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
Short IPC (uniprocessor) "send" (eagerly)
- System-call pre (disable intr)
- Identify dest thread and check:
  - same chief / no ipc redirection?
  - ready-to-receive?
- Shorten: no action required
- Analyze msg and transfer
- Switch to dest thread & address space
- System-call post

Short IPC (uniprocessor) "send" (lazyly)
- System-call pre (disable intr)
- Identify dest thread and check:
  - same chief / no ipc redirection?
  - ready-to-receive?
- Shorten: no action required
- Analyze msg and transfer
- Switch to dest thread & address space
- System-call post
What affects performance?

- Poor code "optimizations" thousands of invocations per second
- Performance is critical:
  - 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

Implementation Goal

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

Note: "payload" from green thread
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
- Avoid Memory References
- Use lazy scheduling for IPC
- Avoid timeouts in common case
- Microkernel minimizes indirect costs
- Single TLB entry

TLB Problem
- Walking a linked list has a TLB footprint

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

Message Transfer

- IBM PowerPC 750, 500 MHz, 32 registers
- Many cycles wasted on pipe flushing 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

- system-call preamble (disable intr)
- identify dest thread and check
  - same chief
  - ready-to-resume?
- analyse msg and transfer
  - long/map:
    - transfer message
    - switch to dest thread & address space
    - system-call postamble

Pagefaults possible!
(Prefetch data to dest address space)

Preserves possible!
(end of thread, 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
    - 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)

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
  - \( 2 \times 2 \) 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
Temporary mapping

- Problems
  - Multiple threads per AS
  - Mappings might change while message is copied

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

Temporary mapping

- Invalidate PTE
- Flush TLB

- When leaving current thread during IPC?

Temporary mapping

- Validating temp mapping requires to store partner id and dest area address in the sender’s tlb.

- Note: receiver’s page mappings might have changed!
**Temporary Mapping**

- **Page Fault**
  - Resolution:

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

  - Leave thread:
    - if mytcb.display := nil then
      - myPDE.TMarea := nil ;
    - if nil := nil then
      - flush TLB

  - Close temp mapping:
    - mytcb.display := nil ;
    - myPDE.TMarea := nil

- **Alternative method**:

  - Requires separation of TLB flush and load PT root.

  - Thread switch:
    - if TLB flushed := true then
      - flush TLB
    - else:
      - flush TLB
    - PT root := ...

**Temporary Mapping**

- **Page Fault**
  - Resolution:

  - Load PT root implicitly.

**Temporary Mapping**

- **Page Fault**
  - Resolution:

  - TM area PF:
    - if myPDE.TMarea = destPDE.destarea
      - myPDE.TMarea := destPDE.destarea .

**Temporary Mapping**

- **Page Fault**
  - Resolution:

  - TM area PF:
    - if myPDE.TMarea = destPDE.destarea then
      - tunnel to (partner) ;
    - access dest area ;
    - tunnel to (my)

    - myPDE.TMarea := destPDE.destarea .
**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>$2 \times n$</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
  - (to nil)
  - implicitly, when ipc blocks

- priorities
- preemption
  - time slices
  - wakeups, interruptions
- timeouts and wake-ups
- time

**Switch to ():**

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

**Switch to ():**

- Optimizations:
  - disp thread is special
  - no user mode
  - no user AS required
  - Can avoid AS switch
  - no dl required
  - Freedom from tlb layout conventions
  - almost stateless
  - No need to preserve internal state between invocations
  - External state must be consistent

- costs ($A \rightarrow B$)
- costs ($B \rightarrow A$)
- costs ($A \rightarrow \text{dis} \rightarrow A$)
- costs ($A \rightarrow \text{dis} \rightarrow B$) are low

$$\text{disp}([A], ap) \leftrightarrow \text{SP}; \quad \text{SP} \leftrightarrow \text{disp-thread bottom}$$
Example: Dispatch with Interrupt

Example: Dispatch with Interrupt

Example: Dispatch with Interrupt

Switch to ():

Priorities

Priorities

Example: Dispatch with Interrupt

Example: Dispatch with Interrupt

Switch to ():

Priorities

Priorities
Priorities, Preemption

What happens when a prio falls empty?

Do if current p = nil then reset current
else if highest active p > 0 then highest active p -- 1
else fi
fi

Preemption

Preemption, time slice exhausted

Do if current p = nil then reset current
else if highest active p > 0 then highest active p -- 1
else fi
fi

Lazy Dispatching

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

22/10/2007
Lazy Dispatching

Thread state toggles frequently (per IPC):
- Insert/Waiting
- Delete/Insert ready list is expensive
- Therefore: delete only from ready list

Lazy Dispatching

Thread state toggles frequently (per IPC):
- Insert/Waiting
- Delete/Insert ready list is expensive
- Therefore: delete only from ready list

Lazy Dispatching

Thread state toggles frequently (per IPC):
- Insert/Waiting
- Delete/Insert ready list is expensive
- Therefore: delete only from ready list

Lazy Dispatching

Thread state toggles frequently (per IPC):
- Insert/Waiting
- Delete/Insert ready list is expensive
- Therefore: delete only from ready list

Timeouts & Wakeups

Operations:
- Insert timeout
- Raise timeout
- Find next timeout
- Delete timeout

Timeout costs are uncritical (issue only after timeout, not 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
- raise timeout costs:
  - delete found entry 20..100 cycles
- delete timeout costs:
  - delete entry 20..100 cycles

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

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
Wakeup Classes

- late list contains soon entries
- late correction phase required

Timeouts & Wakeups

- insert timeout costs:
  - select class + add entry
  - find next timeout costs:
    - search soon class
  - raise timeout costs:
    - delete head
    - delete timeout costs:
      - delete entry
      - late late correction phase required

Lazy Timeouts

- insert(s)
- delete timeout
Lazy Timeouts

- Insert ($t_1$)
- Delete timeout
- Insert ($t_2$)

Lazy Sorting

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

Incremental Sorting

- Combine the cost of sorting with cost of finding first thread to wake
- Problem: every addition to list resets the sorted flag, and thus we must perform entire list walk. But we want to avoid this.
- Alternative: maintain sorted list, and unsorted list. Merge the two lists when necessary.
  - Merge can be incremental bubble sort
  - 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
- Server 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) if
  - level(S) >= level(O)
- A subject (S) can write an object (O) if
  - level(S) <= level(O)

Secure System
- Client (UC)
- Client (C)
- Client (S)
- Client (TS)
- Server
Problem

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

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

Can I trust C₂?
Yes!
Can I trust C1? 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 T1 wants to get some private information from T2.

The chief C2 can send an IPC to T2, so it appears that it came from T2.

From T2:

“From T2”
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**

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

- If \( X \) is:
  - Destination
    - We have a fast path when source and destination can communicate freely
  - Intermediary
    - Enforce security policy
      - Monitor, analyze, reject, modify each IPC
      - Audit communication
      - Debug
  - 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]=> 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
Case 2
- 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