Multicore

2001: IBM POWER4, dual-core PowerPC
2006: Intel Core Duo, dual-core x86
2007: Tilera TILE64, 64 cores
2012: Kalray MPPA-256, 1 socket, 256 cores
2015: Intel Xeon E7
8 sockets × 18 cores/socket × 2 HW threads/core = 288 hardware threads (HTs) to be scheduled by OS!
2013: Oracle SPARC T5
8 sockets × 16 cores/socket × 8 HW threads/core = 1024 HTs!
Why?

» **power wall**: can’t increase frequency without chips running too hot / costing too much

» **memory wall**: RAM outpaced by processor speeds, cannot get data and instructions to processor quickly enough → **caches**

» **ILP (= instruction-level parallelism) wall**: can’t keep pipeline busy w/ single instruction stream

» These trends are unlikely to change in the foreseeable future.
Challenges

1. How to “find” and expose parallelism in applications?

2. How to efficiently schedule and load-balance that many cores/HTs?

3. How to synchronize efficiently across that many cores/HTs?

4. How to synchronize correctly?
Memory Hierarchy

» UMA: uniform memory architecture
» NUMA: non-uniform memory architecture
» cache consistency
» cache-line bouncing
» false sharing
» cache interference
Memory Consistency

» sequential consistency \( \approx \) serializability
  execution equiv. to some **sequential interleaving** of instr.

» relaxed memory models
  » reorder writes w.r.t. program order
  » reorder reads w.r.t. program order
  » reorder reads and writes
  » relaxed atomicity: some processors read some writes early

» memory **barrier / fence**: enforce program order
Scalability and Synchronization
Kernel Scalability Basics

» coarse-grained locking → fine-grained locking
» ensure data structures are cache-line-aligned
» minimize access to shared data structures
» use partitioned per-processor data structures
» maintain cache affinity
» cache partitioning / coloring
» employ efficient & scalable synchronization primitives...
Non-Scalable Ticket Spin Lock

```c
volatile unsigned int arrival_counter = 0, now_serving = 0;
void lock() {
    unsigned int ticket;
    ticket = atomic_fetch_and_inc(&arrival_counter);
    while (ticket != now_serving) {
        ; // do nothing — why is this not scalable?
    }
}
void unlock() {
    memory_barrier(); // when and why needed?
    now_serving++;
}
```
Scalable MCS Queue Lock


```c
struct qnode {
    volatile struct qnode* next;
    volatile bool blocked;
}

struct qnode* last = NULL;
```

> CAS — *compare-and-swap*: given a memory location, an *expected value*, and a new value, store the new value only if the expected value matches the actual value.
Scalable MCS Queue Lock — Lock Operation

```c
void lock(struct qnode* self) {
    struct qnode* prev;
    self->next = NULL;
    prev = atomic_fetch_and_store(&last, self);
    if (prev != NULL) {
        self->blocked = true;
        memory_barrier();
        prev->next = self;
        while (self->blocked)
            ; // do nothing — why is this scalable?
    } else memory_barrier();
}
```
Scalable MCS Queue Lock — Unlock Operation

```c
void unlock(struct qnode* self) {
    memory_barrier();
    if (self->next == NULL) {
        if (compare_and_swap(&last, self, NULL))
            return; // CAS returns true if stored
        else
            while (self->next == NULL)
                ; // do nothing
    }
    self->next->blocked = false;
}
```
Read–Copy Update (RCU)

- Problem with reader–writer locks: every read side critical section requires two writes (to the lock itself)!
- RCU: make (very frequent) reads extremely cheap, at the expense of (infrequent) writers.
- Idea: use execution history to synchronize.
- Shared pointer to current version of shared object; dereferenced exactly once by each reader.
- Instead of updating in place, writer makes a copy, updates the copy, publishes the copy by exchanging current-version pointer, and then (later) garbage–collects the old version.
Simple RCU Implementation

Processor in *quiescent* state: not using RCU-protected resource.

*Grace period*: every processor is guaranteed to have been in quiescent state at least once.

→ *garbage-collect after grace period ends*

> readers: execute non-preemptively

> writer: grace period ends after every processor has context-switched at least once

> multiple writers: serialize with spin lock
Non-Blocking Synchronization

Idea: synchronize, but without mutual exclusion.

» Design data structures to allow *safe concurrent access*.

» No waiting, no possibility of deadlock.

» **Wait-free**: process is guaranteed to progress in bounded number of steps, no matter what.

» **Lock-free**: if two or more processes conflict, at least one is guaranteed to progress; the other(s) may have to *retry*.

» **Obstruction-free**: progress is guaranteed only in absence of contention; all conflicting processes may have to retry.
Example: Wait-free Bounded Buffer

```c
char buffer[BUF_SIZE]; int head = 0; int tail = 0;
```

**Assumption:** one producer, one consumer.
Example: Wait-free Bounded Buffer

```c
char buffer[BUF_SIZE]; int head = 0; int tail = 0;
bool TryProduce(char item) {
    if ((tail + 1) % BUF_SIZE == head)
        return false;  // buffer full
    else {
        buffer[tail] = item;
        tail = (tail + 1) % BUF_SIZE;
        return true;
    }
}
```
Example: Wait-free Bounded Buffer

```c
bool TryConsume(char *item) {
    if (tail == head)
        return false; // buffer empty
    else {
        *item = buffer[head];
        head = (head + 1) % BUF_SIZE;
        return true;
    }
}
```
Example: Lock-free Queue

```c
struct QElem {
    struct Item *item;
    struct QElem *next;
}
struct QElem *last = NULL;
```

Assumption: any number of threads.

- `ldl` — *load-linked*, load a value from memory and start monitoring location that was read
- `stc` — *store-conditional*, store a value to a monitored location, but only if it hasn’t been written since `ldl`
Example: Lock-free Queue

```c
struct QElem {
    struct Item *item;
    struct QElem *next;
}

struct QElem *last = NULL;

void AppendToTail(struct Item *item) {
    struct QElem *new = malloc(sizeof(QElem));
    new->item = item;
    do {
        new->next = ldl(&last);
    } while(!stc(&last, new));
}
```
Example: Lock-free Queue

```c
struct QElem *last = NULL;

bool ItemIsInList(Item *item) {
    struct QElem *current = last;
    while (current != NULL) {
        if (current->item == item)
            return true;
        current = current->next;
    }
    return false;
}
```
Example: Lock-free Queue

```c
struct QElem *RemoveTail() {
  do {
    struct QElem *current = ldl(&last);
    if (current == NULL)
      return NULL;
  } while (!stc(&last, current->next));
  return current;
}
```
Universal Lock-free Object

```c
struct any_object { ... };
struct any_object *current_version;
void do_update() {
  ???
}
```

Like RCU: make a private copy, update copy, then publish with CAS.
Universal Lock-free Object

```c
struct any_object { ... };
struct any_object *current_version;

void do_update() {
    struct any_object *cpy = alloc_object();
    do {
        struct any_object *old = current_version;
        memcpy(cpy, old, sizeof(*old));
        cpy->some_field = ... // perform update on copy
    } while (!CAS(&current_version, old, cpy));
}
```
The ABA Problem

» When is it safe to reclaim or reuse old object?

» ABA problem: cas succeeds despite interleaved updates if expected value happens to be restored

» “same value” (cas) vs. “no writes” (ldl/stc)

» limited solution: tag bits / version counter
  (→ cas2, “double cas”)

» general solution: limited concurrent GC
  (→ e.g., hazard pointers)
OS Design for Multicore
Multikernels

**Idea:** a multicore kernel without shared memory.

Motivation:

- cache coherency can be a *scalability limit*: how many cores/sockets can you keep coherent without slowing down the entire system?
- core specialization will increase *hardware heterogeneity*: fast cores, slow cores, I/O cores, GPUs, integer cores…
- platform *diversity*: run on everything from smartphones to supercomputers; difficult to optimize for any platform
Multikernel Design Principles

1. Make all inter-core communication *explicit*.

2. Make OS structure hardware-neutral.  
   *(On top of a shallow HW-specific layer.)*

3. View state as *replicated* instead of shared.
Multikernel Design (Bauman et al., 2009)

Abstract

Commodity computer systems contain more and more processor cores and exhibit increasingly diverse architectural tradeoffs, including memory hierarchies, interconnects, instruction sets and variants, and IO configurations. Previous high-performance computing systems have scaled in specific cases, but the dynamic nature of modern client and server workloads, coupled with the impossibility of statically optimizing an OS for all workloads and hardware variants pose serious challenges for operating system structures.

We argue that the challenge of future multicore hardware is best met by embracing the networked nature of the machine, rethinking OS architecture using ideas from distributed systems. We investigate a new OS structure, the multikernel, that treats the machine as a network of independent cores, assumes no inter-core sharing at the lowest level, and moves traditional OS functionality to a distributed system of processes that communicate via message-passing.

We have implemented a multikernel OS to show that the approach is promising, and we describe how traditional scalability problems for operating systems (such as memory management) can be effectively recast using messages and can exploit insights from distributed systems and networking. An evaluation of our prototype on multicore systems shows that, even on present-day machines, the performance of a multikernel is comparable with a conventional OS, and can scale better to support future hardware.

1 Introduction

Computer hardware is changing and diversifying faster than system software. A diverse mix of cores, caches, interconnect links, IO devices and accelerators, combined with increasing core counts, leads to substantial scalability and correctness challenges for OS designers.

Such hardware, while in some regards similar to earlier parallel systems, is new in the general-purpose computing domain. We increasingly find multicore systems in a variety of environments ranging from personal computing platforms to data centers, with workloads that are less predictable, and often more OS-intensive, than traditional high-performance computing applications. It is no longer acceptable (or useful) to tune a general-purpose OS design for a particular hardware model: the deployed hardware varies wildly, and optimizations become obsolete after a few years when new hardware arrives.

Moreover, these optimizations involve tradeoffs specific to hardware parameters such as the cache hierarchy, the memory consistency model, and relative costs of local and remote cache access, and so are not portable between different hardware types. Often, they are not even applicable to future generations of the same architecture. Typically, because of these difficulties, a scalability problem must affect a substantial group of users before it will receive developer attention.

We attribute these engineering difficulties to the basic structure of a shared-memory kernel with data structures protected by locks, and in this paper we argue for rethinking the structure of the OS as a distributed system of functional units communicating via explicit messages.
Barrelfish (Bauman et al., 2009)

Barrelfish is the multikernel research OS that popularized the idea.
Monitors in Barrelfish

» keep track of OS state (memory allocation tables, capabilities/access rights, etc.)

» each monitor has a local copy: local operations are extremely fast

» Global operations are synchronized explicitly among all monitors with agreement protocols

» adopt techniques from distributed systems

» e.g., two-phase commit
Message Passing in Barrelfish

- **in general**: common interface for efficient hardware-specific implementations
  - e.g., use network on chip (Noc) in manycore chips (from Tilera, Adapteva, Kalray...)

- **on Intel/AMD x86**: use cache-coherent shared memory as message channel
  - carefully work with cache-coherency protocol

- two cache-coherency interactions per message
  - receiver *monitors* last word of expected message
  - sender invalidates when starting to write cache line
  - receiver fetches when message complete
Multikernel Discussion

Advantages:

» scales by default and transparently handles heterogeneity
» cross-core interaction is explicit and hence easier to debug
» can pick & choose kernels for specific workloads (hard real-time, soft real-time, max. throughput, etc.)

Challenges:

» it scales, but distributed agreement comes with overheads
» is it easier or more difficult to develop?
» is cache-coherence really a limiting factor?