Fault Recovery

- If two processes are “linked” ... when one dies the other will die too

- Purpose: prevent failure from causing incomplete computation

- If one process “monitors” another ... the monitor will be notified when the monitored process dies

- Purpose: fault recovery – monitor can restart monitored process

Linking/Monitoring

- How to start linked/monitored processes:

```
Pid = spawn_link( module, function, arg list )
```

```
{ Pid, Ref } = spawn_monitor( module, function, arg list )
```

- When monitorED process dies, monitorING process receives this tuple as a message:

```
{ 'DOWN', Ref, process, Pid, Reason }
```

- `'DOWN'` and `process` are atoms; i.e., literal values

- `Ref` is variable; has same value as returned by `spawn_monitor/3`

- `Pid` and `Reason` are variables

Exit With a Reason

- Use `exit/1`: `exit(Reason)`
- Examples:

```
exit(gottago)

exit('goodbye cruel world')
```

Erlang Approach to Errors

1. Separate "regular case" code from "error handling" code

2. When error occurs, crash immediately & let error handling code clean up

- Claimed advantages:

  - In other languages, regular case code & error handling code is intertwined: messy to read; hard to write both correctly at same time

  - Fail-fast helps errors not to propagate

Erlang Approach to Scheduling

- Typically one OS thread per core

- Each thread includes "scheduler" logic

- Erlang schedulers cooperate to balance load across cores by migrating Erlang processes as needed

- Each Erlang process starts with some number of "reductions" (usually 1000 or 2000)

- A reduction is approx. one function call

Reductions

- Many operations have a cost measured in "reductions"

  - call BIF (built-in function)
  - function call
  - sending messages
  - garbage collection
  - ... many other things that take time

- When process reduction count reaches zero, it is descheduled

- Erlang provides "soft real time" operation

Tail Call

- A **tail call** is a function call performed as a function's final action

- Recursive example:

```
first_thread_loop(Partner) ->
    {ok, [ X1, X2 ]} = io:fread("two integers> ", "~d~d"),
    Partner ! {X1, X2},
    receive
        Sum ->
            io:fwrite("~p + ~p = ~p~n", [ X1, X2, Sum ]),
            first_thread_loop(Partner)
    end.
```

- Call to `first_thread_loop` is final action

- There is NOTHING else on its line

Example

- Second clause does NOT have a tail call:

```
list_len( [] ) -> 0 ;

list_len( [ _ | Tail ] ) ->
    1 + list_len(Tail) .
```

- "`1 + list_len(Tail)`" is NOT a tail call because after `list_len(Tail)` returns a final action remains (add 1) before function can return

Tail Recursion, I

- Tail calls important in functional languages

- In imperative languages, common to write `while/for` statements that loop thousands or millions of times

- In functional languages, do to the same means thousands of millions of recursive function calls

- Huge stacks seemingly needed to track thousands/millions of active calls ($O(N)$)

- **Tail recursion** is: how to perform thousands/millions of active tail calls using a single stack frame! ($O(1)$)

Tail Recursion, II

- **Tail recursion** aka
Tail call elimination aka
Tail call optimization means:

1. Programmer writes his/her code so that ...
2. Compiler can perform important stack "elimination" optimization

- Term "tail recursion" sometimes refers to style of programming (#1), sometimes to the compiler optimization (#2), sometimes to the combination

Tail Recursion, III

- Tail recursion so important to list-oriented functional languages that standard document of some languages guarantee that compiler will perform the optimization

- Some Erlang doc pages state that programmers "must" write tail calls so the optimization can be applied

How to Write Tail Recursive Functions, I

Example (in the more familiar Java):

```
// factorial written in the "usual" recursive way
private static int f(int N) {
    if (N == 1)
        return 1;
    else
        // multiplication done AFTER function returns!!!
        return N * f(N-1);
}

// written to use tail recursion
private static int f2(int N, int accumulator) {
    if (N == 1)
        return accumulator;
    else
        // multiplication done BEFORE function called!!!
        return f2(N-1, N*accumulator);
}

// API that looks normal & hides the tail recursion
private static int f3(int N) {
    return f2(N, 1);
}
```


How to Write Tail Recursive Functions, II

- To convert non-tail-recursive function to tail-recursive, "`accumulator`" variable is common technique:

  - Accumulate partial result in that variable
  - Pass partial result down sequence of recursive calls

- Works well if result easy to express in a single variable