CHALMERS

UNIVERSITY OF
GOTHENBURG

Synchronization problems with message-passing

Lecture 8 of TDA384/DIT391
Principles of Concurrent Programming

Nir Piterman

Chalmers University of Technology | University of Gothenburg

SP1 2020/2021

Based on course slides by Carlo A. Furia and Sandro Stucki

Today’s menu

Barriers

Resource allocator

Producer-consumer

Readers-writers

Dining philosophers

A gallery of synchronization problems

In today's class, we go through several classical synchronization problems and solve them using processes and message passing.

On the course website you can download fully working implementations of some of the problems.

A gallery of synchronization problems

In today's class, we go through several classical synchronization problems and solve them using processes and message passing.

On the course website you can download fully working implementations of some of the problems.

Solving these problems with message passing has a different style than using semaphores or monitors:

- mutual exclusion is not an issue, since there are no shared variables
- coordination is the main problem, which is achieved by exchanging messages asynchronously

The solutions are in the style of servers, which run event-loop functions that handle requests from clients thus coordinating them.

Barriers

Reusable barriers – recap

```
-module(barrier).
% initialize barrier for ‘Expected’ processes
init(Expected) -> todo.
% block at ‘Barrier’ until all processes have reached it
wait(Barrier) -> todo.
```

Reusable barrier: implement module `barrier` such that:

- a process blocks on `wait` until all processes have reached the Barrier
- after `Expected` threads have executed `wait`, the barrier is closed again

Processes at a reusable barrier

Processes continuously approach the barrier, which must guarantee that they synchronize each access.

---

process$_k$

```
process(Barrier) ->
  % code before barrier
  barrier:wait(Barrier) % synchronize at barrier
  % code after barrier
  process(Barrier).
```

Barrier process

The barrier process keeps track of the processes that have arrived at the barrier:

- when a new process arrives, it sends an arrived message to the barrier; the barrier updates its list of arrived processes
- when the list of arrived processes is complete, the barrier sends a continue message to all processes
- after notifying all processes, the barrier goes back to the initial state, ready for a new iteration

We implement the barrier's event loop as a server function:

```
barrier(Arrived, Expected, PidRefs)
```

where `Arrived` processes have arrived so far, out of a total of `Expected`; `PidRefs` is a list of the pids and unique references of `arrived` messages sent to the barrier (thus it has `Arrived` elements).

The server function `barrier`

```
% event loop of barrier for ‘Expected’ processes
%   Arrived: number of processes arrived so far
%   PidRefs: list of {Pid, Ref} of processes arrived so far
barrier(Arrived, Expected, PidRefs)
   when Arrived =:= Expected ->      % all processes arrived
     % notify all waiting processes
     [To ! {continue, Ref} || {To, Ref} <- PidRefs],
     % reset barrier
     barrier(0, Expected, []);
barrier(Arrived, Expected, PidRefs) ->
   receive  % still waiting for some processes
      {arrived, From, Ref} ->
        % one more arrived: add {From, Ref} to PidRefs list
        barrier(Arrived+1, Expected, [{From, Ref}|PidRefs])
   end.
```

`Arrived` is redundant because it is equal to `length(PidRefs)`, but we keep it for clarity.

The function `wait`

The function `wait` exchanges messages with the `Barrier` process running `barrier`; it is used so that synchronizing processes do not need to know about the format of exchanged messages.

```
% block at ‘Barrier’ until all processes have reached it
wait(Barrier) ->
   Ref = make_ref(),
     % notify barrier of arrival
   Barrier ! {arrived, self(), Ref},
     % wait for signal to continue
   receive {continue, Ref} -> through end.
```

The function `wait`

The function `wait` exchanges messages with the `Barrier` process running `barrier`; it is used so that synchronizing processes do not need to know about the format of exchanged messages.

```
% block at ‘Barrier’ until all processes have reached it
wait(Barrier) ->
   Ref = make_ref(),
     % notify barrier of arrival
   Barrier ! {arrived, self(), Ref},
     % wait for signal to continue
   receive {continue, Ref} -> through end.
```

pid of process executing `wait`

dummy value

Barrier initialization

Initializing a barrier consists of spawning a process running `barrier`.

```
% initialize barrier for ‘Expected’ processes
init(Expected) ->
   spawn(fun () -> barrier(0, Expected, []) end).
```

initially, no processes have arrived yet

The caller gets the barrier’s pid, which should be distributed to all processes that want to use the barrier.

Resource allocator

Resource allocator: the problem – recap

An allocator grants users exclusive access to a number of resources:

- users asynchronously request resources and release them back
- the allocator ensures resources are given exclusively to one user at a time, and keeps tracks of how many resources are available

```
-module(allocator).
% register ‘allocator’ with list of Resources
init(Resources) -> todo.
% get ‘N’ resources from ‘allocator’
request(N) -> todo.
% release ‘Resources’ to ‘allocator’
release(Resources) -> todo.
```

Resource allocator problem: implement `allocator` such that:

- an arbitrary number of users can access the allocator
- users are granted exclusive access to resources

Users

Users continuously and asynchronously access the allocator, which must guarantee proper synchronization.

---

user$_k$

```
user() ->
  % how many resources are needed?
  N = howMany(),
  % get resources from allocator
  Resources = allocator:request(N),
  % do something with resources
  use(Resources),
  % release resources
  allocator:release(Resources),
  user().
```

Allocator process

The allocator process keeps track of the list of available resources:

- when a process requests some resources that are available, the allocator sends a granted message to the process, and removes those just granted from the list of available resources
- when a process releases some resources, the allocator sends a released message to the process, and adds those just released to the list of available resources
- requests that exceed the availability implicitly queue in the allocator's mailbox; they will be served as soon as enough resources are available

We implement the allocator's event loop as a server function:

```
allocator(Resources)
```

where `Resources` is the list of available resources.

The server function `allocator`: handling requests

```
allocator(Resources) ->
   % count how many resources are available
   Available = length(Resources),
   receive
     % serve requests if enough resources are available
     {request, From, Ref, N} when N =< Available ->
        % Granted ++ Remaining =:= Resources
        % length(Granted) =:= N
        {Granted, Remaining} = lists:split(N, Resources),
        % send resources to requesting process
        From ! {granted, Ref, Granted},
        % continue with Remaining resources
        allocator(Remaining);
```

does not match if `N > Available`

The server function `allocator`: handling releases

```
allocator(Resources) ->
   % count how many resources are available
   Available = length(Resources),
   receive
     % serve requests: previous slide...

     % serve releases
     {release, From, Ref, Released} ->
        % notify releasing process
        From ! {released, Ref},
        % continue with previous and released resources
        allocator(Resources ++ Released)
    end.
```

The functions `request` and `release`

The functions `request` and `release` exchange messages with the process registered as `allocator`; they are used so that synchronizing processes do not need to know about the format of exchanged messages.

```
% get ‘N’ resources from ‘allocator’; block if not available
request(N) ->
   Ref = make_ref(),
   allocator ! {request, self(), Ref, N},
   receive {granted, Ref, Granted} -> Granted end.

% release ‘Resources’ to ‘allocator’
release(Resources) ->
   Ref = make_ref(),
   allocator ! {release, self(), Ref, Resources},
   receive {released, Ref} -> released end.
```

Producer-consumer

Producer-consumer: the problem – recap

```
-module(buffer).
% initialize buffer with size ‘Bound’
init_buffer(Bound) -> todo.
% put ‘Item’ in ‘Buffer’; block if full
put(Buffer, Item) -> todo.
% get item from ‘Buffer’; block if empty
get(Buffer) -> todo.
```

Producer-consumer problem: implement `buffer` such that:

- producers and consumer access the buffer atomically
- consumers block when the buffer is empty
- producers block when the buffer is full (bounded buffer variant)

Producers and consumers

Producers and consumers continuously and asynchronously access the buffer, which must guarantee proper synchronization.

producer$_n$

```
producer(Buffer) ->
  % create a new item
  Item = produce(),
  buffer:put(Buffer, Item),
  producer(Buffer).
```

consumer$_m$

```
consumer(Buffer) ->
  Item = buffer:get(Buffer),
  % do something with ‘item’
  consume(Item),
  consumer(Buffer).
```

Note that atomic access is not an issue with processes: a single sequential process will actively modify the content of the buffer in response to messages sent by other processes.

Buffer process: bounded buffer

The buffer process keeps track of the items stored in the buffer:

- when a process asks to get one item and the buffer is not empty, the buffer sends an item message to the process, and removes the item just taken from the buffer list
- when a process asks to put one item and the buffer is not full, the buffer sends a done message to the process, and adds the item just sent to the buffer list
- as in the allocator example, requests that cannot be satisfied (get with empty buffer, and put with full buffer) implicitly queue in the allocator's mailbox; they will be served as soon as it is possible

We implement the buffer's event loop as a server function:

```
buffer(Content, Count, Bound)
```

where `Content` is the list of `Count` available resources, and `Bound` is the buffer's size.

The server function `buffer`: handling requests

```
buffer(Content, Count, Bound) ->
    receive
      % serve gets when buffer not empty
      {get, From, Ref} when Count > 0 ->
          [First|Rest] = Content,       % match first item
          From ! {item, Ref, First},    % send it out
          buffer(Rest, Count-1, Bound); % remove it from buffer
      % serve puts when buffer not full
      {put, From, Ref, Item} when Count < Bound ->
          From ! {done, Ref},                           % send ack
          buffer(Content ++ [Item], Count+1, Bound) % add item to end
    end.
```

The server function `buffer`: handling requests

`Content` managed as FIFO queue

```
buffer(Content, Count, Bound) ->
  receive
    % serve gets when buffer not empty
    {get, From, Ref} when Count > 0 ->
      [First|Rest] = Content,          % match first item
      From ! {item, Ref, First},       % send it out
      buffer(Rest, Count-1, Bound);    % remove it from buffer
    % serve puts when buffer not full
    {put, From, Ref, Item} when Count < Bound ->
      From ! {done, Ref},                                % send ack
      buffer(Content ++ [Item], Count+1, Bound)          % add item to end
  end.
```

Starvation is not possible: when the buffer is neither full nor empty, requests are served in the order they arrive in the mailbox. If the buffer fills up, `put` is disabled; after finitely many `get`s are served the buffer is no longer full, which disables `get`, thus allowing `put` to be served. Similarly, `put` activates `get` when the buffer is empty.

Buffer process: unbounded buffer

In an unbounded buffer, the condition `Count < Bound` always holds:

```
% serve puts
{put, From, Ref, Item} when Count < Bound ->
  % ...
```

Instead of removing the condition (as well as all the occurrences of `Bound`), we can take advantage of Erlang's order between numbers and atoms (every number is less than any atom): setting `Bound` to `infinity` ensures that `Count < Bound` will always evaluate to `true`. This way, we can use the very same implementation both in the bounded and in the unbounded case.

The functions `get` and `put`

The functions `get` and `put` exchange messages with the process with pid `Buffer`; they are used so that synchronizing processes do not need to know about the format of exchanged messages.

```
% get item from ‘Buffer’; block if empty
get(Buffer) ->
   Ref = make_ref(),
   Buffer ! {get, self(), Ref},
   receive {item, Ref, Item} -> Item end.

% put ‘Item’ in ‘Buffer’; block if full
put(Buffer, Item) ->
   Ref = make_ref(),
   Buffer ! {put, self(), Ref, Item},
   receive {done, Ref} -> done end.
```

Readers-writers

Readers-writers: the problem – recap

```
-module(board).
init(Name) -> todo.          % register board with ‘Name’
begin_read(Board) -> todo.   % get read access to ‘Board’
end_read(Board) -> todo.     % release read access to ‘Board’
begin_write(Board) -> todo.  % get write access to ‘Board’
end_write(Board) -> todo.    % release write access to ‘Board’
```

Readers-writers problem: implement `board` such that:

- multiple reader can operate concurrently
- each writer has exclusive access

Invariant: $\#\text{WRITERS} = 0 \lor (\#\text{WRITERS} = 1 \land \#\text{READERS} = 0)$

Other properties that a good solution should have:

- support an arbitrary number of readers and writers
- no starvation of readers or writers

Readers and writers

Readers and writers continuously and asynchronously try to access the board, which must guarantee proper synchronization.

---

reader$_n$

```
reader(Board) ->
  board:begin_read(Board),
  % read messages
  board:end_read(Board),
  reader(Board).
```

writer$_m$

```
writer(Board) ->
  board:begin_write(Board),
  % write messages
  board:end_write(Board),
  writer(Board).
```

Board process – first version

A first solution to the readers-writers problem can extend the idea behind the allocator: serve requests when possible, and let other requests queue in the mailbox. The board process keeps track of the number of readers and writers that are active on the board:

- when a new request to begin reading arrives and no writer is active, the board sends an OK to read message to the requester, and increases the count of readers;
- when a new request to begin writing arrives and no readers or writers are active, the board sends an OK to write message to the requester, and increases the count of writers;
- conversely, when notifications to end read or end write arrive, the board decreases the count of readers or writers;
- requests that cannot be served implicitly queue in the board's mailbox; they will be served as soon as the board is freed

The server function `board_RoW` – first version

```
% ‘Readers’ active readers and ‘Writers’ active writers
board_RoW(Readers, Writers) ->
 receive
  {begin_read, From, Ref} when Writers =:= 0 ->
     From ! {ok_to_read, Ref},
     board_RoW(Readers+1, Writers);
  {begin_write, From, Ref} when (Writers =:= 0) and (Readers =:= 0) ->
     From ! {ok_to_write, Ref},
     board_RoW(Readers, Writers+1);
  {end_read, From, Ref} ->
     From ! {ok, Ref},
     board_RoW(Readers-1, Writers);
  {end_write, From, Ref} ->
     From ! {ok, Ref},
     board_RoW(Readers, Writers-1)
 end.
```

Readers-writers: the first version prioritizes readers

In `board_RoW`, the "waiting conditions" follow directly from the invariant; thus, the resulting solution is correct in that it ensures mutual exclusion according to the readers-writers invariant.

However, it gives priority to readers over writers:

- new reading requests get served without waiting as long as a reader is active
- writing requests waiting in the mailbox have to wait until the last reader sends an `end_read` message
- as long as reading requests keep arriving and queuing in the mailbox, the waiting writing requests will never execute

Exchanging the order of clauses in the **`receive`** does not solve the problem (nor does it give priority to writers over readers): the scenario where readers starve writers can still happen because the condition for writing is stronger than the condition for reading, and writers cannot maintain their condition without the cooperation of readers.

Readers-writers: towards a fair solution

We could achieve fairness by replicating the pattern behind the solution with monitors:

- the board keeps track of the lists of pending read and write requests
- read requests are served as long as there are no active writers and no pending write requests
- notifications to end write let in one pending read request, or one waiting write request if there are no reading requests

This approach works, but it is quite cumbersome to implement with message passing. The main issue is that it requires a duplication of the information that is already implicit in the mailbox queue, which complicates ensuring that messages are processed exactly once.

Readers-writers: fair solution

We implement a fair solution where the board can be in one of two macro states:

empty: there are neither active readers nor active writers
readers: there are some active readers and no active writers

When the board is in macro state empty:

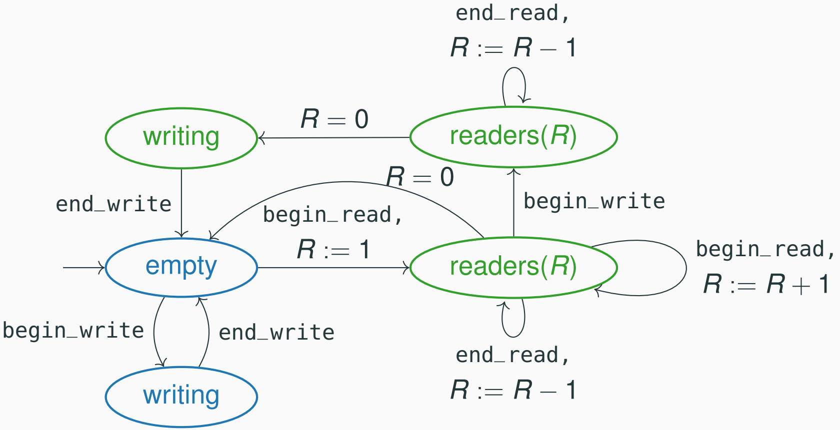
- read requests are served immediately, then the board switches to macro state readers
- write requests are served immediately and synchronously: the board waits until writing ends, then the board is empty again

When the board is in macro state readers:

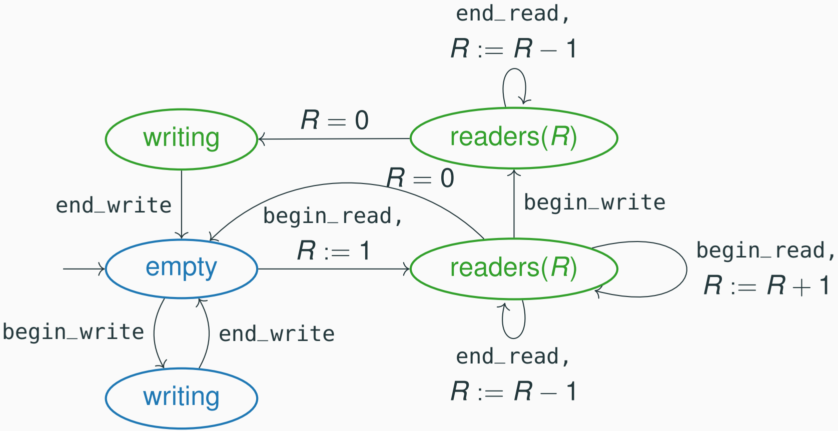
- read requests are served immediately, and the macro state remains readers
- write requests are served as soon as possible: the board waits until all reading ends, then the writing request is served synchronously, and then the board is empty again

Readers-writers: fair solution (cont'd)

This state/transition diagram formalizes the solution illustrated informally above. The partitioning of states in the diagram according to their color corresponds to the macro states empty and readers.



Readers-writers: fair solution (cont'd)



By inspecting the diagram we can understand that it guarantees fairness provided outgoing transitions from the same state have the same priority (they are served in arrival order). The solution in Erlang implements the behavior of this diagram, using two server functions `empty_board` and `readers_board` which call each other.

The server function `empty_board`

```
% board with no readers and no writers
empty_board() ->
   receive
     % serve read request
     {begin_read, From, Ref} ->
        From ! {ok_to_read, Ref}, % notify reader
        readers_board(1);         % board has one reader
     % serve write request synchronously
     {begin_write, From, Ref} ->
        From ! {ok_to_write, Ref}, % notify writer
        receive                    % wait for writer to finish
          {end_write, _From, _Ref} ->
            empty_board()          % board is empty again
        end
   end.
```

The server function `readers_board`: serving write requests

```
% board with no readers (and no writers)
readers_board(0) -> empty_board();

% board with ‘Readers’ active readers (and no writers)
readers_board(Readers) ->
   receive
     % serve write request
     {begin_write, From, Ref} ->
       % wait until all ‘Readers’ have finished
       [receive {end_read, _From, _Ref} -> end_read end
          || _ <- lists:seq(1, Readers)],
       From ! {ok_to_write, Ref}, % notify writer
       receive                    % wait for writer to finish
         {end_write, _From, _Ref} -> empty_board()
       end;                       % board is empty again
```

The server function `readers_board`: serving read requests

Now the order of clauses in the **`receive`** does not matter: requests are processed in the mailbox order because none of the three clauses (`begin_read`, `end_read`, and `begin_write`) has a condition stronger than the others.

```
readers_board(Readers) ->
  receive
    % serve write requests: previous slide...

    % serve read request
    {begin_read, From, Ref} ->
      From ! {ok_to_read, Ref},  % notify reader
      readers_board(Readers+1);  % board has one more reader
    % serve end read
    {end_read, _From, _Ref} ->
      readers_board(Readers-1)   % board has one less reader
  end.
```

The functions `begin_read`, `end_read`, `begin_write`, and `end_write`

As usual, the functions `begin_read`, `end_read`, `begin_write`, and `end_write` exchange messages with the board server process with pid `Board`; they are used so that synchronizing processes do not need to know about the format of exchanged messages.

For example:

```
% get read access to ‘Board’
begin_read(Board) ->
   Ref = make_ref(),
   Board ! {begin_read, self(), Ref},
   receive
     {ok_to_read, Ref} -> ok_to_read
   end.
```

Note that the behavior of the board process changes over time, but the pid `Board` stays the same.

Board initialization

Initializing a board consists of spawning a process running `empty_board`.

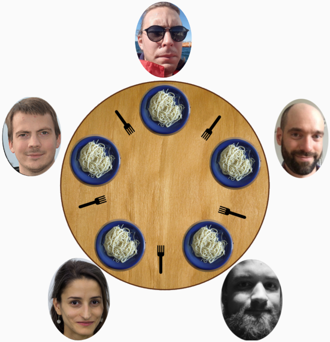
```
% initialize empty board and register with 'Name'
init(Name) ->
  register(Name, spawn(fun empty_board/0)).
```

After initialization, `Name` can be used to access the board.

Dining philosophers

Dining philosophers: the problem – recap

```
-module(philosophers).
% set up table of ‘N’ philosophers
init(N) -> todo.
% philosopher picks up ‘Fork’
get_fork(Fork) -> todo.
% philosopher releases ‘Fork’
put_fork(Fork) -> todo.
```



Dining philosophers problem: implement `philosophers` such that:

- forks are held exclusively by one philosopher at a time
- each philosopher only accesses adjacent forks
- no philosopher starves

Philosophers with waiter

We could replicate solutions based on locking; for example, setting up a server for each pair of forks, which grants access to both forks atomically to the first philosopher that sends a request.

Instead, let's explore an approach that is more congenial to message passing. A waiter process supervises access to the table. Each philosopher asks the waiter for permission to sit before picking up both forks, and notifies the waiter after putting down both forks.

As long as the waiter allows strictly fewer philosophers than the total number of forks to sit around the table at the same time, deadlock and starvation are avoided.

The waiter's interface consists of two functions:

```
% ask ‘Waiter’ to be seated; may wait
sit(Waiter) -> todo.
% ask ‘Waiter’ to leave
leave(Waiter) -> todo.
```

Philosophers

Philosophers continuously alternate between thinking and eating, while coordinating with the waiter.

---

philosopher_k

```
% Forks: fork#{left, right} of fork pids
% Waiter: waiter process
philosopher(Forks, Waiter) ->
   think(),
   sit(Waiter),                 % ask to be seated
   get_fork(Forks#forks.left),  % pick up left fork
   get_fork(Forks#forks.right), % pick up right fork
   eat(),
   put_fork(Forks#forks.left),  % put down left fork
   put_fork(Forks#forks.right), % put down right fork
   leave(Waiter),               % notify leaving
   philosopher(Forks, Waiter).
```

Waiter process

The waiter process keeps track of how many philosophers are eating at the table:

- when a philosopher asks to be seated and the table is not full, the waiter sends an `ok_to_sit` message to the philosopher and increases the count of eating philosophers
- when a philosopher notifies leaving, the waiter sends an `ok_to_leave` message to the philosopher and decreases the count of eating philosophers
- requests to sit that arrive when the table is full queue in the waiter's mailbox; they will be served as soon as a seat frees up

We implement the waiter's event loop as a server function:

```
waiter(Eating, Seats)
```

where `Eating` philosophers are sitting and eating, out of a total of `Seats` available seats (`Seats` is the number of seats that can be occupied at the same time).

The server function `waiter`

```erlang
waiter(Eating, Seats) ->
  receive
      % serve as long as seats are available
    {sit, From, Ref} when Eating < Seats ->
      io:format("~p eating (~p at table)~n", [From, Eating+1]),
      From ! {ok_to_sit, Ref},
      waiter(Eating+1, Seats);   % one more eating
      % can leave at any time
    {leave, From, Ref} ->
      io:format("~p leaving (~p at table)~n", [From, Eating-1]),
      From ! {ok_to_leave, Ref},
      waiter(Eating-1, Seats)    % one less eating
  end.
```

Printing the table’s state at every change is for debugging purposes.

The functions `sit` and `leave`

As usual, the functions `sit` and `leave` hide the format of messages exchanged between philosophers and waiter.

```
% ask 'Waiter' to be seated; may wait
sit(Waiter) ->
   Ref = make_ref(),
   Waiter ! {sit, self(), Ref},
   receive {ok_to_sit, Ref} -> ok end.

% ask 'Waiter' to leave
leave(Waiter) ->
   Ref = make_ref(),
   Waiter ! {leave, self(), Ref},
   receive {ok_to_leave, Ref} -> ok end.
```

The fork processes and functions

Each fork has a fork process which keeps track of whether the fork is free (on the table) or held by a philosopher.

The server function for a fork can be in two states according to whether the fork is held or not.

```
% a fork not held by anyone
fork() ->
   receive
     {get, From, Ref} ->
       From ! {ack, Ref},
       fork(From) % fork held
   end.
```

```
% a fork held by Owner
fork(Owner) ->
   receive
     {put, Owner, _Ref} ->
       fork()  % fork not held
   end.
```

For simplicity, `put` requests do not get an acknowledgment; they just take effect immediately.

The functions `get_fork` and `put_fork`

The structure of `get_fork` and `put_fork` should be obvious by now:

```
% pick up ‘Fork’; block until available
get_fork(Fork) ->
    Ref = make_ref(),
    Fork ! {get, self(), Ref},
    receive {ack, Ref} -> ack end.

% put down ‘Fork’
put_fork(Fork) ->
    Ref = make_ref(),
    Fork ! {put, self(), Ref}.
```

Table initialization

Initializing a table consists of spawning the processes running `waiter`, `fork`, and `philosopher`, as well as connecting each philosopher to their pair of forks.

```
% set up table of ‘N’ philosophers
init(N) ->
     % spawn waiter process
   Waiter = spawn(fun () -> waiter(0, N-1) end),
   Ids = lists:seq(1,N), % [1, 2, ..., N]
     % spawn fork processes
   Forks = [spawn(fun fork/0) || _ <- Ids],
     % spawn philosopher processes
   [spawn(fun () ->
          Left = lists:nth(I, Forks),
          Right = lists:nth(1+(I rem N), Forks), % 1-based indexes
          philosopher(#forks{left=Left, right=Right},
                      Waiter) end)    || I <- Ids].
```

at most `N-1` eating philosophers at once

These slides’ license