分布式容错系统

为什么要做这个系统?

系统要满足什么需求?

- 并发(concurrency)
- 软实时(soft real-time)
- 分布式(distributed)
- 容错性(fault tolerance)
- 持续运行(continuous operation)

并发

- 并发：同时应对（dealing with）多件事情的能力
- 并行：同时做（doing）多件事情的能力

软实时

- 硬实时系统有一个刚性的、不可改变的时间限制，它不允许任何超出时限的错误。超时错误会带来损害甚至导致系统失败、或者导致系统不能实现它的预期目标。例如火箭发射，核反应堆。
- 软实时只能提供统计意义上的实时。有的应用要求系统在 95% 的情况下都会确保在规定的时间内完成某个动作，而不一定要求100%，可以容忍偶然的超时错误。失败造成的后果并不严重，例如在网络中仅仅是轻微地降低了系统的吞吐量。

分布式

- 性能
- 可靠性
- 可扩展性

容错

程序中一定存在 bug，硬件也可能出现问题，容错性就是要做到合理地处理这个故障（直接崩溃 or 重启 等等）

持续运行（热升级）

系统要设计成可以持续运行许多年，意味着在系统不停下来的情况下进行软件和硬件的维护。

操作系统提供的设施

- 线程
- 进程
- 锁

- 进程是具有独立功能的程序在某个数据集合上的一次运行，是系统进行资源分配的单位。
- 线程是操作系统能够运行运算调度的单位。被包含在进程中，是进程中实际运行的单位。

进程线程最大的区别线程运行在共享的内存地址空间里，共享堆的资源。而进程运行在不同的内存地址空间里，不共享堆的资源。可以把进程理解为一组共享地址空间的线程的集合。

Linux 调度的是什么?

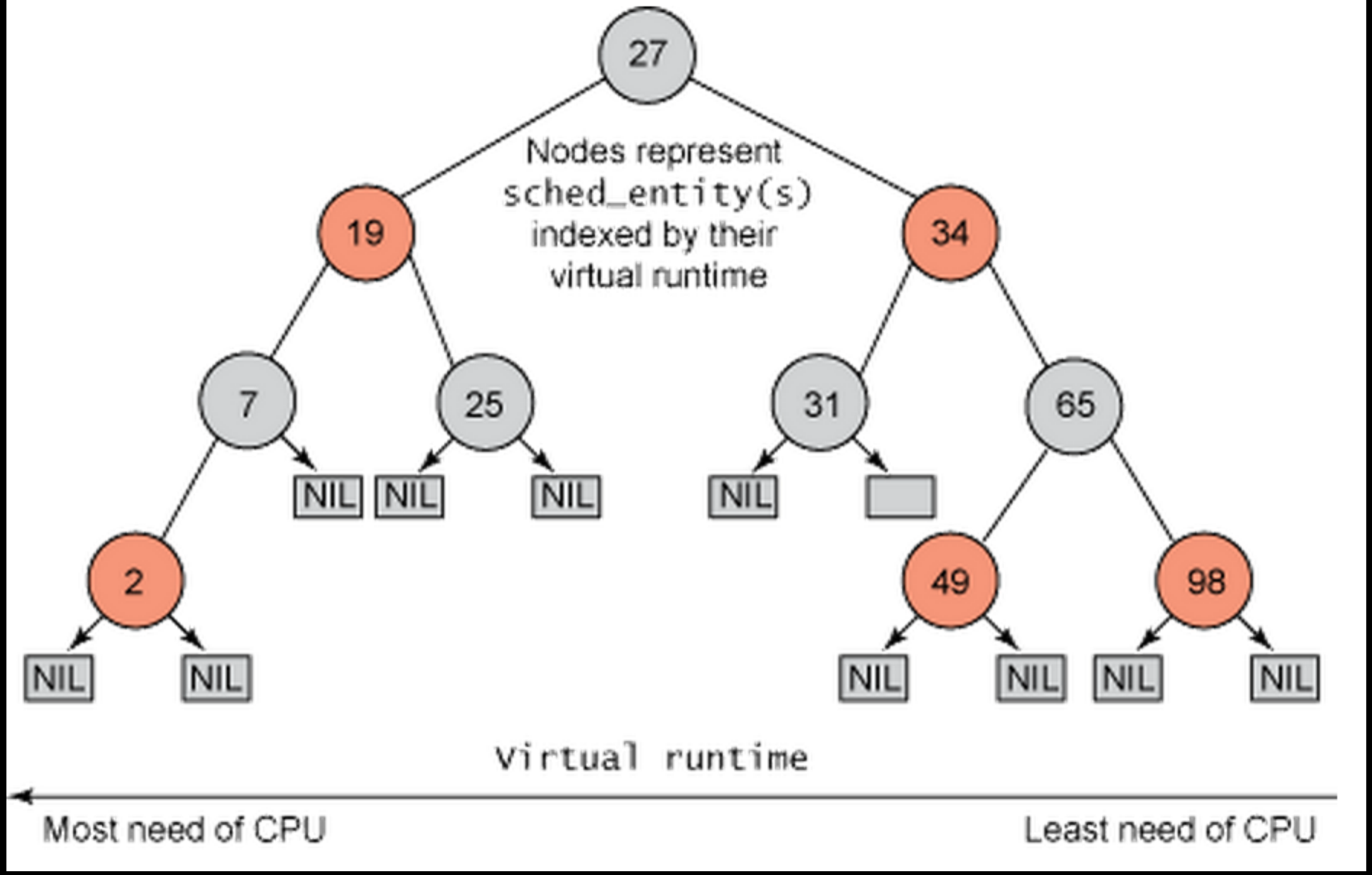
线程? 进程?

task

- 线程
- 单线程的进程

调度策略

Completely Fair Scheduler (CFS)


27
Nodes represent
sched_entity(s)
indexed by their
virtual runtime
19
34
7
25
31
65
NIL
NIL
NIL
NIL
2
49
98
NIL
NIL
NIL
NIL
NIL
NIL
Virtual runtime
Most need of CPU
Least need of CPU

- 任务存储在以时间为顺序的红黑树中，对处理器需求最多的任务 （最低虚拟运行时）存储在树的左侧，处理器需求最少的任务（最高虚拟运行时）存储在树的右侧。
- 调度器然后选取红黑树最左端的节点调度为下一个以便保持公平性。任务通过将其运行时间添加到虚拟运行时， 说明其占用 CPU 的时间

每个进程的ideal_runtime值的大小与它的权重weight成正比
每个进程每次获得CPU使用权，最多执行它对应的ideal_runtime 这样长的时间

进程	A	B	C	D
weight	1	2	3	4
ideal_runtime(ms)	2	4	6	8
runtime(ms)	0	0	0	0
vruntime	0	0	0	0
按vruntime进行排序	A B C D			

vruntime的行走速度和权重值成反比，设定权重权为1的A进程的
vruntime和实际runtime行走速度相同。A先执行，它执行了2ms

进程	A	B	C	D
weight	1	2	3	4
ideal_runtime(ms)	2	4	6	8
runtime(ms)	2	0	0	0
vruntime	2	0	0	0
按vruntime进行排序	B C D A			

B的vruntime值最小,选择B运行，假设B运行了3ms

进程	A	B	C	D
weight	1	2	3	4
ideal_runtime(ms)	2	4	6	8
runtime(ms)	2	3	0	0
vruntime	2	1.5	0	0
按vruntime进行排序	C D B A			

C的vruntime值最小，选择C运行，假设C运行了3ms

进程	A	B	C	D
weight	1	2	3	4
ideal_runtime(ms)	2	4	6	8
runtime(ms)	2	3	3	0
vruntime	2	1.5	1	0
按vruntime进行排序	D C B A			

D的vruntime值最小，选择D运行，假设D运行了8ms

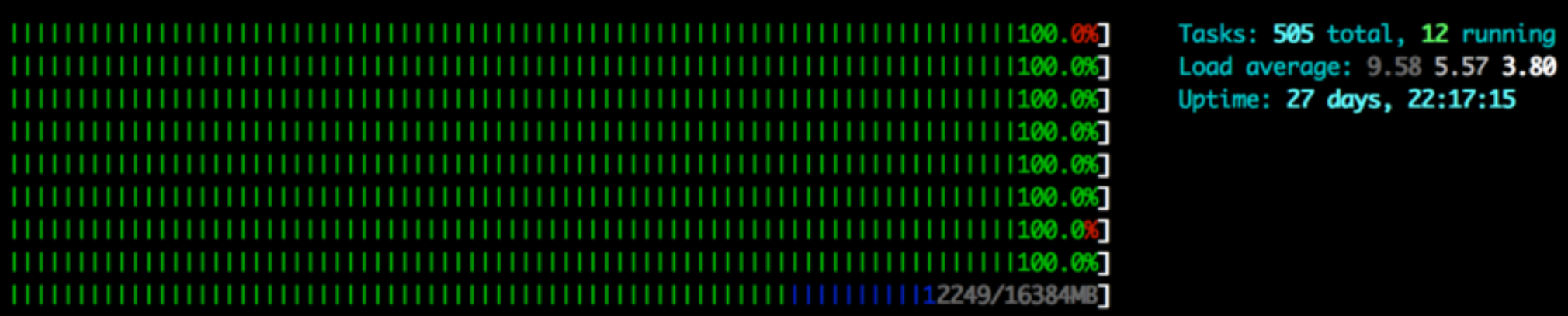
进程	A	B	C	D
weight	1	2	3	4
ideal_runtime(ms)	2	4	6	8
runtime(ms)	2	3	3	8
vruntime	2	1.5	1	2
按vruntime进行排序	C B A D			

进程D运行的时间等于它的ideal_runtime，调度器被激活，重新选择一个进程运行，接着C进程被选中执行，进程C最多可以运行6ms，那么接下来进程C可以连续运行6ms

进程	A	B	C	D
weight	1	2	3	4
ideal_runtime(ms)	2	4	6	8
runtime(ms)	2	3	9	8
vruntime	2	1.5	3	2
按vruntime进行排序	B A D C			

- 每个进程每次获得CPU使用权最多可以执行与它对应的ideal_runtime那么长的时间
- 如果每个进程每次获得CPU使用权时它都执行了它对应的ideal_runtime那么长的时间，整个就绪队列的顺序保持不变。
- 如果某个进程某几次获得CPU使用权时运行的时间小于它ideal_time指定的时间（即它被调度时没有享用完它可以享用的最大时间），按照vruntime进行排序的机制会使得它尽量排在队列的前面，让它尽快把没有享用完的CPU时间弥补起来
- 每个进程在就绪队列中等地的时间不会超过 period，一个 period 内每个进程都有运行的机会

多核测试


100.0%
100.0%
100.0%
100.0%
100.0%
100.0%
100.0%
12249/16384MB
Tasks: 505 total, 12 running
Load average: 9.58 5.57 3.80
Uptime: 27 days, 22:17:15

锁（效率和安全）

- 哲学家就餐
- https://hg.python.org/cpython/file/3.4/Lib/functools.py#l474

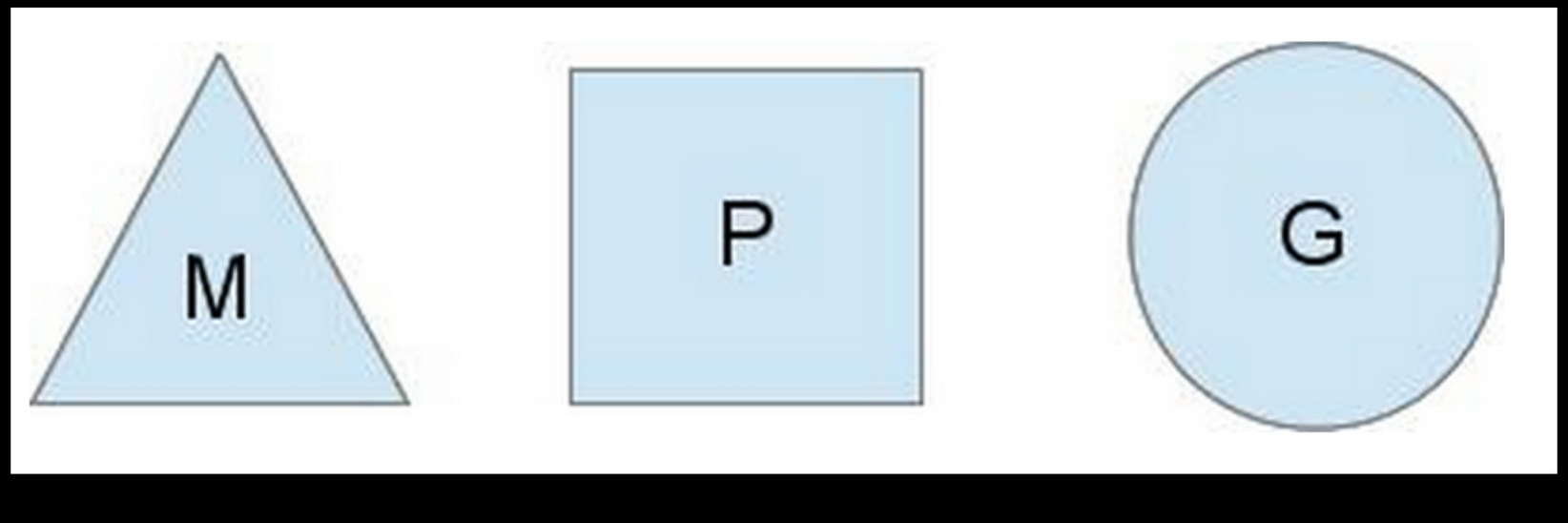
- 不能高并发（多进程资源）
- 支持软实时（调度公平）
- 分布式困难 （进程通信困难）
- 容错性差，没有监督机制
- 不能热升级

Python😢

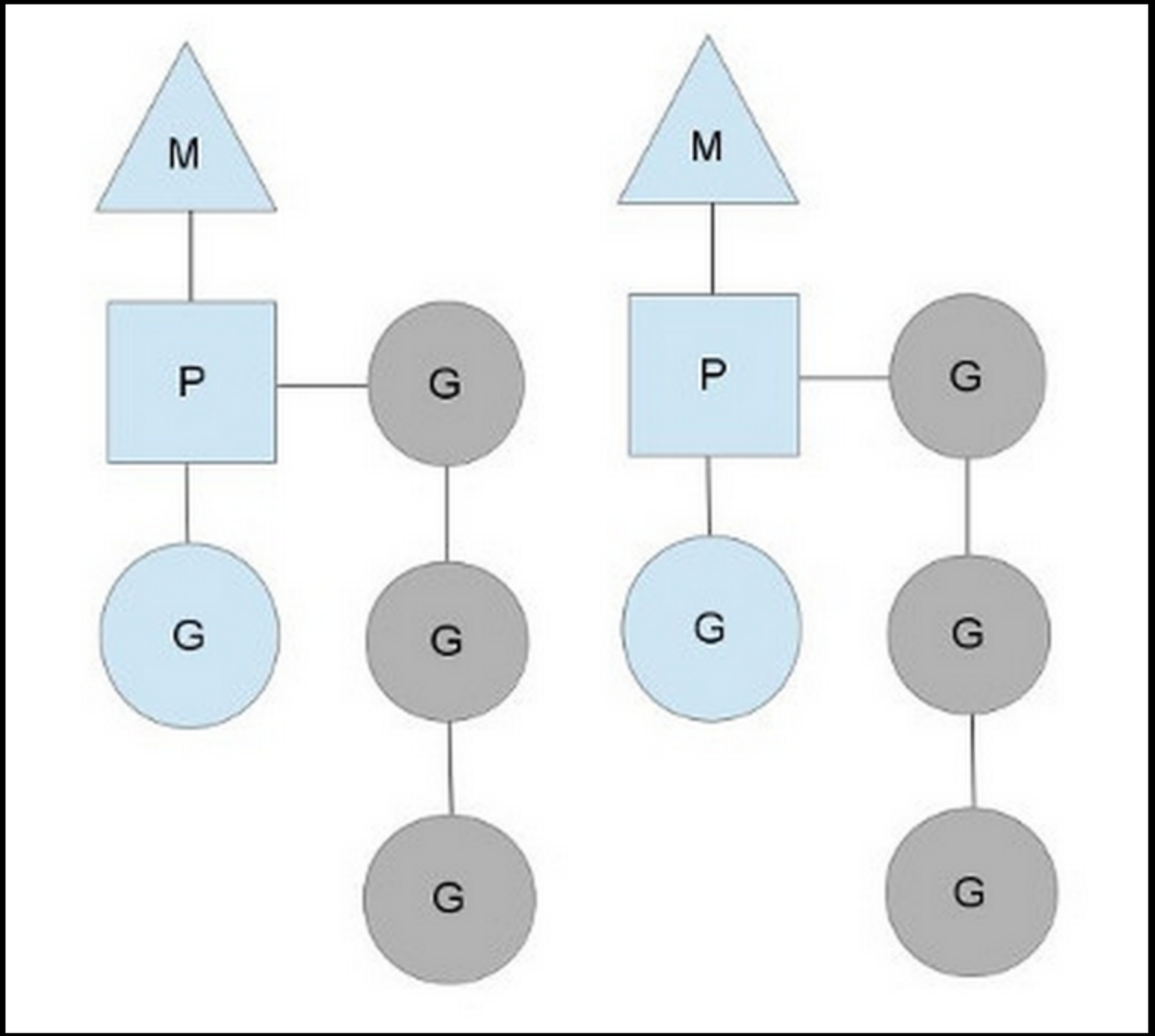
- 高并发（包含 cpu）
- 不支持软实时（多核调度）
- 分布式困难 （跨机器进程通信困难）
- 容错性差，没有监督机制
- 热升级

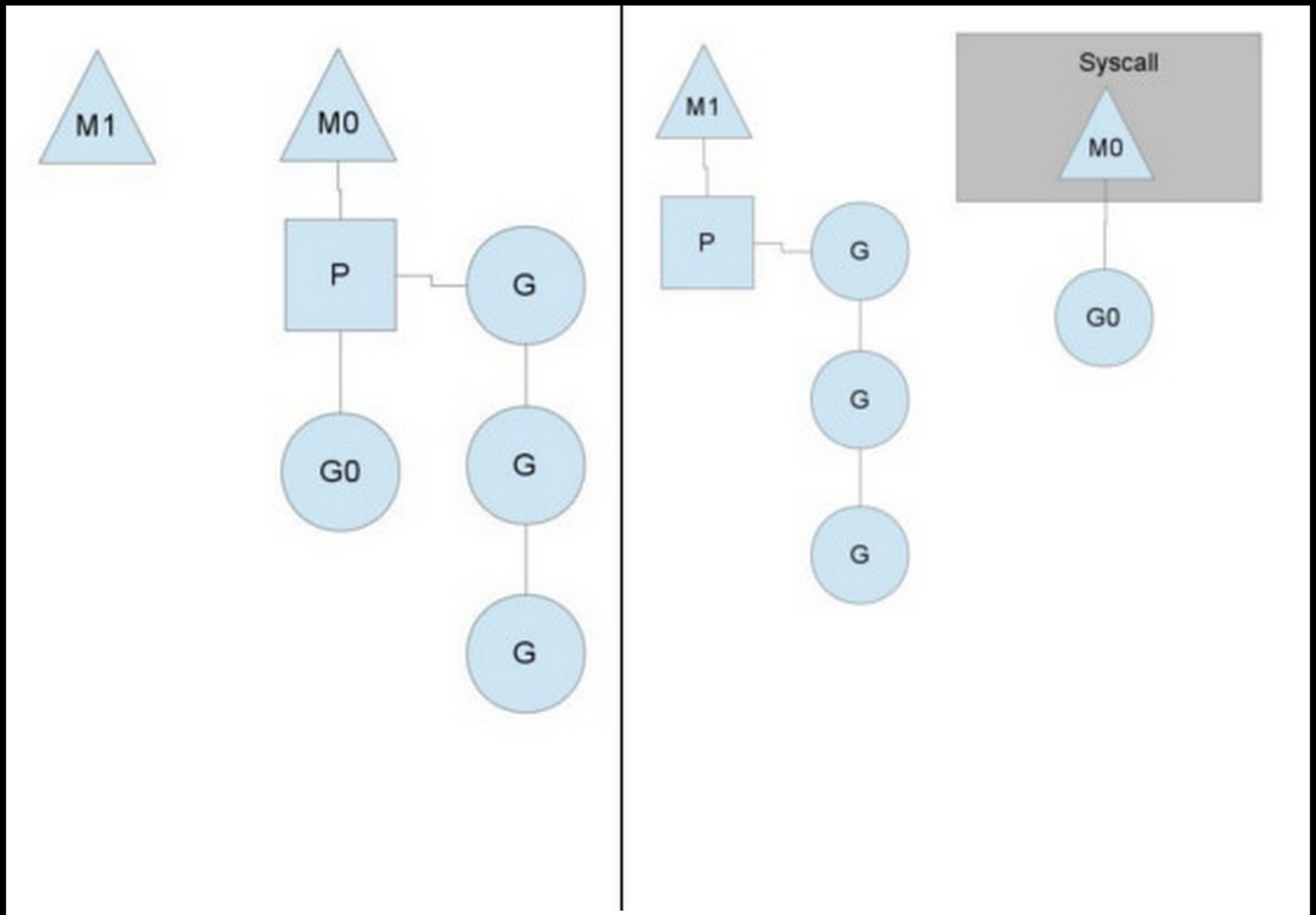
Golang

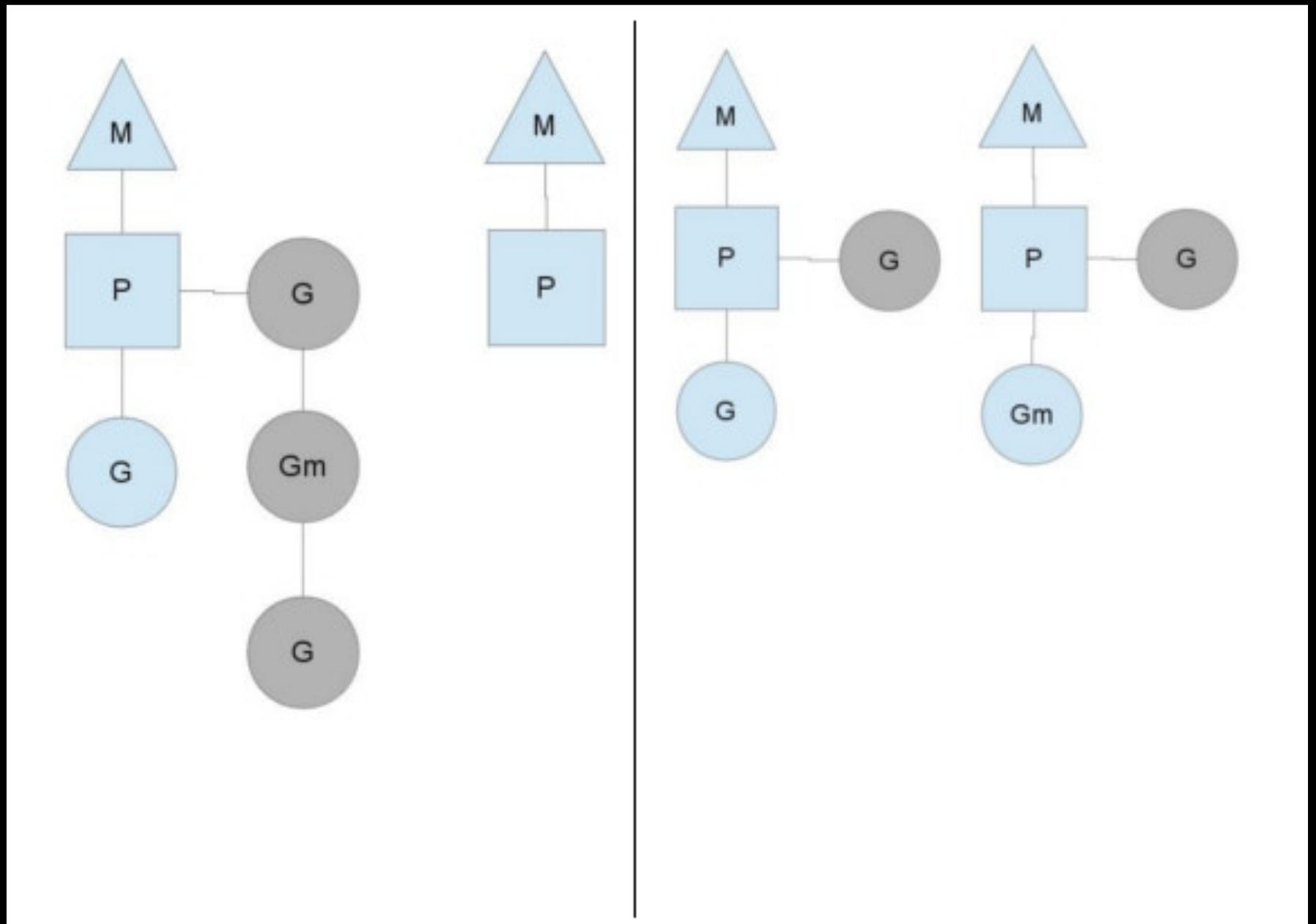
Goroutine



M: 内核 OS 线程
G: 代表一个 goroutine，有自己的栈
P: 抽象处理器，允许多少个 goroutine 并行


M
P
G
G
G
M
P
G
G
G


M1
M0
P
G
G0
G
G
M1
P
G
G
G
Syscall
M0
G0


M
P
G
G
Gm
G
M
P
M
P
G
G
M
P
G
Gm

```go
import (
        "fmt"
        //"runtime"
        "time"
)

func say() {
        for {
        }
}

func hi() {
        for {
                fmt.Println("hi")
        }
}

func main() {
        // runtime.GOMAXPROCS(4)
        go say()
        go hi()
        time.Sleep(100000000000000000)
}
~
~
```

猜下运行结果，去掉注释呢?

容错

服务器在同时处理多个请求，启动了很多 goroutine，有一个 goroutine 出现了不可预知的错误，会发生什么?

```go
package main

import (
        "fmt"
        "net/http"
        "time"
)

func say() {
        time.Sleep(20)
        resp, _ := http.Get("http://example.co/")
        fmt.Println(resp.Body)
        //panic("no value for $USER")
}

func hi() {
        for {
                fmt.Println("hi")
        }
}

func main() {
        go say()
        go hi()
        time.Sleep(100000000000000000)
}
~
```

- 高并发，利用多核
- 软实时（调度不公平，GC）
- 分布式 （跨机器通信困难）
- 容错性 （资源不够隔离，监督设施不完善，没有身份标志）
- 热升级

Erlang/Elixir

- 一切皆进程。
- 进程强隔离。
- 进程不共享资源，无锁。
- 进程生成销毁是轻量操作。
- 消息传递是进程交互的唯一方式。
- 每个进程有其独有的名字。
- 知道进程名字，就可以发消息，否则不可以。
- 进程要么好好跑，要么死翘翘。
- 错误处理异地化。

Process

```
-module(basic).
-export([loop/0]).

loop() ->
    receive
        {rectangle, Width, Ht} ->
            io:format("Area of rectangle is ~p~n",[Width * Ht]),
            loop();
        {square, Side} ->
            io:format("Area of square is ~p~n", [Side * Side]),
            loop()
end.
~
```

Erlang 解决的是多线程的并发模型带来的锁，它的解决办法是提供一种新的用户态进程并发模型。除此以外，任何不是因多线程的原因所产生的锁（以及相关问题），Erlang 的并发模型都没有解决。

Erlang Scheduler

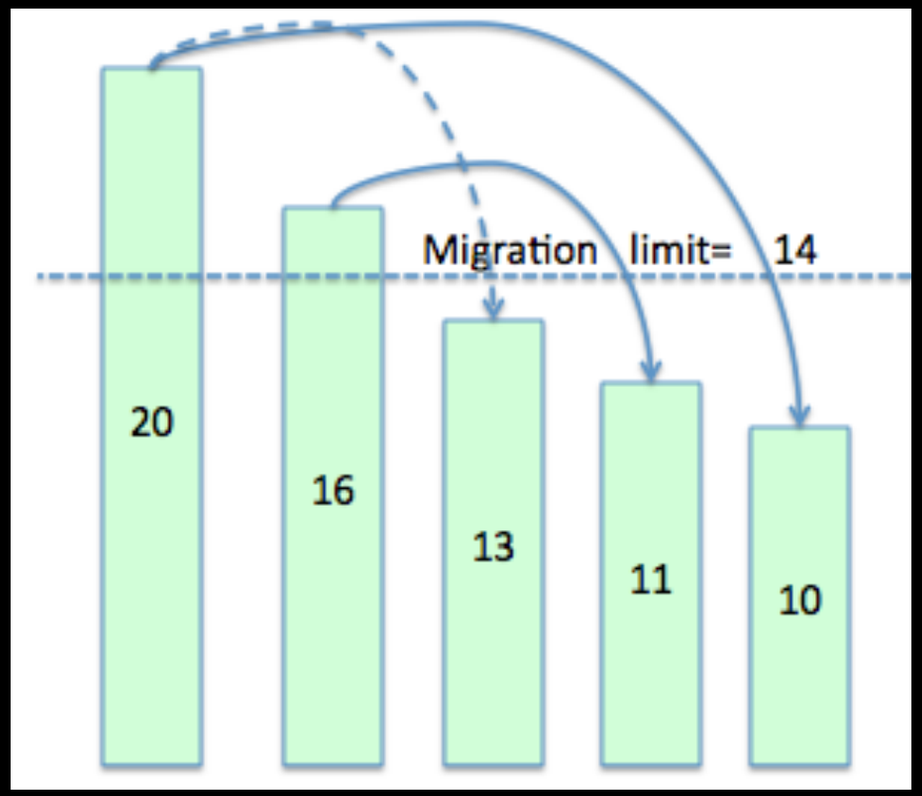
- 进程调度运行在用户空间： Erlang进程不同于操作系统进程，Erlang的进程调度也跟操作系统完全没有关系，是由Erlang虚拟机来完成的

- 调度是抢占式的：每一个进程在创建时，都会分配一个固定数目的reduction（R15B中，这个数量默认值是2000），每一次操作（函数调用），reduction就会减少，当这个数量减少到0时或者进程没有匹配的消息时，抢占就会发生（无视优先级）

- 每个进程公平的使用CPU：每个进程分配相同数量的reduction，可以保证进程可以公平的（不是相等的）使用CPU资源

- 调度器保证软实时性：Erlang中的进程有优先级，调度器可以保证在下一次调度发生时，高优先级的进程可以优先得到执行

- 多调度器、多运行队列，利用多核



- 计算各队列长度与“Migtation limit”差值
- 找到差值中正最大和负最小的队列，记录一个从前者到后者进行任务迁移路径，以达到二者都接近“Migtation limit”
- 重复步骤1，直到达到负载均衡

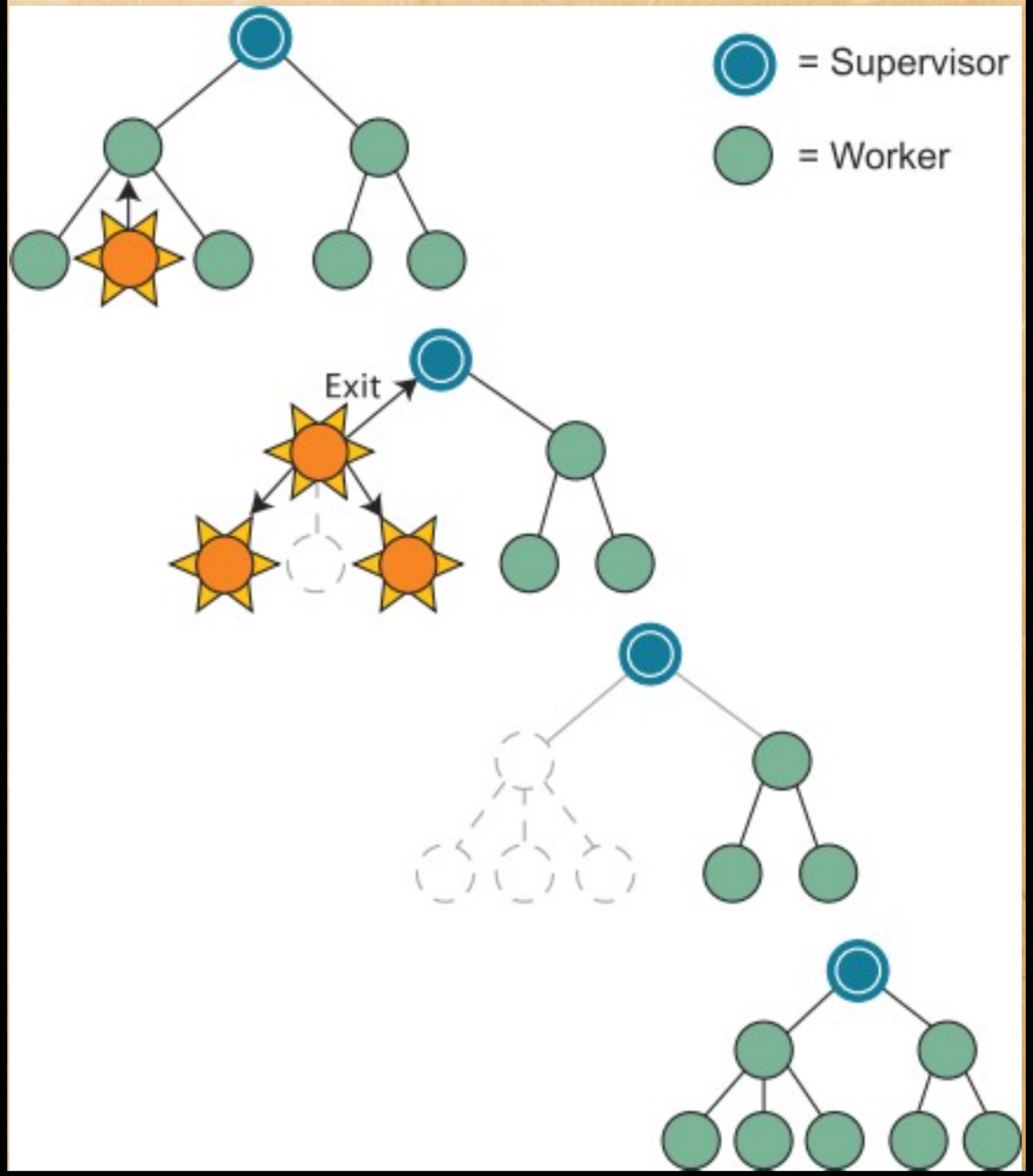
为什么 Erlang 可以这么调度 Go 却不可以?

Erlang 调度器之所以能够实现优先级轮转调度，主要是得益于其基于虚拟机的执行方式：由于每条Erlang指令都需要经过 BEAM 解释执行，因此 process 的运行完全处于BEAM的监控之下，BEAM可以方便的完成对进程的切换。与之相对，由于 Go 的 Goroutine 与 RUNTIME 都是 Native 执行的，其在执行上的地位是平等的，RUNTIME 没有能力切换一个执行中的 Goroutine，除非其自己调出或调用 RUNTIME 功能，因而只能实现协作调度。

容错性

任其崩溃，强力监督

- 粒度可控，不同的模块可以用不同的处理方式。
- 容易诊断，如果在错误发生后第一时间进行通知，就能得到确切的现场信息，便于调试。在错误发生后继续运行经常会导致更多错误发生，让调试变得更加困难。
- 简化系统架构，把应用程序和错误恢复当成两个独立的问题来思考，而不是一个交叉的问题。


= Supervisor
= Worker
Exit

```erlang
-module(fault).

%% http://www.ituring.com.cn/article/69952

-export([on_exit/2, start/0, log_error/2, keep_alive/2]).


log_error(Pid, Why) ->
    io:format("~p died with: ~p~n", [Pid, Why]).


start() ->
    receive
        X -> list_to_atom(X)
    end.


on_exit(Pid, F) ->
    spawn(fun() ->
            Ref = monitor(process, Pid),
            receive
                {'DOWN', Ref, process, Pid, Why} ->
                    F(Pid, Why)
            end
        end).

keep_alive(Name, Fun) ->
    register(Name, Pid = spawn(Fun)),
    on_exit(Pid, fun(_Why) -> keep_alive(Name, Fun) end).
~
```

分布式

```erlang
-module(kvs).
-export([start/0, store/2, lookup/1]).


%% http://www.ituring.com.cn/article/69993

start() -> register(kvs, spawn(fun() -> loop() end)).

store(Key, Value) -> rpc({store, Key, Value}).

lookup(Key) -> rpc({lookup, Key}).

rpc(Q) ->
    kvs ! {self(), Q},
    receive
        {kvs, Reply} ->
            Reply
    end.

loop() ->
    receive
        {From, {store, Key, Value}} ->
            put(Key, {ok, Value}),
            From ! {kvs, true},
            loop();
        {From, {lookup, Key}} ->
            From ! {kvs, get(Key)},
            loop()
    end.
```

热升级

- 将修改后的代码重新编译并加载
- 外部调用

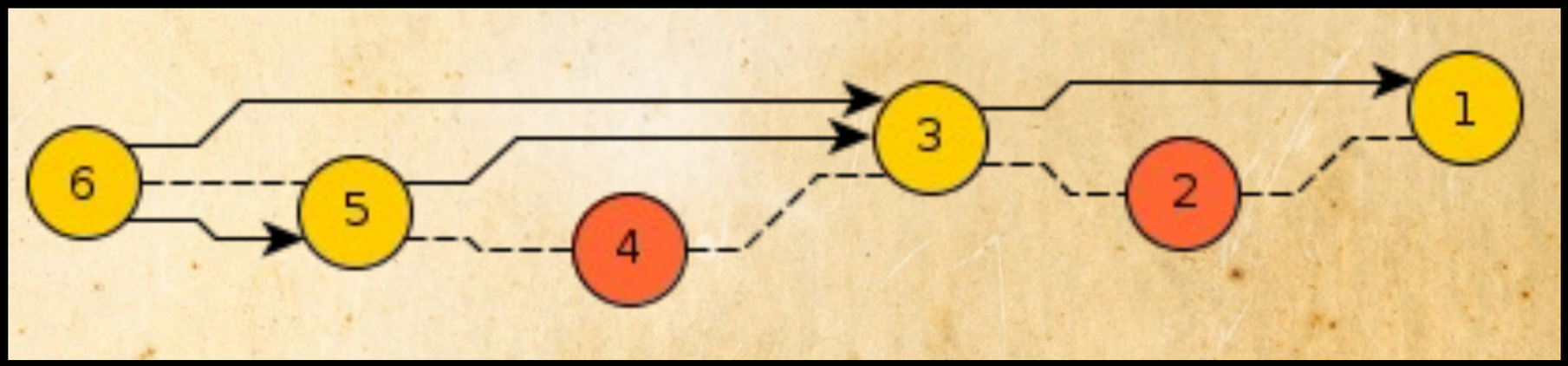
```erlang
-module(update_example).

-export([start/0, run/0]).

%% http://wudaijun.com/2015/04/erlang-hotcode/


f() ->
    io:format("this is old code~n").

run() ->
    f(),
    timer:sleep(5000),
    ?MODULE:run().

start() ->
    spawn(fun() -> run() end).
```

函数式

- 不是出于学院派的喜好
- 函数式天生适合并发系统

优势

- 变量只读。
- 并发状态无需加锁。
- 便于热代码替换。
- 简化 GC 实现。



由于变量不可修改,老旧对象不会持有年轻对象的引用,内存中的对象引用关系必定是一张从年轻对象指向老旧对象的 DAG

OTP

OTP is set of Erlang libraries and design principles providing middleware to develop these systems. It includes its own distributed database, applications to interface towards other languages, debugging and release handling tools.

- 生产效率——运用OTP可在短时间内交付产品级的系统。
- 稳定性——基于OTP的代码可以更集中于逻辑，并避免重新实现那些容易出错而每个实际系统又都必备的基础功能，如进程管理、服务器、状态机等。
- 监督——这是由框架提供的一套简便的监视和控制运行时系统的机制，既有自动化方式，也有图形用户界面方式。
- 可升级——框架为处理代码升级提供了一套系统化的模式。
- 可靠的代码库——OTP的代码坚如磐石并全部经过严格的实战检验。

No Silver Bullet

Q & A

- 硬实时和软实时区别
- Does linux schedule a process or a thread?
- Linux - threads and process scheduling priorities
- How Linux handles threads and process scheduling
- Linux进程调度(1)：CFS调度器的设计框架

- Linux - Threads and Process
- FINDING THE LINUX THREAD ID FROM WITHIN PYTHON USING CTYPES
- Linux 2.6 Completely Fair Scheduler 内幕
- Linux cfs调度器_理论模型
- 从几个问题开始理解CFS调度器

- 理解Load Average做好压力测试
- Erlang 热更新
- 如何看待七牛 CEO 许式伟开源的 Cerl?
- Erlang并发机制 –进程调度
- Erlang & Go 的并发调度浅析
- 《面对软件错误构建可靠的分布式系统》
- 《Erlang 程序设计》
- 《Erlang OTP 并发编程实战》