

# G22.2110 Programming Languages

7/26/2007  
Concurrency

## Outline

- Concurrency
- Garbage collection
- Wrapping up
- Course evaluations

## Concurrency

- Also known as multi-threading
- Thread = language level active entity
  - Own program counter (PC) and stack
  - Can run at the same time as (concurrently with) other threads
  - May share memory (globals, heap) with other threads
- Ada “task” is roughly same as Java “thread”
  - Scott book uses “task” for different concept

## Example

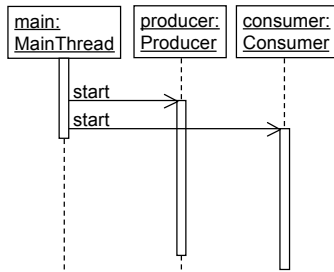
<http://www.bowdoin.edu/~allen/pl/parallel/BoundedBuffer.java>

```
class Producer extends Thread {
    private Buffer buffer;
    public Producer(Buffer b) {
        buffer = b;
    }
    public void run() {
        for (int i = 0; i < 20; i++) {
            Buffer.put(new Date());
            Thread.sleep(1000);
        }
    }
}

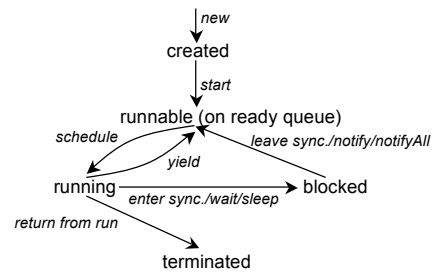
class Consumer extends Thread {
    private Buffer buffer;
    public Consumer(Buffer b) {
        buffer = b;
    }
    public void run() {
        for (int i = 0; i < 20; i++) {
            System.out.println(buffer.get());
            Thread.sleep(3000);
        }
    }
}

public static void main(String[] arg) {
    Buffer buffer = new Buffer(5);
    Producer producer = new Producer(buffer);
    producer.start();
    Consumer consumer = new Consumer(buffer);
    consumer.start();
}
```

## Sequence Diagram



## Thread States



## Buffer, Version 1 (Buggy!)

```

1. class Buffer {
2.     private Date[] buf;
3.     private int in = -1, out = -1, count = 0;
4.     public Buffer(int s) { buf = new Date[s]; }
5.     public void put(Date d) {
6.         while (count >= buf.size) { /* do nothing */ }
7.         count++;
8.         buf[++in % buf.size] = d;
9.         //
10.    }
11.    public Date get() {
12.        while (count == 0) { /* do nothing */ }
13.        count--;
14.        Date d = buf[++out % buf.size];
15.        //
16.        return d;
17.    }
18. }

```

Martin Hirzel G22.2110 NYU 4/17/2007 7

## Race Condition

buf	in	out	count	code
0 1 2 3 4			0	
////	-1	-1	0	p: while(count >= buf.size) { }
				p: count++
	1			p: buf[++in % buf.size] = d
d	0	-1		c: while(count == 0) { }
				c: count--
	0			c: d = buf[++out % buf.size]
	0			c: System.out.println(d)
				p: while(count >= buf.size) { }
				p: count++
	1			c: while(count == 0) { }
				c: count--
	0			c: d = buf[++out % buf.size]
	1			c: System.out.println(d)

Prints "null"

Martin Hirzel G22.2110 NYU 4/17/2007 8

## Race Condition

- Multiple concurrent non-atomic accesses to shared data, at least one is a write
  - Threads are "racing" against each other
- Difficult to debug:
  - Silent (may crash much later, or just produce wrong answer)
  - Non-deterministic (hard to reproduce, hard to fix with confidence)
- Critical section = code that temporarily violates invariant on shared data

Martin Hirzel G22.2110 NYU 4/17/2007 9

## Synchronization in Java

- Lock associated with object
  - In case of synchronized method: "this", i.e., the shared buffer
- When thread attempts to enter:
  - If another thread active: block
- When thread leaves:
  - If another thread waiting, wake it up
- Achieves mutual exclusion

Martin Hirzel G22.2110 NYU 4/17/2007 10

## Buffer, Version 2 (Still Buggy!)

```

1. class Buffer {
2.     private Date[] buf;
3.     private int in = -1, out = -1, count = 0;
4.     public Buffer(int s) { buf = new Date[s]; }
5.     public synchronized void put(Date d) {
6.         while (count >= buf.size) { /* do nothing */ }
7.         count++;
8.         buf[++in % buf.size] = d;
9.         //
10.    }
11.    public synchronized Date get() {
12.        while (count == 0) { /* do nothing */ }
13.        count--;
14.        Date d = buf[++out % buf.size];
15.        //
16.        return d;
17.    }
18. }

```

Martin Hirzel G22.2110 NYU 4/17/2007 11

## Deadlock

- Producer loops until there is free slot; but since producer holds lock, consumer can not create free slot
- Coffman conditions:
  - Mutual exclusion
  - Hold-and-wait
  - No preemption
  - Circular wait
- To fix example: wait, but don't hold
  - Condition notification

Martin Hirzel G22.2110 NYU 4/17/2007 12

## Buffer, Version 3 (Correct)

```

1. class Buffer {
2.     private Date[] buf;
3.     private int in = -1, out = -1, count = 0;
4.     public Buffer(int s) { buf = new Date[s]; }
5.     public synchronized void put(Date d) {
6.         while (count >= buf.size) wait();
7.         count++;
8.         buf[++in % buf.size] = d;
9.         notify();
10.    }
11.    public synchronized Date get() {
12.        while (count == 0) wait();
13.        count--;
14.        Date d = buf[++out % buf.size];
15.        notify();
16.        return d;
17.    }
18. }

```

Martin Hirzel G22.2110 NYU 4/17/2007 13

## Synchronization in Ada

- Protected types and objects
- Three kinds of methods:
  - Function (can only read fields)
  - Procedure (can read and write fields)
  - Entry (procedure with barrier condition)
- Protected by reader-writer lock
  - Multiple readers can execute concurrently
  - Writer excludes readers and other writers

Martin Hirzel G22.2110 NYU 4/17/2007 14

## Protected Type Example (Wheeler's Lovelace tutorial Section 13.4)

```

protected type Resource is
entry Seize; -- Acquire this resource exclusively.
procedure Release; -- Release the resource.
private
    Busy : Boolean := False;
end Resource;

protected body Resource is
entry Seize when not Busy is
begin
    Busy := True;
end Seize;
procedure Release is
begin
    Busy := False;
end Release;
end Resource;

```

Martin Hirzel G22.2110 NYU 4/17/2007 15

## How to Avoid Concurrency Bugs

- Don't write concurrent code
- If you have to write concurrent code: don't use shared memory
- If you have to use shared memory:
  - Share immutable data
  - Reuse libraries of concurrent containers
- If you can't reuse libraries:
  - Read a book
  - Use annotations

Martin Hirzel G22.2110 NYU 4/17/2007 16

## Shared Memory Concurrency

Java	Ada
Subclass of Thread	Task type
Instance of a subclass of Thread	Task, or instance of task type
Class where all methods are synchronized	Protected type
while (!c) wait();	Entry guard, or wait(c);
notify();	Making guard true, or signal(c);

Martin Hirzel G22.2110 NYU 4/17/2007 17

## Message Passing

- Concurrent programming without shared memory (e.g., distributed)
- Prevents data races, can still have deadlocks
- In C or Fortran: MPI
- In Java:
  - Libraries (sockets)
  - Frameworks (J2EE)
- In Ada: part of language
  - Task has "entry" subroutines
  - Call blocked until task does "accept"

Martin Hirzel G22.2110 NYU 4/17/2007 18

## Bounded Buffer with Message Passing (Scott Figure 12.21)

```

task body Buffer is
  Size: constant Integer := 10;
  subtype Index is Integer range 1..Size;
  Buf : array (Index) of Bdata;
  Next_Empty, Next_Full : Index := 1;
  Full_Slots : Integer range 0..Size := 0;
begin
  loop
  select
  when Full_Slots < Size =>
    accept Insert(D : in Bdata) do
      Buf(Next_Empty) := D;
    end;
    Next_Empty := Next_Empty mod Size + 1;
    Full_Slots := Full_Slots + 1;
  or
  when Full_Slots > 0 =>
    accept Remove(D : out Bdata) do
      D := Buf(Next_Full);
    end;
    Next_Full := Next_Full mod Size + 1;
    Full_Slots := Full_Slots - 1;
  end select;
  end loop;
end Buffer;

```

Martin Hirzel G22.2110 NYU 4/17/2007 19

## Outline

- Concurrency
- Garbage collection
- Wrapping up
- Course evaluations

## Heap Memory Management

- Heap allocation
  - malloc, new, cons, ::
- Heap deallocation
  - Make space for future allocation (prevent OutOfMemoryError / crash)
  - Explicit: free, delete
  - Automatic: garbage collection

## Explicit Deallocation

```

struct list* cons(int car, struct list* cdr) {
  struct list* result =
    (struct list*)malloc(sizeof(struct list));
  result->_car = car;
  result->_cdr = cdr;
  return result;
}

struct list* x = cons(2, cons(5, cons(1, NULL)));

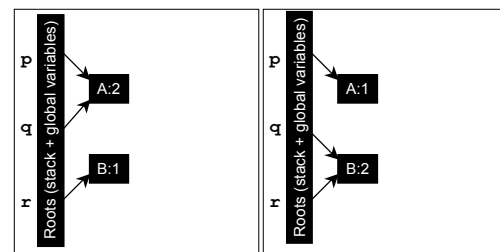
while (NULL != x) {
  struct list* y = x;
  x = x->_cdr;
  free(y);
}

```

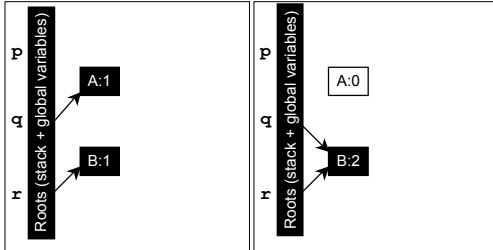
## Motivation for Garbage Collection

- Explicit deallocation is tedious ...
  - Keep track of what needs to be deallocated
  - Write the deallocation code
  - Manipulate low-level pointers
- ... and error-prone
  - Memory leak: missing deallocation
  - Dangling reference: use after deallocation
  - Hard to debug: non-local symptoms

## Reference Counting: decr(oldTgt), incr(newTgt)



## Reference counting: deallocate when refCount == 0

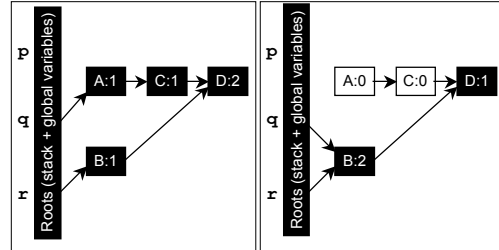


Martin Hirzel

G22.2110 NYU 4/17/2007

25

## Reference counting: recursive decrement/deallocate

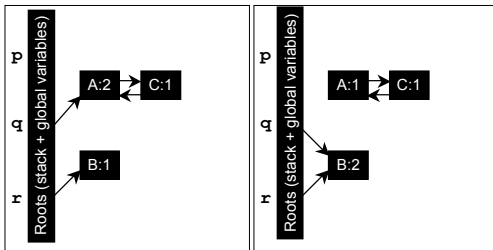


Martin Hirzel

G22.2110 NYU 4/17/2007

26

## Reference counting: does not deallocate cyclic garbage

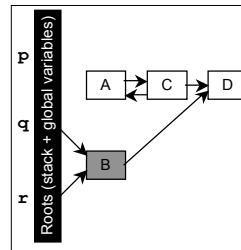


Martin Hirzel

G22.2110 NYU 4/17/2007

27

## Tracing garbage collection: reachability traversal



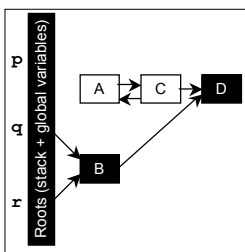
- Tricolor abstraction
  - White = not reached
  - Gray = reached but may have pointers to white
  - Black = reached and has no pointers to white
- Init: roots gray, rest white
- While any gray object X:
  - Mark successors(X) gray
  - Mark X black

Martin Hirzel

G22.2110 NYU 4/17/2007

28

## Tracing garbage collection: deallocation



- Termination
  - No more gray objects
  - All black objects survive
  - All white objects are garbage
  - Deallocate white objects

Martin Hirzel

G22.2110 NYU 4/17/2007

29

## Tracing garbage collection: Algorithms

- Mark-sweep
  - White → gray: set mark bit, put on stack
  - Gray → black: scan and mark successors
  - White → deallocated: sweep whole heap
- Copying
  - White → gray: set forwarding pointer, copy
  - Gray → black: scan and copy successors
  - White → deallocated: discard all originals

Martin Hirzel

G22.2110 NYU 4/17/2007

30

## Pragmatics

- Costs of garbage collection
  - Throughput
  - Responsiveness
  - Space
- Generational garbage collection
  - Group objects by age
  - Hypothesis: most objects die young
  - Collect young generation more frequently: most garbage collections are cheap
  - Use remembered set for old→young pointers

Martin Hirzel

G22.2110 NYU 4/17/2007

31

## Outline

- Concurrency
- Garbage collection
- Wrapping up
- Course evaluations

Martin Hirzel

G22.2110 NYU 4/17/2007

32

## Metainterpreters

- Interpreter = program that executes another program (e.g., python, sml)
- Metainterpreter = interpreter written in the language it interprets
- Also known as “metacircular interpreters”, see Scott Section 10.3.5
- Can define semantics of homo-iconic languages as fixed point (Lisp, Scheme)
- Writing a metainterpreter stretches new programming languages

Martin Hirzel

G22.2110 NYU 4/17/2007

33

## Core exam syllabus

<http://www.cs.nyu.edu/web/Academic/Graduate/exams/syllabi/core.html>

- Algorithms
- Operating systems
- Compilers
- Programming languages
  - Syntactic issues: regular expressions, context-free grammars (CFG), BNF.
  - Imperative languages: program organization, control structures.
  - Types in imperative languages: strong typing, type equivalence, unions and discriminated types in C and Ada.
  - Block structure, visibility and scoping issues, parameter passing.
  - Systems programming and weak typing: exposing machine characteristics, type coercion, pointers & arrays in C. Red: not in this class
  - Run-time organization of block-structured languages: static scoping, activation records, dynamic and static chains, displays.
  - Programming in the large: abstract data types, modules, packages and namespaces in Ada, Java, and C++.
  - Functional programming: list structures, higher order functions, lambda expressions, garbage collection, metainterpreters in Lisp and Scheme. Type inference and ML.
  - Object-Oriented programming: classes, inheritance, polymorphism, dynamic dispatching. Constructors, destructors, and multiple inheritance in C++, interfaces in Java.
  - Generic programming: parameterized units and classes in C++, Ada, and Java.
  - Concurrent programming: threads and tasks, communication, race conditions and deadlocks, protected methods and types in Ada and Java.

Martin Hirzel

G22.2110 NYU 4/17/2007

34

## How to learn a language

- I. Use peers & gurus
- II. Install tools
- III. Read tutorial
- IV. Find language and library reference
- V. Read example programs
- VI. Write example programs: I/O, types, control flow, libraries
- VII. Understand error messages
- VIII. Practice

Martin Hirzel

G22.2110 NYU 4/17/2007

35

## Topics 2<sup>nd</sup> half of semester

- Explicit memory management
- Pointer arithmetic
- Type equivalence, type compatibility
- Object-oriented programming languages
- Virtual method dispatch
- Parameter passing modes
- Exceptions
- Polymorphism (inclusion, parametric)
- Type inference in ML

Martin Hirzel

G22.2110 NYU 4/17/2007

36

# Questions?

Martin Hirzel

G22.2110 NYU 4/17/2007

37

## Outline

- Concurrency
- Garbage collection
- Wrapping up
- **Course evaluations**

Martin Hirzel

G22.2110 NYU 4/17/2007

38