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
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

running

wait to receive

running

running
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

wait to receive

running

running

running
IPC
IPC

Note “payload” from green thread
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: hard-wire TLB entries for kernel code and data.

Single TLB entry.

TCB state, grouped by cache lines.
Walking a linked list has a TLB footprint.
Avoid Table Lookups

\[ \text{TCB} = \text{TCB}_\text{area} + (\text{thread}_\text{no} \& \text{TCB}_\text{size}_\text{mask}) \]
Validate Thread ID

Are the thread IDs equal?
Branch Elimination

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

if (slow)
    enter_slow_path()

- 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

Diagram:
- Resources bitfield
  - 11
- Copy area
- Debug registers
Message Transfer

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

![Graph showing L4Ka::Pistachio IPC performance for Pentium 3. The x-axis represents the number of message registers, ranging from 0 to 60. The y-axis represents cycles, ranging from 0 to 600. Two lines are plotted: one for Inter C-Path (purple dots) and one for Inter FastPath (red squares). The graph illustrates the performance difference between the two paths.]
Inter vs. Intra Address Space

L4Ka::Pistachio IPC performance
Pentium 3

number message registers vs. cycles

- 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

Pagefaults possible!
(in source and dest address space)

Preemptions 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

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
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
  - $2 \times 2n$ 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 $n$)
temporary mapping
temporary mapping

- select dest area (4+4 M)
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
temporal 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?
temporary mapping

- invalidate PTE
- flush TLB

when leaving curr thread *during* ipc:
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:

- ![Diagram of memory mapping](image)

*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
```

Close temp mapping:
```
mytcb.waddr := nil.
myPDE.TMarea := nil
```

**why?**

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

Leave thread:

<math>\text{if } \text{mytcb.waddr} \neq \text{nil} \text{ then}
\begin{align*}
\text{myPDE.TMarea} & := \text{nil} ; \\
\text{flush TLB} ; \\
\text{TLB flushed} & := \text{true}
\end{align*}
\text{fi .}
</math>

Thread switch :

<math>\ldots
\begin{align*}
\text{if TLB just flushed}
& \text{ then TLB flushed} := \text{false} \\
& \text{else flush TLB}
\end{align*}
\text{fi ;}
\text{PT root} := \ldots
</math>

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.
Page Fault Resolution:

- Temporary mapping

current AS
temporary mapping

- Page Fault Resolution:
temporary mapping

- Page Fault Resolution:
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
## Cost estimates

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

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

Thread A

Dispatcher Thread

switch to (dispatcher)

select next ready thread, assume B

Thread B

switch to (B)
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

- \( \text{tcb}[A].sp := SP; \)
- \( SP := \text{disp thread bottom} \)

Thread \( A \)

Dispatcher Thread

- switch to (dispatcher)
- select next ready thread, assume \( B \)
- switch to (\( B \))

Thread \( B \)

- \( SP := \text{tcb}[A].sp ; \)
- if \( B \neq A \) then
- switch from \( A \) to \( B \)
- else return
- fi.

- \( \text{Why ??} \)

- \( \text{costs} (A \rightarrow B) \)
- \( \approx \text{costs} (A \rightarrow \text{disp} \rightarrow B) \)
  - \( \text{costs} (\text{select next}) \)
  - \( \text{costs( } A \rightarrow \text{disp} \rightarrow A \text{) are low} \)
**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*

---

**Switch to ():**

$tcb[A].sp := SP$;
$SP := \text{disp thread bottom}.$

Why does this always work?
Example: Simple Dispatch
Example: Simple Dispatch
Example: Dispatch with ‘Tick’
Example: Dispatch with ‘Tick’
Example: Dispatch with ‘Tick’
Example: Dispatch with Interrupt
Example: Dispatch with Interrupt

Diagram showing the stacking of dispatcher stacks and thread contexts.
Example: Dispatch with Interrupt

Dispatcher stack

Int Thrd

Local State

x eip cs flg esp ss

tcb A

Local State

x eip cs flg esp ss

Dispatcher stack
Switch to (\texttt{()}):

- dispatcher thread is also \texttt{idle thread}

\begin{verbatim}
B := A ;
do
  B := next ready (B) ;
  if B \neq \texttt{nil}
    then return
  fi ;
  idle
od .
\end{verbatim}
Priorities

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

```java
do
    p := 255;
    do
        if current[p] ≠ nil
            then B := current[p] ;
                return
        fi ;
        p -= 1
    until p < 0 od ;
    idle
od .
```

- ready tcb list per prio
- 'current tcb' per list
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 $p := \max (\text{new } p, \text{highest active } p)$.
Priorities, Preemption

- What happens when a prio falls empty?
Priorities, Preemption

- What happens when a prio falls empty?

```plaintext
do
  if current[highest active p] ≠ nil
    then round robin if necessary;
      B := current[highest active p];
      return
  elsif 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.
```

What happens when a prio falls empty?
Preemption

- Preemption, time slice exhausted

```plaintext
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 .
```
Preemption

- Preemption, time slice exhausted

```plaintext
do
  if current[\text{highest active p}] \neq \text{nil}
    then round robin if necessary;
    B := current[\text{highest active p}];
    \text{return}
  elif highest active p > 0
    then highest active p -= 1
  else
    \text{idle}
  fi
od.

round robin if necessary:
  if current[\text{hi act p}].\text{rem ts} = 0
    then current[\text{hi act p}].\text{rem ts} := \text{new ts};
    current[\text{hi act p}] := \text{next}
  fi.
```
Lazy Dispatching

Thread state toggles frequently (per ipc)
- \textit{ready} ↔ \textit{waiting}
  - delete/insert ready list is expensive
  - therefore: delete \textit{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)
- \textit{ready} \Leftrightarrow \textit{waiting}
  - delete/insert ready list is expensive
  - therefore: delete \textit{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)
- \textit{ready} \leftrightarrow \textit{waiting}
  - delete/insert ready list is expensive
  - therefore: delete \textit{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\textsubscript{[highest active p]} ≠ nil
    then B := current\textsubscript{[highest active p]}; return
  elif highest active p > 0
    then highest active p -= 1
  else
    idle
  fi
od .

round robin if necessary:
  while curr\textsubscript{[hi act p]} ≠ nil do
    if curr\textsubscript{[hi act p]}\textsubscript{.state} ≠ ready
      then delete from list (curr\textsubscript{[hi act p]})
    elif curr\textsubscript{[hi act p]}\textsubscript{.rem ts} = 0
      then curr\textsubscript{[hi act p]}\textsubscript{.rem ts} := new ts
    else leave round robin if necessary
    fi ;
    curr\textsubscript{[hi act p]} := next ;
  od .