Microkernel Construction

IPC Implementation

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)

- 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: hardwire TLB entries for kernel code and data.
- Single TLB entry.

TLB Problem
- Walking a linked list has a TLB footprint.

Avoid Table Lookups
- TCB = TCB_area + (thread_no & TCB_size_mask)

Validate Thread ID
- Are the thread IDs equal?

Branch Elimination
- Reduces branch prediction footprint.
- 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

**Copy area**

**Debug registers**

**Slow & Fast**

- User mode
  - IPC wait via slow path
  - IPC send via fast path

- User mode
  - IPC wait via fast path

**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.
- Up to 10 physical registers
- Virtual register copy loop

**Slow Path vs. Fast Path**

**Inter vs. Intra Address Space**
IPC - Implementation

Long IPC

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)

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 out of kernel buffer
- costs for $n$ words
  - $2 \times n$ 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
- 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

switch to dest space

current AS

[When leaving curr thread during ipc?]

inhibit PTE
flush TLB
Temporary mapping

When returning to thread during ipc:

- 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!

Temporary mapping

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

Why?

Leaves thread:
if mytcb.waddr a nil then
myPDE.TMarea := nil ;
flush TLB;
if dest AS = my AS then

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.

Temporary mapping

Page Fault Resolution:

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.

Temporary mapping

Page Fault Resolution:
Page Fault Resolution:

- If $\text{myPDE.TMarea} = \text{destPDE.destarea}$ then
  - tunnel to (partner) ;
  - access dest area ;
  - tunnel to (my)
- $\text{myPDE.TMarea} := \text{destPDE.destarea}.$

Temporary mapping

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

Cost estimates

<table>
<thead>
<tr>
<th></th>
<th>Copy in - copy out</th>
<th>Temporary mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td>R/W operations</td>
<td>$2 \times 2n$</td>
<td>$2n$</td>
</tr>
<tr>
<td>Cache lines</td>
<td>$3 \times n/8$</td>
<td>$2 \times n/8$</td>
</tr>
<tr>
<td>Small $n$ overhead cache misses</td>
<td>$n/8$</td>
<td>$0$</td>
</tr>
<tr>
<td>Large $n$ cache misses</td>
<td>$5 \times n/8$</td>
<td>$3 \times n/8$</td>
</tr>
<tr>
<td>Overhead TLB misses</td>
<td>$0$</td>
<td>$n / \text{words per page}$</td>
</tr>
<tr>
<td>Startup instructions</td>
<td>$0$</td>
<td>50</td>
</tr>
</tbody>
</table>
486 IPC costs
- Mach: copy in/out
- L4: temp mapping

Dispatching topics:
- thread switch
  - (to a specific thread)
  - to next thread to be scheduled
  - 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 ():
- Optimizations:
  - disp thread is special
  - no user mode
  - no own AS required
  - Can avoid AS switch
  - Freedom from tcb layout conventions
  - almost stable (low priority)
    - No need to preserve internal state between invocations
    - External state must be consistent
  - costs (A → B)
  - costs (A → disp → B)
  - costs (select next)
  - costs (A → disp → A) are low

Switch to ():
- labels:
  - 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
Example: Simple Dispatch

Example: Dispatch with ‘Tick’

Example: Dispatch with Interrupt
**Example: Dispatch with Interrupt**

Dispatcher stack

```
Int Thrd  
esp  
flg  
cs  
eip  
edi ... eax  
```

tcb A  

Local State  

```
Local State  
sp  
flg  
cs  
eip  
edi ... eax  
```

Switch to (): dispatcher thread is also idle thread

```
Thread A
```

Dispatcher Thread

```
Thread B
```

Priorities

- Optimization
  - keep highest active prio

```
do
  p := 255;
doif current\[p\] \(\ne\) nil
  then B := current\[p\] ;
  fi
  if current\[p\] \(\ne\) nil
  then return
  fi
  if B \(\ne\) nil
  then return
  fi
  p -= 1
  until p < 0 od;
```

Priorities, Preemption

```
Highest active p := max (new p, highest active p)
doiif current\[p\] \(\ne\) nil
  then B := current\[p\] ;
  fi
  if current\[p\] \(\ne\) nil
  then return
  fi
  if highest active p \(\ne\) nil
  then highest active p -= 1
  else idle
  fi
  if highest active p \(\ne\) nil
  then highest active p -= 1
  else idle
  fi
  p := 110 (int/wakeup)
```

---

9/10/2003
Priorities, Preemption

- What happens when a priority falls empty?

```
do
  if current_highest_active_p ≠ nil
    B := current_highest_active_p;
    return
  elif highest_active_p > 0
    highest_active_p -= 1
  else
    idle
  fi
od.
```

Round Robin if necessary:
```
if curr[hi act p].rem_ts = 0
  curr[hi act p] := next;
  current[hi act p].rem_ts := new_ts
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

Whenever reaching a non-ready thread,
- delete it from list
- proceed with next

Timeouts & Wakeups

- Operations:
  - insert timeout
  - raise timeout
  - find next timeout
  - delete timeout

Idea 1: unsorted list

- Insert timeout costs:
  - Search + insert entry
  - find next timeout costs:
    - parse entire list
  - raise timeout costs:
    - delete found entry
  - delete timeout costs:
    - delete entry

*raised-timeout costs are uncritical
(since only after timeout was taken)
- most timeouts are never raised!
Timeouts & Wakeups

Idea 1: 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

Idea 2: sorted list
- 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

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

Wakeup Classes

- Late list contains soon-entries
  - Late correction phase required

- Late late list contains late-entries
  - Late late correction phase required
Wakeup Classes

- now
- soon
- late
- late
- late
- late
- list
- list
- list

$\tau_{soon}$

$\max s$? (length of soon list)

$\leq$ timeouts to be raised in $\tau_{max}$ + new timeouts in $\tau_{max}$

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

Timeouts & Wakeups

- Insert timeout costs:
  - select class + add entry $10 + 20..100$ cycles
  - find next timeout costs:
    - search soon class $10..50$ cycles
  - raise timeout costs:
    - delete head $20..100$ cycles
  - delete timeout costs:
    - delete entry $20..100$ cycles

- Raised timeout costs are uncritical
  - Issuer only after timeout exp time
  - BUT most timeouts are never raised!

Lazy Timeouts

- Insert $t_1$
- Delete timeout

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
  - low: 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.

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)

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

Within all systems 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

Can I trust C2? Yes!

Can I trust C3? Yes!
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 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.

Remote IPC

Node A

Node B

Clans & Chiefs
- Remote IPC
- Multi-level security
- Debugging
- Heterogeneity
Secure System using Clans & Chiefs

- Client (UC)
- Server
- C
- Client (C)
- Client (TS)
- S
- Client (TS)
- Client (S)
- TS
- Chief
- Client (C)
- Client (S)
- Client (S)
- Client (S)
- Client (C)
- Chief
- Chief
- Chief
- Chief

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

- 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

**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**

- Redirection Controller
- Server
- Client (C)
- Client (S)
- Client (TS)
- Client (UC)
- Client (US)
- Client (SC)
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