

Lecture #37

Today: Side excursions into nitty-gritty stuff: Threads, storage management.

Threads

- So far, all our programs consist of single sequence of instructions.
- Each such sequence is called a *thread* (for “thread of control”) in Java.
- Java supports programs containing *multiple* threads, which (conceptually) run concurrently.
- Actually, on a uniprocessor, only one thread at a time actually runs, while others wait, but this is largely invisible.
- To allow program access to threads, Java provides the type `Thread` in `java.lang`. Each `Thread` contains information about, and controls, one thread.
- Simultaneous access to data from two threads can cause chaos, so are also constructs for controlled communication, allowing threads to *lock* objects, to *wait* to be notified of events, and to *interrupt* other threads.

But Why?

- Typical Java programs always have > 1 thread: besides the main program, others clean up garbage objects, receive signals, update the display, other stuff.
- When programs deal with asynchronous events, is sometimes convenient to organize into subprograms, one for each independent, related sequence of events.
- Threads allow us to insulate one such subprogram from another.
- GUIs often organized like this: application is doing some computation or I/O, another thread waits for mouse clicks (like 'Stop'), another pays attention to updating the screen as needed.
- Large servers like search engines may be organized this way, with one thread per request.
- And, of course, sometimes we *do* have a real multiprocessor.

Java Mechanics

- To specify the actions “walking” and “chewing gum”:

```
class Chewer1 implements Runnable {  
    public void run()  
    { while (true) ChewGum(); }  
}  
  
class Walker1 implements Runnable {  
    public void run()  
    { while (true) Walk(); }  
}
```

```
// Walk and chew gum  
Thread chomp  
    = new Thread(new  
Chewer1());  
Thread clomp  
    = new Thread(new  
Walker1());  
chomp.start(); clomp.start();
```

- Concise Alternative (uses fact that Thread implements Runnable):

```
class Chewer2 extends Thread {  
    public void run()  
    { while (true) ChewGum(); }  
}  
  
class Walker2 extends Thread {  
    public void run()  
    { while (true) Walk(); }  
}
```

```
Thread chomp = new Chewer2(),  
      clomp = new Walker2();  
chomp.start();  
clomp.start();
```

Avoiding Interference

- When one thread has data for another, one must wait for the other to be ready.
- Likewise, if two threads use the same data structure, generally only one should modify it at a time; other must wait.
- E.g., what would happen if two threads simultaneously inserted an item into a linked list at the same point in the list?
- A: Both could conceivably execute

```
p.next = new ListCell(x, p.next);
```

with the *same* values of p and p.next; one insertion is lost.

- Can arrange for only one thread at a time to execute a method on a particular object with either of the following equivalent definitions:

```
void f(...) {  
    synchronized (this) {  
        body of f  
    }  
}
```

```
synchronized void f(...) {  
    body of f  
}
```

Communicating the Hard Way

- Communicating data is tricky: the faster party must wait for the slower.
- Obvious approaches for sending data from thread to thread don't work:

```
class DataExchanger {  
    Object value = null;  
    Object receive() {  
        Object r; r = null;  
        while (r == null)  
            { r = value; }  
        value = null;  
        return r;  
    }  
    void deposit(Object data) {  
        while (value != null) { }  
        value = data;  
    }  
}
```

```
DataExchanger exchanger  
= new DataExchanger();  
-----  
// thread1 sends to thread2 with  
exchanger.deposit("Hello!");  
-----  
// thread2 receives from thread1 with  
msg = (String) exchanger.receive();
```

- **BAD:** One thread can monopolize machine while waiting; two threads executing deposit or receive simultaneously cause chaos.

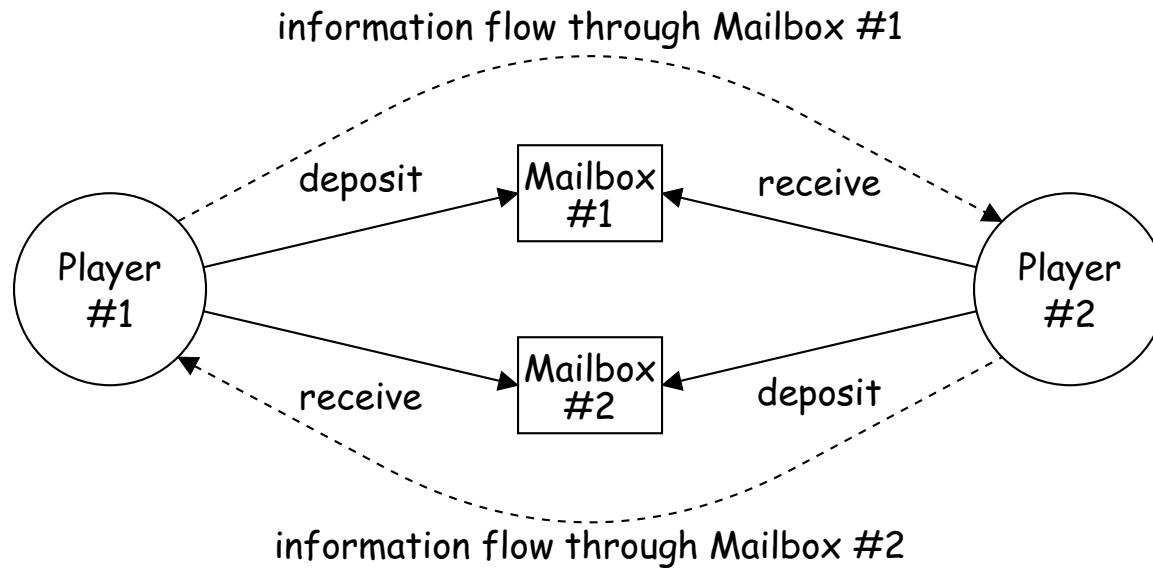
Primitive Java Facilities

- wait method on Object makes thread wait (not using processor) until notified by notifyAll, unlocking the Object while it waits.
- Example, ucb.util.mailbox has something like this (simplified):

```
interface Mailbox {  
    void deposit(Object msg) throws InterruptedException;  
    Object receive() throws InterruptedException;  
}  
  
class QueuedMailbox implements Mailbox {  
    private List<Object> queue = new LinkedList<Object>();  
  
    public synchronized void deposit(Object msg) {  
        queue.add(msg);  
        this.notifyAll(); // Wake any waiting receivers  
    }  
  
    public synchronized Object receive() throws InterruptedException {  
        while (queue.isEmpty()) wait();  
        return queue.remove(0);  
    }  
}
```

Message-Passing Style

- Use of Java primitives very error-prone. Wait until CS162.
- Mailboxes are higher-level, and allow the following program structure:



- Where each Player is a thread that looks like this:

```
while (! gameOver()) {  
    if (myMove())  
        outBox.deposit(computeMyMove(lastMove));  
    else  
        lastMove = inBox.receive();  
}
```

More Concurrency

- Previous example can be done other ways, but mechanism is very flexible.
- E.g., suppose you want to think during opponent's move:

```
while (!gameOver()) {  
    if (myMove())  
        outBox.deposit(computeMyMove(lastMove));  
    else {  
        do {  
            thinkAheadALittle();  
            lastMove = inBox.receiveIfPossible();  
        } while (lastMove == null);  
    }  
}
```

- `receiveIfPossible` (written `receive(0)` in our actual package) doesn't wait; returns null if no message yet, perhaps like this:

```
public synchronized Object receiveIfPossible()  
    throws InterruptedException {  
    if (queue.isEmpty())  
        return null;  
    return queue.remove(0);  
}
```

Coroutines

- A *coroutine* is a kind of synchronous thread that explicitly hands off control to other coroutines so that only one executes at a time, like Python generators. Can get similar effect with threads and mailboxes.
- Example: recursive inorder tree iterator:

```
class TreeIterator extends Thread {  
    Tree root; Mailbox r;  
    TreeIterator(Tree T, Mailbox r) {  
        this.root = T; this.dest = r;  
    }  
    public void run() {  
        traverse(root);  
        r.deposit(End marker);  
    }  
    void traverse(Tree t) {  
        if (t == null) return;  
        traverse(t.left);  
        r.deposit(t.label);  
        traverse(t.right);  
    }  
}
```

```
        void treeProcessor(Tree T) {  
            Mailbox m = new QueuedMailbox();  
            new TreeIterator(T, m).start();  
            while (true) {  
                Object x = m.receive();  
                if (x is end marker)  
                    break;  
                do something with x;  
            }  
        }
```

Use In GUIs

- Java runtime library uses a special thread that does nothing but wait for *events* like mouse clicks, pressed keys, mouse movement, etc.
- You can designate an object of your choice as a *listener*; which means that Java's event thread calls a method of that object whenever an event occurs.
- As a result, your program can do work while the GUI continues to respond to buttons, menus, etc.
- Another special thread does all the drawing. You don't have to be aware when this takes place; just ask that the thread wake up whenever you change something.

Highlights of a GUI Component

```
/** A widget that draws multi-colored lines indicated by mouse. */
class Lines extends JComponent implements MouseListener {
    private List<Point> lines = new ArrayList<Point>();

    Lines() { // Main thread calls this to create one
        setPreferredSize(new Dimension(400, 400));
        addMouseListener(this);
    }
    public synchronized void paintComponent(Graphics g) { // Paint thread
        g.setColor(Color.white);    g.fillRect(0, 0, 400, 400);
        int x, y;    x = y = 200;
        Color c = Color.black;
        for (Point p : lines)
            g.setColor(c); c = chooseNextColor(c);
            g.drawLine(x, y, p.x, p.y); x = p.x; y = p.y;
    }
}
public synchronized void mouseClicked(MouseEvent e) // Event thread
{ lines.add(new Point(e.getX(), e.getY())); repaint(); }
...
}
```

Interrupts

- An *interrupt* is an event that disrupts the normal flow of control of a program.
- In many systems, interrupts can be totally *asynchronous*, occurring at arbitrary points in a program, the Java developers considered this unwise; arranged that interrupts would occur only at controlled points.
- In Java programs, one thread can interrupt another to inform it that something unusual needs attention:

```
otherThread.interrupt();
```

- But otherThread does not receive the interrupt until it waits: methods `wait`, `sleep` (wait for a period of time), `join` (wait for thread to terminate), and mailbox `deposit` and `receive`.
- Interrupt causes these methods to throw `InterruptedException`, so typical use is like this:

```
try {  
    msg = inBox.receive();  
} catch (InterruptedException e) { HandleEmergency(); }
```

Remote Mailboxes (A Side Excursion)

- RMI: Remote Method Interface allows one program to refer to objects in another program.
- We use it to allow mailboxes in one program be received from or deposited into in another.
- To use this, you define an *interface* to the remote object:

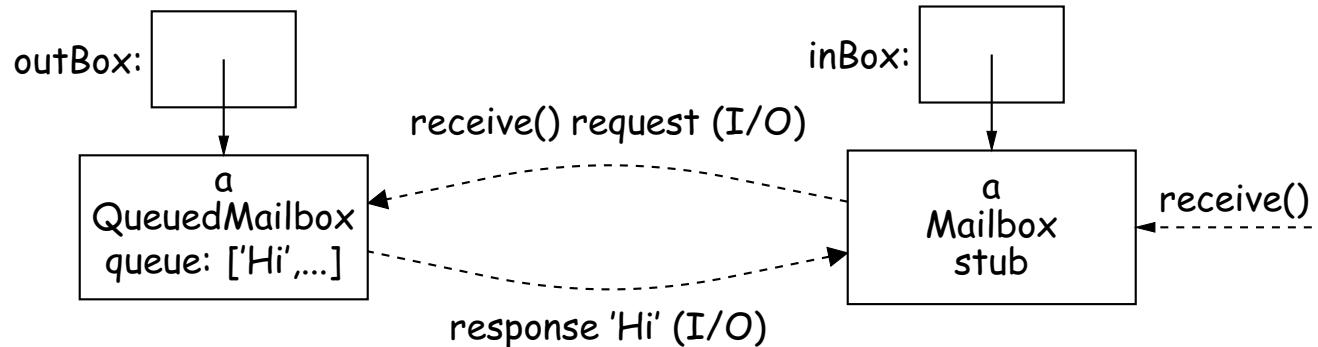
```
import java.rmi.*;  
interface Mailbox extends Remote {  
    void deposit(Object msg)  
        throws InterruptedException, RemoteException;  
    Object receive()  
        throws InterruptedException, RemoteException;  
    ...  
}
```

- On machine that actually will contain the object, you define

```
class QueuedMailbox ... implements Mailbox {  
    Same implementation as before, roughly  
}
```

Remote Objects Under the Hood

```
// On machine #1:          // On Machine #2:  
Mailbox outBox           Mailbox inBox  
= new QueuedMailbox();    = get outBox from machine #1
```



- Because `Mailbox` is an interface, hides fact that on Machine #2 doesn't actually have direct access to it.
- Requests for method calls are relayed by I/O to machine that has real object.
- Any argument or return type OK if it also implements `Remote` or can be `serialized`—turned into stream of bytes and back, as can primitive types and `String`.
- Because I/O involved, expect failures, hence every method can throw `RemoteException` (subtype of `IOException`).

Scope and Lifetime

- *Scope* of a declaration is portion of program text to which it applies (is *visible*).
 - Need not be contiguous.
 - In Java, is static: independent of data.
- *Lifetime* or *extent* of storage is portion of program execution during which it exists.
 - Always contiguous
 - Generally dynamic: depends on data
- Classes of extent:
 - *Static*: entire duration of program
 - *Local* or *automatic*: duration of call or block execution (local variable)
 - *Dynamic*: From time of allocation statement (*new*) to deallocation, if any.

Explicit vs. Automatic Freeing

- Java has no explicit means to free dynamic storage.
- However, when no expression in any thread can possibly be influenced by or change an object, it might as well not exist:

```
IntList wasteful()
{
    IntList c = new IntList(3, new IntList(4, null));
    return c.tail;
    // variable c now deallocated, so no way
    // to get to first cell of list
}
```

- At this point, Java runtime, like Scheme's, recycles the object c pointed to: *garbage collection*.

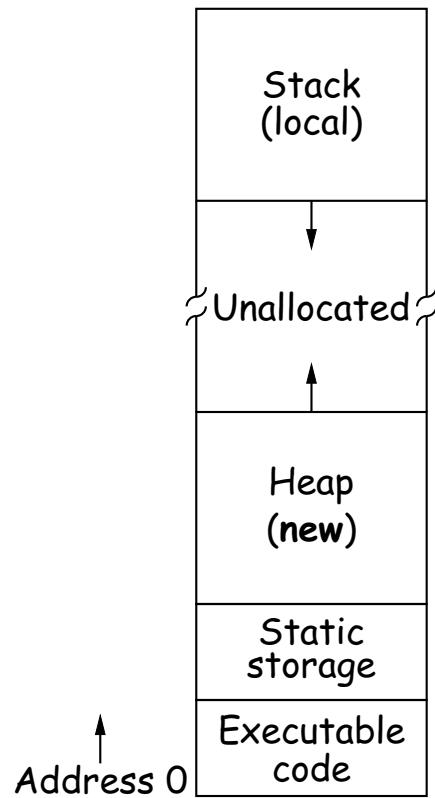
Under the Hood: Allocation

- Java pointers (references) are represented as integer addresses.
- Corresponds to machine's own practice.
- In Java, cannot convert integers \leftrightarrow pointers,
- But crucial parts of Java runtime implemented in C, or sometimes machine code, where you can.
- Crude allocator in C:

```
char store[STORAGE_SIZE]; // Allocated array
size_t remainder = STORAGE_SIZE;

/** A pointer to a block of at least N bytes of storage */
void* simpleAlloc(size_t n) { // void*: pointer to anything
    if (n > remainder) ERROR();
    remainder = (remainder - n) & ~0x7; // Make multiple of 8
    return (void*) (store + remainder);
}
```

Example of Storage Layout: Unix



- OS gives way to turn chunks of unallocated region into heap.
- Happens automatically for stack.

Explicit Deallocating

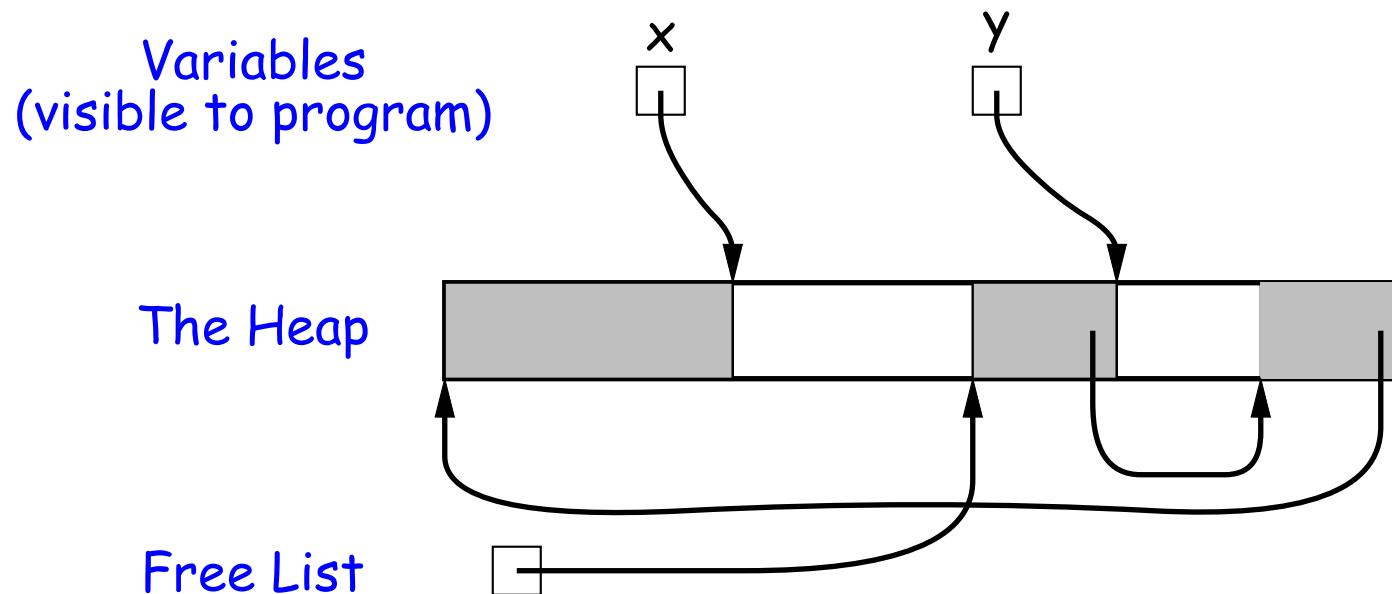
- C/C++ normally require explicit deallocation, because of
 - Lack of run-time information about what is array
 - Possibility of converting pointers to integers.
 - Lack of run-time information about *unions*:

```
union Various {  
    int Int;  
    char* Pntr;  
    double Double;  
} X; // X is either an int, char*, or double
```

- Java avoids all three problems; automatic collection possible.
- Explicit freeing can be somewhat faster, but rather error-prone:
 - Memory corruption
 - Memory leaks

Free Lists

- Explicit allocator grabs chunks of storage from OS and gives to applications.
- Or gives recycled storage, when available.
- When storage is freed, added to a *free list* data structure to be recycled.
- Used both for explicit freeing and some kinds of automatic garbage collection.

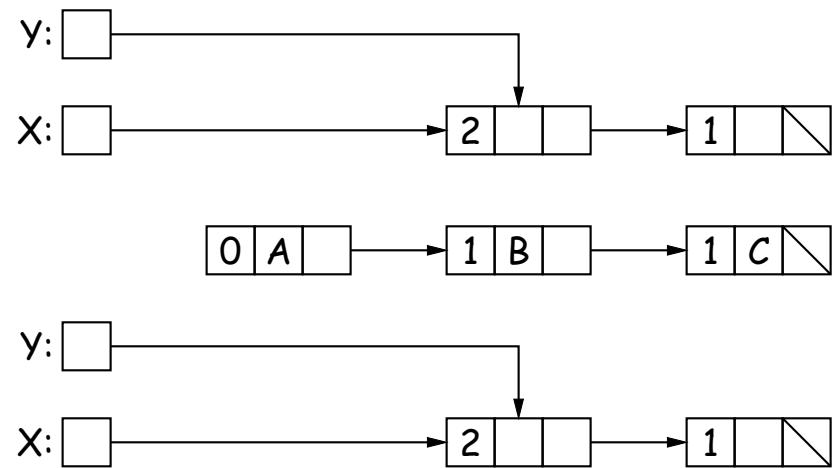
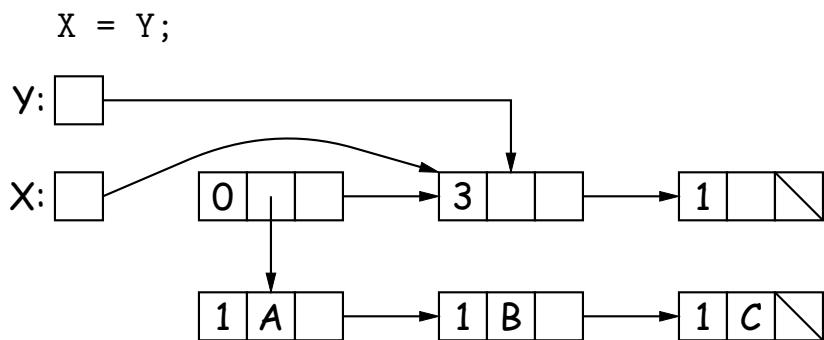
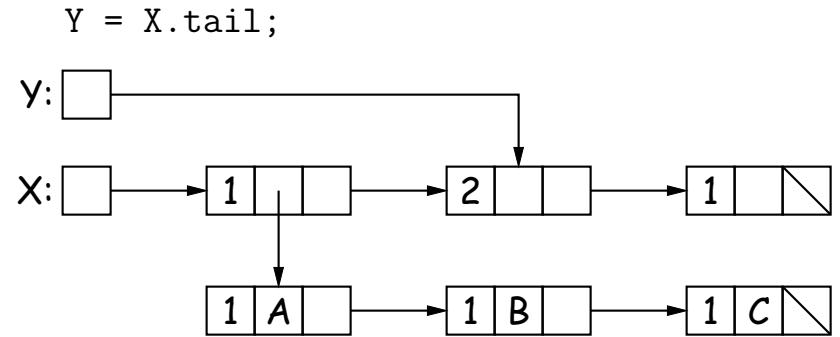
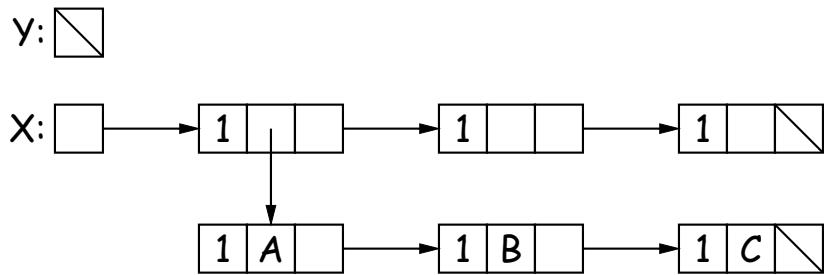


Free List Strategies

- Memory requests generally come in multiple sizes.
- Not all chunks on the free list are big enough, and one may have to search for a chunk and break it up if too big.
- Various strategies to find a chunk that fits have been used:
 - *Sequential fits:*
 - * Link blocks in LIFO or FIFO order, or sorted by address.
 - * Coalesce adjacent blocks.
 - * Search for *first fit* on list, *best fit* on list, or *next fit* on list after last-chosen chunk.
 - *Segregated fits:* separate free lists for different chunk sizes.
 - *Buddy systems:* A kind of segregated fit where some newly adjacent free blocks of one size are easily detected and combined into bigger chunks.
- Coalescing blocks reduces *fragmentation* of memory into lots of little scattered chunks.

Garbage Collection: Reference Counting

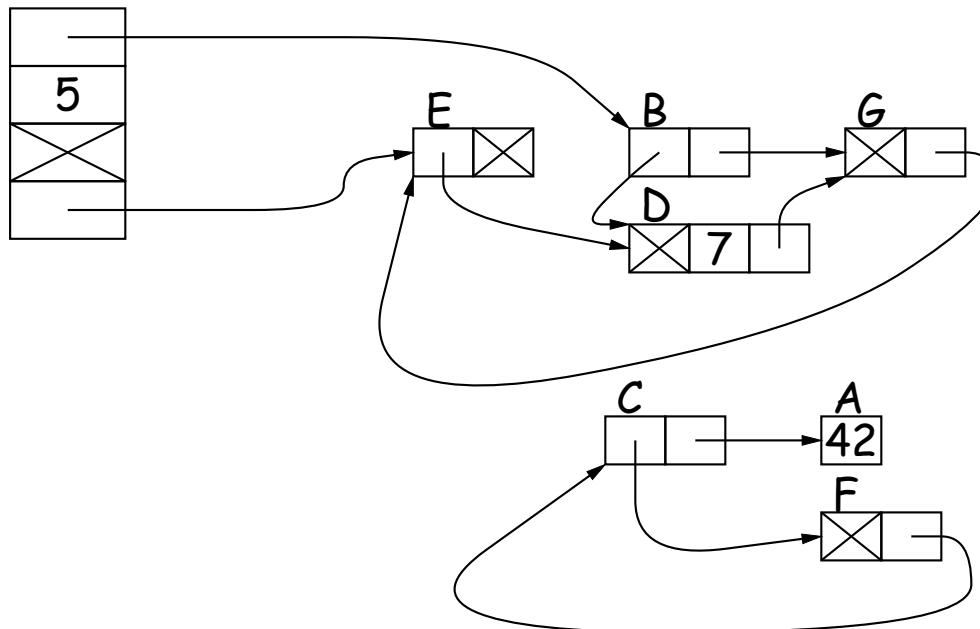
- Idea: Keep count of number of pointers to each object. Release when count goes to 0.



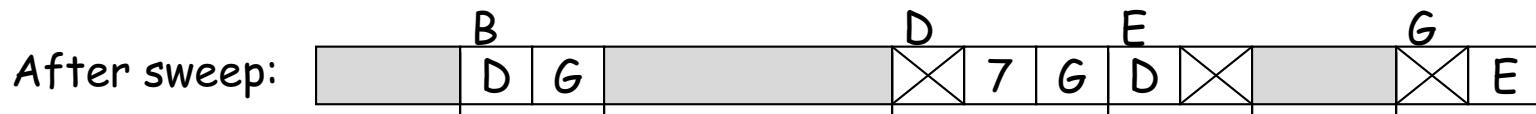
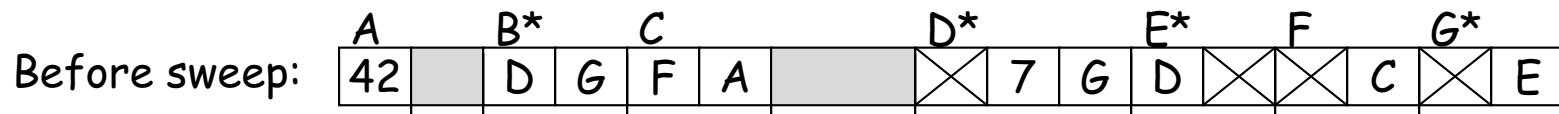
... etc., until:

Garbage Collection: Mark and Sweep

Roots (locals + statics)



1. Traverse and **mark** graph of objects.
2. **Sweep** through memory, freeing unmarked objects.



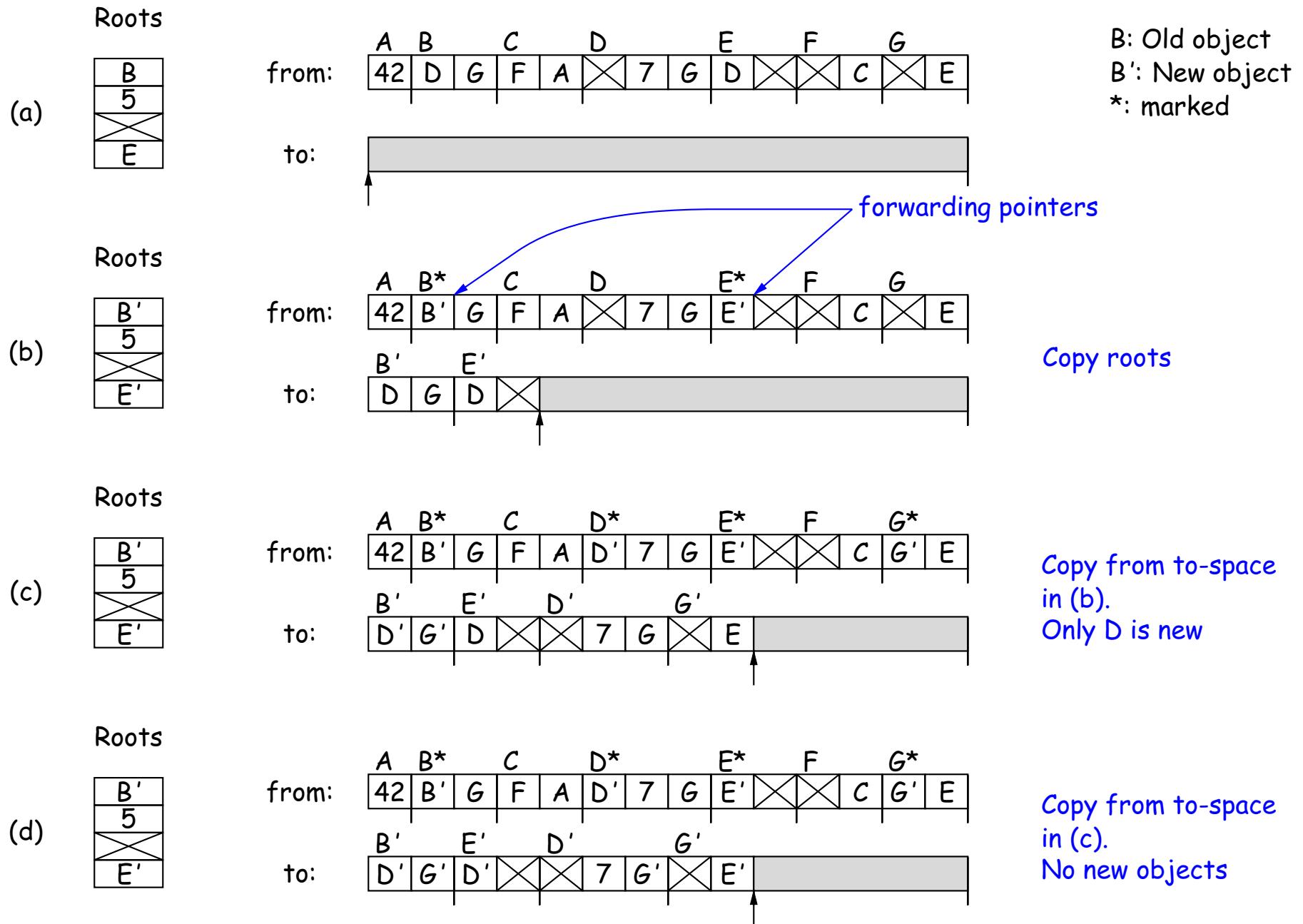
Cost of Mark-and-Sweep

- Mark-and-sweep algorithms don't move any existing objects—pointers stay the same.
- The total amount of work depends on the amount of memory swept—i.e., the total amount of active (non-garbage) storage + amount of garbage. Not necessarily a big hit: the garbage had to be active at one time, and hence there was always some “good” processing in the past for each byte of garbage scanned.

Copying Garbage Collection

- Another approach: *copying garbage collection* takes time proportional to amount of active storage:
 - Traverse the graph of active objects breadth first, *copying* them into a large contiguous area (called "to-space").
 - As you copy each object, mark it and put a *forwarding pointer* into it that points to where you copied it.
 - The next time you have to copy an already marked object, just use its forwarding pointer instead.
 - When done, the space you copied from ("from-space") becomes the next to-space; in effect, all its objects are freed in constant time.

Copying Garbage Collection Illustrated



Most Objects Die Young: Generational Collection

- Most older objects stay active, and need not be collected.
- Would be nice to avoid copying them over and over.
- *Generational garbage collection* schemes have two (or more) from spaces: one for newly created objects (*new space*) and one for "tenured" objects that have survived garbage collection (*old space*).
- A typical garbage collection collects only in new space, ignores pointers from new to old space, and moves objects to old space.
- As roots, uses usual roots plus pointers in old space that have changed (so that they might be pointing to new space).
- When old space full, collect all spaces.
- This approach leads to much smaller *pause times* in interactive systems.

There's Much More

- These are just highlights.
- Lots of work on how to implement these ideas efficiently.
- *Distributed garbage collection*: What if objects scattered over many machines?
- *Real-time collection*: where predictable pause times are important, leads to *incremental* collection, doing a little at a time.