CS452/652 Real-Time Programming Course Notes

Daniel M. Berry, Cheriton School of Computer Science
University of Waterloo

Real-Time Java

Java was originally designed to run embedded systems.

However, its core specification does not address real-time concerns.

In particular, its reliance on garbage collection (GC) for memory management precludes serious use of Java for programming RT systems.

Real-Time Specification for Java

The issues that make RT Java difficult are:

- asynchronous events, interrupts, timers, and clocks
- scheduling, synchronization, communication
- garbage-collected memory management, lack of direct access to physical memory
- lack of direct access to physical devices

Scheduling

Ordinary Java threads (Java-ese for “tasks”) have priorities, but the system does not necessarily always choose the highest priority thread, i.e., priorities are only advice to the scheduler.

Even a high priority thread gets delayed for an unbounded amount of time when GC occurs.

Three Kinds of Threads

In an attempt to get control over thread scheduling in the presence of potential GC, RT Java has three kinds of threads:

1. No heap real-time (NHRT) threads
2. Real-time (RT) threads
3. Regular threads

No Heap Real-Time Threads

- have no access to the heap; ∴ they cause no GC

  Remember that GC is triggered when a heap allocation, via a new, fails because there is not enough free memory to do the allocation.

- real-time scheduling: they are scheduled strictly according to their priorities

- for threads with the tightest deadlines

Real-Time Threads

- RT scheduling: they are scheduled strictly according to their priorities.
- have access to heap; ∴ they can cause GC
- for threads that can tolerate delay caused by GC

Regular Threads

as in ordinary Java

Required for RT Threads

- 28 fixed priority levels
- preemptive scheduling, as in *your* kernels
- avoid priority inversion via priority inheritance, which can be implemented using subclassing

Thread Communication

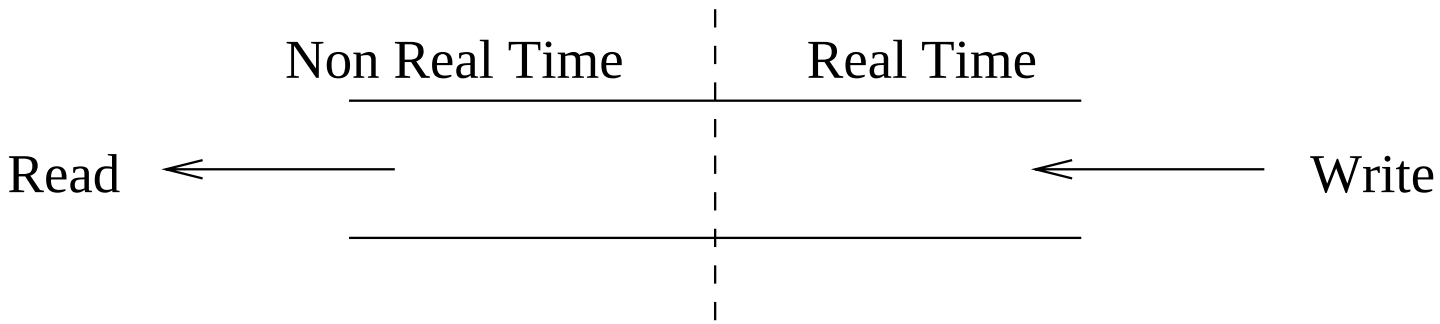
If a high-priority NHRT thread attempts to lock an object held by a lower-priority RT or regular thread, then the holding thread's priority is boosted to that of the high-priority NHRT thread, in a priority inheritance move.

But, if the holding thread is delayed by GC, then so is the NHRT thread. This is very *BAD*!

∴, we need a way for NHRT threads to communicate with RT threads and regular threads without risk of having wait for GC.

Wait-Free Write Queue

Allows communication from a RT thread to a non-RT thread:



The write side is non-blocking and therefore can suffer no GC delays.

The read side is blocking.

Wait-Free Write Queue, Cont'd

An NHRT thread can send data to an RT thread or a regular thread by writing to one of these wait-free write queues. The recipient task may block when it tries to read.

The queue data structure is statically allocated (Why?) and is therefore limited in size.

Therefore, data may be lost if queue is full, as old data get overwritten.

Garbage Collection

Several approaches, which are independent of RT considerations:

- mark and sweep
- reference counting
- copying

Roots of GC

Garbage collectors's job is to is to determine what is *accessible* so that the stuff that is not accessible can be turned into free memory to be used for future allocations.

Accessible data are those that may be reached by any chain of pointers originating at a *root*.

The roots is the minimal set of pointer variables allowing the non-terminated threads to reach all of their data.

Roots of GC, Cont'd

If you have only one thread, the root could be the pointers that are normally in a task descriptor.

If you have multiple threads, the roots could be pointers to all non-terminated threads.

If you have multiple threads, the root could be the pointer to the thread table.

Mark-and-Sweep GC

Mark:

Initially, every root object is found
but not yet followed;

Foreach object *x* not yet followed {
Mark whatever *x* points to as in use;
Do the same for any object found within *x*;
}

Requires a stack of objects found but not yet followed.

Mark-and-Sweep GC, Cont'd

Sweep:

Scan the heap:

Reclaim all unmarked memory;

Unmark all marked memory;

Reference Counting

For each object in memory, track how many variables point to it.

Need to update reference counts on every pointer assignment, decreasing the count of the object pointed to by the old value and increasing the count of the object pointed to by the new value.

Reference Counting, Cont'd

If, as an object's reference count is decreased, the count arrives at zero, then reclaim the object and decrement the reference count of every object the reclaimed object points to. This may recurse arbitrarily.

But, reference counting fails to notice that a set of objects that point only to each other is inaccessible, because each object's reference count is permanently at least one.

Copying

- Split the heap into two halves: OLD and NEW.
- Allocate memory from OLD, until it fills up.
- Then copy all accessible objects into NEW, compressing to one end of NEW and thus squeezing out garbage.
- Update all references to point to the new location of the referenced object.
- Exchange OLD and NEW.

Reference Counting

Reference counting suffers huge space and time overhead:

- It requires two count updates with each pointer assignment, plus a check for zero.
- If the count of any object goes to zero, an unbounded amount of additional count updates may be triggered.
- It requires one counter for each object that can hold the maximum possible count.

Mark-and-Sweep and Copying

These generally cannot happen while the program is running and the status of a pointer can change under the garbage collector’s nose.

Therefore, the program must wait an unbounded amount of time for the garbage collector to finish.

Conclusion

Garbage Collection is a HUGE problem for RT systems.

This is why *your* programs have stack allocation of procedure activation records and static allocation of global data structures.

RT GC

How can we put a bound on the amount of time spent on GC?

This is NOT a fully solved problem, to the extent that usually the solution is to make sure that GC is never needed, e.g., by having stack allocation of procedure activation records and static allocation of global data structures.

Java Itself

Java has two kinds of variables or objects:

- local variables, for blocks, functions, and methods; these are allocated upon entry to the declaring block, function, or method and are deallocated upon exit from that declaring block, function, or method; implemented using pushing into and popping from a stack of activation records

- new objects, in the heap

RT Java

RT Java has three types of memory for the new objects:

1. immortal memory
2. scoped memory
3. heap memory

Immortal Memory

- shared among all threads
- never garbage-collected
- reclaimed only when the whole program terminates

Scoped Memory

- for objects with well-defined lifetimes; each scoped object is allocated in a region tied to a block, i.e., a scope; thus the object's lifetime is that of the block, because exit from this block by the last task causes reclamation of all objects in the region tied to the block
- similar to stack allocation in C or C++

Scoped Memory, Cont'd

- regions are explicitly entered and exited by threads, reference counted.
- regions, not objects, are reference counted, and the count of a region counts not references to the region, but the number of threads that have entered the region but have not yet exited the same; thus there are no cycles invalidate reference counting.

Heap Memory

All other Java objects, which are garbage collected.

Running a Thread in a Scope

```
ScopedMemory lt = new LTMemory(min,max)
lt.enter(new Runnable(){
    public void run(){
        /* inside the scope */
    }
});
/* outside the scope */
```

Threads in Scopes, Cont'd

Other memory classes include

- VTMemory: scoped memory with potentially variable time allocation as opposed to guaranteed linear time allocation for LTMemory
- HeapMemory
- ImmortalMemory

Threads in Scopes, Cont'd

Note that you may use *any* appropriate thread object as the argument for enter including

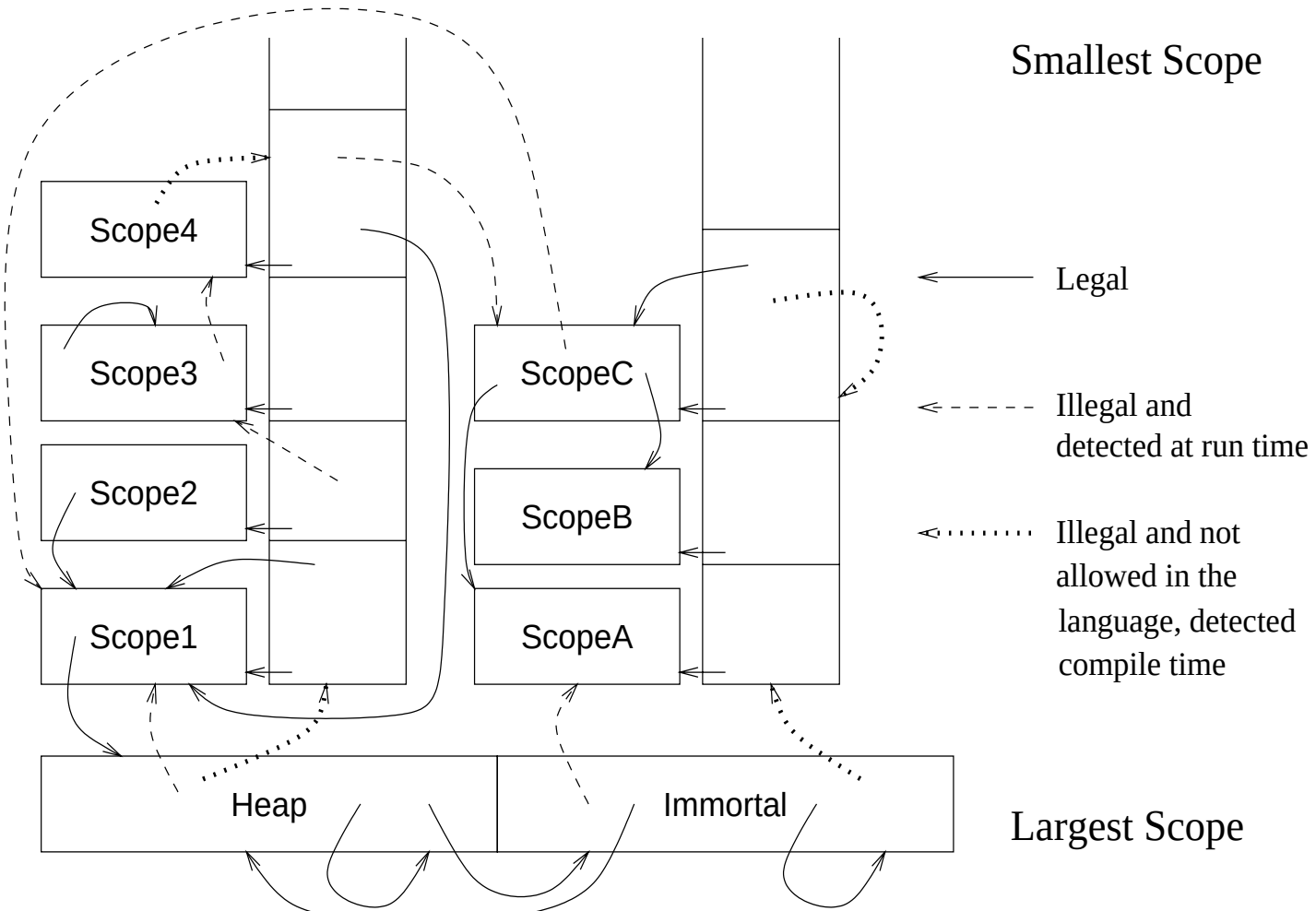
- a previously created one and
- the this of a thread object.

By "appropriate", I mean that sending a NHRT thread into a HeapMemory object is not appropriate.

Rules

- A heap object can reference any heap object or any immortal object, but not any scoped object.
- An immortal object can reference any immortal object, any heap object, but not any scoped object.
- A scoped object can reference any immortal object, any heap object, or any scoped object in the same or a larger scope.
- A local variable can reference any immortal object, any heap object, or any scoped object in a region that is referenced by the thread that owns the local variable and is in the same or a larger scope.

Rules, Cont'd


Smallest Scope
Scope4
Scope3
Scope2
Scope1
ScopeC
ScopeB
ScopeA
Heap
Immortal
Largest Scope
Legal
Illegal and detected at run time
Illegal and not allowed in the language, detected compile time

Rules, Cont'd

These rules are enforced at assignment time at run time.

They ensure that no pointer points to an object that can disappear before the pointer disappears.

Rules, Cont'd

Assuming that upward is increasing addresses, each pointer consists of three parts:

1. the pointer itself
2. its scope, e.g.,
   a pointer to the base of the activation record defining the scope if the object is scoped and
   0 if the object is immortal or in the heap.
3. its thread, e.g.,
   a pointer to the base of the thread owning the pointed to object if the object is scoped and
   0 if the object is immortal or in the heap.

Rules, Cont'd

A pointer assignment is legal IF

- the target of the pointer has a 0 scope and 0 thread

OR

- the place to which the pointer is being copied has a scope ≥ than that of the target and has a thread = to that of the target.

Rules, Cont'd

The only kind of object *o* that could disappear, leaving a dangling pointer that used to point to *o* is is a scoped object in a scope that is being exited by the last of its tasks.

This dangling pointer could be in only:

- an immortal or heap object

or

- in a longer lasting scoped object

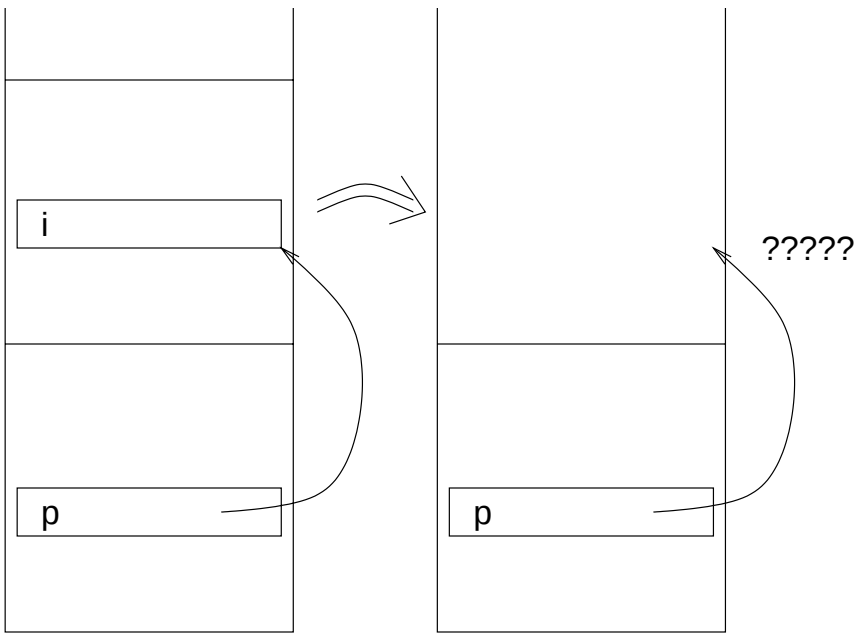
So, any assignment that can produce a potentially dangling pointer is outlawed, as in Algol 68.

Violation of Rules

I will show you a RT Java program that *I believe* behaves like this C program:

```
{int p*;
    …
    {int i;
        …
        p = &i;  /* causes upward pointer */
    }
    …
}
```

Violation of Rules, Cont'd



Violation of Rules, Cont'd

```
{ScopedMemory ltOuter =
        new LTMemory(1024,1024)
ltOuter.enter(new Runnable(){
    public void run(){T p;
    ScopedMemory ltInner =
            new LTMemory(1024,1024)
    ltInner.enter(new Runnable(){
    public void run(){T i = new T;
            p = i;  /* not allowed */
    }}); /* inner scope */
}});} /* outer scope */
```

My Opinion of RT Java

This stuff is incredibly complex.

I cannot believe that any one has written any real RT system using this stuff.

I could not even find examples in the Web of illegal programs, and I am not sure that my example is right.

If I were using RT Java, I would use only immortal memory and NHRT threads and be done with it.

No Scoped Memory

If there is no scoped memory, can the standard GC procedures be optimized?

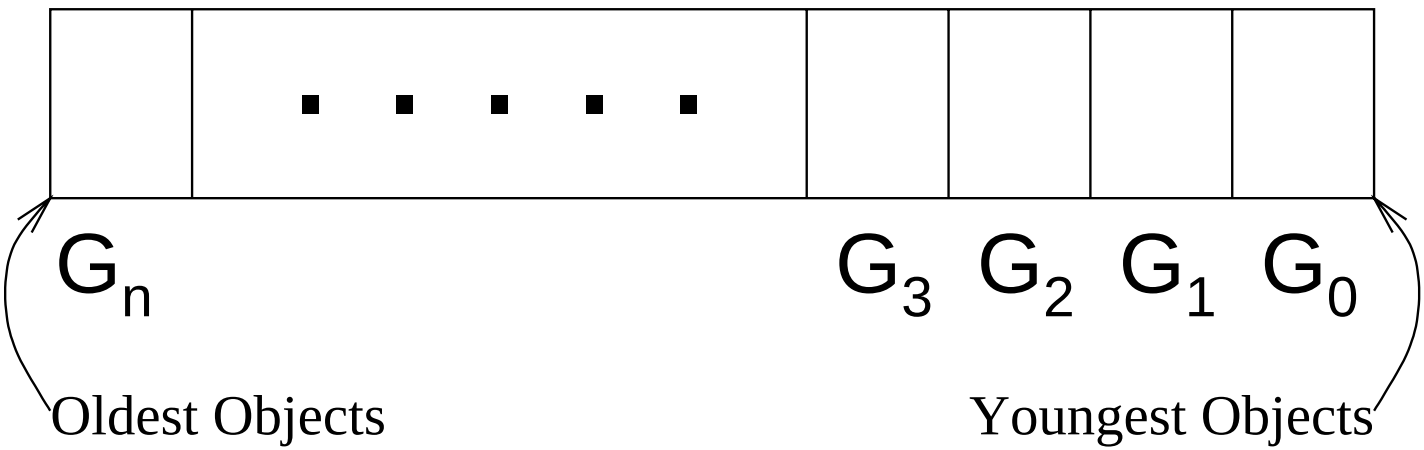
Generational GC

Assumptions based on observations:

- The longer an object lasts, the less likely it is to die.
- So, the older an object gets, the less often it should be scanned.

Generations

So divide the heap into generations:



On GC

Do Mark-and-Sweep GC or Copying GC on *only* G_0.

The roots of the GC include:

- program variables
- pointers into G_0 from older generations

On GC, Cont'd

If there are too many pointers into G_0 from older generations, then the GC may be too slow, …

but these older objects pointing to newer objects tend to be rare;

however, the algorithm must remember which objects these older objects pointing to newer objects are.

On GC, Cont'd

If an object survives a few GC iterations, move it to the next higher generation.

As higher generations fill up, GC them as well, but that should be infrequent.

Incremental GC

- Allow the garbage *collector* to be interleaved with the main program, called the *mutator* because it keeps changing pointers and what is accessible.
- Collect a little bit of garbage at a time.

If the collector collects garbage too slowly, the mutator has to wait for the collector to free more space.

If the collector collects garbage too fast, the collector may hog valuable cycles, causing the mutator to miss its deadlines.

Equilibrium

So an equilibrium must be reached, one cell allocated, one cell freed.

Paper: Henry C. Baker, Jr., “List Processing in Real Time on a Serial Computer”, *Communications of the ACM*, **21**:4, pp. 280–294, April, 1978

This paper gives a real-time incremental GC algorithm and a significant part of the paper describes the conditions under which equilibrium is reached and maintained.

Preliminaries

Two important concepts are used in Baker's algorithm:

- Updating pointers
- Read barrier

Updating Pointers

The literature often calls updating pointers as “forwarding pointers” because of the forwarding addresses, i.e., forwarding pointers, involved, but the act itself should be called “updating” as updating its address for a client is what an organization does when the post office notifies it of a client’s forwarding address.

Updating Pointers, Cont'd

Updating *p*: given an instance of a pointer *p* that points into a datum *d* in OLD, make *p* point to *d*'s copy in NEW.

There are three cases:

1. *d* has not yet been copied to NEW.
2. *d* has already been copied to NEW.
3. *p* is not a pointer at all or *d* is in memory not subjected to GC.

Updating Pointers, Cont'd

1. If *d* has not yet been copied to NEW, then *d* is copied into NEW at the place pointed to by next; next is incremented by the size of *d*; the address of this new copy of *d* in NEW is put into the first word of *d* in OLD; change *p* to be this forwarding pointer.

(It's OK to overwrite the OLD *d* with the forwarding pointer since *d*'s real value as already been saved in the NEW copy!)

Updating Pointers, Cont'd

2. If *d* has already been copied to NEW, then the first word of *d* contains a forwarding pointer pointing to *d*'s copy in NEW; change *p* to be this forwarding pointer.

(Since no value in *d* could point into NEW, this forwarding pointer cannot be part of *d*'s original value.)

Updating Pointers, Cont'd

3. If *p* is not a pointer at all or *d* is in memory not subjected to GC, then do nothing.

Read Barrier

Many a GC algorithm has a write or read barrier, of something that must be done by the mutator every time it writes or reads, respectively.

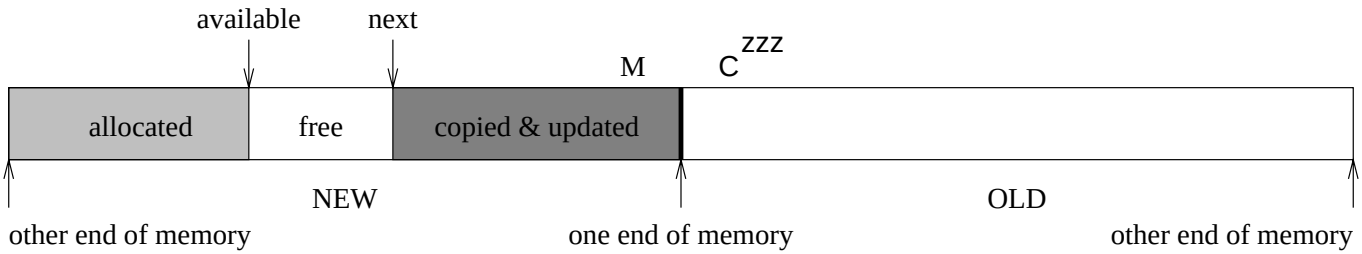
In a read barrier, generally whenever the mutator reads a datum, it must check something about that datum and then possibly do something, often a function of the value of the datum.

Read Barrier, Cont'd

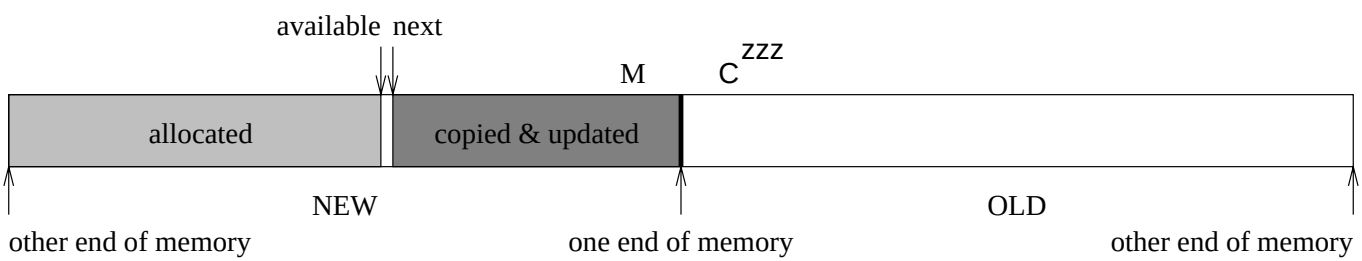
For Baker's algorithm, the read barrier is that if what is read from the datum is a pointer pointing to OLD, the pointer in the datum must be updated, as described above.

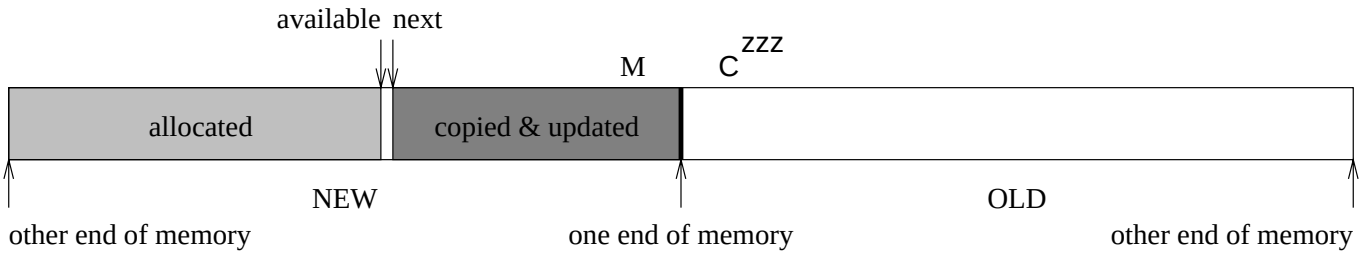
Baker's Algorithm

Baker's algorithm is based on Copy GC, and uses the same OLD and NEW half memories.

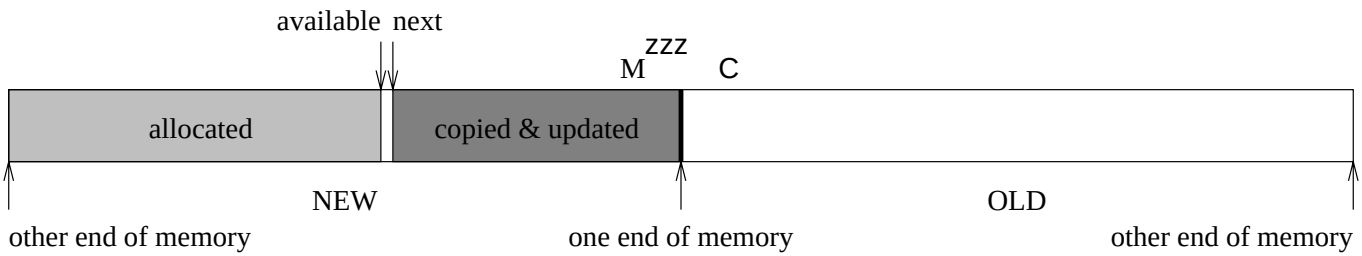


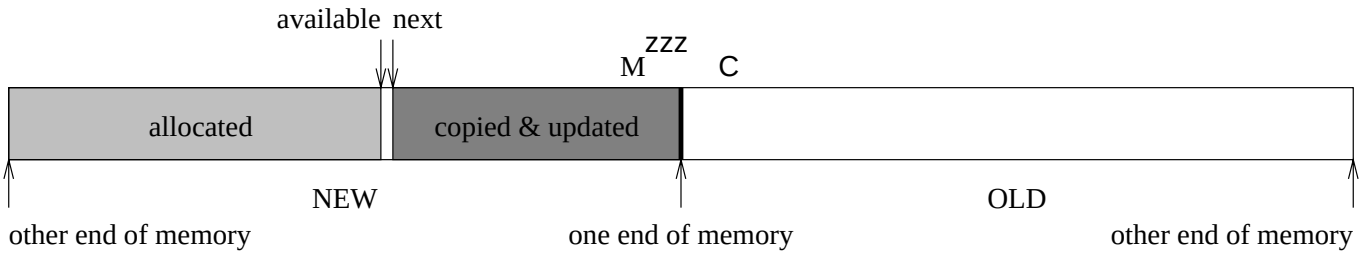
1. GC begins when the mutator's attempt to allocate in NEW fails due to insufficient free space.



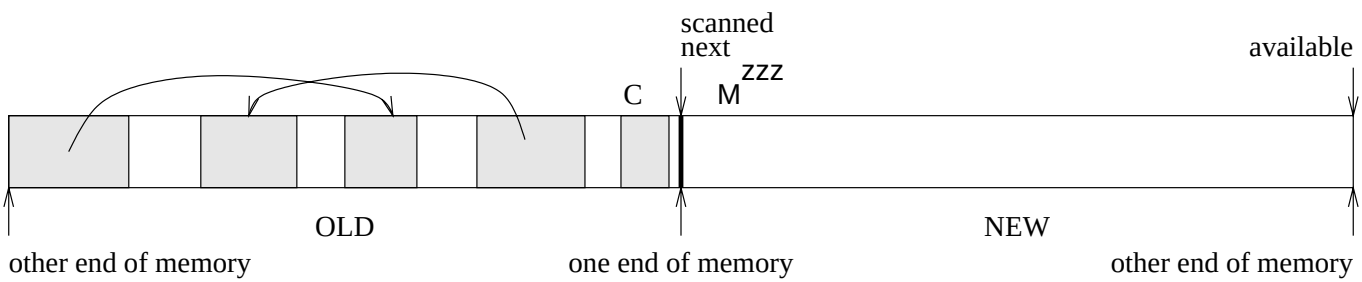


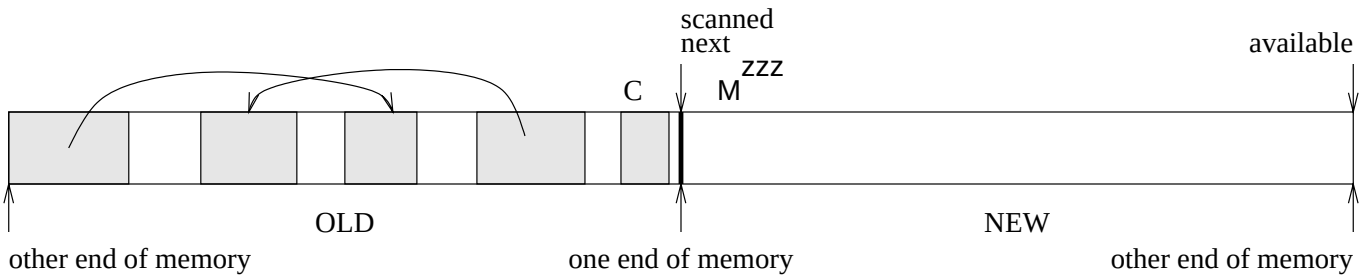
2. The mutator is blocked and the collector is awakened.



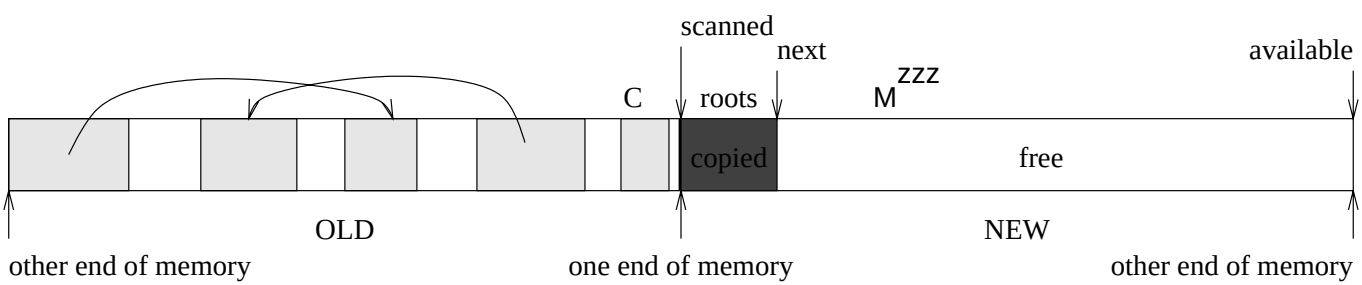


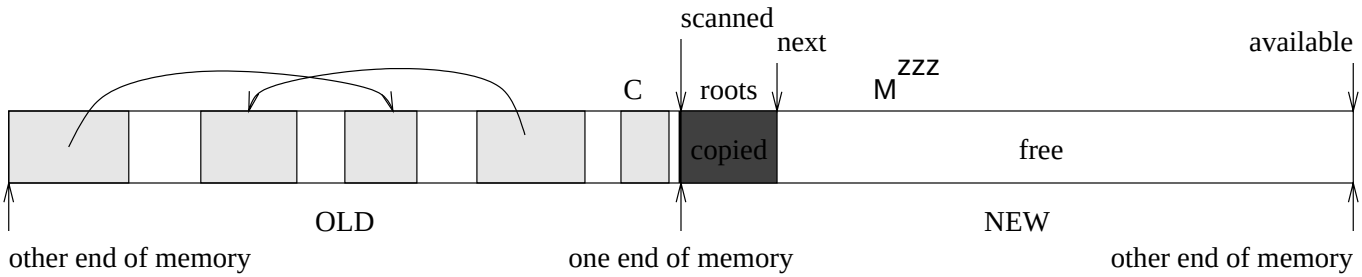
3. OLD and NEW are swapped.



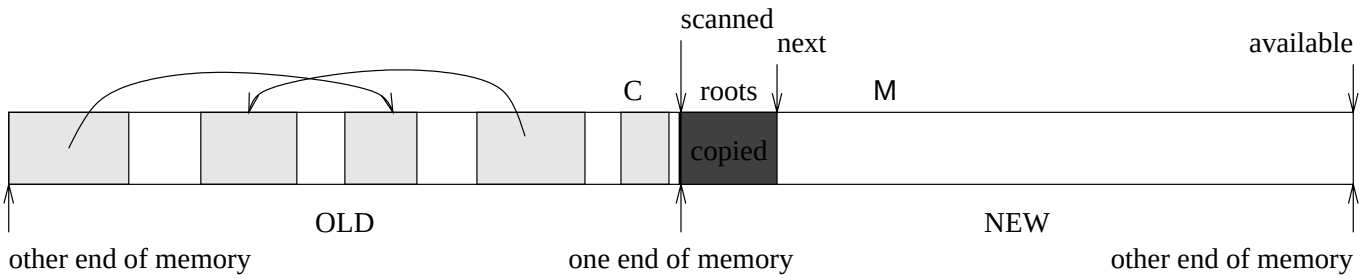


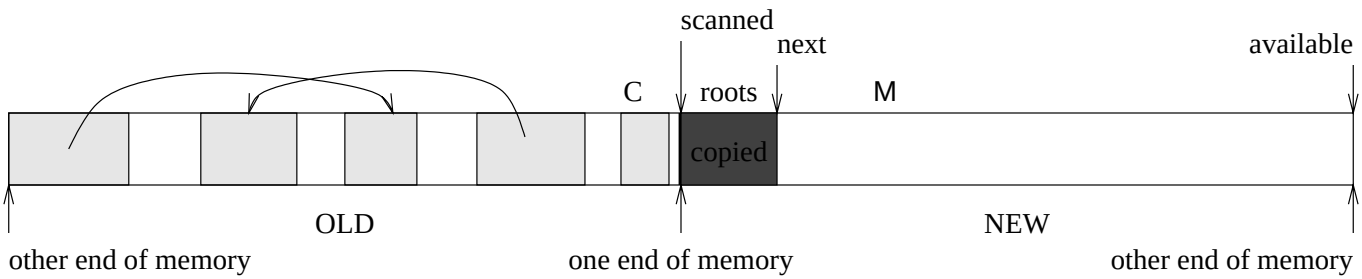
4. Every root pointer is copied to one end of the NEW space, at the place indicated by next, and next is incremented by the size of the roots. (Alternatively, in addition, each root pointer is updated.)



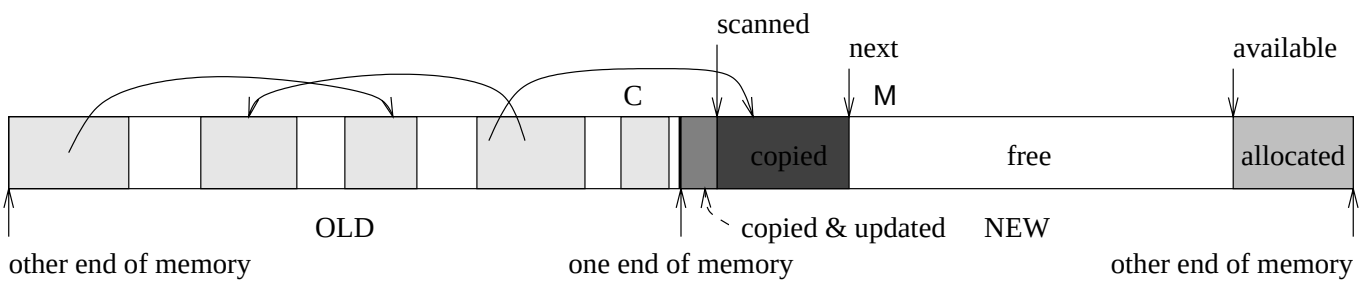


5. Then the mutator is made ready, with the root pointers not yet updated and no other accessible data copied to NEW.





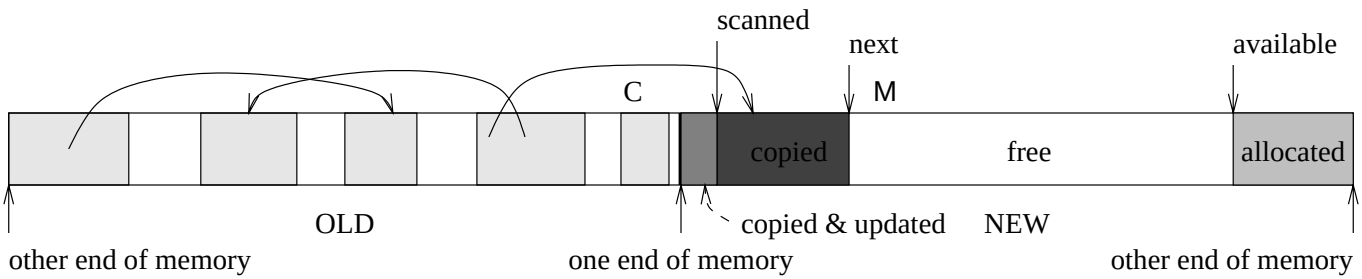
6. Every call by the mutator to allocate results in updating a few more pointers, and memory for the allocate is taken from the other end of NEW, at the place indicated by available and available is decremented by the size of the allocation.



Read Barrier

read barrier: If the mutator ever reads a pointer *p* that still points into OLD space, then update is done with *p*, possibly copying the target of *p* from OLD to NEW, and in any case changing *p* to point to the NEW copy.

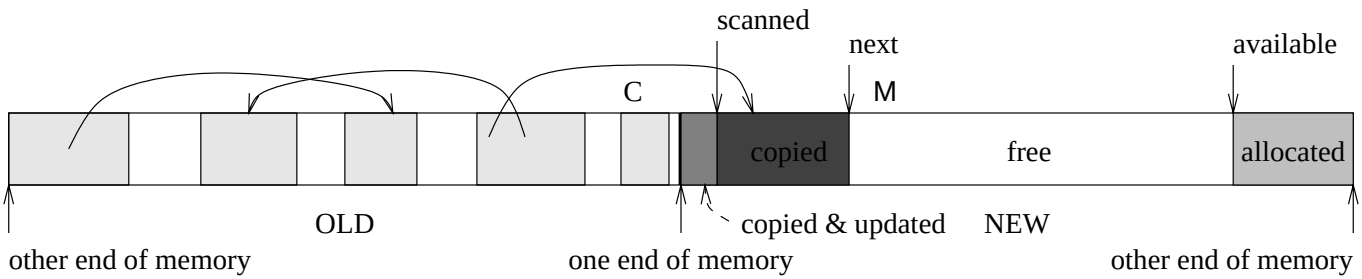
Invariants



Before where scanned points, every pointer points to NEW.

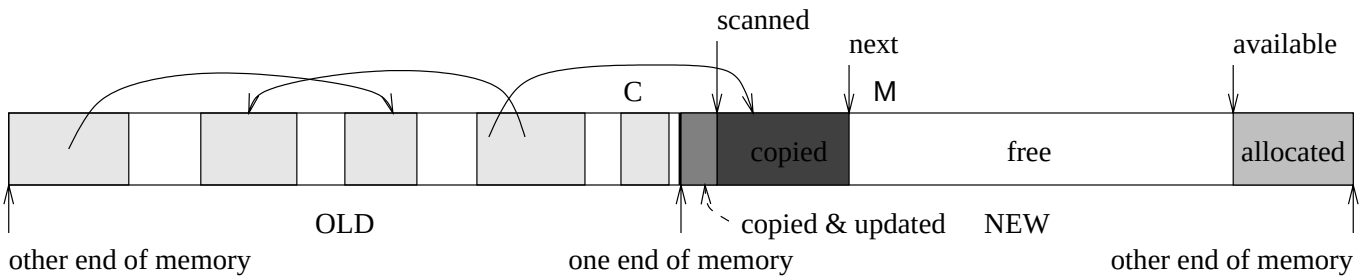
Between where scanned points and where next points, any object in NEW can contain pointers pointing to OLD.

Invariants, Cont'd

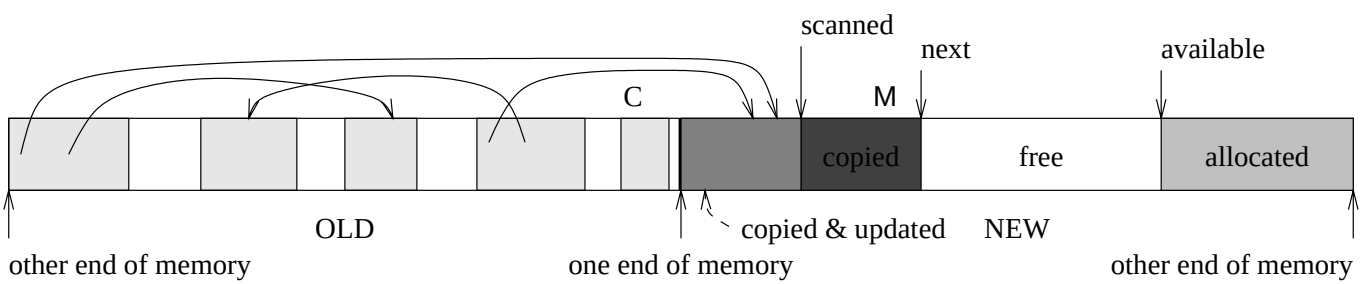


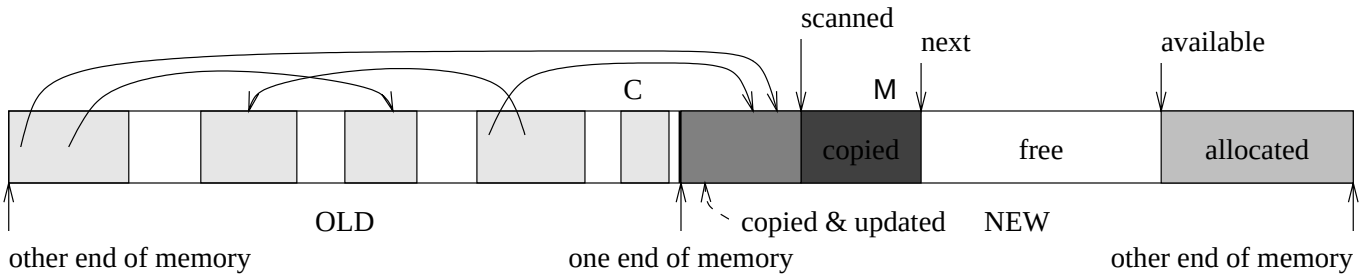
Between where next points and where available points is free space.

After where available points are all objects allocated since the last swap of OLD and NEW.

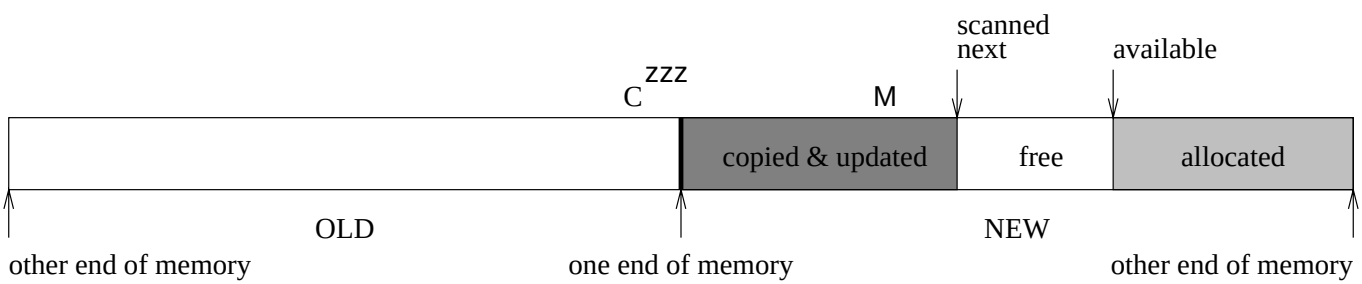


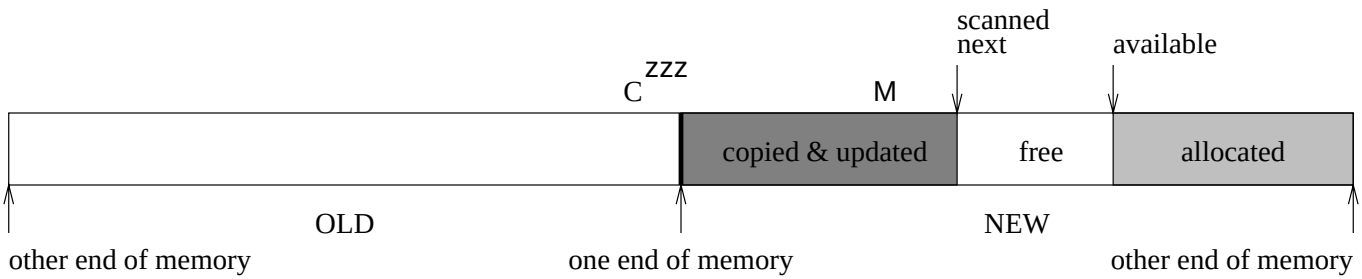
7. Every time the mutator allocates a word, scanned is incremented by at least one word, i.e., at least one more pointer is updated.



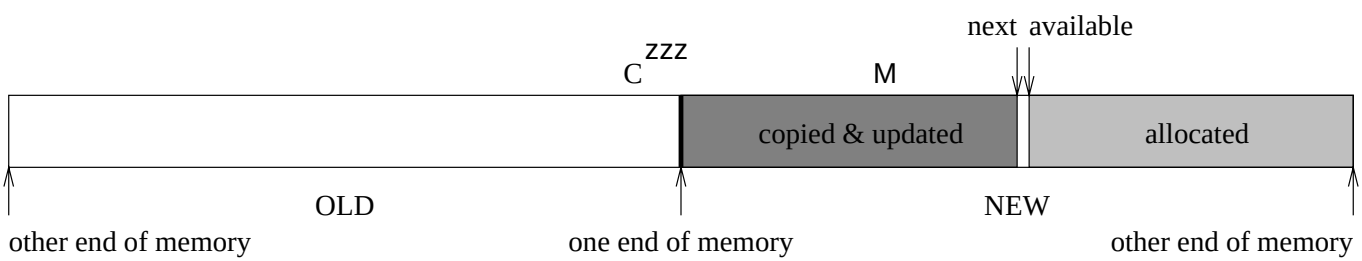


8. This updating occurs until scanned = next, and then the collector blocks.





9. The mutator continues until the difference between next and available is not enough for the next allocate and thus, NEW has filled up. We have step 0 again, but with the picture flipped about the center.



Note that any object allocated after where available points, while the collector is running is never scanned, i.e., its pointers are not updated, because they do not need to be.

∴, Baker's algorithm does not copy any more data than were alive at the swap of spaces.

∴, Copying of accessible data from OLD to NEW cannot fill up NEW unless there is absolutely no garbage in OLD.

Overhead of Baker's Algorithm

The biggest cost of Baker's algorithm is the read barrier. Each read potentially costs an update, which may include a copying of an arbitrarily long datum.

If 10% of the instructions are fetches from a heap object, and each fetch requires two instructions to determine whether the fetched value is a pointer pointing into OLD, then the overhead of maintaining the read barrier is at least 20%

Final Exam

Monday 17 December, 4:00 pm–6:30 pm, RCS 205

- all aspects of your kernel and application; you might be asked to implement a new kernel feature, a new project feature, or modify an existing one
- scheduling theory
- scheduling in RT Java
- scoped memory
- garbage collection: mark-and-sweep, reference counting, copying, generational, incremental

Final Project

The documentation for the project is due at 4:30pm on 30 November, electronically with submit and hard copy of non-code in the lock box.

The demo of project is Tuesday 4 December, 8:30am, in the train room.

Final Project, Cont'd

The final version of the code for the project is due Tuesday 4 December at 4:30pm electronically with submit.

PS:
Please have taken a bath before the demo, especially if you have been coding 24/7 right up to the point of the demo! ☺