μ-Kernel Construction (7)

Virtual Memory Mapping
Address Space Construction

\[ \sigma_0 \]

physical memory
Mechanisms

- Address space construction
  - Map [populate]
  - Unmap [wipe out]

- Integrity and security
  - Access permissions [rwx]

- Resource control
  - Use bits [accessed or dirty]
  - Page fault messages [detect page use]
Review: Message Construction

- Typed items occupy one or more words
- Three currently defined items
  - Map item (2 words)
  - Grant item (2 words)
  - String item (2+ words)
- Typed items can have arbitrary order
Review: Map and Grant Items

- Two words
  - Send base
  - Fpage
- Lower bits of send base indicates map or grant item
Review: Receiving Messages

- Receiver buffers are specified in registers ($BR_0 \ldots BR_{33}$)

- First BR ($BR_0$) contains “Acceptor”
  - May specify receive window (if not nil-fpage)
  - May indicate presence of receive strings/buffers (if s-bit set)
Agreed to receive mapping (BR₀).
Unmap

Implicit consent to unmap.
Implicit consent to unmap.
Access Rights – Map

- Mapper may restrict access rights
- Cannot extend its own access rights

\[ r = \text{Read} \]
\[ w = \text{Write} \]
\[ x = \text{eXecute} \]
Access Rights – Unmap

- Mapper may revoke partial access rights
- Cannot extend other’s access rights
- Unmap transitively affects mappings

\[ r = \text{Read} \]
\[ w = \text{Write} \]
\[ x = \text{eXecute} \]
Mapping Regions
Mapping Regions: Flex Pages

- Abstraction: flex page
  - Contiguous regions of virtual address space
    - Sparse physical mappings possible
  - Called \textit{fpage}
  - Abstracts architecture’s page sizes

- Fpage semantics
  - Inseparable object
  - Aligned to its size
  - Size is power of 2, min. $1024=2^{10}$ byte
Fpage Encoding

Fpage( base, size=2^s )

s ≥ 10
base mod 2^s = 0

Special cases
- Complete address space (base=0, s=1)
- Nothing: nilpage (0)

<table>
<thead>
<tr>
<th>base / 2^{10}</th>
<th>s</th>
<th>0 r w x</th>
</tr>
</thead>
</table>

- 22/54 bits
- 6 bits
- 4 bits
Mapping Fpages

Map( fpage₁ to offset₁, fpage₂ to offset₂, fpage₃ to offset₃ )

MapAccept( fpage )

- Receive window too large?
  - Use offset (aka send base)
- Common case: page faults
  - Receive fpage is entire address space
  - Send offset is fault address
Status Bits

Referenced, Written, executed

Reset not visible here

Query & reset

Bitwise OR

Reset not visible here
Status Bits

- Referenced, Written, Executed
- Query & reset
- Bitwise OR
- Reset not visible here

Diagram:
- A
- B
- C
- D
- E
Resource Control – Not Enough Memory

1. Find set of pages $X$ to swap to disk
   - Check status bits
2. Unmap $X$ from idle client $C$
3. Write $X$ to disk
4. Map $X$ to hungry client
5. What happens when $C$ accesses $X$?
   1. Page fault
   2. Pager repeats algorithm, but to satisfy $C$
Page Miss Handling

```
pager: while (true) do
    wait for page fault;
    touch page;
    send page mapping;
    od.
```
Initial Address Space

- Idempotent mappings to sigma0
  - Virtual address = physical address
- User-level knows all virtual to physical mappings
  - Necessary for DMA
Implementation
Mapping Regions

- Implementation
  - Based on page tables
  - Physical page (frame)
    - Basic mapping unit
    - Determines minimum alignment

- Minimum fpage size
  - Physical page size
Mapping Pages

- Map pages by copying page table entries
- No support (yet) for
  - Recursive unmap
  - Combined status bits

Physical memory

map

AS(A)

AS(B)
Mapping Database

physical frames

physical frames

A

B

C

D

E

X

A[1]

B[7]

D[0]

C[2]

E[0]
Mapping Database – Create Mapping
How to find D[0] in the tree?
Search?
Mapping Database – Create Mapping

How to find D[0] in the tree? PTE Link?
Mapping Database – Create Mapping

How to find \( D[0] \) in the tree? Hash Table, Cache?
Mapping Database – Create Mapping
Mapping Database – Unmap
Mapping Database – Unmap

How to find E[0] and X[8] PTEs?
How to find E[0] and X[8] PTEs? PTE link?
Mapping Database – Unmap
Mapping Database – Unmap
More Implementation
General Implementation Guidelines

- Avoid recursion
  - Serialize recursive algorithms
- Allow for preemption
  - Implement appropriate locking mechanisms
- Minimize memory requirements
  - Make the right design decisions
  - Make structures compact
- Make it general
Avoiding Recursion – Basic Idea
How do we find B’s subtree?
Avoiding Recursion – Insertion

Inserting new node F
Avoiding Recursion – Deletion

Delete node D
Design Decisions – Mapnode Pointers

No need to parse mapping database

Useful for map and unmap operations

Mapnode link from page table entry
Design Decisions – x86 Implementation

Add 4KB
Design Decisions – Mapnode Caching

- Parse mapping database
- Update cache
- Miss
- Cache
Design Decisions – Backlink Pointers

No need to parse page tables

Very useful for unmap operations

Backlink pointer from mapping node
Design Decisions – Backlink Caching

Diagram showing a tree structure with nodes A, B, C, D, and E, and a cache.
Design Decisions – Integrated Solution

- Page tables and mapping database are integrated
  - Simple with hardware independent page tables
  - A bit more tricky with hardware dependent page tables
  - Requires preallocation of all MDB nodes
    - At least for all entries in valid page tables
  - Don’t combine for cache reasons
Locking

- Completely serialized
  - Only a single thread may map/unmap at a given time

- Coarse grained locking
  - Only a single thread may map/unmap pages inside some given physical region

- Fine grained locking
  - Only a single thread may access a given mapping node at a given time
Locking – Coarse Grained

Exclusive access
(map tree corresponds to a physical page)

Region lock array


lock
Locking – Fine Grained

- Each mapping node has two locks
  - Uplock protects previous (backlink) pointer
  - Downlock protects next pointer
Locking – Fine Grained – Avoid Deadlock

- When an uplock is acquired
  - Thread may hold it as long as it wants

- When a downlock is acquired
  - Thread must release it if it does not manage to acquire the uplock of its child/sibling
Locking – Fine Grained – Szenario 1

Thread T1 tries to unmap node E

Thread T2 tries to unmap node E
Locking – Fine Grained – Szenario 1

- T1 acquires A’s downlock
- T1 cannot acquire E’s uplock
- T2 acquires E’s uplock
Locking – Fine Grained – Szenario 1

T1 releases A’s downlock

T2 acquires A’s downlock
Locking – Fine Grained – Szenario 1

T2 removes the links between A and E

T2 releases the locks
Locking – Fine Grained – Szenario 2

Thread 1 tries to unmap node E

Thread 2 tries to create a new mapping
Locking – Fine Grained – Szenario 2

T1 acquires A’s downlock

T2 acquires E’s downlock

T1 acquires E’s uplock
Locking – Fine Grained – Szenario 2

T1 cannot acquire E’s downlock

T1 removes the links between A and E

T1 releases the lock
Locking – Fine Grained – Szenario 2

T1 acquires E’s downlock

T2 releases E’s downlock

T2 inserts the new mapping

T1 acquires E’s downlock

T2 releases E’s downlock

T2 inserts the new mapping
Structure Packing

- Use only required number of bits
  - E.g., page directory address does not require 32 bits

- Store multiple values in one word
  - Occasionally one of two values is known
    - When iterating a doubly linked list, either the previous (or the following) node is known
    - Must iterate from start or end (cannot start in between)
  - Store only ‘valueA XOR valueB’
  - Then
    - valueA == stored value XOR valueB
    - valueB == stored value XOR valueA