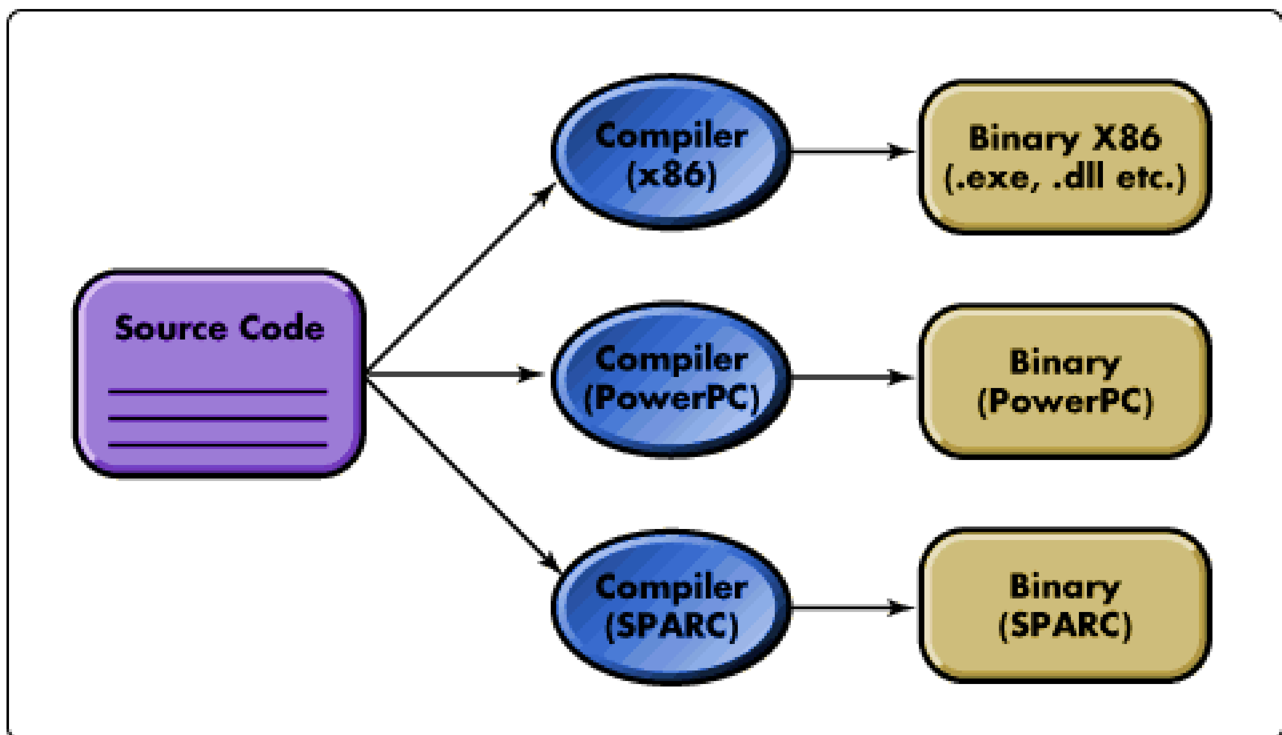
Revision Concepts before course starts

1. Java compilation model
2. Subroutines/methods
3. Storage allocation
   (a) variables
   (b) objects

Compilation Model

Java is (largely) portable. Historically many high level languages have come in significantly different versions that were different enough that any significant program created in one couldn't run under the other system. And even when a language has come with a clear standard, the same program might behave differently on different systems simply because of hardware differences between computers.

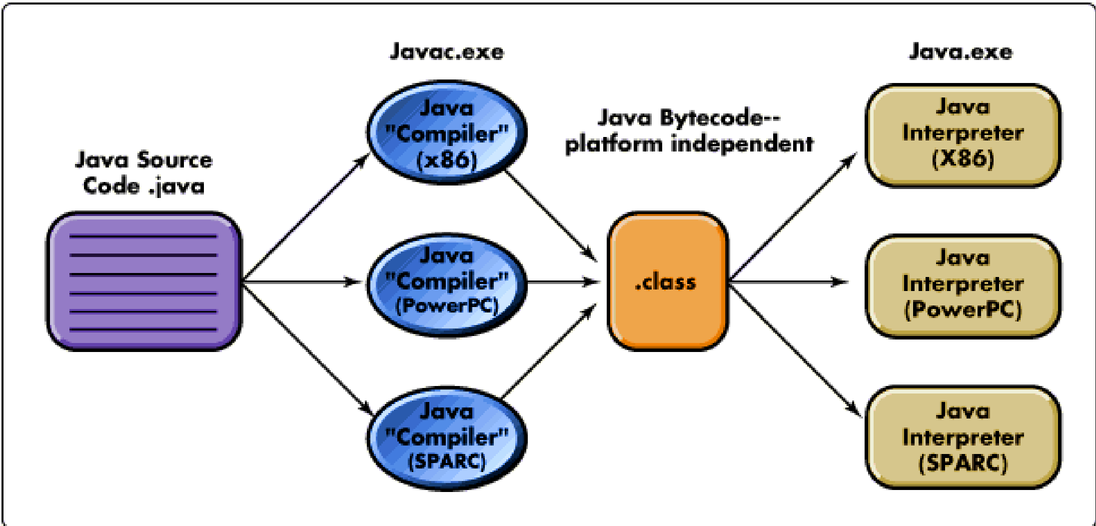


Java has been largely successful in avoiding these difficulties by employing a hybrid model of translation from source code to machine code. First, Java's developers proposed a universal intermediate primitive language called *bytecode*. This language is a kind of primitive Esperanto that is similar to every machine's machine language. Java comes with a compiler that translates every Java program to bytecode.

Then comes a second phase. For every distinct machine with a distinct architecture, there is a final, interpreter-driven translation phase. A different final phase has been written for every computer. This final phase deals with the differences between machines. The diagram below illustrates this two-phased approach to Java compilation.

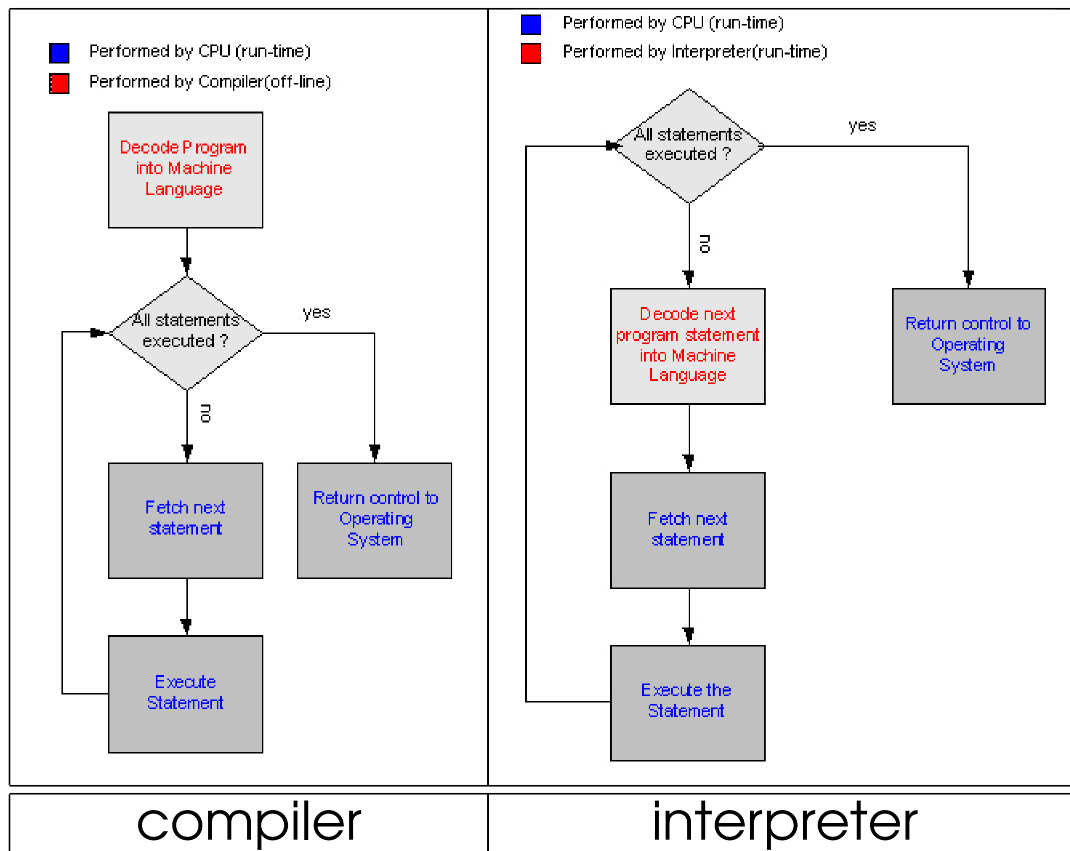
The Process to compile a java program...

RUN ANY WHERE

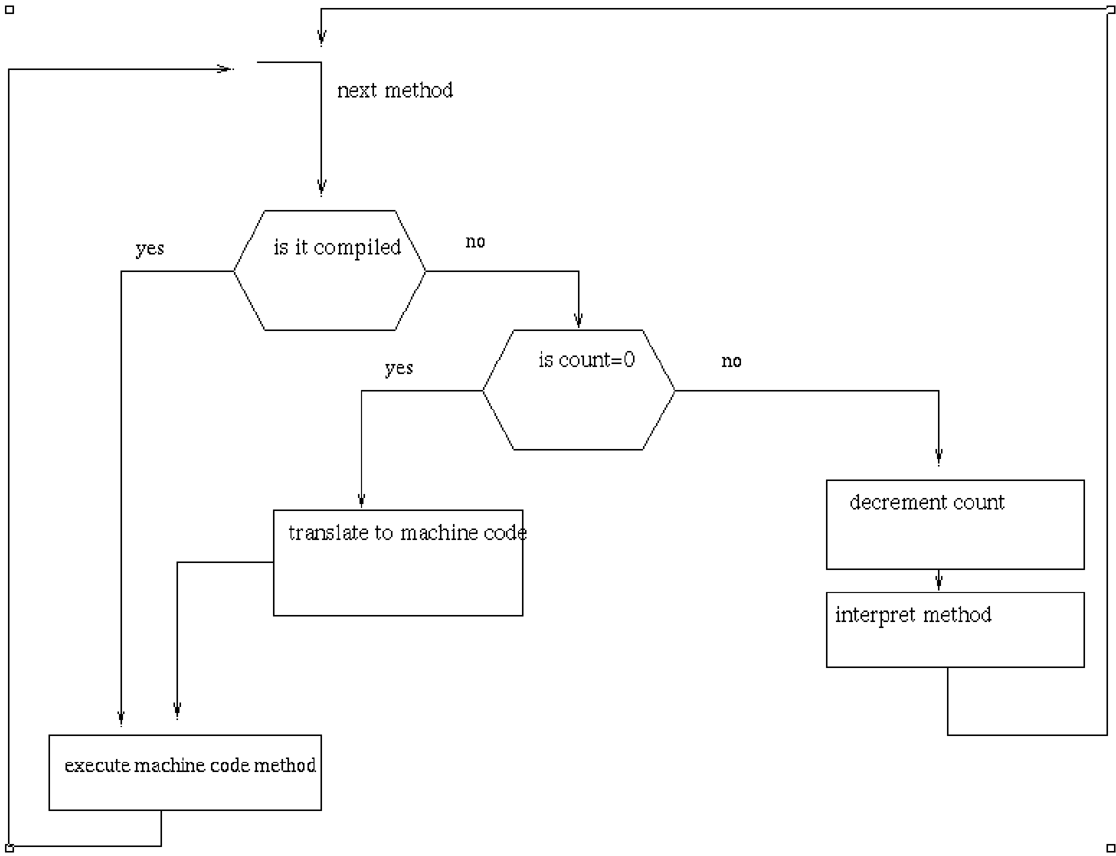


The model used by compilers and interpreters is different.

In a fully compiled language the source language is translated directly into machine code



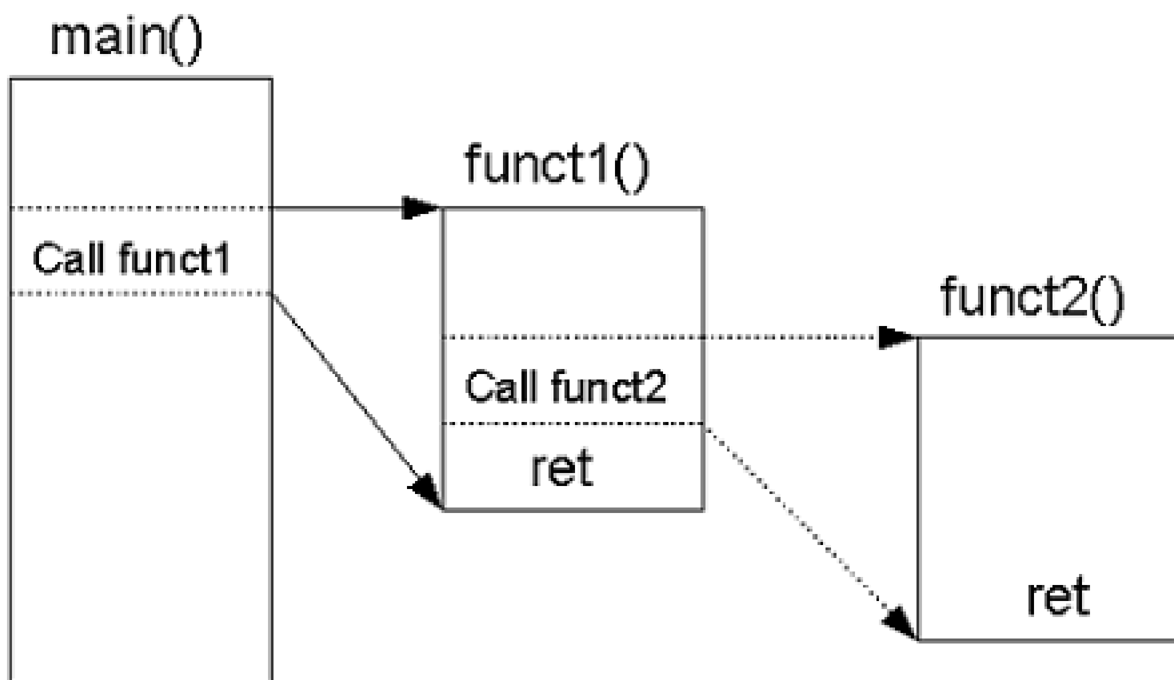
Just in time compiler as in Sun's HotSpot combines both an interpreter and a compiler



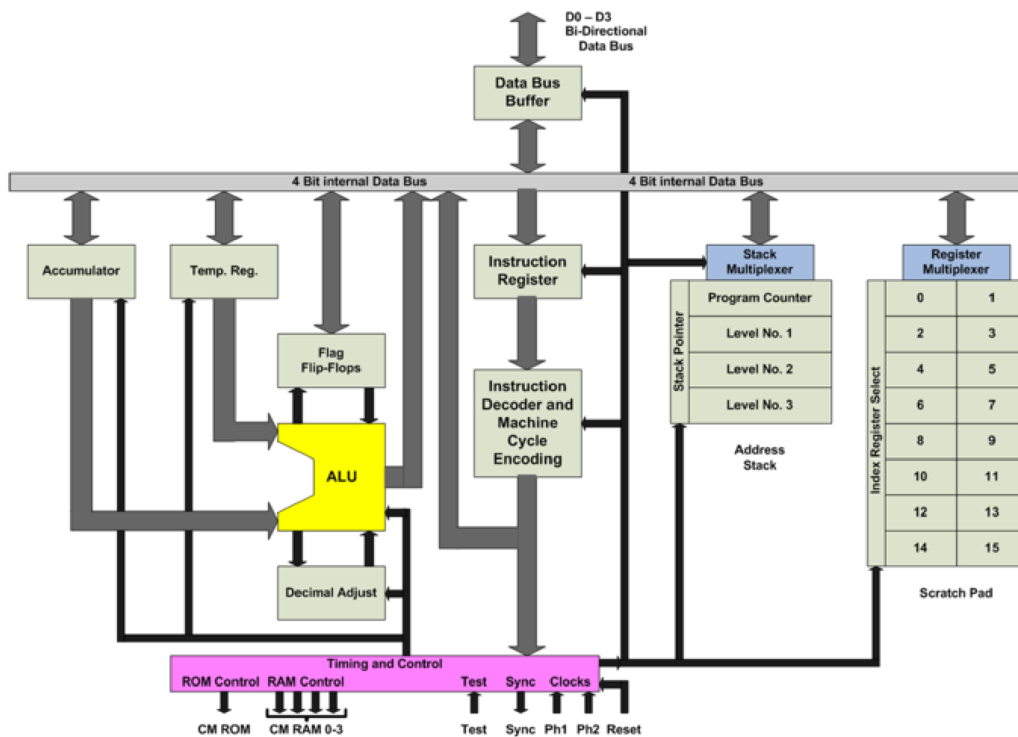
Methods, procedures, functions, subroutines

All of the above are alternative names for the same basic concept.

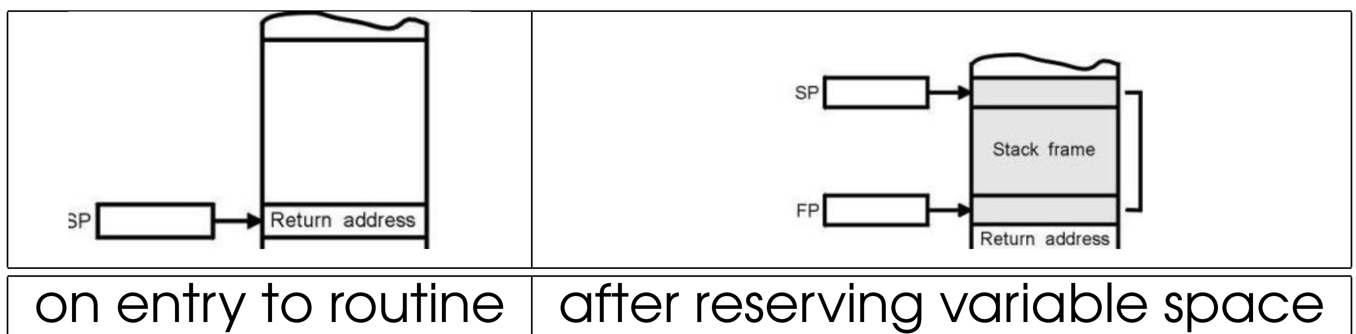
The basic idea is that the program branches to a position and remembers where it came from.



In the very first micropocessor (Intel 4004) this was done by having a hardware stack on the processor chip onto which the program counter could be saved when a subroutine call was made.



Later microprocessors used a stack in main memory which allowed much greater nesting of subroutine calls. But you can still get stack overflow!



on entry to routine	after reserving variable space

The stack frame is where the local variables of a method are stored.

HEAPS

Java is an Object Oriented Language which uses a Heap for storage allocation. If you are to understand how java works you need at least a basic understanding of memory allocation techniques.

These are hidden from you by the Java system, but they still have a big influence on what you can and can not do.

MEMORY ALLOCATION

1. C style

2. Garbage collectors

   (a) Ref count

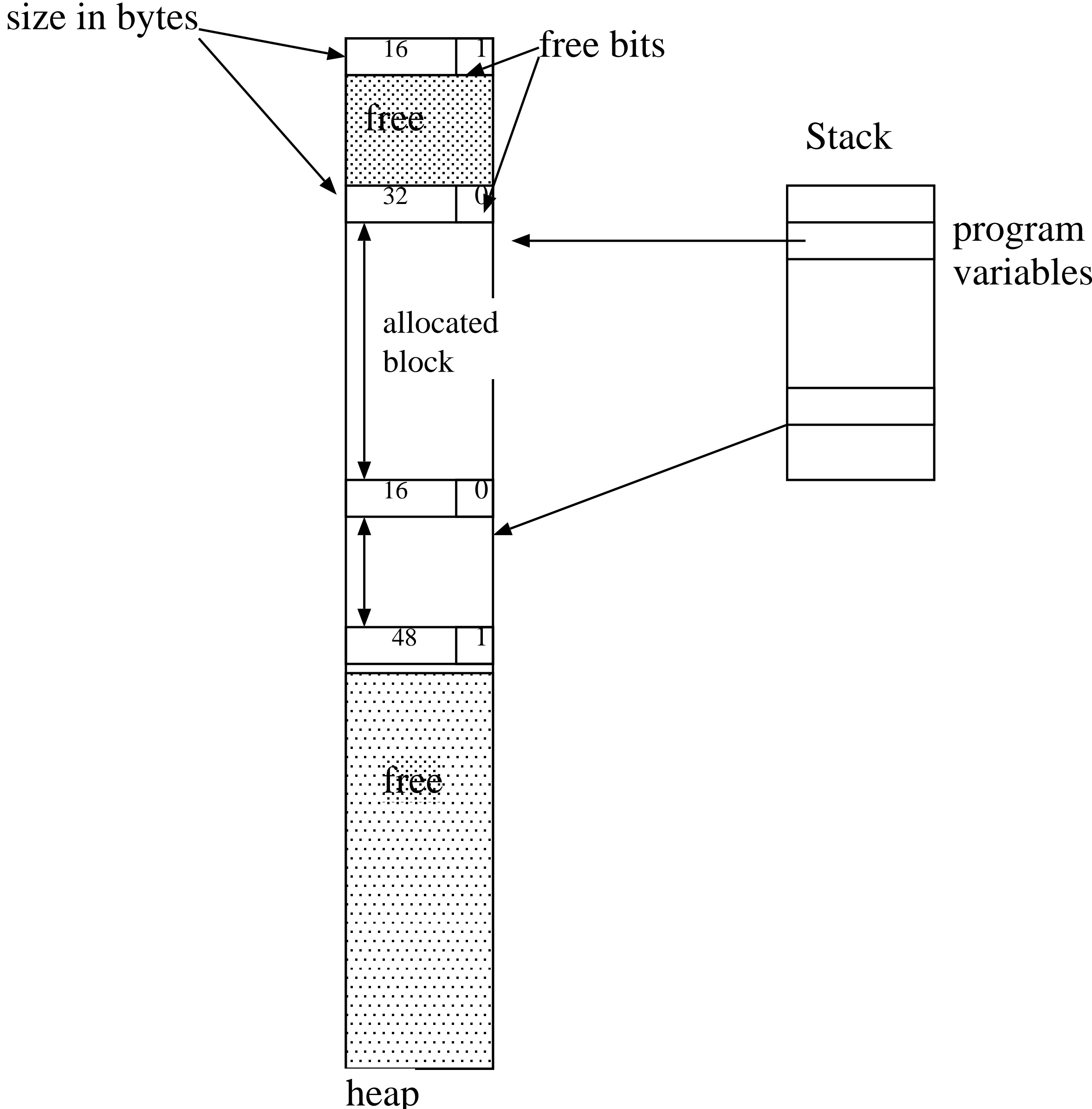
   (b) Mark sweep

   (c) Conservative

3. Persistent store

Malloc

Data structure used



MALLOC algorithm

```
int heap[HSIZE];
char *scan(int s)
{int i;
 for(i=0;i<HSIZE;i+=heap[i]> >2)
  if(heap[i]&1&&heap[i]&0xfffffffe>(s+4))
  {
   heap[i]^=1;
   return &heap[i+1];
  }
 return 0;
}
char *malloc(int size)
{ char *p=scan(size);
  if(p)return p;
  merge();
  p=scan(size);
  if(p)return p;
  heapoverflow();
}
```

FREE ALGORITHM

This simply toggles the free bit.

```
free(char *p){heap[((int)p>>2)-1]^=1;}
```

Merge algorithm

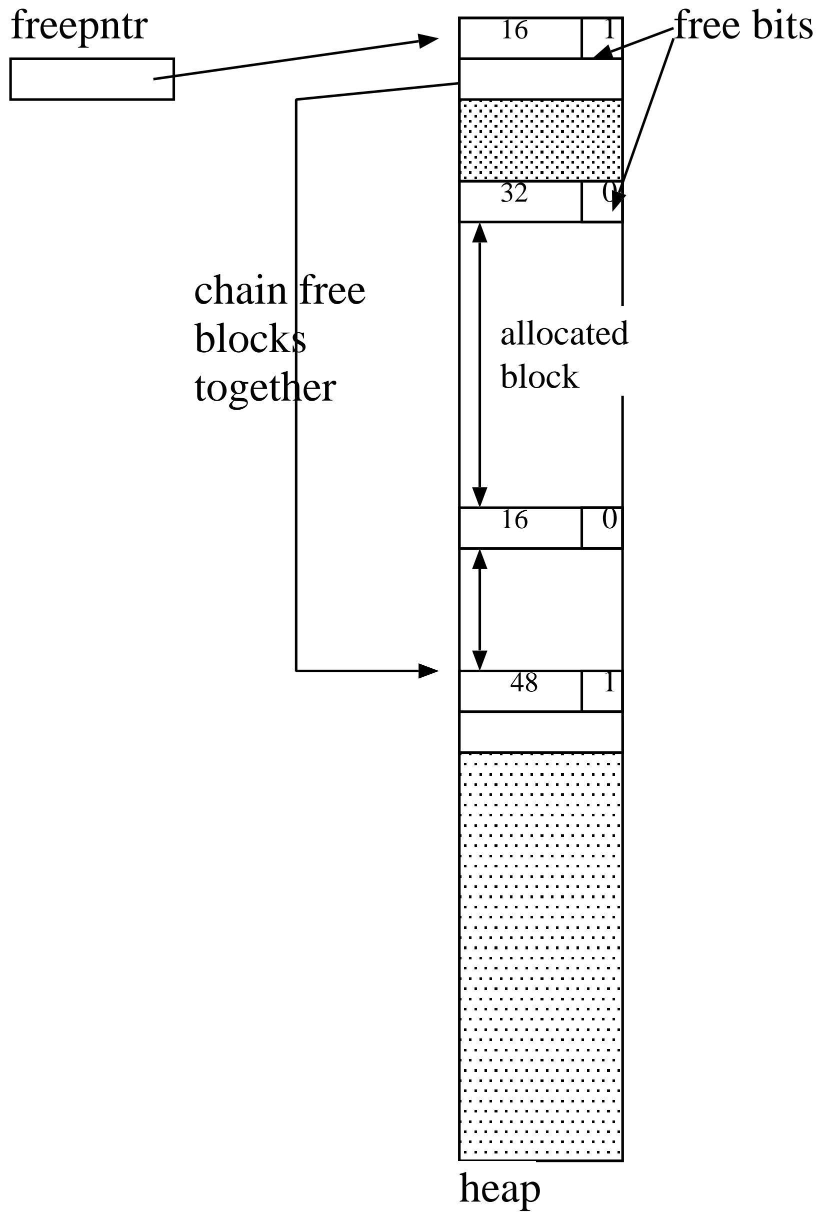
```
merge()
{ int i;
  for(i=0;i<HSIZE;i+=heap[i]>>2)
   if(heap[i]&1&&heap[heap[i]>>2]&1)
    heap[i]+=(heap[heap[i]>>2]^1);
}
```

Problem

May have to chase long list of allocated blocks before a free block is found.

Solution

Use a free list


freepntr
16
1
free bits
32
0
chain free
blocks
together
allocated
block
16
0
48
1
heap

Problem

the head of the free list will accumulate lots of small and unusable blocks, scanning these will slow access down

Solution

use two free pointers

1. points at the head of the free-list

2. points at the last allocated item on the free list

when allocating use the second pointer to initiate the search, only when it reaches the end of the heap, do we re-initialise it from the first pointer.

Idea of garbage collection

1. Relieve the coder of the problem of keeping track of unused blocks.

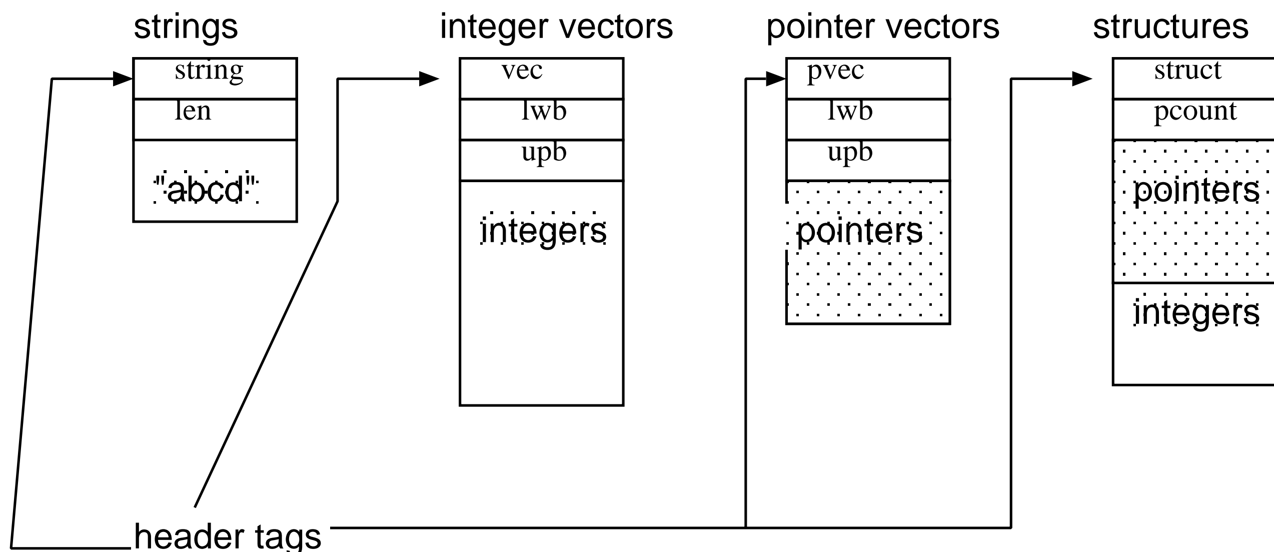
2. Fix the memory leaks which tend to occur when the coder has to do this.

3. Automatically free heap blocks which are unreachable from program variables.

Organisation of heap objects for garbage collection

Each object is given a header word that can indicate whether it contains pointers. We need to be able to find all pointers in a heap block.

A possible approach is:



Each object has its pointers segregated from the non-pointers - for instance the class

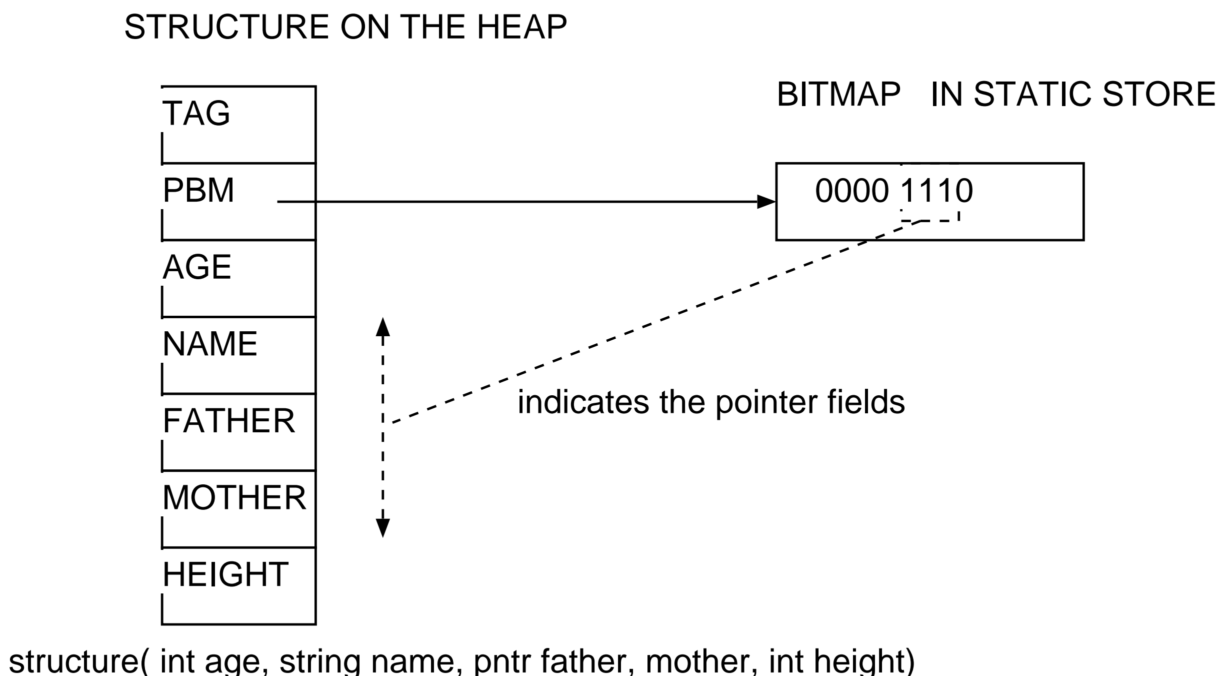
```
person{
  int age;
  string name;
  person father, mother;
  int cmsheight;};
```

could be represented on the heap by a C struct looking like

```
struct person{ short tag;    // = STRUCT_TAG
               short pntrs;  // = 3
               void * name,*father,*mother;
               int age, cmsheight;}
```

An implication of this approach is that the programmer does not know in what order the data fields will actually occur. This does not matter in most cases, but where one has to write structures to disk or use the language to control i/o devices it can be desirable to have a known mapping from declaration order to field addresses.

An alternative it to retain the original order of fields but to have a pointer to a bitmap that specifies which words in the object are pointers:



Roots

A garbage collector must preserve all data reachable from certain *roots* these are pointers that are referred to by program variables that are currently in scope. These divide into:

1. global variables - typically stored in the data section but may be in several distributed chunks

2. variables in procedure invocations on the stack

In both cases we have the problem of specifying which variables are pointers and distinguishing these from other data on the stack.

Global variables can be subdivided into

1. Globals in languages like Ada, C, Delphi which allow multiple compilation units. Such languages usually either provide no garbage collector, or if they do provide one, only have a conservative one. An exception to this was Algol68 which had a non-conservative garbage collector.

2. Globals in classic block structured languages (Standard Pascal for instance). These can be treated as special cases of procedure invocations with the main program being just a procedure that includes all the others.

Locals

We need some method of scanning the stack and distinguishing between pointers and non-pointers. Here are some techniques

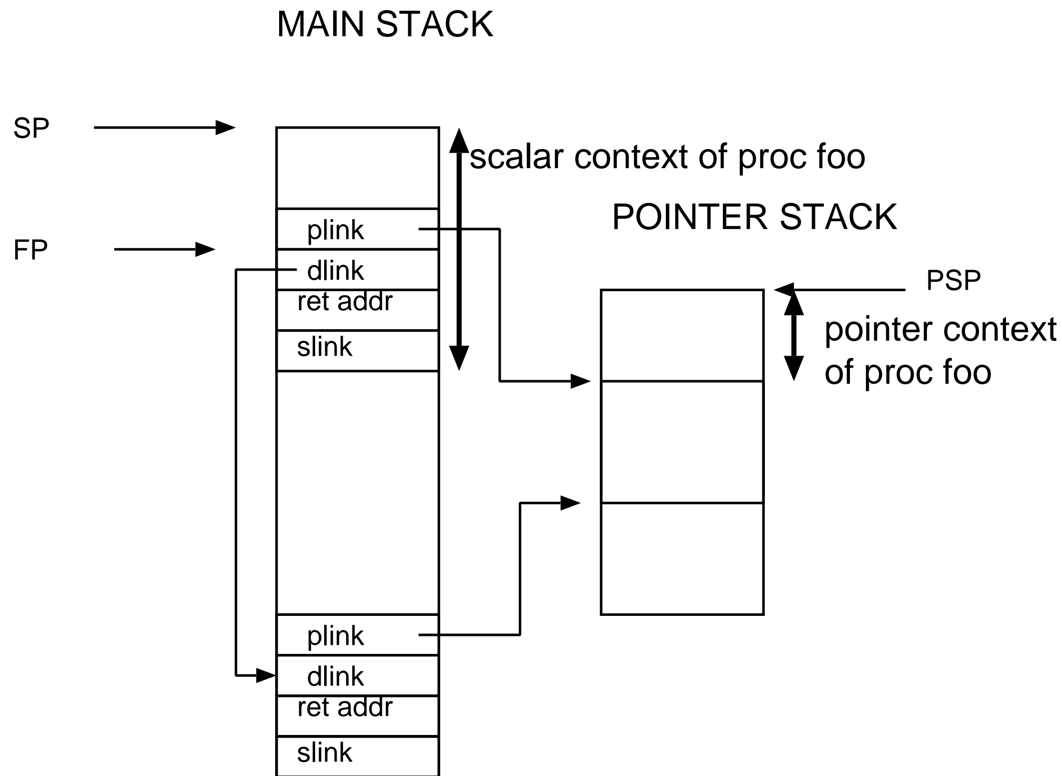
1. Use a bitmap to describe the procedure context, push a pointer to this on entry to the procedure. This follows the same technique as used above for structs.

2. Use two stacks, one to hold all pointers, the other for non-pointers.

3. Tag every word on the stack - used on the Linn Rekursiv computer.

4. Use iterator functions associated with each activation record.

2 stacks

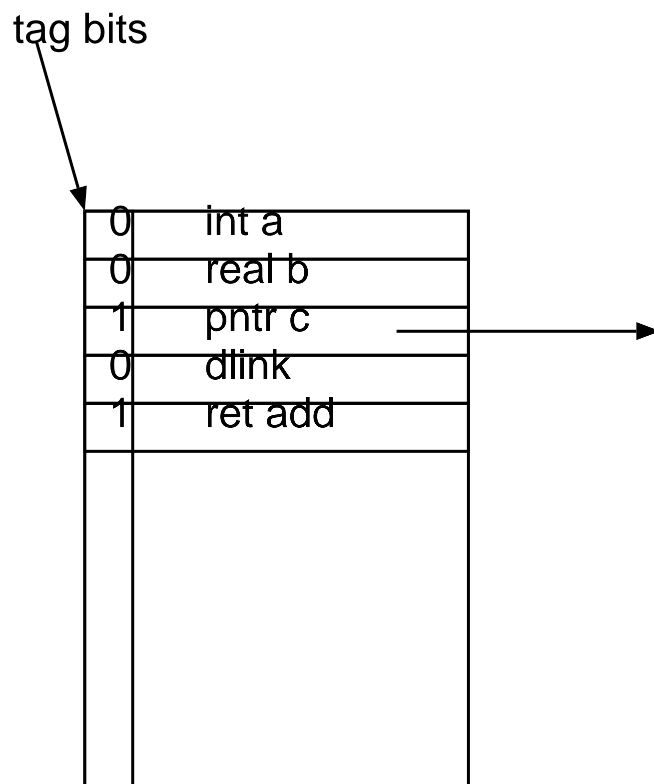


Need a second stack pointer register (PSP)

All pointer vars stored on the stack and accessed via plink on main stack

Garbage collector visits all pointers on pstack

Tagged stack



Local iterator

Given the source procedure

```
foo(scalar x, vec1 p,vec1 q->scalar)
{
..... etc
}
```

transform to

```
foo(scalar x, vec1 p,vec1 q->scalar)
{
  trav((vec1 x->void)gc->void)
  { gc(p);gc(q)}
..... etc
}
```

Ensure that the closure of trav is the first closure in the frame of each procedure. Garbage collector calls `trav` for each frame passing in a visitor function `gc` which does the actual garbage collection.

Ref counting

Each a node on the heap has an associated count of pointers to it. When the count falls to 0 the object is freed. Rules:

1. When an object is allocated on the heap its count=0
2. When a pointer is pushed on the stack, increment the count of the object associated
3. When a pointer is dropped from the stack, decrement the count of the associated object
4. on `a:=b`, decrement the count of `a` and increment the count of `b`.
5. on `free(x)` decrement count of all pointers in `x`

Advantages of reference counts

- Performance is relatively predictable, not much sudden usage of CPU for garbage collection

- Items freed as soon as possible, allows smaller heaps to be used.

Disadvantages

- Does not collect circular pointer structures as for these the reference count never reaches zero. This is not a problem in some languages.

Mark Sweep collection

This is probably the most popular garbage collection technique, it uses a two pass approach

1. Visit each heap object recursively reachable from the stack and set a mark bit in the header word.

2. Scan the heap and set the free bit for any object that does not have the mark bit set, clearing the mark bits as you do so.

At the end all unreachable objects will have their free bit set.

Mark pseudo code

```
mark ()
 for each pointer p in stack do
  rmark(p);
rmark(p)
 if p.markbit=1 return;
 p.markbit:=1;
 for each pointer q in p do
   rmark(q);
```

Advantages of Mark-Sweep

- Finds and disposes of all garbage
- Relatively simple to understand and implement
- Addresses of objects do not change

Disadvantages of Mark-Sweep

- Can lead to fragmentation
- Can lead to pauses in execution whilst it runs

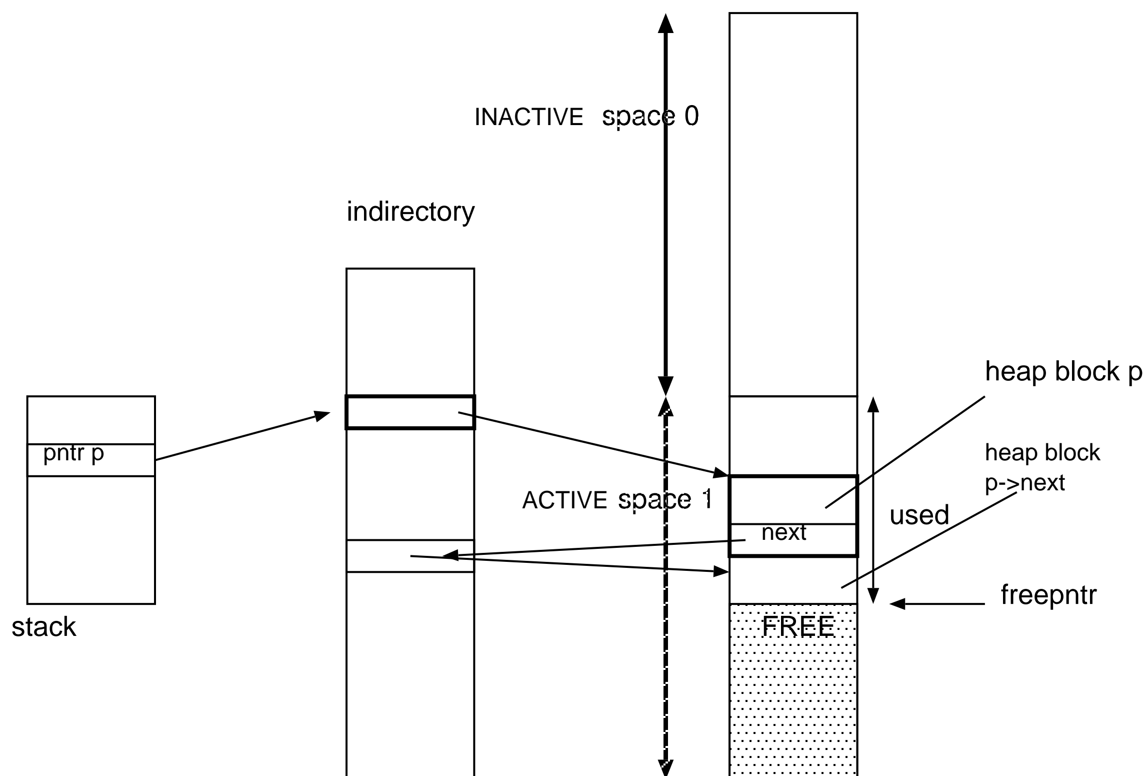
Response: Semi space algorithms

These are designed to overcome the fragmentation problem, they can be extended to ovecome the pauses.

Semi-space heap layout

1. uses 2 heap spaces

2. pointers are indirect via indirectory

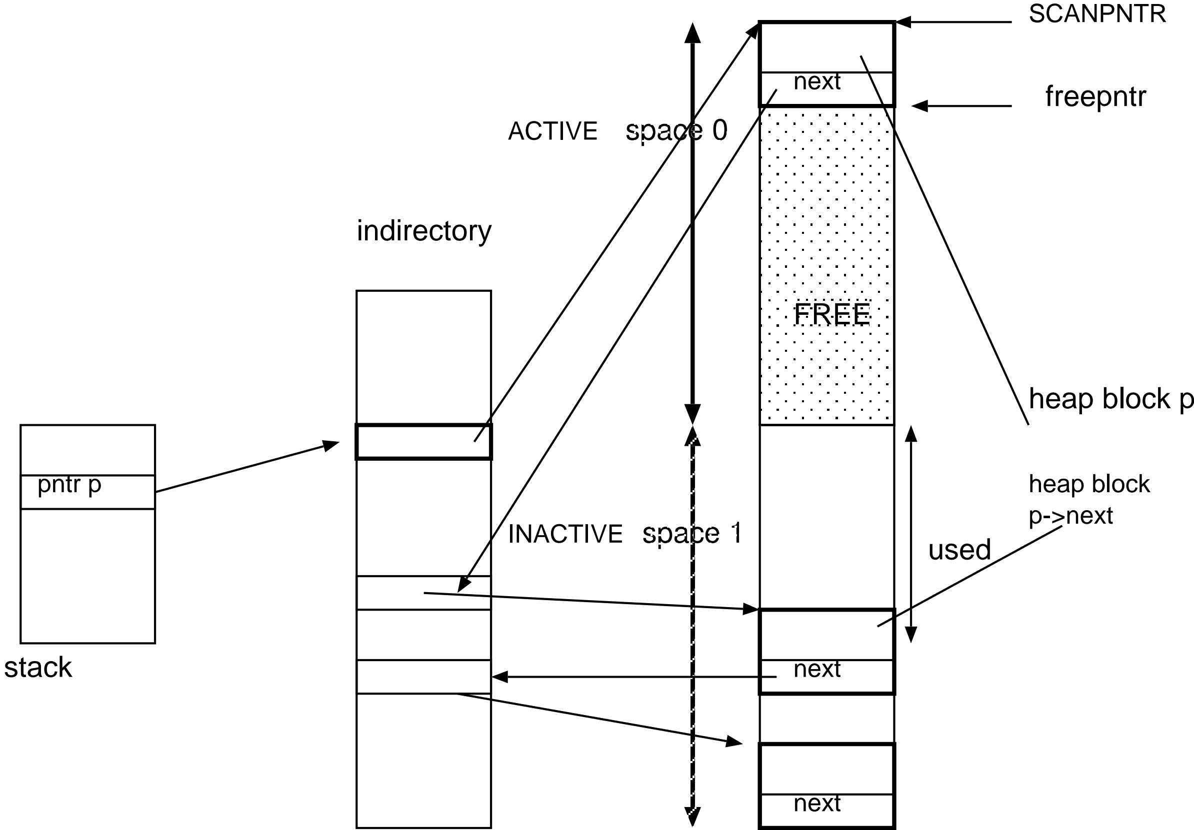


On heap full

1. toggle the active heap

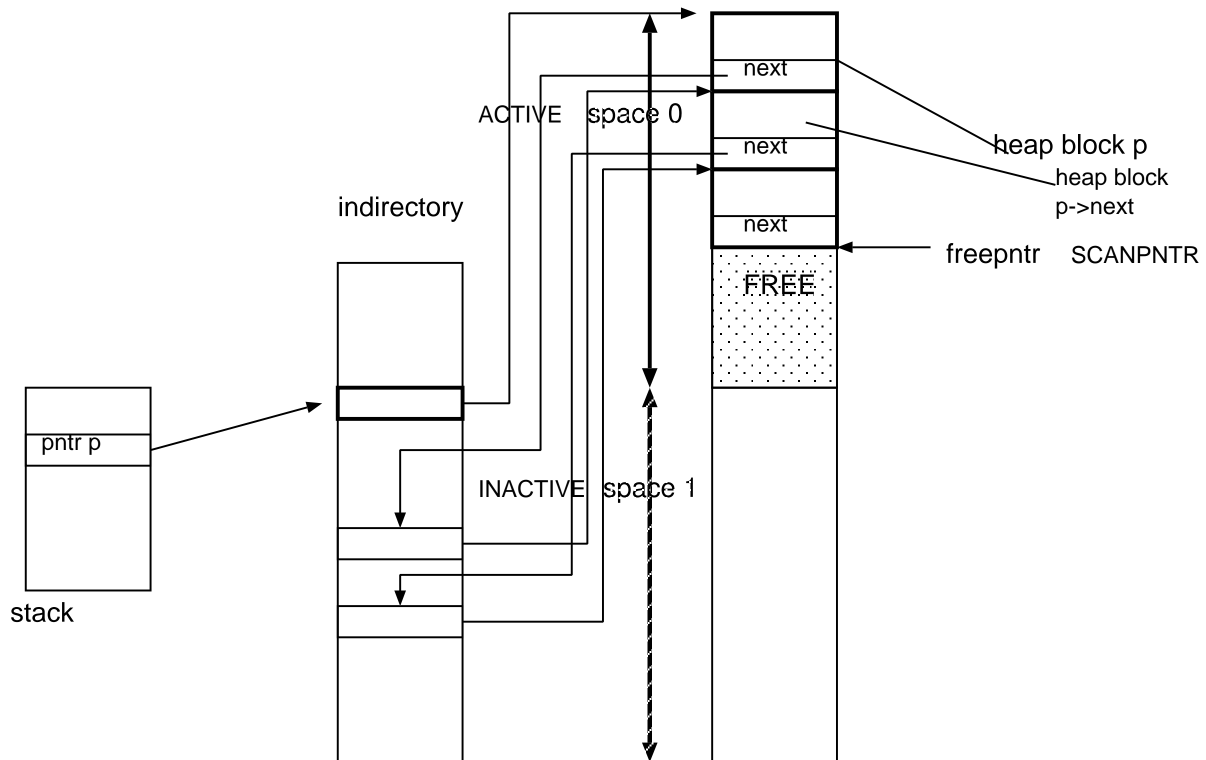
2. Copy each object pointed to by the stack to the new heap, adjusting the indirectory entry

Now looks like:



Advance scanpntr

In the second pass, the scan pointer is moved forward until it reaches the freepntr, copying every uncopied object for which it encounters a pointer. At the end of garbage collection heap looks like this.



Recover indirectory

The indirectory is another resource that has to be recovered. After the semi-space copying is done, one can go through the indirectory and chain together all entries that point into the inactive areas. These then become free indirectory entries that can be used to point to new objects.

Advantages

1. Data is compacted so that there are no fragmented chunks,
2. Concurrent implementations are possible (see what follows)
3. Better performance on paged virtual memory
4. Will garbage collect loop structures

Disadvantages

1. Twice as much space is used
2. Indirectory slows down access

Who uses copying garbage collection?

Derivatives of this basic copying algorithm are popular in object oriented languages.

Java uses an algorithm similar to the one described. This has implications for C interfaces to java, since Java objects can be moved by the garbage collector during program runs, C code must not retain pointers to them.

Concurrent garbage collection

With semi-space collectors it is possible to run the garbage collector concurrently with the main program. This prevents annoying pauses

The concurrent garbage collection is spawned after the stage of copying over all objects reachable from the stack has been achieved.

It then runs in parallel with the main program advancing the scan pointer and copying objects from the inactive to the active area.

The compiler must ensure that if a pointer field in an object in the active area is assigned a pointer to an object currently in the inactive area, that object must be copied accross to the active area before the assignment.

If this is not done, the concucurrent collector could loose pointers to lexically reachable objects.

Conservative Collectors

The garbage collectors desribed so far are suitable for type safe languages. In C the existence of union types and the tendency of programmers to copy pointers into integers means that type information can not be relied upon to find pointers.

This does not rule out garbage collection for C.

If you know that the heap exists within a certain address range, `heapbase..heaptop` then this information can be exploited to find pointers.

Scan the stack, check each word on it to see if it is in the range of heap addresses, if it is, assume it is a pointer and mark the object it points to.

Problems

1. You will include a few integers that just happen to be in the right address range. Call these *pseudo objects*.

   (a) At one level, this does not matter too much, it simply means that not all the garbage will be collected.

2. If you try to mark a *pseudoobject* they you might set a mark bit in the middle of a valid object and corrupt it.

   (a) Solution: use a distinct bitmap to hold the mark bits.

Example approach

- Allocate store in 16 byte chunks. The bitmaps have a bit for each 16 byte block of the heap.

- Use 3 data structures

  1. Heap proper

  2. Start bitmap, with a bit set for the start of each allocated heap object

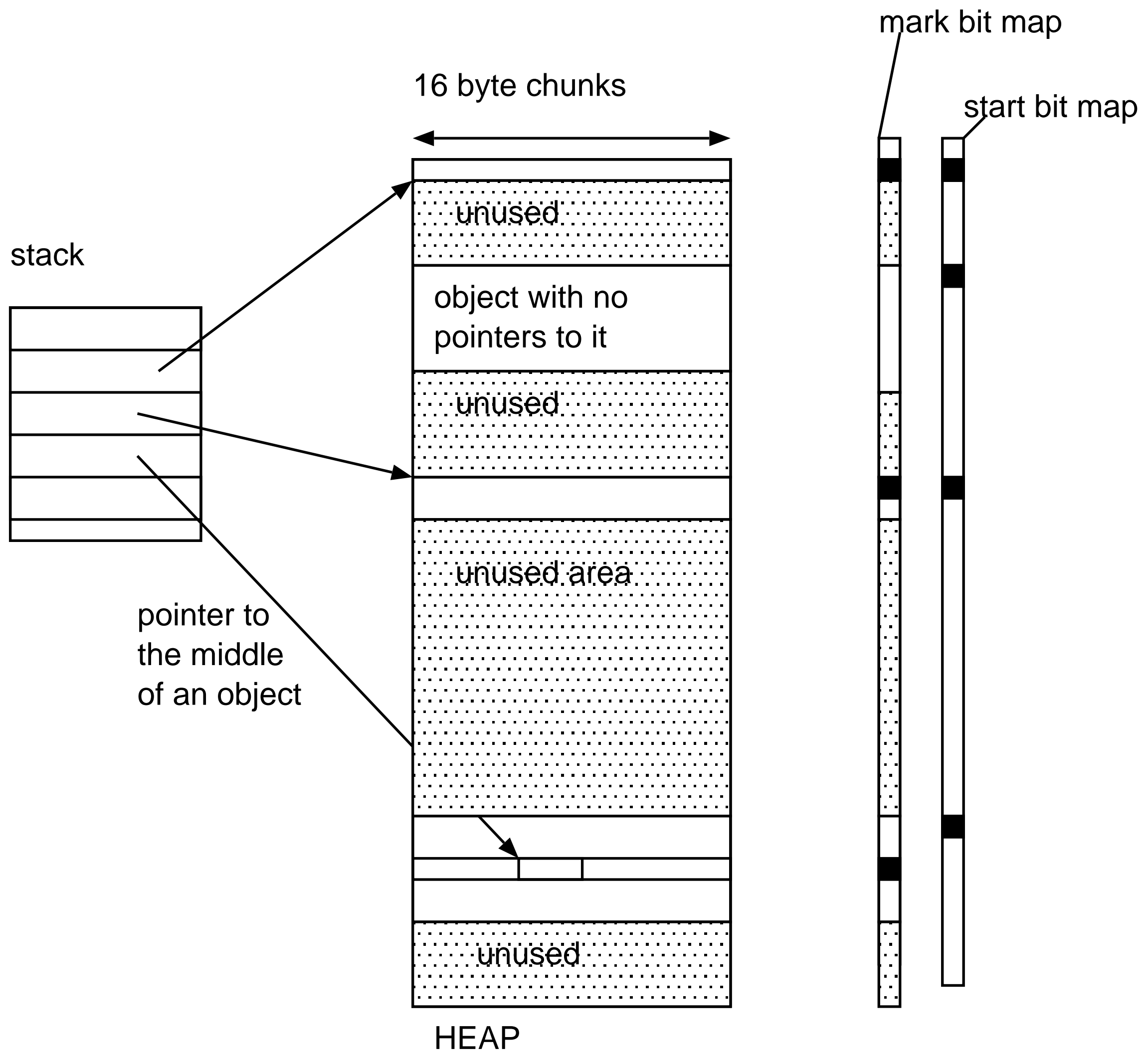
  3. Mark bitmap, with a bit set for each chunk into which a pointer is found

- When marking P

  1. set bit P in the mark bitmap,
  2. and determine the start of the block by scanning backwards through the start bitmap.
  3. then call mark on all words in the heap block

- When sweeping, free a heap block if none of its 16 byte chunks is marked.

- Marking must search for pointers on every possible byte boundary.


mark bit map
16 byte chunks
start bit map
unused
stack
object with no
pointers to it
unused
unused area
pointer to
the middle
of an object
unused
HEAP

Persistent Heaps

A number of programming languages support Persistence: the notion that data on the heap outlive the program and can be loaded into the memory space of another program.

APL supported this with its 'Workspace' concept

Smalltalk supported it

PS-algol introduced it to imperative languages

PJama - persistent java from Sun supported it, as did a number of other Java implementations.

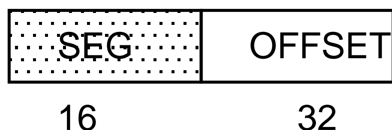
Persistence is based on a generalisation of garbage collection techniques to cover filestore as well as RAM. It allows arbitrary databases to be built up in the heap, giving all the benefits of the program languages : structuring, strong type checking etc, with the long term storage of files.

We will look at two approaches to persistence - the Smalltalk and the PS-algol approach.

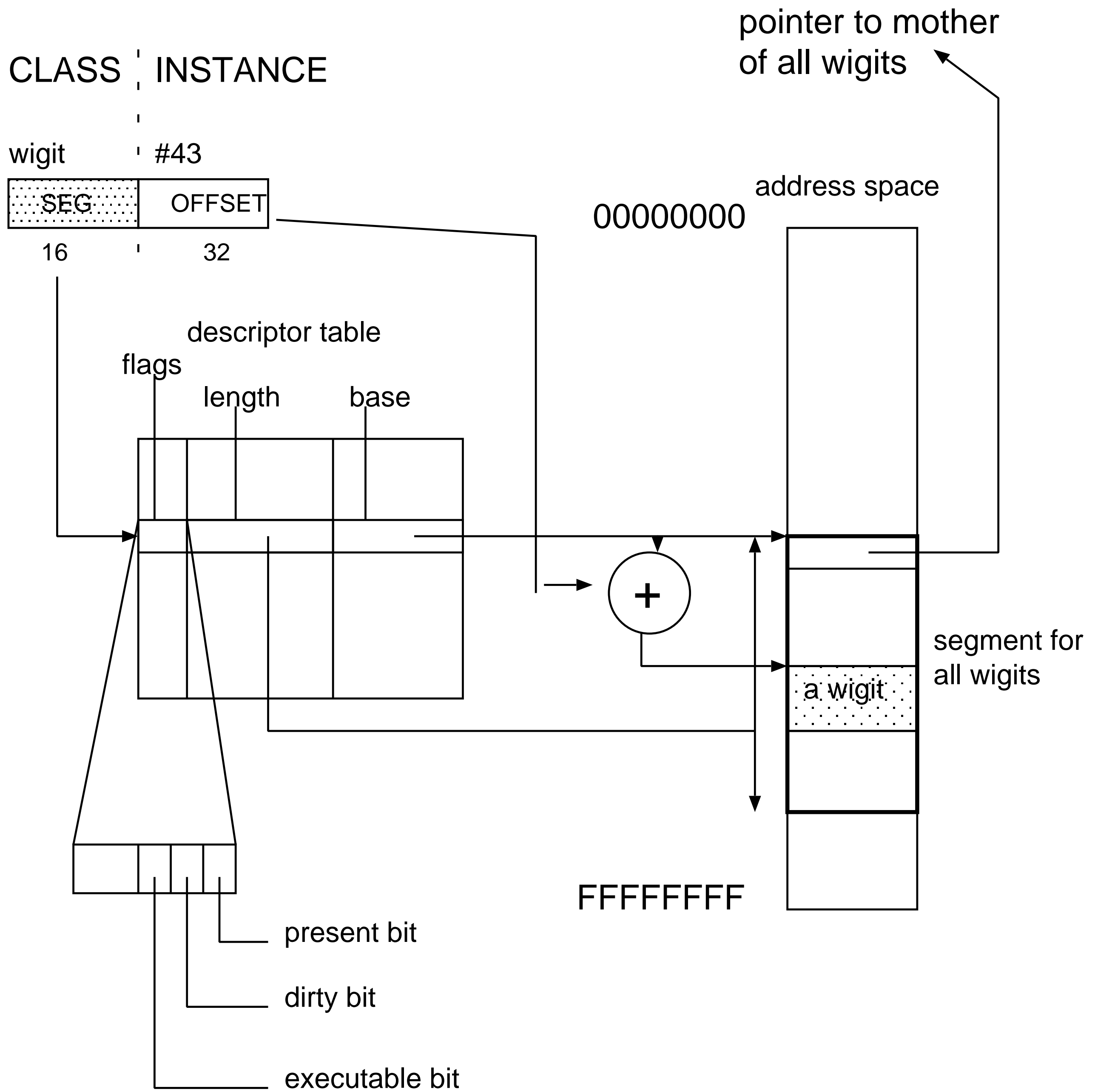
Smalltalk store

Segmented virtual memory approach

Addresses in the Smalltalk virtual machine split into two parts segment and offset. Originally these were only 20 bits in all, but here I give an example based on the segmented address structure of the Intel CPUs.



These are interpreted by the memory management system as follows



If the present bit is not set, an address fault occurs and the run time system can load the segment into memory from disk.

This is like paging except that the segments are of variable size, and can be much bigger than pages, or alternatively very small.

The language run time system allocates addresses so that

1. All objects of the same class share the same segment. Thus the segment number in the pointer to an object encodes its type.

2. At the start of each segment is a pointer to the class descriptor of the type, which can be used to find methods etc.

3. At program terminate, segments are swapped back to disk to ensure that the programmers environment persists.

4. Garbage collection by reference counting.