Structuring the Control

• Hints on “control-in-the-small”
• Essential elements of “control-in-the-large” (parallel execution)
Control in-the-small

- From **if-then-else, while, case, for, ...**
- …to low-level constructs (GOTO and its mates) ...
- ... to more elaborate constructs; nondeterministic choice:
  - **if**
    - C1 $\rightarrow$ S1;
    - C2 $\rightarrow$ S2;
    - ...
  - **endif**
    - C1, ...Cn *not* mutually exclusive!
- will see a generalization of this for process synchronization
Concurrent computing

• A new computation model
• Set of concurrent computations (tasks) that proceed in parallel and interact only occasionally
• Each process has its own thread of control
• Parallelism may be physical or logical
Quasi-concurrency: coroutines
Parallelism and cooperation

Correct cooperation requires synchronization

- delay producer if buffer full
- delay consumer if buffer empty
- prohibit parallel modification of “buffer length”
Types of tasks

• They can share the same address space
  – threads
    • they can communicate via shared memory
• They do not share the same address space
  – processes
    • they communicate via messages
Mutual exclusion

• Some actions must be executed in mutual execution by two or more tasks; they must be atomic (logically indivisible) operations

• e.g. update buffer length
  – length = length + 1
  – length = length - 1

what if

A reads length
B reads length
A adds 1 to read value
B adds 1 to read value
A stores result back
B stores result back
Java threads: a short survey

The Thread class and its -main- methods:

– public Thread ( //thread name or empty)
– public start
– public run
– public sleep
– public interrupt (or notify)
– public boolean isAlive
– ...
The state diagrams of threads

- Born
- Start
- Ready
- Yield, interrupt, ...
- Notify
- Running
- Send I/O
- I/O completed
- Wait
- Sleeping
- Sleep
- Complete
- Dead
- Blocked
- Sleep quantum expired
Synchronized methods and the concept of monitor

- A *synchronized* method in Java guarantees access in mutual exclusion to a shared resource.
- Threads’ -possibly synchronized- methods are suspended and resumed through wait and notify methods.
- A shared object that exploits such synchronization mechanisms may implement the concept of monitor (shared, passive object).
- Monitor first introduced in concurrent Pascal (by Hoare and Brinch-Hansen).
- The risk of deadlock in concurrent programming:
  - P1 is blocked waiting to access resource A.
  - P2 owns A but is blocked waiting to access resource B.
  - B is owned by P1.
Main program for the Java producer/consumer example

```java
public class Main { /* Entry point of the program */
    public static void main (String args[])
    {
        Buffer B = new Buffer (100); // create shared buffer
        Consumer C = new Consumer (B); // create consumer
        Producer P = new Producer (B);  // create producer
        C.start(); // start consumer thread
        P.start(); // start producer thread
    }
}
```
Producer/Consumer objects in Java

class Producer extends Thread {
    private Buffer buff; //construct a producer object
    Producer (Buffer b) {
        buff = b;
    }
    public void run() { //produce things and put them in buffer
        buff.put (...); //buff is a shared buffer object
    }
}

Consumer is similar … its run method contains
…x = buf.get();

This approach is non longer recommended…
Alternative to extending Thread class: implement Runnable interface + code for run method (1)

```java
class Producer implements Runnable {
    private Buffer buff;
    Producer (Buffer b) {
        buff = b;
    }
    public void run() {
        // produce things and put them in buffer
        buff.put(...);
        ...
    }
}

... similarly for Consumer class ...
```

- Useful when need to make “runnable” an already existing class
  - Create a new class that
    - inherits from the one that already exists
    - implements runnable
- Nice example of the using multiple inheritance
Alternative to extending Thread class: implement Runnable interface + code for run method  (2)

Steps for running separate producer and consumer tasks in separate threads

```java
public class Main { /* Entry point of the program */
    public static void main (String args[])
    {
        Buffer B = new Buffer (100); // create shared buffer
        Runnable c = new Consumer (B); // create consumer
        Runnable p = new Producer (B); // create producer
        Thread pt = Thread(p); // construct a Thread object from Producer
        Thread ct = Thread(c); // construct a Thread object from Consumer
        pt.start(); // start producer thread
        ct.start(); // start consumer thread
    }
}
```
Synchronized buffer in Java

public class Buffer {
    private int n;  // size of buffer
    private int [] contents;  // contents of buffer
    private int in, out = 0;  // indexes of where to read from/write to
    private int total = 0;  // number of items in the buffer
    Buffer (int size) {
        n = size;
        contents = new int [n];
    }
}
Synchronized buffer in Java -2

```java
public synchronized void put (int item) {
    while (!(total < n))
        try { wait(); } // wait until there is space
        // waiting will terminate with an InterruptedException
        catch (InterruptedException e) { }
    contents [in] = item;
    System.out.println("Buffer: write at " + in + " item " + item);
    if (++in == n) in = 0;
    total++;
    notify(); // wake up any blocked threads
    // notifyAll() wakes all blocked threads
}
```
public synchronized int get () {
    int temp;
    while (!(total > 0))
        try { wait(); } // wait till there is something
        // waiting will terminate with an InterruptedException
    catch (InterruptedException e) { }
    temp = contents[out];
    System.out.println("Buffer: read from ", out + " item ", + temp);
    if (++out == n) out = 0;
    total--;
    notify(); // wake up any blocked threads
    // notifyAll() wakes all blocked threads
    return temp;
}
Task synchronization in Ada

- Guardians and rendez-vous
- The Ada style of designing concurrent systems
- In Ada a shared object is active (whereas a monitor is passive)
  - it is managed by a guardian process which can accept rendez-vous requests from tasks willing to access the object
Tasks using the resource

Guardian

Resource

Request

Service

Guardian.GET(c);
...
Guardian.PUT(c);

Request

Service

Request

Service
A guardian task

\begin{verbatim}
loop
  select
    when NOT_FULL
      accept PUT (C: in CHAR) do
      \textit{This is the body of PUT; the client calls it as if it were a normal procedure}
    end ;
  or
    when NOT_EMPTY
      accept GET (C: out CHAR) do
      \textit{This is the body of GET; the client calls it as if it were a normal procedure}
    end ;
  end select ;
end loop ;
\end{verbatim}

PUT and GET are the \textit{entries} of the guardian task

\textbf{note nondeterministic acceptance of rendez-vous requests (select+when)}
Real-time software (hints)

- Case where some processes exist in the environment

- E.g., put operation in a shared buffer invoked by a plant sending data to a controller
  - plant cannot be suspended if buffer full!
    - design must ensure that produces never finds the buffer full
      - this constrains the speed of the consumer process in the controller
Distributed software (hints)

• Issues to consider
  – module-machine binding
  – intermodule communication
    • e.g., remote procedure call or message passing
  – access to shared objects
    • may require replication for efficiency reasons
Client-server architecture

- The most popular distributed architecture
- Server modules provide services to client modules
- Clients and servers may reside on different machines
Issues

• Binding modules to machines
  – static vs. dynamic (*migration*)

• Inter-module communication
  – e.g., RPC
  – IDL to define interface of remote procedures

• Replication and distribution
Middleware

- Layer residing between the network operating system and the application
- Helps building network applications
- Provides useful services
  - Name services, to find processes or resources on the network
  - Communication services, such as message passing or RPC