

Experiences from Implementing Multiprocessor Support for an Industrial Operating System Kernel

Simon Kågström Håkan Grahn Lars Lundberg

Department of Systems and Software Engineering
Blekinge Institute of Technology
Ronneby, Sweden
<http://www.bth.se/tek>

IN PARTNERSHIP WITH THE

Knowledge Foundation



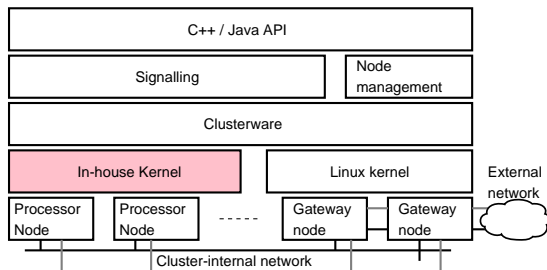
- 1 The Uniprocessor Kernel
 - Overview of the Operating System
 - Address Space Handling
- 2 The Multiprocessor Port
 - Overview of the Multiprocessor Port
 - Processor-local data
 - Multithreaded containers with CPU-local data
- 3 Experiences from the Implementation
 - Development Experiences
- 4 Conclusions



- 1 The Uniprocessor Kernel
 - Overview of the Operating System
 - Address Space Handling
- 2 The Multiprocessor Port
 - Overview of the Multiprocessor Port
 - Processor-local data
 - Multithreaded containers with CPU-local data
- 3 Experiences from the Implementation
 - Development Experiences
- 4 Conclusions



The In-House Operating System

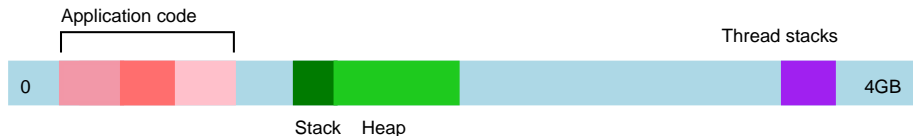


- Distributed, fault-tolerant cluster system
- Distributed RAM-resident database
- The system uses an asynchronous programming model centered around the database
- **Static** processes spawn **dynamic** processes to perform tasks



The In-house Operating System, II

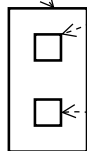
- The operating system is based around three basic entities:
 - **Containers**: address spaces
 - **Processes**: resource containers
 - **Threads**: execution context
- The programming model relies on fast processes:
 - Efficient address space handling, small memory requirements
 - Applications are always mapped in memory
 - No paging to disk



Address space handling

Container memory
data structure

Container page
directory pointer



Stack page
table entry

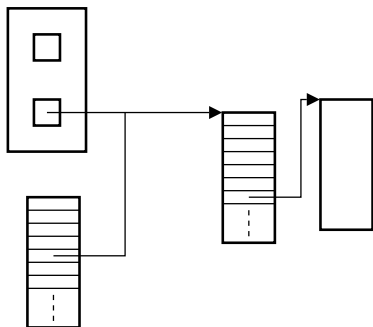


Global page directory

- The data structure has a pointer to a private page directory
- ... and a stack page table entry
- Containers start without a private address space



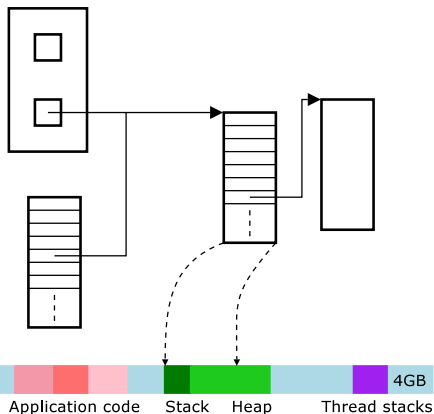
Address space handling



- The data structure has a pointer to a private page directory
- ... and a stack page table entry
- Containers start without a private address space
 - The stack page entry is copied to the global page directory

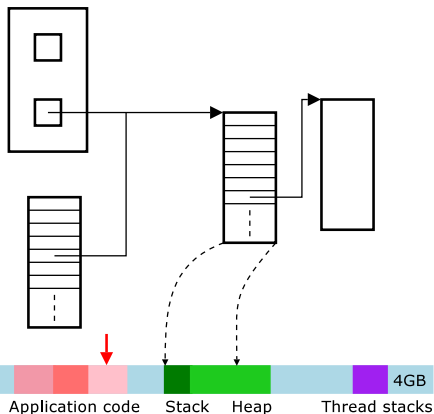


Address space handling



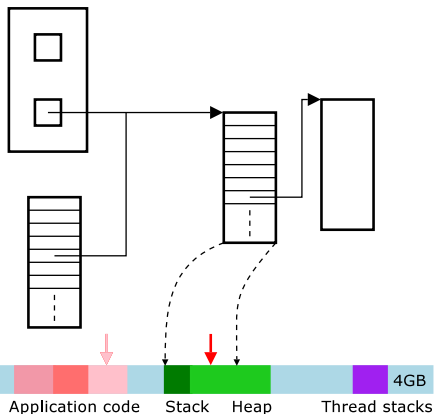
- The data structure has a pointer to a private page directory
- ... and a stack page table entry
- Containers start without a private address space
 - The stack page entry is copied to the global page directory
 - The page table covers the stack, global variables and part of the heap

Address space handling



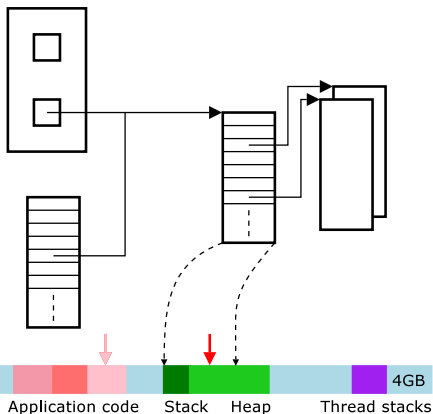
- The data structure has a pointer to a private page directory
- ... and a stack page table entry
- Containers start without a private address space
 - The stack page entry is copied to the global page directory
 - The page table covers the stack, global variables and part of the heap

Address space handling



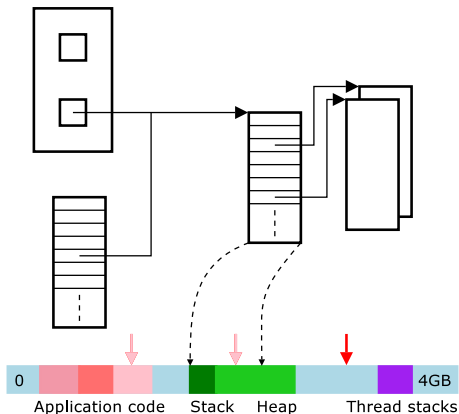
- The data structure has a pointer to a private page directory
- ... and a stack page table entry
- Containers start without a private address space
 - The stack page entry is copied to the global page directory
 - The page table covers the stack, global variables and part of the heap

Address space handling



- The data structure has a pointer to a private page directory
- ... and a stack page table entry
- Containers start without a private address space
 - The stack page entry is copied to the global page directory
 - The page table covers the stack, global variables and part of the heap

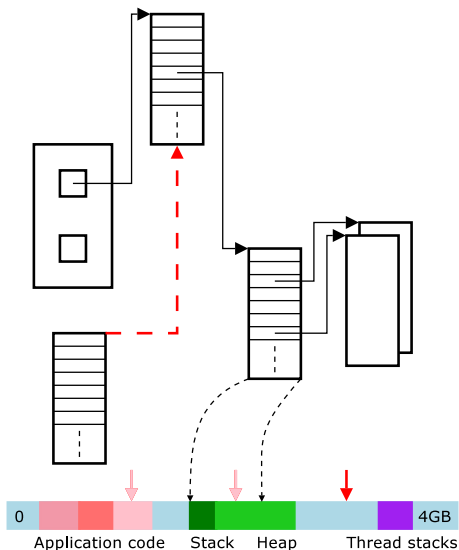
Address space handling



- The data structure has a pointer to a private page directory
- ... and a stack page table entry
- Containers start without a private address space
 - The stack page entry is copied to the global page directory
 - The page table covers the stack, global variables and part of the heap



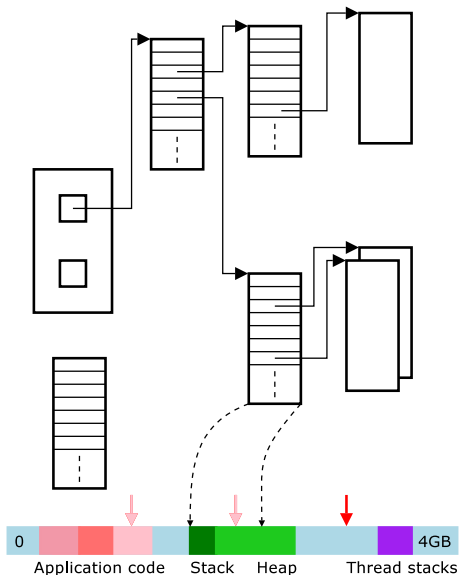
Address space handling



- The data structure has a pointer to a private page directory
- ... and a stack page table entry
- Containers start without a private address space
 - The stack page entry is copied to the global page directory
 - The page table covers the stack, global variables and part of the heap
- On access outside the page table, the global page directory is copied to a private



Address space handling



- The data structure has a pointer to a private page directory
- ... and a stack page table entry
- Containers start without a private address space
 - The stack page entry is copied to the global page directory
 - The page table covers the stack, global variables and part of the heap
- On access outside the page table, the global page directory is copied to a private



- 1 The Uniprocessor Kernel
 - Overview of the Operating System
 - Address Space Handling
- 2 The Multiprocessor Port
 - Overview of the Multiprocessor Port
 - Processor-local data
 - Multithreaded containers with CPU-local data
- 3 Experiences from the Implementation
 - Development Experiences
- 4 Conclusions

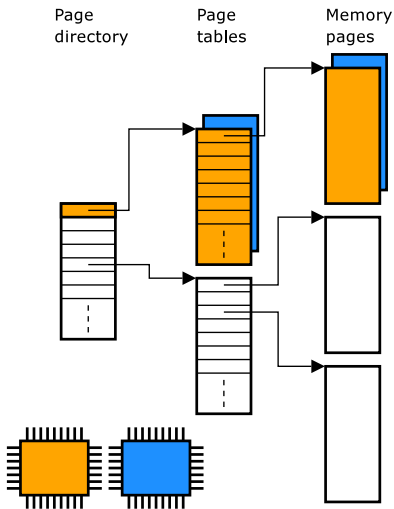


Overview of the Multiprocessor Port

- We have used a giant locking scheme for the port
 - A single lock protects the entire kernel
 - Similar to Linux 2.0 and early SMP versions of FreeBSD
- External interrupts are routed to one processor, timer interrupts are processor-local
- Portions of the code, e.g., floating point support, IA-32 segmentation and startup has been modified to support multiprocessors
- A set of data structures, (the current thread, kernel stack, etc.) have been made CPU-local
- The reason for the simple approach was to simplify the first port and provide a ground to improve the port later



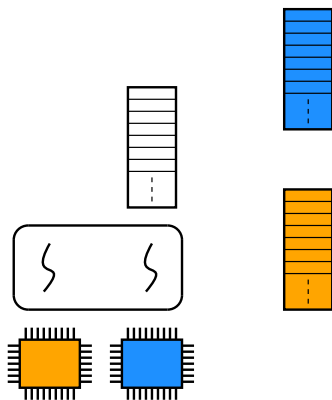
Processor-local data



- We reserve an area of virtual memory for CPU-local variables
- All CPU-local variables are placed in a special section of the ELF-file
 - An attribute specifier is added to declarations
- This allows CPU-local variables to be accessed exactly as before
- Some special structures, e.g., the kernel stack, need indirection

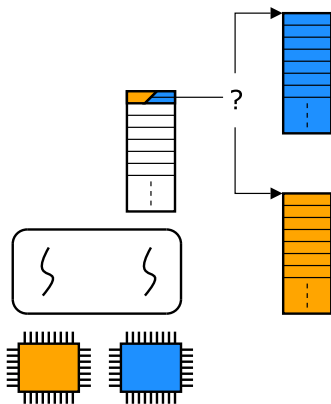


Multithreaded Containers



- CPU-local data is a problem with multithreaded containers

Multithreaded Containers



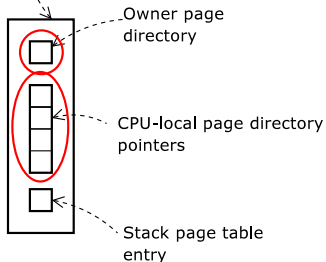
- CPU-local data is a problem with multithreaded containers
 - Two processors in one address space access the same CPU-locals

- CPU-local data is a problem with multithreaded containers
 - Two processors in one address space access the same CPU-locals
- Multithreaded processes are rare in the in-house operating system
 - The programming model promote starting new processes
 - Mainly found in Java applications



Multithreaded Containers, II

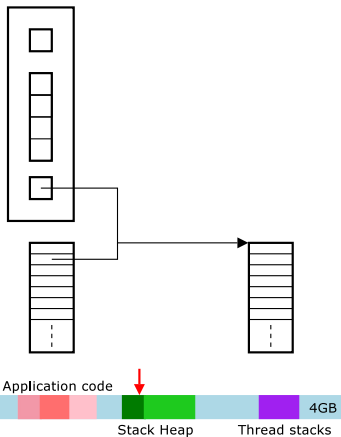
Container memory
data structure



- The container data structure has been extended
 - CPU-private page directory pointers
 - “Owner” page directory



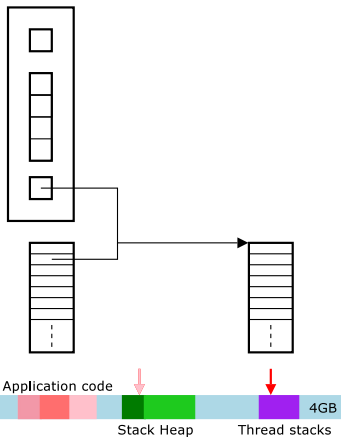
Multithreaded Containers, II



- The container data structure has been extended
 - CPU-private page directory pointers
 - “Owner” page directory
- Applications always start single threaded with the global page directory



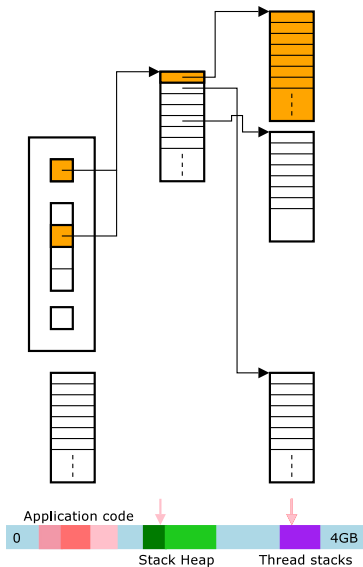
Multithreaded Containers, II



- The container data structure has been extended
 - CPU-private page directory pointers
 - “Owner” page directory
- Applications always start single threaded with the global page directory



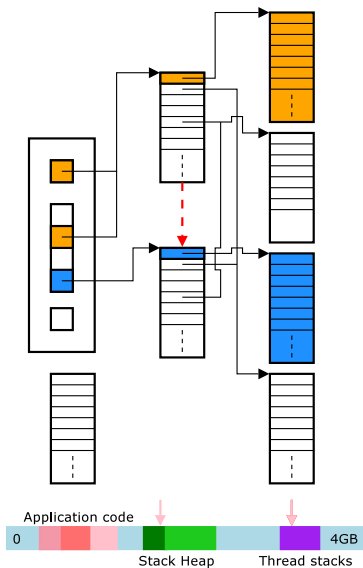
Multithreaded Containers, II



- The container data structure has been extended
 - CPU-private page directory pointers
 - “Owner” page directory
- Applications always start single threaded with the global page directory
- As the second thread starts, the executing CPU becomes owner of the address space



Multithreaded Containers, II



- The container data structure has been extended
 - CPU-private page directory pointers
 - “Owner” page directory
- Applications always start single threaded with the global page directory
- As the second thread starts, the executing CPU becomes owner of the address space
- Next CPU copies the owner address space and sets up a private page directory



- 1 The Uniprocessor Kernel
 - Overview of the Operating System
 - Address Space Handling
- 2 The Multiprocessor Port
 - Overview of the Multiprocessor Port
 - Processor-local data
 - Multithreaded containers with CPU-local data
- 3 Experiences from the Implementation
 - Development Experiences
- 4 Conclusions



Development experiences

- Performance is limited with the giant locking approach
 - We found that even with a simple user-space loop, the system spends around 36% of the time in-kernel
 - The maximum speedup is therefore limited to $\frac{100}{36 + \frac{64}{2}} \approx 1.47$ by Amdahl's law
 - We got a 20% improvement for the user-space loop
- The project took around two years to implement
 - One developer, part-time
- Specialized operating system, complex setup and configuration
- Working off-site, complicating interaction with the developers



- Large and complex code base of the in-house operating system
 - Around 2.5M lines of code totally
 - Around 160,000 lines were relevant for our purposes
- We ended up adding around 2,300 lines of code in new files
 - Processor startup
 - Locking implementation
- 1,600 lines of existing code was modified
 - Taking and releasing the giant lock
 - Implementation of multithreaded containers
- Only around 1% of the relevant code base has been modified



- 1 The Uniprocessor Kernel
 - Overview of the Operating System
 - Address Space Handling
- 2 The Multiprocessor Port
 - Overview of the Multiprocessor Port
 - Processor-local data
 - Multithreaded containers with CPU-local data
- 3 Experiences from the Implementation
 - Development Experiences
- 4 Conclusions



- We have presented the design and implementation of multiprocessor support for an industrial operating system kernel
- The operating system is a fault-tolerant cluster system
- The programming model places special demands on the kernel implementation
- We chose a giant locking approach for the first port
- We also present an efficient solution to the problem with multithreaded containers
- The performance of a giant locking solution is limited
- Although the modified code base is limited, the implementation took longer than we had expected



Questions?



- We did performance measurements on a 2-way Pentium II computer
- The applications on the in-house operating system are mainly system bound
 - Database transactions and network traffic
 - Background processes which do database replication, logging etc.
- We found that even with a simple user-space loop, the system spends around 36% of the time in-kernel
 - Therefore, the maximum speedup on our system is limited to $\frac{100}{36 + \frac{64}{2}} \approx 1.47$ by Amdahl's law
- The giant locking approach limits achievable performance
 - Around 20% of the time is spent on spinning on the giant lock on the multiprocessor
 - We got a 20% improvement over the uniprocessor for the user-space loop

