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 Importance
(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)
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

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
  - poor 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:
    - e.g. use lazy scheduling
  - avoid in common case:
    - e.g. timeouts

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

---

Optimized Memory

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

---

TLB Problem

- Walking a linked list has a TLB footprint.

---

Avoid Table Lookups

- thread ID
- thread no
- version

- TCB = TCB_area + (thread_no & TCB_size_mask)

---

Validate Thread ID

- thread ID
- thread no
- version

- Are the thread IDs equal?

---

Branch Elimination

- Common case: 1
  - Reduces branch prediction footprint
  - Avoids mispredicts & stalls & flushes
  - Increases latency for slow path

- Common case: 0
TCB Resources

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

Message Transfer

- IBM PowerPC 750, 550 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
- Slow Path vs. Fast Path
- Number of message requests vs. IPC performance

Inter vs. Intra Address Space

- L4Ka: Pistachio IPC performance
- Pentium 3
- Inter vs. Intra Address Space
- Number of message requests vs. IPC performance

IPC - Implementation

Long IPC

- System-call preamble (disable intr)
- Identify dest thread and check name check
- Ready to receive?
- Analyze msg and transfer
- Long/msg:
- 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!
(end of timeslice, device interrupt…)

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

IPC - mem copy

- Why is it needed? Why not share?
  - Security
  - 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**
- 2×2 g r/w operations
- 3×r/w cache lines
- 1×r/w overhead cache misses (small r)
- 4×r/w cache misses (large r)

Temporary mapping

- Select dest area (4+4 M)
- Map into source AS (kernel)
- Copy data
- Switch to dest space
temporary mapping

- invalidate PTE
- flush TLB

when leaving curr thread during ipc?

current AS

- invalidate PTE
- flush TLB

when leaving curr thread during ipc:

current AS

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:
Requires separation of TLB flush and load PT root!

Thread switch:
- if TLB flushed := true
  
Alternative method:
- if myPDE.TMarea != nil
  then
    TLB flushed := false
    myPDE.TMarea := destPDE.destarea.

Page Fault Resolution:
- if myPDE.TMarea = destPDE.destarea
  then
    load PT root implicitly includes TLB flush on x86.

Page Fault Resolution:
- if myPDE.TMarea != nil
  then
    TLB flushed := true

Page Fault Resolution:
- if myPDE.TMarea = destPDE.destarea
  then
    load PT root implicitly includes TLB flush on x86.
### Cost estimates

<table>
<thead>
<tr>
<th>Operation</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>$3 \times n/B$</td>
<td>$2 \times n/B$</td>
</tr>
<tr>
<td>Small cache misses</td>
<td>$n/B$</td>
<td>0</td>
</tr>
<tr>
<td>Large cache misses</td>
<td>$5 \times n/B$</td>
<td>$3 \times n/B$</td>
</tr>
<tr>
<td>TLB misses</td>
<td>0</td>
<td>$n$ 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

![Graph showing Mach and L4 costs over time](image)

### Dispatching topics:

- **thread switch**
  - (to a specific thread)
  - to next thread to be scheduled
  - implicitly, when IPC blocks

- **priorities**
  - time slices
  - wakeups, interruptions

- **preemption**
  - timeouts and wake-ups
  - time

### Switch to ()

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

**Thread A**

1. Dispatcher Thread
   - switch to (unattached)
   - select next ready thread, return

2. switch to (A)

**Thread B**

1. Dispatcher Thread
   - switch to (unattached)
   - select next ready thread, return

2. switch to (A)

### Switch to ()

**Thread A**

1. Dispatcher Thread
   - $\text{thread}[A].sp := SP$;
   - $SP := \text{disp thread bottom}$

2. switch to (unattached)
   - select next ready thread, return

**Thread B**

1. Dispatcher Thread
   - $SP := \text{thread}[A].sp$;
   - if B = A then
     - switch from A to B
   - else return $f_1$

**Why?**

- Optimizations:
  - disp thread is special
  - no user mode
  - no save AS required
  - Can avoid AS switch
  - no AS required
  - Freedom from job layout conventions
  - almost static/nightly (per process)
  - No need to preserve internal state between invocations
  - External state must be consistent

- costs ($A \rightarrow B$)
- costs ($B \rightarrow A$)

- costs ($A \rightarrow B$)
- costs ($B \rightarrow A$) are low
Switch to ():

Example: Simple Dispatch

Example: Dispatch with 'Tick'

Example: Dispatch with 'Tick'
Example: Dispatch with Interrupt

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

Priorities
- Optimization
- keep highest active prio

```cpp
do p := 255;
if current_prio = nil
  then B := current_prio;
  fi
  p := 1
until p < 0 od;
if
```
What happens when a prio falls empty?

Do
- if current then B := current
- else
  - if highest active p > 0
  - then highest active p := 1
  - else
    - fi
    - fi
- fi
- p := p - 1
- until p < 0
- od.

Preemption

- Preemption, time slice exhausted

Lazy Dispatching

Thread state toggles frequently (per ipc)
- ready ↔ waiting
  - delete/insert ready list is expensive
  - therefore: delete body from ready list
Lazy Dispatching
Thread state toggles frequently (per ipc)
- \texttt{ready} ↔ \texttt{waiting}
- delete\slash insert ready list is expensive
- therefore: delete \texttt{lazy} from ready list

\textbf{Operations:}
- insert timeout
- raise timeout
- find next timeout
- delete timeout
Timeouts & Wakeups

Idea 1: unsorted list
- insert timeout costs: $n^2 \times 10..50 + 20..100$ cycles
- find next timeout costs:
  - parse entire list
  - $n = 10..50$ cycles
- raise timeout costs:
  - delete found entry
  - 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:
  - parse entire list
  - $n = 10..50$ cycles
- raise timeout costs:
  - delete list
  - 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:
  - parse entire list
  - $n = 10..50$ cycles
- raise timeout costs:
  - delete list
  - 20..100 cycles

Wakeup Classes

now

insert timeout (now + $\Delta$)

soon

late

too expensive

too expensive

too expensive - too complicated

19/09/2008
Wakeup Classes

- Late list contains soon entries
- Late correction phase required

Timeouts & Wakeups

Idea 4: unsorted wakeup classes
- Insert timeout costs:
  - Select class + add entry
  - Find next timeout costs:
    - Search soon class
    - Raise timeout costs:
      - Delete head
      - Delete timeout costs:
        - Delete entry

- Raised timeout costs are unrealistic (occur only after timeout exp time)

BUT most timeouts are never raised!
Lazy Timeouts

- \( t_1 \) insert
- \( t_2 \) delete timeout
- \( t_3 \) insert

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

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

Issue

- How common is insertion compared to wake up list searching/sorting?
  - Very
    - IPC more frequent than ‘ticks’
    - Wakeup queues always unsorted
    - Approach seems dubious

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
- 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

Client (UC) → Server → Client (TS) → Client (UC) → Client (C) → Client (S)

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 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 ... 

Direction-Preserving Deceiving

Can I trust C2? Yes!
Direct-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 \( T_2 \) wants 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 \( T_2 \) wants 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 it 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

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
  \[ R(S,D) = x \text{ and } I[x] => D \]

Case 1

\[ I[S] => D \text{ if } R(S,D) = I \]
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 is 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, Capabilities