

Supporting Parallelism in
Operating Systems & Programming Languages

Memory Management

Kornilios Kourtis
<kornilios.kourtis@inf.ethz.ch>

Memory ALLOCATIOn recap

`malloc()` for multi-processors

Virtual Memory

Stack Management

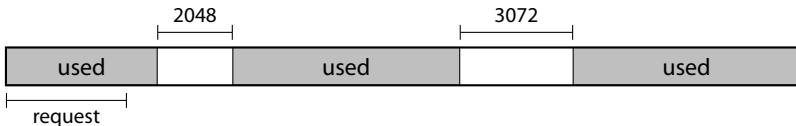
NUMA

Allocation

- ▶ managing space
- ▶ Interface:
 - ▶ `ptr = malloc(size)`
 - ▶ `free(ptr)`
- ▶ Concerns:
 - ▶ response time
 - ▶ space utilization (fragmentation)
- ▶ Allocator needs to track
 - ▶ in-use space
 - ▶ free space

Fragmentation

- ▶ External fragmentation
 - ▶ not enough contiguous space for allocation
- ▶ Internal fragmentation
 - ▶ space wasted in allocated vs requested size



Sequential fits

(linear list of free blocks)

Policies:

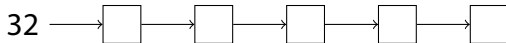
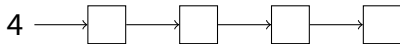
- ▶ first, best, next, worst

Mechanisms

- ▶ header field for each block
(a lot of memory bugs due to user corrupting block header)
- ▶ boundary tags (check for coalescing)
- ▶ pointers in each free block
(e.g., for implementing free list)

Segregated free lists

► free-list per size (class)



Segregated free lists approaches

- ▶ Simple segregated storage
 - ▶ if free list is empty:
 - ▶ allocate memory (superblock)
 - ▶ create new blocks from superblock
 - ▶ release superblock when all blocks are unused
- ▶ Segregated fits
 - ▶ if free list is empty: try larger class
 - ▶ release: try to coalesce

Segregated free lists variants

- ▶ buddy allocator:
 - ▶ classes of 2^x size
 - ▶ best-fit
 - ▶ splitting: a 2^x block is split into two 2^{x-1} sized (buddy) blocks
 - ▶ coalescing: when a block is freed
 - ▶ find “buddy” block (easily using address manipulation)
 - ▶ if it's free, merge them
- ▶ slab allocator:
 - ▶ user specifies:
 - ▶ size classes
 - ▶ constructor/destructors
 - ▶ slab: contiguous space for storing objects

Linux

- ▶ zoned buddy allocator
- ▶ SLAB/SLUB/SLOB allocators

slabtop:

OBJ	ACTIVE	USE	OBJ	SIZE	SLABS	OBJ/SLAB	CACHE	SIZE	NAME
615136	615117	99%	0.85K	153784	4	615136K	ext4_inode_cache		
641440	630788	98%	0.19K	32072	20	128288K	dentry		
682354	674415	98%	0.10K	18442	37	73768K	buffer_head		
58576	58095	99%	0.55K	8368	7	33472K	radix_tree_node		
47872	32510	67%	0.17K	2176	22	8704K	vm_area_struct		
18260	11352	62%	0.19K	913	20	3652K	filp		
5824	5046	86%	0.54K	832	7	3328K	inode_cache		
4920	4427	89%	0.60K	820	6	3280K	proc_inode_cache		
2824	2821	99%	0.93K	706	4	2824K	btrfs_inode_cache		
2568	2246	87%	1.00K	642	4	2568K	size-1024		

Memory ALLOCATIOn recap

malloc() for multi-processors

Virtual Memory

Stack Management

NUMA

The Hoard allocator

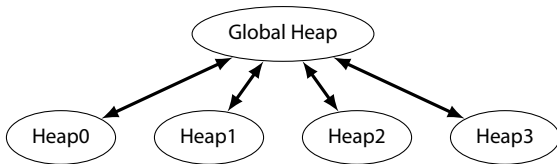
[Berger et al. 2000]

Features for a multithreaded memory allocator:

- ▶ speed
- ▶ scalability
- ▶ false sharing
- ▶ low fragmentation

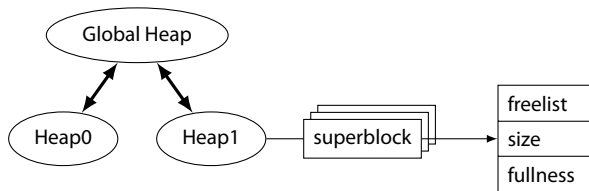
The Hoard allocator

[Berger et al. 2000]



- ▶ global heap and per-processor heap
- ▶ stats:
 - ▶ u_i : memory used
 - ▶ a_i : memory allocated from the OS
- ▶ local heap is found by hasing tid

The Hoard allocator: superblocks



- ▶ each heap maintains a set of *superblocks*
 - ▶ contiguous memory space
 - ▶ all of size S (page aligned)
 - ▶ each superblock contains blocks of same size
 - ▶ each superblock maintains a LIFO free list (why LIFO?)
- ▶ kept within fullness groups

The Hoard allocator: Bounding blowup

- ▶ return SBs to global heap
- ▶ conditions (parametrized by f, K)
 - ▶ heap not more than f empty ($u_i < (1 - f) \cdot a_i$)
 - ▶ has more than K 's worth of free memory ($a_i - u_i > K \cdot S$)
- ▶ note: we might return a non-empty SB on the global heap

The Hoard allocator: allocation

malloc()

1. for objects larger than $S/2$ use OS facilities (e.g., `mmap()`)
2. find a superblock
 - 2.1 check for a suitable SB in the local heap
 - 2.2 if none, check for a suitable SB from global heap
 - 2.3 if none, allocate SB from OS
3. allocate block from superblock
4. update stats
5. return block from superblock
 - ▶ suitable: empty or of the same size class

The Hoard allocator: release `free()`

1. for objects larger than $S/2$ use OS facilities (e.g., `mmap()`)
2. find the superblock of the object
3. find the owning heap of the superblock
4. deallocate block from superblock
5. update stats
6. transfer a superblock to the global heap if needed

information required:

- ▶ block → SB (stored block header)
- ▶ SB → owning Heap

TCMalloc

(google's Thread Caching Malloc)

- ▶ spans: run of contiguous pages
 - ▶ possibly different sizes
- ▶ mapping from pages to corresponding span
 - ▶ allows to find the span of an object (useful in `free()`)
 - ▶ in 32 bit using global array, in 64-bit using radix tree
- ▶ a per-thread cache with per-size lists
- ▶ a global cache with per-size lists
- ▶ global page heap
- ▶ small object ($\leq 32K$)
- ▶ large objects

TCMalloc: allocation/release

- ▶ large objects: rounded to page size
 - ▶ a global page heap
- ▶ small objects (per-size lists, 170)
 1. one cache per thread: no locking on the fast path
 2. if empty, try global free lists
 3. if empty, global page heap
- ▶ Deallocation
 - ▶ find span → determine if small or large
 - ▶ small → return it to the cache
 - ▶ large → return it to the global page heap
 - ▶ try to coalesce it with spans of neighbour pages
 - ▶ GC free lists after a limit (2MB)

jemalloc

(by Jason Evans/used by freebsd,facebook)

- ▶ chunks (4MiB) – OS interaction unit
- ▶ small/large objects
 - ▶ can locate corresponding chunk using pointers
 - ▶ live in page runs
- ▶ huge objects
 - ▶ directly use chunks
 - ▶ metadata in rb tree
- ▶ arenas
 - ▶ allocated to threads in a RR fashion
 - ▶ maintain their own chunks (arena chunks)
 - ▶ carve page-runs out of their chunks
- ▶ thread caches

Memory ALLOCATIOn recap

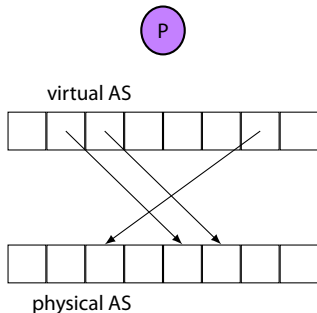
`malloc()` for multi-processors

Virtual Memory

Stack Management

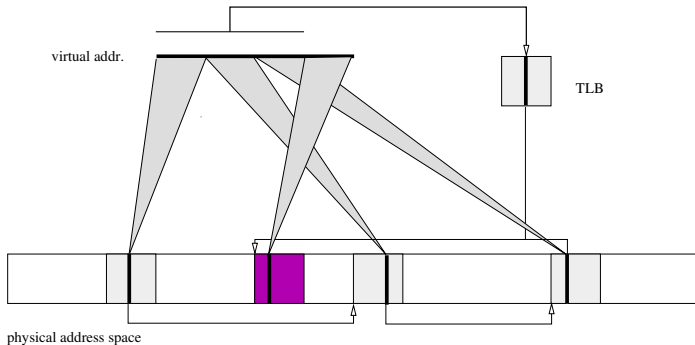
NUMA

VM recap: paging



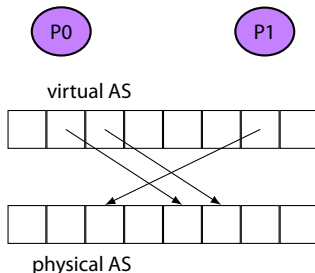
- ▶ originally to facilitate larger VAS than PAS
- ▶ page mappings maintained in page tables
 - ▶ include permissions

VM recap: address translation



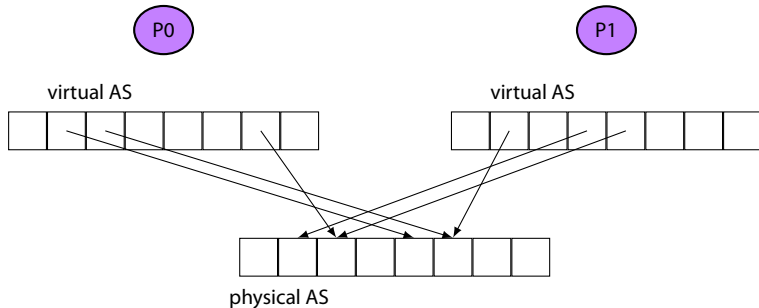
- ▶ **page tables**
 - ▶ map virtual addresses to physical
 - ▶ maintained by the OS
 - ▶ traversed by the hardware
- ▶ **TLB**

Sharing memory: address space sharing



- ▶ Sharing all the page tables
- ▶ Thread-local storage (TLS)
 - ▶ `__thread` in gcc
 - ▶ in x86 usually implemented using segment registers
- ▶ synchronization when updating mappings
- ▶ TLB shoot-down

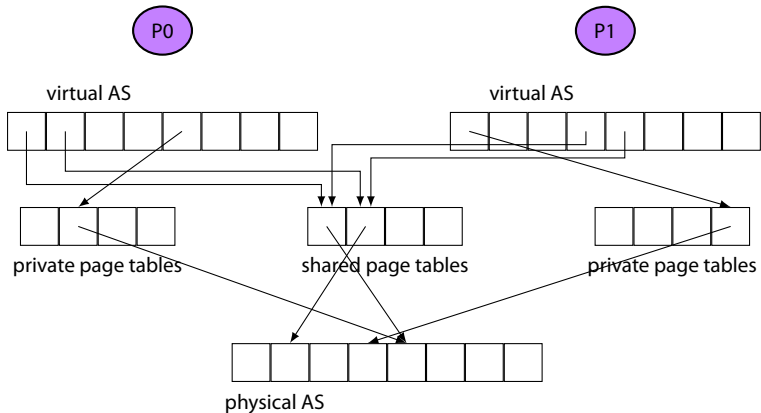
Sharing memory: separate address spaces



- ▶ `shm_open()`, `mmap()`
- ▶ duplicate mappings

Corey: partially shared page-tables

[Boyd-Wickizer et al. '08]



Memory ALLOCATIOn recap

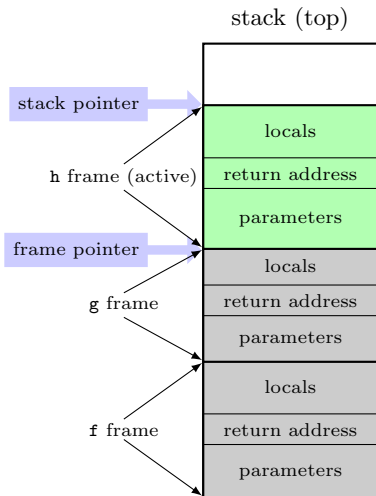
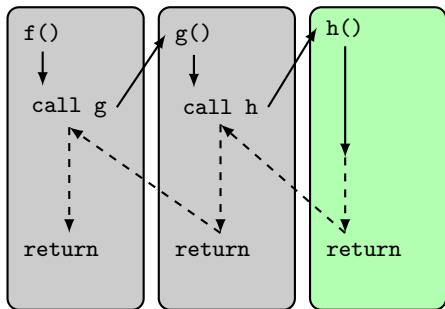
malloc() for multi-processors

Virtual Memory

Stack Management

NUMA

Traditional execution stack



Execution stack for single-threaded applications

- ▶ linear
- ▶ managed by the OS
 - ▶ on page faults, OS automatically allocates pages (up to a point)
 - ▶ minimum size: page size
 - ▶ cannot reclaim space from the stack

Execution stack for parallel applications

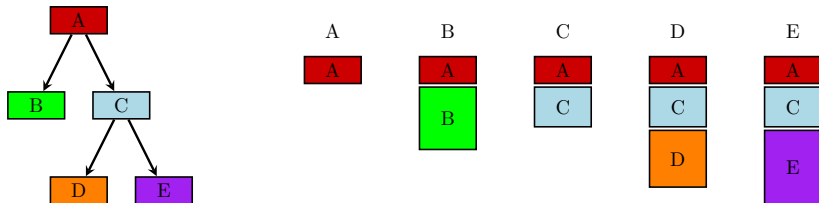
pthread (NPTL):

- ▶ each thread has its own stack
- ▶ allocated statically (default: 2MB)

Parallel tasks:

- ▶ each needs its own stack
- ▶ static stack:
 - ▶ should be large enough to cover worst-case scenario
 - ▶ wasteful for programs with parallel slack
- ▶ one OS thread per parallel task: highly inefficient

Cilk: cactus stack



- ▶ also, spaghetti stack
- ▶ initially developed for control mechanisms incompatible with linear stack
 - ▶ e.g., continuations

gcc's split stacks

- ▶ used by gccgo
- ▶ solution to static stack issues: split stacks
- ▶ allow for stacks to grow (or shrink)
- ▶ gcc insert checks to expand the stack as needed
- ▶ `-fsplit-stack` gcc switch
- ▶ note: concurrency vs parallelism
(e.g. see: Concurrency is not Parallelism (it's better) – Rob Pike)

Memory ALLOCATIOn recap

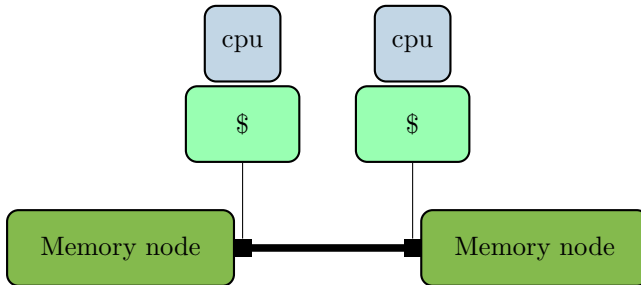
malloc() for multi-processors

Virtual Memory

Stack Management

NUMA

NUMA



- ▶ (numa) node balancing
- ▶ favour local access

Linux NUMA

- ▶ Policies:
 - ▶ local: always allocate on the local node
 - ▶ interleave: allocate from nodes in a round-robin fashion
 - ▶ user-defined

- ▶ libnuma
- ▶ numactl

NUMA page migration

- ▶ scheduler tries not to move processes into different nodes
- ▶ but it might happen

page migration

- ▶ migrate pages automatically
- ▶ no need to migrate a page that is not accessed
 - ▶ lazy migration
 - ▶ maintain access statistics

Bibliography

- ▶ Dynamic Storage Allocation: A survey and Critical Review
Paul R. Wislon et al.
1995
- ▶ The Slab Allocator: An Object-Caching Kernel Memory Allocator
Jeff Bonwick
USTC'94
- ▶ Hoard: A Scalable Memory Allocator for Multithreaded Applications
Berger et al.
ASPLOS 2000
- ▶ TCMalloc : Thread-Caching Malloc ([html](#))
Sanjay Ghemawat

Bibliography

- ▶ Scalable memory allocation using jemalloc ([html](#))
Jason Evans
- ▶ Corey: an operating system for many cores
Boyd-Wickizer et al.
OSDI '08
- ▶ The thorny problem of the cactus stack ([html](#))
Matteo Frigo