Chapter 9

Dynamic Systems

Dynamic Systems

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

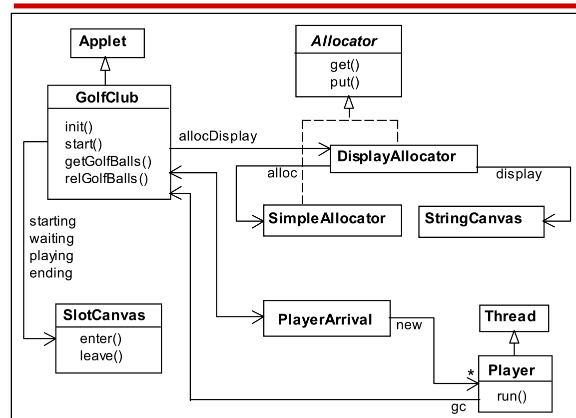
9.1 Golf Club Program

Players at a Golf Club hire golf balls and then return them after use.

Player d4 is waiting for four balls

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.

Golf Club - Java implementation



The Java interface **Allocator** permits us to develop a few implementations of the golf ball allocator without modifying the rest of the program.

DisplayAllocator class implements this interface and delegates calls to **get** and **put** to **SimpleAllocator**.

```
public interface Allocator {
  public void get(int n) throws InterruptedException;
  public void put(int n);
}
```

Java implementation - SimpleAllocator monitor

```
public class SimpleAllocator implements Allocator{
  private int available;

  public SimpleAllocator(int n)
    { available = n; }

  synchronized public void get(int n)
           throws InterruptedException {
    while (n>available) wait();
    available -= n;
  }

  synchronized public void put(int n) {
    available += n;
    notifyAll();
  }
}
```

get blocks a calling thread until sufficient golf balls are available.

A novice thread requesting a large number of balls may be overtaken and remain blocked!

Java implementation - Player thread

```
class Player extends Thread {
  private GolfClub gc;
  private String name;
  private int nballs;

  Player(GolfClub g, int n, String s) {
     gc = g; name = s; nballs =n;
  }

  public void run() {
    try {
      gc.getGolfBalls(nballs,name);
      Thread.sleep(gc.playTime);
      gc.relGolfBalls(nballs,name);
    } catch (InterruptedException e){}
  }
}
```

The run() method terminates after releasing golf balls. New player threads are created dynamically.

9.2 Golf Club Model

Allocator:

```
const N=5       //maximum #golf balls
range B=0..N    //available range

ALLOCATOR  =  BALL[N],
BALL[b:B]  =  (when (b>0) get[i:1..b]->BALL[b-i]
              |put[j:1..N]          ->BALL[b+j]
              ).
```

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

Players:

How do we model the potentially infinite stream of dynamically created player threads?

Cannot model infinite state spaces, but can model infinite (repetitive) behaviors.

Golf Club Model

Players:

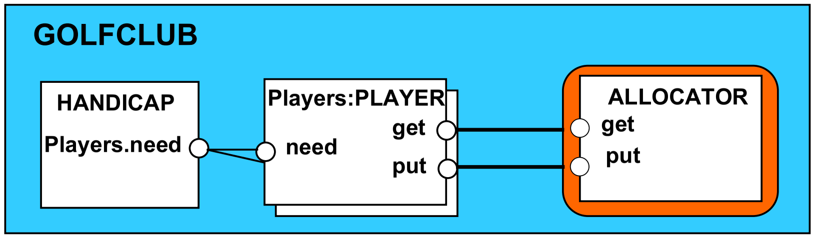
```
range R=1..N   //request range

PLAYER      = (need[b:R]->PLAYER[b]),
PLAYER[b:R] = (get[b]->put[b]->PLAYER[b]).

set Experts  = {alice,bob,chris}
set Novices  = {dave,eve}
set Players  = {Experts,Novices}

HANDICAP  =
 ({Novices.{need[3..N]},Experts.need[1..2]}
           -> HANDICAP
 ) +{Players.need[R]}.
```

Fixed population of golfers: infinite stream of requests.

Players is the union of Experts and Novices.

Constraint on need action of each player.

Golf Club Model - Analysis




```
||GOLFCLUB =( Players:PLAYER
            ||Players::ALLOCATOR
            ||HANDICAP).
```

Safety? *Do players return the right number of balls?*

Liveness? *Are players eventually allocated balls ?*

Golf Club Model - Liveness

```
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
     bob.need.2
     chris.need.2
     chris.get.2
     dave.need.5
     eve.need.5
Cycle in terminal set:
     alice.get.2
     alice.put.2
Actions in terminal set:
     {alice, bob, chris}.{get, put}[2]
```

Novice players dave and eve suffer starvation. They are continually overtaken by experts alice, bob and chris.

9.3 Fair 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]->NEXT[t%TM+1]).
```

Players and Allocator:

```
PLAYER      = (need[b:R]->PLAYER[b]),
PLAYER[b:R]= (ticket[t:T]->get[b][t]->put[b]
              ->PLAYER[b]).

ALLOCATOR        = BALL[N][1],
BALL[b:B][t:T]  =
  (when (b>0) get[i:1..b][t]->BALL[b-i][t%TM+1]
  |put[j:1..N]              ->BALL[b+j][t]
  ).
```


Fair Allocation - Analysis

Ticketing increases the size of the model for analysis. We compensate by modifying the HANDICAP constraint:

```
HANDICAP =
  ({Novices.{need[4]},Experts.need[1]}-> HANDICAP
  ) +{Players.need[R]}.
```

Experts use 1 ball, Novices use 4 balls.

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

```
progress NOVICE = {Novices.get[R][T]}
progress EXPERT = {Experts.get[R][T]}
```

Safety?

Liveness?

9.4 Revised Golf Club Program - FairAllocator monitor

```
public class FairAllocator implements Allocator {
  private  int available;
  private  long turn = 0; //next ticket to be dispensed
  private  long next = 0; //next ticket to be served

 public FairAllocator(int n) { available = n; }

  synchronized public void get(int n)
          throws InterruptedException {
    long myturn = turn;  ++turn;
    while (n>available || myturn != next) wait();
    ++next; available -= n;
    notifyAll();
  }

  synchronized public void put(int n) {
    available += n;
    notifyAll();
  }
}
```

Block calling thread until sufficient balls and next turn.

Why is it necessary for get to include notifyAll()?

Revised Golf Club Program - FairAllocator

Players g1 and h1 are waiting. Even though two balls are available, they cannot overtake player f4.

What happens if c, d and e all return their golf balls?

9.5 Bounded Allocation

Allocation in arrival order is not efficient. A **bounded allocation** scheme allows experts to overtake novices but denies starvation by setting an **upper bound on the number of times a novice can be overtaken**.

We model players who have overtaken others as a **set**.

```
const False = 0
const True  = 1
range Bool  = 0..1

ELEMENT(Id=0)  = IN[False],
IN[b:Bool]     = ( add[Id]           -> IN[True]
                 | remove[Id]        -> IN[False]
                 | contains[Id][b]   -> IN[b]
                 ).
||SET = (forall[i:T] (ELEMENT(i))).
```

A SET is modeled as the parallel composition of elements

Bounded Allocation - Allocator model

We model bounded overtaking using tickets, where ticket numbers indicate the order in which players make their requests. The allocator records which ticket number is next.

Overtaking occurs when we allocate balls to a player whose turn - indicated by his/her ticket number – is subsequent to a waiting player with the next ticket. The overtaking player is added to the **overtaking set**, and a count ot is incremented to indicate the number of times next has been overtaken.

When the count equals the bound, we allow allocation to the next player only. When allocation is made to the next player, we update next to indicate the next (waiting) player. We skip the ticket numbers of overtaking players who already received their allocation, remove each of these intervening players from the overtaking set and decrement the overtaking count ot accordingly. (This is achieved in the local process, WHILE, in the ALLOCATOR model.)

Bounded Allocation - Allocator model

```
ALLOCATOR  = BALL[N][1][0],     //initially N balls, 1 is next, empty set
BALL[b:B][next:T][ot:0..Bd] =
     (when (b>0 && ot<Bd) get[i:1..b][turn:T] ->
          if (turn!=next) then
               (add[turn] -> BALL[b-i][next][ot+1])
          else
               WHILE[b-i][next%TM+1][ot]
     |when (b>0 && ot==Bd) get[i:1..b][next] ->
               WHILE[b-i][next%TM+1][ot]
     |put[j:1..N] -> BALL[b+j][next][ot]
),
WHILE[b:B][next:T][ot:0..Bd] =
     (contains[next][yes:Bool] ->
          if (yes) then
             (remove[next] -> WHILE[b][next%TM+1][ot-1])
          else BALL[b][next][ot]
)+{add[T],remove[T]}.
```

Bounded Allocation - Allocator model

where

```
const N   =  5      // maximum #golf balls
const Bd  =  2      // bound on overtaking
range B   =  0..N   // available range

const TM  = N + Bd  // maximum ticket
range T   = 1..TM   // ticket values
```

```
||GOLFCLUB = (Players:PLAYER
             || ALLOCATOR||TICKET||SET
             || HANDICAP
             )/ {Players.get/get, Players.put/put,
                 Players.ticket/ticket}.
```


Bounded Allocation - an explanatory Trace

```
eve.need.4          Experts Eve and Dave
dave.need.4
chris.need.1        Novices Alice, Bob and Chris
alice.need.1
bob.need.1
alice.ticket.1
alice.get.1.1       Alice gets 1 ball, ticket 1
contains.2.0        Ticket 2 is next
bob.ticket.2
bob.get.1.2         Two allocated, three available
contains.3.0        Ticket 3 is next
dave.ticket.3       Dave needs four balls: waits
chris.ticket.4
chris.get.1.4       Chris overtakes
add.4
eve.ticket.5        Eve needs four balls: waits
alice.put.1
alice.ticket.6
alice.get.1.6       Alice overtakes
add.6
bob.put.1
bob.ticket.7
bob.get.1.7         Bob overtakes: bound reached
add.7
```

Using animation, we can perform a scenario and produce a trace.

Bounded Allocation - an explanatory Trace

```
chris.put.1
chris.ticket.8      Chris waits: three available
alice.put.1
alice.ticket.1      Alice waits: four available
dave.get.4.3        Dave gets four balls
contains.4.1        remove intervening overtaker
remove.4
contains.5.0        Ticket 5 (Eve) is next
dave.put.4
dave.ticket.2
alice.get.1.1       Alice overtakes: bound reached
add.1
bob.put.1
bob.ticket.3
eve.get.4.5         Eve gets four balls
contains.6.1        remove intervening overtakers
remove.6
contains.7.1
remove.7
contains.8.0        Ticket 8 (Chris) is next
. . .
```

Exhaustive checking:

Safety?

Liveness?

Can we also specify the bounded nature of this allocator as a safety property?

Bounded Allocation – Safety Property

For each player, check that he/she is not overtaken more than bound times. Overtaking is indicated by an allocation to another player whose ticket t lies between the turn of the player and the latest ticket.

```
property BOUND(P='alice) =
    ({Players\{[P]}}.ticket[T] -> BOUND
    |[P].ticket[t:T]   -> WAITING[t][t][0]
    |[Players].get[R][T]        -> BOUND
    ),
WAITING[turn:T][latest:T][overtaken:0..Bd] =
    ([P].get[b:R][turn] -> BOUND
    |{Players\{[P]}}.get[b:R][t:T] ->
        if ((t>turn && (t<=latest || latest<turn))
          ||(t<turn && (t<=latest && latest<turn)))
        then WAITING[turn][latest][overtaken+1]
        else WAITING[turn][latest][overtaken]
    |Players.ticket[last:T] ->WAITING[turn][last][overtaken]
).
```

Action labels used in expressions or as parameter values must be prefixed with a single quote.

9.6 Bounded Overtaking Allocator - implementation

Implementation of the `BoundedOvertakingAllocator` monitor follows the algorithm in the model.

Novice player f4 has been overtaken by expert players g1, h1 and i1. Since the overtaking bound of three has been exceeded, players j1 and k1 are blocked although there are two golf balls available.

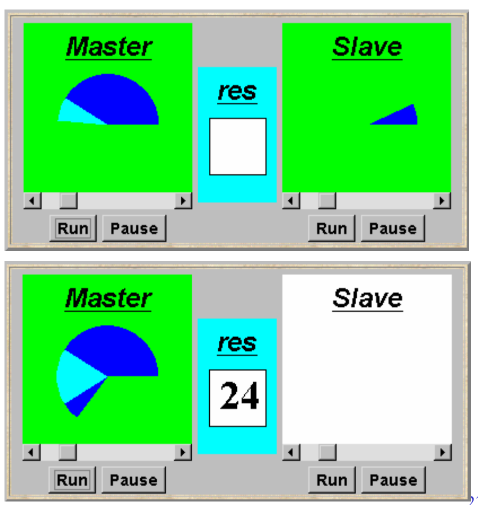
9.7 Master-Slave Program

A Master thread creates a Slave thread to perform some task (eg. I/O) and continues.

Later, the Master synchronizes with the Slave to collect the result.

How can we avoid busy waiting for the Master?

Java `class Thread` provides method `join()` which waits for the thread to die, i.e. by returning from `run()` or as a result of `stop()`.



Java implementation - Master-Slave

```
class Master implements Runnable {
  ThreadPanel slaveDisplay;
  SlotCanvas  resultDisplay;

  Master(ThreadPanel tp, SlotCanvas sc)
    {slaveDisplay=tp; resultDisplay=sc;}

  public void run() {
    try {
      String res=null;
      while(true) {
        while (!ThreadPanel.rotate());
        if (res!=null) resultDisplay.leave(res);
        Slave s = new Slave();        // create new slave thread
        Thread st = slaveDisplay.start(s,false);
        while (ThreadPanel.rotate()); // continue execution
        st.join();                    // wait for slave termination
        res = String.valueOf(s.result()); //get and display result from slave
        resultDisplay.enter(res);
      }
    } catch (InterruptedException e){}
  }
}
```

Slave thread is created and started using the `ThreadPanel` method `start`.

Java implementation - Master-Slave

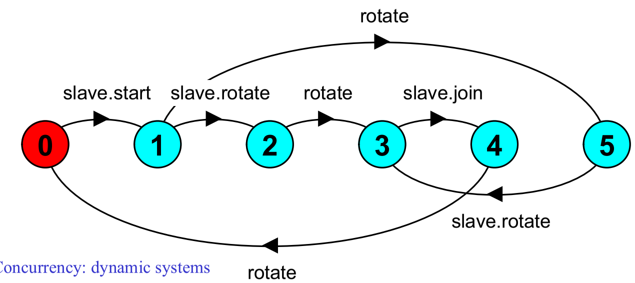
```
class Slave implements Runnable {
  int rotations = 0;

 public void run() {
    try {
      while (!ThreadPanel.rotate()) ++rotations;
    } catch (InterruptedException e){}
  }

  int result(){
    return rotations;
  }
}
```

Slave method `result` need not be synchronized to avoid inteference with the Master thread. Why not?

9.8 Master-Slave Model

```
SLAVE = (start->rotate->join->SLAVE).

MASTER = (slave.start->rotate
         ->slave.join->rotate->MASTER).

||MASTER_SLAVE = (MASTER || slave:SLAVE).
```

join is modeled by a synchronized action.



slave.rotate and rotate are interleaved ie. concurrent

Summary

- 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**

Course Outline

2. Processes and Threads
3. Concurrent Execution
4. Shared Objects & Interference
5. Monitors & Condition Synchronization
6. Deadlock
7. Safety and Liveness Properties
8. Model-based Design

The main basic Concepts, Models, Practice

Advanced topics …

9. **Dynamic systems**
10. Message Passing
11. Concurrent Software Architectures
12. Timed Systems
13. Program Verification
14. Logical Properties