9.1 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.

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.

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

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

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

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

Players:

Players is the union of Experts and Novices.

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

9.2 Golf Club Model

Players:

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

Fixed population of golfers: infinite stream of requests.

Constraint on need action of each player.
Golf Club Model - Analysis

**GOLFCLUB**

| HANDICAP | Players:PLAYER get, need | ALLOCATOR get, put |

Safety? Do players return the right number of balls?

Liveness? Are players eventually allocated balls?

Golf Club Model - Liveness

```lict
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]

9.3 Fair Allocation

Allocation in arrival order, using tickets:

```c
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:

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

Experts use 1 ball, Novices use 4 balls.

Safety? Liveness?
9.4 Revised Golf Club Program - FairAllocator monitor

```java
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();
    }
}
```

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

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

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]}

where
const N = 5 // maximum #golf balls
const Bd = 2 // bound on overtaking range
const B = 0..N // available range
const TM = N + Bd // maximum ticket range
const 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

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

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.

\[
\text{property } \text{BOUND} = \{ \text{Players} \{ [P] \}.\text{ticket} [t] \rightarrow \text{BOUND} \\
| \text{[P].ticket} [t:T] \rightarrow \text{WAITING}[t][t][0] \\
| \text{[Players].get}[R][T] \rightarrow \text{BOUND} \}.
\]

\[
\text{WAITING}[\text{turn}:T][\text{latest}:T][\text{overtaken}:0..Bd] = \\
(\text{[P].get}[b:R][\text{turn}] \rightarrow \text{BOUND} \\
| \text{[Players} \{ [P] \}.\text{get}[b:R][t:T] \rightarrow \\
\text{if } ((t>\text{turn} \&\& (t<=\text{latest} || \text{latest}<\text{turn})) \\
\text{|| } (t<\text{turn} \&\& (t<=\text{latest} && \text{latest}<\text{turn})) \\
\text{then } \text{WAITING}[\text{turn}][\text{latest}][\text{overtaken}+1] \\
\text{else } \text{WAITING}[\text{turn}][\text{latest}][\text{overtaken}] \\
| \text{Players.ticket}[\text{last}:T] \rightarrow \text{WAITING}[\text{turn}][\text{last}][\text{overtaken}] 
\}.
\]

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 \( \text{join()} \) which waits for the thread to die, i.e. by returning from \( \text{run()} \) or as a result of \( \text{stop()} \).
### Java implementation - Master-Slave

```java
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 interference 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).

\[\text{slave.rotate and rotate are interleaved i.e. concurrent}\]

---

### 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)
  - Java `join()` method

---

### Course Outline

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

---

Advanced topics ...