 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)

running

locked running

wait to receive

wait

locked wait

running

- 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

IPC - mem copy

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

Slide 1:

copy in - copy out

- copy into kernel buffer

Slide 2:

copy in - copy out

- copy into kernel buffer
- switch spaces

Slide 3:

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)

Slide 4:

temporary mapping

Slide 5:

temporary mapping

- select dest area (4+4 M)

Slide 6:

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?

current AS

temporary mapping

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

current AS

temporary mapping

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

current AS

temporary mapping

- when returning to thread *during* ipc:

current AS

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:

current AS

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**

why?

Close temp mapping:
  mytcb.waddr := *nil* .

  myPDE.TMarea := *nil*   **??**

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

temporary mapping

- **Alternative method:**

Requires separation of
TLB flush
and
load PT root
!

Does therefore not work reasonably on x86.

Load PT root implicitly includes TLB flush on x86.

current AS

Leave thread:
  if mytcb.waddr ≠ *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:**

current AS

temporary mapping

- **Page Fault Resolution:**

current AS

temporary mapping

- **Page Fault Resolution:**

current AS

THE UNIVERSITY OF NEW SOUTH WALES

Universität Karlsruhe (TH)

---

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

Universität Karlsruhe (TH)

---

temporary mapping

- SMP

P1

THE UNIVERSITY OF NEW SOUTH WALES

Universität Karlsruhe (TH)

---

temporary mapping

- SMP
  - TM area per processor

P2

P1

THE UNIVERSITY OF NEW SOUTH WALES

Universität Karlsruhe (TH)

---

temporary mapping

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

P2

P1 AS

P2 AS

THE UNIVERSITY OF NEW SOUTH WALES

Universität Karlsruhe (TH)

Cost estimates

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

THE UNIVERSITY OF NEW SOUTH WALES

Universität Karlsruhe (TH)

486 IPC costs

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

[μs]

Mach

400

300

200

L4 + cache flush

L4

100

raw_copy

0

0   1000   2000   3000   4000   5000

msg len

THE UNIVERSITY OF
NEW SOUTH WALES

Universität Karlsruhe (TH

Dispatching

THE UNIVERSITY OF
NEW SOUTH WALES

Universität Karlsruhe (TH
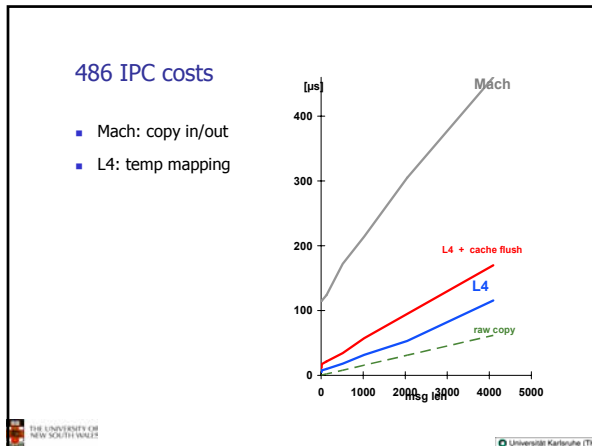
Dispatching topics:

- 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

THE UNIVERSITY OF
NEW SOUTH WALES

Universität Karlsruhe (TH

Switch to ():

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

Thread A

Dispatcher Thread

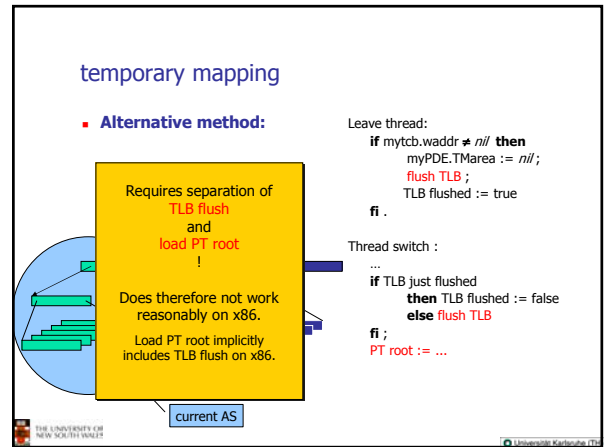switch to (*dispatcher*)

select next
ready thread,
assume B

switch to (B)

Thread B

THE UNIVERSITY OF
NEW SOUTH WALES

Universität Karlsruhe (TH

Switch to ():

- 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

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

Thread A

Dispatcher Thread

switch to (*dispatcher*)

select next
ready thread,
assume B

switch to (B)

Thread B

- costs (A → B)
- ≈ costs (A → disp → B)
  - costs (select next)
- costs(A → disp → A) are low

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

Why ??

THE UNIVERSITY OF
NEW SOUTH WALES

Universität Karlsruhe (TH

Switch to ():

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

Thread A

Dispatcher Thread

switch to (*dispatcher*)

select next
ready thread,
assume B

switch to (B)

Thread B

Issue:
  If preempted, thread A is not in a "good" state ⇒
    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 ≠ A then
  switch from A to B
else return
fi .

THE UNIVERSITY OF
NEW SOUTH WALES

Universität Karlsruhe (TH

11

Example: Simple Dispatch

Example: Simple Dispatch

tcb B	edi ... eax	Local State	x	eip	cs	flg	esp	ss
tcb A	edi ... eax	Local State	x	eip	cs	flg	esp	ss
Dispatcher stack		Local Variables						
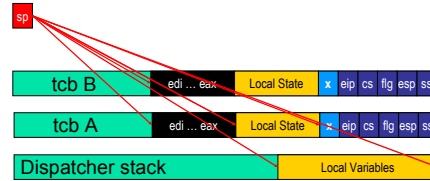
Example: Dispatch with 'Tick'

Example: Dispatch with 'Tick'

tcb B	edi ... eax	Local State	x	eip	cs	flg	esp	ss
tcb A		Local State	x	eip	cs	flg	esp	ss
Dispat	Local State	x	eip	cs	flg	esp	ss	Local Variables

Example: Dispatch with 'Tick'

tcb B	edi ... eax	Local State	x	eip	cs	flg	esp	ss
tcb A	edi ... eax	Local State	x	eip	cs	flg	esp	ss
Dispatcher stack		Local Variables						

Example: Dispatch with Interrupt

12

Example: Dispatch with Interrupt

Int Thrd	...edi ... eax	Local State	x	eip	cs	flg	esp	ss
tcb A		Local State	x	eip	cs	flg	esp	ss
Dispat	Local State	x	eip	cs	flg	esp	ss	Local Variables

Example: Dispatch with Interrupt

Int Thrd	...edi ... eax	Local State	x	eip	cs	flg	esp	ss
tcb A		edi ... eax	Local State	x	eip	cs	flg	esp	ss
Dispatcher stack								

Switch to ():

- dispatcher thread is also
  - *idle thread*


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

Thread A

Dispatcher Thread

switch to (dispatcher)

select next ready thread, assume B

switch to (B)

Thread B

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 .
```
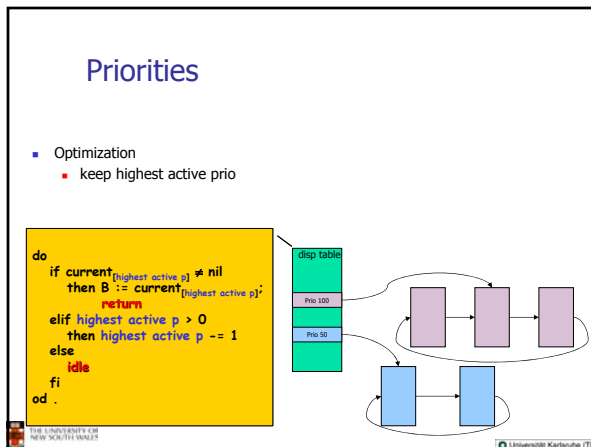
Priorities, Preemption

highest active p :=
    max (new p, 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 .
```

p= 110
intr/wakeup

13

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 .
```

disp table
Prio 110
Prio 100
Prio 50

Remaining
time slice
> 0 ?

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 .
```

disp table
Prio 110
Prio 100
Prio 50

Remaining
time slice
> 0

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 .
```

disp table
Prio 100
Prio 50

Remaining
time slice
= 0

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 .
```

disp table
Prio 100
Prio 50

Remaining
time slice
:= new ts

Lazy Dispatching

Thread state toggles frequently (per ipc)
- ready ↔ waiting
  - delete/insert ready list is expensive
  - therefore: delete *lazily* from ready list

disp table
Prio 100
Prio 50

ready → ready → ready

Lazy Dispatching

Thread state toggles frequently (per ipc)
- ready ↔ waiting
  - delete/insert ready list is expensive
  - therefore: delete *lazily* from ready list

disp table
Prio 100
Prio 50

ready → waiting → ready

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

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

too expensive
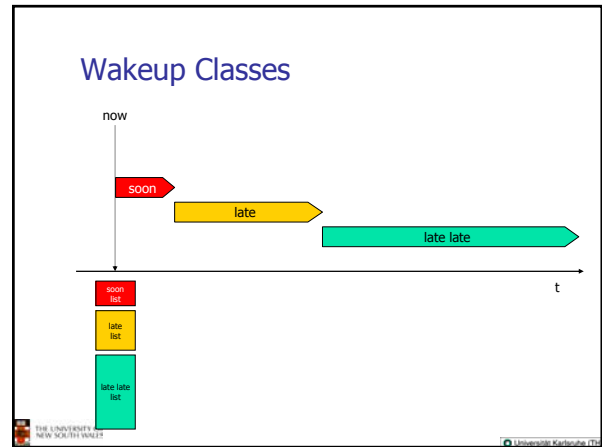
Timeouts & Wakeups

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

too expensive
too complicated

Wakeup Classes

now

insert timeout ($now + \Delta$)

soon

late

late late

t

soon list

late list

late late list

Wakeup Classes

now

soon

late

late late

t

soon list

late list

late late list

Wakeup Classes

now

soon

late

late late

t

soon list

late list

late late list

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

Wakeup Classes

now

soon

late

late late

t

soon list

late list

late late list

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

Wakeup Classes

now
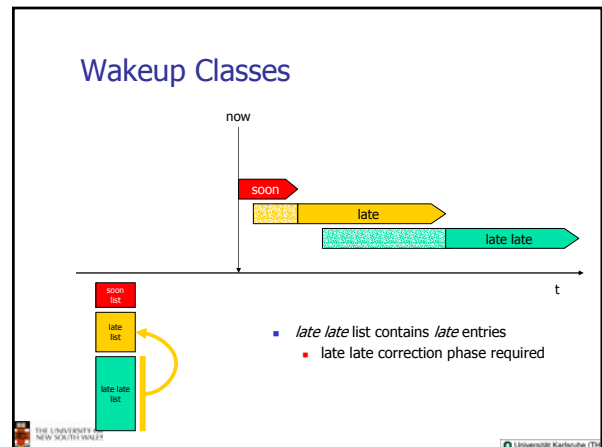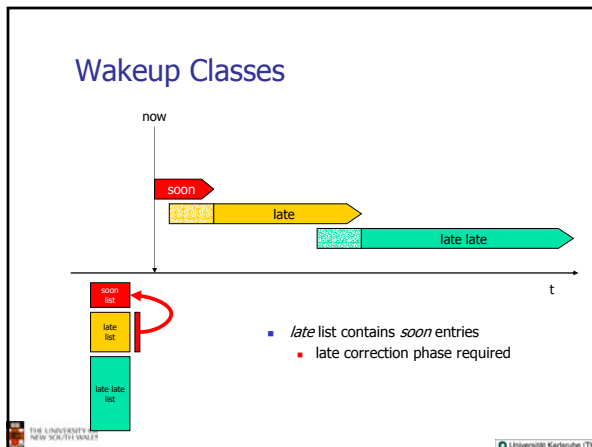
τ_{soon}

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
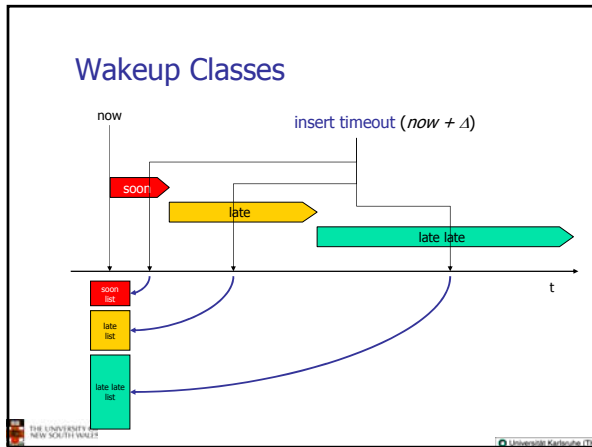
THE UNIVERSITY OF NEW SOUTH WALES

Universität Karlsruhe (TH)

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 $s..n \times 10..50$

still too expensive

- *raise timeout* costs:
  - delete head 20..100 cycles
- *delete timeout* costs:
  - delete entry 20..100 cycles

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

THE UNIVERSITY OF NEW SOUTH WALES

Universität Karlsruhe (TH)

Lazy Timeouts

insert (t_1)

soon

late

late late

t

soon list

Late

list

late late list

t_1

THE UNIVERSITY OF NEW SOUTH WALES

Universität Karlsruhe (TH)

Lazy Timeouts

insert (t_1)
delete timeout

soon

late

late late

t

soon list

Late

list

late late list

$\emptyset$

THE UNIVERSITY OF NEW SOUTH WALES

Universität Karlsruhe (TH)

Lazy Timeouts

insert (t_1)
delete timeout

insert (t_2)

soon

late

late late

t

soon list

Late

list

late late list

t_2

THE UNIVERSITY OF NEW SOUTH WALES

Universität Karlsruhe (TH)

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

THE UNIVERSITY OF NEW SOUTH WALES

Universität Karlsruhe (TH)

17

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

Authentication
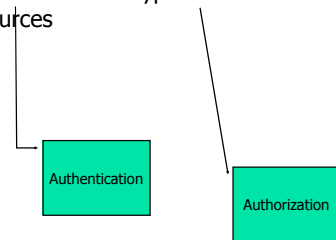
Authorization

The Microkernel Approach

User application

File system

TCP/IP
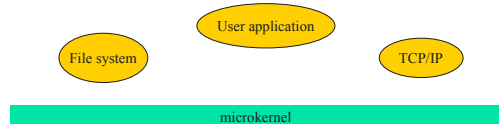
microkernel

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.

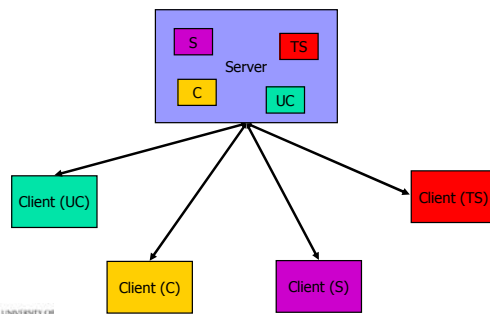THE UNIVERSITY OF NEW SOUTH WALES   Universität Karlsruhe (TH)

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???

THE UNIVERSITY OF NEW SOUTH WALES   Universität Karlsruhe (TH)

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) >= level(O)
- A subject (S) can write an object (O) iff
  - level(S) <= level(O)

THE UNIVERSITY OF NEW SOUTH WALES   Universität Karlsruhe (TH)

Secure System

THE UNIVERSITY OF NEW SOUTH WALES   Universität Karlsruhe (TH)

Problem

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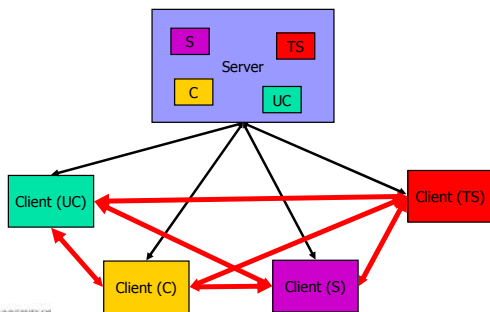
Conclusion

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.

THE UNIVERSITY OF NEW SOUTH WALES   Universität Karlsruhe (TH)

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

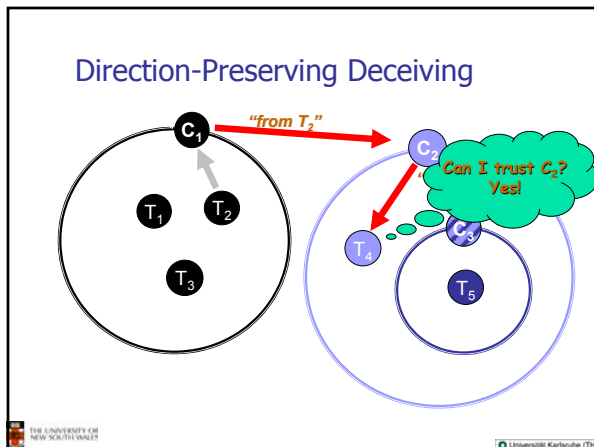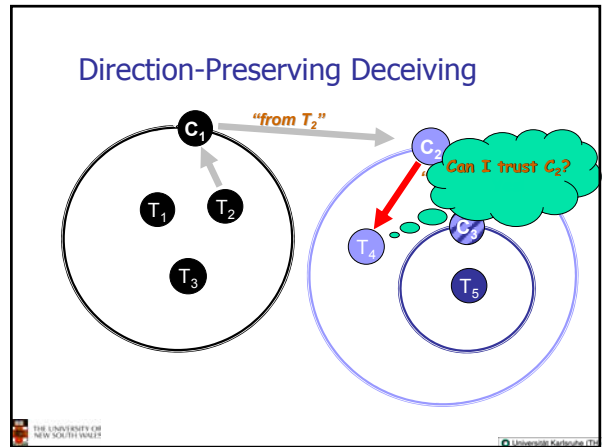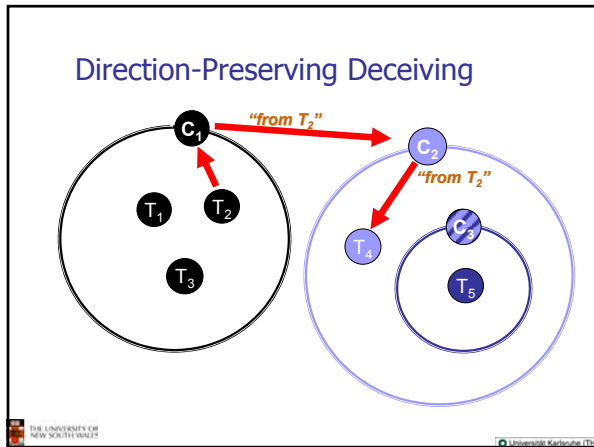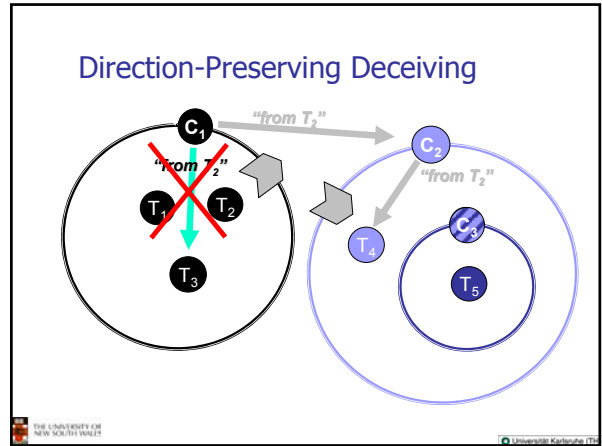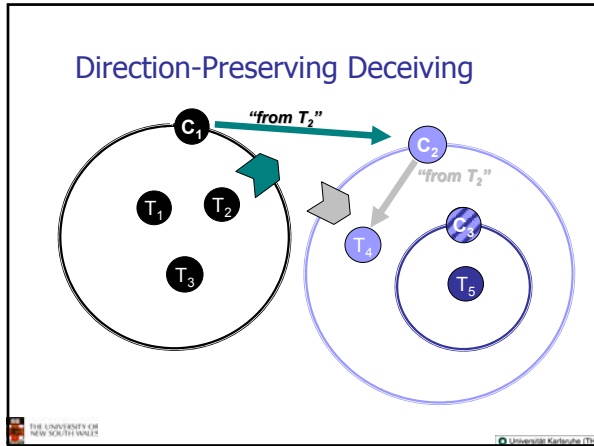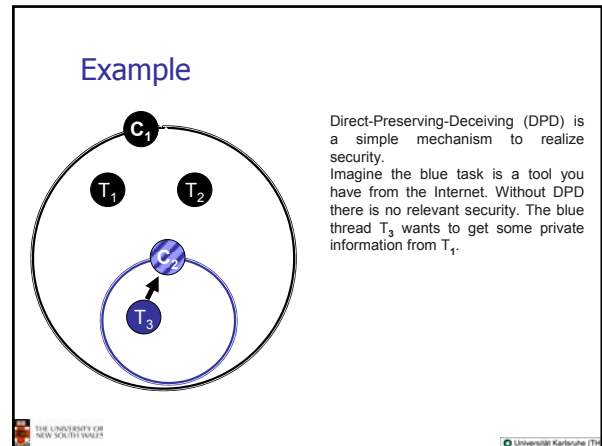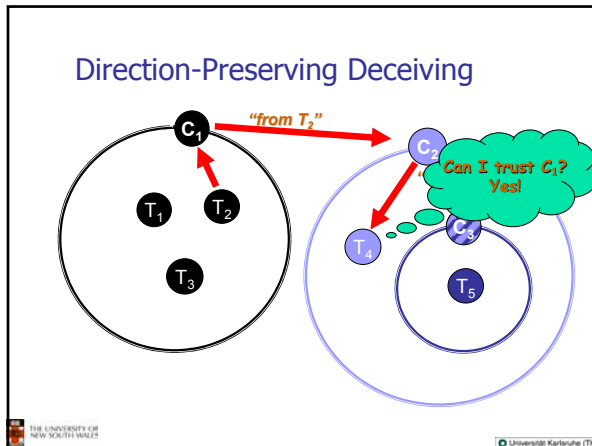- A *clan* is a set of tasks headed by a chief task

---

Intra-Clan IPC
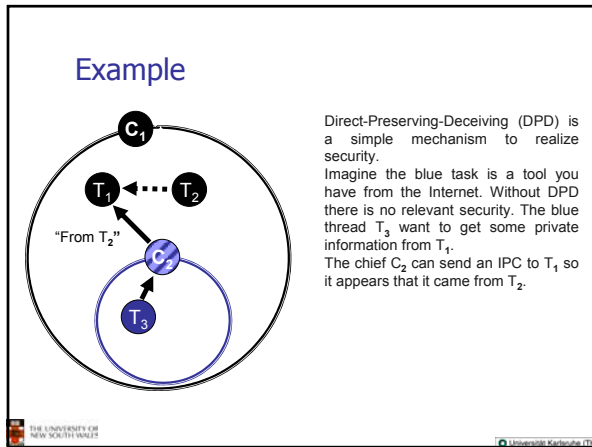
- Direct IPC by microkernel

Inter-Clan IPC

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

Direction-Preserving Deceiving
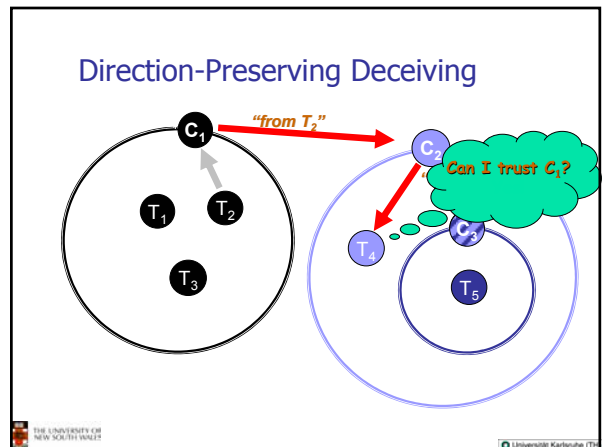
"from T_2"
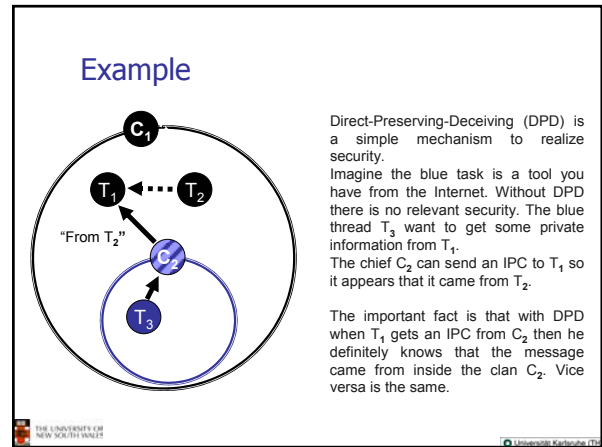
Direction-Preserving Deceiving

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.
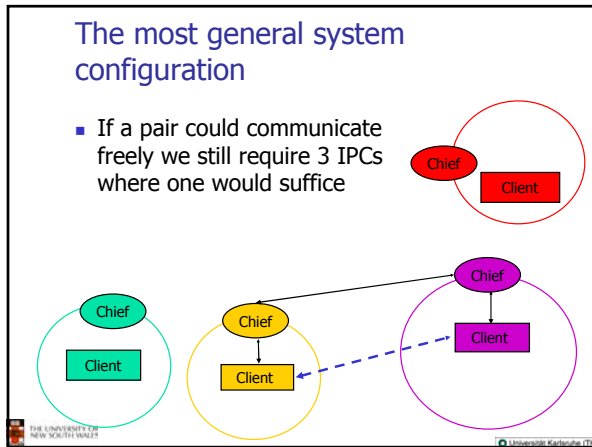
Example

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.
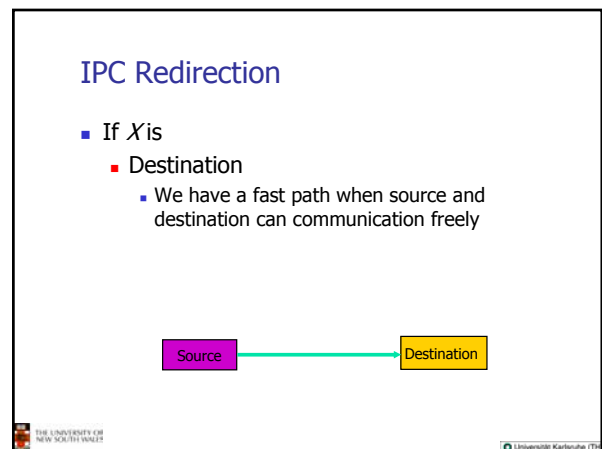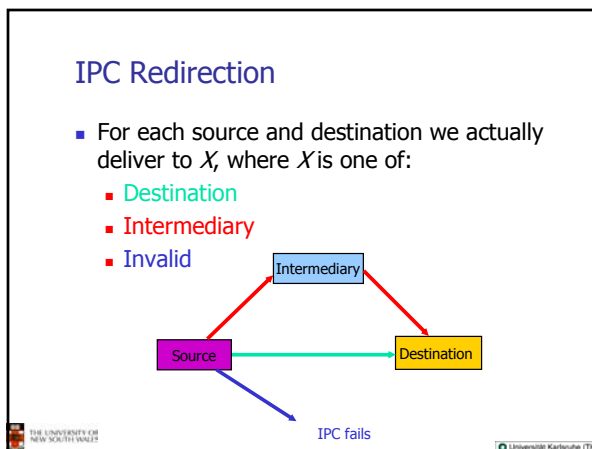
Remote IPC

Node A   **Node B**
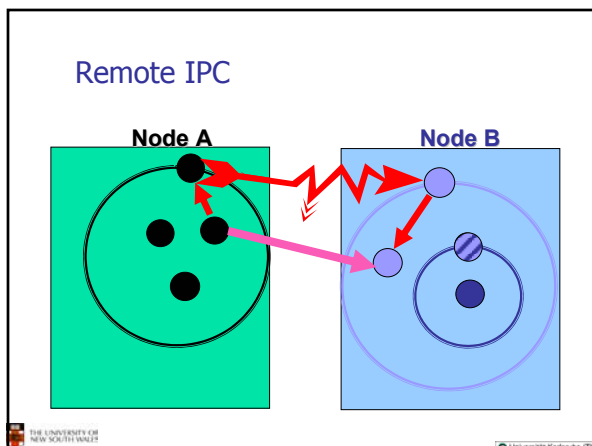
Clans & Chiefs

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

22

Secure System using Clans & Chiefs
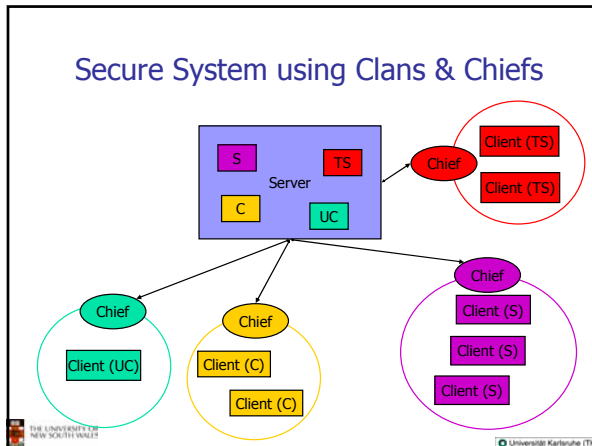
Problems with Clans & Chiefs

- 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 communication freely

23

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

Intermediary

Source

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]=> D
  - Iff R(S,D) = I or
  - R(S,D) = x and I[x]=>D

Case 1

- I[S]=>D if R(S,D) = I

Intermediary

From S

Source

Destination

Case 2

- I[S]=>D if R(S,D) = x, and I[x]=>D

X

Intermediary

From S

Source

Destination

Secure System using IPC Redirection

Redirection Controller

Server

S

TS

C

UC

Client (S)

Client (C)

Client (TS)

Client (C)

Client (UC)

Client (TS)

Client (S)

Client (S)

IPC Redirection can implement Clans & Chiefs
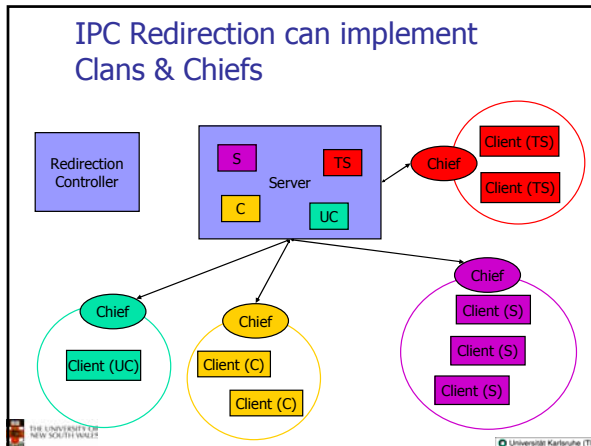
Disadvantages and Issues

- 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