# COSC441 Concurrent Programming

## Patterns of Concurrency and Parallelism

Richard A. O'Keefe

September 19, 2017

# Outline

- map/2 again
- pmap/2
- problems with pmap/2
- spawn_link/2
- throttling
- supervision
- behaviours

# map/2 again

% return $[F(X_1), \ldots, F(X_n)]$

```
map(F, [X|Xs]) →
    [F(X) | map(F, Xs)];
map(_, []) →
    [].
```

# pmap/2

```
% return [F(X_1), ..., F(X_n)]
% computing the results in separate threads.

pmap(F, [X|Xs]) ->
    S = self(),
    P = spawn(fun () -> S ! {self(), F(X)} end),
    Ys = pmap(F, Xs)
    receive {P, Y} -> [Y|Ys] end;
pmap(_, []) ->
    [].
```

# Problems with pmap/2: process dictionary

- Erlang processes each own a "dictionary"
- erase(Key) $\rightarrow$ Value | undefined
- get(Key) $\rightarrow$ Value | undefined
- put(Key, Value) $\rightarrow$ OldValue | undefined
- Keys and Values may be *any* Erlang term
- spawning does not copy the process dictionary; if $F$ uses or changes it switching from map/2 to pmap/2 will break.

# Problems with pmap/2: waiting forever

- If one of the new processes crashes, pmap/2 will wait forever.
- We can make the calling process and all the child processes crash if any of them does by using spawn_link instead of spawn.
- Erlang has two ways to notice the death of a process: *monitoring* (one-way link) and *linking* (two-way link).

# monitor/2

- monitor(process, Pid) $\rightarrow$ Ref
- The calling process receives $\{$'DOWN', Ref, process, Pid, Info$\}$ where
- Info = noproc (process didn't exist), noconnection (connection to process lost), or the exit reason if Pid crashed.
- This message is sent once and only once.
- There is a timing window: create process, it crashes, call monitor, get noproc Info instead of exit reason. *Always beware of timing windows!*

# Using monitor/2

$P$ = spawn(**fun** () →
    **receive** _ → ok **end**,
    $S$ ! {self(), $F(X)$}
**end**),
$R$ = erlang:monitor(process, $P$),
$P$ ! go,
...

# Using monitor/2

```
receive
    {'DOWN', R, process, P, Info} ->
        handle failure
;   {P, Y} ->
        [Y|Ys]
end
```

Oh dear. To handle failure, we want to kill the other child processes.

# Avoiding the timing window

- There is a combination function spawn_monitor/1 that *atomically* spawns a new process and monitors it. The result is a {Pid,Ref} pair.
- There is a function spawn(Node, Fun) that starts a new process on another node.
- There happens to be no spawn_monitor/2.
- So the "spawn; wait; twiddle; resume" pattern is still worth knowing, also in other languages.

# Linking

- Links are bidirectional.
- link(Pid) links Pid and the caller.
- unlink(Pid) removes a link.
- If two processes are *linked*, and one of them crashes, the other will receive an exit signal.
- If A sends B an exit signal with reason R,
  - If B is trapping exits, it receives a message {'EXIT',A,R}.
  - If B is not trapping exits, B exits with reason R.

# Trapping exits

- process_flag(trap_exit, true) makes the caller trap exits (get 'EXIT' messages)
- process_flag(trap_exit, false) makes the caller die in response to exit signals
- either way the old value is returned.
- "application" code normally doesn't do this, but library code like pmap/2 might.

# Timing windows again

- It is possible for the new process to crash before you can link to it.
- We can use the "spawn; wait; twiddle; resume" pattern.
- spawn_link/1 *atomically* creates a new process and links to it.
- spawn_link(Node, Fun) *does exist*.
- "application" code *does* use spawn_link.

# Throttling

- Creating an Erlang process is *cheap*
- but it isn't *free*.
- If you are after speedup and have $N$ cores, $2N$ processes might be useful, but $N^2$ will not be.
- If you need $N^2$ processes for *concurrent structure*, fine.
- pmap($F$, $Xs$) creates length($Xs$) processes.
- How do you create fewer?

```
pmap4(F, Xs) ->
    S = self(),
    Pids = map(fun (Part) ->
        spawn_link(fun ->
            S ! {self(), map(F, Part)} end)
        end, split4(Xs)),
    join4(map(fun (Pid) ->
            receive {Pid,Ans} -> Ans end
        end, Pids)).
```

# What do we have?

- A *design pattern* called Master-Worker.
- Two *reusable components*.
- Prefer components to patterns.

# It looks sequential, but. . .

foldl($F$, $A$, $[X|Xs]$) $\rightarrow$
    foldl($F$, $F(A, X)$, $Xs$);
foldl(_, $A$, []) $\rightarrow$
    $A$.

This looks sequential. For arbitrary $F$, it is. But for *associative* $F$, that is, $F(X, F(Y, Z)) = F(F(X, Y), Z)$, it doesn't have to be.

```
% folding over a binary tree.

fold(F, L, {fork,X,Y}) ->
    S = self(),
    P = spawn(fun () -> S ! {self(),fold(F,L,Y)} end),
    U = fold(F, L, X),
    receive {P,V} -> F(U, V) end;
fold(_, L, {leaf,X}) ->
    L(X).
```

# Algorithmic Skeletons

- "Algorithmic Skeletons (*alias* Parallelism Patterns) take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by combining the basic ones."
- Not limited to parallelism: concurrency too.
- Look at Wikipedia in class.
- *Not* a closed set!

# Supervision

- A *supervisor* is a process that is monitoring a number of *child* processes.
- If the supervisor dies, the children pointless and should be killed.
- If a child dies, the supervisor should make a decision on what to do, *e.g.*,
  - kill everything
  - kill everything and restart
  - restart the dead process
  - make do without ...

# Behaviours

- Erlang modules embodying concurrent patterns
- “OTP Design Principles User’s Guide”
- handle startup, termination, reload
- as well as core pattern
- Obviously parameterised, but how?

# Behaviour parameters

- Provide the name of a *callback module*
- Each behaviour has a set of callback functions
- The callback module must implement them
- Using a module instead of a function or functions means the module can be hot-loaded.