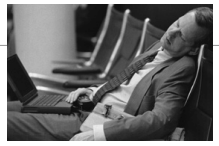
Review for Final

Prof. Stephen A. Edwards

Concurrency in Java

The Sleep Method



```
public void run() {
  for(;;) {
    try {
      sleep(1000); // Pause for 1 second
    } catch (InterruptedException e) {
       return;       // caused by thread.interrupt()
    }
    System.out.println("Tick");
  }
}
```


Per-Object Locks

- **Each Java object has a lock that may be owned by at least one thread**
- **A thread waits if it attempts to obtain an already-obtained lock**
- **The lock is a counter: one thread may lock an object more than once**

The Synchronized Statement

- **A synchronized statement gets an object's lock before running its body**

```
Counter mycount = new Counter;
synchronized(mycount) {
  mycount.count();
}
```

"get the lock for mycount before calling count()"

- **Releases the lock when the body terminates**
- **Choice of object to lock is by convention**

Synchronized Methods

```
class AtomicCounter {
  private int _count;

  public synchronized void count() {
    _count++;
  }
}
```

"get the lock for the AtomicCounter object before running this method"

This implementation guarantees at most one thread can increment the counter at any time

wait() and notify()

- Each object has a set of threads that are waiting for its lock (its wait set)

```
synchronized (obj) {      // Acquire lock on obj
  obj.wait();             // suspend
                          // add thread to obj's wait set
                          // relinquish locks on obj
In other thread:
obj.notify();             // enable some waiting thread
```


wait() and notify()

- Confusing enough?
- notify() nodeterministically chooses one thread to reawaken (may be many waiting on same object)
  - What happens when there's more than one?
- notifyAll() enables all waiting threads
  - Much safer?

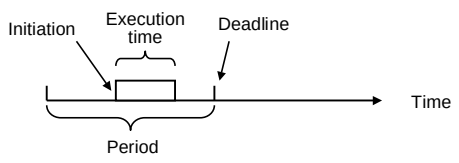
Real-Time Operating Systems

Priority-based Scheduling

- Typical RTOS based on fixed-priority preemptive scheduler
- Assign each process a priority
- At any time, scheduler runs highest priority process ready to run
- Process runs to completion unless preempted

Typical RTOS Task Model

- Each task a triplet: (execution time, period, deadline)
- Usually, deadline = period
- Can be initiated any time during the period



Key RMS Result

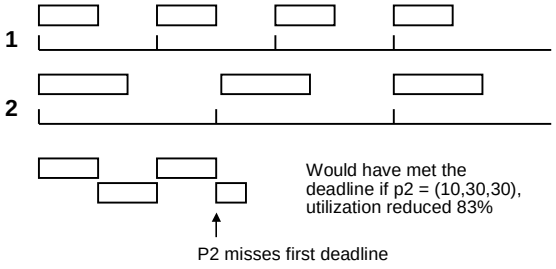
- Rate-monotonic scheduling is optimal:

  If there is fixed-priority schedule that meets all deadlines, then RMS will produce a feasible schedule

- Task sets do not always have a schedule
- Simple example: P1 = (10, 20, 20) P2 = (5, 9, 9)
  - Requires more than 100% processor utilization

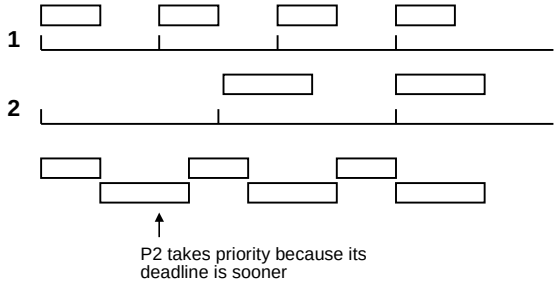
RMS Missing a Deadline

- **p1 = (10,20,20) p2 = (15,30,30) utilization is 100%**



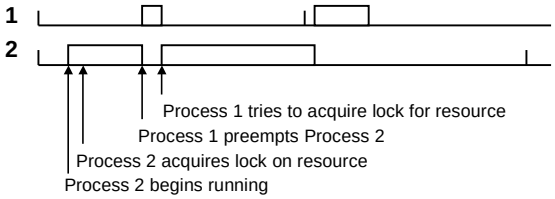
EDF Meeting a Deadline

- **p1 = (10,20,20) p2 = (15,30,30) utilization is 100%**



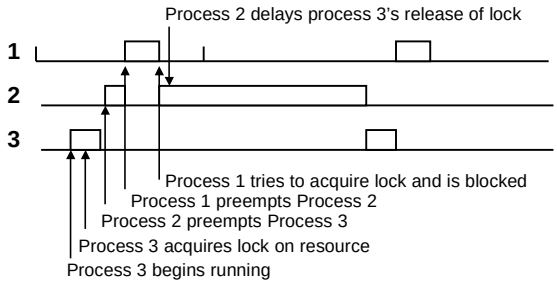
Priority Inversion

- **RMS and EDF assume no process interaction**
- **Often a gross oversimplification**
- **Consider the following scenario:**



Nastier Example

- **Higher priority process blocked indefinitely**



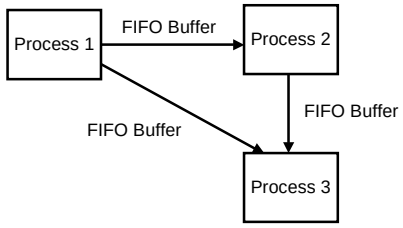
Priority Inheritance

- **Solution to priority inversion**
- **Temporarily increase process's priority when it acquires a lock**
- **Level to increase: highest priority of any process that might want to acquire same lock**
  - **I.e., high enough to prevent it from being preempted**
- **Danger: Low-priority process acquires lock, gets high priority and hogs the processor**
  - **So much for RMS**

Dataflow Languages

Dataflow Language Model

- **Processes communicating through FIFO buffers**



A Kahn Process

- **From Kahn's original 1974 paper**

```
process f(in int u, in int v, out int w)
{
  int i; bool b = true;
  for (;;) {
    i = b ? wait(u) : wait(w);
    printf("%i\n", i);
    send(i, w);
    b = !b;
  }
}
```

Process interface includes FIFOs

wait() returns the next token in an input FIFO, blocking if it's empty

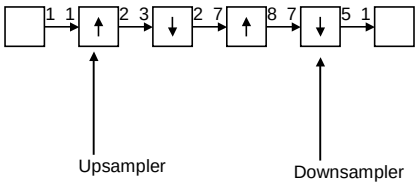
send() writes a data value on an output FIFO

Tom Parks' Algorithm

- **Schedules a Kahn Process Network in bounded memory if it is possible**
- **Start with bounded buffers**
- **Use any scheduling technique that avoids buffer overflow**
- **If system deadlocks because of buffer overflow, increase size of smallest buffer and continue**

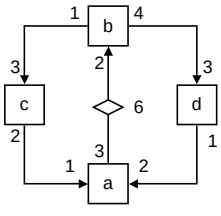
Multi-rate SDF System

- **DAT-to-CD rate converter**
- **Converts a 44.1 kHz sampling rate to 48 kHz**



Calculating Rates

- **Each arc imposes a constraint**



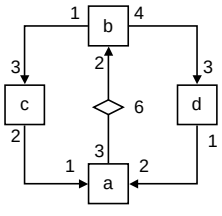
$3a - 2b = 0$
$4b - 3d = 0$
$b - 3c = 0$
$2c - a = 0$
$d - 2a = 0$

Solution:

$a = 2c$
$b = 3c$
$d = 4c$

Scheduling Example

- **Theorem guarantees any valid simulation will produce a schedule**



a=2 b=3 c=1 d=4

Possible schedules:

BBBCDDDDAA

BDBDBCADDA

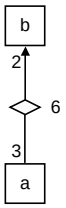
BBDDBDDCAA

… many more

BC … is not valid

Finding Single-Appearance Schedules

- Always exist for acyclic graphs
  - Blocks appear in topological order
- For SCCs, look at number of tokens that pass through arc in each period (follows from balance equations)
- If there is at least that much delay, the arc does not impose ordering constraints
- Idea: no possibility of underflow



a=2 b=3

6 tokens cross the arc

delay of 6 is enough

Summary of Dataflow

- Processes communicating exclusively through FIFOs
- Kahn process networks
  - Blocking read, nonblocking write
  - Deterministic
  - Hard to schedule
  - Parks' algorithm requires deadlock detection, dynamic buffer-size adjustment

Summary of Dataflow

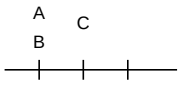
- Synchronous Dataflow (SDF)
- Firing rules:
  - Fixed token consumption/production
- Can be scheduled statically
  - Solve balance equations to establish rates
  - Any correct simulation will produce a schedule if one exists
- Looped schedules
  - For code generation: implies loops in generated code
  - Recursive SCC Decomposition
- CSDF: breaks firing rules into smaller pieces
  - Scheduling problem largely the same

Esterel

Basic Esterel Statements

- Thus

```
emit A;
present A then emit B end;
pause;
emit C
```



- Makes A & B present the first instant, C present the second

Signal Coherence Rules

- Each signal is only present or absent in a cycle, never both
- All writers run before any readers do
- Thus

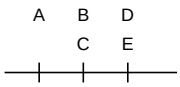
```
present A else
    emit A
end
```

is an erroneous program

The || Operator

- Groups of statements separated by || run concurrently and terminate when all groups have terminated

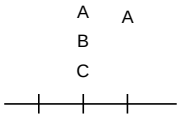
```
[
  emit A; pause; emit B;
||
  pause; emit C; pause; emit D
];
emit E
```



Bidirectional Communication

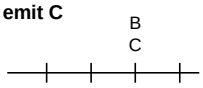
- Processes can communicate back and forth in the same cycle

```
[
  pause; emit A; present B then emit C end;
  pause; emit A
||
  pause; present A then emit B end
]
```



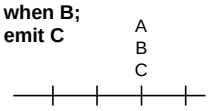
Strong vs. Weak Abort

```
abort
  pause;
  pause;
  emit A;
  pause
when B;
emit C
```



Strong abort: emit A not allowed to run

```
weak abort
  pause;
  pause;
  emit A;
  pause
when B;
emit C
```



Weak abort: emit A allowed to run, body terminated afterwards

Causality

- Can be very complicated because of instantaneous communication
- For example: this is also erroneous

```
abort
  emit B
when A
||
 [
  present B then emit A end;
  pause
 ]
```

Emission of B indirectly causes emission of A

What To Understand About Esterel

- Synchronous model of time
  - Time divided into sequence of discrete instants
  - Instructions either run and terminate in the same instant or explicitly in later instants
- Idea of signals and broadcast
  - "Variables" that take exactly one value each instant and don't persist
  - Coherence rule: all writers run before any readers
- Causality Issues
  - Contradictory programs
  - How Esterel decides whether a program is correct

What To Understand About Esterel

- Compilation techniques
  - Automata
    - Fast code
    - Doesn't scale
  - Netlists
    - Scales well
    - Slow code
    - Good for causality
  - Control-flow
    - Scales well
    - Fast code
    - Bad at causality
- Compilers, documentation, etc. available from www.esterel.org

Verilog

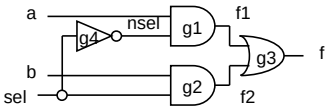
Multiplexer Built From Primitives

```
module mux(f, a, b, sel);
output f;
input a, b, sel;

and   g1(f1, a, nsel),
      g2(f2, b, sel);
or    g3(f, f1, f2);
not   g4(nsel, sel);

endmodule
```

Verilog programs built from modules

Each module has an interface

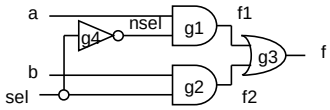
Module may contain structure: instances of primitives and other modules



Multiplexer Built From Primitives

```
module mux(f, a, b, sel);
output f;
input a, b, sel;

and   g1(f1, a, nsel),
      g2(f2, b, sel);
or    g3(f, f1, f2);
not   g4(nsel, sel);

endmodule
```

Identifiers not explicitly defined default to wires

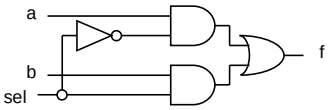


Multiplexer Built With Always

```
module mux(f, a, b, sel);
output f;
input a, b, sel;
reg f;

always @(a or b or sel)
  if (sel) f = a;
  else f = b;

endmodule
```

Modules may contain one or more *always* blocks

Sensitivity list contains signals whose change triggers the execution of the block

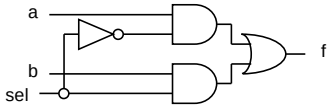


Multiplexer Built With Always

```
module mux(f, a, b, sel);
output f;
input a, b, sel;
reg f;

always @(a or b or sel)
  if (sel) f = a;
  else f = b;

endmodule
```

A *reg* behaves like memory: holds its value until imperatively assigned otherwise

Body of an *always* block contains traditional imperative code

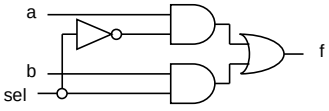


Mux with Continuous Assignment

```
module mux(f, a, b, sel);
output f;
input a, b, sel;

assign f = sel ? a : b;

endmodule
```

LHS is always set to the value on the RHS

Any change on the right causes reevaluation



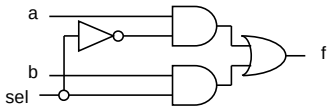
Mux with User-Defined Primitive

```
primitive mux(f, a, b, sel);
output f;
input a, b, sel;

table
  1?0 : 1;
  0?0 : 0;
  ?11 : 1;
  ?01 : 0;
  11? : 1;
  00? : 0;
endtable
endprimitive
```

Behavior defined using a truth table that includes "don't cares"

This is a less pessimistic than others: when a & b match, sel is ignored

(others produce X)

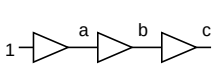


Nonblocking Looks Like Latches

- **RHS of nonblocking taken from latches**
- **RHS of blocking taken from wires**

```
a = 1;
b = a;
c = b;
```

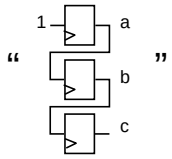
"



"

```
a <= 1;
b <= a;
c <= b;
```

"



"

Register Inference

- **Combinational:**

```
reg y;
always @(a or b or sel)
  if (sel) y = a;
  else y = b;
```

Sensitive to changes on all of the variables it reads

Y is always assigned

- **Sequential:**

```
reg q;
always @(d or clk)
  if (clk) q = d;
```

q only assigned when clk is 1

Summary of Verilog

- **Systems described hierarchically**
  - **Modules with interfaces**
  - **Modules contain instances of primitives, other modules**
  - **Modules contain initial and always blocks**
- **Based on discrete-event simulation semantics**
  - **Concurrent processes with sensitivity lists**
  - **Scheduler runs parts of these processes in response to changes**

Modeling Tools

- **Switch-level primitives**
  - **CMOS transistors as switches that move around charge**
- **Gate-level primitives**
  - **Boolean logic gates**
- **User-defined primitives**
  - **Gates and sequential elements defined with truth tables**
- **Continuous assignment**
  - **Modeling combinational logic with expressions**
- **Initial and always blocks**
  - **Procedural modeling of behavior**

Language Features

- **Nets (wires) for modeling interconnection**
  - **Non state-holding**
  - **Values set continuously**
- **Regs for behavioral modeling**
  - **Behave exactly like memory for imperative modeling**
  - **Do not always correspond to memory elements in synthesized netlist**
- **Blocking vs. nonblocking assignment**
  - **Blocking behaves like normal "C-like" assignment**
  - **Nonblocking updates later for modeling synchronous behavior**

Language Uses

- Event-driven simulation
  - Event queue containing things to do at particular simulated times
  - Evaluate and update events
  - Compiled-code event-driven simulation for speed
- Logic synthesis
  - Translating Verilog (structural and behavioral) into netlists
  - Register inference: whether output is always updated
  - Logic optimization for cleaning up the result

SystemC

Quick Overview

- A SystemC program consists of module definitions plus a top-level function that starts the simulation
- Modules contain processes (C++ methods) and instances of other modules
- Ports on modules define their interface
  - Rich set of port data types (hardware modeling, etc.)
- Signals in modules convey information between instances
- Clocks are special signals that run periodically and can trigger clocked processes
- Rich set of numeric types (fixed and arbitrary precision numbers)

Three Types of Processes

- METHOD
  - Models combinational logic
- THREAD
  - Models testbenches
- CTHREAD
  - Models synchronous FSMs

METHOD Processes

- Triggered in response to changes on inputs
- Cannot store control state between invocations
- Designed to model blocks of combinational logic

METHOD Processes

```
SC_MODULE(onemethod) {
  sc_in<bool> in;
  sc_out<bool> out;

  void inverter();

  SC_CTOR(onemethod) {

    SC_METHOD(inverter);
    sensitive(in);

  }
};
```

Process is simply a method of this class

Instance of this process created

and made sensitive to an input

METHOD Processes

- Invoked once every time input “in” changes
- Should not save state between invocations
- Runs to completion: should not contain infinite loops
  - Not preempted

```
void onemethod::inverter() {
  bool internal;
  internal = in;
  out = ~internal;
}
```

Read a value from the port

Write a value to an output port

THREAD Processes

- Triggered in response to changes on inputs
- Can suspend itself and be reactivated
  - Method calls wait to relinquish control
  - Scheduler runs it again later
- Designed to model just about anything

THREAD Processes

```
SC_MODULE(onemethod) {
  sc_in<bool> in;
  sc_out<bool> out;

  void toggler();

  SC_CTOR(onemethod) {

    SC_THREAD(toggler);
    sensitive << in;
  }

};
```

Process is simply a method of this class

Instance of this process created

alternate sensitivity list notation

THREAD Processes

- Reawakened whenever an input changes
- State saved between invocations
- Infinite loops should contain a wait()

```
void onemethod::toggler() {
  bool last = false;
  for (;;) {
    last = in; out = last; wait();
    last = ~in; out = last; wait();
  }
}
```

Relinquish control until the next change of a signal on the sensitivity list for this process

CTHREAD Processes

- Triggered in response to a single clock edge
- Can suspend itself and be reactivated
  - Method calls wait to relinquish control
  - Scheduler runs it again later
- Designed to model clocked digital hardware

CTHREAD Processes

```
SC_MODULE(onemethod) {
  sc_in_clk clock;
  sc_in<bool> trigger, in;
  sc_out<bool> out;

  void toggler();

  SC_CTOR(onemethod) {

    SC_CTHREAD(toggler, clock.pos());
  }

};
```

Instance of this process created and relevant clock edge assigned

CTHREAD Processes

- **Reawakened at the edge of the clock**
- **State saved between invocations**
- **Infinite loops should contain a wait()**

```
void onemethod::toggler() {
  bool last = false;
  for (;;) {
    wait_until(trigger.delayed() == true);
    last = in; out = last; wait();
    last = ~in; out = last; wait();
  }
}
```

Relinquish control until the next clock cycle in which the trigger input is 1

Relinquish control until the next clock cycle

SystemC 1.0 Scheduler

- **Assign clocks new values**
- **Repeat until stable**
  - **Update the outputs of triggered SC_CTHREAD processes**
  - **Run all SC_METHOD and SC_THREAD processes whose inputs have changed**
- **Execute all triggered SC_CTHREAD methods. Their outputs are saved until next time**

Scheduling

- **Clock updates outputs of SC_CTHREADs**
- **SC_METHODs and SC_THREADs respond to this change and settle down**
- **Bodies of SC_CTHREADs compute the next state**