6.172
Performance
Engineering
of Software
Systems

LECTURE 12

Parallel Storage Allocation

Julian Shun

Review of Memory-Allocation Primitives

Heap Storage in C

堆中

- **Allocation**

```
void* malloc(size_t s);
```

可能多于 s

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

- **Aligned allocation**

对齐分配：与a对齐，至少s

```
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:

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

usage: ① fit within cache lines to reduce the number of cache misses ② vectorization operators also require 与a对齐的内存地址

- **Deallocation**

fail to free → memory leak 内存泄漏

```
void free(void *p);
```

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

free -- more than 1 → double freeing

Allocating Virtual Memory

The mmap() system call can be used to allocate virtual memory by memory mapping: 文件映射到内存

```
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 result in a segfault.

Properties of mmap()

不会立刻分配

- 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.

标记

写的时候才

- 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.

所以 mmap 1T 在 1G 之中

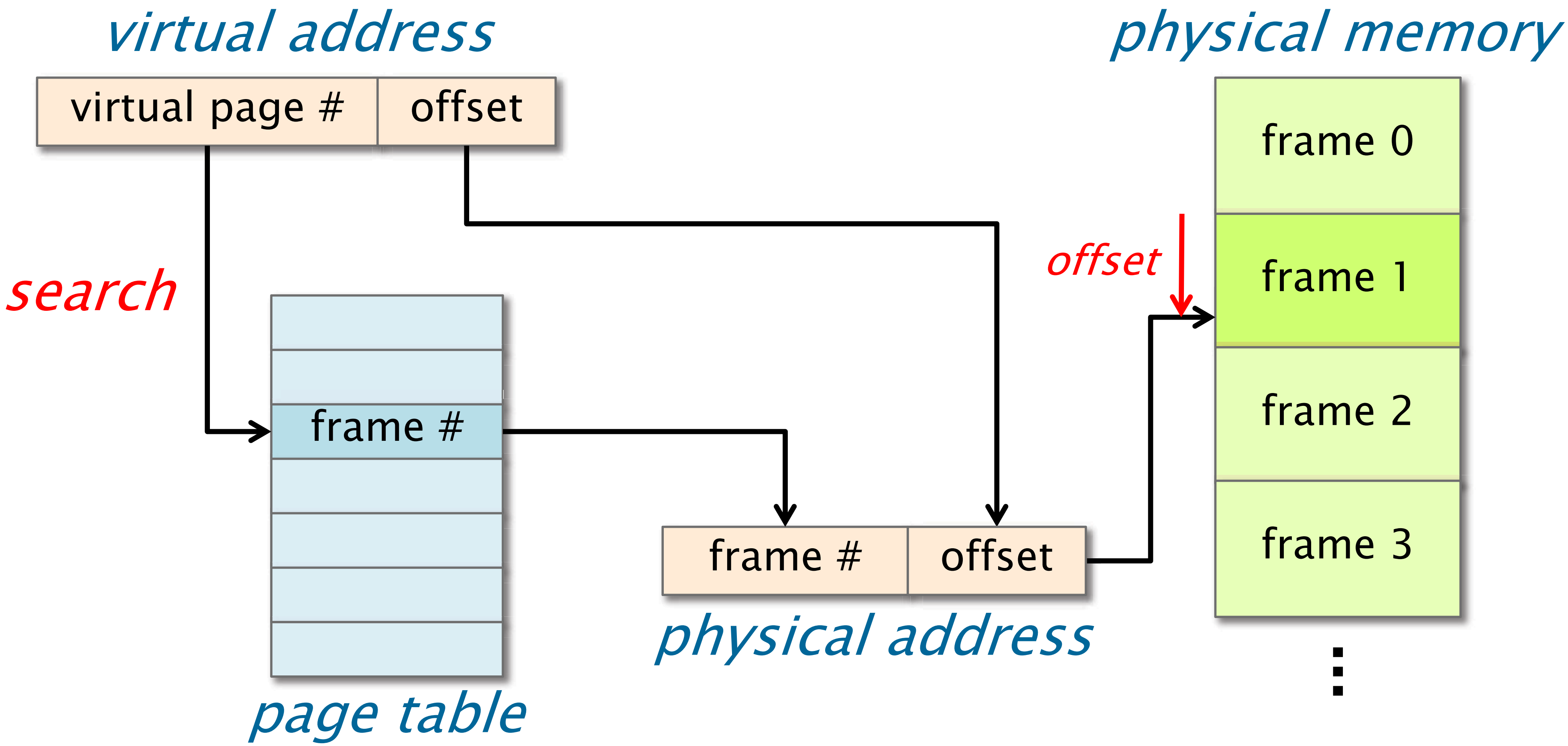
- You can mmap() a terabyte of virtual memory on a machine with only a gigabyte of DRAM.
- A process may die from running out of physical memory well after after the mmap() call.

What’s the Difference…

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

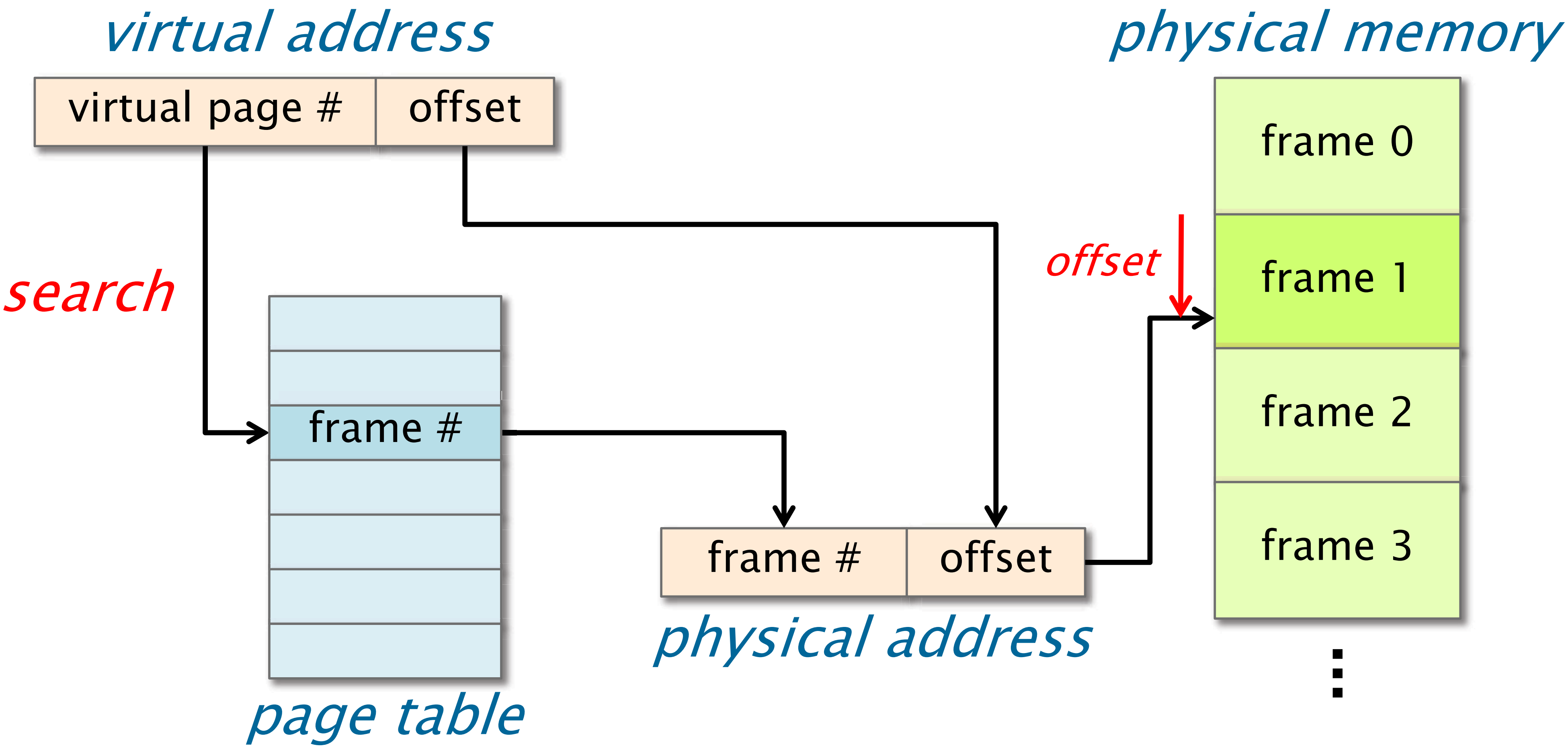
- The functions malloc() and free() are part of the memory-allocation interface of the heap-management code in the C library.
- The heap-management code uses available system facilities, including mmap(), to obtain memory (virtual address space) from the kernel. 获取内存
- The heap-management code within malloc() attempts to satisfy user requests for heap storage by reusing freed memory whenever possible. malloc 不够时，也会调用 mmap
  重用 free memory
  小额分配用 malloc
  减少外部碎片
- When necessary, the malloc() implementation invokes mmap() and other system calls to expand the size of the user’s heap storage.
  mmap 很强大，只在大片分配

Address Translation



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

Address Translation

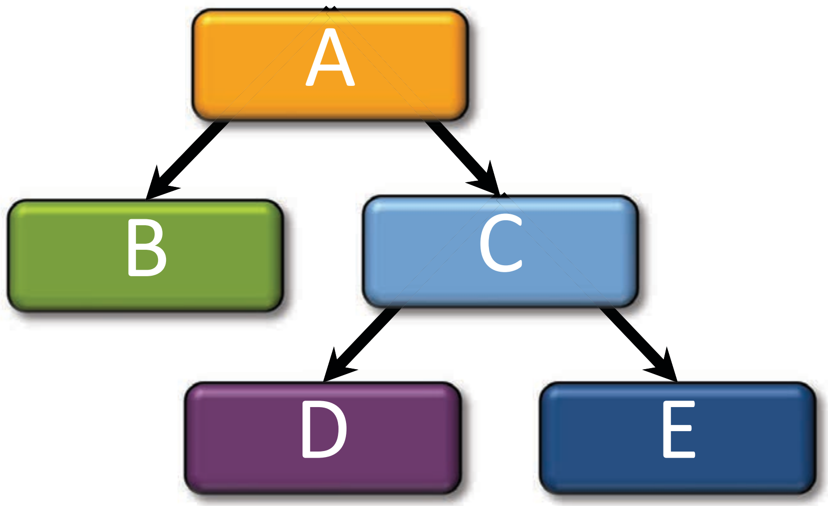


Since page-table lookups are costly, the hardware contains a translation lookaside buffer (TLB) to cache recent page-table lookups.

CACTUS STACKS

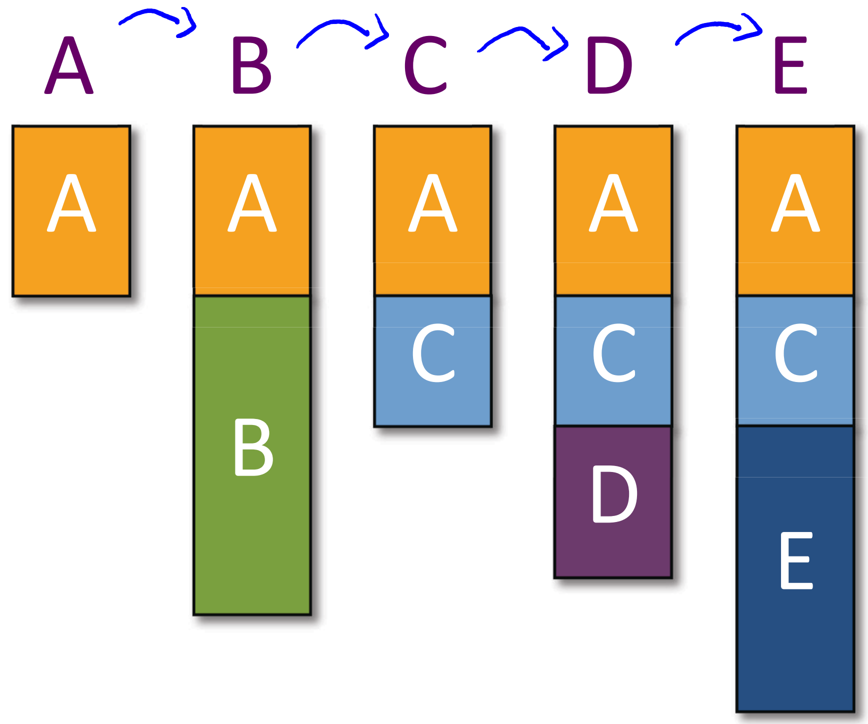
Traditional Linear Stack

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



invocation tree

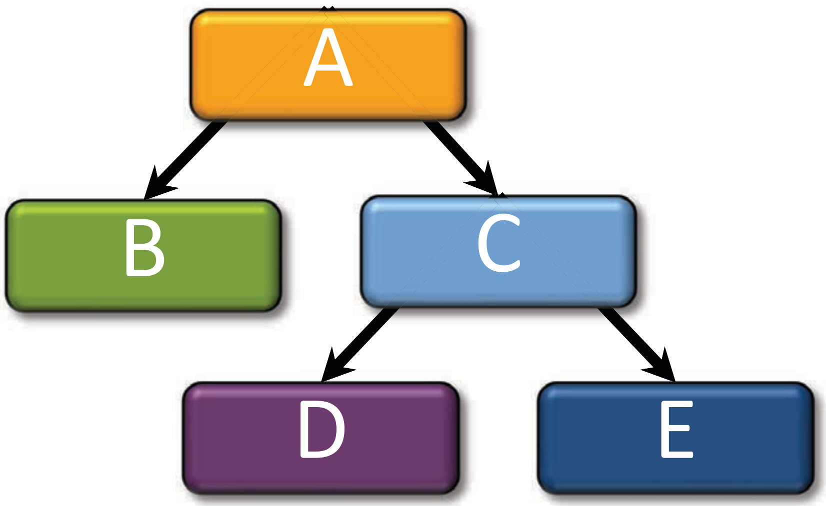
调用



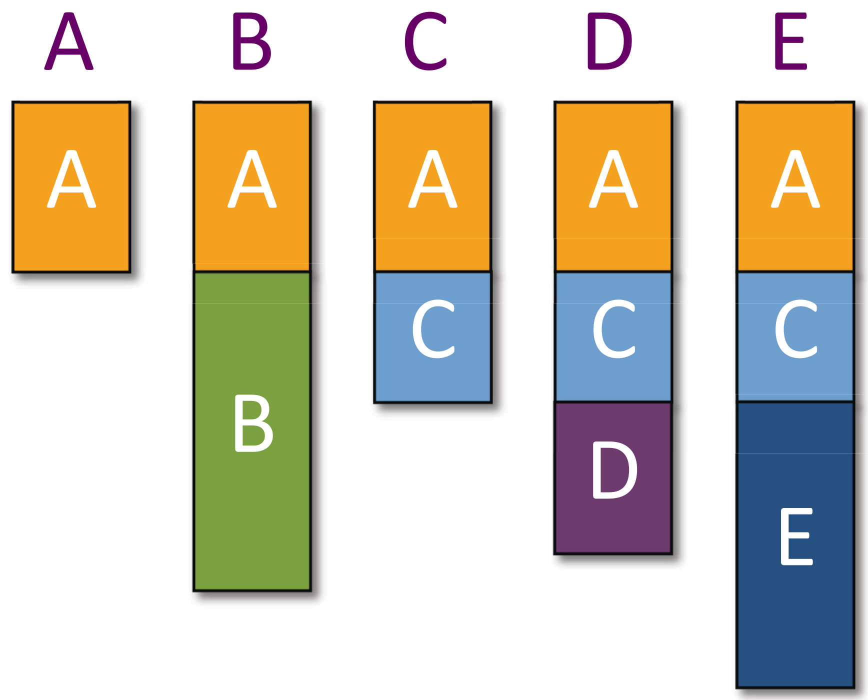
views of stack

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.



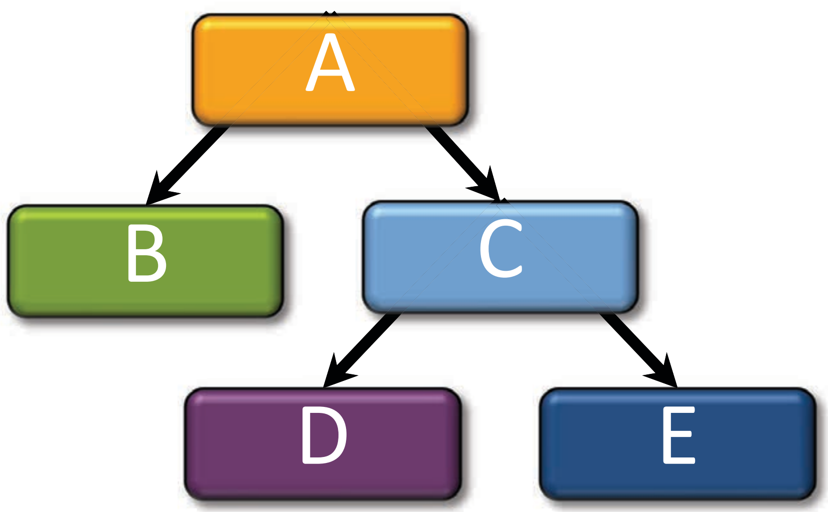
invocation tree



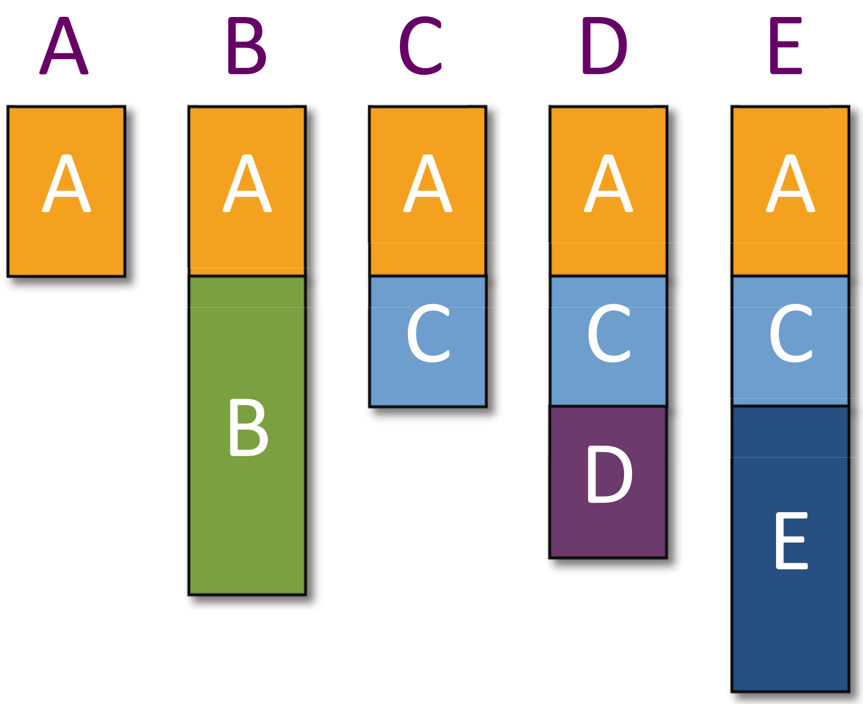
views of stack

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.

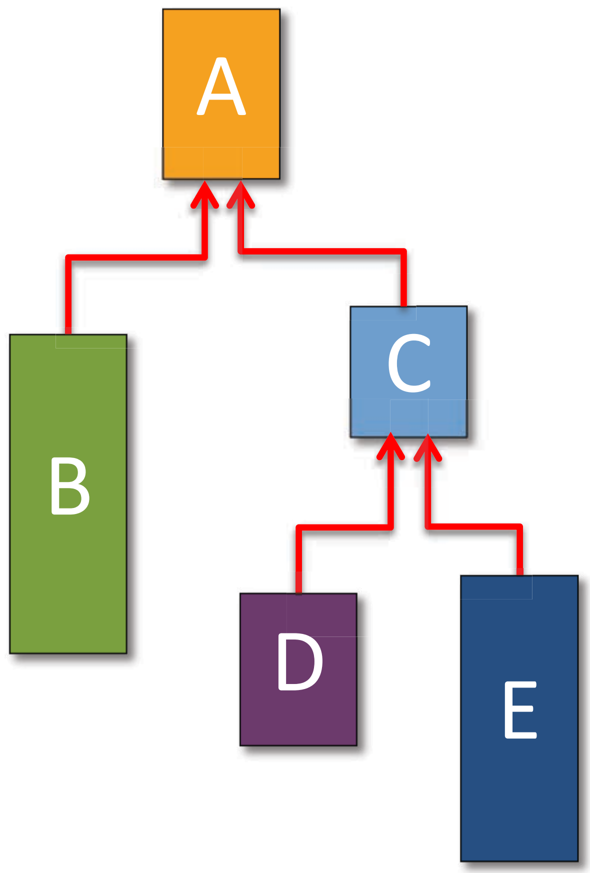
不在栈上进行分配

⇒ 堆分配

问题：① 性能优化

② 互关链接：混合使用线性和该方案，可能出问题。

一些代码要用到栈逻辑

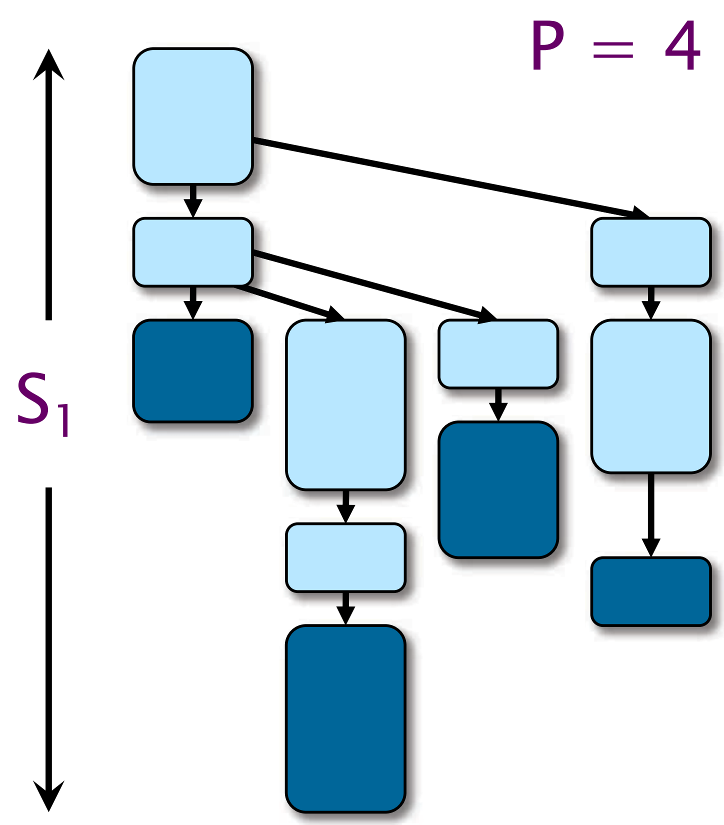


Space Bound

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 $S_P \leq P\,S_1$.

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

每个都是一个处理器



D&C Matrix Multiplication

分治矩阵

```
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);
  }
}
```

8

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

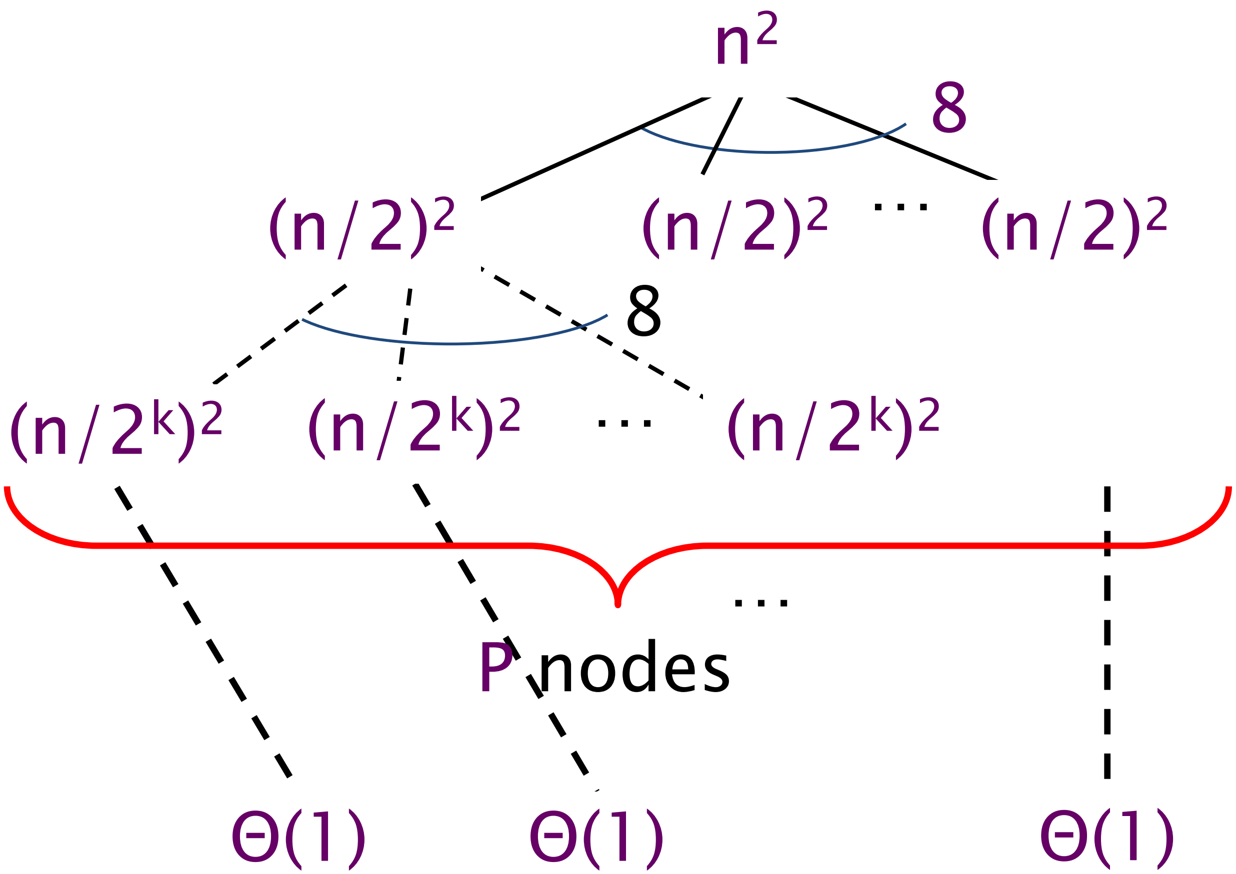
$= \Theta(n^2)$

By the busy-leaves property, we have

$S_P(n) = O(Pn^2)$.

We can actually prove a stronger bound.

Worst-Case Recursion Tree



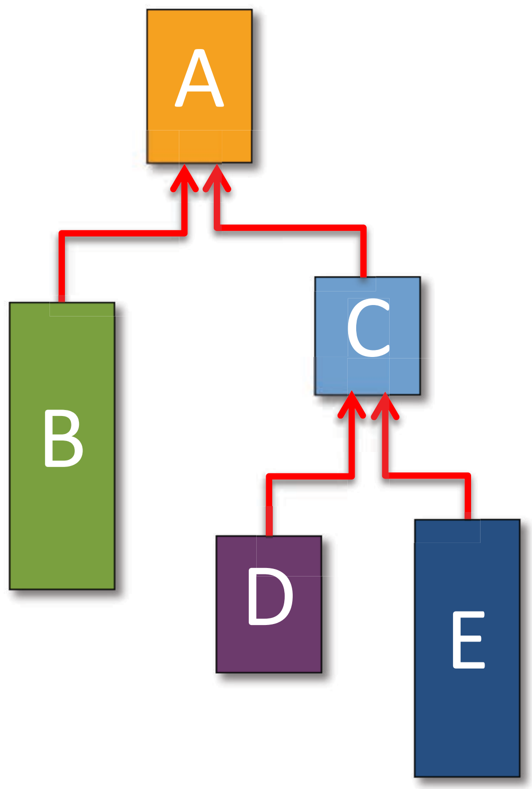
Branch fully (8-way) until we get to a level k with P nodes and then branch serially from there on.

P nodes 都在下面，将占更大的空间

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 $\Theta(P(n/2^{(\lg P)/3})^2) = \Theta(P^{1/3}n^2)$.

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.



Basic Properties of Storage Allocators

Allocator Speed

分配器

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! 在小块中分配速度更重要

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.

更多时间在写，分配占比不大

但对于小块就要加速，因为时间占比大

Fragmentation

用户用到的

Definition. The user footprint is the maximum over time of the number U of bytes in use by the user program (allocated but not freed). The allocator footprint is the maximum over time of the number A of bytes of memory provided to the allocator by the operating system. The fragmentation is $F = A/U$.

系统会提供更多或mmap没释放

A越接近U越好，因为用户可能用了所有操作系统分配的

Remark. A grows monotonically for many allocators.

Theorem (proved in Lecture 11). The fragmentation for binned free lists is $F_V = O(\lg U)$. ■

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 Allocation Strategies

并行分配存储

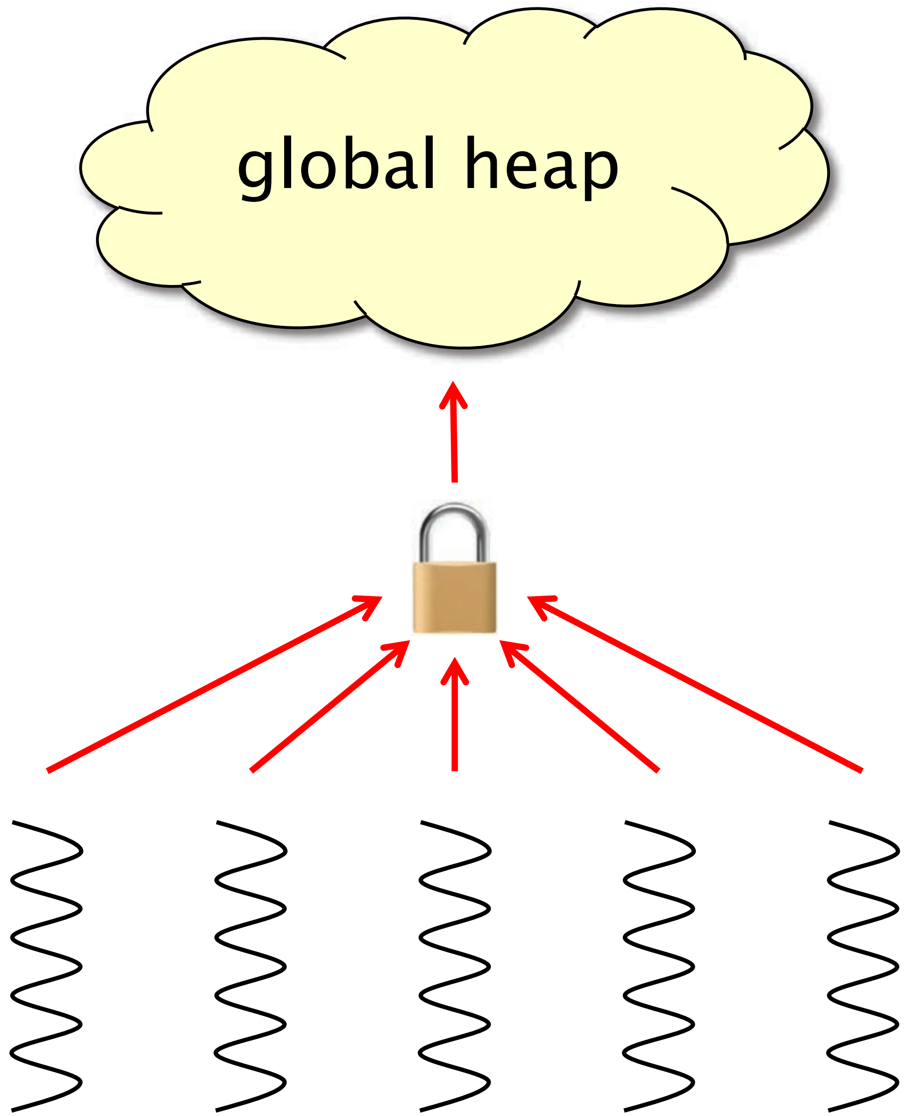
Strategy 1: Global Heap

- 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

可扩展性

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

随着线程的增加，执行分配的时间不应该增加

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

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

A. Small blocks!

Q. Why? 大块在写

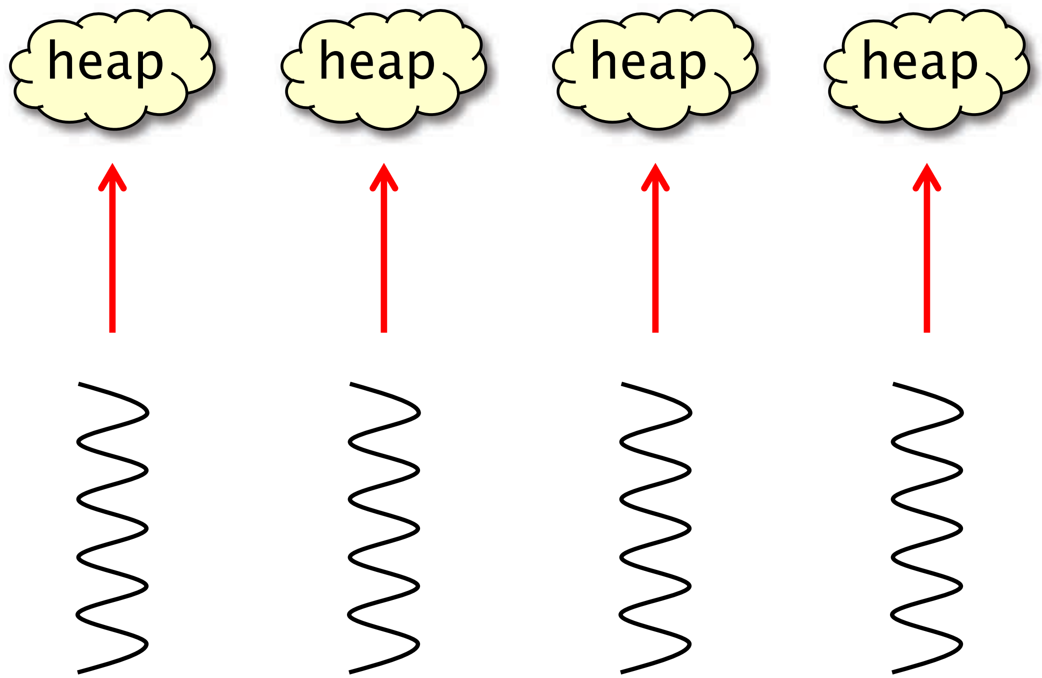
A. Typically, a user program writes all the bytes of an allocated block, making it hard for a thread allocating large blocks to issue allocation requests at a high rate. In contrast, if a program allocates many small blocks in parallel, contention can be a significant issue.

Strategy 2: Local Heaps

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

☺ Fast — no synchronization.

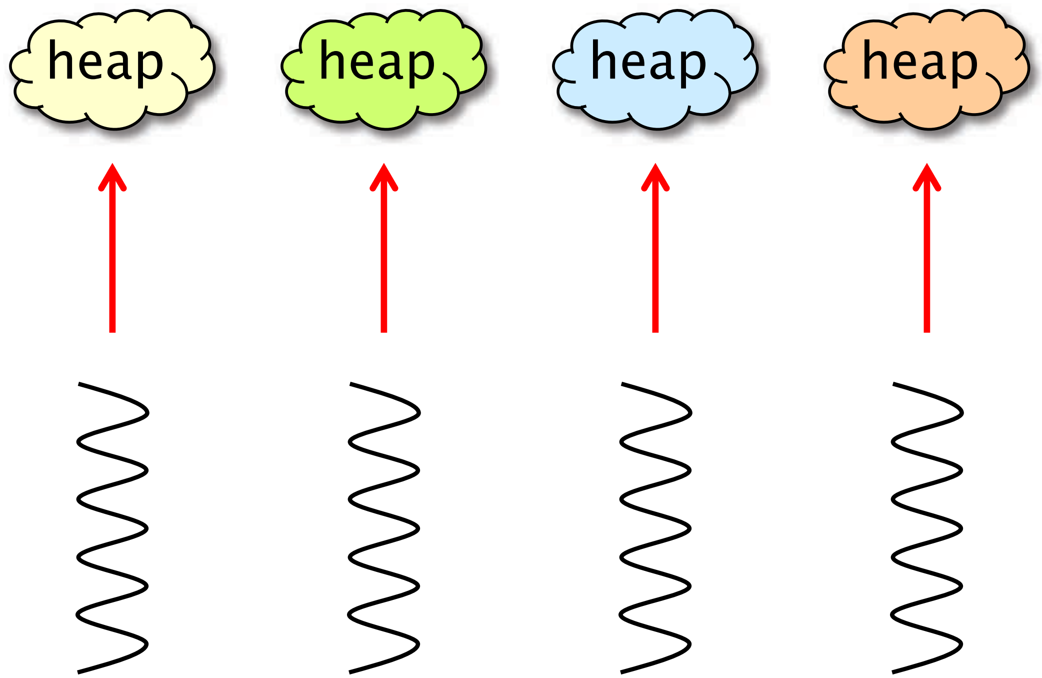
☹ Suffers from memory drift: blocks allocated by one thread are freed on another ⇒ unbounded blowup.



一个块在一个线程创建，却被另一线程释放

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. 同步

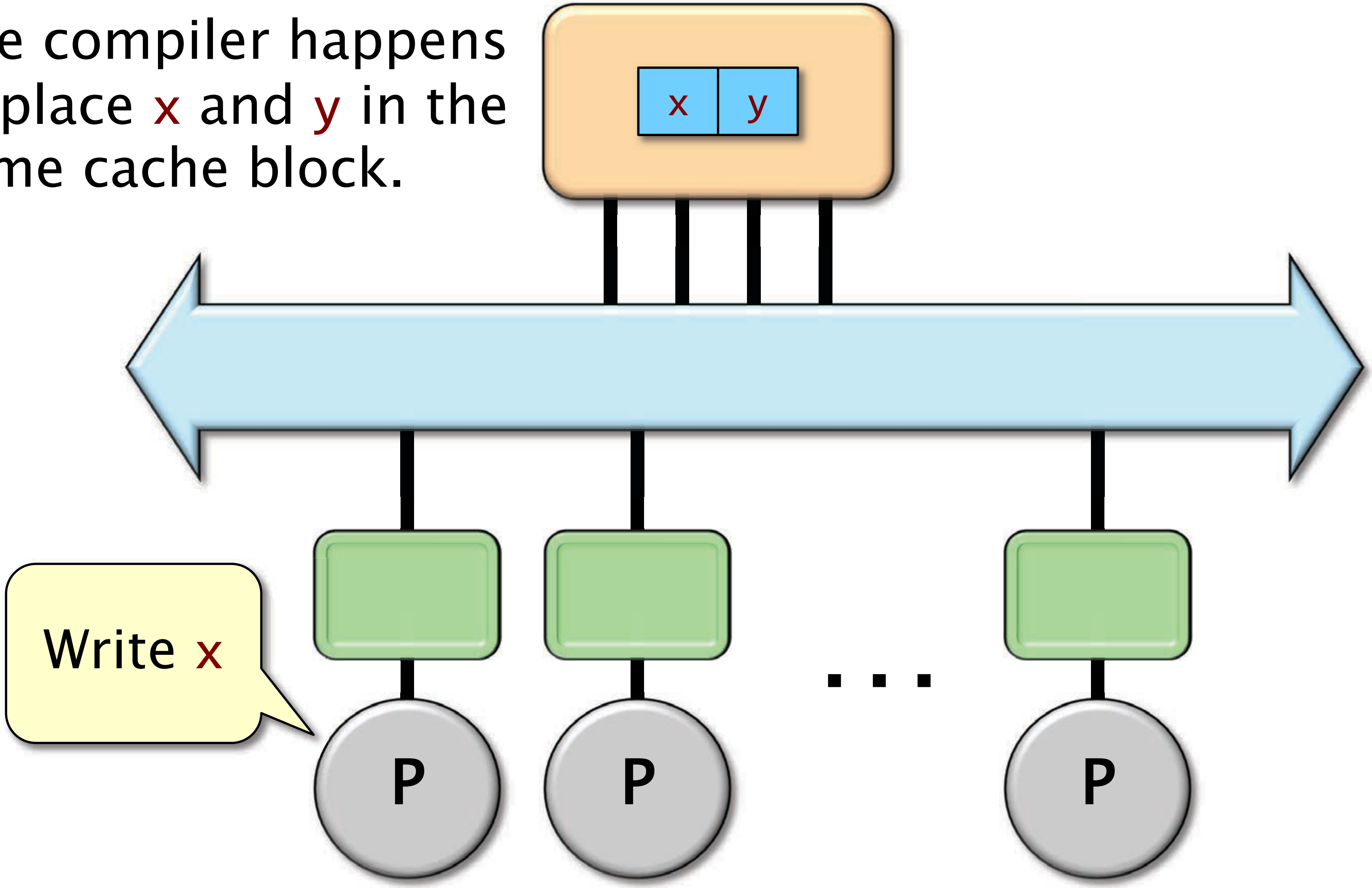
😐 Blowup ≤ P.

☺ Resilience to false sharing. 虚假分享

FALSE SHARING

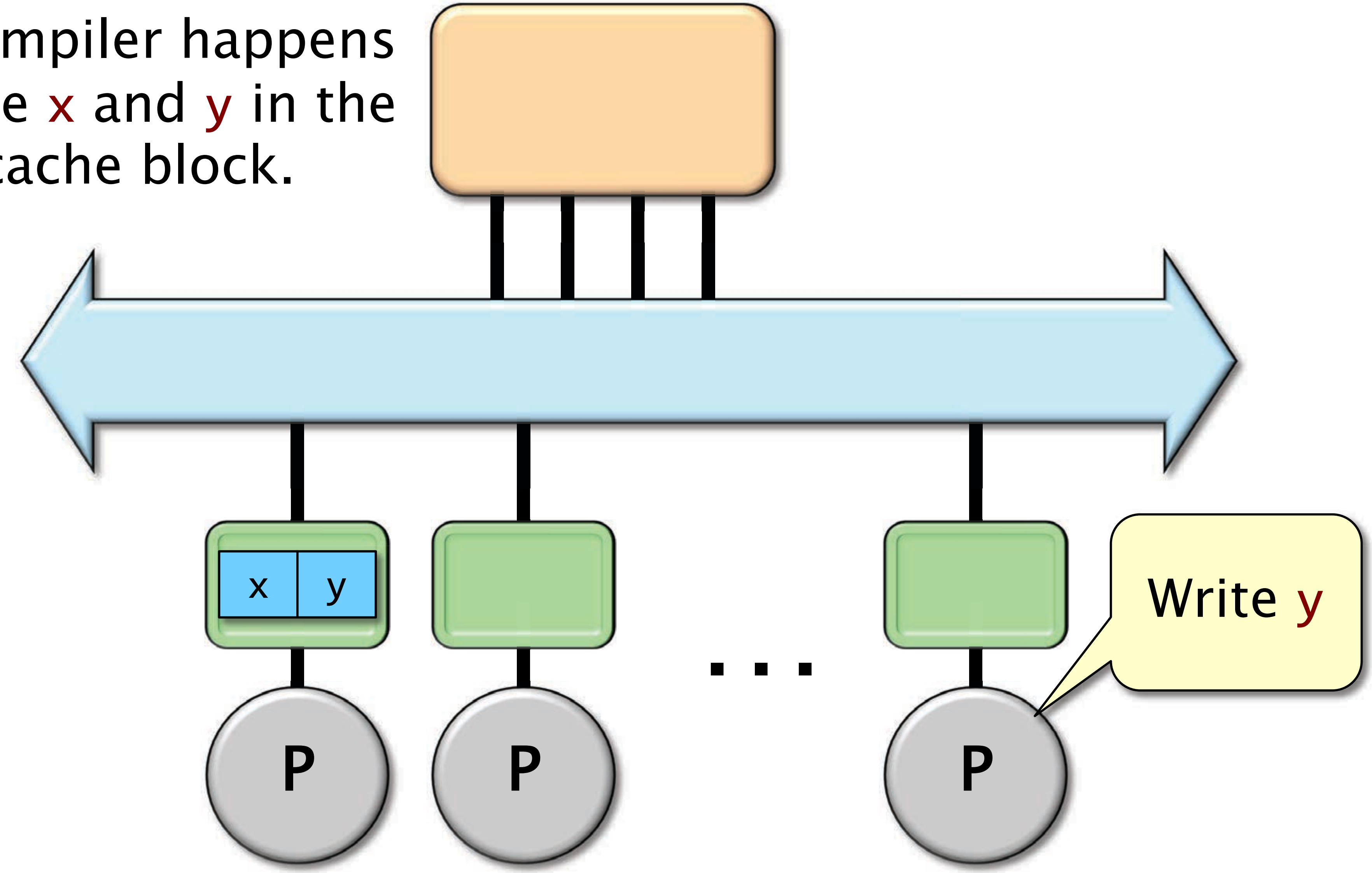
False Sharing Example

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



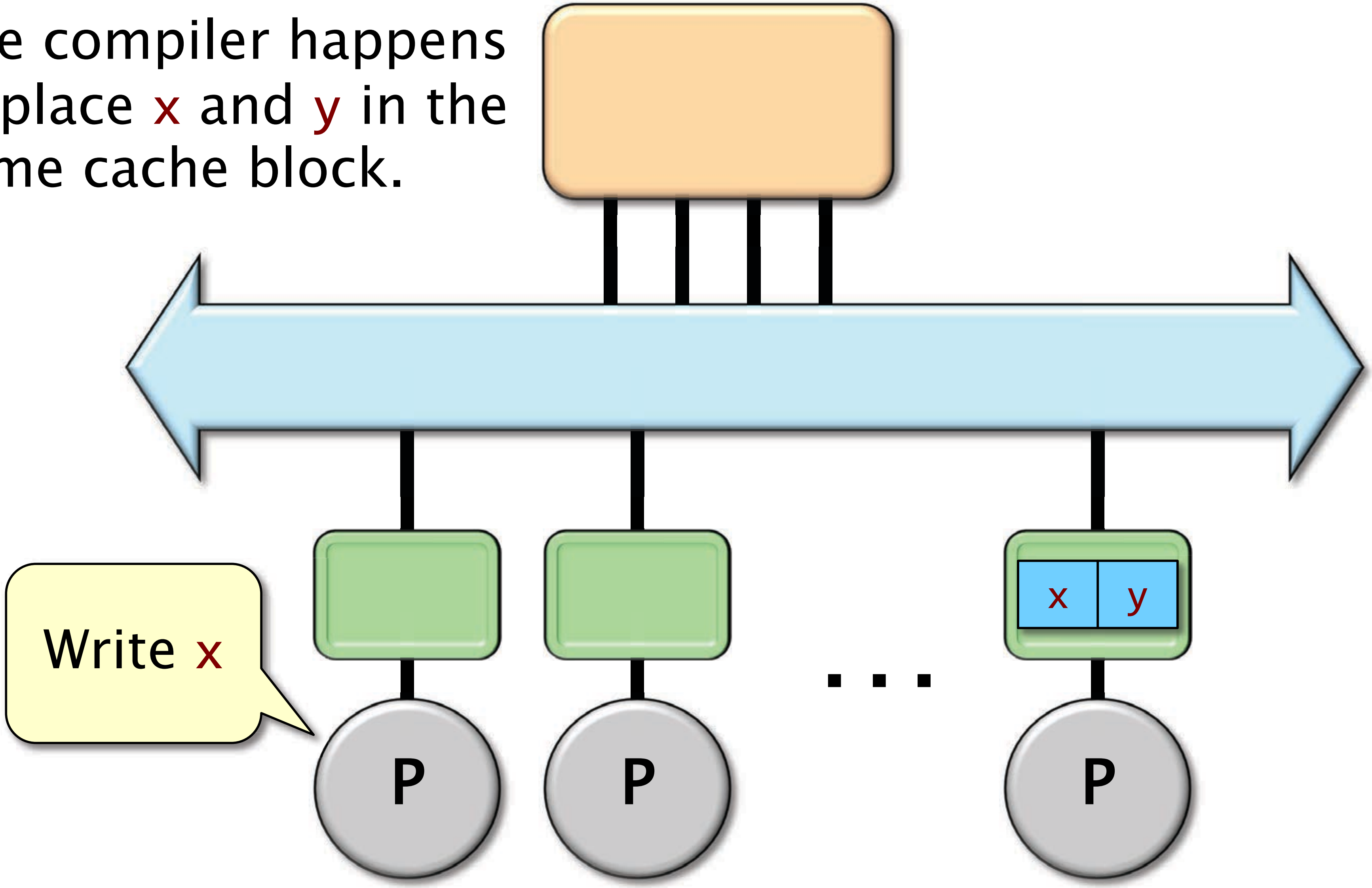
False Sharing Example

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



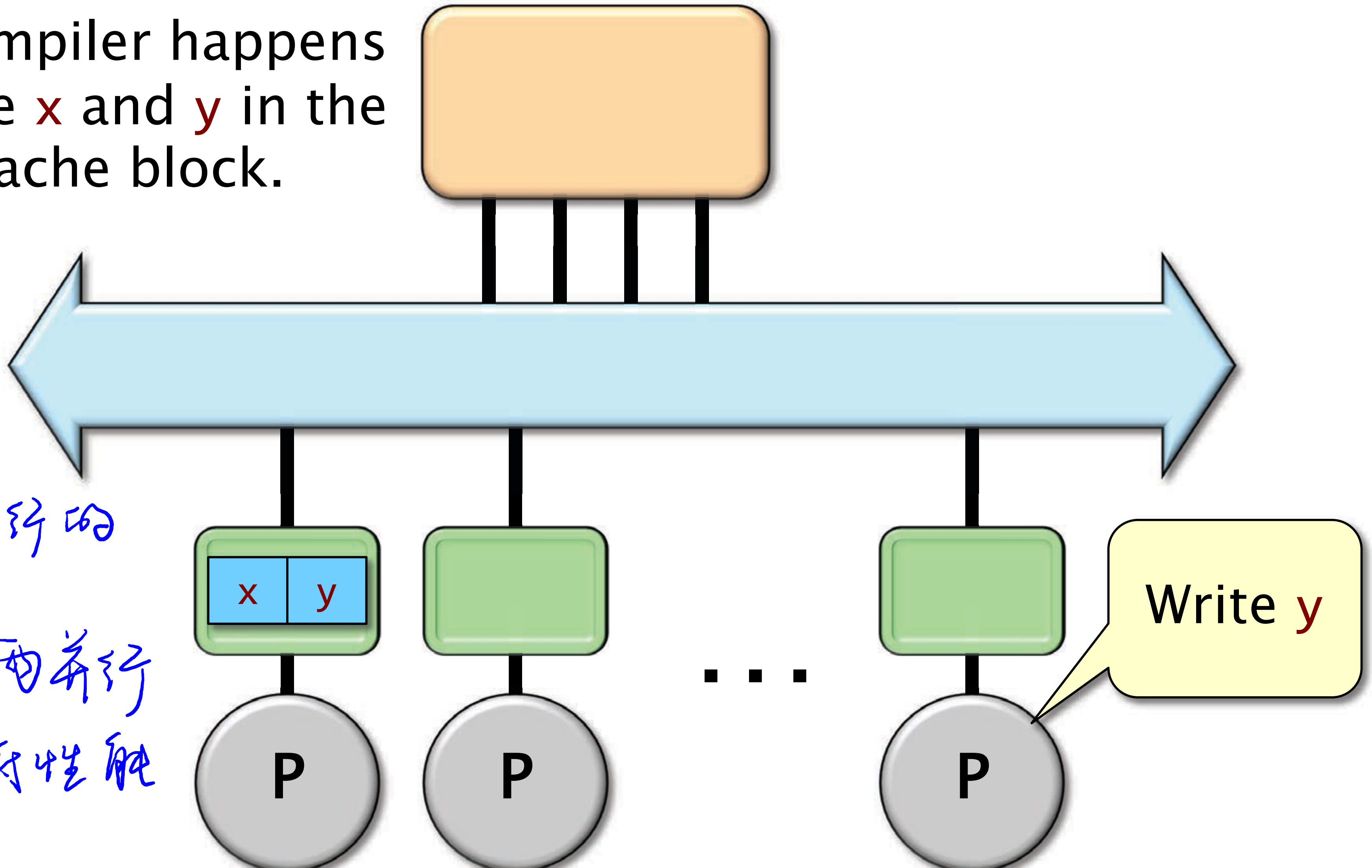
False Sharing Example

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.



Cache是用行的

这样这两并行
会降低性能

How False Sharing Can Occur

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

对象对齐
cache line
浪费

- 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:

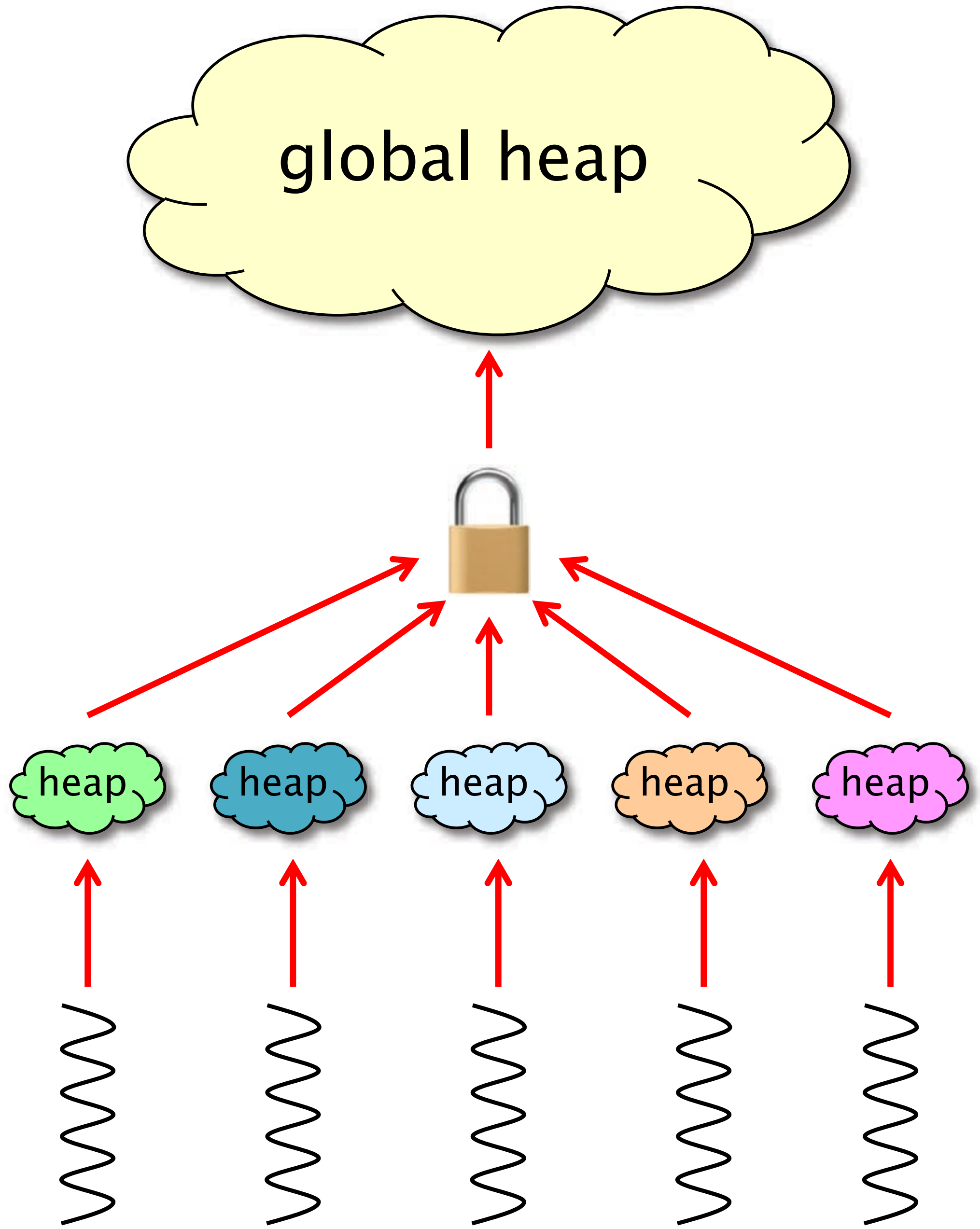
- Actively, when the allocator satisfies memory requests from different threads using the same cache block. 不同线程用同一缓存块
- Passively, when the program passes objects lying on the same cache line to different threads, and the allocator reuses the objects' storage after the objects are freed to satisfy requests from those threads.

BACK TO PARALLEL HEAP ALLOCATION

The Hoard Allocator

- 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.



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 u_i be the in-use storage in heap i, and let a_i be the storage owned by heap i. Hoard maintains the following invariant for all heaps i:

$$u_i \geq \min(a_i - 2S, a_i/2),$$

where S is the superblock size.

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

```
put x back in heap i;
if (u_i < min(a_i - 2S, a_i/2)) {
  move a superblock that is at least 1/2 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 U be the user footprint for a program, and let A be Hoard's allocator footprint. We have

$$A \leq O(U + SP) ,$$

and hence the blowup is

$$A/U = O(1 + SP/U) . \blacksquare$$

Proof. Analyze storage in local heaps.
Recall that $u_i \geq \min(a_i - 2S, a_i/2)$.
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 $O(U)$. $\blacksquare$

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.

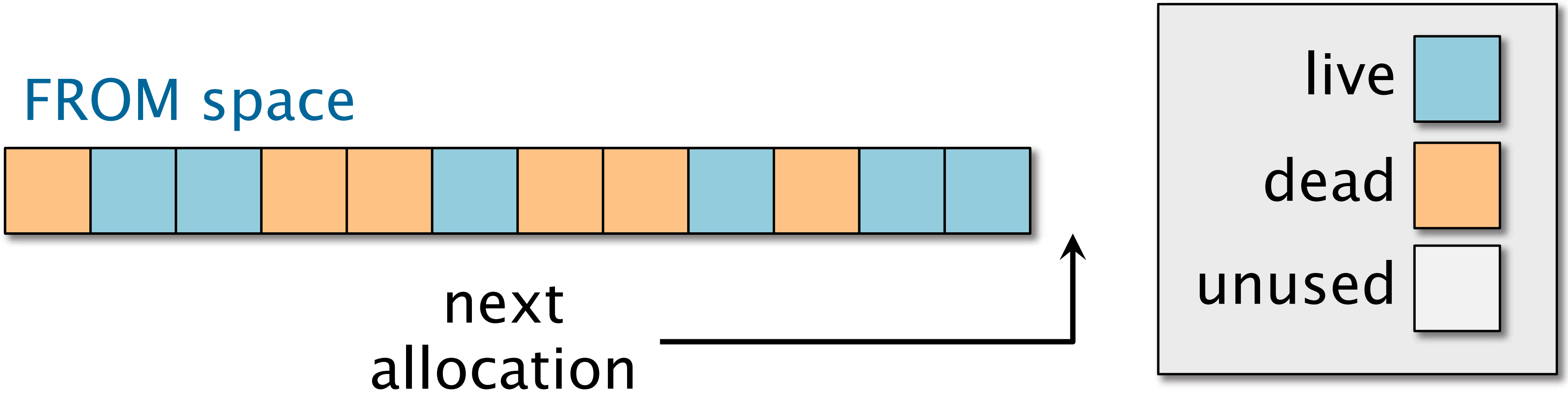
SuperMalloc is an up-and-coming contender. (See paper by Bradley C. Kuszmaul.)

Allocator Speeds

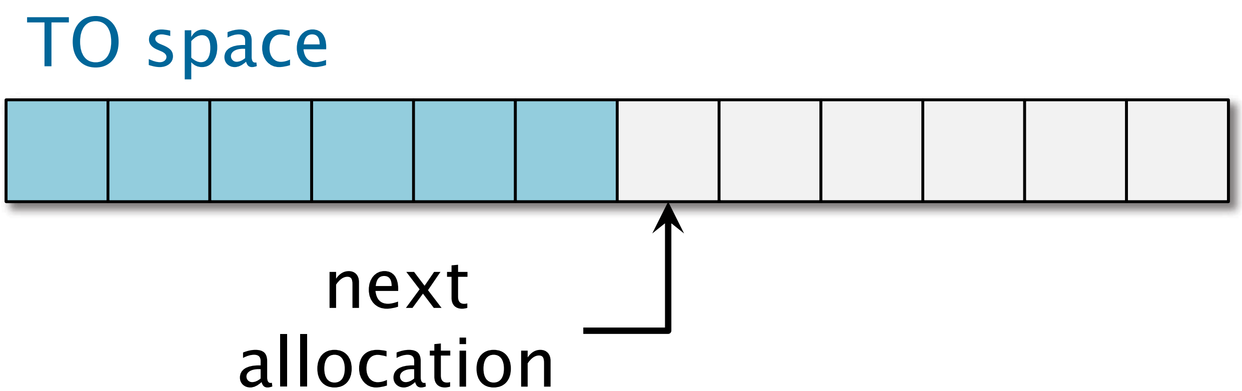
Allocator	SLOC	32 threads
Default	6,281	0.97 M/s
Hoard	16,948	17.1 M/s
jemalloc	22,230	38.2 M/s
SuperMalloc	3,571	131.7 M/s

Garbage Collection

Copying Garbage Collector



When the FROM space is "full," copy live storage using BFS with the TO space as the FIFO queue.

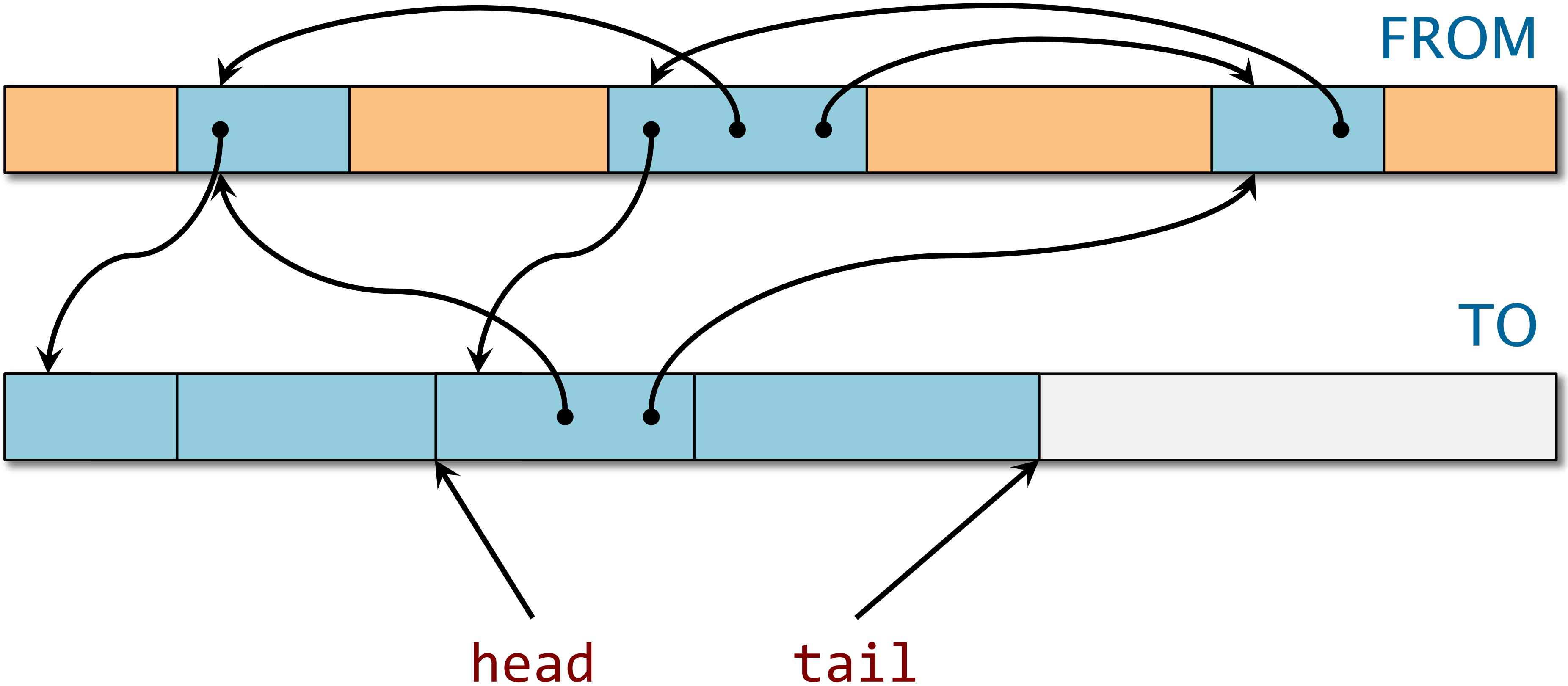


Updating Pointers

Since the FROM address of an object is not generally equal to the TO address of the object, pointers must be updated.

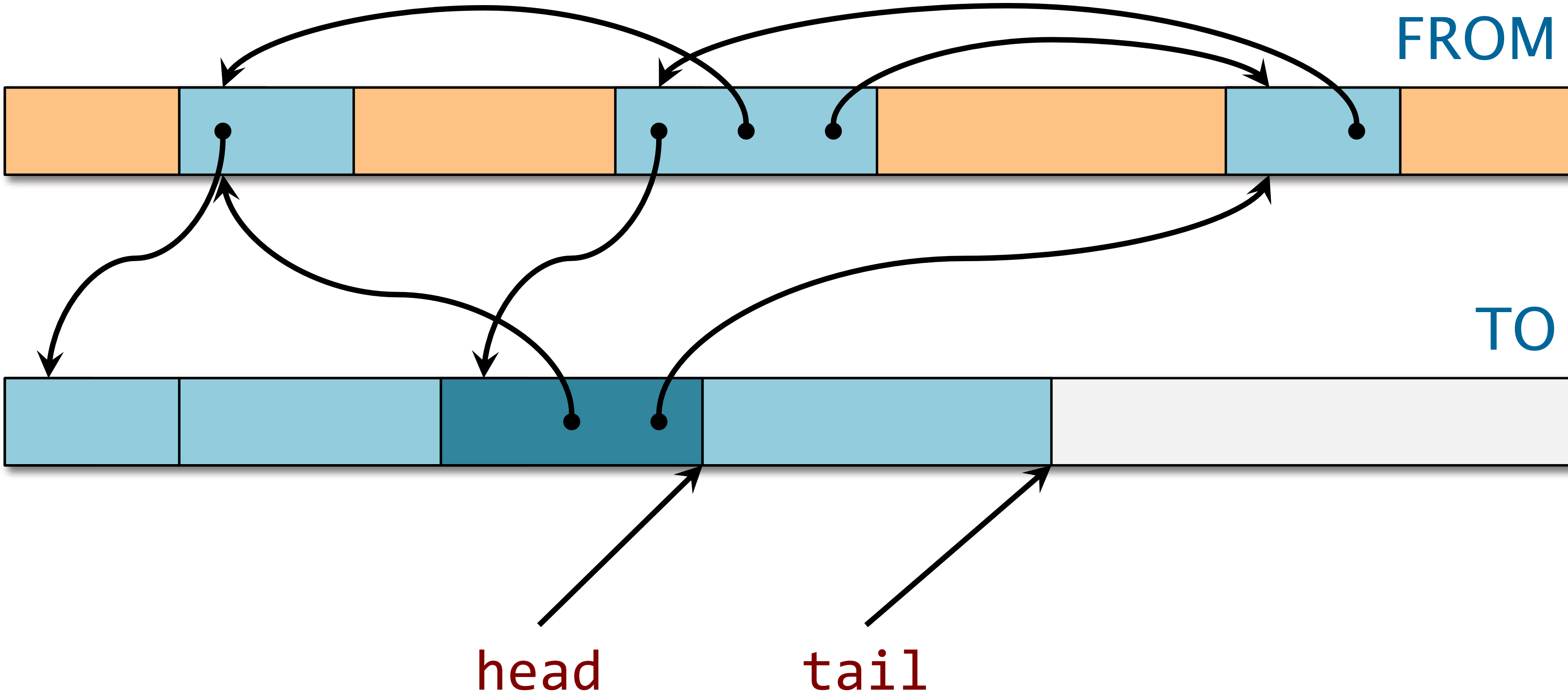
- When an object is copied to the TO space, store a forwarding pointer in the FROM object, which implicitly marks it as moved.
- When an object is removed from the FIFO queue in the TO space, update all its pointers.

Example



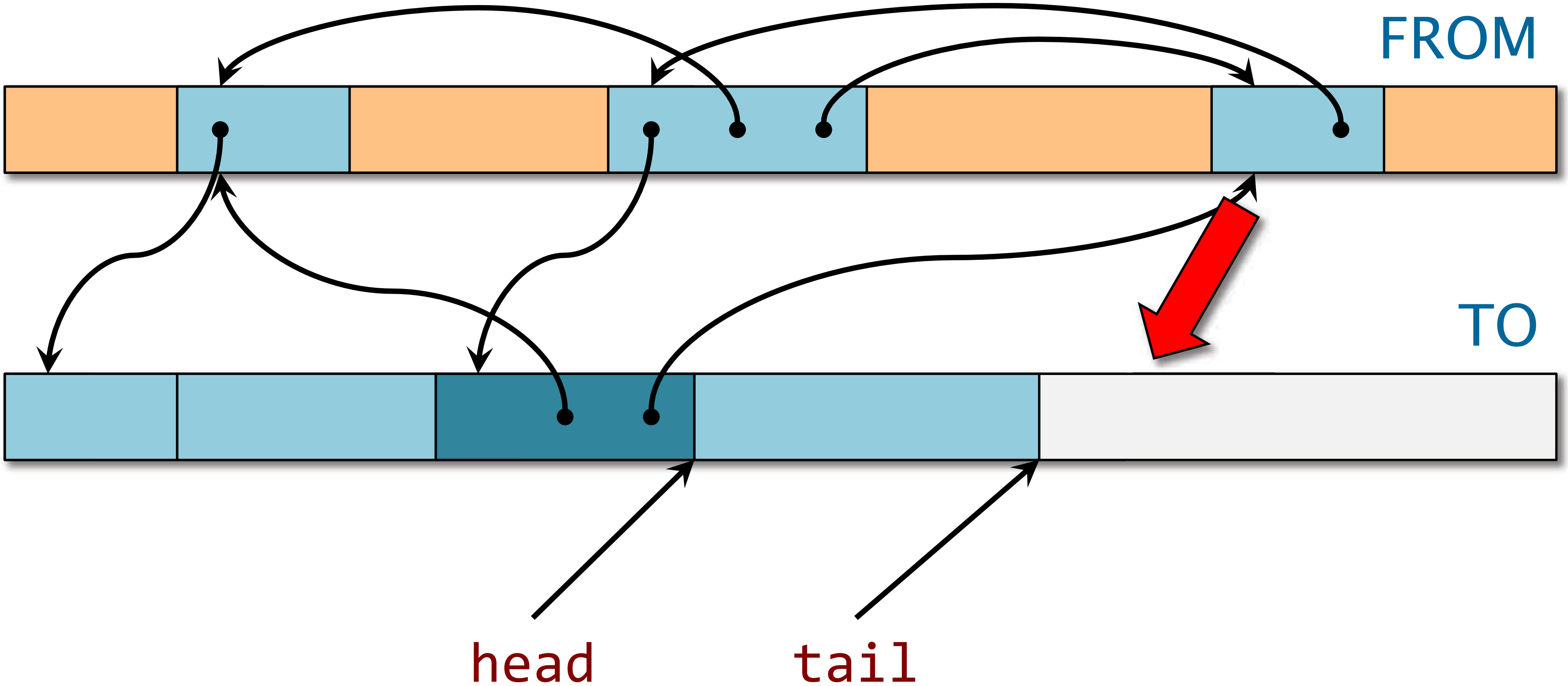
Remove an item from the queue.

Example



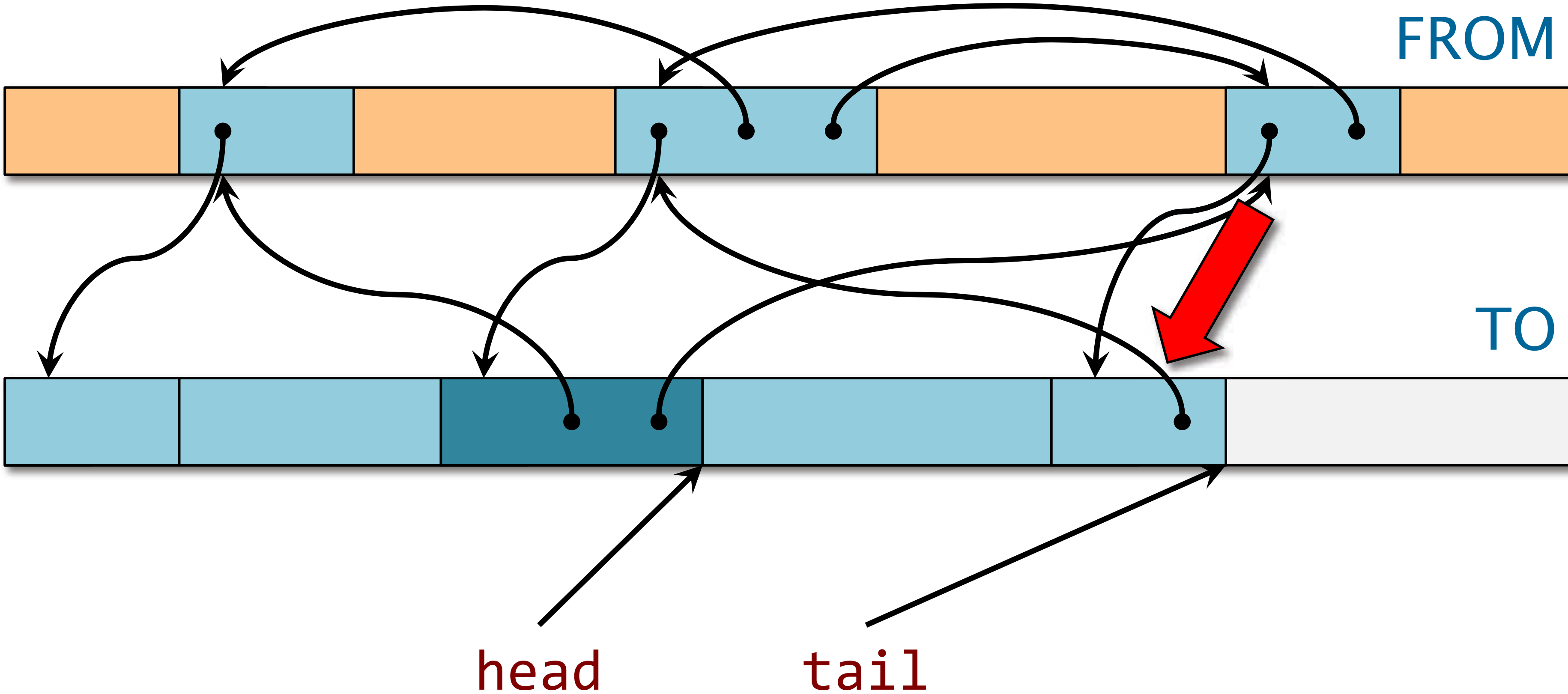
Remove an item from the queue.

Example



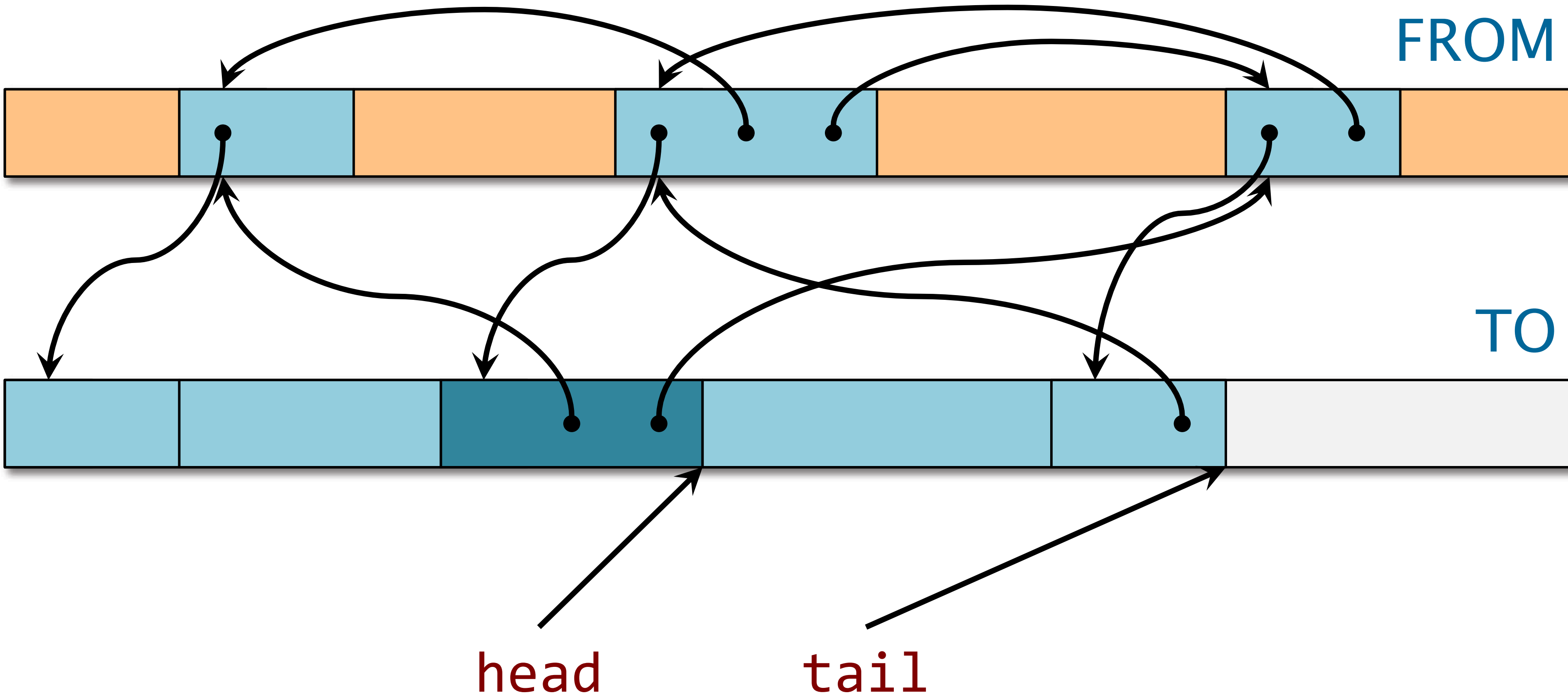
Enqueue adjacent vertices.

Example



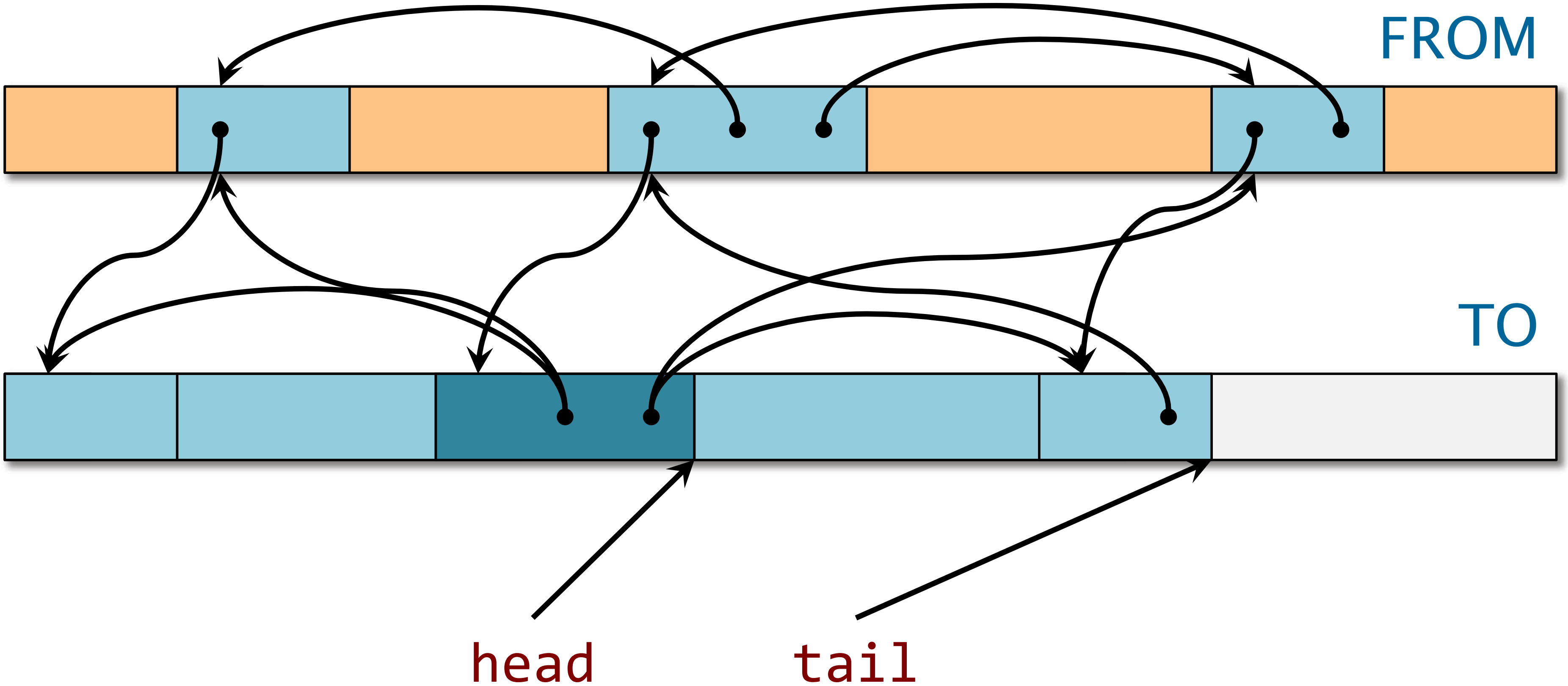
Enqueue adjacent vertices.
Place forwarding pointers in FROM vertices.

Example



Update the pointers in the removed item to refer to its adjacent items in the TO space.

Example



Update the pointers in the removed item to refer to its adjacent items in the TO space.

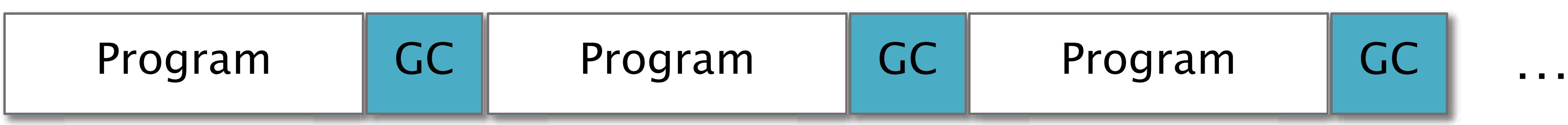
Types of Garbage Collectors

Stop–the–world collector 大的停止来清理

- Program pauses once in a while and garbage collector (GC) does work across all of memory
- High program pause times

Incremental collector 小清

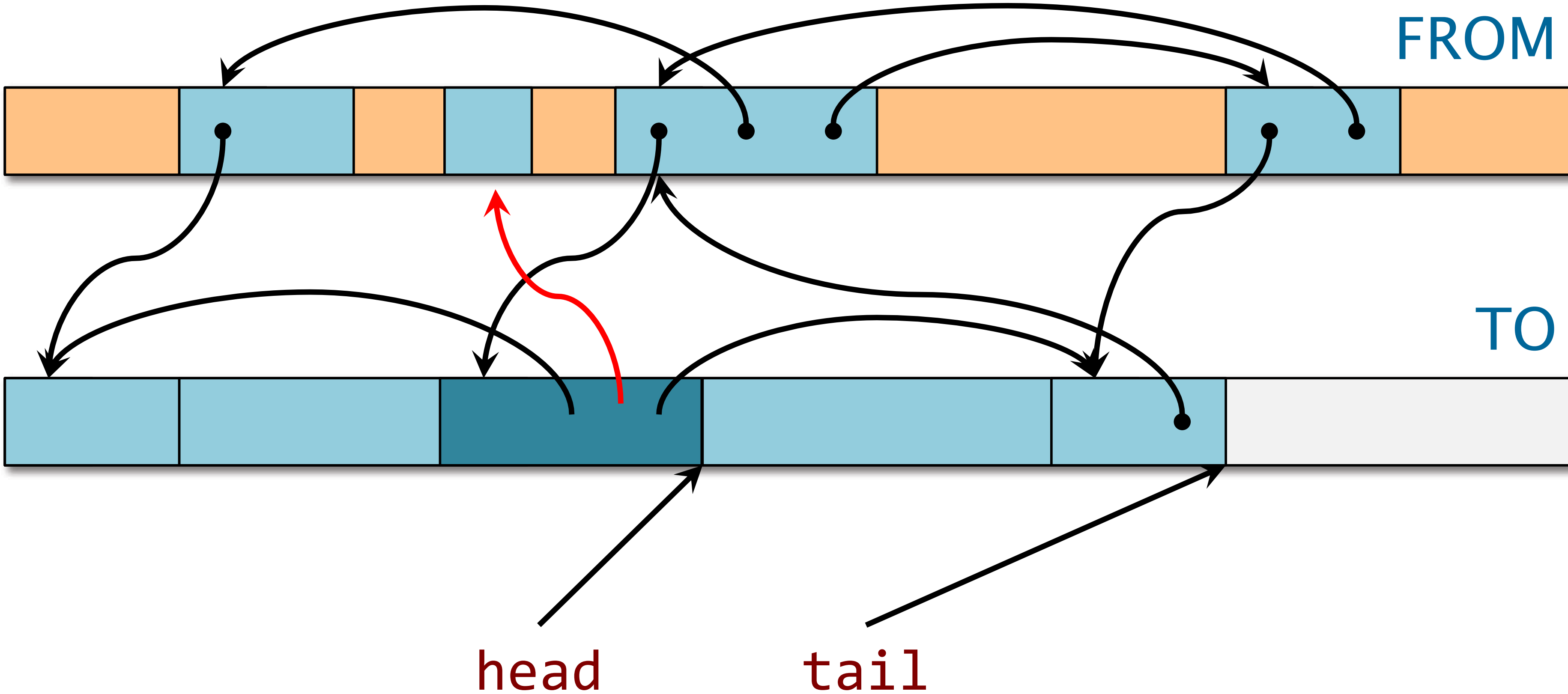
- Collector cleans up a small part of memory every time it executes
- Low program pause times



Running Collector with Program

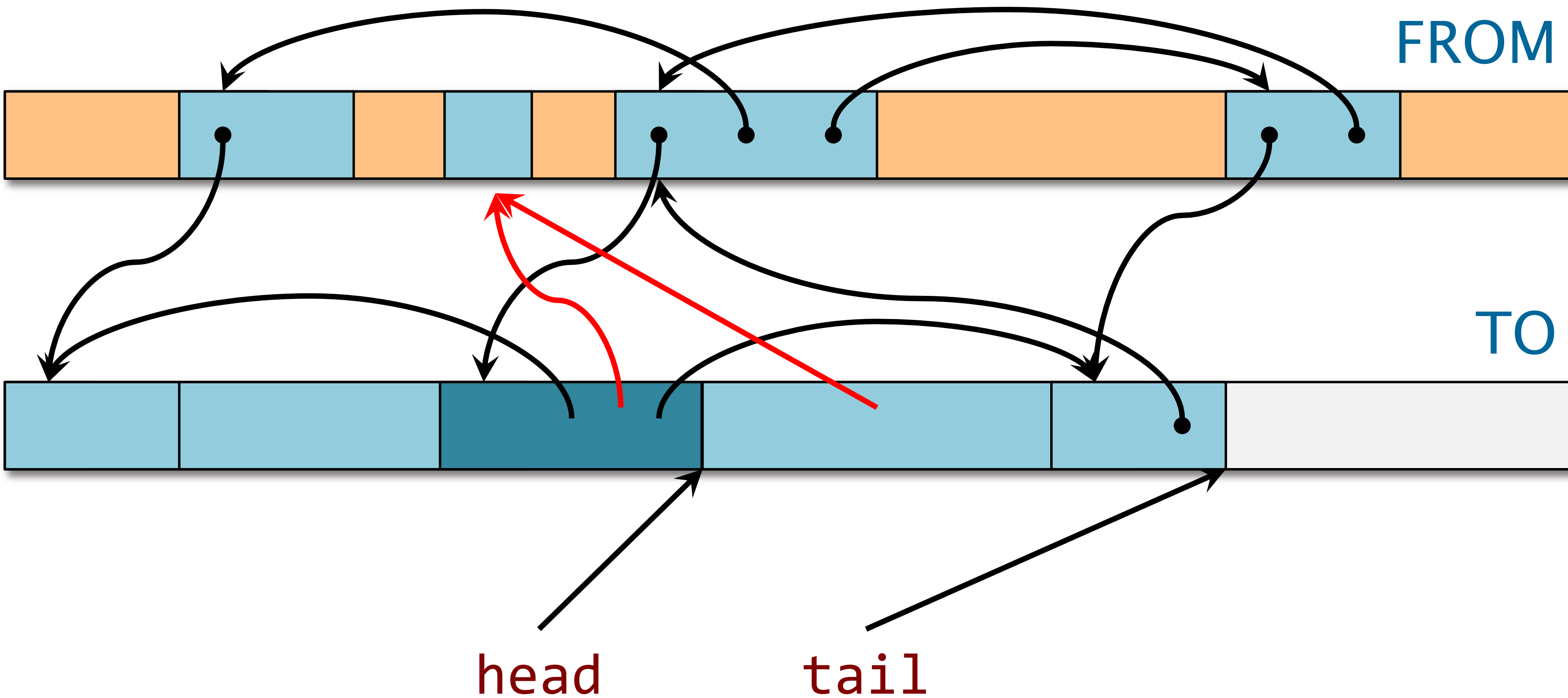
- Incremental version of copying collector.
- When it is time to collect, application program and garbage collector take turns running.

Running Collector with Program



If an object O already dequeued in BFS gains a reference to another object O', the BFS may not find O' and it will be freed.

Running Collector with Program



If an object O already dequeued in BFS gains a reference to another object O', the BFS may not find O' and it will be freed.

Baker's Algorithm

- Program follows forward pointer if there is one.
- Whenever the program accesses an object not in the TO space, mark object as explored and copy it over to the TO space.
- Whenever the program allocates an object, put it in the TO space.
- Requires a read barrier to intercept every read with a check, which is expensive.
- This algorithm is conservative in that it does not necessarily collect all garbage. Why?

Nettles-O'Toole Algorithm

- Program works only in FROM space until garbage collection is finished.
- Replicates the objects by keeping mutations to FROM-space objects in a log.
- Garbage collector applies the mutations to corresponding TO-space objects.
- Requires a write barrier to log mutations on every write
  - This is expensive, but writes are usually much less frequent than reads.
- Is this algorithm conservative?

Garbage Collection Glossary

- Stop-the-world: Garbage collector does all of its work across memory while pausing program.
- Incremental: Garbage collector runs incrementally, allowing pause times to be bounded.
- Parallel: Multiple collector threads are running simultaneously.
- Concurrent: At least one program thread and one collector thread are running simultaneously.

Parallel and Concurrent GC

- Based on Nettles-O'Toole algorithm
- High-level idea
  - Use per-processor local stacks for search
  - Maintain a shared stack for load balancing
    - Processors periodically transfer objects between local and shared stack
  - Use synchronization primitives (test-and-set and fetch-and-add) to manage concurrent accesses

See "On Bounding Time and Space for Multiprocessor Garbage Collection" (PLDI 1999), and "A Parallel, Real-Time Garbage Collector" (PLDI 2001) by Cheng and Blelloch

Summary

- malloc() vs. mmap()
- Cactus stacks
- Cilk space bound of $S_P \le P\,S_1$ and better bound for matrix multiply
- Parallel allocation strategies: global heap, local heaps, local ownership
- Incremental garbage collection
- Parallel and concurrent garbage collection