The balanced tree approach is correct in the sense that it is guaranteed to mark all of the nodes that are reachable from the roots. This is a necessary guarantee, as application users would certainly not appreciate having their allocated blocks prematurely returned to the free list. However, it is conservative in the sense that it may incorrectly mark blocks that are actually unreachable, and thus it may fail to free some garbage. While this does not affect the correctness of application programs, it can result in unnecessary external fragmentation.

The fundamental reason that Mark&Sweep collectors for C programs must be conservative is that the C language does not tag memory locations with type information. Thus, scalars like `int`s or `float`s can masquerade as pointers. For example, suppose that some reachable allocated block contains an `int` in its payload whose value happens to correspond to an address in the payload of some other allocated block *b*. There is no way for the collector to infer that the data is really an `int` and not a pointer. Therefore, the allocator must conservatively mark block *b* as reachable, when in fact it might not be.

9.11 Common Memory-Related Bugs in C Programs

Managing and using virtual memory can be a difficult and error-prone task for C programmers. Memory-related bugs are among the most frightening because they often manifest themselves at a distance, in both time and space, from the source of the bug. Write the wrong data to the wrong location, and your program can run for hours before it finally fails in some distant part of the program. We conclude our discussion of virtual memory with a look at of some of the common memory-related bugs.

9.11.1 Dereferencing Bad Pointers

As we learned in Section 9.7.2, there are large holes in the virtual address space of a process that are not mapped to any meaningful data. If we attempt to dereference a pointer into one of these holes, the operating system will terminate our program with a segmentation exception. Also, some areas of virtual memory are read-only. Attempting to write to one of these areas terminates the program with a protection exception.

A common example of dereferencing a bad pointer is the classic `scanf` bug. Suppose we want to use `scanf` to read an integer from `stdin` into a variable. The correct way to do this is to pass `scanf` a format string and the *address* of the variable:

```
scanf("%d", &val)
```

However, it is easy for new C programmers (and experienced ones too!) to pass the *contents* of `val` instead of its address:

```
scanf("%d", val)
```

In this case, `scanf` will interpret the contents of `val` as an address and attempt to write a word to that location. In the best case, the program terminates immediately with an exception. In the worst case, the contents of `val` correspond to some valid read/write area of virtual memory, and we overwrite memory, usually with disastrous and baffling consequences much later.

9.11.2 Reading Uninitialized Memory

While bss memory locations (such as uninitialized global C variables) are always initialized to zeros by the loader, this is not true for heap memory. A common error is to assume that heap memory is initialized to zero:

```
1   /* Return y = Ax */
2   int *matvec(int **A, int *x, int n)
3   {
4       int i, j;
5
6       int *y = (int *)Malloc(n * sizeof(int));
7
8       for (i = 0; i < n; i++)
9           for (j = 0; j < n; j++)
10              y[i] += A[i][j] * x[j];
11      return y;
12  }
```

In this example, the programmer has incorrectly assumed that vector `y` has been initialized to zero. A correct implementation would explicitly zero `y[i]` or use `calloc`.

9.11.3 Allowing Stack Buffer Overflows

As we saw in Section 3.10.3, a program has a *buffer overflow bug* if it writes to a target buffer on the stack without examining the size of the input string. For example, the following function has a buffer overflow bug because the `gets` function copies an arbitrary-length string to the buffer. To fix this, we would need to use the `fgets` function, which limits the size of the input string.

```
1   void bufoverflow()
2   {
3       char buf[64];
4
5       gets(buf); /* Here is the stack buffer overflow bug */
6       return;
7   }
```

9.11.4 Assuming That Pointers and the Objects They Point to Are the Same Size

One common mistake is to assume that pointers to objects are the same size as the objects they point to:

```
1   /* Create an nxm array */
2   int **makeArray1(int n, int m)
3   {
4       int i;
5       int **A = (int **)Malloc(n * sizeof(int));
6
7       for (i = 0; i < n; i++)
8           A[i] = (int *)Malloc(m * sizeof(int));
9       return A;
10  }
```

The intent here is to create an array of n pointers, each of which points to an array of m `int`s. However, because the programmer has written `sizeof(int)` instead of `sizeof(int *)` in line 5, the code actually creates an array of `int`s.

This code will run fine on machines where `int`s and pointers to `int`s are the same size. But if we run this code on a machine like the Core i7, where a pointer is larger than an `int`, then the loop in lines 7–8 will write past the end of the `A` array. Since one of these words will likely be the boundary-tag footer of the allocated block, we may not discover the error until we free the block much later in the program, at which point the coalescing code in the allocator will fail dramatically and for no apparent reason. This is an insidious example of the kind of "action at a distance" that is so typical of memory-related programming bugs.

9.11.5 Making Off-by-One Errors

Off-by-one errors are another common source of overwriting bugs:

```
1   /* Create an nxm array */
2   int **makeArray2(int n, int m)
3   {
4       int i;
5       int **A = (int **)Malloc(n * sizeof(int *));
6
7       for (i = 0; i <= n; i++)
8           A[i] = (int *)Malloc(m * sizeof(int));
9       return A;
10  }
```

This is another version of the program in the previous section. Here we have created an n-element array of pointers in line 5 but then tried to initialize $n + 1$ of its elements in lines 7 and 8, in the process overwriting some memory that follows the `A` array.

9.11.6 Referencing a Pointer Instead of the Object It Points To

If we are not careful about the precedence and associativity of C operators, then we incorrectly manipulate a pointer instead of the object it points to. For example, consider the following function, whose purpose is to remove the first item in a binary heap of `*size` items and then reheapify the remaining `*size - 1` items:

```
1  int *binheapDelete(int **binheap, int *size)
2  {
3      int *packet = binheap[0];
4
5      binheap[0] = binheap[*size - 1];
6      *size--; /* This should be (*size)-- */
7      heapify(binheap, *size, 0);
8      return(packet);
9  }
```

In line 6, the intent is to decrement the integer value pointed to by the `size` pointer. However, because the unary `--` and `*` operators have the same precedence and associate from right to left, the code in line 6 actually decrements the pointer itself instead of the integer value that it points to. If we are lucky, the program will crash immediately. But more likely we will be left scratching our heads when the program produces an incorrect answer much later in its execution. The moral here is to use parentheses whenever in doubt about precedence and associativity. For example, in line 6, we should have clearly stated our intent by using the expression `(*size)--`.

9.11.7 Misunderstanding Pointer Arithmetic

Another common mistake is to forget that arithmetic operations on pointers are performed in units that are the size of the objects they point to, which are not necessarily bytes. For example, the intent of the following function is to scan an array of `int`s and return a pointer to the first occurrence of `val`:

```
1  int *search(int *p, int val)
2  {
3      while (*p && *p != val)
4          p += sizeof(int); /* Should be p++ */
5      return p;
6  }
```

However, because line 4 increments the pointer by 4 (the number of bytes in an integer) each time through the loop, the function incorrectly scans every fourth integer in the array.

9.11.8 Referencing Nonexistent Variables

Naive C programmers who do not understand the stack discipline will sometimes reference local variables that are no longer valid, as in the following example:

```
1  int *stackref ()
2  {
3      int val;
4
5      return &val;
6  }
```

This function returns a pointer (say, `p`) to a local variable on the stack and then pops its stack frame. Although `p` still points to a valid memory address, it no longer points to a valid variable. When other functions are called later in the program, the memory will be reused for their stack frames. Later, if the program assigns some value to `*p`, then it might actually be modifying an entry in another function's stack frame, with potentially disastrous and baffling consequences.

9.11.9 Referencing Data in Free Heap Blocks

A similar error is to reference data in heap blocks that have already been freed. Consider the following example, which allocates an integer array `x` in line 6, prematurely frees block `x` in line 10, and then later references it in line 14:

```
1  int *heapref(int n, int m)
2  {
3      int i;
4      int *x, *y;
5
6      x = (int *)Malloc(n * sizeof(int));
7
8      . . . // Other calls to malloc and free go here
9
10     free(x);
11
12     y = (int *)Malloc(m * sizeof(int));
13     for (i = 0; i < m; i++)
14         y[i] = x[i]++; /* Oops! x[i] is a word in a free block */
15
16     return y;
17 }
```

Depending on the pattern of `malloc` and `free` calls that occur between lines 6 and 10, when the program references `x[i]` in line 14, the array `x` might be part of some other allocated heap block and may have been overwritten. As with many

memory-related bugs, the error will only become evident later in the program when we notice that the values in y are corrupted.

9.11.10 Introducing Memory Leaks

Memory leaks are slow, silent killers that occur when programmers inadvertently create garbage in the heap by forgetting to free allocated blocks. For example, the following function allocates a heap block x and then returns without freeing it:

```
1  void leak(int n)
2  {
3      int *x = (int *)Malloc(n * sizeof(int));
4
5      return;  /* x is garbage at this point */
6  }
```

If leak is called frequently, then the heap will gradually fill up with garbage, in the worst case consuming the entire virtual address space. Memory leaks are particularly serious for programs such as daemons and servers, which by definition never terminate.

9.12 Summary

Virtual memory is an abstraction of main memory. Processors that support virtual memory reference main memory using a form of indirection known as virtual addressing. The processor generates a virtual address, which is translated into a physical address before being sent to the main memory. The translation of addresses from a virtual address space to a physical address space requires close cooperation between hardware and software. Dedicated hardware translates virtual addresses using page tables whose contents are supplied by the operating system.

Virtual memory provides three important capabilities. First, it automatically caches recently used contents of the virtual address space stored on disk in main memory. The block in a virtual memory cache is known as a page. A reference to a page on disk triggers a page fault that transfers control to a fault handler in the operating system. The fault handler copies the page from disk to the main memory cache, writing back the evicted page if necessary. Second, virtual memory simplifies memory management, which in turn simplifies linking, sharing data between processes, the allocation of memory for processes, and program loading. Finally, virtual memory simplifies memory protection by incorporating protection bits into every page table entry.

The process of address translation must be integrated with the operation of any hardware caches in the system. Most page table entries are located in the L1 cache, but the cost of accessing page table entries from L1 is usually eliminated by an on-chip cache of page table entries called a TLB.