User-level Management of Kernel Memory

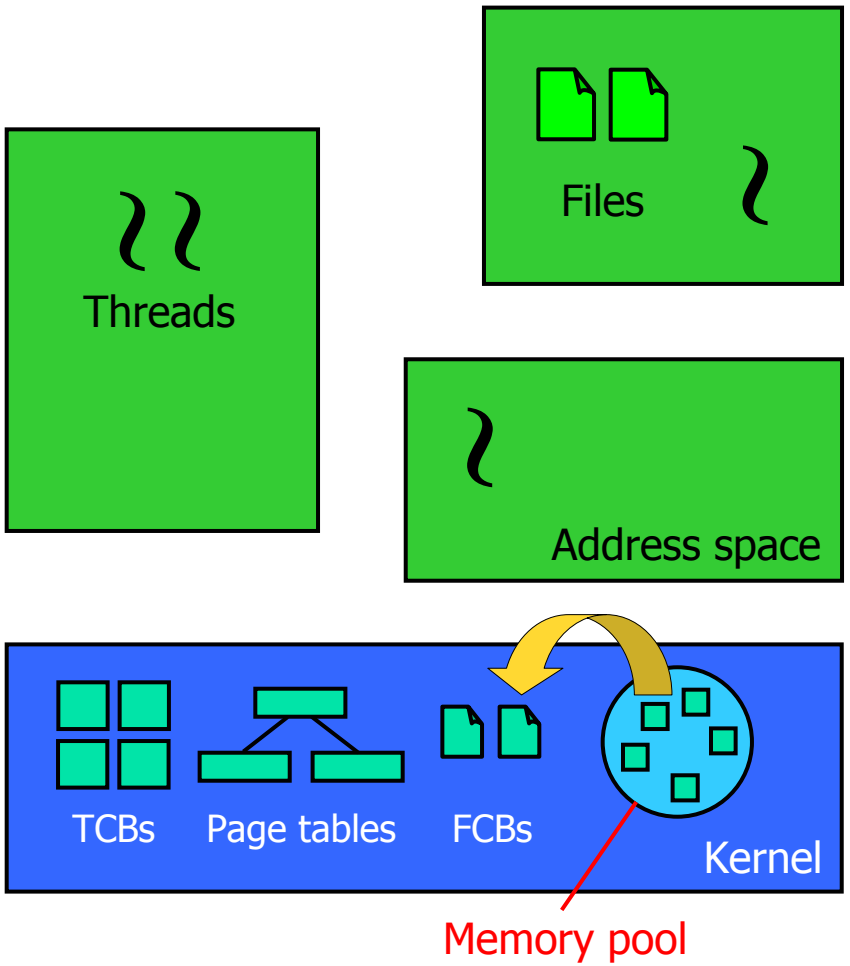
Andreas Haeberlen

University of Karlsruhe
Karlsruhe, Germany

Kevin Elphinstone

University of New South Wales
Sydney, Australia

Motivation: Kernel memory



- Kernel memory is needed to implement core abstractions
- Resource is limited; policy required to control allocation

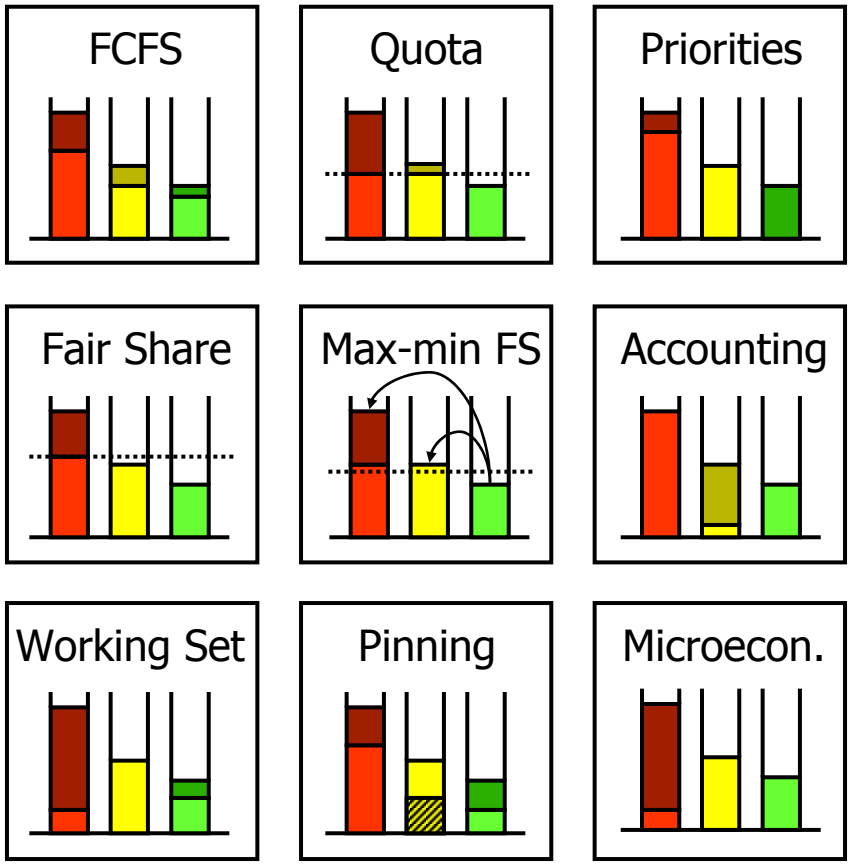
Motivation: Need for policy

```
while (1)
  if (!fork())
    exit();
```

```
gamma login: ahae
Password: secret
bash: fork: Resource temporarily
            unavailable
bash-2.03$ uname
bash: fork: Resource temporarily
            unavailable
bash-2.03$
```

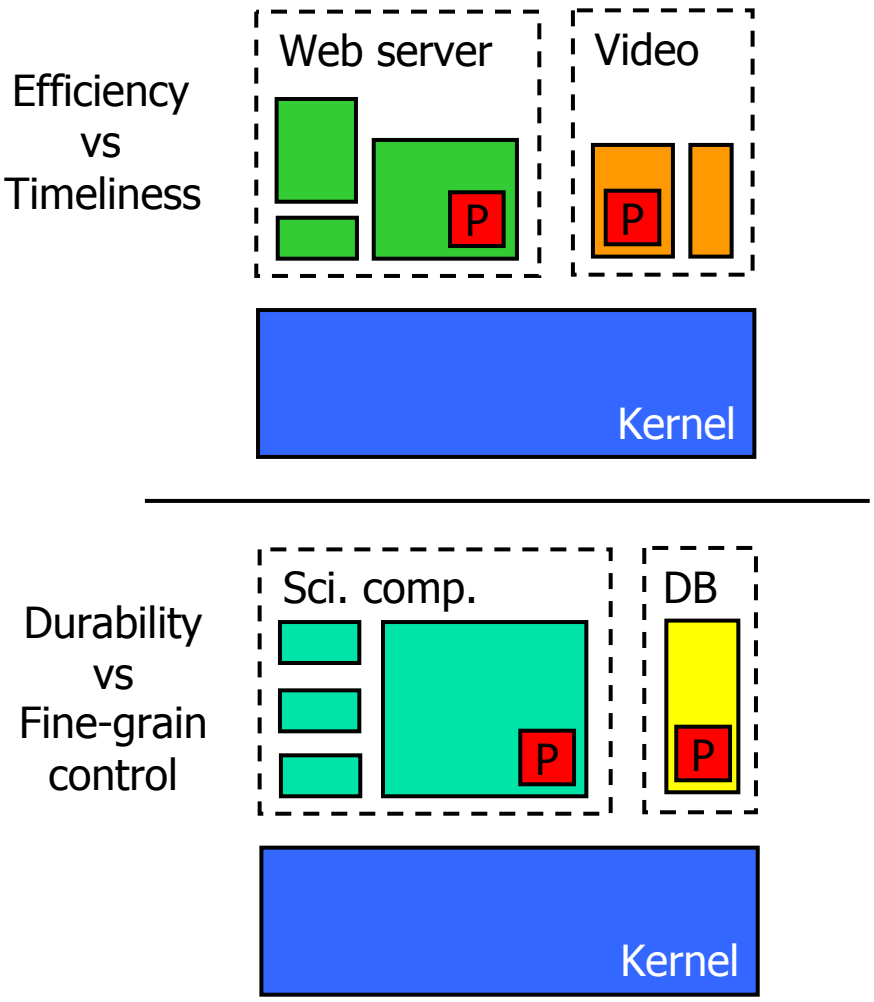
- FCFS is dangerous:
  - Denial of Service
  - No isolation
  - Not predictable
- Need a better policy

Motivation: The Perfect Policy



- Different scenarios require different policies
- "Perfect Policy" not known
- Solution: Move policy to user level, kernel only provides mechanism

Motivation: Flexibility

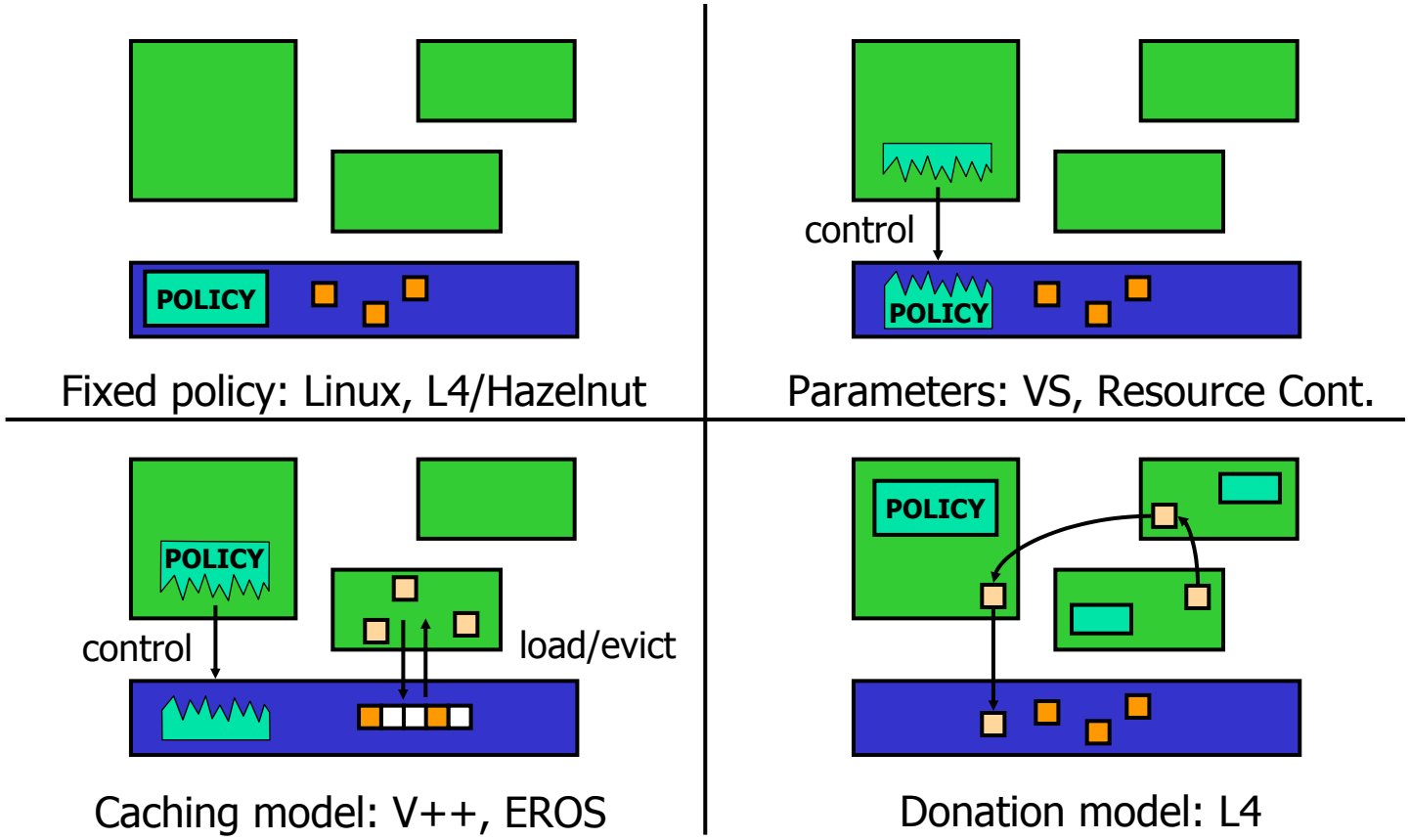


- Problem 1: Different systems have different requirements

  ⇒ Single policy is often a compromise
- Problem 2: Applications know their future needs better than the kernel

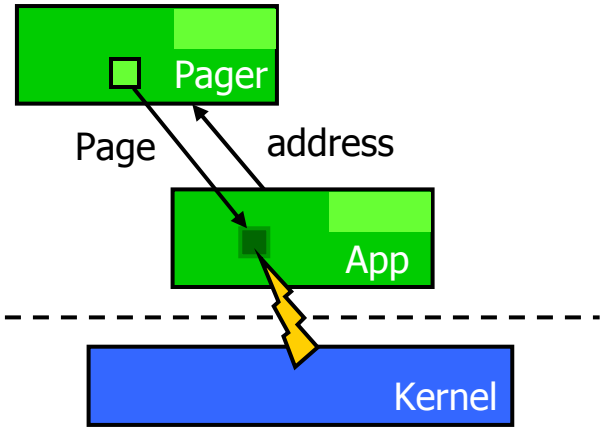
  ⇒ Suboptimal decisions
- Solution: Kernel provides mechanism, policy implemented at user level
  - Used extensively for normal virtual memory

Existing solutions



- Revocation and preemption are not supported

The Mechanism

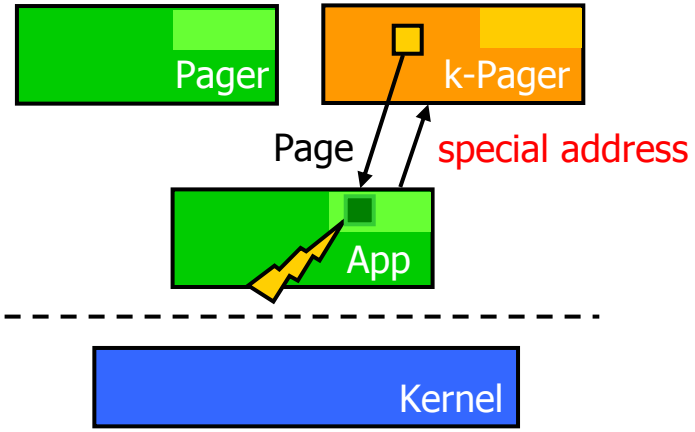


User page fault

1. Thread touches missing page
2. Kernel catches page fault, blocks thread, notifies pager
3. Pager allocates memory
4. Thread is resumed

. . .

5. Pager revokes memory



Kernel page fault

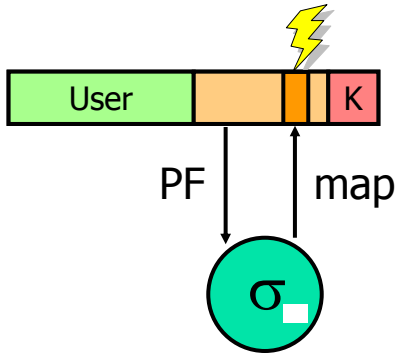
1. Thread invokes kernel primitive
2. Kernel detects missing metadata, blocks thread, notifies pager
3. Pager allocates memory
4. Thread is resumed

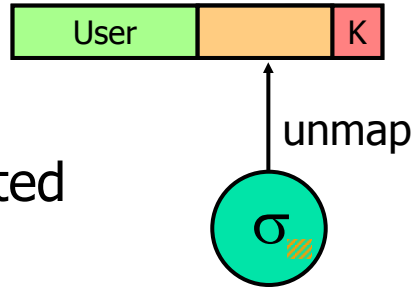
. . .

5. Pager revokes memory

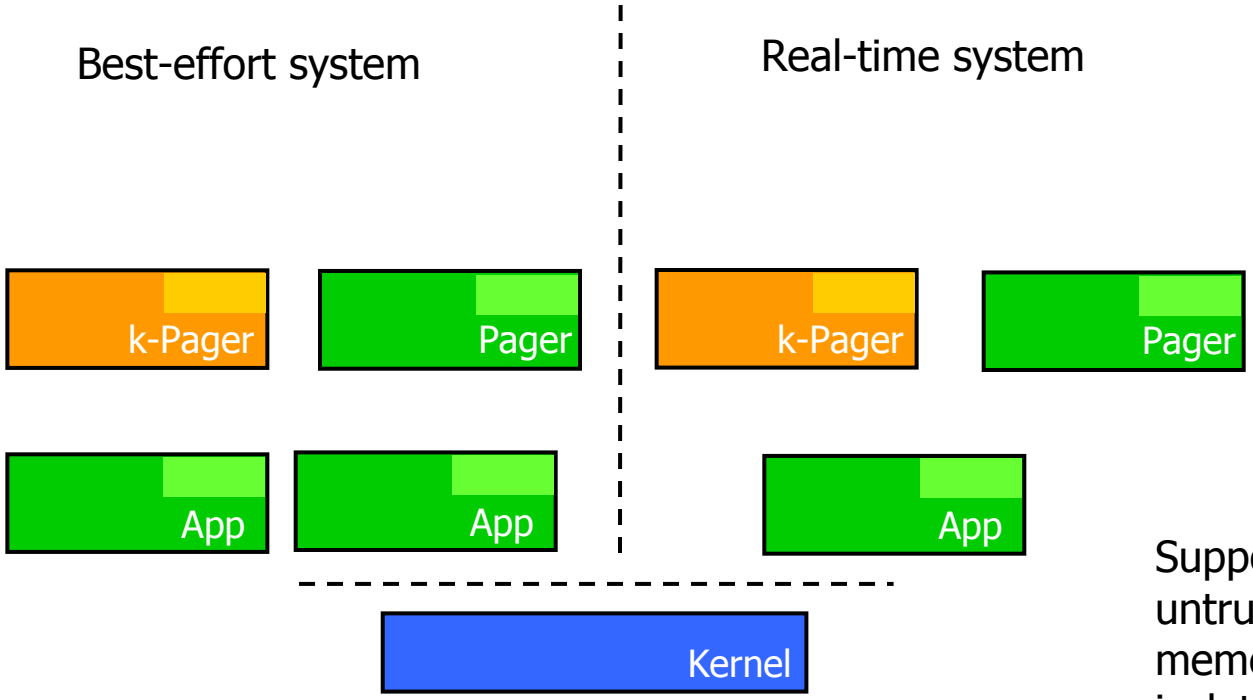
The Mechanism

- Kernel memory resources have user-visible names
- Tasks send page fault messages to request additional kernel memory
- Manager responds by mapping ordinary memory to the task
- Memory that is being used by the kernel is not accessible from user level
- All kernel memory can be preempted at any time
- Upon preemption, kernel objects are converted into an external representation
- Tasks generate faults for preempted objects; when mapped back, they are fully restored



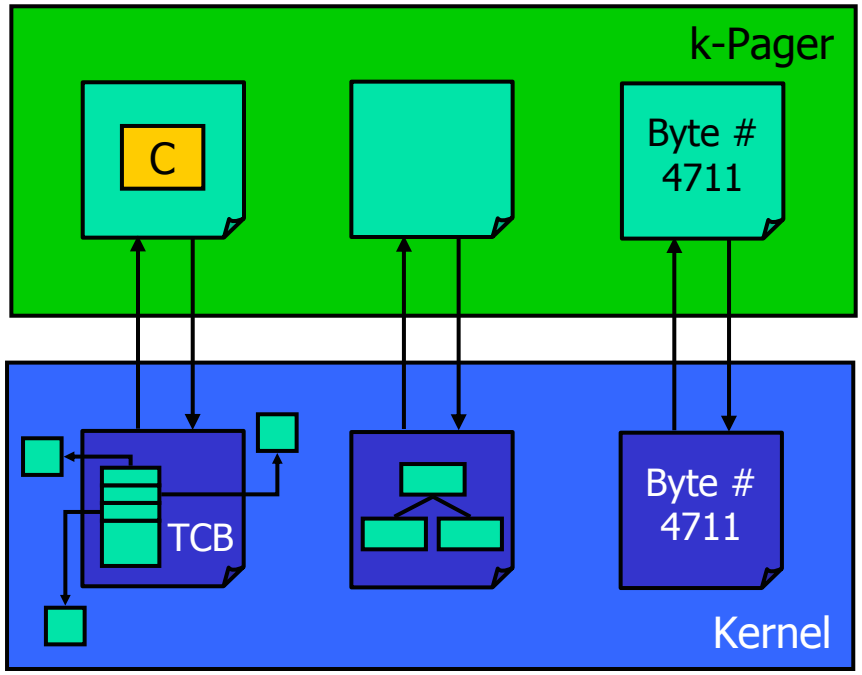


The Vision



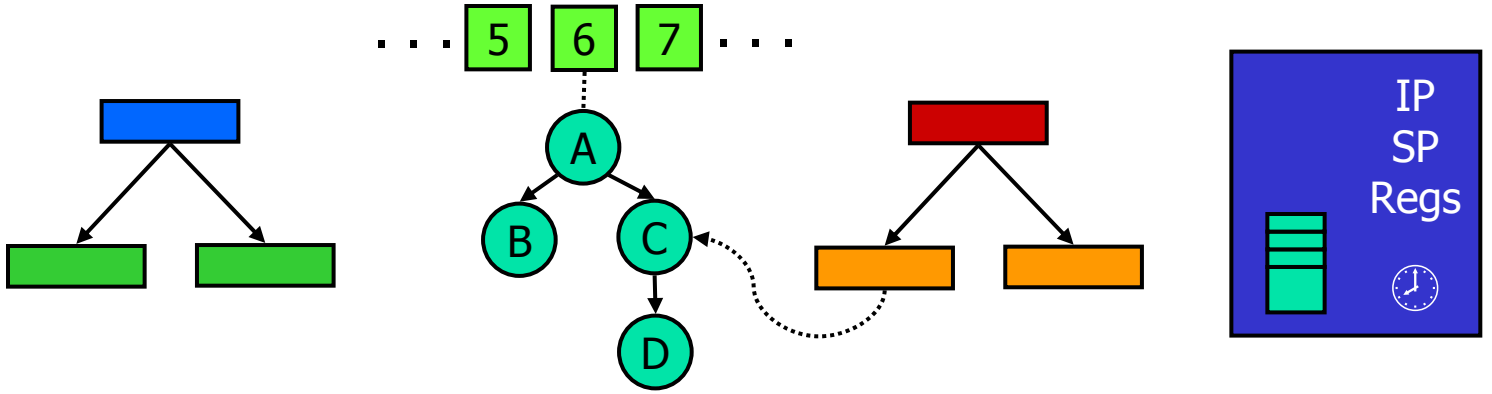
Support multiple concurrent untrusted managers of kernel memory while preserving isolation.

Maintaining Protection



- Before kernel data is exported to user level, it is converted into a safe external representation
- Safe: Pager cannot use it to gain additional privileges
- Three broad classes of kernel data:
  - Sensitive
  - Redundant
  - Safe

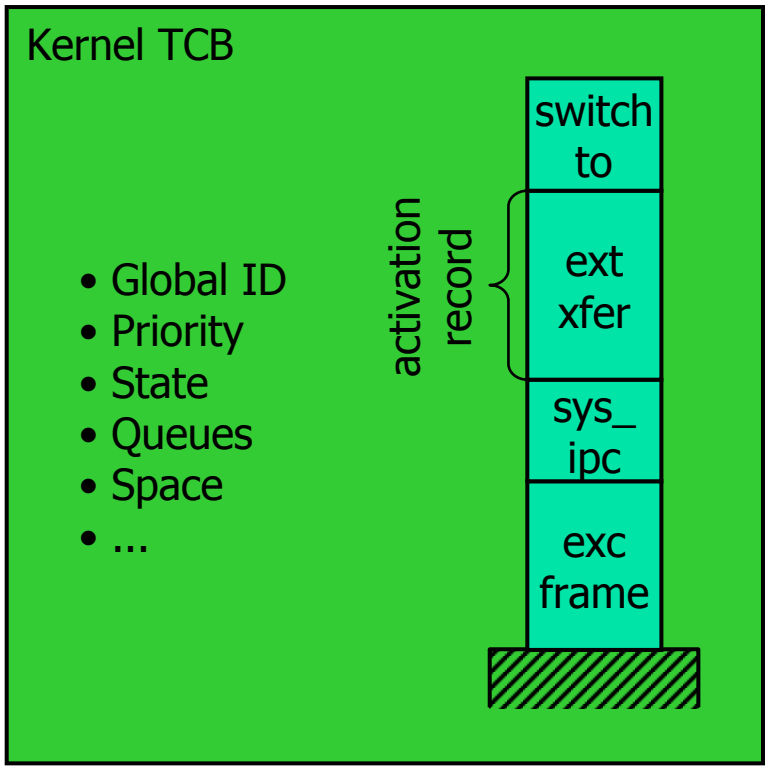
Experiment: L4



Page tables vaddr → paddr	**Mapping DB** Mem. alloc.	**Node tables** vaddr → mapnode	**TCBs** metadata
Redundant	(Redundant)	Sensitive	Sensitive/Safe
Discarded	Localized	Localized	Localized/Exported (~90%)

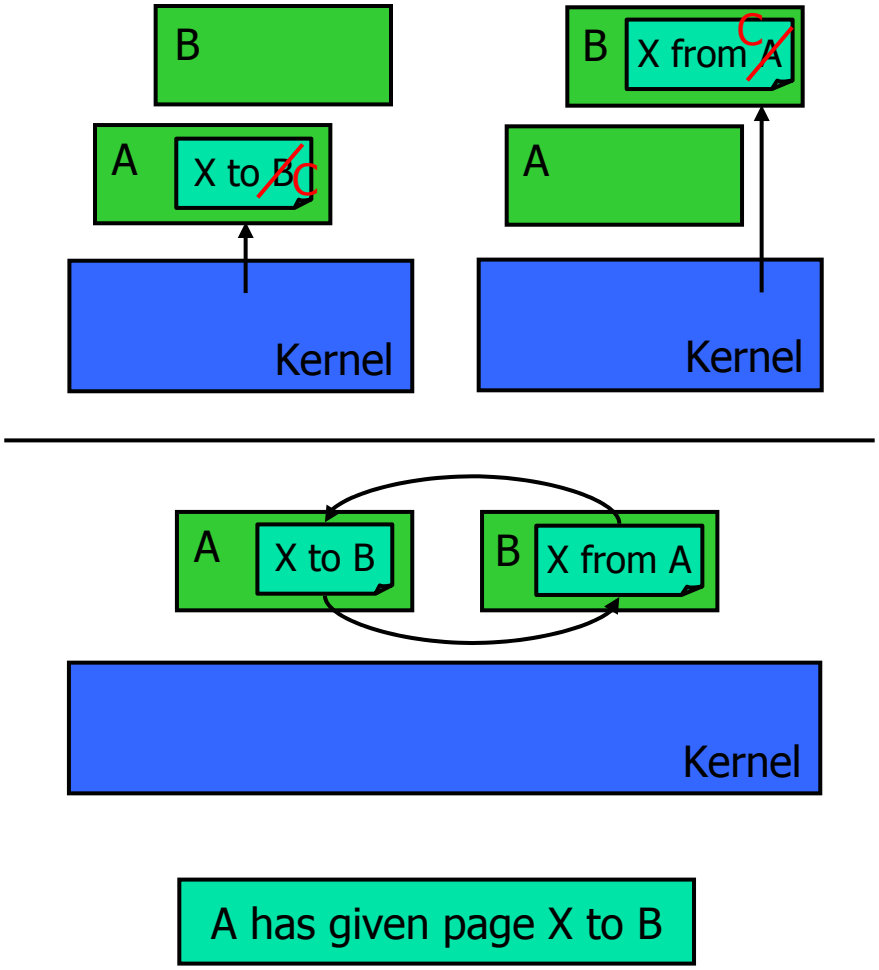
- Implementation in the L4 microkernel (IA32)
- User-level VM and threads, address spaces, IPC

Preemption: The kernel stack



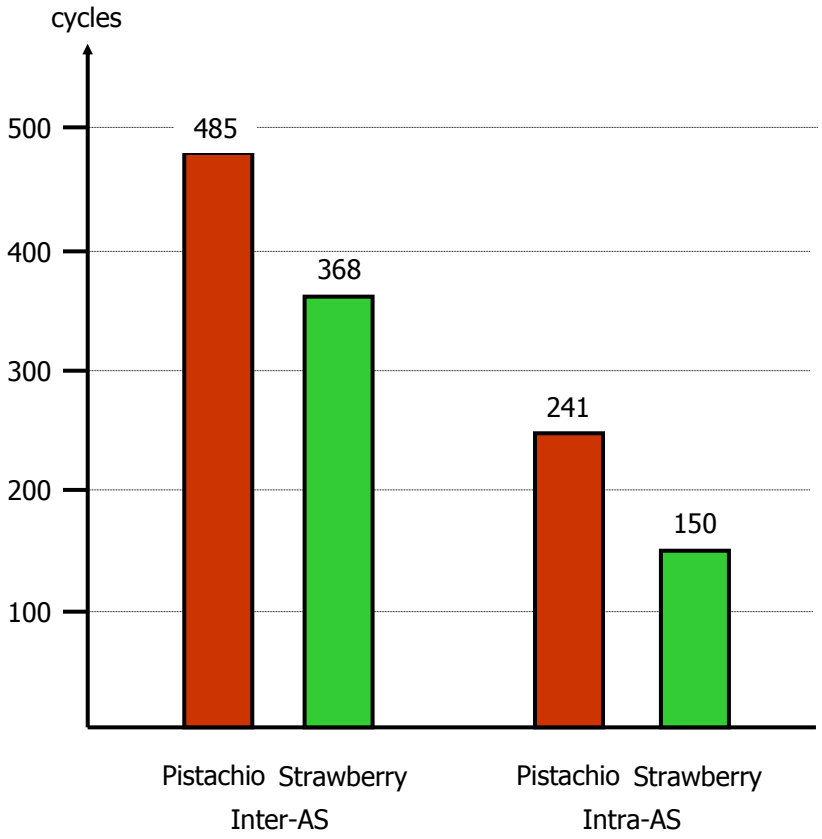
- Kernel stack can be preempted, but...
- ... it is extremely difficult to parse (spill variables)
- Idea: Use global stack, continuations
- Global stack can be preallocated
- Additional benefit: Smaller cache footprint

Example: Shared Kernel Data



- How to export kernel data that describes a relationship between multiple tasks?
- Re-import requires agreement of all participating tasks
- Solution: Split data between all tasks, add mutual references for efficient validation

Evaluation: Performance



X.2 Ping-pong Benchmark
Dual Pentium II/400
KDB disabled, no assertions

- Problem: Additional resource checks add overhead to IPC
- Comparison to L4/Pistachio, both kernels unoptimized (preliminary results)
- Good performance seems possible
- Other benchmarks are more interesting, e.g. preemption cost

Evaluation

Application	User	Kernel
init	76k	48k
bash	392k	52k
portmap	96k	48k
getty	80k	48k
inetd	100k	48k
smbd	260k	48k
emacs	2,700k	60k

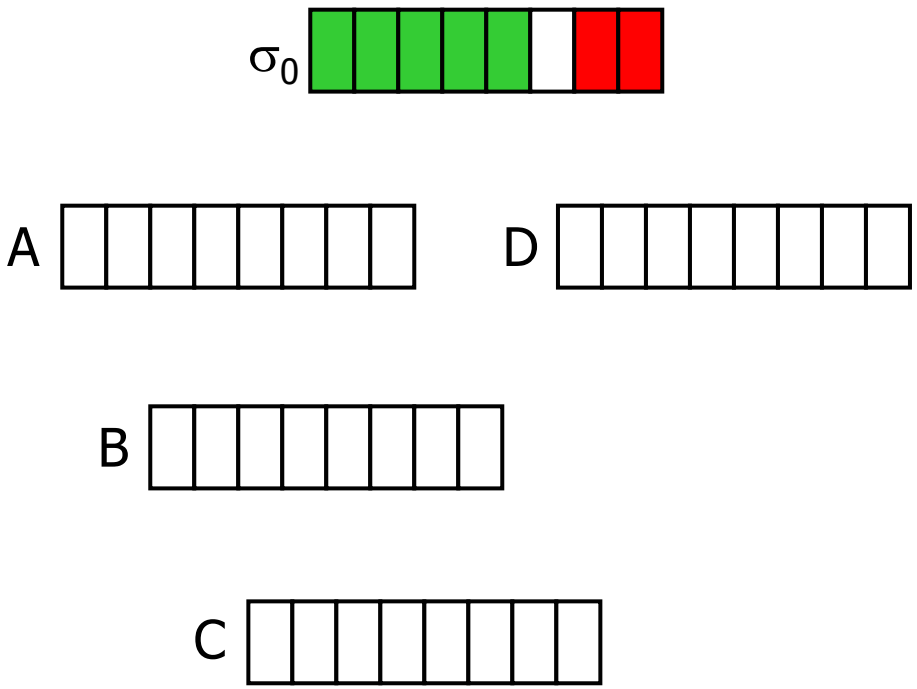
Allocated internally	1 fault	18,000 cyc
Requested from pager	3 faults	21,400 cyc

- Experiment 1: Kernel memory usage on an L4Linux system
- Result: Significant part of memory is used for kernel metadata
- Experiment 2: Cost for kernel PF handling, compared to in-kernel memory allocation
- Dual PII/400 system
- Result: ~4µs / fault

Conclusions

- Good kernel memory management is important for security and performance
- The proposed mechanism allows multiple concurrent user-level policies and supports preemption
- The mechanism requires additional page faults, but they are not expensive

Future Work: Persistence



- Side effect: System can easily be made persistent
- By unmapping all memory, σ_0 can obtain a snapshot which already is in ext. representation

Future Work

- Develop realistic policies and apply them to realistic scenarios.