Lecture #37

xcursions into nitty-gritty stuff: Threads, storage management.

Threads

ır programs consist of single sequences of instructions.

equence is called a *thread* (for "thread of control") in

's programs containing *multiple* threads, which (conceptually)
ntly.

gram access to threads, Java provides the type `Thread`
. Each `Thread` contains information about, and controls,

s access to data from two threads can cause chaos, so
lso constructs for controlled communication, allowing
ock objects, to *wait* to be notified of events, and to
her threads.

But Why?

a uniprocessor, only one thread at a time actually runs,
wait, but this is largely invisible. So why bother with

programs always have > 1 thread: besides the main
hers clean up garbage objects, receive signals, update
other stuff.

ms deal with asynchronous events, it is sometimes convenient
into subprograms, one for each independent, related
events.

w us to insulate one such subprogram from another.
rovide a form of modularization.

rganized like this: application is doing some computation
her thread waits for mouse clicks (like 'Stop'), another
on to updating the screen as needed.

's like search engines may be organized this way, with
er request.

se, sometimes we *do* have a real multiprocessor.

Java Mechanics

he actions "walking" and "chewing gum":

```
l implements Runnable {
id run()
 (true) ChewGum(); }

l implements Runnable {
id run()
 (true) Walk(); }
```

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

rnative (uses fact that Thread implements Runnable):

```
 extends Thread {
 run()
true) ChewGum(); }

 extends Thread {
 run()
true) Walk(); }
```

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


Avoiding Interference

read has data for another, one must wait for the other

wo threads use the same data structure, generally only
odify it at a time; other must wait.

ould happen if two threads simultaneously inserted an
nked list at the same point in the list?

d conceivably execute

```
ew ListCell(x, p.next);
```

e values of `p` and `p.next`; one insertion is lost.

for only one thread at a time to execute a method on a
ject with either of the following equivalent definitions:

```
) {
onized (this) {
ody of f
```

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


Communicating the Hard Way

ng data is tricky: the faster party must wait for the

roaches for sending data from thread to thread don't

```
changer {
lue = null;
ceive() {
 r; r = null;
(r == null)
= value; }
= null;
 r;

sit(Object data) {
(value != null) { }
= data;
```

```
DataExchanger exchanger
  = new DataExchanger();

--------------------------------

// thread1 sends to thread2 with
exchanger.deposit("Hello!");

--------------------------------

// thread2 receives from thread1 with
msg = (String) exchanger.receive();
```

hread can monopolize the machine while waiting; two
cuting `deposit` or `receive` simultaneously cause chaos.

Primitive Java Facilities

. method makes the current thread wait (not using processor)
l by `notifyAll`, unlocking the Object while it waits.

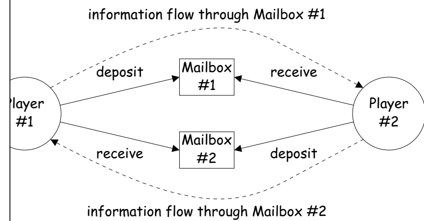
. `ucb.util.mailbox` has something like this (simplified):

```
ilbox {
osit(Object msg) throws InterruptedException;
eceive() throws InterruptedException;


Mailbox implements Mailbox {
List<Object> queue = new LinkedList<Object>();

ynchronized void deposit(Object msg) {
e.add(msg);
.notifyAll(); // Wake any waiting receivers


ynchronized Object receive() throws InterruptedException {
e (queue.isEmpty()) wait();
rn queue.remove(0);
```


Message-Passing Style

ava primitives very error-prone. CS162 goes into alternatives.

e higher-level, and allow the following program structure:



Player is a thread that looks like this:

```
ameOver()) {
Move())
tBox.deposit(computeMyMove(lastMove));

stMove = inBox.receive();
```


More Concurrency

mple can be done other ways, but mechanism is very

you want to think during opponent's move:

```
meOver()) {
Move())
tBox.deposit(computeMyMove(lastMove));

{
 thinkAheadALittle();
 lastMove = inBox.receiveIfPossible();
hile (lastMove == null);
```

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

```
chronized Object receiveIfPossible()
rows InterruptedException {
eue.isEmpty())
turn null;
 queue.remove(0);
```


Coroutines

is a kind of synchronous thread that explicitly hands
o other coroutines so that only one executes at a time,
generators. Can get similar effect with threads and

cursive inorder tree iterator:

```
or extends Thread {
ailbox r;
(Tree T, Mailbox r) {
t = T; this.dest = r;

run() {
(root);
t(End marker);

e(Tree t) {
 null) return;
(t.left);
t(t.label);
(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

e library uses a special thread that does nothing but
nts like mouse clicks, pressed keys, mouse movement,

ignate an object of your choice as a *listener;* which
Java's event thread calls a method of that object whenever
urs.

your program can do work while the UI continues to
uttons, menus, etc.

cial thread does all the drawing. You don't have to
en this takes place; just ask that the thread wake up
u change something.

Highlights of a GUI Component

```
hat draws multi-colored lines indicated by mouse. */
tends JComponent implements MouseListener {
<Point> lines = new ArrayList<Point>();

/ Main thread calls this to create one
edSize(new Dimension(400, 400));
stener(this);

ronized void paintComponent(Graphics g) { // Paint thread
(Color.white);   g.fillRect(0, 0, 400, 400);
  x = y = 200;
Color.black;
 p : lines)
or(c); c = chooseNextColor(c);
ne(x, y, p.x, p.y); x = p.x; y = p.y;


ronized void mouseClicked(MouseEvent e) // Event thread
d(new Point(e.getX(), e.getY())); repaint(); }
```

Interrupts

is an event that disrupts the normal flow of control of

ems, interrupts can be totally *asynchronous,* occurring
points in a program. The Java developers considered
nd arranged that interrupts would occur only at controlled

grams, one thread can interrupt another to inform it
ing unusual needs attention:

```
d.interrupt();
```

read does not receive the interrupt until it waits: methods
(wait for a period of time), join (wait for thread to
nd library methods like mailbox deposit and receive.

causes these methods to throw InterruptedException,
e is like this:

```
= inBox.receive();
nterruptedException e) { HandleEmergency(); }
```


mote Mailboxes (A Side Excursion)

e Method Interface allows one program to refer to
other program.

allow mailboxes in one program to be received from or
into in another.

you define an *interface* to the remote object:

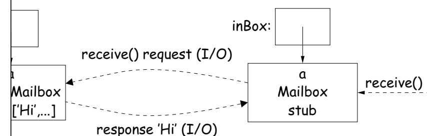
```
a.rmi.*;
Mailbox extends Remote {
eposit(Object msg)
rows InterruptedException, RemoteException;
receive()
rows InterruptedException, RemoteException;
```

ine that actually will contain the object, you define

```
edMailbox ... implements Mailbox {
mplementation as before, roughly
```


Remote Objects Under the Hood

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



lbox is an interface type, you don't see whether you are
mailbox or at a (remote) stub that stands in for it.

method calls are relayed by I/O to the machine that
ct.

nt or return type OK if it also implements Remote or
lized—turned into a stream of bytes and back, as can
es and String.

is involved, expect failures, hence every method can
eException (subtype of IOException).

New Topic: Storage Management

Scope and Lifetime

f a declaration is portion of program text to which it
sible).

be a contiguous region.

s in Python, it is static: independent of data.

or *extent* of storage is the portion of program execution
it exists.

ntiguous.

dynamic: depends on data

xtent:

ntire duration of program

utomatic: duration of the execution of a call or block—as
ariables or parameters.

From time of explicit allocation (**new**) to deallocation,

Explicit vs. Automatic Freeing

explicit means to free dynamic storage.

en no expression in any thread can possibly be influenced
an object, it might as well not exist:

```
steful()

 c = new IntList(3, new IntList(4, null));
 c.tail;
iable c now deallocated, so no way
get to first cell of list
```

t, Java's runtime, like Scheme's, "recycles" the object
d to: *garbage collection*.

Under the Hood: Allocation

s (references) are represented as integer addresses.

 to machine's own practice.

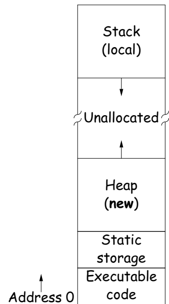
not convert integers ↔ pointers,

arts of Java's runtime are implemented in C, or sometimes
e, where you can conflate integers and pointers.

a crude allocator in C:

```
[STORAGE_SIZE];  // Allocated array
ainder = STORAGE_SIZE;

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


xample of Storage Layout: Unix



a way to turn chunks of unallocated region into heap.

omatically for stack.

Explicit Deallocating

lly require explicit deallocation, because of

un-time information about types and array sizes;

/ of converting pointers to integers;

un-time information about *unions:*

```
Various {
t Int;
ar* Pntr;
uble Double;
// X is either an int, char*, or double
```

all three problems; automatic collection possible.

ing can be somewhat faster, but rather error-prone:

orruption (freeing twice, freeing something that isn't
valid pointer.)

eaks (failing to ever release soemthing.)

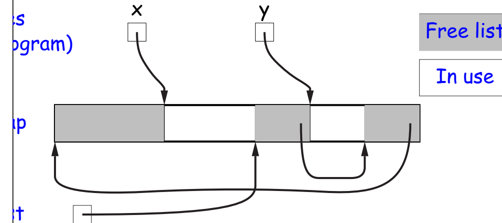
Free Lists

ator grabs chunks of storage from OS to give to applications.

ycled storage, when available.

e is freed, it is added to a *free list* data structure to

or explicit freeing and some kinds of automatic storage



Free List Strategies

uests generally come in multiple sizes.

ks on the free list are big enough, and one may have to
 chunk and break it up if too big.

tegies to find a chunk that fits have been used:

l fits:

ocks in LIFO or FIFO order, or sorted by address.

e adjacent blocks.

 for *first fit* on list, *best fit* on list, or *next fit* on list
ast-chosen chunk.

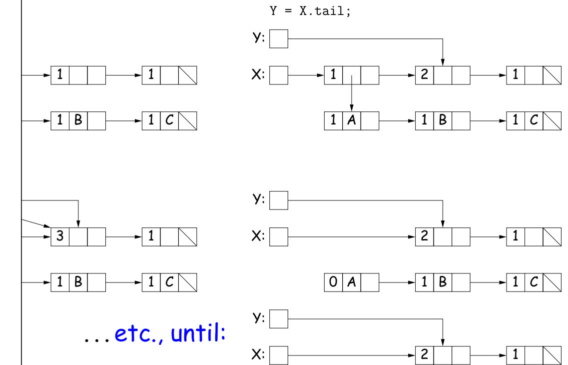
ed fits: separate free lists for different chunk sizes.

tems: A kind of segregated fit where some newly adjacent
ks of one size are easily detected and combined into
unks.

locks reduces *fragmentation* of memory into lots of
red chunks.

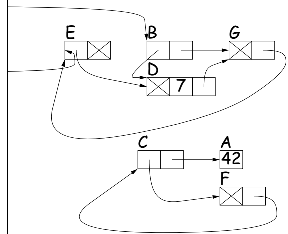
: Garbage Collection: Reference Counting

count of number of pointers to each object. Release
oes to 0.

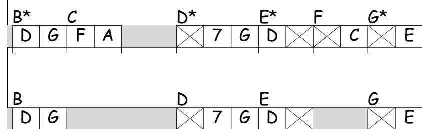


rbage Collection: Mark and Sweep

s)



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



Cost of Mark-and-Sweep

veep algorithms don't move any exisiting objects—pointers
e.

ount of work depends on the amount of memory swept—i.e.,
unt of active (non-garbage) storage + amount of garbage.
rily a big hit: the garbage had to be active at one time,
here was always some "good" processing in the past for
garbage scanned.

Copying Garbage Collection

roach: *copying garbage collection* takes time proportional
active storage:

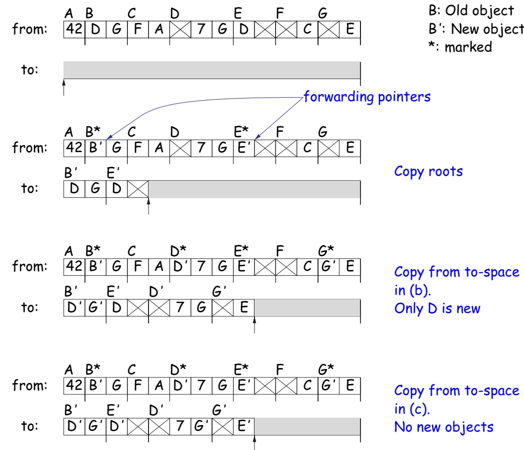
the graph of active objects breadth first, *copying* them
e contiguous area (called "to-space").

py each object, mark it and put a *forwarding pointer*
t points to where you copied it.

time you have to copy an already marked object, just
rwarding pointer instead.

e, the space you copied from ("from-space") becomes
o-space; in effect, all its objects are freed in constant

ying Garbage Collection Illustrated



jects Die Young: Generational Collection

bjects stay active, and need not be collected.

e to avoid copying them over and over.

garbage collection schemes have two (or more) from
for newly created objects (*new space*) and one for
jects that have survived garbage collection (*old space*).

bage collection collects only in new space, ignores pointers
old space, and moves objects to old space.

s usual roots plus pointers in old space that have changed
might be pointing to new space).

ace full, collect all spaces.

h leads to much smaller *pause times* in interactive systems.

There's Much More

st highlights.

on how to implement these ideas efficiently.

garbage collection: What if objects scattered over many

llection: where predictable pause times are important,
emental collection, doing a little at a time.