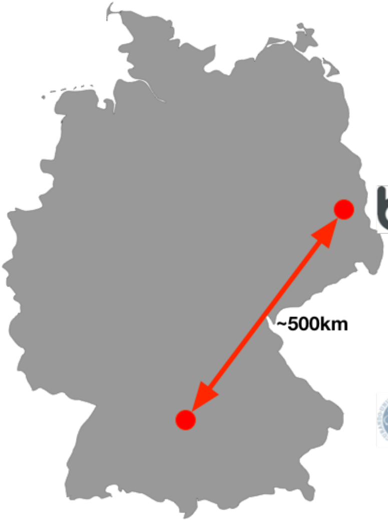
March 3, 2017

SCALABLE OPERATING SYSTEMS
FOR MANY THREADS ON MANY-CORES

Stefan Wesner, Lutz Schubert, Vladimir Nikolov,
Stefan Bonfert, Jörg Nolte, Robert Kuban, Randolf Rotta, . . .

---

[1] Research supported by German BMBF grant 01IH13003.



b-tu Brandenburgische Technische Universität Cottbus - Senftenberg

ulm university universität uulm

Cottbus: Verteilte Systeme & Betriebssysteme

- Jörg Nolte
- Randolf Rotta
- Robert Kuban
- Philipp Gypser
- ...

Ulm: Organisation und Management von Informationssystemen

- Stefan Wesner
- Lutz Schubert
- Vladimir Nikolov
- Stefan Bonfert
- ...

RESEARCH FIELDS

Cottbus

- operating systems, distributed & parallel middleware
- . . . for multi- and many-core processors:
  memory consistency, NVRAM, 100Gb/s End2End wireless
- . . . and for embedded wireless ad-hoc networks:
  self-stabilization, energy management, event driven processing

Ulm

- programming models for heterogeneous systems (embedded and HPC)
- scheduling in dynamic systems
- adaptive resource management
- real-time scheduling

OUTLINE

1. Our Research Team

2. **Project Background**

3. Many-Core Scalability Bottom Up

4. Summary

MYTHOS: MANY THREADS OPERATING SYSTEM

- BMBF HPC3 call
  „Anwendungsorientierte HPC-Software für skalierbare Parallelrechner“
- Oct. 2013 – Sep. 2016
- `https://github.com/ManyThreads/mythos`

Application Partners

- HLRS Stuttgart: Michael Resch - molecular dynamics
- University Siegen: Sabine Roller - fluid dynamics
- Alcatel Lucent: Peter Domschitz - distributed media processing

APPLICATION SCENARIOS & REQUIREMENTS

Scientific Computing

- single application owns all cores & memory
- user-level scheduling of application's tasks

⇒ fast thread suspend and wake-up instead of polling

⇒ huge number of threads manipulate shared address space

Media Processing Cloud

- isolated 3rd-party compute tasks
- supervisor configures tasks, communicates data, minimizes number of actives cores & processors

⇒ dynamic creation, migration and teardown of simple protection domains & threads

PROJECT GOALS/CONSTRAINTS

Target Hardware

- Intel XeonPhi Knights Corner:
  PCIe card, 60 cores, 240 threads, 8GiB memory
- use host CPU (Linux) for communication instead of virtualization
- exploit shared memory and caches

Minimal Base Kernel

- focus on threads and address spaces
- avoid in-kernel policies, heuristics, magic, . . .
- spatial scheduling: thread placement instead of time sharing
- no POSIX interfaces (yet)

CHALLENGES

Scalability Bottlenecks from Hidden Sharing

- e.g. memory pools, translation tables, queues, services. . .
- automatic partitioning is wasteful, complex, inflexible

⇒ enable explicit sharing, reward localized usage

Active Waiting wastes Time * Power

- sharing implies synchronization
- assume high latency, high contention

⇒ use waiting time for other tasks or sleep,
reward asynchronous / event-driven applications

OUTLINE

1. Our Research Team
2. Project Background
3. **Many-Core Scalability Bottom Up**
4. Summary

SHARING: FRIEND OR FOE?

Advantages of Sharing

- efficient use of memory (no replication)
- fast HW support for atomic updates
- flexible reconfiguration, migration

Disadvantages

- HW bottlenecks (directory contention, RFO latency)
- races, inconsistent concurrent operations
- danger of false sharing

⇒ trade-off based on application
MyThOS: mitigate disadvantages, reward localized usage

ASYNCHRONOUS SYNCHRONISATION VER. 1

Tasklets between hardware threads: fire&forget

- small messages: next ptr, function ptr, arguments
- each core has 2 queues: incoming and local tasklets
- if empty return to user mode or sleep; wakeup IPI on first push

Location-Based Serialization

- all objects assigned to specific HW threads
  send method calls as tasklet to *correct* queue

⇒ manual placement of responsibilities
fragile usage (e.g. concurrent deletion)

ASYNCHRONOUS SYNCHRONISATION VER. 2

Object Monitors [Mesa], Delegation Queue Locking [Guarded Sections]

Delegating Object Monitors

- method calls encoded in tasklets, supplied by caller
- object monitor serializes them
- delegation chooses responsible HW thread
- tasklet is returned via response method to caller’s object

Deferred Synchronous

- tasklets are statically allocated during object creation
- flow control: block next requests until internal tasklets have returned
- # of tasklets = # of logical control flows

EXAMPLE CALL

```
Adder* addServer = ...;
Tasklet myTask;
addServer->add(&myTask, this, 47, 11);

class Adder {
 Monitor monitor;
public:
 void add(Tasklet* t, AddResult* res, int a, int b) {
   monitor.request(t,
           [=](Tasklet* t){this->addImpl(t,res,a,b);} );
 }

private:
 void addImpl(Tasklet* t, AddResult* res, int a, int b) {
   res->result(t, a+b);
   monitor.requestDone();
 }
};
```

EXAMPLE CALL

```
Adder* addServer = ...;
Tasklet myTask;
addServer->add(&myTask, this, 47, 11);

class Adder {
 Monitor monitor;
public:
 void add(Tasklet* t, AddResult* res, int a, int b) {
   monitor.request(t,
           [=](Tasklet* t){this->addImpl(t,res,a,b);} );
 }

private:
 void addImpl(Tasklet* t, AddResult* res, int a, int b) {
   res->result(t, a+b);
   monitor.requestDone();
 }
};
```

EXAMPLE CALL

```
Adder* addServer = ...;
Tasklet myTask;
addServer->add(&myTask, this, 47, 11);

class Adder {
 Monitor monitor;
public:
 void add(Tasklet* t, AddResult* res, int a, int b) {
   monitor.request(t,
           [=](Tasklet* t){this->addImpl(t,res,a,b);} );
 }

private:
 void addImpl(Tasklet* t, AddResult* res, int a, int b) {
   res->result(t, a+b);
   monitor.requestDone();
 }
};
```

EXAMPLE CALL

```
Adder* addServer = ...;
Tasklet myTask;
addServer->add(&myTask, this, 47, 11);

class Adder {
 Monitor monitor;
public:
 void add(Tasklet* t, AddResult* res, int a, int b) {
   monitor.request(t,
           [=](Tasklet* t){this->addImpl(t,res,a,b);} );
 }

private:
 void addImpl(Tasklet* t, AddResult* res, int a, int b) {
   res->result(t, a+b);
   monitor.requestDone();
 }
};
```

EXAMPLE CALL

```
Adder* addServer = ...;
Tasklet myTask;
addServer->add(&myTask, this, 47, 11);

class Adder {
 Monitor monitor;
public:
 void add(Tasklet* t, AddResult* res, int a, int b) {
   monitor.request(t,
           [=](Tasklet* t){this->addImpl(t,res,a,b);} );
 }

private:
 void addImpl(Tasklet* t, AddResult* res, int a, int b) {
   res->result(t, a+b);
   monitor.requestDone();
 }
};
```

CONCURRENT OBJECT DELETION

Problem

- pending tasklets while user requests deletion

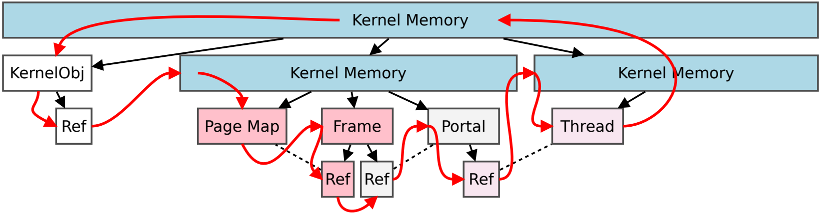
Solution

- reference counting delays removal until safe [Magenta]
- smart pointers enable forced removal of references, hierarchy of dependent pointers [like OC tree]
- fence multicast over all active HW threads resolves temporary references [like RCU gc]

EXPLICIT ALLOCATION AGAINST HIDDEN SHARING

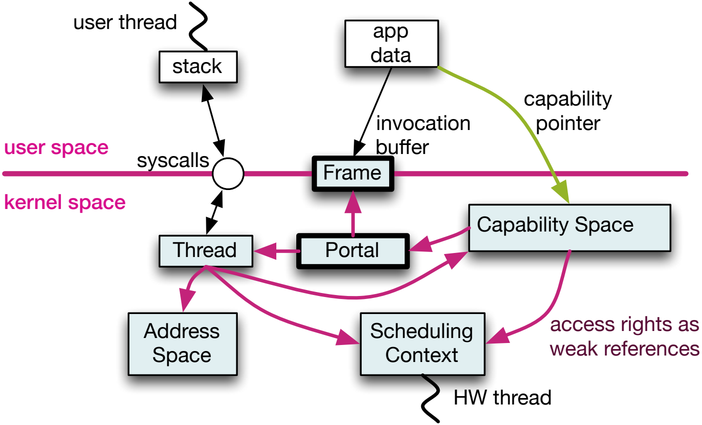
Kernel memory as first-class object [seL4]

- range of unused memory for new kernel objects, can be split ⇒ user can distribute load over cores
- MyThOS: atomic ops and locking just on neighbor nodes, local synchronization on partitioned memory, detect user’s race conditions

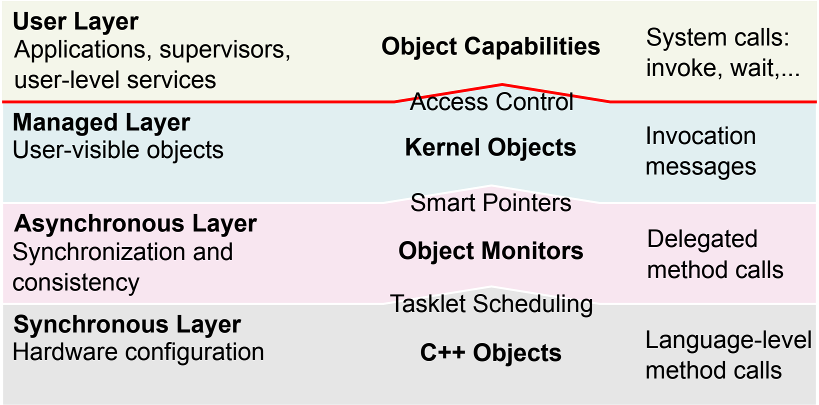


ASYNCHRONOUS CAPABILITY INVOCATIONS

Object Capabilities [seL4, Fiasco.OC], Portals separated from Threads



LAYERED ARCHITECTURE



User Layer Applications, supervisors, user-level services	**Object Capabilities**	System calls: invoke, wait,...
	Access Control	
Managed Layer User-visible objects	**Kernel Objects**	Invocation messages
	Smart Pointers	
Asynchronous Layer Synchronization and consistency	**Object Monitors**	Delegated method calls
	Tasklet Scheduling	
Synchronous Layer Hardware configuration	**C++ Objects**	Language-level method calls

SUMMARY

Challenge: eliminate hidden sharing, reduce active waiting

Solution: Delegation, Object Monitors, Explicit Kernel Memory

	on-demand allocation	explicit sharing
big kernel lock	?	seL4
fine-grained locking	Fiasco.OC, Linux	MyThOS
message passing	?	Barrelfish

Results

- chaotic recursive fork/join: doesn't crash (too often)
- core sleep too eager + high IPI latency ⇒ needs tuning