**One hour, CLOSED book. Reading and writing small snippets of PROMELA, but no access to SPIN.**

Concurrency

* + Two or more events are said to be concurrent if they occur within the same time interval
  + In concurrent programming, there are two basic units of execution:
    - **Process**
    - A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.
    - **Threads**
    - Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.
    - Informally – a set of threads blocked with no possibility of progress.
    - Formally – a set of threads, each holding an SMR needed by another thread in the set and waiting to acquire a resource which is already held
* Original (OS centric processes)

– Better resource utilization.

– Fairness among multiple users with multiple computations.

• Current (process centric threads)

– Exploiting multiple processors

– Moore's Law running out of steam (multi-core).

– Modeling: Divide & conquer on loosely related tasks.

– Simplify handling asynchronicity (e.g., mouse events)

– Throughput (even on single CPU systems)

– Responsiveness

* Safety: Nothing bad happens

– Incorrect behavior in context of concurrency

– Race conditions

– Memory barrier (caching)

– Overly optimistic compiler optimizations

* Liveness: Good things eventually happen

– One or more threads cannot make progress

– Deadlock

* Fairness: Let's share, boys and girls

– Starvation

– Livelock

* Performance

– TANSTAAFL

– Context switching overhead

– Disabled compiler optimizations

• Testing, hair-pulling, and Heisenbugs

The Ultimate Culprit - Shared, Mutable State

* + Shared Mutable Resources (SMR)
  + Deadlock (identifying, preventing, detecting)
  + Four necessary and sufficient conditions for deadlock to be possible.
  + Necessary means all must hold for deadlock to be possible.
  + Sufficient means if all hold deadlock is possible.

– Exclusive use of resources

– No preemption of resource hold

– Serial acquisition of resources

– Cyclic hold-and-wait graph

* + Having these four conditions guarantees that deadlock is possible. It does not guarantee that it will happen.
  + Deadlock can occur with both individual and pooled resources.

• Goal is to design deadlock out of the system

– Eliminate one of the four conditions

– Use allocation methods, such as, Bankers Algorithm, that will not allocate into an unsafe state

• Detect and recover:

– Detection – periodically scan allocation graph for deadlocks

– Recover – kill a thread

• Use a different concurrency mechanism not prone to deadlock

– Software Transaction Memory

* + Safety
  + Liveness
  + Starvation
  + Interleaving

WHY IS IT BAD?

-The fundamental operation is assignment to change state.

– Assignable variables are mutable.

– May be exposed as public (bad karma).

– May be exposed via interface methods (medium warm karma).

– Things get tricky very fast when > 1 thread can invoke a mutating function.

HOW TO FIX IT

– Make things immutable.

– Hide shared state behind sequential access.

– Provide mechanisms to support controlled access to shared, mutable state.

A shared, mutable resource (SMR) could be a shared mutable variable, or a device such as a communication channel, disk, or printer.

• Safety (job #1):

Mutually exclusive access to shared, mutable resource (SMR)

• Liveness 1:

If threads are trying to access an SMR, one eventually does.

• Liveness 2:

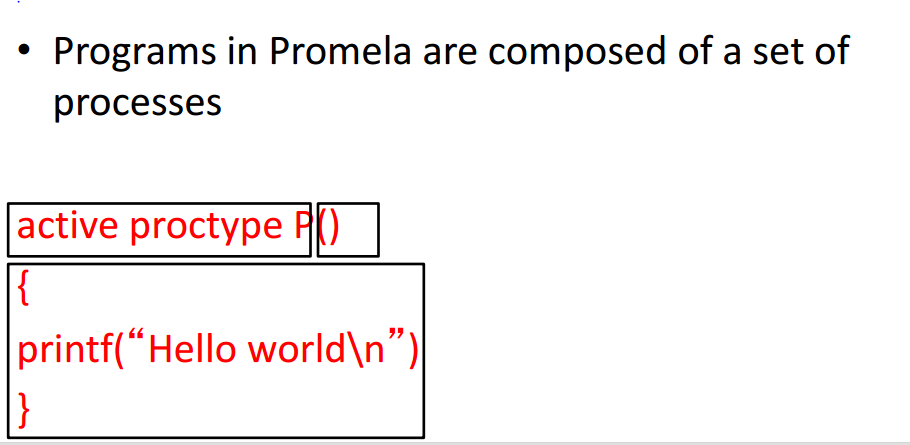
A thread holding an SMR eventually releases it.

• Fairness (no starvation):

If a thread is trying to access an SMR, it eventually gains access.

PROMELA (**PRO**CESS **ME**TA **LA**NGUAGE)

* simple language to model concurrent systems
* SPIN is a model checker that verifies these models



Use as few bits as possible to avoid combinatorial explosion

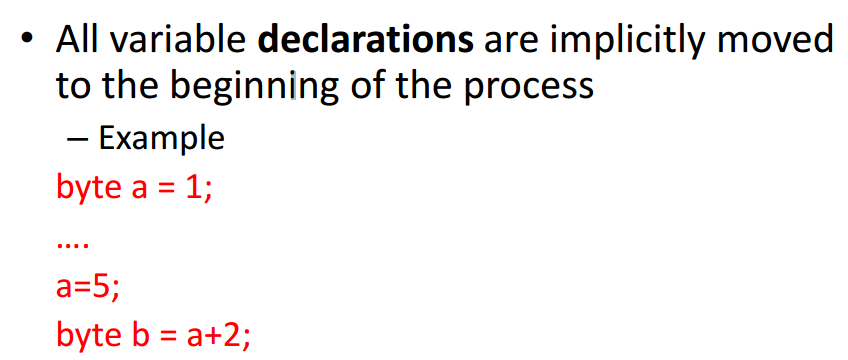
• The general rule: "smaller is better"

• All variables initialize to zero by default

• No character type in Promela

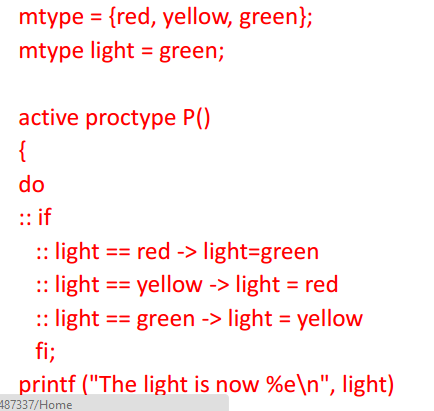
• No string variables in Promela

• No floating-point data types - Scope of a variable is entire process



**mtype** is used to give symbolic name (enum -  enables for a variable to be a set of predefined constants.)

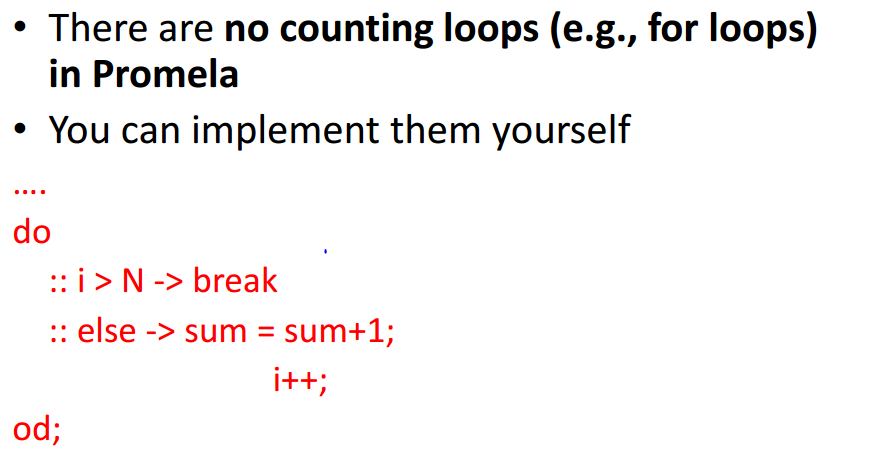
**CAN ONLY USE ONE FOR THE ENTIRE PROGRAM**

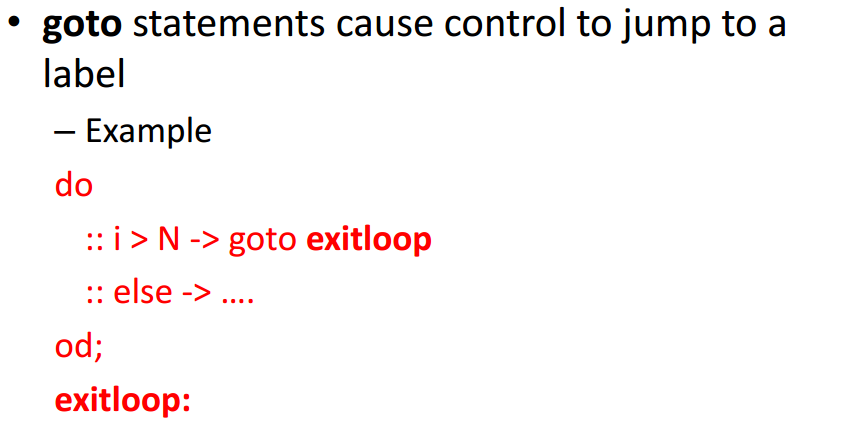


If there is more than one if statement choice that’s true. Promela will randomly pick one

Do statement (while loop-repetitively)

-Completion of the sequence of statements causes the execution to return to the beginning of the do statement





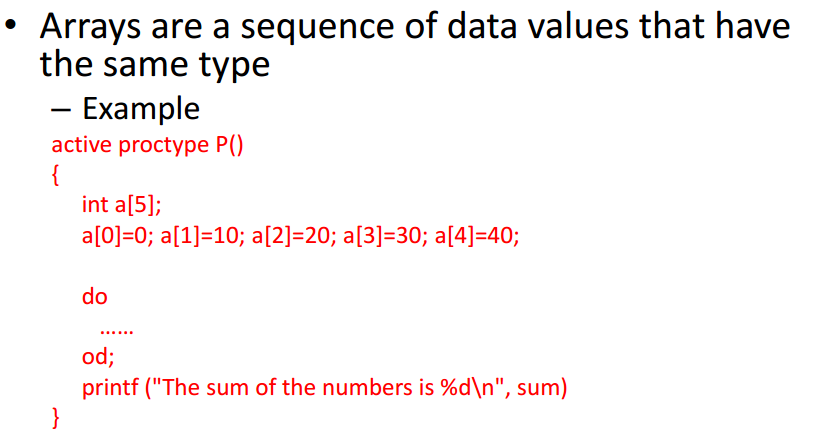
:: guard-statement -> command

-Jumps are allowed only within a process

• Jump statement labels need to be unique, per process

• Cannot place labels in front of guards

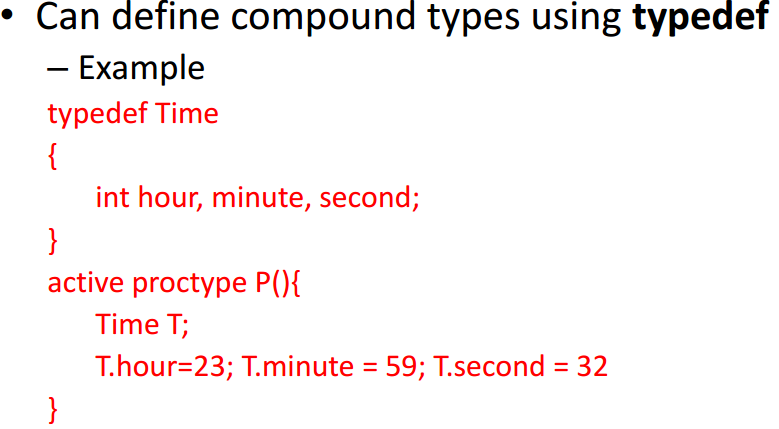
-break/goto is only way to get out of loop



Starts at 0

-Array bounds are constants and cannot be changed

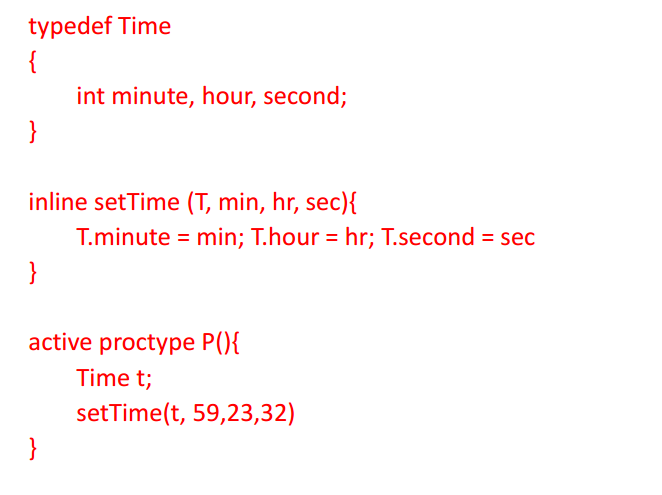
• Promela only allows one-dimensional arrays



-Promela does not have functions or procedures

• You can group statements together using the inline construct

• Does not create new scope for local variables



Assertions are placed between program statements and are evaluated by the model

checker for counterexamples

As noted above, you may be asked to "output" the results of a given section of PROMELA or provide a small snippet of PROMELA code.

* + Proctype
    - We can instantiate multiple processes using

– active [n] proctype P()

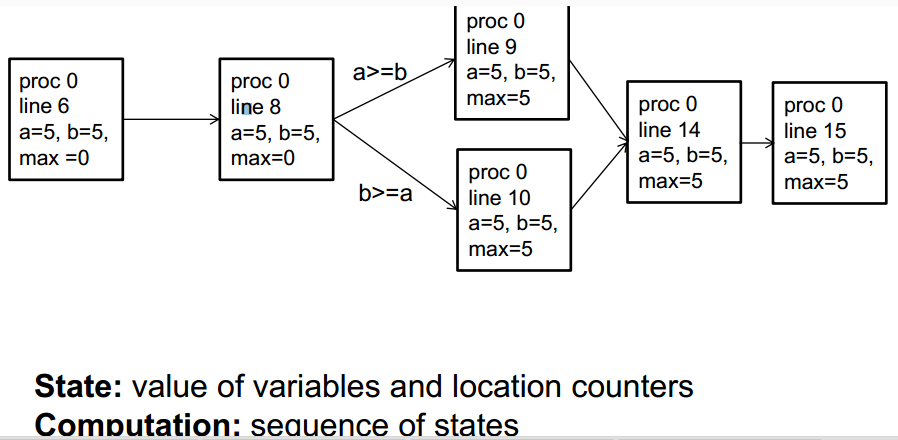
– where n is the number of processes we would like to ` instantiate

* + - We use the predefined variable \_pid to determine the process
    - Another way we can instantiate multiple processes is using
      * proctype P(byte id)

…

run P(1)

run P(2)

* + - We can use the predefined varaible \_nr\_pr to determine the number of active processes
  + Executability/Selection (Randomly selects one)
  + guards (nondeterministic execution)
    - are the “if” statements
    - if all guards are true one is randomly selected
    - if none of them are true the process is **BLOCKED** until one is true
    - can overlap
  + verification (accept(), LTL)
    - In a deterministic program, there is only one computation. Therefore, a single simulation will suffice
    - In a nondeterministic or concurrent program, we need to check all possible computations atomic
  + Verifier systematically checks the correctness among all possible computations

-As the program complexity increases, the number of computations will increase

• SPIN achieves efficiency by generating an optimized program called a verifier for a model -guided simulation helps locate where the bug occurs

Synchronization

* + Critical Sections
  + Mutual Exclusion
  + Busy-wait
  + Blocking
  + Semaphores
  + Atomicity
  + An expression or statement of a process that is executed entirely without
  + the possibility of interleaving is called atomic.
  + Assignments, jumps, skip, and expressions are atomic
  + In particular, conditional expressions are atomic:
  + (p -> q : r), C-style syntax, brackets required
  + Guarded commands are not atomic