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

```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();
}
```

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

Sequence Diagram

Thread States
Buffer, Version 1 (Buggy!)

```java
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.  public Date get() {
11.    while (count == 0) { /* do nothing */ }
12.    count--;
13.    Date d = buf[++out % buf.size];
14.    return d;
15.  }
16. }
```

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

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

Buffer, Version 2 (Still Buggy!)

```java
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.  public synchronized Date get() {
11.    while (count == 0) { /* do nothing */ }
12.    count--;
13.    Date d = buf[++out % buf.size];
14.    return d;
15.  }
16. }
```

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

Race Condition

- buff in  out  count  code
- 0 1 2 3 4
- // // // //  -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”
Buffer, Version 3 (Correct)

```java
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. }
```

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

Protected Type Example
(Wheeler’s Lovelace tutorial Section 13.4)

```ada
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;
```

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

Shared Memory Concurrency

<table>
<thead>
<tr>
<th>Java</th>
<th>Ada</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subclass of Thread</td>
<td>Task type</td>
</tr>
<tr>
<td>Task, or instance of task type</td>
<td></td>
</tr>
<tr>
<td>Class where all methods are synchronized</td>
<td>Protcted type</td>
</tr>
<tr>
<td>while (lc) wait();</td>
<td>Entry guard, or wait(c);</td>
</tr>
<tr>
<td>notify();</td>
<td>Making guard true, or signal(c);</td>
</tr>
</tbody>
</table>

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”
Bounded Buffer with Message Passing (Scott Figure 12.21)

```
task Buffer is
  entry Insert(D : in Bdata);
  entry Remove(D : out Bdata);
end Buffer;

main:
 MainThread
  producer:
  Consumer
  new
  buffer:
  insert
  accept
  remove
  accept
end Buffer;
```

Outline

- Concurrency
- Garbage collection
- Wrapping up
- Course evaluations

Heap Memory Management

- Heap allocation
  - malloc, new, cons, ::
- Heap deallocation
  - Make space for future allocation
  - Explicit: free, delete
  - Automatic: garbage collection

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

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);
}
```
Reference counting:
deallocate when refCount == 0

Reference counting:
recursive decrement/deallocate

Reference counting:
does not deallocate cyclic garbage

Tracing garbage collection:
reachability traversal

Tracing garbage collection:
deallocation

Tracing garbage collection:
Algorithms

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

• 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
**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

**Outline**

- Concurrency
- Garbage collection
- Wrapping up
- Course evaluations

**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

**Core exam syllabus**

http://www.cs.nyu.edu/web/Academic/Graduate/exams/syllabii/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
  - Block structure, visibility and scoping issues, parameter passing
  - Systems programming and weak typing
  - Compiling machine characteristics, type coercion, pointers & arrays in C
  - 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++
  - Dynamic dispatching, inheritance, polymorphism, dynamic dispatching
  - Abstract classes, interfaces, and multiple inheritance in C++, Java, and C++

**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

**Topics 2nd 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
Questions?

Outline

• Concurrency
• Garbage collection
• Wrapping up
• Course evaluations