Restroom Problem

```
byte men = 0;
byte women = 0;
byte mutexH = 1;
byte mutexM = 1;
byte mutexHM = 1;
byte toilet = 10;

inline acquire(sem) {
 atomic {
  sem>0;
  sem--
 }
}

inline release(sem) {
 sem++
}
active [4] proctype menThread() {
 /* Complete */
 acquire(mutexHM);
 acquire(mutexH);

 men++;
 if
 :: men == 1 -> acquire(mutexM)
 :: else -> skip
 fi;
 release(mutexH);
 release(mutexHM);
 acquire(toilet);
 assert(women == 0 && men <= 10);
 release(toilet);
 acquire(mutexH);
 men--;
 if
 :: men == 0 -> release(mutexM);
 :: else -> skip;
 fi;
 release(mutexH);
}

active [4] proctype womanThread() {
 /* complete */
 acquire(mutexHM);
 acquire(mutexM);

 women++;
 if
 :: women == 1 -> acquire(mutexH);
 :: else -> skip;
 fi;
 release(mutexM);
 release(mutexHM);
 acquire(toilet);
 assert(men == 0 && women <= 10);
 release(toilet);
 acquire(mutexM);
 women--;
 if
 :: women == 0 -> release(mutexH);
 :: else -> skip;
 fi;
```

semaphores

```
byte sem = 1;
byte critical = 0;
inline acquire(sem) {
  atomic {
    sem > 0;
    sem--
  }
}
inline release(sem) {
  sem++
}
active [2] proctype user() {
  do
    :: acquire(sem);
    printf("%d is in the
            CS\n", _pid);
    critical++;
    assert(critical==1)
    critical--;
    assert(critical==0)
    release(sem)
  od
}
```

bar

```
byte mutex = 1;
byte ticket = 0;
byte m = 0;
byte w = 0;
inline acquire(sem) {
    atomic {
        sem > 0;
        sem--
    }
}
inline release(sem) {
    sem++
}
active [5] proctype man(){
    acquire(mutex);
    acquire(ticket);
    acquire(ticket);
    release(mutex);
    m++;
    assert(m*2 <= w)
}
active [10] proctype woman(){
    atomic {
        release(ticket);
        w++
    }
}
```

endstate

```
byte request = 0;
active proctype Server1() {
  do
  :: request == 1 ->
    printf("Service 1\n");
    request = 0;
  od
}
active prototype Server2() {
  do
  :: request == 2 ->
    printf("Service 2\n");
    request = 0;
  od
}
active prototype Client() {
  request = 1;
  request == 0;
  request = 2;
  request == 0;
}
```

concat

```
-module(concat).
-compile(export_all).
concat(Dict) ->
    receive
        {CPID, start} ->
            %CPID ! {self(), ok},
            concat(dict:store(CPID, "", Dict));
        {CPID, add, S} ->
            %CPID ! {self(), ok},
            concat(dict:store(CPID,
                dict:fetch(CPID, Dict) ++ S, Dict));
        {CPID, done} ->
            String = dict:fetch(CPID, Dict),
            CPID ! {self(), result, String},
            concat(dict:erase(CPID, Dict));
        stop ->
            stop
    end.

client1(SPID) ->
    SPID ! {self(), start},
    %receive X -> X end,
    SPID ! {self(), add, "Hello"},
    timer:sleep(10),
    %receive X -> X end,
    SPID ! {self(), add, " World"},
    %receive X -> X end,
    SPID ! {self(), done},
    receive {SPID, result, S} -> io:format("~p", [S]) end.

client2(SPID) ->
    SPID ! {self(), start},
    %receive X -> X end,
    SPID ! {self(), add, "Test"},
    %receive X -> X end,
    SPID ! {self(), add, " Message"},
    %receive X -> X end,
    SPID ! {self(), add, " Complete"},
    SPID ! {self(), done},
    receive {SPID, result, S} -> io:format("~p", [S]) end.

start() ->
    SPID = spawn(?MODULE, concat, [dict:new()]),
    C1 = spawn(?MODULE, client1, [SPID]),
    C2 = spawn(?MODULE, client2, [SPID]).
```

Prime numbers Server

```
-module(primeA).
-compile(export_all).
isPrime(N) when N == 2 ->
    true;
isPrime(N) when N < 2 orelse N rem 2 == 0 ->
    false;
isPrime(N) ->
    isPrimeH(N, 3).

isPrimeH(N, K) when K * K > N ->
    true;
isPrimeH(N, K) when N rem K == 0 ->
    false;
isPrimeH(N, K) ->
    isPrimeH(N, K + 2).
prime() ->
    receive
        {CID, N} ->
            CID ! {self(), isPrime(N)},
            prime();
        stop ->
            stop
    end.
start() ->
    SID = spawn(?MODULE, prime, []),
    SID ! {self(), 1},
    receive
        {SID, B} -> io:format("~p\n", [B])
    end,
    SID ! {self(), 7},
    receive
        {SID, B2} -> io:format("~p\n", [B2])
    end,
    SID ! stop.
```

Guess the number <-

```
-module (guess).
-compile(export_all).

start() ->
    S = spawn(fun server/0).
    %spawn(?MODULE, client, [S, 0]).
server() ->
    receive
        {From, Ref, start} ->
            LS = spawn(?MODULE, loop_server,
                [rand:uniform(100)]),
            From!{self(), Ref, LS}
    end,
    server().
loop_server(N) ->
    receive
        {From, Ref, N} ->
            From!{self(), Ref, gotIt};
        {From, Ref, _} ->
            From!{self(), Ref, tryAgain},
            loop_server(N)
    end.
client(S) ->
    Ref = make_ref(),
    S!{self(), Ref, start},
    receive
        {S, Ref, Servlet} ->
            client_loop(Servlet,0)
    end.
client_loop(S, T) ->
    N = rand:uniform(100),
    Ref = make_ref(),
    S!{self(), Ref, N},
    receive
        {From, Ref, gotIt} ->
            io:format("Got it after ~w
                attempts, it was ~w~n", [T, N]);
        {From, Ref, tryAgain} ->
            client_loop(S, T+1)
    end.
```

turnstiles

```
-module(ts).
-compile(export_all).
%% counter: should support two "operations":
%% read and bump
counter(N) ->
   receive
       {From,R,bump} ->
           From!{self(),R,ok},
           counter(N+1);
       {From,R,read} ->
           From!{self(),R,N},
           counter(N);
       stop ->
           stop
end.
%% turnstile: should bump the counter every
%% so many milliseconds
turnstile(TPID,_CPID,0) ->
    TPID!{self(),done};
turnstile(TPID,CPID,N) ->
    timer:sleep(rand:uniform(50)),
    R=make_ref(),
    CPID!{self(),R,bump},
    receive
       {CPID,R,ok} ->
           turnstile(TPID,CPID,N-1)
    end.
start() ->
    CPID = spawn(?MODULE,counter,[0]),
    T1PID = spawn(?MODULE,turnstile,[self(),
        CPID,50]),
    T2PID = spawn(?MODULE,turnstile,[self(),
        CPID,50]),
    %%% Wait for some amount of time and then
    %% print value of counter
    %%% timer:sleep(rand:uniform(2000)),
    receive
        {T1PID,done} ->
            receive
                {T2PID,done} ->
                    ok
            end
    end,
    R=make_ref(),
    CPID!{self(),R,read},
    receive
       {CPID,R,N} ->
           io:format("The value of the counter
            is ~w~n",[N])
    end,
    CPID!stop.
```

Barriers

```
-module(bs).
-compile(export_all).
coordinator(N) ->
    S = spawn(?MODULE,barrier_loop,
        [N, N, []]),
    register(barrier, S).
barrier_loop(0,N,L) ->
    [From!{Ref,ok} || {From,Ref} <- L],
    barrier_loop(N,N,[]);
barrier_loop(N,M,L) when N>0 ->
    receive
        {From,Ref,arrived} ->
            barrier_loop(N-1,M,
                [{From,Ref} | L])
    end.
run_clients() ->
    spawn(fun client1/0),
    spawn(fun client2/0).
client1() ->
    io:format("a~n"),
    B = whereis(barrier),
    R = make_ref(),
    B!{self(), R, arrived},
    receive
        {R,ok} ->
            io:format("b~n")
    end.
client2() ->
    io:format("c~n"),
    B = whereis(barrier),
    R = make_ref(),
    B!{self(), R, arrived},
    receive
        {R,ok} ->
            io:format("d~n")
    end.
```

```
-module(fs).
-compile(export_all).
start(State,F) ->
    S=spawn(?MODULE,server,[State,F]),
    register(server,S).
server(State,F) ->
    receive
    {From, Ref, apply, N} ->
        case (catch (F(State,N))) of
        {'EXIT',Reason} ->
            From!{self(),Ref,error,Reason};
        {NewState,Result} ->
            From!{self(),Ref,Result},
            server(NewState,F)
        end;
    {From, Ref, update, G} ->
        From!{self(),Ref,okUpdate},
        server(State,G);
    {From, Ref, getState} ->
        From!{self(),Ref,State},
        server(State,F)
    end.
run_client(X) ->
    spawn(?MODULE,client,[X]).
client(X) ->
    S = whereis(server),
    R = make_ref(),
    S!{self(),R,apply,X},
    receive
    {From,Ref,Result} ->
        io:format("Result is ~w~n",
            [Result]);
    {From,Ref,error,Reason} ->
        io:format("Error in computation:
            ~w~n",[Reason])
    end.
```

General server <-

timer

```
-module(myTimer).
-compile(export_all).
sendTicks([]) ->
    sent;
sendTicks([H|T]) ->
    H ! tick,
    sendTicks(T).
myTimer(PidList) ->
    receive
        stop -> done
    after 0 ->
        timer:sleep(1000),
        sendTicks(PidList),
        myTimer(PidList)
    end.
client() ->
    receive
        tick ->
            io:format("Received Tick\n"),
            client();
        stop ->
            done
    end.
start() ->
    C1 = spawn(?MODULE, client, []),
    C2 = spawn(?MODULE, client, []),
    C3 = spawn(?MODULE, client, []),
    Server = spawn(?MODULE, myTimer, [[C1, C2, C3]]),
    timer:sleep(3500),
    C1 ! stop,
    C2 ! stop,
    C3 ! stop,
    Server ! stop.
```

Factorial server <-

```
-module(gs).
-compile(export_all).

fact(0) ->
    1;
fact(N) when N>0 ->
    N * fact(N-1).

start(F) ->
    spawn(?MODULE, server, [0,F]).

server(State, F) ->
    receive
        {From, Ref, compute, N} ->
            {UState,Result} = F(State,N),
            From!{self(), Ref, fact(N)},
            server(UState, F);
            }
        {From, Ref, update, G} ->
            From!{self(), Ref, okUpdate},
            server(State,G);
        {From, Ref, getState} ->
            From!{self(), Ref, State},
            server(State,F);
    end.

client(S,X) ->
    R = make_ref(),
    S!{self(), R, X},
    receive
        {S, R, compute, FN} ->
            io:format("Got ~w~n", [FN])
end.
```