

Algorithms for Dynamic Memory Management (236780)

Lecture 12

Lecturer: Erez Petrank



Last Week

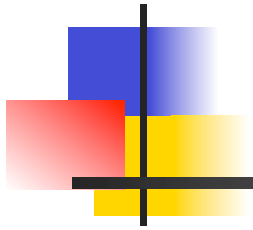
- Compressor
- Cache consciousness



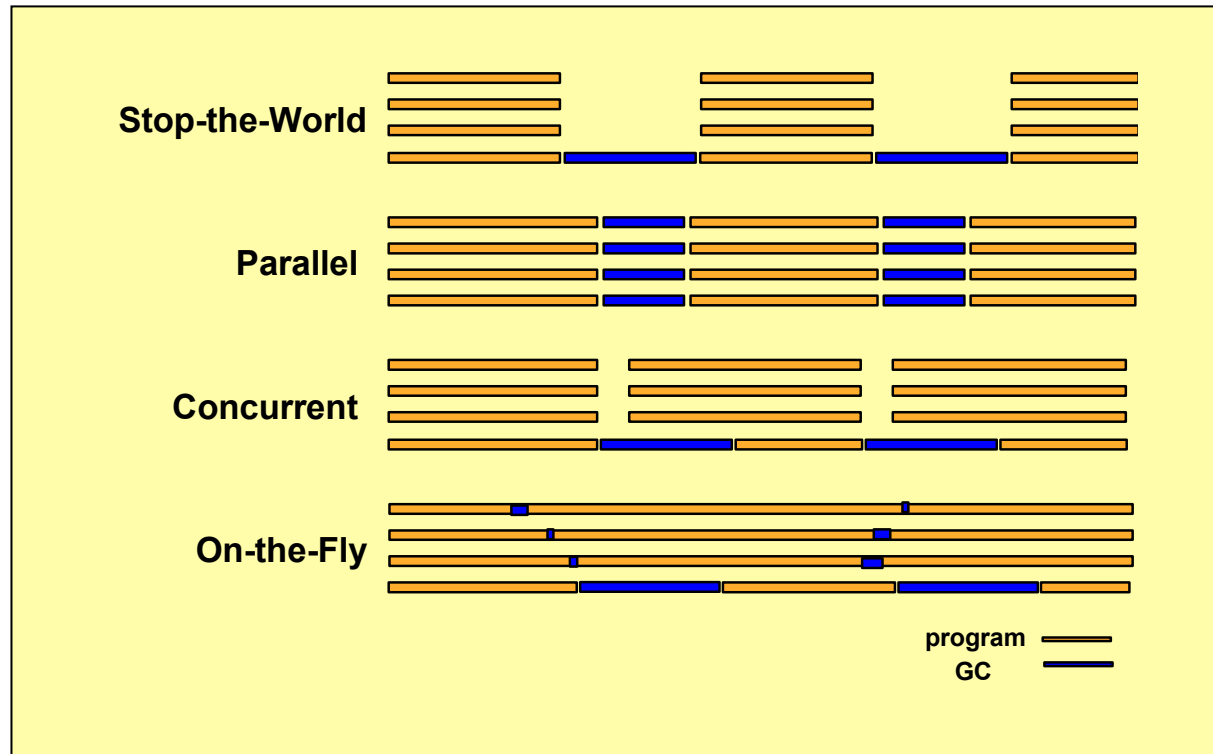
Topics Today

- Part I: parallel GC
- Part II: Allocation techniques

Parallel Garbage Collection



Recall Terminology

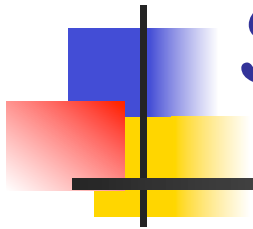




Plan

- Part 1: Parallel Mark & Sweep
- Part 2: Parallel Copying
- **Bibliography:**
 - [Endo-Taura-Yonezawa 1997] "A Scalable Mark-Sweep Garbage Collector on Large-Scale Shared-Memory Machines"
 - [Kolodner-Petrank 1999] "Parallel Copying Garbage Collection using Delayed Allocation"
 - [Flood-Detlefs-Shavit-Zhang 2001] "Parallel garbage collection for shared memory multiprocessors"

Part I: Parallel Mark & Sweep





Motivation

- Concurrent GC is not scalable enough
 - One collector cannot compete with 31 allocating mutators.
 - Parallel collectors do not compete with program, but need to cooperate work on shared resources.
- No previous work (at 1997) provided measurements.
- In this work a 64-way SMP is used.
- Speedup reported.



Design Considerations

- **Goals:**
 - Scalability
 - Load balancing
 - Locality of reference
 - Simplicity
- **Main problem:**
 - The work cannot be partitioned statically, we must allow the collector to balance work.
 - Idea: overpartition the work.



Base Collector: Boehm GC

- Boehm GC: a C/C++ library for mark-sweep conservative GC.
- Allocations (via gcalloc)
 - Allocations are serialized.
 - Heap divided into 4KB blocks which are initially in a free list.
 - Large objects are allocated from a free list sorted by address.
 - Small objects are allocated from within blocks.
 - Each block contains objects of the same size
- Download: http://www.hpl.hp.com/personal/Hans_Boehm/gc .



Base Collector: Boehm GC

- Every block has a separate mark bitmap (ratio 1:32). Marking requires sync.
- One global markstack requiring sync as well.
- DFS style marking from roots.
- Sweep:
 - If a block is fully empty, then it is returned to free list of blocks. Adjacent blocks are coalesced,
 - If mark bit is set on a heap block, block is kept for future allocations.

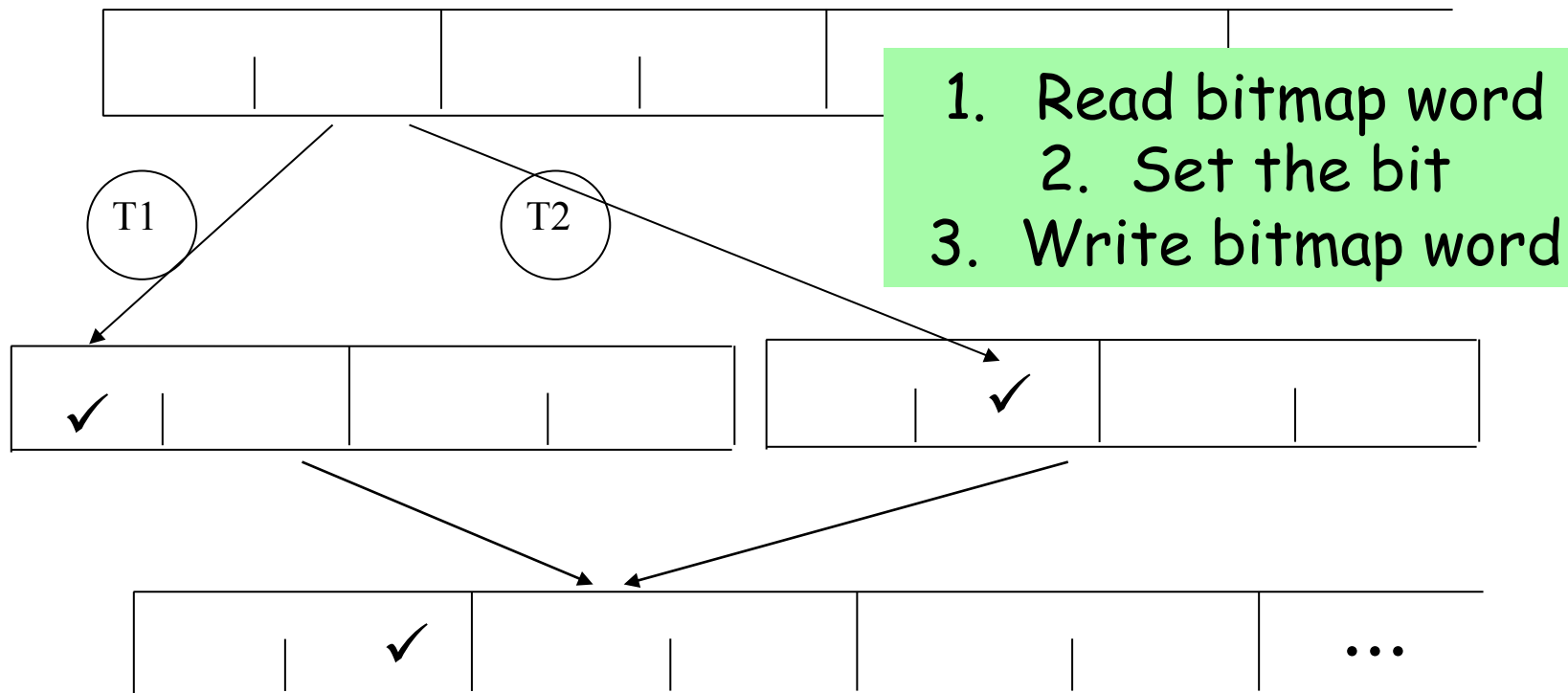


Parallel Collector

- Each collector runs on a separate process with a local markstack.
- When *GC* invoked, all program processes are stopped via a signal.
- Each process marks its local roots and then proceeds marking via its local markstack.
- Marking requires synchronization.

Why Locking of Bitmap is Required

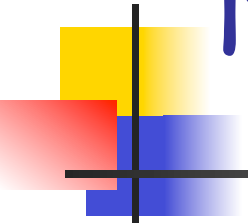
Two threads mark two objects concurrently.
Both objects are mapped to the same bitmap word.





Parallel Sweeping

- Each process assigned part of the heap (64 blocks at a time).
 - All blocks that have **not** become empty are put in local free lists (cheap)
 - All free blocks are first processed locally (sorting, coalescing). Then, lock is taken and they are put in global free list.

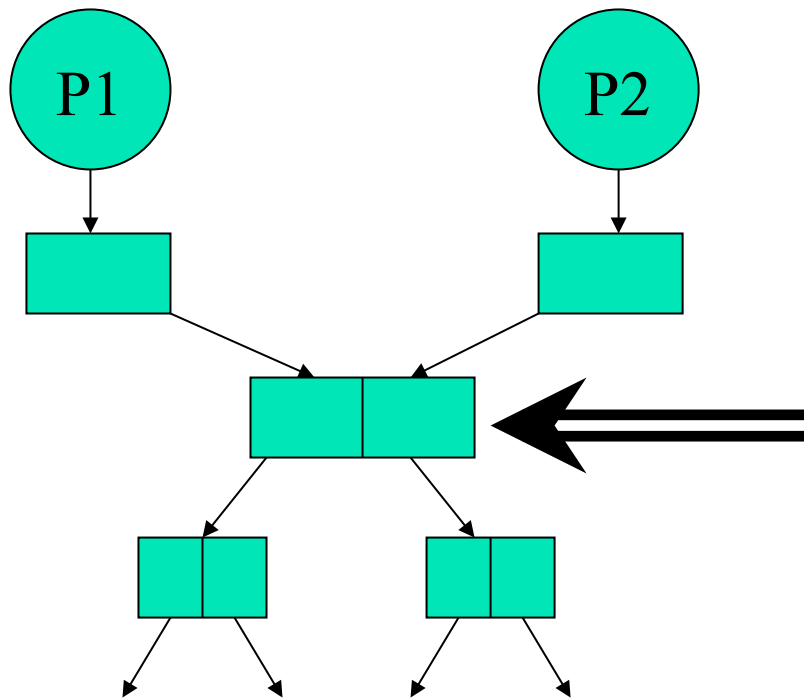


Naive Implementation

- This naive implementation results in hardly any speedup.
- Thus, improvements required.

Introducing Load Balancing

- Problem: Consider the following shared tree:



The process that first marks the tree's root will have to mark the rest of the tree!



Stealable Mark Queues

- Each processor keeps a "stealable mark queue".
- Once in a while, if stealable mark queue empty, move $\frac{1}{2}$ of the jobs from markstack to stealable mark queue.
- If processor idle - search for jobs in stealable mark queues. (Steal $\frac{1}{2}$ of a s.m.q.)
 - ✓ Idle processors help the busy ones
 - ✗ Sync on stealable mark queues
 - ✗ Tougher termination detection



Termination

- Keep a global counter with the number of empty stacks and empty queues.
- Counter is updated whenever a processor fills its mark queue, becomes idle, or obtains a task.
- When counter reaches twice the number of processors - GC ends.



Empirical Evaluation

- With stealable mark queues, the algorithm exhibits at most 12x speed-up on a 64-way SMP.
- Next, 4 improvements.



1: Split Large Objects

- A process that must mark a large object (e.g. 400KB) is tied up for a long time (load imbalance)
- Solution: Break objects into 512-byte chunks before inserting in mark stack



2: Skip Locked Queues

- Sometimes many processes attempt to steal from the same queue and must wait to enter the critical section (contention)
- Solution: If lock can't be acquired on first try, give up and go to the next queue.



3: Markbits Test

- Sync. (lock) is used to mark objects
- However, many times the object is already marked !
- **Improvement:** Read the bit first without sync. If not set, use sync to set it.



4: Improve Termination Detect

- Global counter was used to keep track of empty mark and steal queues (**contention**)
- **Improvement:** "Mark Stack Empty" and "Steal Stack Empty" flags kept on each processor. Terminate if all flags are set and global "interrupted" flag is clear.



Empirical Evaluation

- Desired: run various number of processors on the same snapshot of the heap.
- Problem: snapshot depends on number of processors. Not clear how to continue with varying number of processors from the same snapshot.
- Approach: Devise a formula for workload and compare workload/time ratios.



Workload and Speed-up

- Workload of a collection is
$$W = a_1x_1 + a_2x_2 + a_3x_3 + a_4x_4 + a_5x_5$$
 - x1= no. marked objects marked and scanned
 - x2=no. visits to already marked obj's
 - x3=no. visits to objects with no pointers
 - x4=no. empty heap blocks
 - x5=no. non-empty blocks
- Weights were determined by experiment
 - a1=0.50, a2=0.16, a3=0.02, a4=2.0, a5=1.3
- Speed=W/t, (for t = GC elapsed time)
- Speed-up on N proc's = $(W_n/t_n)/(W_1/t_1)$



Benchmark Apps

- BH -- simulates motion of N moving bodies (here N = 5000)
- CKY -- context-free grammar parser (run on 67 sentences, 10-40 words per sentence).

Mark Speed-up Results

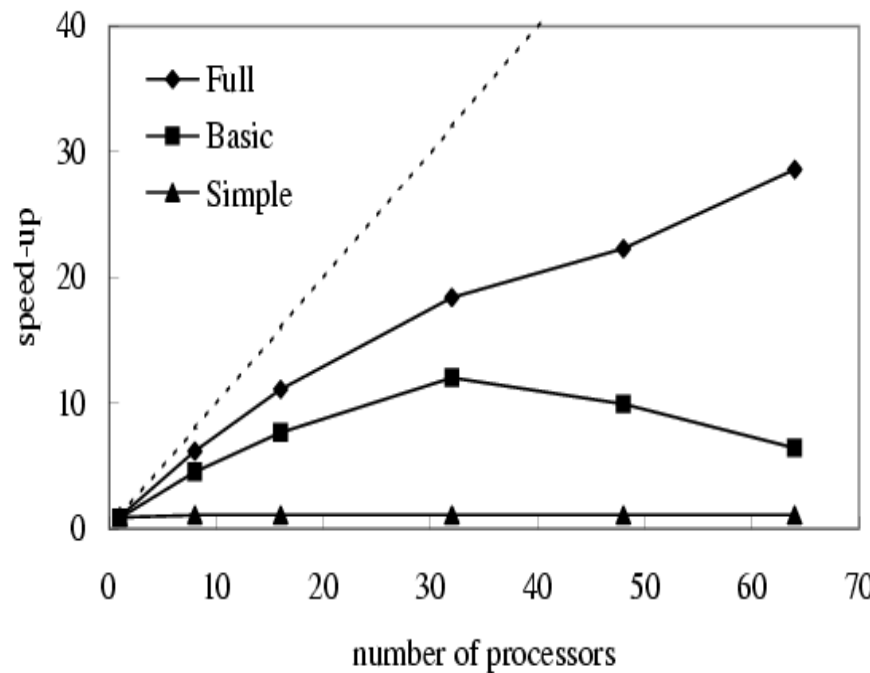


Figure 5: Average marking speed-up in CKY.

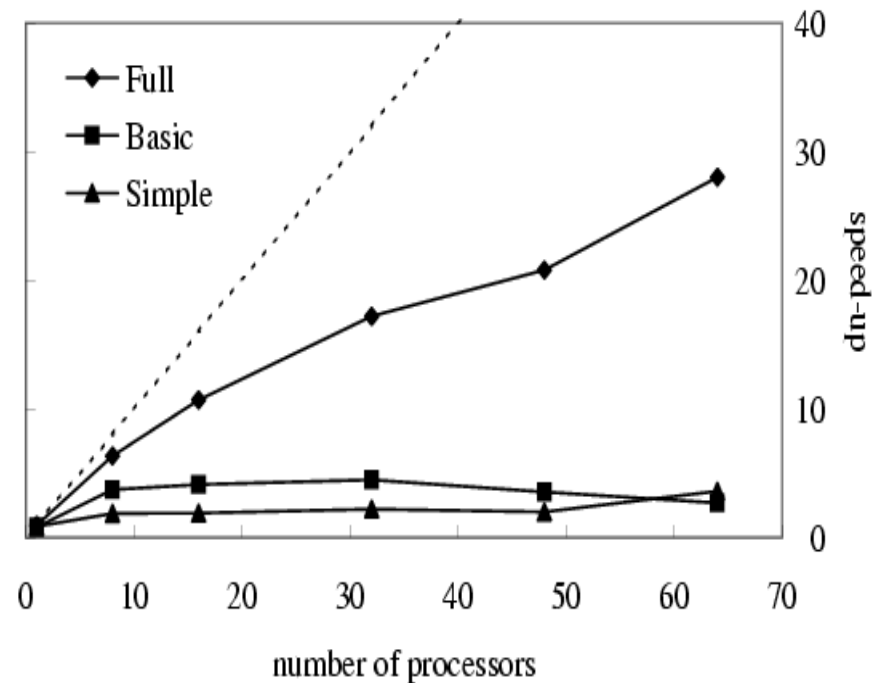


Figure 6: Average marking speed-up in BH.

Indiv. Improvement Effect

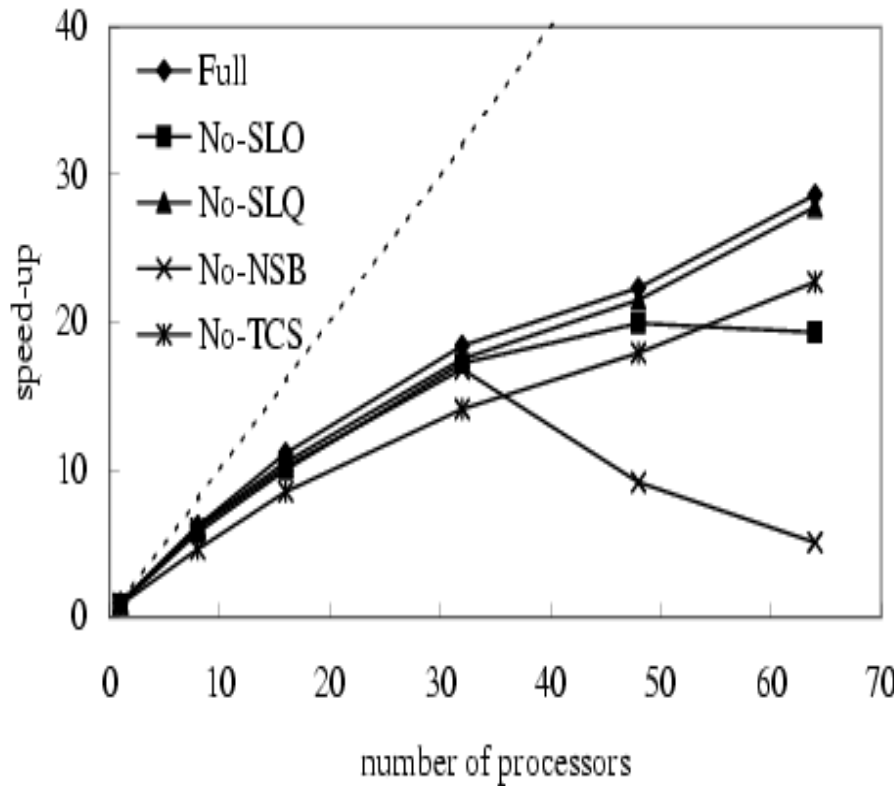


Figure 7: Effect of each optimization in CKY.

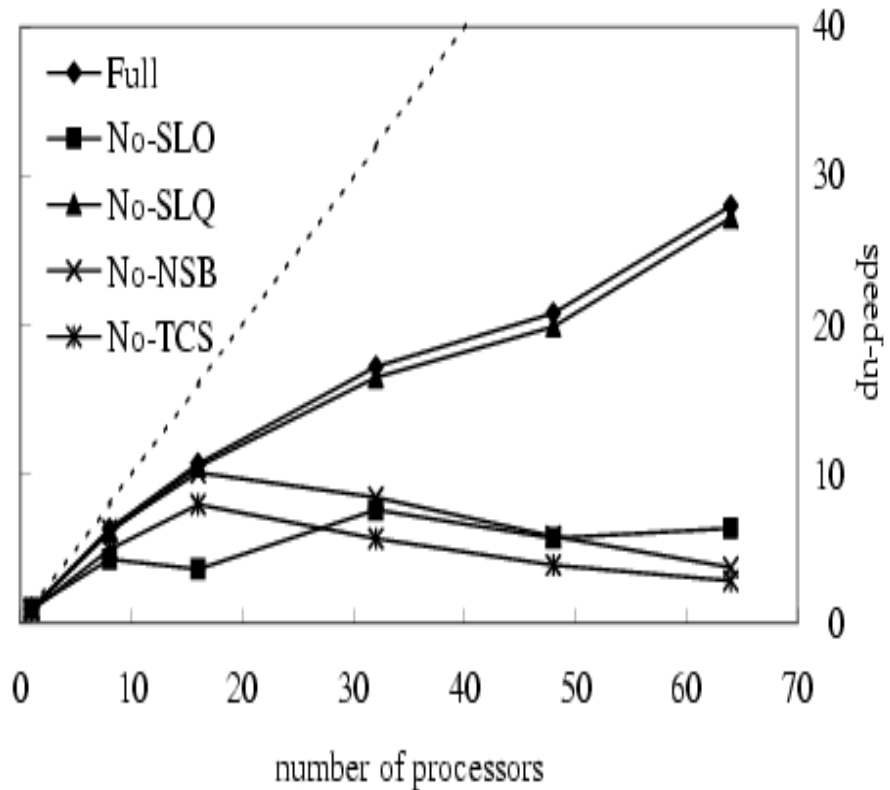


Figure 8: Effect of each optimization in BH.

No-SLO = w/o splitting large objects, No-SLQ = w/o non-blocking mark queues search, No-NSB = w/o improved termination detection, No-TCS = w/o non-blocking bitmap reads



Summary

- Parallelizing a mark-and-sweep collector is possible.
- With the ideas raised in the paper the authors got speed-up around 30 with 64 processors. (Quite good!)

Parallel Collection for a Copying Collector



Halstead 84

Imai-Tick 91

Kolodner-Petrank 99

Flood-Detlefs-Shavit-Zhang 01



Copy Algorithm: Reminder

- Divides heap: *to-space* and *from-space*
- On GC, stop all threads, scan reachable objects and copy them into *to-space*



Main (new) Problem: Allocation

- Concurrent allocation in to-space (by several collectors):
- Option 1: (Halstead): Every processor will write in a separate memory area
 - Problem: Uneven allocations cause wasted space and fragmentation
- Option 2: Compete (via synchronization) on a shared free pointer in to-space.
 - Problem: [FDSZ] tried that and got too much contention.



Parallel Allocation (cont'd)

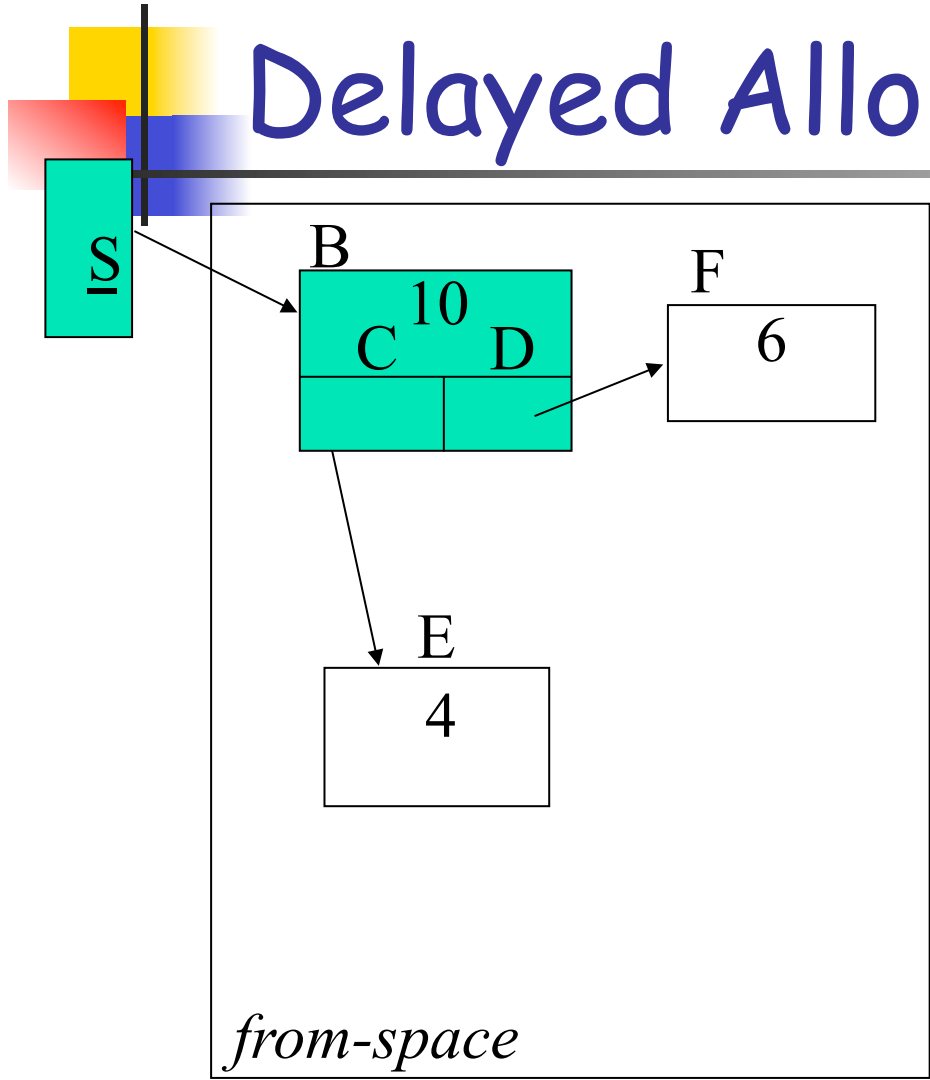
- Option 3: (Halstead): Allocate memory *blocks* to each processor; when filled, allocate another.
 - Problem: doesn't solve fragmentation
- Option 4 (Imai & Tick): Each processor has several blocks for allocations. Block i for allocations of sizes 2^{i-1} to 2^i .
 - Problem: Complicated block management, and doesn't completely solve fragmentation.



Parallel Allocation (cont'd)

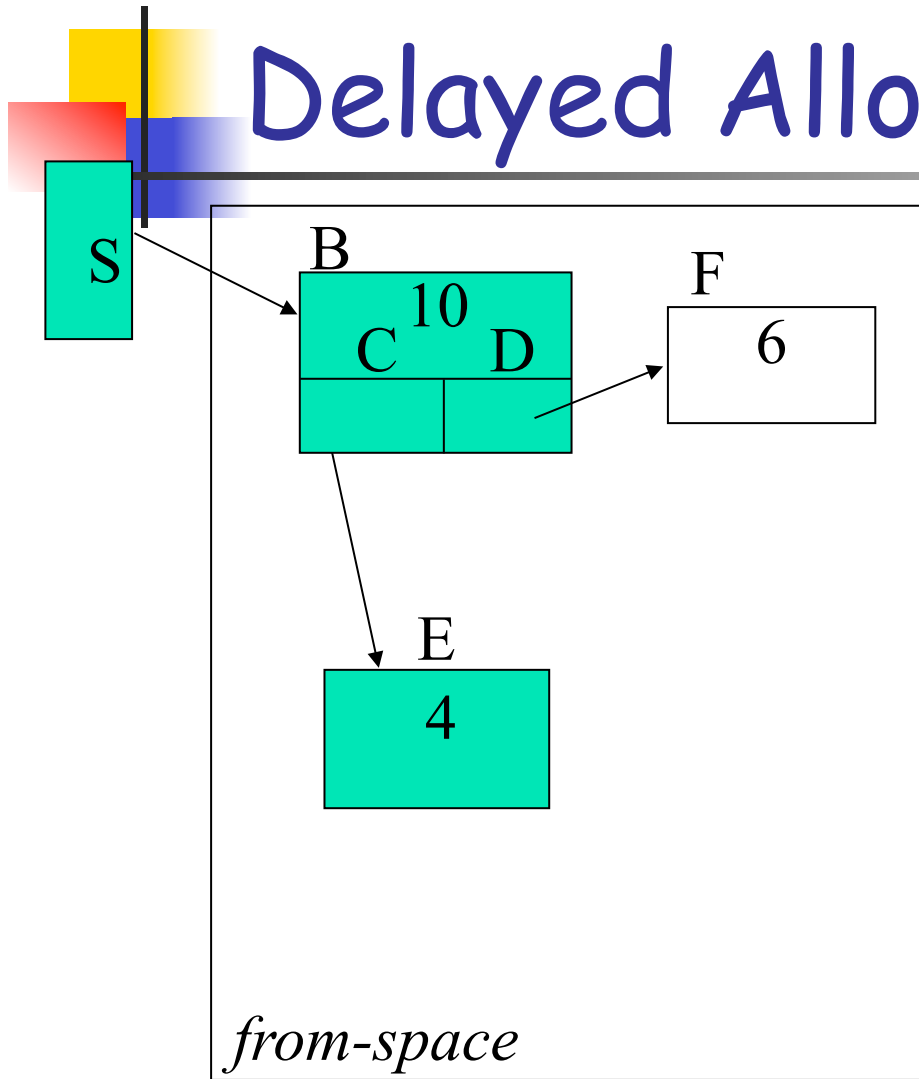
- Option 5 [KP]: *delayed allocation*:
 - Collectors do not really copy the objects.
 - Instead, they record intents to copy.
 - When the record is large enough, they allocate space and actually copy.
- No fragmentation
- Reduced synchronization.

Delayed Allocation



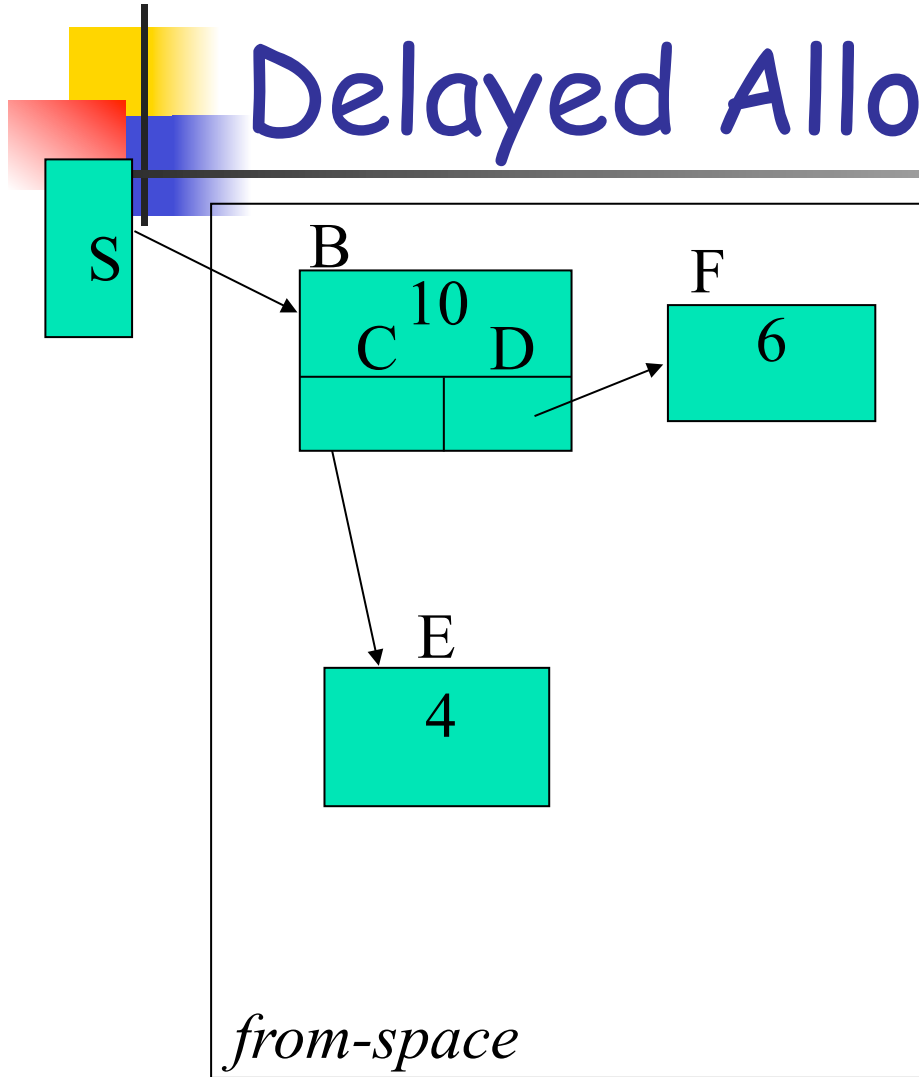
total	
10	
object addr	parent addr
B	S

Delayed Allocation



total	14
object addr	parent addr
B	S
E	C

Delayed Allocation



total	20
object addr	parent addr
B	S
E	C
F	D



Partitioning the work

- Option 1 [KP]: the space in to-space that requires scanning is partitioned to jobs.
- Option 2 [FDSZ]: work with a markstack that contains all objects to be scanned.
 - Now same solutions as with mark and sweep.
 - Use stealable mechanisms for load balancing.
- Option 3 (recall IBM's mostly concurrent work): work packets.



Issues

- What happens when other collectors access an object that is being recorded?
- Two flags:
 - Work flag - signifying object is being handled
 - Done flag - signifying object already copied
- When a collector gets to an object:
 - It competes on the work bit (cmp&swp)
 - If won - add object to log
 - If failed - don't wait!
Record parent in a special parents log.



Termination Detection is Flawed

mse

sqe

mse

sqe

mse

sqe

mse

sqe

mse

sqe

interrupted

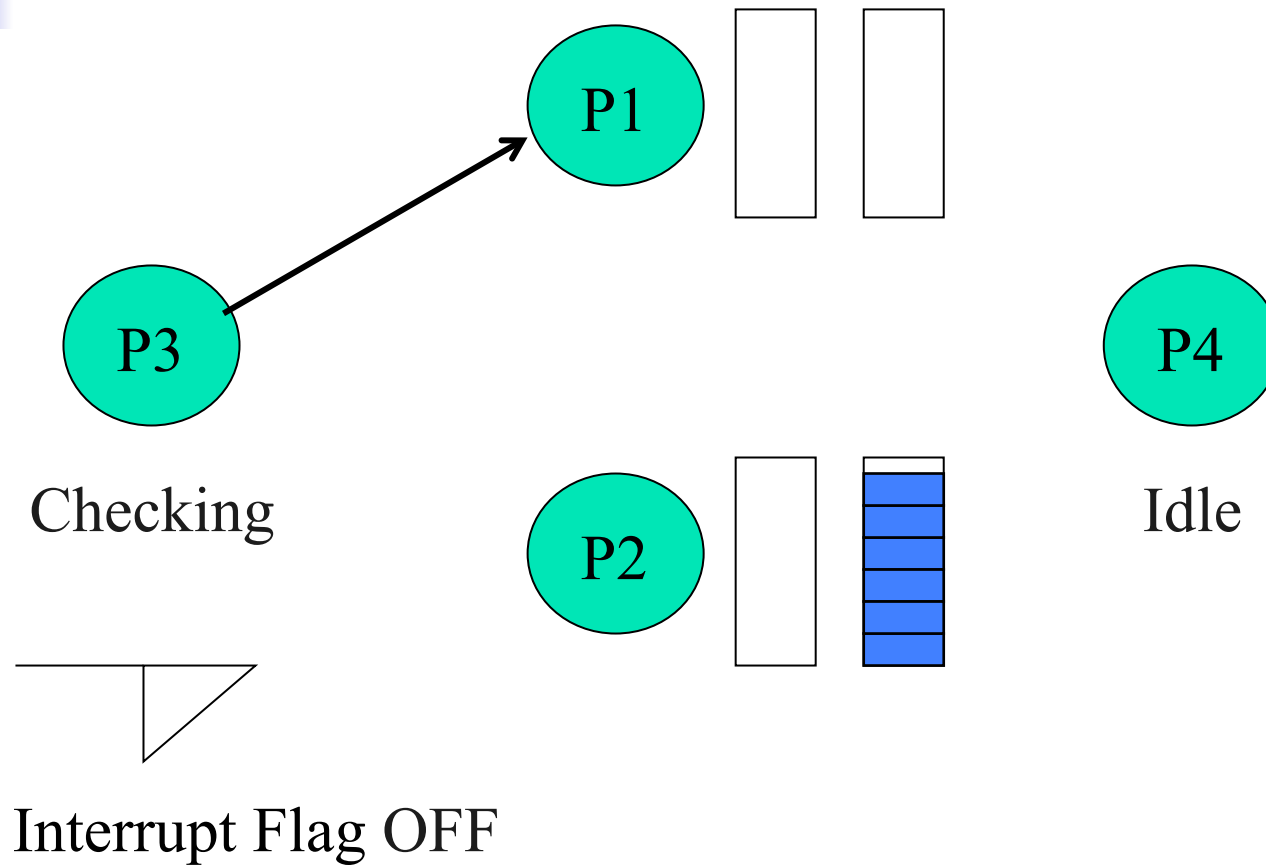
- Recall algorithm:
- Flags "Mark Stack Empty" & "Steal queue Empty" are kept for each processor. Flag *interrupted* is global.
- When a process obtains work it sets *interrupted*.
- When a thread cannot get a job it does the following check:
Clear *interrupted*, go over all local flags to check if all queues and stacks are empty. If all are empty and *interrupted* is clear - termination has been detected.



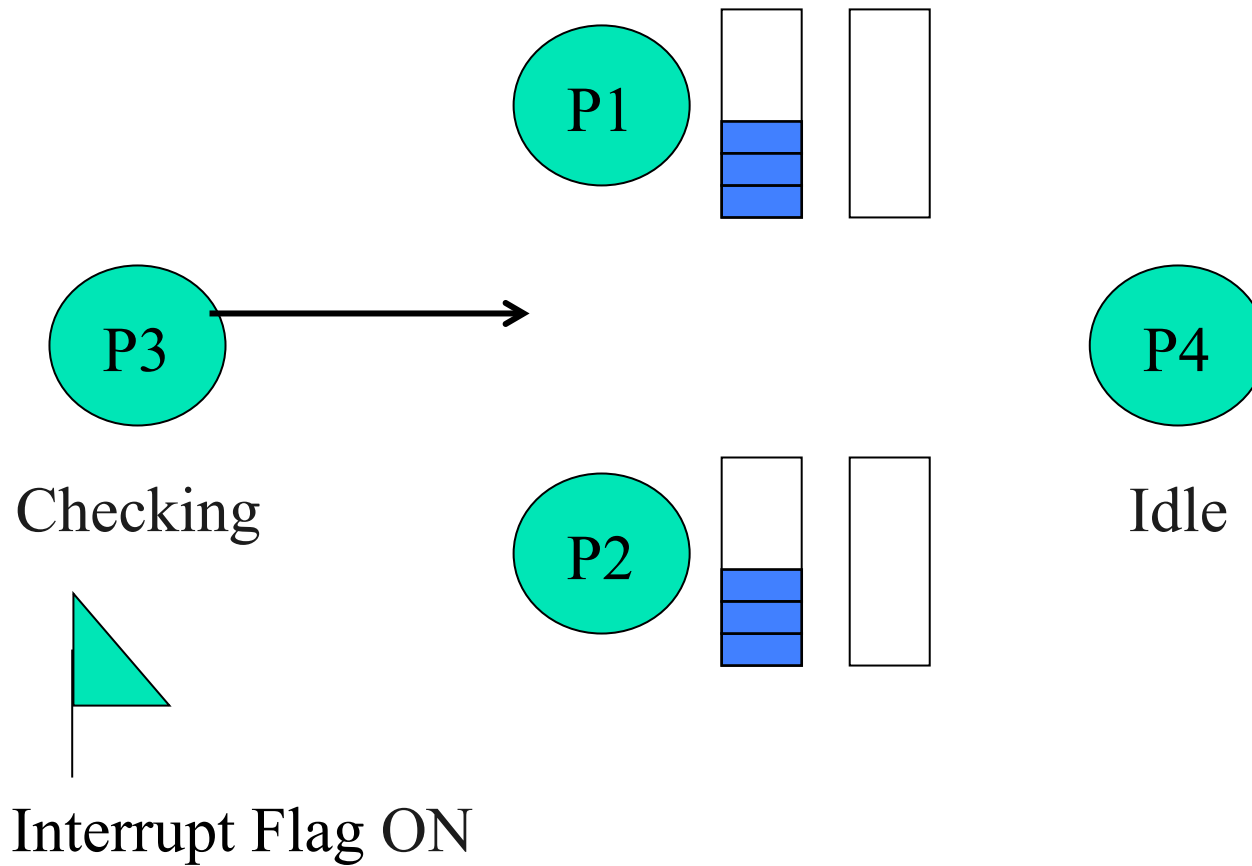
Termination Detection is Flawed

- Problem: more than one process may decide to detect termination.
- Thus, more than one thread may clear interrupted...

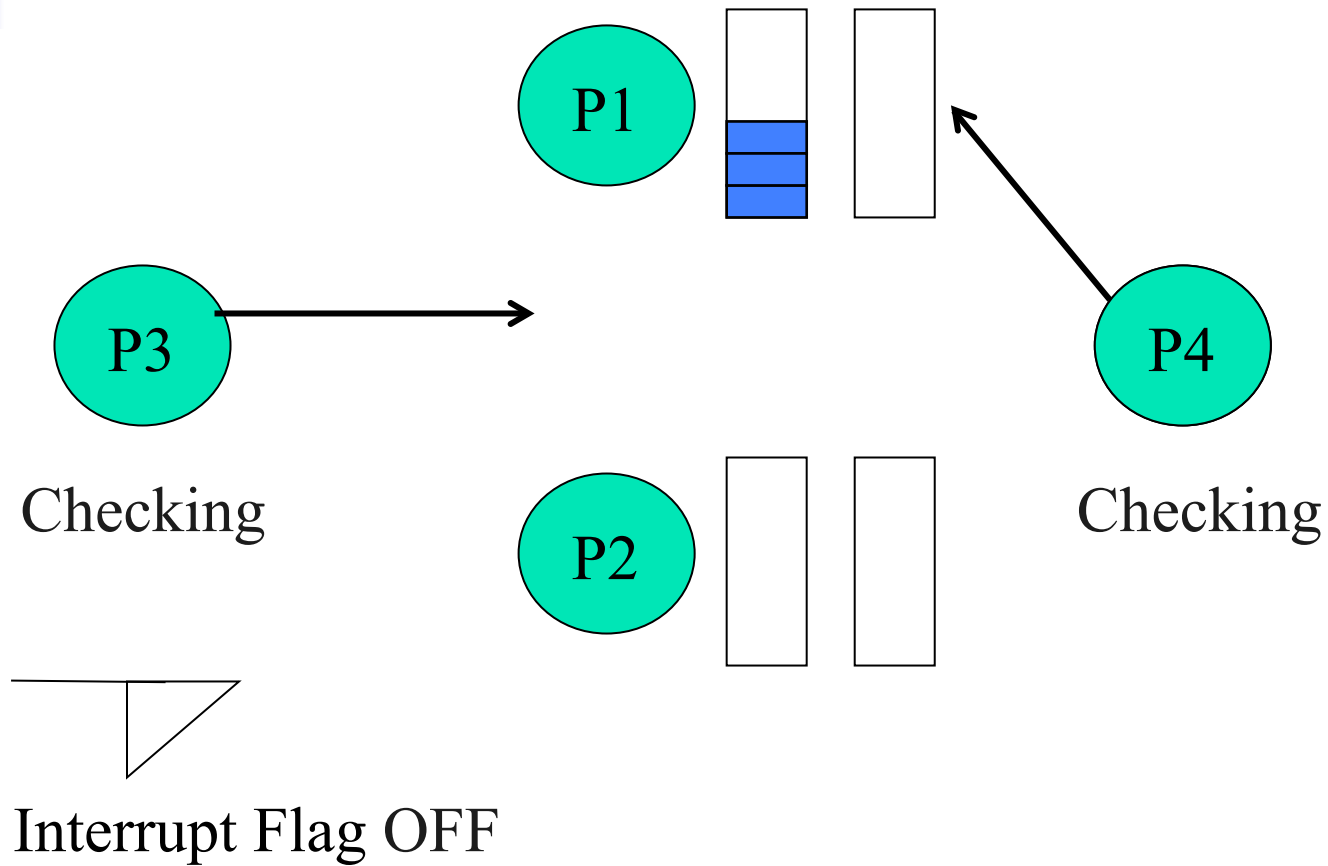
Bad Detection Scenario



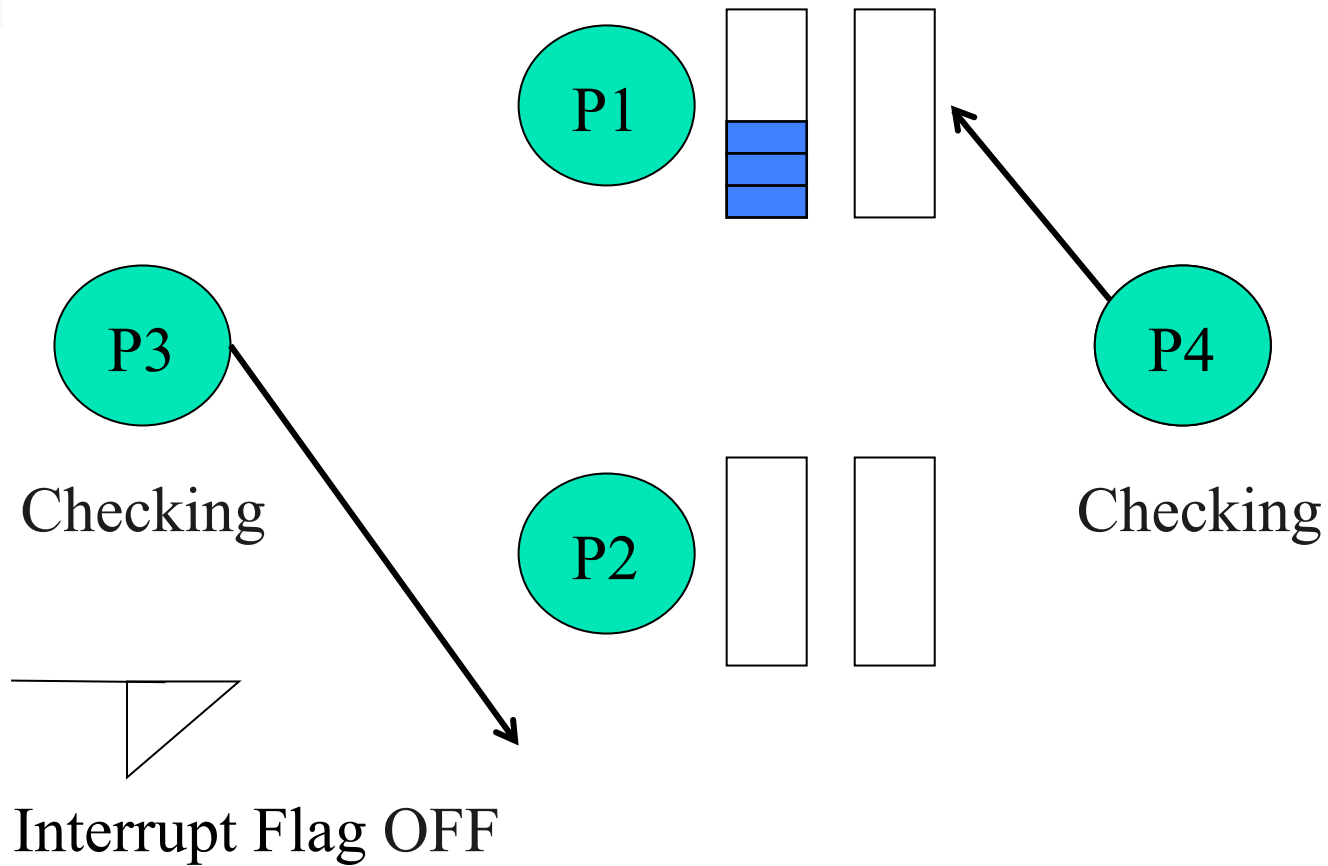
Bad Detection Scenario



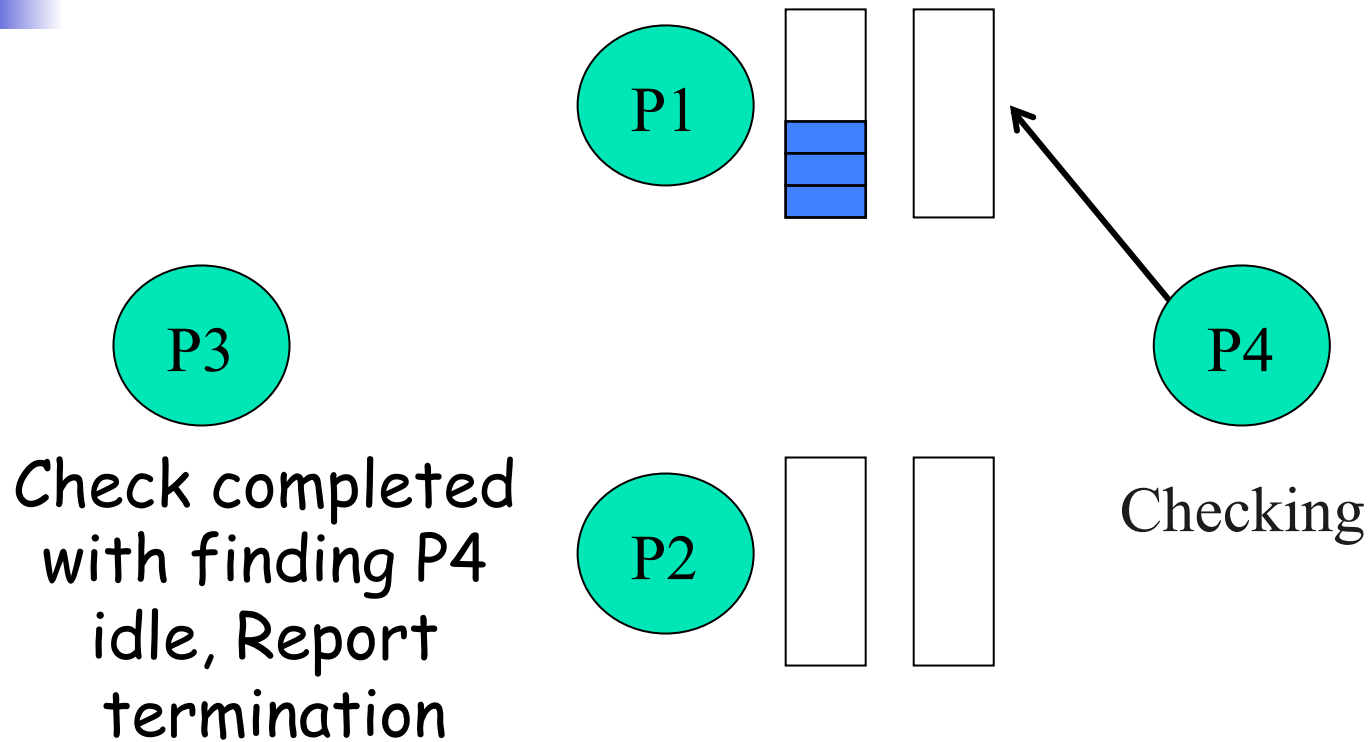
Bad Detection Scenario



Bad Detection Scenario



Bad Detection Scenario



- Solution: Add the detector ID to the flag



Modified Detection

- Flags “Mark Stack Empty” & “Steal queue Empty” are kept for each processor. Flag `interrupted` is global. The number `process-id` is global.
- When a process obtains work it sets `interrupted`.
- When a thread cannot get a job it does the following check:
Writes its id in `process-id`, clears `interrupted`, goes over all local flags. If all are clear, and `interrupted` is clear, and `process-id` was not modified - termination has been detected.



Summary of Parallel Copying

- Copying is similar to mark-and-sweep except for allocation in to-space.
- Delayed allocation allows allocating with no fragmentation and with little synchronization.



Allocation Techniques



Allocation techniques

- Fragmentation
- Some basic notions and techniques
- Doug Lea's allocator
- Boehm's allocator
- Allocation caches (IBM)
- Immix [Blackburn, Mckinley 08]
- Hoard [Berger, McKinley, Blumofe, Wilson 00]



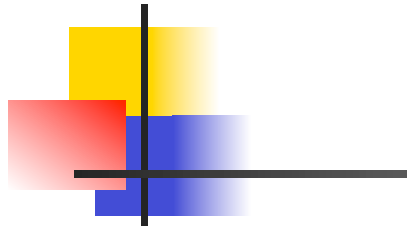
Allocator is measured by

- Speed of allocation
- Fragmentation
- (Speed of reclamation)
- Cache-conscious placement.

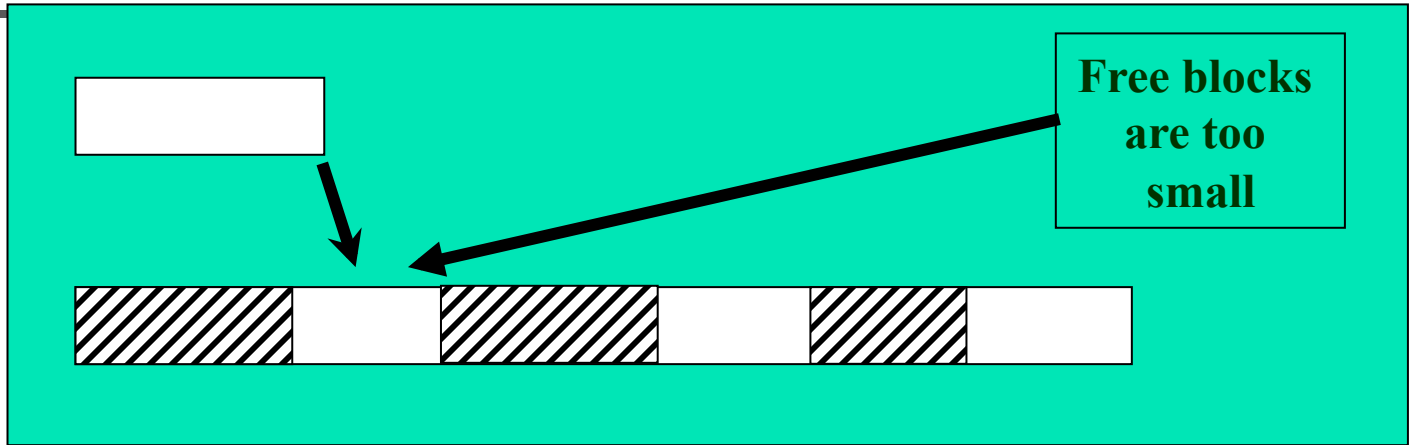


Fragmentation

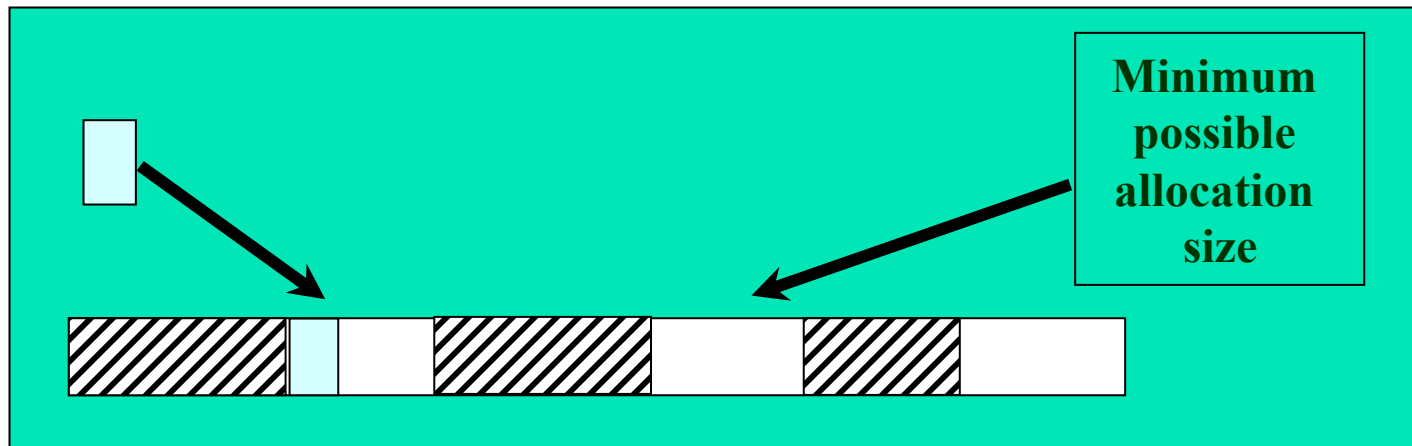
- Inability to reuse memory that is free
 - Severity determined by distribution of holes, and requests of the program.
 - Caused by reclamation or allocation policies (e.g., allow only certain sizes on one page).
- External fragmentation: holes outside the objects.
- Internal fragmentation: allocated area is larger than requested area.



External frag.



Internal frag.





Basic Notions and Techniques

- Various fits: best fit, first fit, next fit, worst fit, optimal fit, half fit.
- Buddy systems
- Indexed fits
- Segregated free lists
- Boundary tags

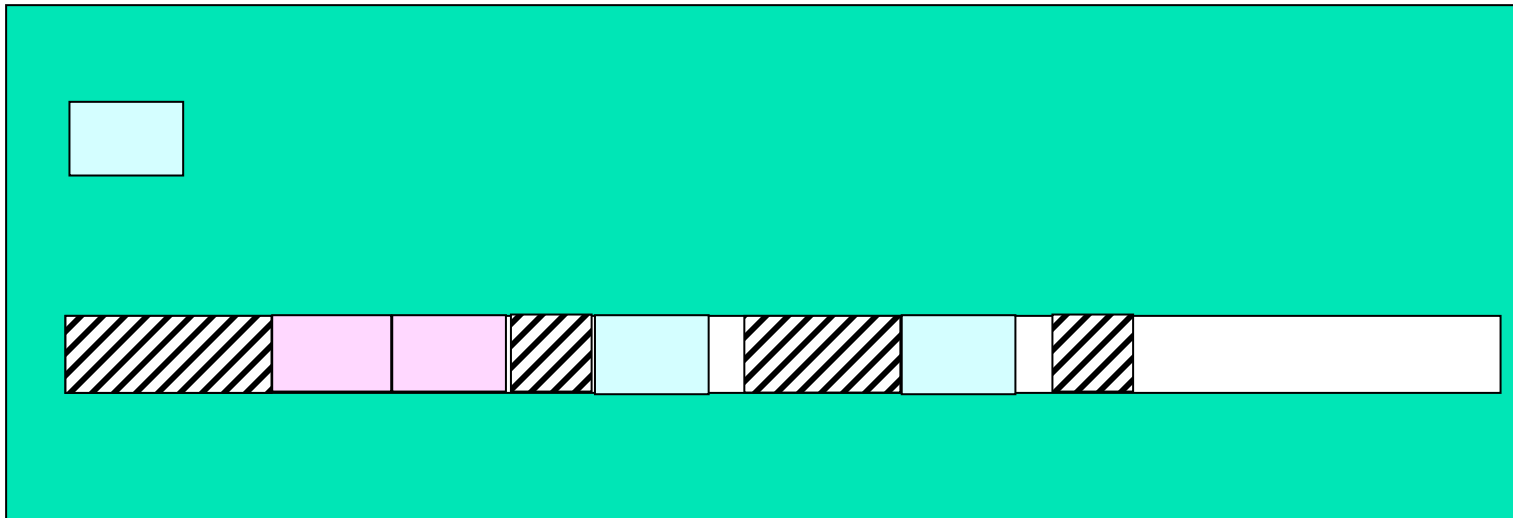


Best fit

- Allocate in the smallest free block that may satisfy the request.
- In practice,
 - Exploit most of the used "hole".
 - remainder will be quite small and perhaps unusable.



Best Fit Illustrated





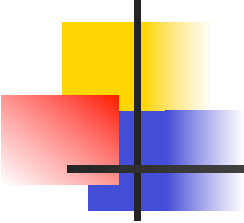
Best Fit (Cont.)

- **Naive implementation:** search the whole free-list (linear complexity, unacceptable).
- **Common implementations:** use balanced binary trees, or keep a list of available blocks for each possible allocation size (“indexed fits” or “segregated fits”).
- **Note difference** between **policy** and **implementation** (Wilson et al.)
- Best fit has low fragmentation with typical benchmarks.



First Fit

- Allocate in the first sufficiently large free block.
 - Typically address-ordered,
 - can be “free-list ordered”, or other.
- Search, split found block, put remainder on free-list.



First Fit behavior

- Lots of small blocks near the beginning of the list.
- These "splinters" (שבבים, רסיסים) increase fragmentation and may increase search times.



Next Fit

- Like first fit but start looking for the current fit where previous one found.
- **Advantages:**
 - prevents splinters accumulation.
 - Good locality: objects from different phases of program execution may become adjacent.
 - Less fragmentation if objects from different phases have different expected lifetimes.
- **Disadvantage:**
 - Does not concentrate allocations even if live data space is small.



Next Fit Performance

- In practice, causes more fragmentation than best fit or address-ordered first fit.
- Still, fragmentation is not high with typical benchmarks.



Worst Fit

- The largest free block is used.
- The idea: avoid creating small, unusable fragments.
- This works quite badly in practice.



Optimal and Half Fit

- **Optimal Fit**
 - a limited search of the list “samples” the list,
 - further search finds a fit as good or better.
- **Half Fit**
 - Find a block twice the requested size and split it.
 - Hopefully: soon a similar request arrives.

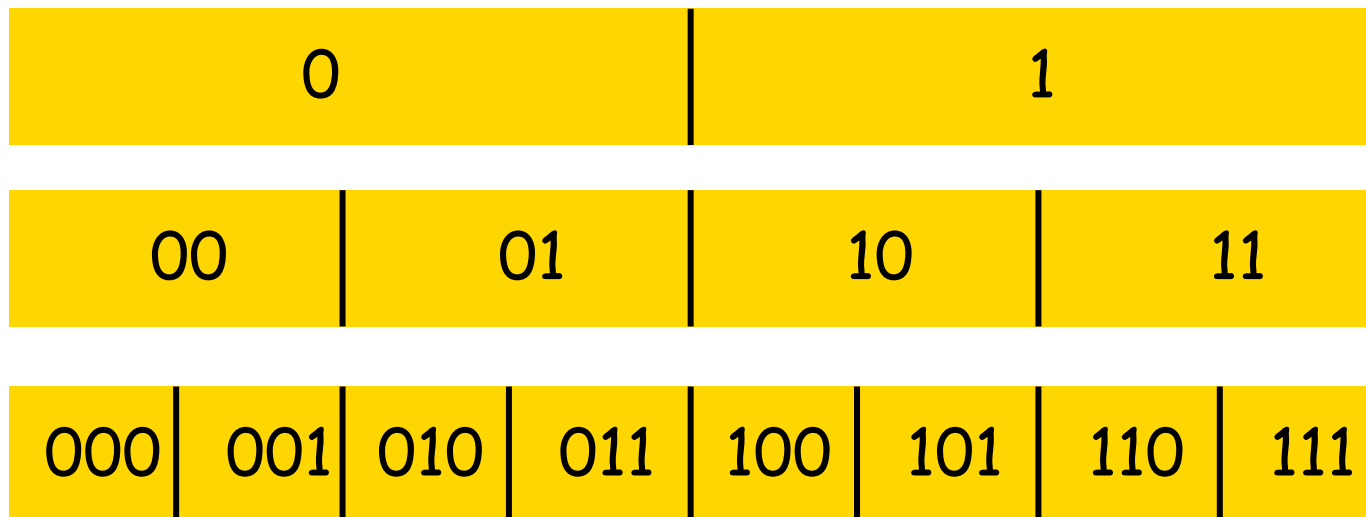


Buddy Systems

- The entire heap area is split into two large areas (which are buddies).
- Each area is further split into two smaller areas (which are buddies) and so on.
- For each allowable size, a separate free list is maintained, in an array of free lists.
- **Main benefit: efficient (though limited) splitting and coalescing.**



Buddy Systems





Buddy Systems

- A free block can (only) be merged with its *buddy*.
- Given a freed block, it is easy to find its buddy by a simple address computation.
 - If buddy is free - coalesce.
 - If a buddy is partially or entirely in use it cannot be merged with its buddy.



Buddy Systems

- + **Advantages:**
 - + Easy coalescing
- **Disadvantages:**
 - Internal fragmentation is relatively high: about 28% (rounding to a power of 2).
 - Coalescing is limited.



Indexed Fits

- Use an indexing data structure to obtain efficient searching of a desired policy.
- Examples:
 - Best fit with a balanced tree.
 - Buddy systems
 - Bitmap fits - use a bitmap to search for free space...



Modern Allocators



Header Fields

- Most allocators use a hidden “header” field within each allocated area to store useful information like:
 - Size of the block, whether the block is in use, its class object, locking info, hash info, garbage collection info (reference count, mark-bit) etc.



Coalescing via Boundary Tags

- For coalescing efficiency, allocated areas may also contain a "footer" field, with size of block.
- When a block is freed, examine footer of preceding block and header of following block for coalescing.
- **Space saver: use footer only if object is free. Use a bit in the next object header to indicate if it is.**



Segregated Free Lists

- Typical structure: an array of free lists.
 - Each list holds free chunks of a particular size.
 - A freed chunk is pushed to the appropriate free list by the collector.
 - Allocation uses appropriate list.
 - A pool of free blocks is kept aside.
- Preferred implementation for approximate best fit.



Segregated Free Lists Variations

- Each free list has a range of sizes for allocation.
 - Search for a chunk in the list using best fit, first fit, or next fit. (Typically --- first fit.)
- Number of lists.
 - A small number of lists may increase internal fragmentation or allocation time.
 - A large number of lists may increase space overhead.



Block-Oriented Segregated Free Lists

- Heap is partitioned to blocks (typically 4KB = page size).
- A block allocates only objects of same size.
- When a list is empty, a new block is obtained and split into same size objects.
- Reclamation:
 - Freed chunks are added to the appropriate list.
 - When a full block is reclaimed - it is returned to a pool and may later serve a different size.



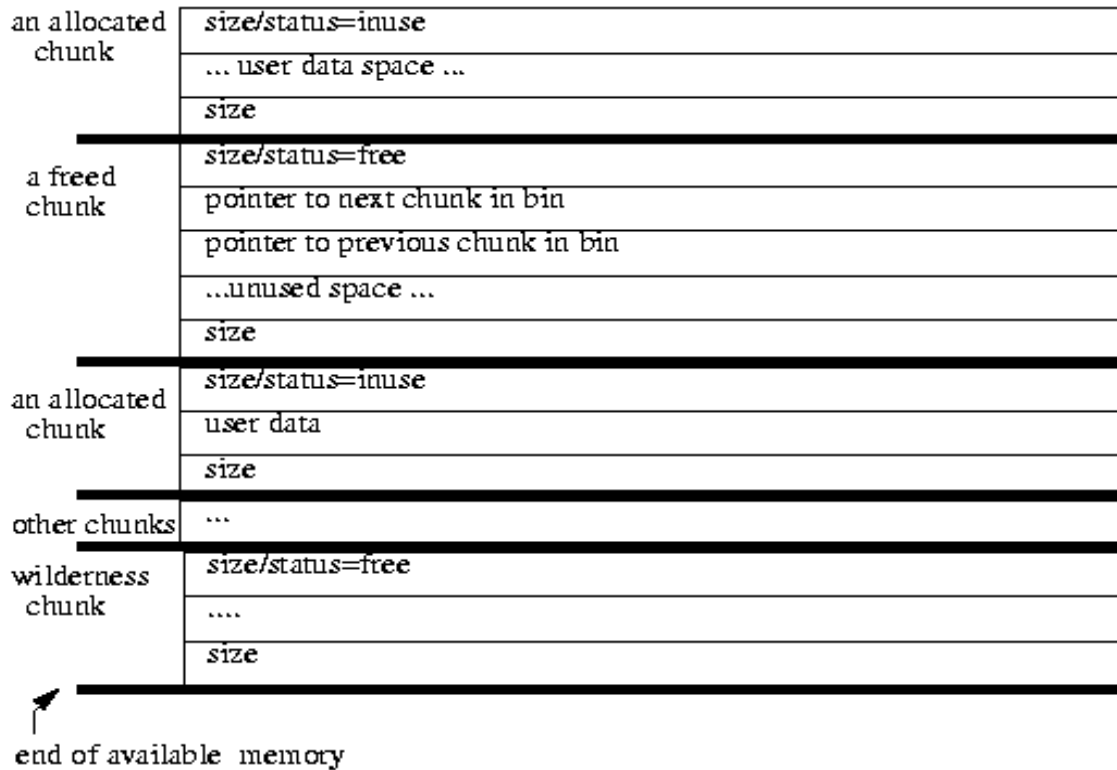
Segregated Free Lists Block Oriented

- Advantages
 - Shortened headers: (no need for size).
 - Efficient in time and space since typical programs use few sizes.
- Disadvantages:
 - External fragmentation.



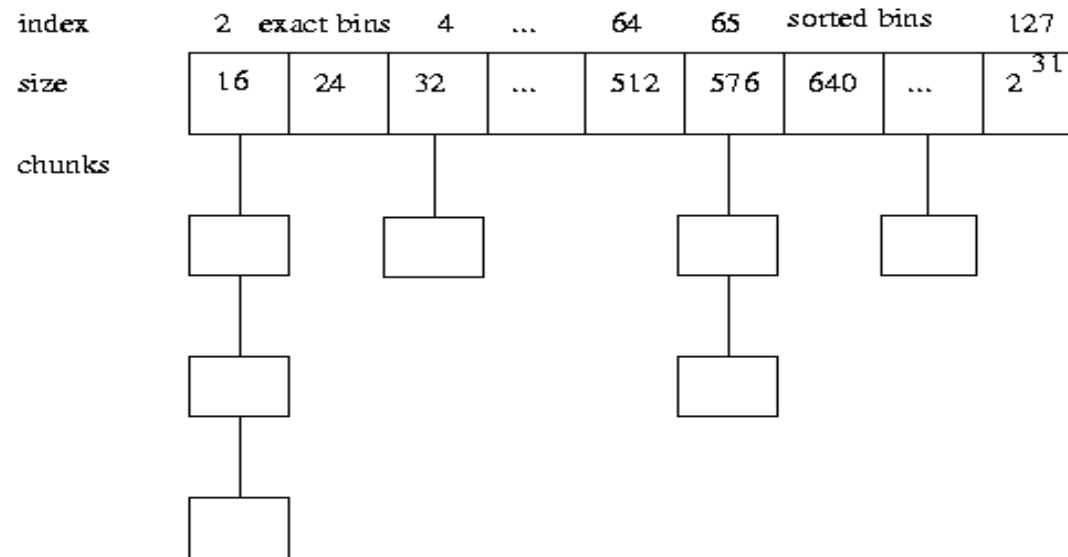
Doug Lea's Malloc

Boundary Tags for coalescing & traversing starting from any chunk in any direction.



Algorithm used

- Segregated free lists - Available chunks are maintained in bins, grouped by size.



- (Almost) Best fit



Improving Locality

- Use the following modification:
 - Start by trying to allocate in the appropriate bin.
 - If that fails, try using the leftover of the space used for previous allocation.
 - If that fails, get the smallest size that can accommodate this object.
 - If that fails too, use the wilderness (the free space at the end of the heap).
- Resulting algorithm is not best fit (but close).
 - Note wilderness Preservation.
- <http://gee.cs.oswego.edu/dl/html/malloc.html>



Boehm's allocator (and collector)

- http://www.hpl.hp.com/personal/Hans_Boehm/gc/
- Collector by Boehm-Demers-Wiser.
- May be used as a leak-detector
- Distributed with the GNU compiler
- Available for most standard PC and UNIX platforms, Win32, OS/2, and UNIX environments.
- The collector uses a mark-sweep algorithm. With incremental and generational support.



Boehm's allocator (and collector)

- Memory split into blocks, typically block size = page size.
- Free blocks are maintained in a tree sorted by address.
- Two level allocator:
 - Large objects are allocated from tree of blocks.
 - Each block is dedicated to a single object size.
 - Small objects are allocated in blocks.
 - The allocator maintains separate free lists for each size of object.



Parallel Allocation



Allocation for SMP's

- Boehm's allocator and segregated free lists in general can be extended for a multiprocessor.
 - Typically, by maintaining segregated free lists per thread.
- Next:
 - Allocation caches (IBM).
 - Immix.
 - HOARD.



Allocation caches

- Two goals:
 - Reduce contention on allocation.
 - Allow “bump-pointer” allocation for mark-sweep.
- **Method:** let each thread obtain a “local cache” using synchronization.
- After obtaining the local cache: allocate (small objects) from it locally with a bump pointer.



Method with Mark-Sweep

- All available spaces are kept in a free-list, created by sweep.
- When a local cache is needed, the first large-enough space is taken via first-fit.
- There is a minimum and maximum size for a cache.
 - Minimum - because we do not want to switch caches too often. Switching involves synchronization.
 - Maximum - because we do not want one thread to use all free space as its own cache, starving the other threads.
- All small objects are allocated from the cache.



The Free List

- If a free chunk on the list is too large, only a piece of it is taken for the cache, and the rest is left in the free list.
- Allocation of larger objects (say, more than half the minimum cache size) is done directly from the free-list via first-fit.
- A simple optimization: sweep does not put small spaces in the free list. All free chunks are large enough to serve as caches.



Adding Bit-Wise Sweep

- Letting sweep ignore small chunks is fine with allocation caches.
- A major overhead of sweep is finding & freeing each chunk of 0's in the mark-bit table.
- Method: sweep only looks for zero words.
- When found - free the whole chunk of zeros.
- Advantage: looking for a zero **word** is fast.
- Disadvantage: inaccurate decision which size is minimal.



Use with Copying Collectors

- Copying collectors already employ bump-pointer allocation, but may suffer contention on that pointer.
- The local allocation method lets each thread obtain a local allocation cache using the global bump-pointer, but then allocate inside the local cache with no contention.
- Large objects are still allocated using the global pointer.



Allocation Caches Properties

- Cache behavior: it is believed that allocating sequentially is very good for program locality.
- Local caches provide bump-pointer allocation to mark-sweep.
- It is fast and cache-friendly.
- Most allocations are executed locally in the local cache.
- Synchronization is seldom and thus contention is low.



Immix

- [Blackburn, McKinley PLDI'08]
- Heap is partitioned into
 - Aligned 128 bytes lines, and
 - Aligned 32KB blocks.
- A line-map has one bit per line indicating if it is free.
- Collector fills that line-map during the mark phase.



Allocation Procedure

- When a thread needs space to allocate, it obtains a block.
- Partially-full blocks are used before empty ones.
- A free block is used as an allocation cache.
- In a partially-full block:
 - Search (locally) for the next free line,
 - Group that line with all consequent free lines to form an allocation cache.
- When recyclable space in the block is exhausted, the thread gets another block.



Larger Objects

- Each thread also uses an overflow mechanism that keeps an empty block for overflow allocations
- Medium objects (greater than a line):
 - If the object cannot be allocated in the current hole, it is allocated using the overflow mechanism.
 - This eliminates skipping over many holes without allocating.
- Large objects are allocated directly from the pool of free blocks.



Corresponding Collection

- Mark-sweep
- Mark phase marks objects and lines.
- Sweep returns
 - free blocks to the global pool, and
 - partially free blocks to the recyclable block pool.
- Defragmentation by partial compaction attempting to free blocks.



Properties

- Bump pointer allocation for mark-sweep.
- The 128-byte lines suffer less space loss compared to the minimum 512-byte minimum cache size for allocation caches.
- Designated mark-sweep collection to support the allocator, filling a line-map and categorizing blocks.
- Designated partial compaction.

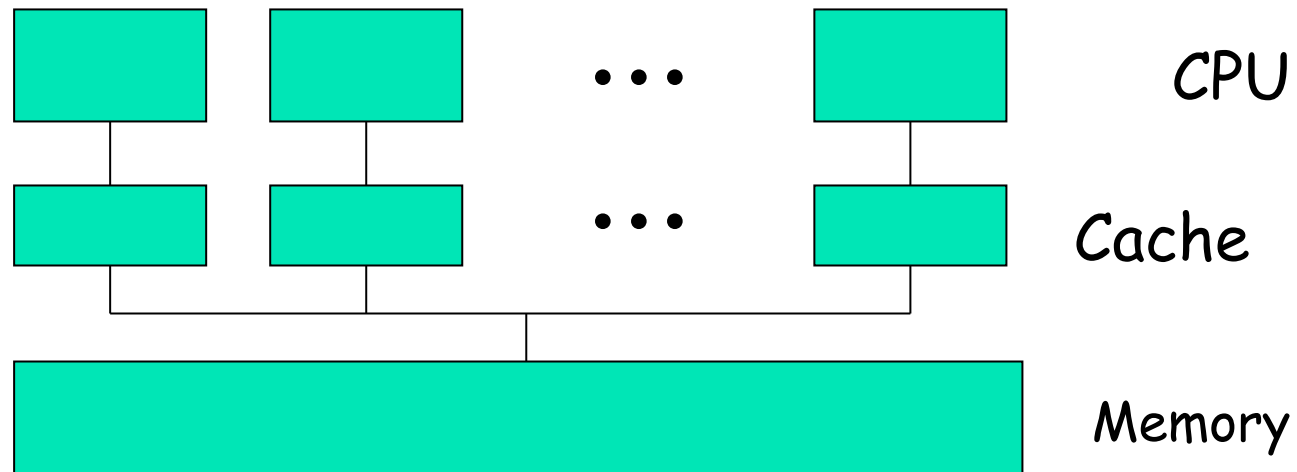


Hoard

- A Scalable Memory Allocator for Multithreaded Applications [Berger- McKinley-Blumofe-Wilson 01].
- Download from <http://www.hoard.org> .
- **Goal:** achieve efficiency and scalability on a multiprocessor.
- **Strategy:** avoid contention, avoid false sharing.



An SMP



Costly access to memory is ameliorated by the use of fast caches.



What is Sharing?

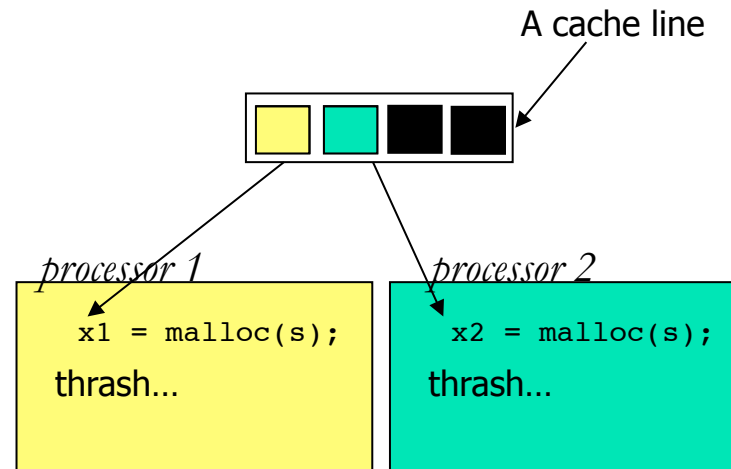
- If one object is **used by several threads**, then its content must be repeatedly consolidated between the caches.
- This is done by the **cache coherence protocol**.
- Use of caches is not as efficient in this case.



What is False Sharing?

- Suppose a cache line holds two objects O_1 , O_2 (or more) allocated by two threads T_1 , T_2 (or more).
- When T_1 and T_2 access their objects, sharing is falsely created because these objects which reside on the same cache line.
- “Thrashing”.

Allocator-Induced False Sharing



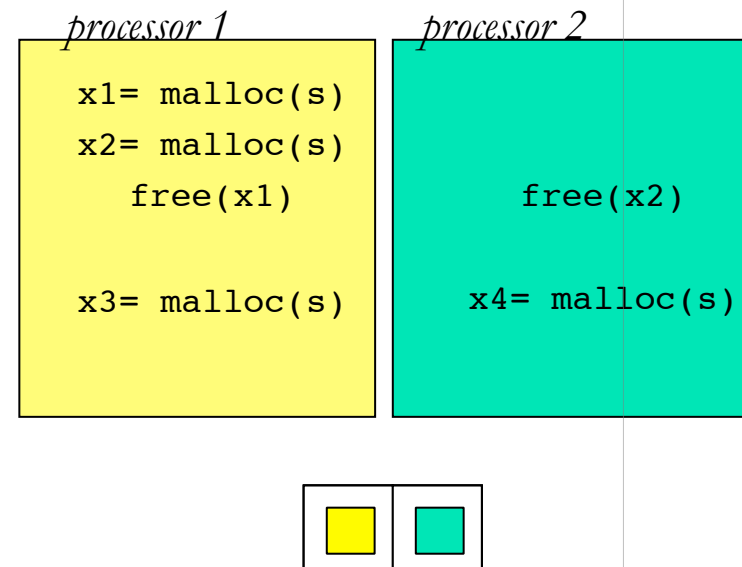
Using Local Heaps (Only)

Using one heap per thread:

malloc gets memory from the processor's heap or the system

free puts memory on the processor's heap

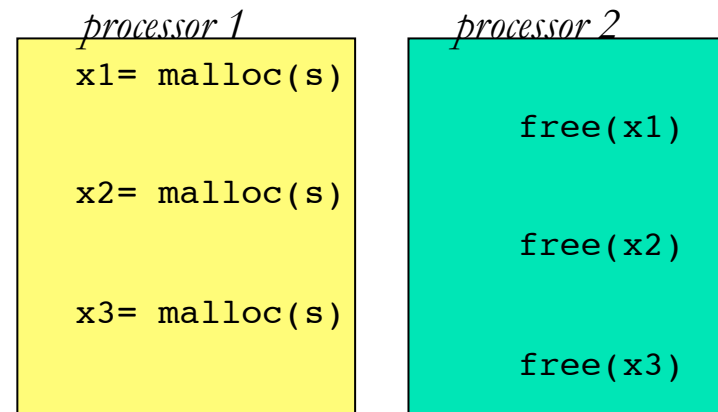
Creates thrashing.



Problems with Pure Local Heaps

Moreover, memory consumption can grow without bound!

Producer-consumer:
processor 1 allocates
processor 2 frees



Allowing Ownership

free puts memory back on
the *originating*
processor's heap.

Avoids unbounded memory
consumption

processor 1

```
x1= malloc(s)
```

```
x2= malloc(s)
```

processor 2

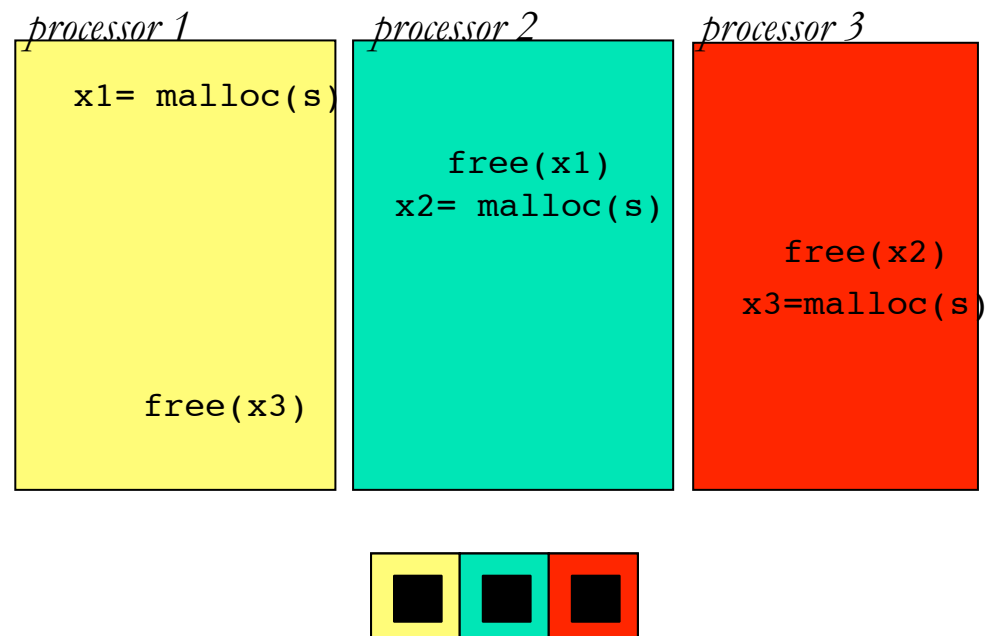
```
free(x1)
```

```
free(x2)
```



Still, Problems

- memory consumption can blow up by a factor of P .
- Problem: free chunks "belong".
- Round-robin producer-consumer:
processor i allocates
processor $i+1$ frees.





Using Local Heaps

- Use thread local heaps consisting of page-sized blocks.
 - Avoid contention on allocations.
 - Avoid false sharing: cache lines do not split between blocks.
- Each local heap employs block-oriented segregated free lists.
 - Namely, it consists of blocks, each holding objects of one size.



Using Local Heaps

- When the fraction of free memory in the local heap exceeds **the empty fraction**, blocks are returned to a global pool of blocks.
 - Avoid blowup in memory consumption, allow reuse of memory.
- When local heap is full, a new block is obtained from the global pool.



Large Objects & Recycling

- **Large objects** (more than half a block) are allocated directly from the operating system.
- **Small objects** are allocated from the thread's local heap in one of its blocks.
- **Recycling a block:** when a block is completely free the size of its objects may change.
 - Reduce fragmentation.

Hoard Example

malloc gets memory from a *block* on its heap.

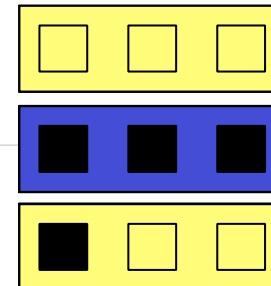
free returns memory to its *block*. If the heap is "too empty", it moves a block to the global heap.

processor 1

```
x1= malloc(s)
...some mallocs
...some frees
free(x7)
```

global heap

Empty fraction = $1/3$



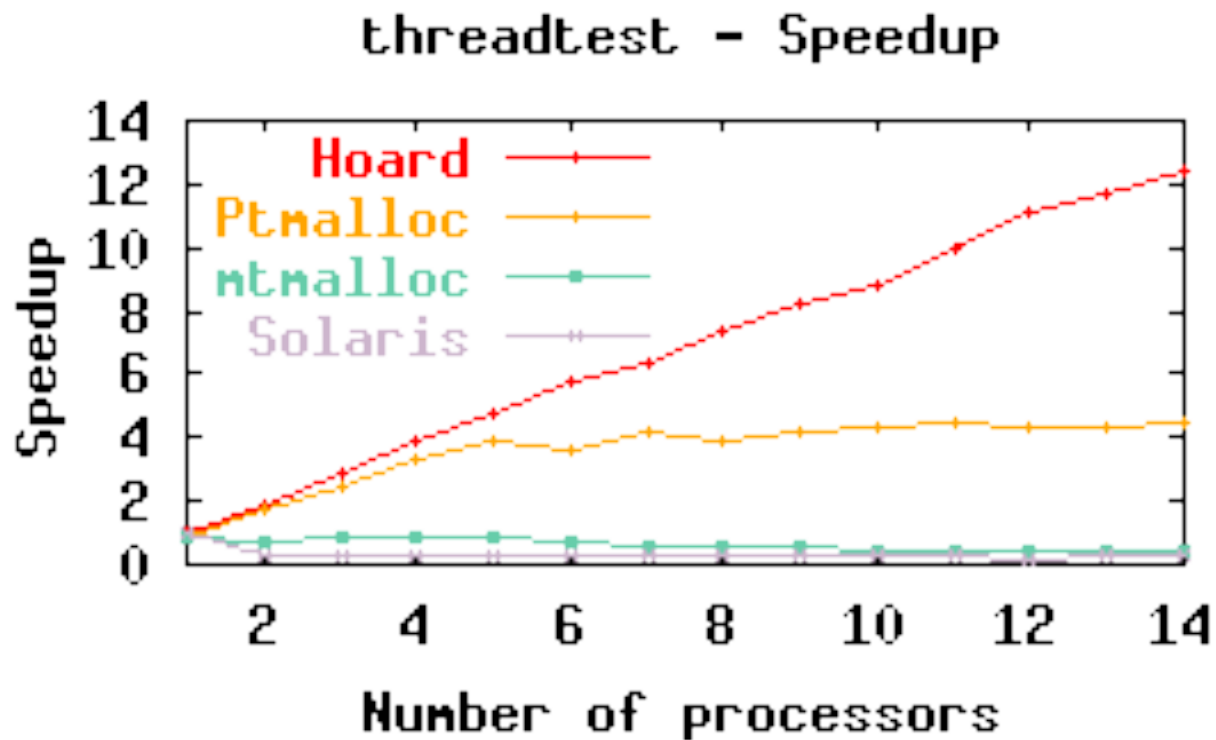


Compared Allocators

- Solaris: Default allocator for Solaris 7
- Ptmalloc: Linux allocator included in the GNU C library.
- MTalloc: a multiple heap allocator included with Solaris 7 for use with multithreaded parallel application.

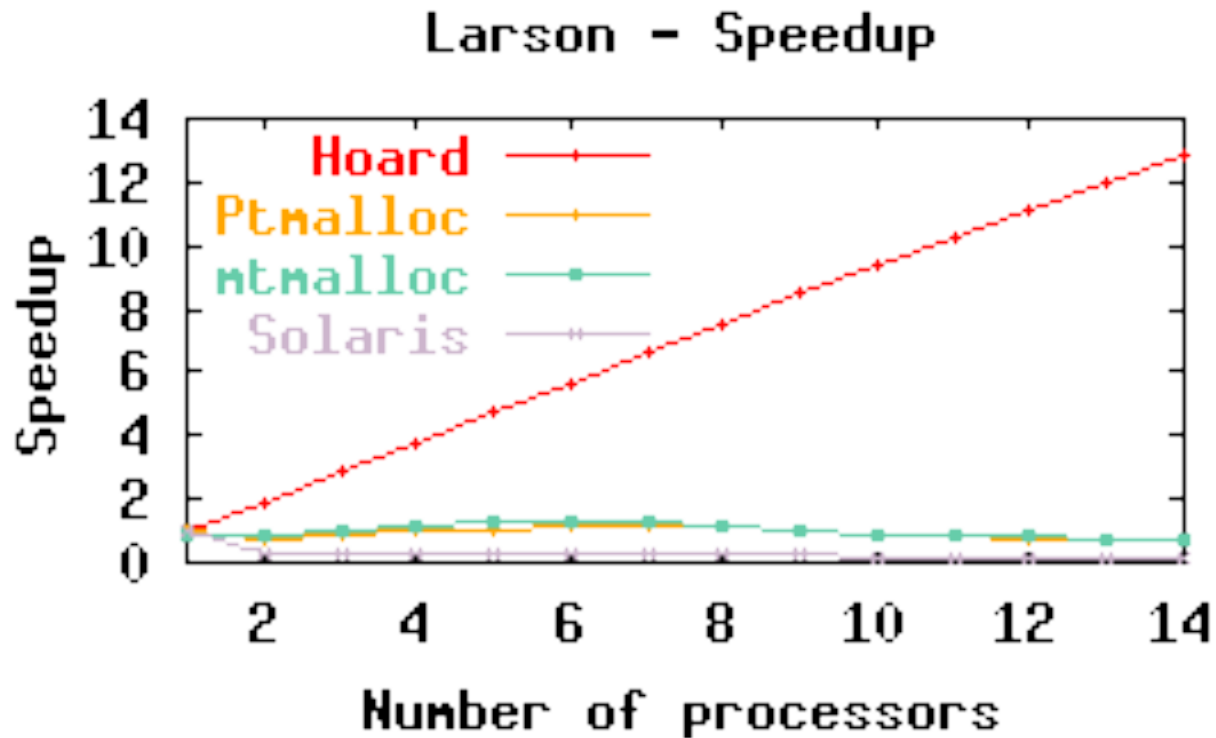
$$\text{speedup}(x,P) = \frac{\text{runtime}(\text{Solaris allocator, one processor})}{\text{runtime}(x \text{ on } P \text{ processors})}$$

Performance: *threadtest*



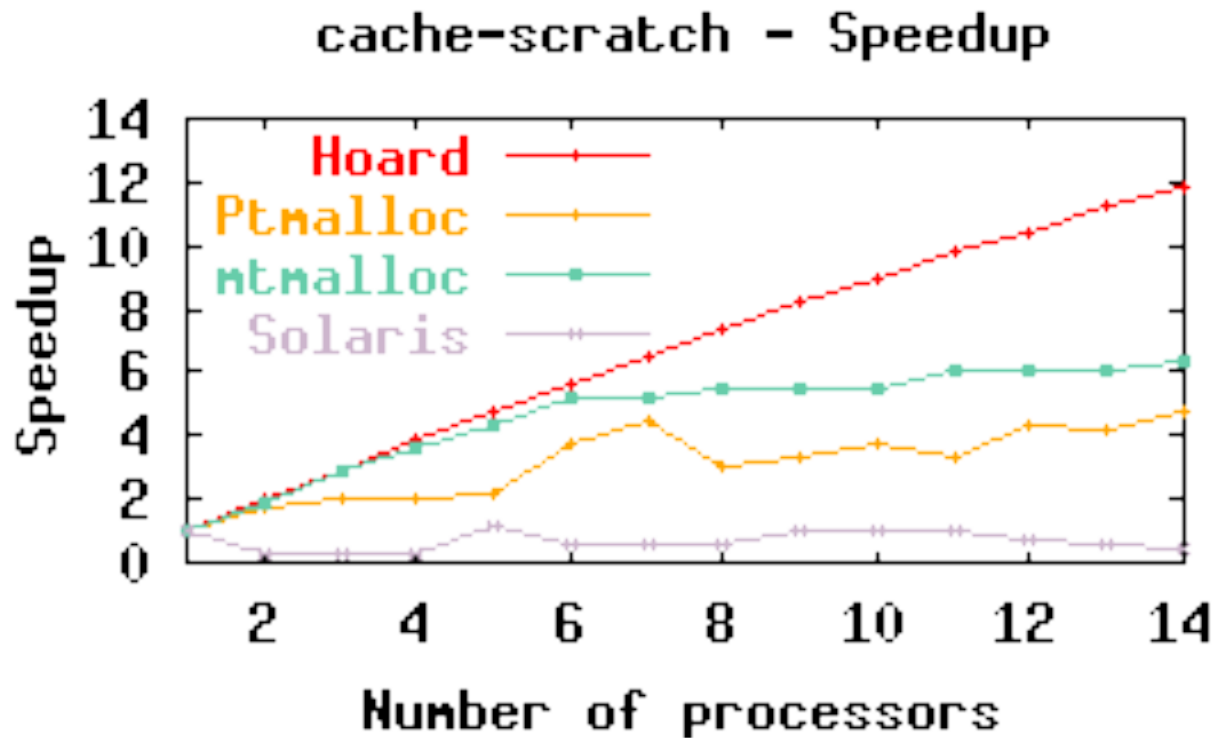
$$\text{speedup}(x,P) = \frac{\text{runtime}(\text{Solaris allocator, one processor})}{\text{runtime}(x \text{ on } P \text{ processors})}$$

Performance: *Larson*



Server-style benchmark with sharing

Performance: *false sharing*



Each thread reads & writes heap data



Fragmentation Results

On most standard uniprocessor benchmarks,
Hoard's fragmentation was low:

<i>p2c</i> (Pascal-to-C):	1.20	<i>espresso</i> :	1.47
<i>LRUsim</i> :	1.05	<i>Ghostscript</i> :	1.15

Within 20% of Lea's allocator

On the multiprocessor benchmarks
and other codes:

Fragmentation was between 1.02 and 1.24 for all but one
anomalous benchmark (*shbench*: 3.17).



Local Caches versus Hoard

- They have not been compared experimentally.
- Local caches may provide faster allocation.
- Hoard is more protective against false sharing.
- Fragmentation: ????
 - Hoard: use of blocks with a single size, local heaps.
 - Local caches: use only large enough caches.