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

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) "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
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
  - 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:
    * 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

Message Transfer
- up to 10 physical registers
- virtual register copy loop
- Many cycles wasted on pipe flushes for privileged instructions.

IBM PowerPC 750, 500 MHz, 32 registers

Slow Path vs. Fast Path

Inter vs. Intra Address Space

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

- Pagefaults possible on source and dest address space
- Preemptions possible end of timeslice, device interrupt
Long IPC (uniprocessor)
- system-call pre (disable int)
- identify dest thread and check
  - same chart
  - 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

Pagefaults possible!

Long IPC (uniprocessor)
- system-call pre (disable int)
- identify dest thread and check
  - same chart
  - 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

Object small than a page or not aligned

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
  - $2 \times 2n$ 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 $n$)
  - $1 \times n/8$ overhead cache misses
  - $4 \times n/8$ cache misses

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 curr 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

  - current AS

- when returning to thread during ipc:

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

- **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; flush TLB; TLB flushed := true.

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

**Alternative Method:**

- Leave thread:
  - if mytcb.waddr ≠ nil then
    - myPDE.TMarea := nil; flush TLB; TLB flushed := true.

- Thread switch:
  - if TLB just flushed then TLB flushed := false.

- Does therefore not work reasonably on x86.

- Load PT root implicitly includes TLB flush on x86.

**Temporary Mapping:**

- Page Fault Resolution:

---

**Temporary Mapping:**

- Page Fault Resolution:

---

**Temporary Mapping:**

- Page Fault Resolution:

---

**Temporary Mapping:**

- Page Fault Resolution:
Cost estimates

<table>
<thead>
<tr>
<th>R/W operations</th>
<th>Copy in - copy out</th>
<th>Temporary mapping</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2 × 2n</td>
<td>2n</td>
</tr>
<tr>
<td>Cache lines</td>
<td>3 × n/8</td>
<td>2 × 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 × n/8</td>
<td>3 × n/8</td>
</tr>
<tr>
<td>Overhead 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

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

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
  - no need to preserve internal state
  - External state must be consistent
- costs (A → B)
  - costs (A → disp → B)
  - costs (select next)
  - costs (A → disp → A) are low

Switch to ():

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

8/09/2006
Switch to (A):

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

Example: Simple Dispatch

### Why does this always work?

Example: Dispatch with 'Tick'

### Local State

- Local Variables

-Diagram of thread state transition

Example: Dispatch with 'Tick'

-Diagram of thread state transition

Example: Dispatch with 'Tick'

-Diagram of thread state transition

Example: Dispatch with 'Tick'

-Diagram of thread state transition
Example: Dispatch with Interrupt

Dispatcher stack

Int Thrd

tcb A

Disp Table

Priorities
- 0 (lowest) ... 255
- hard priorities
- round robin per prio
- dynamically changeable

Priorities
- Optimization
- keep highest active prio

Switch to ():
- dispatcher thread is also
  - idle thread

Disp Table

Prio 100
Prio 50

do
  p := 255;
do
  if current[p] ≠ nil then B := current[p] ;
  fi ;
  p := p - 1;
  if p = 0 then idle ;
  else
    [current]
  fi

Disp Table

Prio 100
Prio 50

do
  if current[p] ≠ nil then B := current[p] ;
  fi ;
  B := next ready (B);
  if B = nil then
    idle ;
  else
    [current]
  fi

Disp Table

Prio 100
Prio 50

do
  p := 255;
do
  if current[p] ≠ nil then B := current[p] ;
  fi ;
  p := p - 1;
  if p = 0 then idle ;
  else
    [current]
  fi

Disp Table

Prio 100
Prio 50
Priorities, Preemption

- What happens when a prio falls empty?

```
do if current[prio empty?] is nil then
    return
else
    highest active p := 0
    else
        highest active p := 1
    fi
    id := id
    until p < 0 od
```

Priorities, Preemption

- What happens when a prio falls empty?

```
do if current[highest active p] is nil then
    return
else
    highest active p := 0
    else
        highest active p := 1
    fi
    id := id
    until p < 0 od
```

Preemption

- Preemption, time slice exhausted

```
do if current[prio empty?] is nil then
    return
else
    highest active p := 0
    else
        highest active p := 1
    fi
    id := id
    until p < 0 od
```

Lazy Dispatching

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

Preemption

- Preemption, time slice exhausted

```
do if current[prio empty?] is nil then
    return
else
    highest active p := 0
    else
        highest active p := 1
    fi
    id := id
    until p < 0 od
```

Priorities, Preemption

- What happens when a prio falls empty?

```
do if current[highest active p] is nil then
    return
else
    highest active p := 0
    else
        highest active p := 1
    fi
    id := id
    until p < 0 od
```
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

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, do
- delete it from list
- proceed with next