Dynamic Systems

CS 2SD3

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Concepts: **dynamic** creation and deletion of **processes**

Resource allocation example – varying number of users and resources.

master-slave interaction

Models: **static - fixed populations with cyclic behavior**

interaction

Practice: **dynamic** creation and deletion of **threads**

(# active threads varies during execution)

Resource allocation algorithms

Java join() method

Golf Club Program

- Players at a Golf Club hire golf balls and then return them after use.
- Expert players tend not to lose any golf balls and only hire one or two.
- Novice players hire more balls, so that they have spares during the game in case of loss.
- However, they buy replacements for lost balls so that they return the same number that they originally hired.

Allocator: Allocator will accept requests for up to b balls, and block requests for more than b balls.

$$
\begin{array}{ll}
const\ N = 4 & //\text{maximum \#golf balls} \\
range\ B = 0..N & //\text{available range} \\
ALLOCATOR = BALL[N], & \\
BALL[b:B] = (when\ (b>0)\ get[i:1..b] \to BALL[b-i] & \\
\qquad\qquad\qquad \mid \underbrace{put[j:1..N]}_{??} \to BALL[b+j]). &
\end{array}
$$

?? - may potentially lead to an error

Players:

- How do we model the potentially infinite stream of dynamically created player processes?
- We cannot model infinite state spaces, but can model infinite (repetitive) behaviors.

Golf Club Model

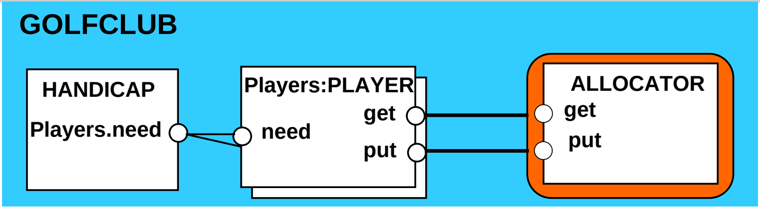
Players: Fixed population of golfers: infinite stream of requests.

$range\ R = 1..N$ $\quad$ //request range
$PLAYER = (need[b : R] \rightarrow PLAYER[b]),$
$PLAYER[b : R] = (get[b] \rightarrow put[b] - > PLAYER[b]).$

$set\ Experts = \{alice, bob, chris\}$
$setNovices = \{dave, eve\}$
$setPlayers = \{Experts, Novices\}$, i.e. $Players = Experts \cup Novices$

$HANDICAP$ $\quad$ //constraint on need action of each player.
$= (\{Novices.\{need[3..N]\}, Experts.need[1..2]\} \rightarrow HANDICAP)$
$\quad +\{Players.need[R]\}.$

$\|\ GOLFCLUB =$
$\quad (Players : PLAYER \parallel Players :: ALLOCATOR \parallel HANDICAP).$



$\| \; GOLFCLUB =$
$(Players : PLAYER \;\|\; Players :: ALLOCATOR \;\|\; HANDICAP).$

Safety: Do players return the right number of balls?

- **Yes**, for this case it can even be proved in some formal way!

Liveness: Are players eventually allocated balls?

- **Yes**, we can check:

  $progress\ NOVICE = \{NOVICES.get[R]\}$
  $progress\ EXPERT = \{EXPERTS.get[R]\}$

  However, are these properties correctly defined?

  Recall: $progress\ P = \{a_1, a_2, \ldots, a_n\}$ defines a *progress property P* which asserts that in any state of a target system, there is always a *continuation* trace which contains **at least one** element of $\{a_1, a_2, \ldots, a_n\}$. Don't we need **'for all'**?

Adverse Scheduling, i.e. looking for potential problems of an implementation

$progress\ NOVICE = \{NOVICES.get[R]\}$
$progress\ EXPERT = \{EXPERTS.get[R]\}$
$\|\ ProgressCheck \ = \ GOLFCLUB >> \{Players.put[R]\}.$

Progress violation: *NOVICE*
Trace to terminal set of states: $alice.need.2 \rightarrow bob.need.2 \rightarrow chris.need.2 \rightarrow chris.get.2 \rightarrow dave.need.4 \rightarrow eve.need.4$

- There are now only $3 = 5 - 2$ balls so to give 4 balls to *dave* or *eve*, *chris* need to execute *put*[ ]. But *put*'s have low priority! NOVICE players *dave* and *eve* suffer starvation. Actions in terminal set: $\{alice, bob, chris\}.\{get, put\}[2]$
- In fact, some EXPERTs also may suffer starvation, but there is no violation of *progress EXPERT*?
- Why? Because of '*at least one*' in the definition of *progress*!
- Weird !

Fair (but Inefficient) Allocation

- Allocation in arrival order, using tickets:

  $const\ TM = 5$ // maximum ticket

  $range\ T = 1..TM$ // ticket values

  $TICKET = NEXT[1],$

  $NEXT[t:T] = (ticket[t] \leftarrow NEXT[t\ mod\ (TM+1)]).$

- Players and Allocator:

  $PLAYER = (need[b:R] \rightarrow PLAYER[b]),$

  $PLAYER[b:R] =$

  $(ticket[t:T] \rightarrow get[b][t] \rightarrow put[b] \rightarrow PLAYER[b]).$

  $ALLOCATOR = BALL[N][1],$

  $BALL[b:B][t:T] =$

  $(when\ (b>0)\ get[i:1..b][t] \rightarrow BALL[b-i][t\ mod\ (TM+1)]$

  $\mid put[j:1..N] \rightarrow BALL[b+j][t]).$

- Ticketing increases the size of the model for analysis.
- We compensate by modifying the *HANDICAP* constraint:
  $$HANDICAP = (Novices.need[4], Experts.need[1] \rightarrow HANDICAP) + Players.need[R].$$
- Experts use 1 ball, Novices use 4 balls.

$$\| GOLFCLUB = (Players : PLAYER \; \| \; Players :: (ALLOCATOR \, \| \, TICKET) \| HANDICAP).$$

$const\ TM = 5$ // maximum ticket
$range\ T = 1..TM$ // ticket values
$TICKET = NEXT[1],$
$NEXT[t:T] = (ticket[t] \leftarrow NEXT[t\ mod\ (TM+1)]).$

$PLAYER = (need[b:R] \rightarrow PLAYER[b]),$
$PLAYER[b:R] =$
$(ticket[t:T] \rightarrow get[b][t] \rightarrow put[b] \rightarrow PLAYER[b]).$

$ALLOCATOR = BALL[N][1],$
$BALL[b:B][t:T] =$
$(when\ (b>0)\ get[i:1..b][t] \rightarrow BALL[b-i][t\ mod\ (TM+1)]$
$\mid put[j:1..N] \rightarrow BALL[b+j][t]).$

$HANDICAP = (Novices.need[4], Experts.need[1] \rightarrow$
$HANDICAP) + Players.need[R].$

$=====================================$

$\| \ GOLFCLUB = (Players : PLAYER$
$\| \ Players :: (ALLOCATOR \| TICKET) \| HANDICAP).$

Allocation in Arrival Order, Using Tickets: Analysis

- **Safety** (balls returned) is satisfied.
- **Liveness**, i.e.
  progress $NOVICE = \{Novices.get[R][T]\}$, and
  progress $EXPERT = \{Experts.get[R][T]\}$,
  is satisfied.
- It is still weird as it does not guarantee '*real liveness*'.
- **Adverse Scheduling**, i.e.
  $\| \, ProgressCheck = GOLFCLUB >> \{Players.put[R]\}$
  is also OK.
- Allocation in arrival order is not efficient. A better solution, called *bounded allocation* is discussed in the textbook. However it is too complex to be presented in class. It requires using some tools to be understood!

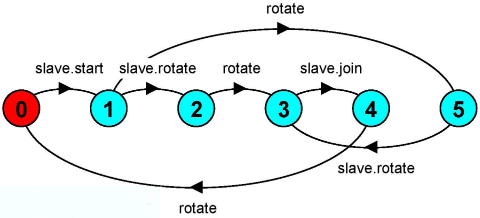
Master-Slave System

- A Master process/thread/module/etc creates a Slave process/thread/module/etc to perform some task (eg. I/O) and continues.
- Later, the Master synchronizes with the Slave to collect the result.
- Often the Slave dies after giving the result to the Master.

Master-Slave Model

$SLAVE = (start \rightarrow rotate \rightarrow join \rightarrow SLAVE)$.
$MASTER = (slave.start \rightarrow rotate \rightarrow slave.join \rightarrow rotate \rightarrow MASTER)$.
$\| \; MASTER\_SLAVE = (MASTER \; \| \; slave : SLAVE)$.

- *join* is modeled by a synchronized action.
- *slave.rotate* and *rotate* are interleaved ie. concurrent.
- Probably *master.rotate* instead of *rotate* would be a better name.



- Why only one slave?
- The model is vastly oversimplified!

My Comments

- FSP model **does not** work for dynamic system.
- The authors of the textbook did their best to apply FSP for modelling dynamic systems, but the results are weak and they seem to support my assertion.
- **A solution:** Extend FSP by adding some constructions from **pi-calculus**. FSP is a mixture of CSP (C. A. R. Hoare 1978-82) and CCS (R. Milner 1976-84), *pi-calculus* is also by R. Milner $\sim$ 1990-94.
- Neither standard Petri nets nor standard Model Checking techniques work well with dynamic systems.