Performance
Engineering of
Software Systems

LECTURE 13

Parallel Storage Allocation

Saman Amarasinghe

October 27, 2022

Summary from Tuesday

	Manual	Reference Counting	Mark and Sweep	Stop and Copy
Ease of Use	Bad	Medium	Good	Good
Throughput	Good	Medium	Medium	Bad
Latency	Good	Good	Bad	Bad
External Fragmentation	Bad	Bad	Bad	Good
Example	C `malloc/free`	C++ `std::shared_ptr`	Java	C#

REVIEW OF MEMORY-ALLOCATION PRIMITIVES

Heap Storage in C

- **Allocation**

  `void* malloc(size_t s);`

  Effect: Allocate and return a pointer to a block of memory containing at least `s` bytes.

- **Deallocation**

  `void free(void *p);`

  Effect: `p` is a pointer to a block of memory returned by `malloc()` or `memalign()`. Deallocate the block.

- **Aligned allocation**

  `void* memalign(size_t a, size_t s);`

  Effect: Allocate and return a pointer to a block of memory containing at least `s` bytes, aligned to a multiple of `a`, where `a` must be an exact power of `2`:

  `assert(0==((size_t) memalign(a, s))%a)` .

Allocating Virtual Memory

The `mmap()` system call can be used to allocate virtual memory by memory mapping:

```
void *p = mmap(0,                           // Don't care where
               size,                        // #bytes
               PROT_READ | PROT_WRITE,      // Read/write
               MAP_PRIVATE | MAP_ANON,      // Private anonymous
               -1,                          // no backing file
               0                            // offset (N/A)
);
```

The Linux kernel finds a contiguous, unused region in the address space of the application large enough to hold `size` bytes, modifies the page table, and creates the necessary virtual-memory management structures within the OS to make the user's accesses to this area "legal" so that accesses won't segfault.

Properties of `mmap()`

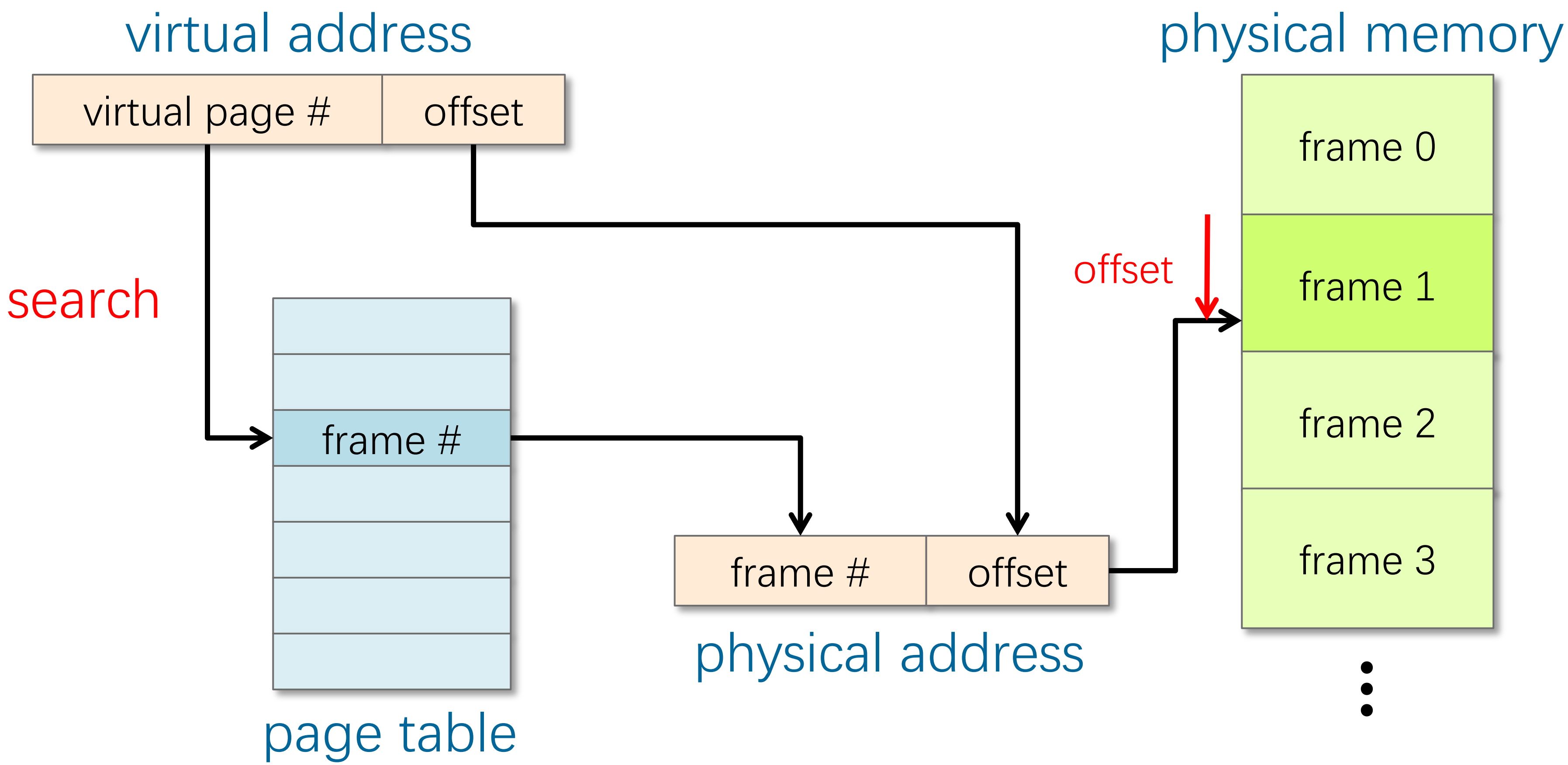
- `mmap()` is lazy. It does not immediately allocate physical memory for the requested allocation.
- Instead, it populates the page table with entries pointing to a special zero page and marks the page as read only.
- The first write into such a page causes a page fault.
- At that point, the OS allocates a physical page, modifies the page table, and restarts the instruction.
- You can `mmap()` a terabyte of virtual memory on a machine with only a gigabyte of DRAM.
- A process may die from running out of physical memory well after the `mmap()` call.

What’s the Difference…

…between `malloc()` and `mmap()` used in this way?

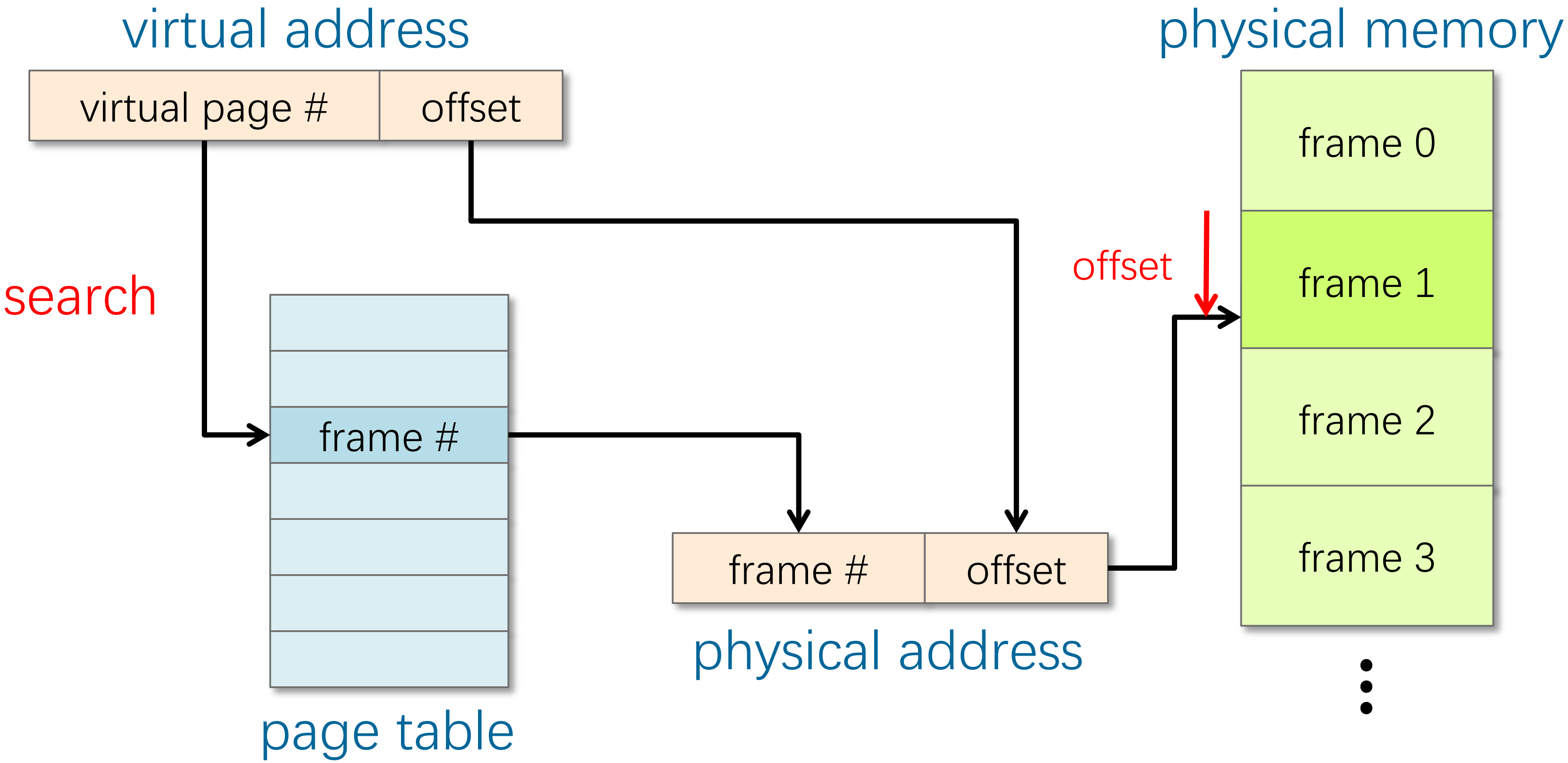
- The functions `malloc()` and `free()` are part of the memory-allocation interface of the heap-management code in the C library.
- The heap-management code uses available system facilities, including `mmap()`, to obtain memory (virtual address space) from the kernel.
- The heap-management code within `malloc()` attempts to satisfy user requests for heap storage by reusing freed memory whenever possible.
- When necessary, the `malloc()` implementation invokes `mmap()` and other system calls to expand the size of the user’s heap storage.

Address Translation

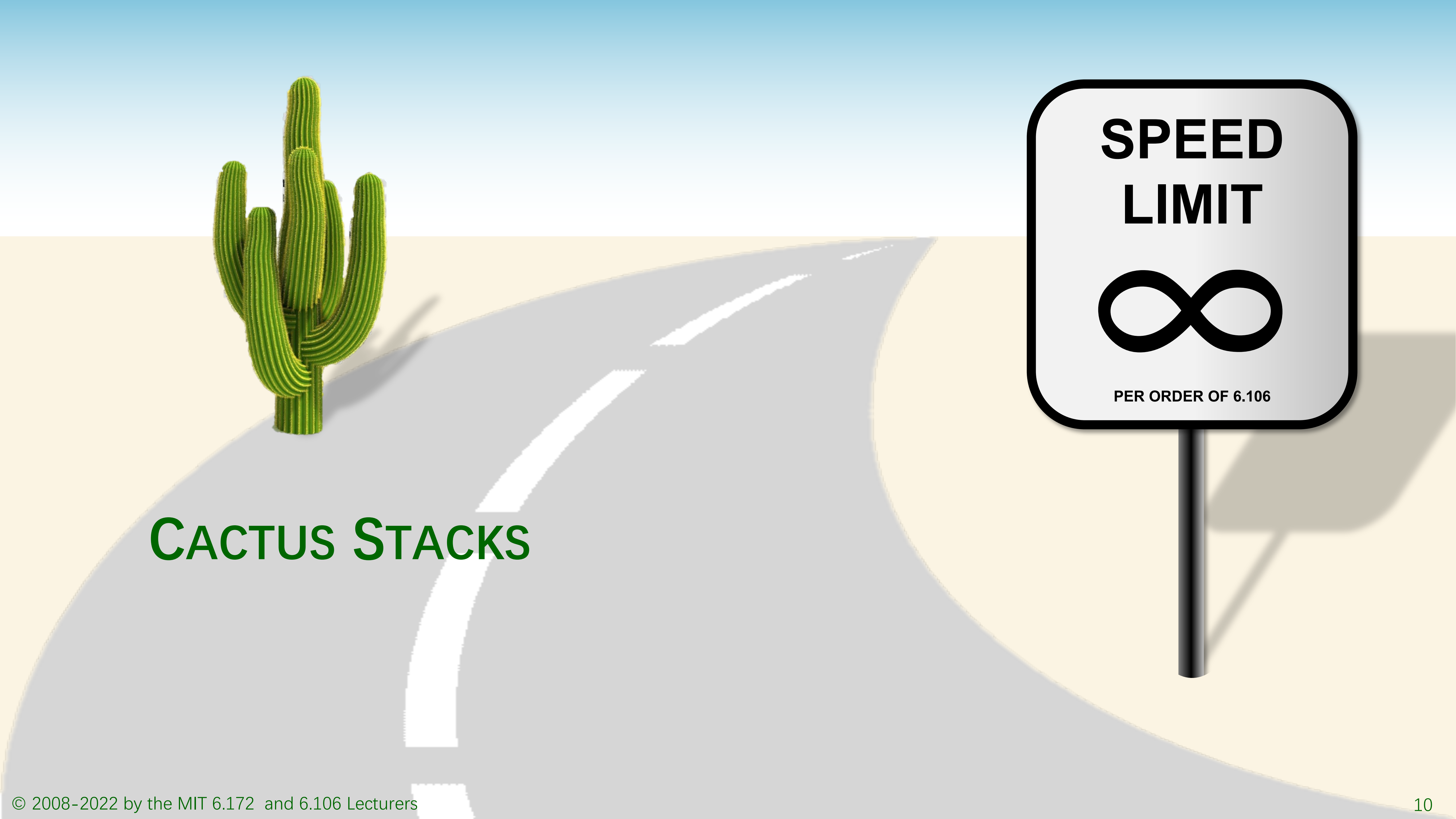


If the virtual page does not reside in physical memory, a page fault occurs.

Address Translation



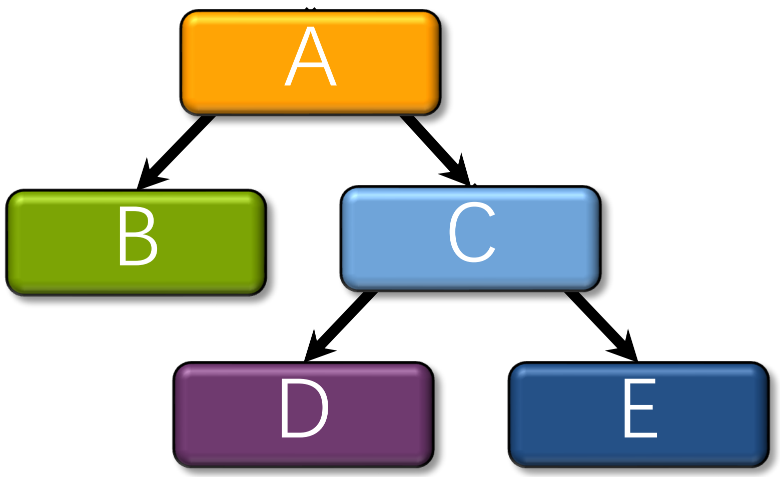
Since page-table lookups are costly, the hardware contains a translation lookaside buffer (TLB) to cache recent page-table lookups.



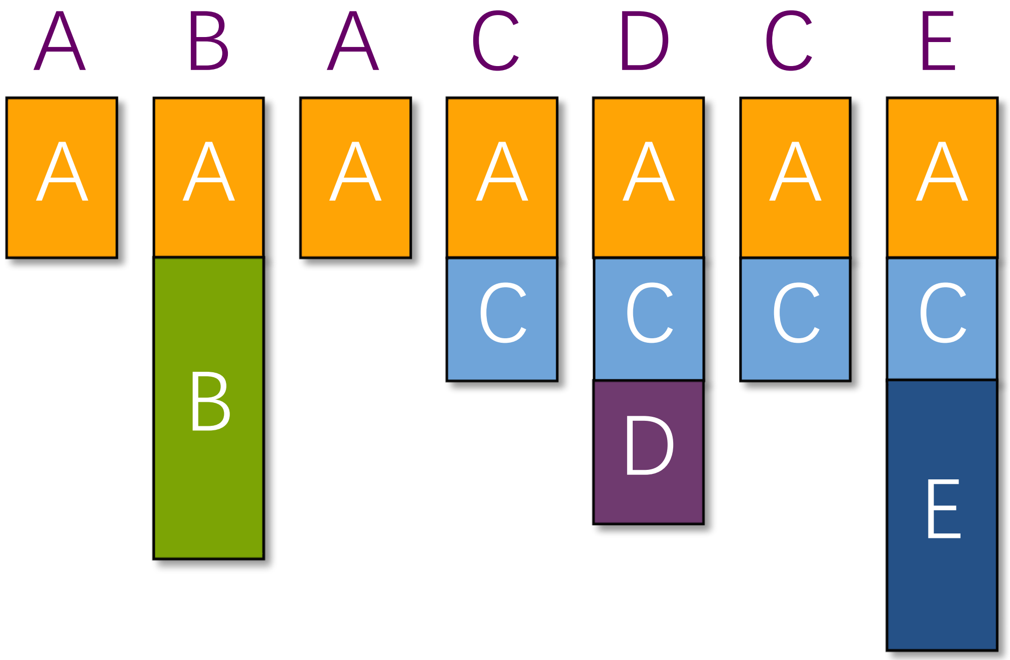
CACTUS STACKS

Traditional Linear Stack

An execution of a serial C/C++ program can be viewed as a serial walk of an invocation tree.



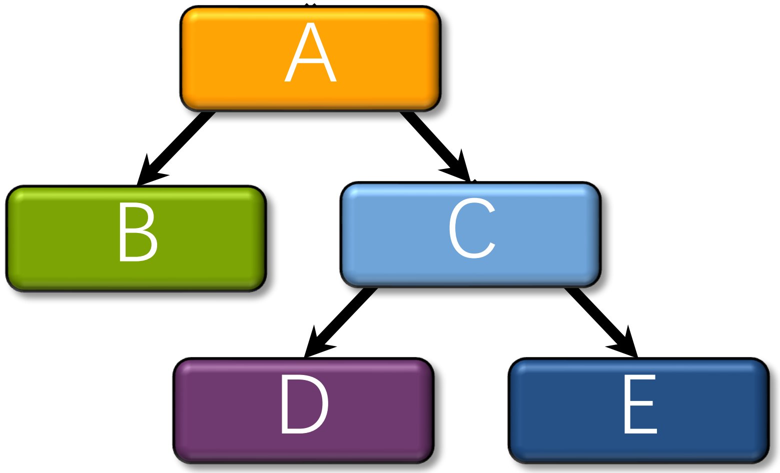
invocation tree



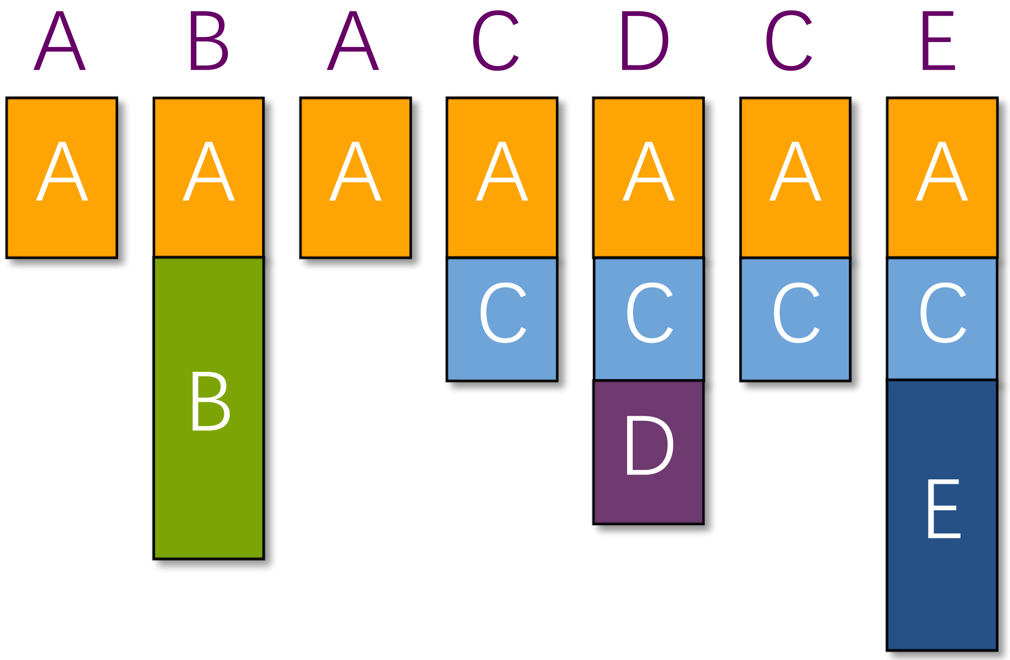
views of stack

Traditional Linear Stack

Rule for pointers: A parent can pass pointers to its stack variables down to its children, but not the other way around.



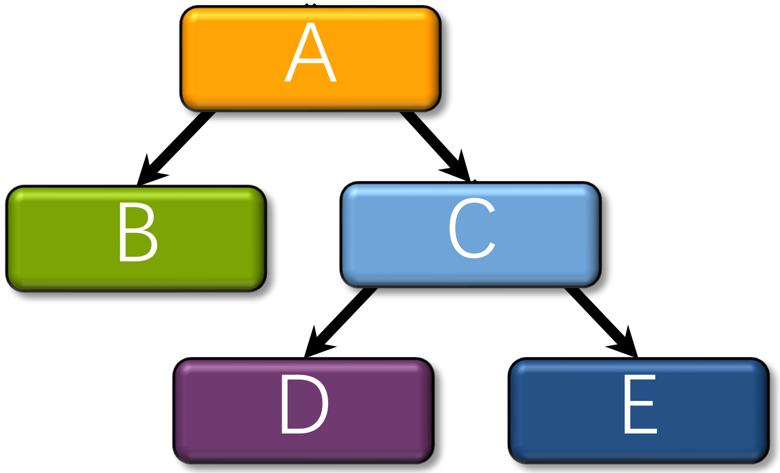
invocation tree



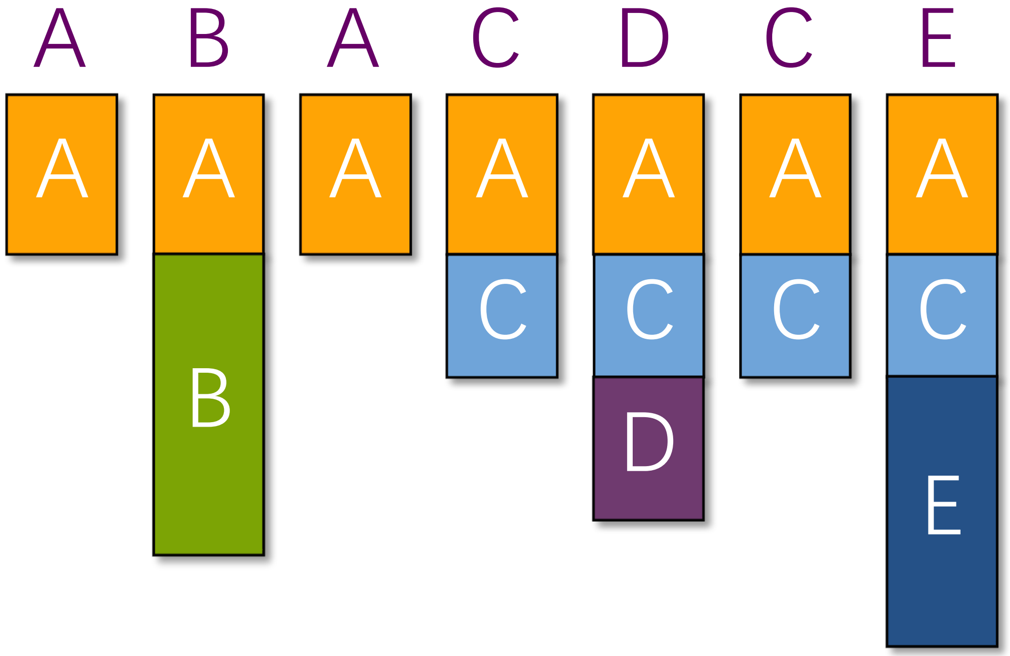
views of stack

Cactus Stack

A cactus stack supports multiple views in parallel.



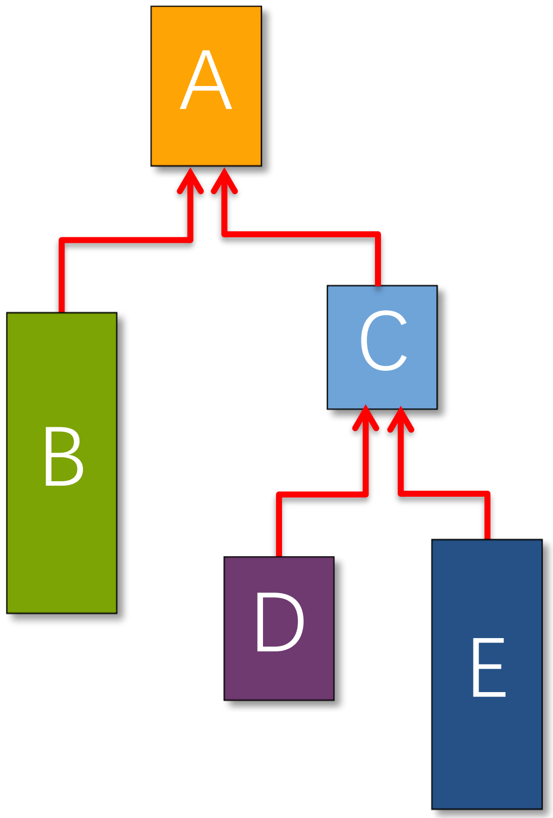
invocation tree



views of stack

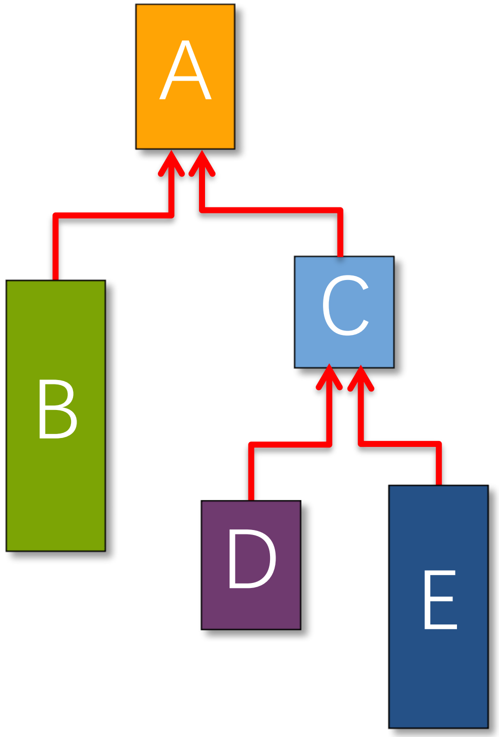
Heap-Based Cactus Stack

A heap-based cactus stack allocates frames off the heap.



Interoperability

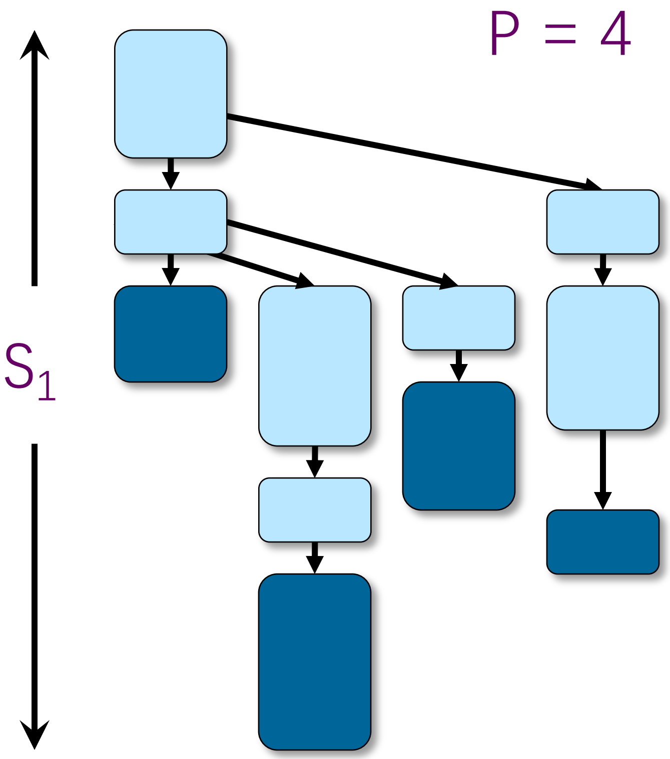
Problem: With heap-based linkage, parallel functions fail to interoperate with legacy and third-party serial binaries. Our implementation of Cilk uses a less space-efficient strategy that preserves interoperability by using a pool of linear stacks.



Space Bound

Theorem. Let S_1 be the stack space required by a serial execution of a Cilk program. The stack space of a P-worker execution using a heap-based cactus stack is at most $S_P \leqslant P\,S_1$.

Proof. Cilk's work-stealing algorithm maintains the busy-leaves property: Every active leaf frame has a worker executing it. ■



D&C Matrix Multiplication

```
void mm_dac(double *restrict C, int n_C,
            double *restrict A, int n_A,
            double *restrict B, int n_B,
            int n)
{ // C = A * B
  assert((n & (-n)) == n);
  if (n <= THRESHOLD) {
    mm_base(C, n_C, A, n_A, B, n_B, n);
  } else {
    double *D = malloc(n * n * sizeof(*D));
    assert(D != NULL);
    #define n_D n
    #define X(M,r,c) (M + (r*(n_ ## M) + c)*(n/2))
    cilk_spawn mm_dac(X(C,0,0), n_C, X(A,0,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,0,1), n_C, X(A,0,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,0), n_C, X(A,1,0), n_A, X(B,0,0), n_B, n/2);
    cilk_spawn mm_dac(X(C,1,1), n_C, X(A,1,0), n_A, X(B,0,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,0), n_D, X(A,0,1), n_A, X(B,1,0), n_B, n/2);
    cilk_spawn mm_dac(X(D,0,1), n_D, X(A,0,1), n_A, X(B,1,1), n_B, n/2);
    cilk_spawn mm_dac(X(D,1,0), n_D, X(A,1,1), n_A, X(B,1,0), n_B, n/2);
               mm_dac(X(D,1,1), n_D, X(A,1,1), n_A, X(B,1,1), n_B, n/2);
    cilk_sync;
    m_add(C, n_C, D, n_D, n);
    free(D);
  }
}
```

Allocations of the temporary matrix D obey a stack discipline.

Analysis of D&C Matrix Mult.

Work: $T_1(n) = \Theta(n^3)$

Span: $T_\infty(n) = \Theta(\lg^2 n)$

Space: $S_1(n) = S_1(n/2) + \Theta(n^2)$

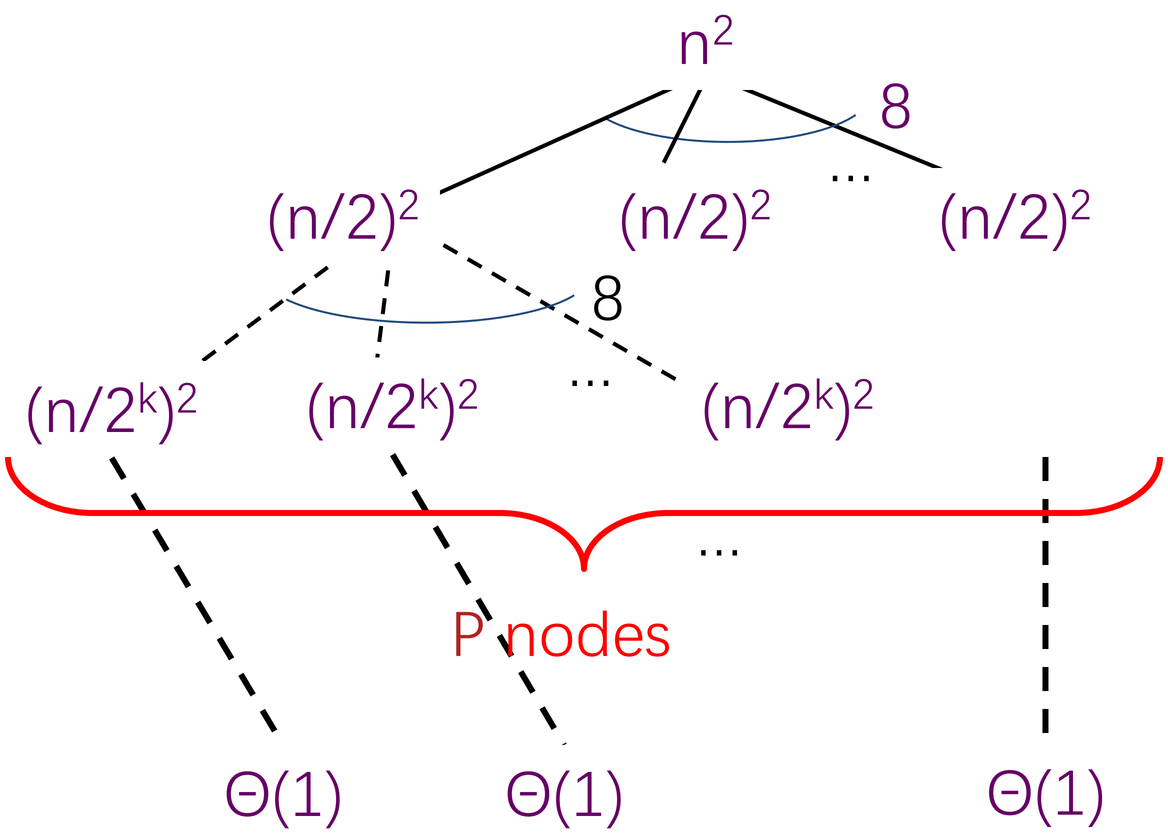
$= \Theta(n^2)$

By the busy-leaves property, we have

$$S_P(n) = O(P\,n^2).$$

We can actually prove a stronger bound.

Worst-Case Recursion Tree



Branch fully (8-way) until we get to a level k with P nodes and then branch serially from there on.

We have $8^k = P$, which implies that $k = \log_8 P = (\lg P)/3$.

The cost per level grows geometrically from the root to level k and then decreases geometrically from level k to the leaves.

Thus, the space is $\Theta(P(n/2^{(\lg P)/3})^2) = \Theta(P^{1/3}n^2)$.

BASIC PROPERTIES OF STORAGE ALLOCATORS

Allocator Speed

Definition. Allocator **speed** is the number of allocations and deallocations per second that the allocator can sustain.

Q. Is it more important to maximize allocator speed for large blocks or small blocks?

A. Small blocks!

Q. Why?

A. Typically, a user program writes all the bytes of an allocated block. A large block takes so much time to write that the allocator time has little effect on the overall runtime. In contrast, if a program allocates many small blocks, the allocator time can represent a significant overhead.

Fragmentation

Definition. The **user footprint** is the maximum over time of the number M of bytes in use by the user program (allocated but not freed). The **allocator footprint** is the maximum over time of the number H of bytes of memory provided to the allocator by the operating system. The **fragmentation** is $F = H/M$, and the space utilization is M/H.

Remark. H grows monotonically with time for many allocators.

Theorem (proved in Lecture 12)**.** The fragmentation for binned free lists is $F = O(\lg M)$. ■

Remark. Modern 64-bit processors provide about 2^{48} bytes of virtual address space. A big server might have 2^{40} bytes of physical memory.

Fragmentation Glossary

- Space overhead: Space used by the allocator for bookkeeping.
- Internal fragmentation: Waste due to allocating larger blocks than the user requests.
- External fragmentation: Waste due to the inability to use storage because it is not contiguous.
- Blowup: For a parallel allocator, the additional space beyond what a serial allocator would require.

Parallel Heap Allocation Strategies

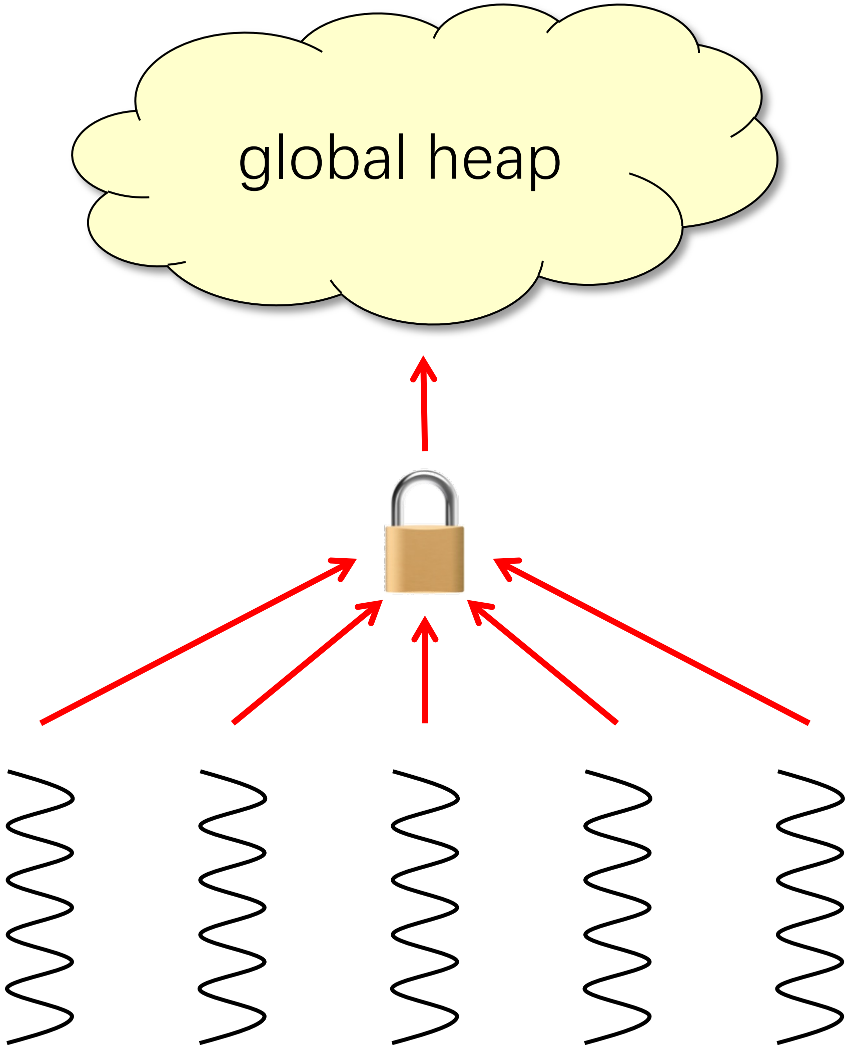
Strategy 1: Global Heap

- Default C allocator.
- All threads (processors) share a single heap.
- Accesses are mediated by a mutex (or lock-free synchronization) to preserve atomicity.

☺ Blowup = 1.

☹ Slow — acquiring a lock is like an L2-cache access.

☹ Contention can inhibit scalability.



Scalability

Ideally, as the number of threads (processors) grows, the time to perform an allocation or deallocation should not increase.

- The most common reason for loss of scalability is lock contention.

Q. Is lock contention more of a problem for large blocks or for small blocks?

A. Small blocks!

Q. Why?

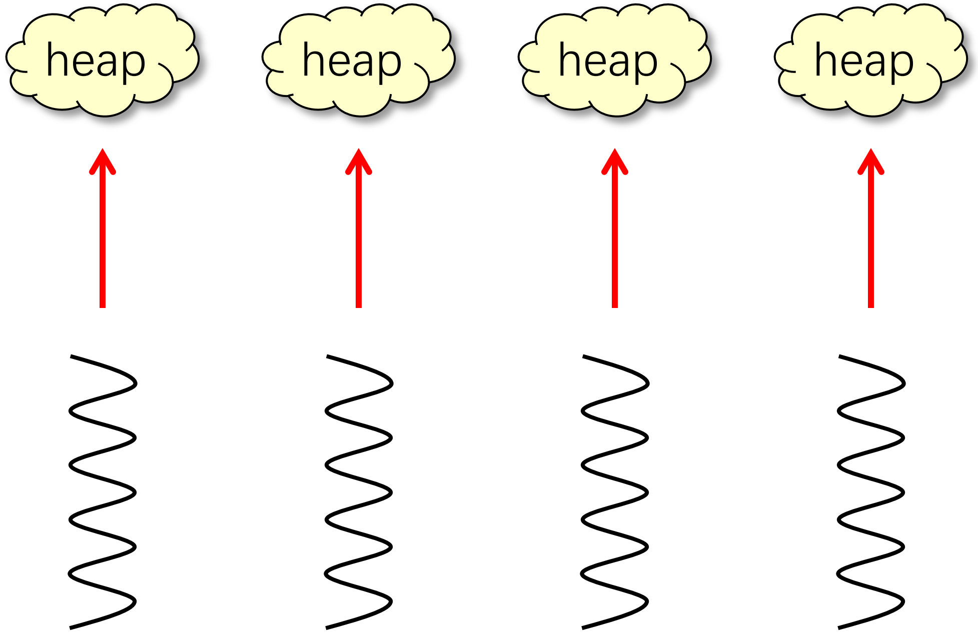
A. Typically, a user program writes all the bytes of an allocated block, making it hard for a thread allocating large blocks to issue allocation requests at a high rate. In contrast, if a program allocates many small blocks in parallel, contention can be a significant issue.

Strategy 2: Local Heaps

- Each thread allocates out of its own heap.
- No locking is necessary.

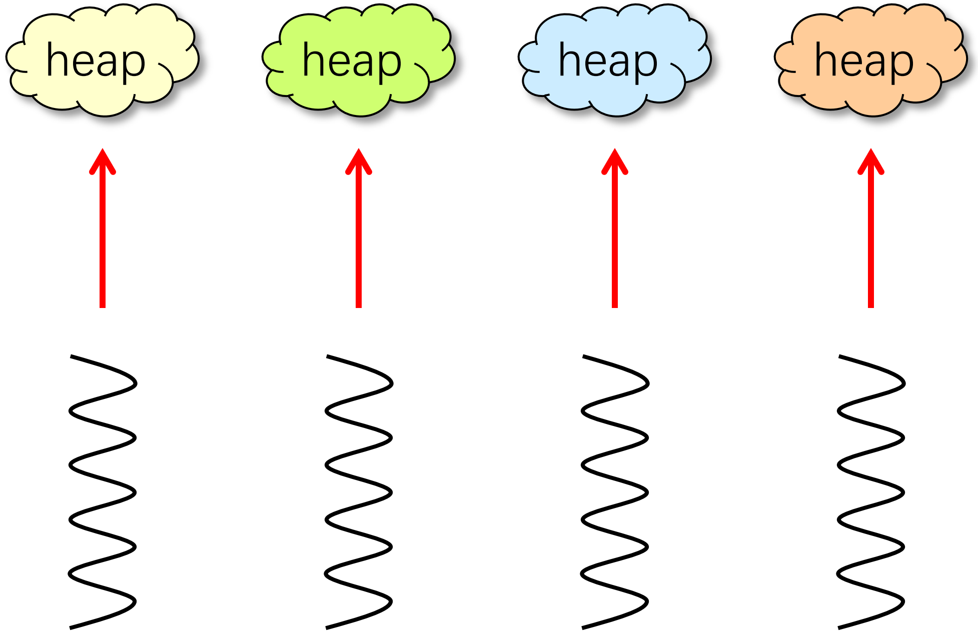
☺ Fast — no synchronization.

☹ Suffers from memory drift: blocks allocated by one thread are freed on another ⇒ unbounded blowup.



Strategy 3: Local Ownership

- Each object is labeled with its owner.
- Freed objects are returned to the owner's heap.

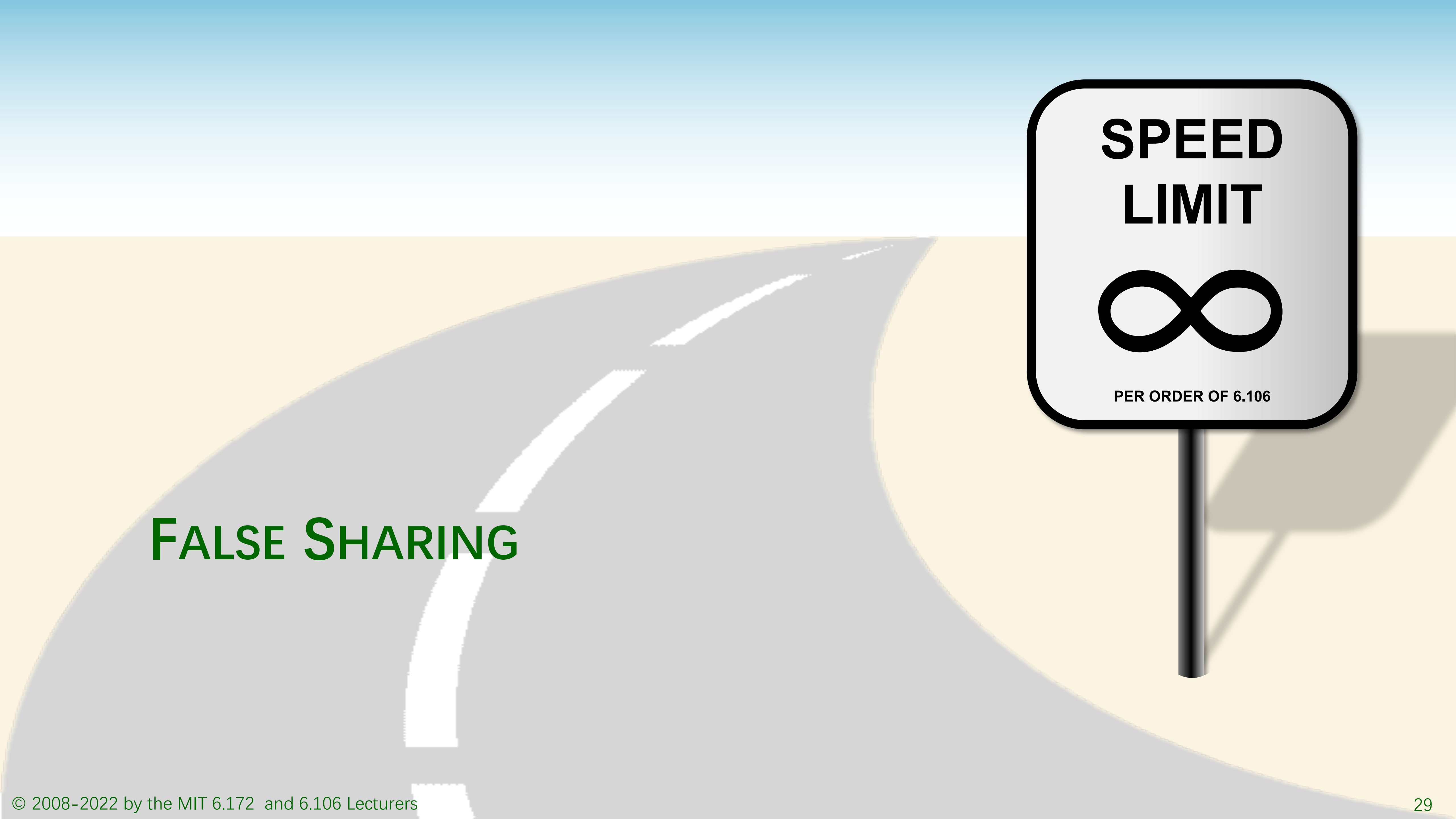


☺ Fast allocation and freeing of local objects.

☹ Freeing remote objects requires synchronization.

😐 Blowup $\leqslant P$.

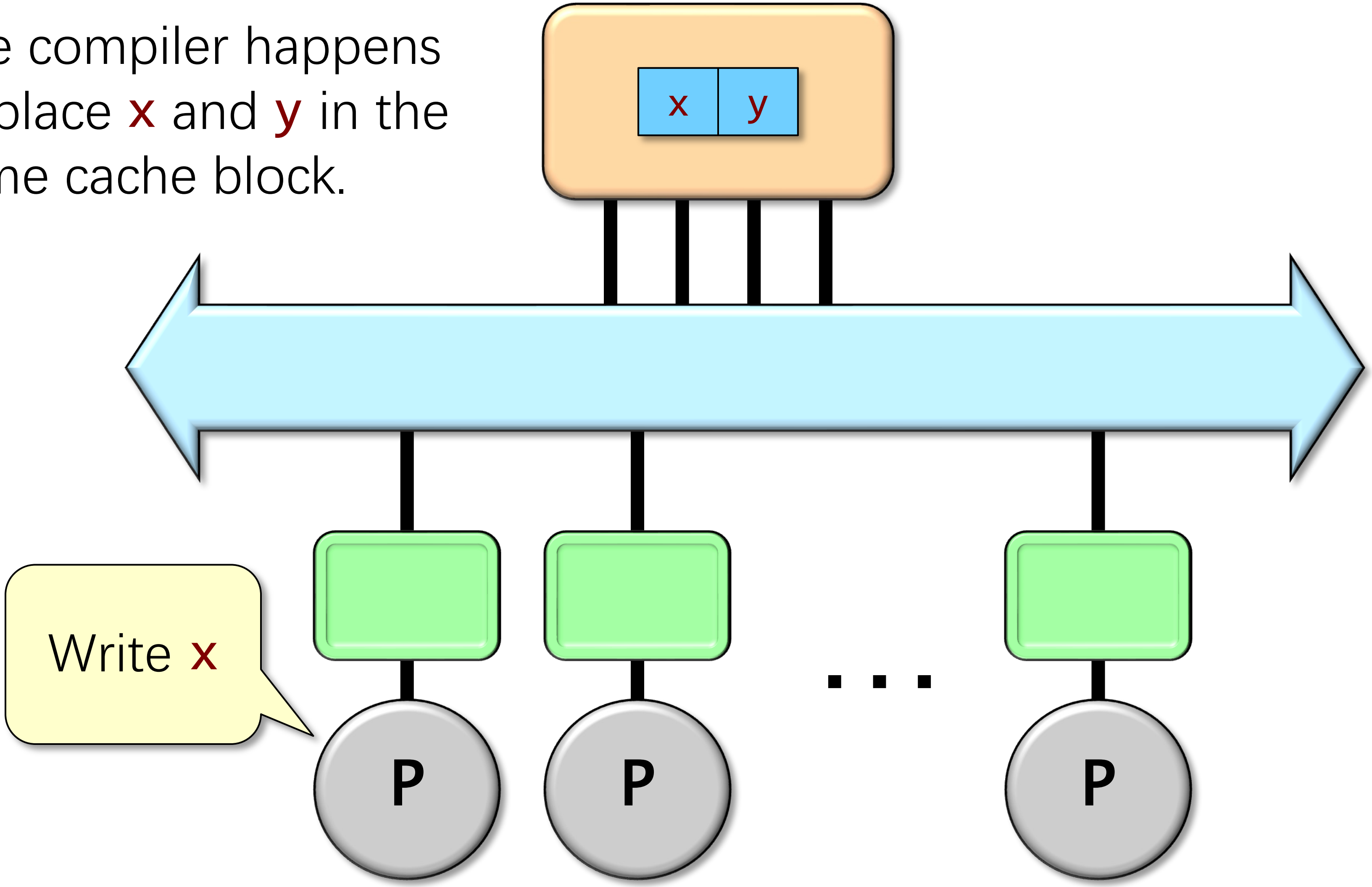
☺ Resilience to false sharing.



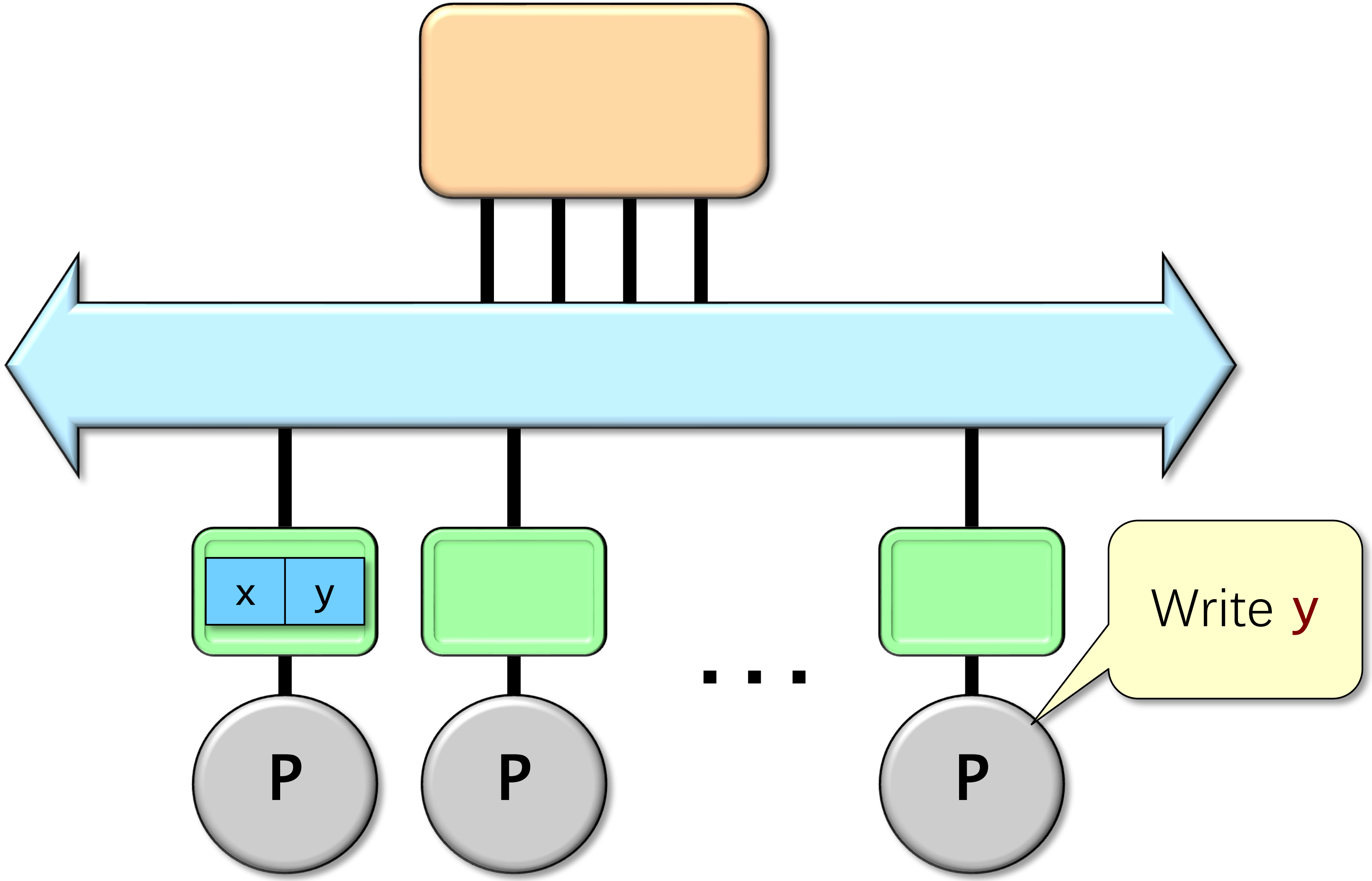
FALSE SHARING

False Sharing Example

The compiler happens to place x and y in the same cache block.

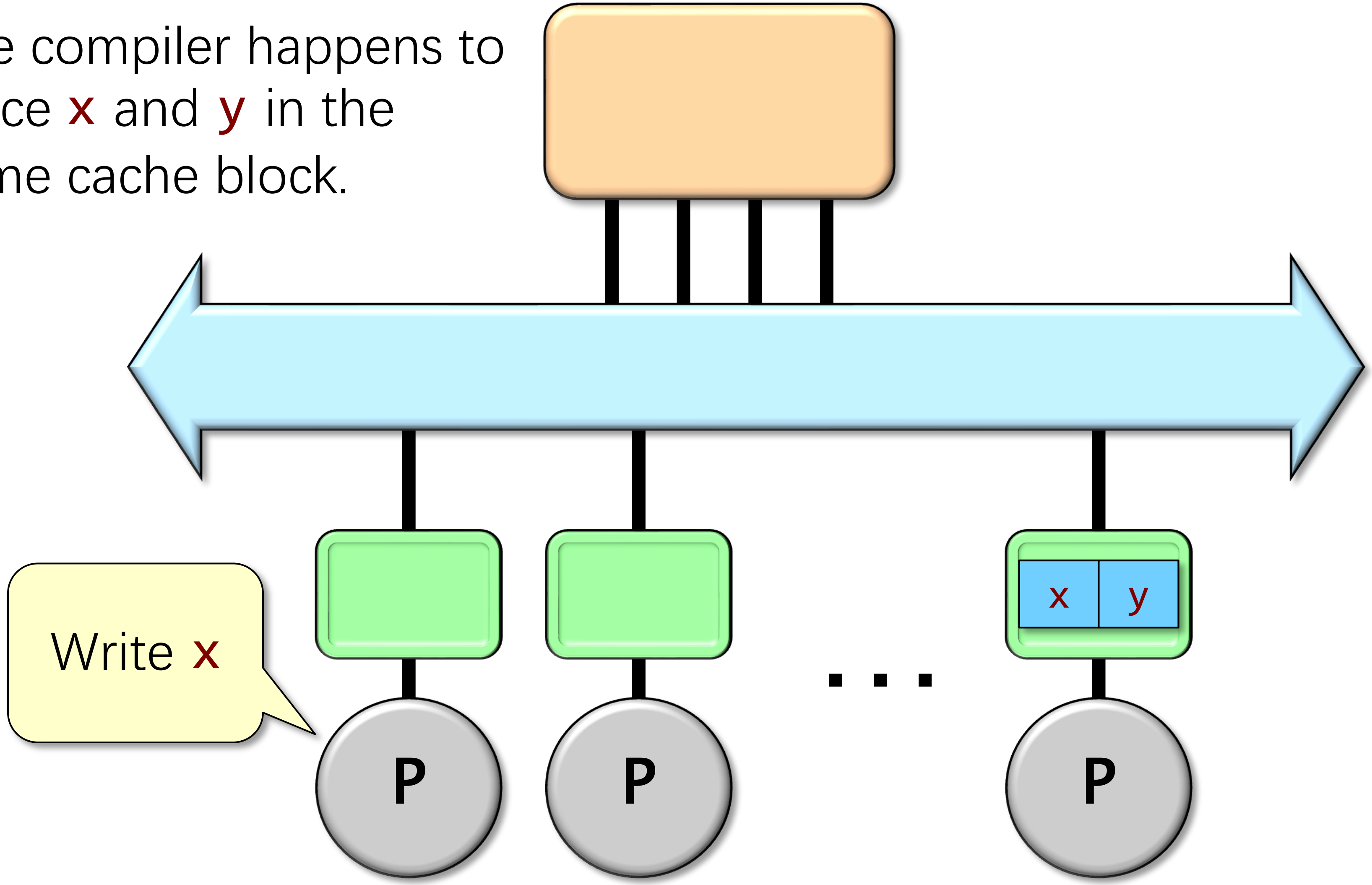


False Sharing Example



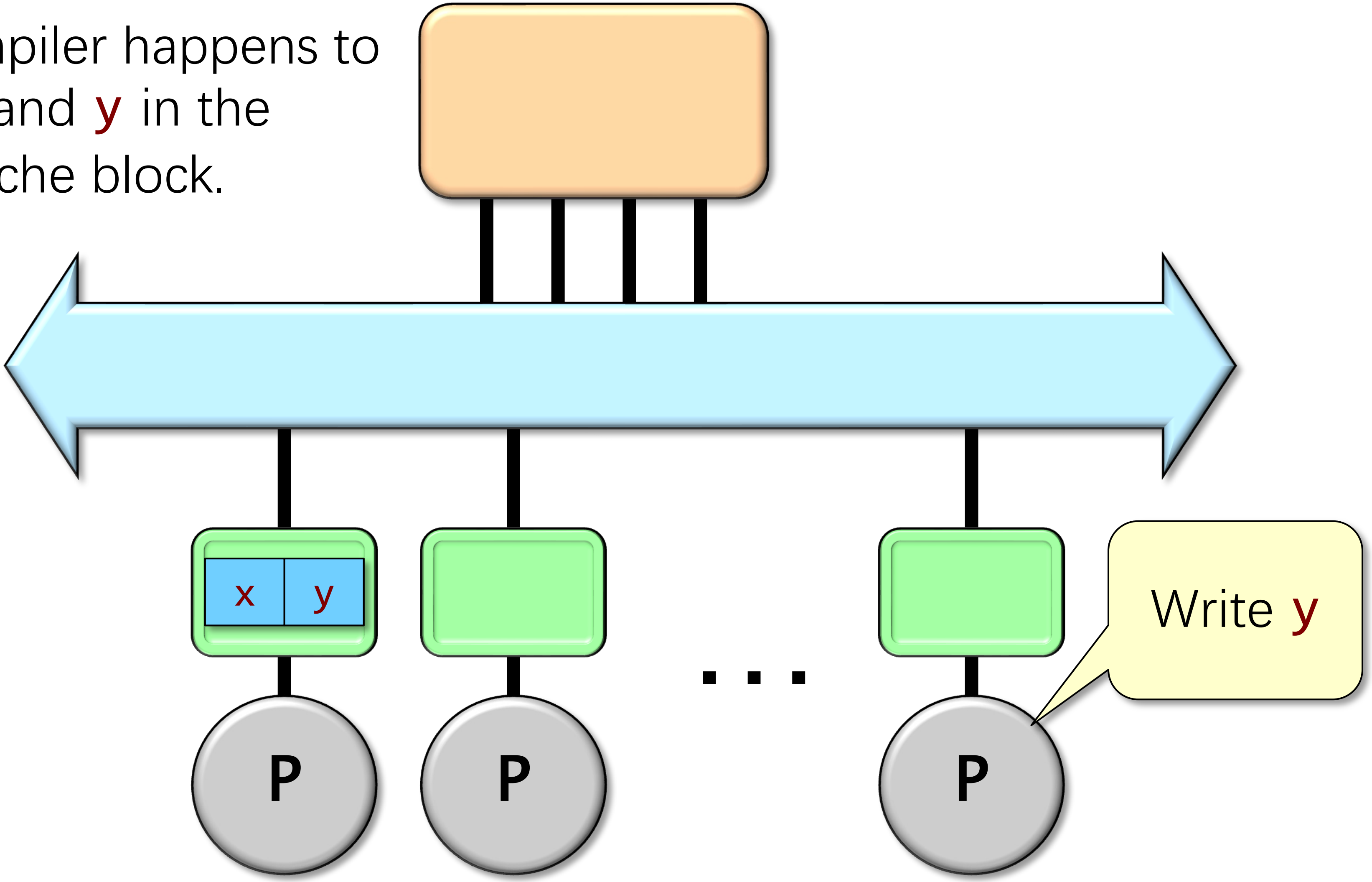
False Sharing Example

The compiler happens to place x and y in the same cache block.



False Sharing Example

The compiler happens to place x and y in the same cache block.



How False Sharing Can Occur

A program can induce false sharing having different threads process nearby objects.

- The programmer can mitigate this problem by aligning the object on a cache-line boundary and padding out the object to the size of a cache line, but this solution can be wasteful of space.

An allocator can induce false sharing in two ways:

- Actively, when the allocator satisfies memory requests from different threads using the same cache block.
- Passively, when the program passes objects lying on the same cache line to different threads, and the allocator reuses the objects' storage after the objects are freed to satisfy requests from those threads.

BACK TO PARALLEL HEAP ALLOCATION

The Hoard Allocator

(See reading.)

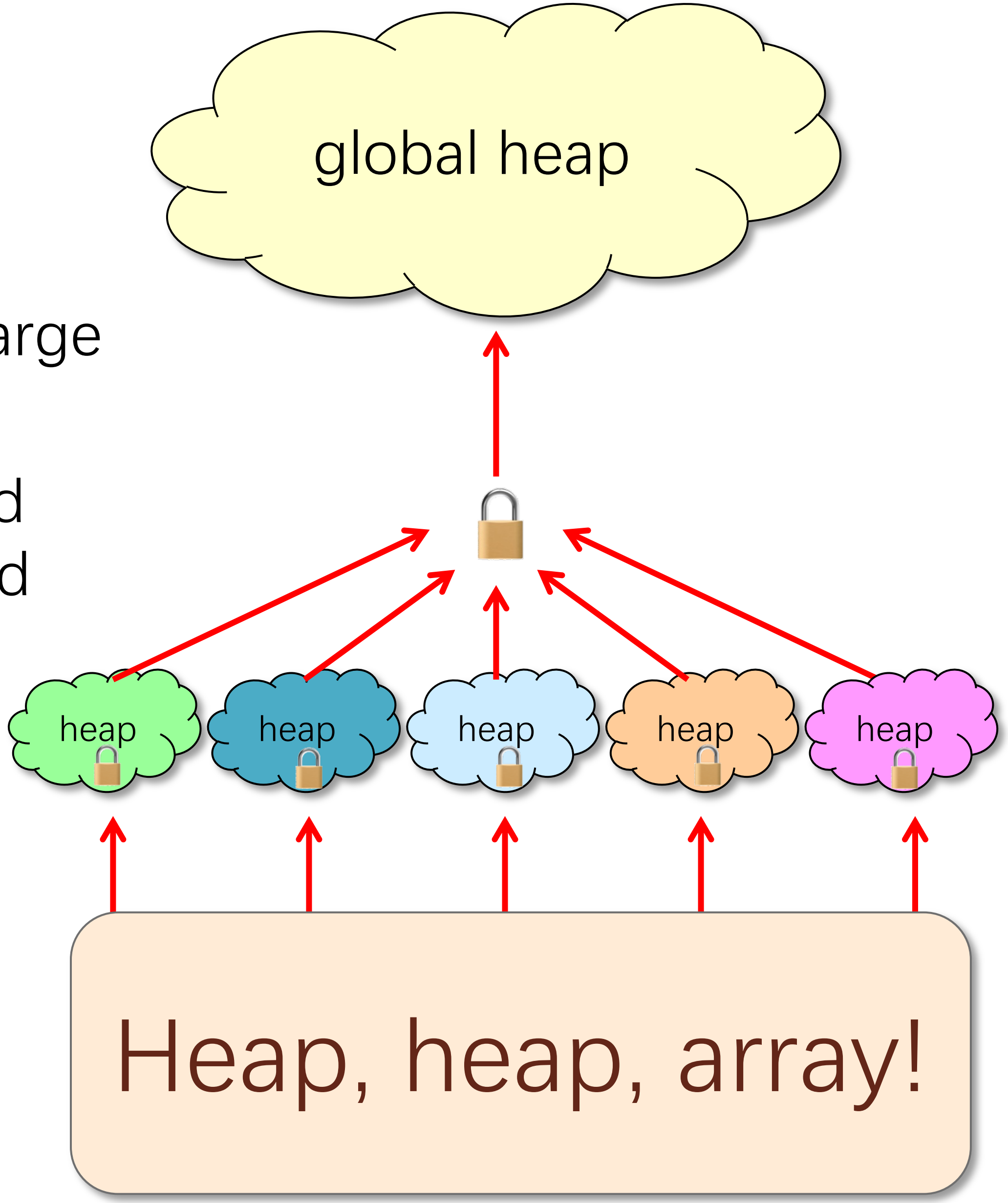
- P local heaps.
- 1 global heap.
- Memory is organized into large superblocks of size S.
- Only superblocks are moved between the local heaps and the global heap.

☺ Fast.

☺ Scalable.

☺ Bounded blowup.

☺ Resilience to false sharing



global heap

heap

heap

heap

heap

heap

Heap, heap, array!

Hoard Allocation

Assume without loss of generality that all blocks are the same size (fixed-size allocation).

`x = malloc()` on thread `i`:

```
if (there exists a free object in heap i) {
   x = an object from the fullest nonfull superblock in i’s heap;
} else {
   if (the global heap is empty) {
      B = a new superblock from the OS;
   } else {
      B = a superblock in the global heap;
   }
   set the owner of B to i;
   x = a free object in B;
}
return x;
```

Hoard Deallocation

Let m_i be the in-use storage in heap `i`, and let h_i be the storage owned by heap `i`.

Hoard maintains the following invariant for all heaps `i`:

$$m_i \geqslant \min(h_i - 2S, h_i/2),$$

where S is the superblock size.

`free(x)`, **where `x` is owned by thread `i`:**

```
put x back in heap i;
if ($m_i < \min(hi - 2S, h_i/2)$) {
   move a superblock that is at least half empty from
   heap i to the global heap;
};
```

Hoard's Blowup

Lemma. The maximum storage allocated in global heap is at most maximum storage allocated in local heaps.

Theorem. Let M be the user footprint for a program, and let H be Hoard's allocator footprint. We have

$$H \leqslant O(SP + M) ,$$

and hence the blowup is

$$H/M = O(SP/M + 1) . \blacksquare$$

Proof. Analyze the storage in local heaps.

Recall that $m_i \geqslant \min(h_i - 2S, h_i/2)$.

First term: at most $2S$ unutilized storage per heap for a total of $O(SP)$.

Second term: allocated storage is at most twice the used storage for a total of $O(M)$. ■

Other Solutions

jemalloc is like Hoard, with a few differences:

- jemalloc has a separate global lock for each different allocation size.
- jemalloc allocates the object with the smallest address among all objects of the requested size.
- jemalloc releases empty pages using

  `madvise(p, MADV_DONTNEED, ...)` ,

  which zeros the page while keeping the virtual address valid.
- jemalloc is a popular choice for parallel systems due to its performance and robustness.

SuperMalloc (see reading) is an interesting contender.

Allocator Speeds

Allocator	SLOC	32 threads
Default	6,281	0.97 M/s
Hoard	16,948	17.1 M/s
jemalloc	22,230	38.2 M/s
SuperMalloc	3,571	131.7 M/s

DRAM Antics

Levels of the Memory Hierarchy

Capacity
Access Time
Cost

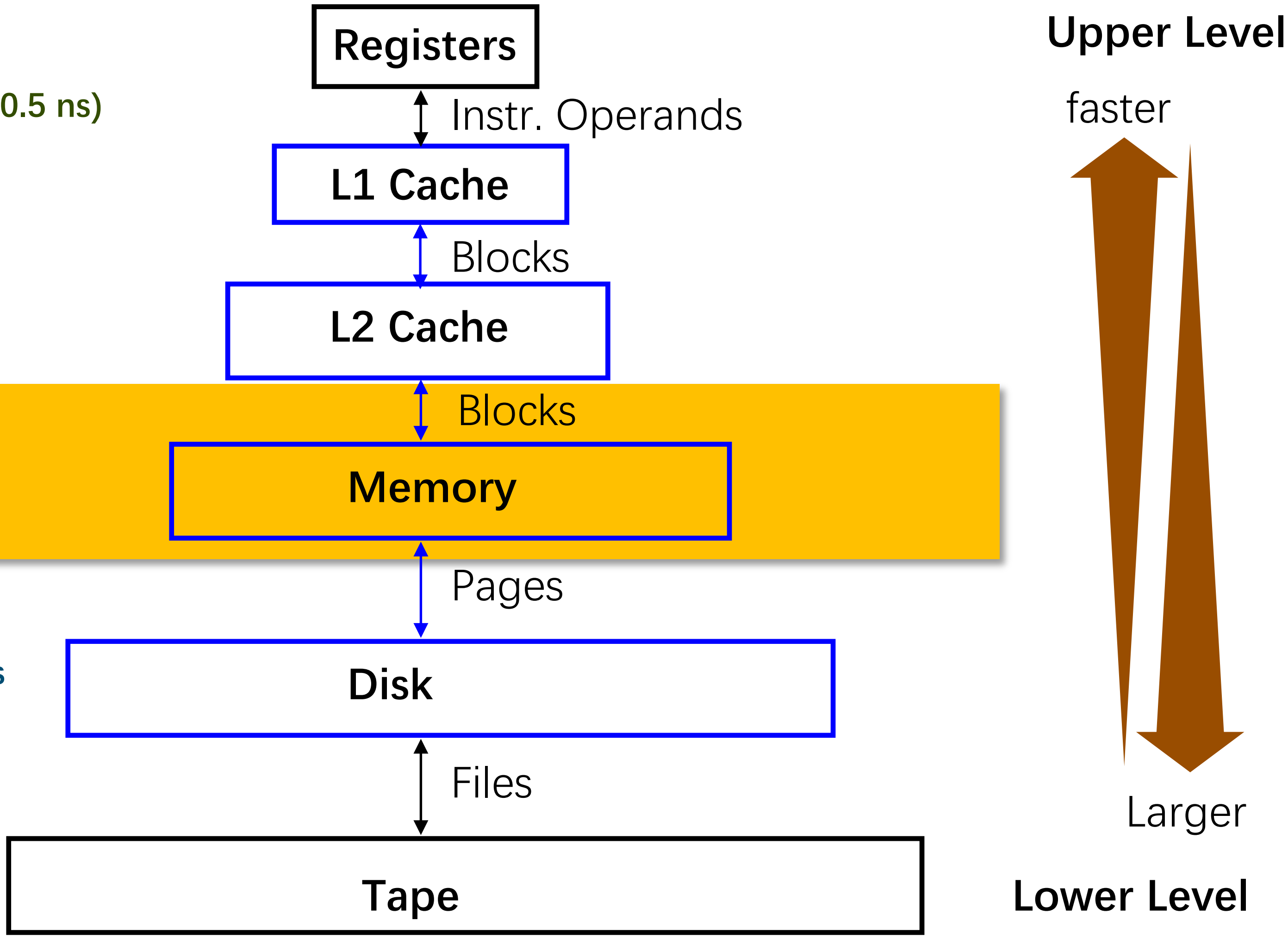
CPU Registers
100s Bytes
300 – 500 ps (0.3-0.5 ns)

L1 and L2 Cache
10s-100s K Bytes
~1 ns - ~10 ns
$1000s/ GByte

Main Memory
G Bytes
80ns- 200ns
~ $100/ GByte

Disk
10s T Bytes, 10 ms
(10,000,000 ns)
~ $1 / GByte

Tape
infinite
sec-min
~$1 / GByte



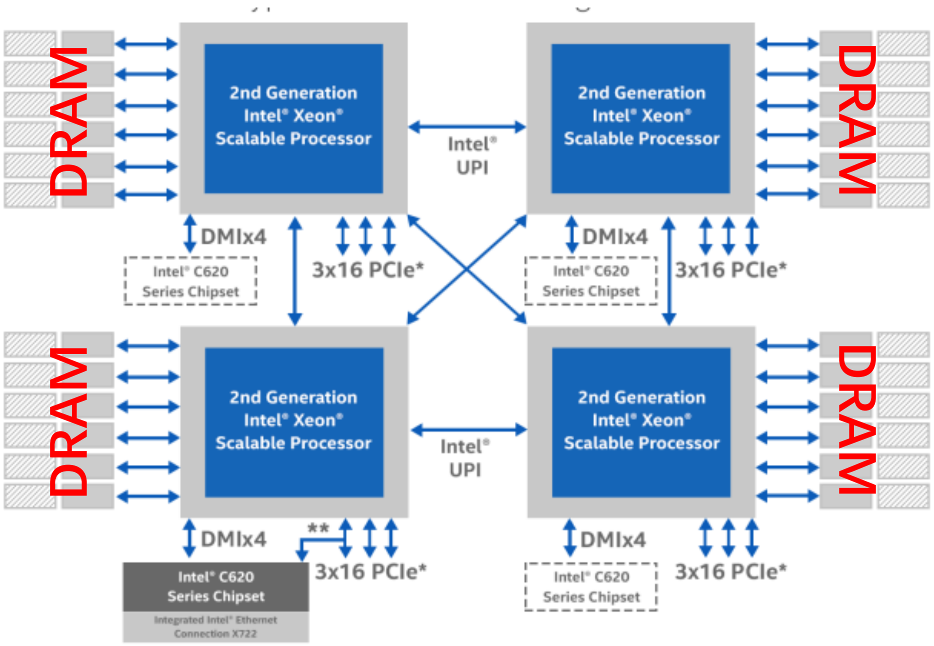
DRAM accesses

Many programs may tax the DRAM

- Bulk reads or writes
  - Example: Video editing

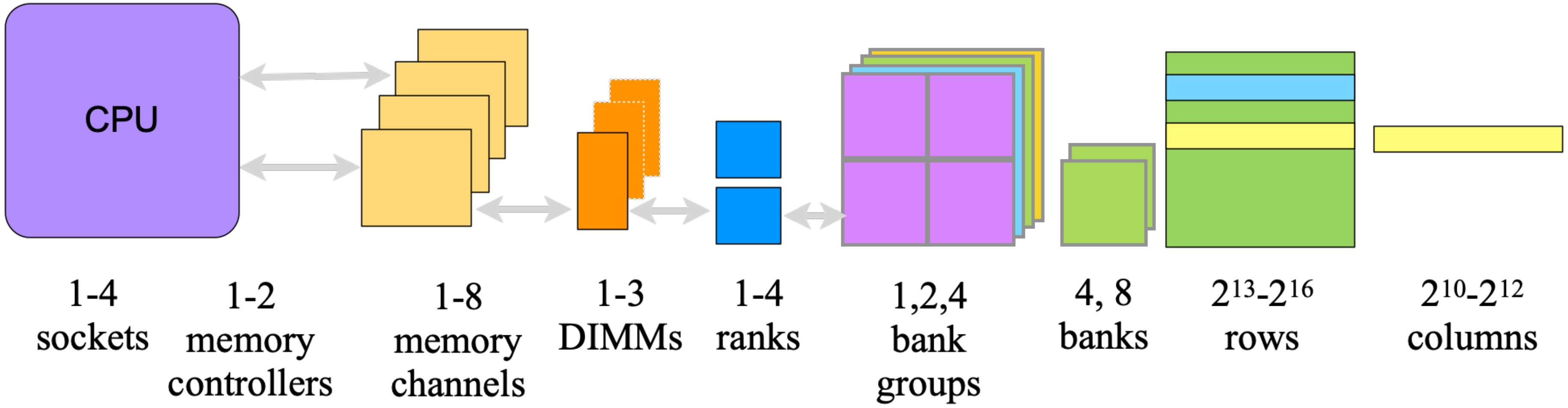
- Accesses without locality
  - Example: Graph analytics

DRAM Layout



Each socket has its own DRAMs

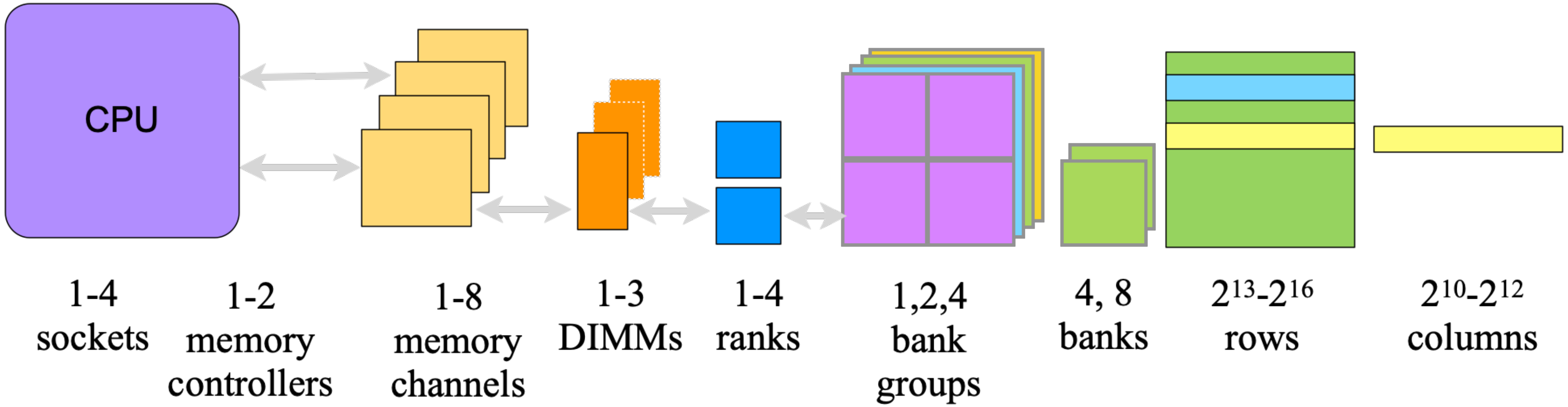
CPU to DRAM



Long pipeline from the CPU to DRAM

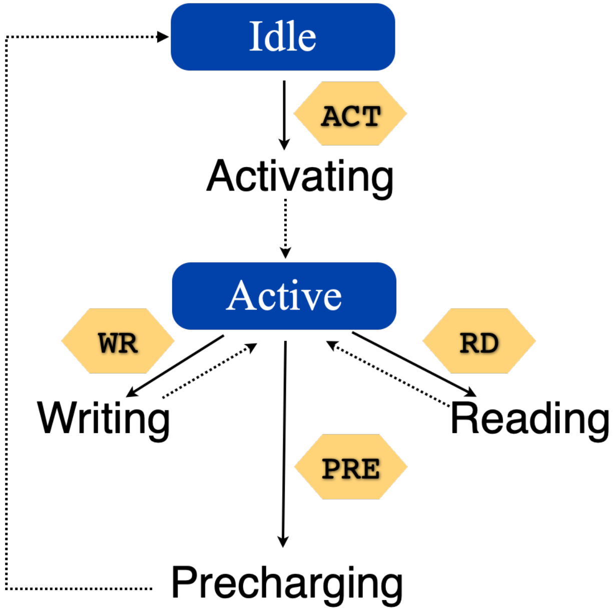
- Fanout at each level
  - Unfortunately, intel randomizes → uniformly slow ☹
- Bulk access at the row granularity

CPU to DRAM

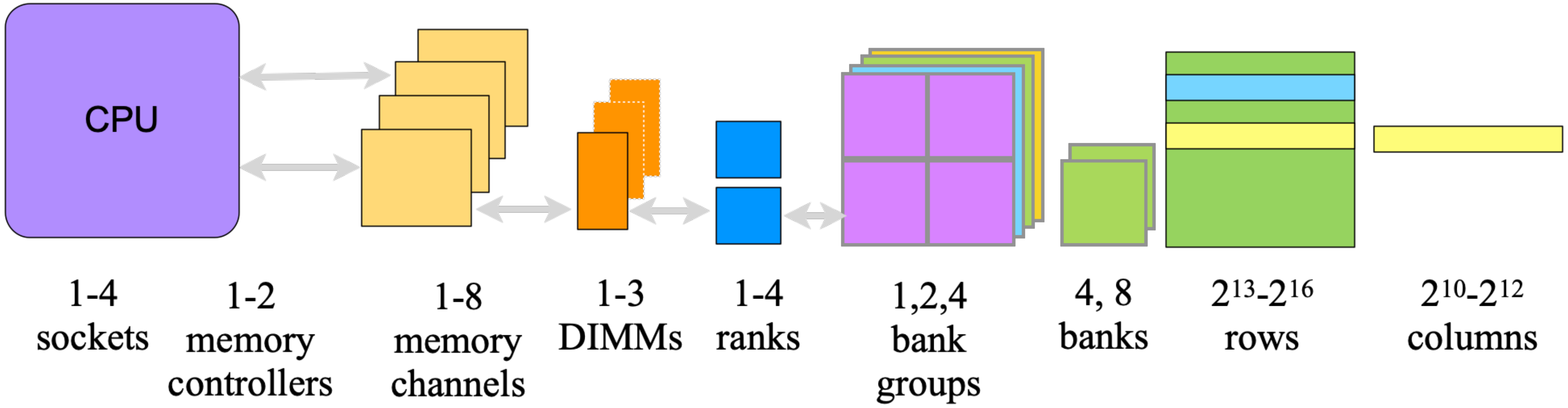


Each DRAM is

- A complex state machine

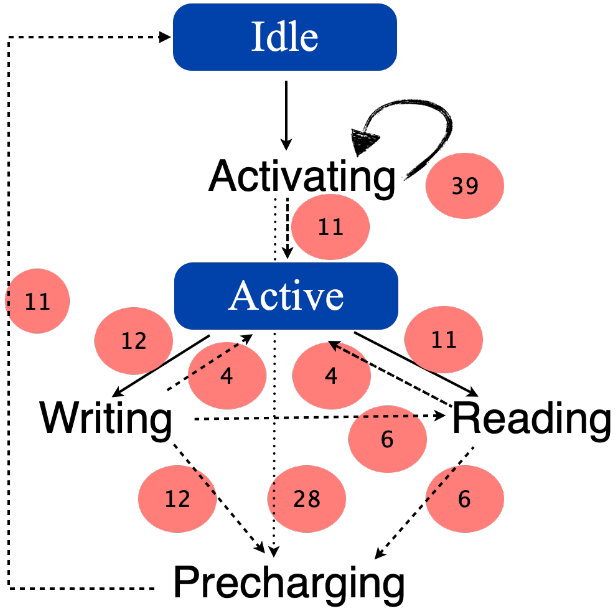


CPU to DRAM



Each DRAM is

- A complex state machine
- Slow to respond

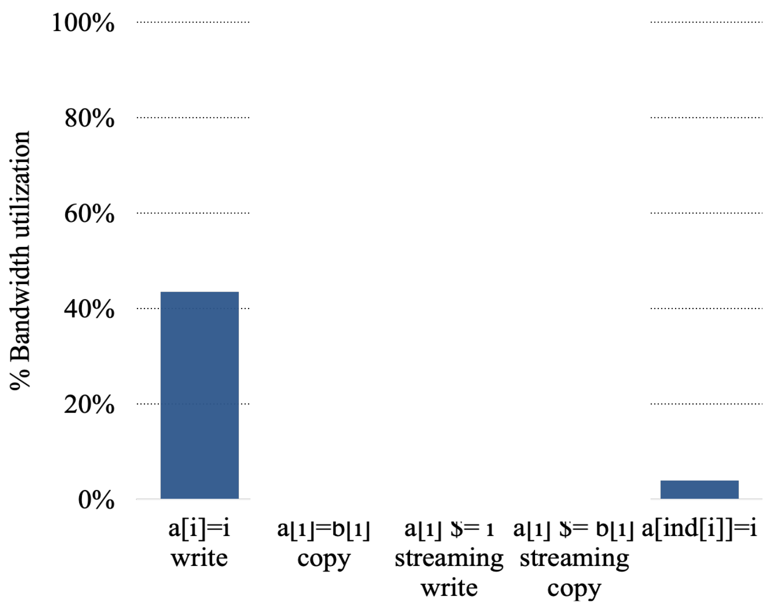


1 cycle = 1.25 ns

DRAM Performance

When writing DRAM utilization is low

- First need to read
- Then change the DRAM state
- Finally, write



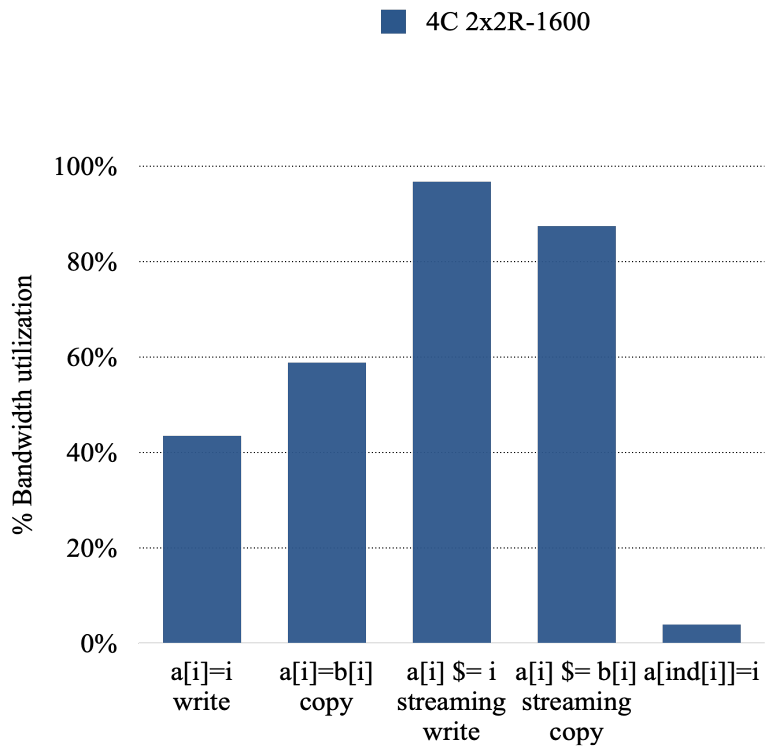
DRAM Performance

When writing DRAM utilization is low

- First need to read
- Then change the DRAM state
- Finally, write

Why need to read???

- Streaming Writes



DRAM Performance

```
void copy(int n, int * restrict src, int * restrict dst) {
  for (int i = 0; i < n; i++)
    dst[i] = src[i];
 }
```

```
#include <immintrin.h>

void copy(int n, int * restrict src, int * restrict dst) {
  int vector_len = 256 / 32; // 8?
  int remainder = n % vector_len;
  for (int i = 0; i < remainder; i++)
    dst[i] = src[i];
  for (int i = 0; i < n / vector_len; i++) {
    __m256i *dst_ptr = (__m256i *)(dst + remainder + i * vector_len);
    __m256i *src_ptr = (__m256i *)(src + remainder + i * vector_len);
    _mm256_stream_si256(dst_ptr, _mm256_stream_load_si256(src_ptr));
  }
}
```