PROCESSES, THREADS AND ~~UNIX~~ RUBY

Oleg Ivanov
@morhekil
http://morhekil.net
http://speakmy.name

who can name all of these 3 guys?

who's that? well, irrelevant

Dennis Ritchie - C Programming Language,
Unix operating system together with...

Ken Thompson - B Language, Unix OS, Regular Expressions, Go Language

Unix is user-friendly;
it's just picky about who its friends are.

POSIX

- **P**ortable **O**perating **S**ystem **I**nterface
- Fully POSIX-compliant: OS X, QNX, Solaris
- Mostly POSIX-compliant: GNU/Linux, *BSD
- Ruby implements lots of POSIX functionality, and often - with exactly the same API (Kernel#fork for fork(2), Process.wait for wait(2), etc)

BASIC PROCESS ATTRIBUTES

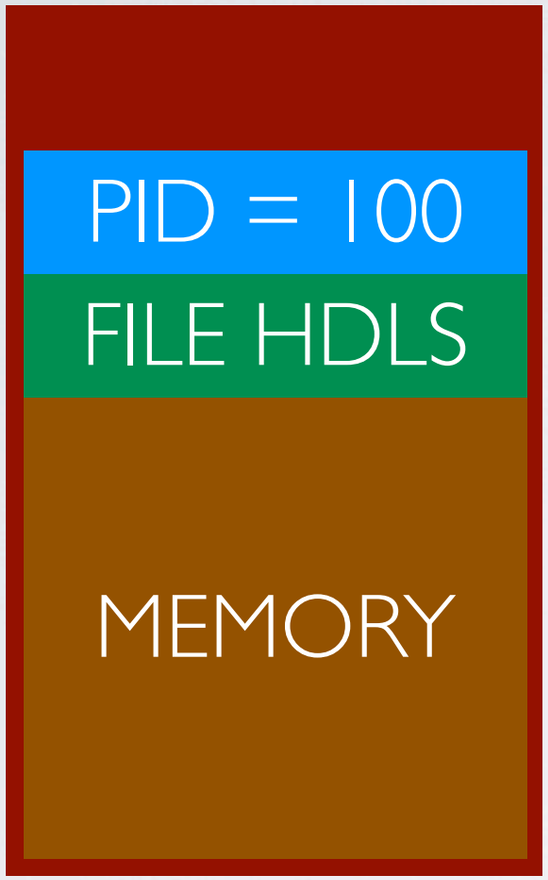
```
# ps -eo pid,ppid,ni,user,thcount,args | grep -E 'PID|unicorn'
  PID  PPID  NI USER     THCNT COMMAND
11883     1   0 www          2 unicorn_rails master -c config/unicorn.rb
11977 11883   0 www          3 unicorn_rails worker[0] -c config/unicorn.rb
11982 11883   0 www          3 unicorn_rails worker[1] -c config/unicorn.rb
11987 11883   0 www          3 unicorn_rails worker[2] -c config/unicorn.rb
11993 11883   0 www          3 unicorn_rails worker[3] -c config/unicorn.rb
```

- Process ID
- Parent Process ID
- Nice level
- Owner
- Thread count
- Name

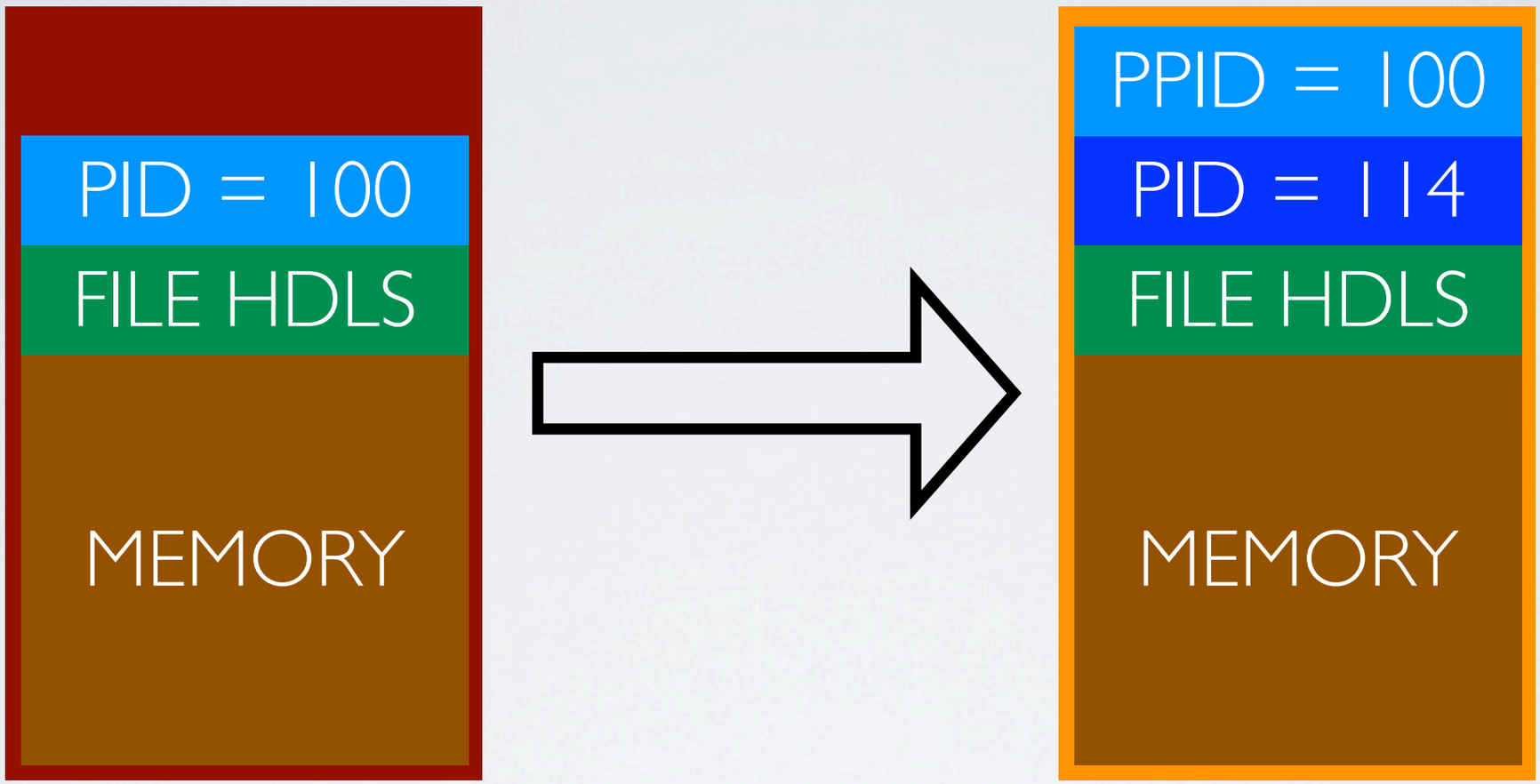
FORKING

- fork(2) - "create child process" system call
- parent's PID is child's PPID
- child receives a **copy** of parent's memory
- child receives parent's open file descriptors (files, sockets, etc)
- child's memory is independent of parent's memory

FORKING



FORKING



FORKING IN RUBY: IF/ELSE

```
if fork
  puts "executing if block"
else
  puts "executing else block"
end
```

FORKING IN RUBY: IF/ELSE

```ruby
if fork
  puts "executing if block"
else
  puts "executing else block"
end
```

```
% ruby tmp.rb
executing if block
executing else block
```

FORKING IN RUBY: IF/ELSE

```
if fork
  puts "executing if block"
else
  puts "executing else block"
end
```

```
% ruby tmp.rb
executing if block
executing else block
```

Confusing? Let's rewrite it a bit

FORKING IN RUBY: IF/ELSE

```ruby
puts "I am #{Process.pid}"

if fork
  puts "executing if block in #{Process.pid}"
else
  puts "executing else block #{Process.pid}"
end
```

FORKING IN RUBY: IF/ELSE

```ruby
puts "I am #{Process.pid}"

if fork
  puts "executing if block in #{Process.pid}"
else
  puts "executing else block #{Process.pid}"
end
```

```
% ruby tmp.rb
I am 18290
executing if block in 18290
executing else block in 18291
```

FORKING IN RUBY: IF/ELSE

```ruby
puts "I am #{Process.pid}"

if fork
  puts "executing if block in #{Process.pid}"
else
  puts "executing else block #{Process.pid}"
end
```

```
% ruby tmp.rb
I am 18290
executing if block in 18290
executing else block in 18291
```

Kernel#fork returns:

- in **child** process - **nil**
- in **parent** process - **pid** of the child process

FORKING IN RUBY: BLOCK

```ruby
fork do
  # child process code
  puts "I am a child"
end

# parent process code
puts "I am the parent"
```

- **child** process exits at the end of the block
- **parent** process skips the block

MEMORY MANAGEMENT

- when child exits, its memory is destroyed
- use case: fork child processes to run memory-hungry code

WHY?

- Ruby is bad at releasing memory back to the system
- so ruby processes grow, but don't shrink

COPY-ON-WRITE

- when child is forked, its memory is not really copied until it's been written to
- so we don't have to copy the whole memory at once

COPY-ON-WRITE

- when child is forked, its memory is not really copied until it's been written to
- so we don't have to copy the whole memory at once
- **but only if we're using Ruby 2.0!**

RUBY GC: 2.0 VS 1.9

GC path for bitmap marking landed in 2.0:

https://github.com/ruby/ruby/commit/50675fdba1125a841ed494cb98737c97bd748900#L3L1641

```
 gc_mark(rb_objspace_t *objspace, VALUE ptr, int lev)
 {
     register RVALUE *obj;
+    register uintptr_t *bits;

     obj = RANY(ptr);
     if (rb_special_const_p(ptr)) return; /* special const not marked */
     if (obj->as.basic.flags == 0) return;       /* free cell */
-    if (obj->as.basic.flags & FL_MARK) return;  /* already marked */
-    obj->as.basic.flags |= FL_MARK;
+    bits = GET_HEAP_BITMAP(ptr);
+    if (MARKED_IN_BITMAP(bits, ptr)) return;  /* already marked */
+    MARK_IN_BITMAP(bits, ptr);
     objspace->heap.live_num++;
```

In 1.9:

- GC marks stored in the objects
- memory writes in every object on every GC run
- memory gets copied by OS

In 2.0:

- GC marks stored in external bitmap
- no memory writes in the objects
- no copies of unchanged memory

BASIC COMMUNICATION

- Exit status
- Process name

PROCESS EXIT CODE

- returned when process finished executing
- `0` usually indicates success, `1` and other - error
- but it is merely a matter of interpretation
- “errors” can be interpreted as program-specific responses

PROCESS EXIT CODES: RUBY

```
Kernel#exit(status = true)
Kernel#exit!(status = false)
```

```ruby
code = ARGV.first.to_i
exit code
```

```
% ruby tmp.rb 0
% echo $?
0
% ruby tmp.rb 5
% echo $?
5
```

PROCESS EXIT CODES: RUBY

```
Kernel#exit(status = true)
Kernel#exit!(status = false)
```

```ruby
code = ARGV.first.to_i
exit code
```

```
% ruby tmp.rb 0
% echo $?
0
% ruby tmp.rb 5
% echo $?
5
```

basic shell logic:

```
% ruby tmp.rb 5 && echo "Yep"
% ruby tmp.rb 1 || echo "Nope"
Nope
% ruby tmp.rb 0 && echo "Yep"
Yep
```

0 = success
1 = failure

} convention

PROCESS NAME

- can be changed in runtime
- can be controlled by the process itself
- is often overlooked as a simple, but powerful communication media

PROCESS NAME: RUBY

```ruby
1.upto(10) do |n|
  $0 = "zomg process: #{n*10}%"
  sleep 2
end
```

```
% watch "ps ax | grep zomg"

Every 2.0s: ps ax | grep zomg

43121 s003  S+     0:00.02 zomg process: 40%
```

PROCESS NAME: RUBY

```ruby
1.upto(10) do |n|
  $0 = "zomg process: #{n*10}%"
  sleep 2
end
```

```
% watch "ps ax | grep zomg"

Every 2.0s: ps ax | grep zomg

43121 s003  S+     0:00.02 zomg process: 40%
```

Process name can communicate any status:

- task progress
- request being executed
- job being run
- number of workers
- etc

CASE STUDY 1.1: BRIDGE

web workers:

```
class ApplicationController < ActionController::Base
  around_filter :manage_memory

  def manage_memory
    old_0 = $0
    begin
      $0 = "rails:#{request.method} #{controller_name}##{action_name}#{'.xhr' if request.xhr?}"
      yield
    ensure
      $0 = old_0
    end
  end
end
```

- mostly informational in this case
- but sometimes can be useful to cross-reference with other stuck processes (e.g. long db queries)

CASE STUDY 1.2: BRIDGE

stable master / horses:

```ruby
# lib/stable_master.rb
class StableMaster
  def run_jobs
    old_0 = $0

    while @running
      ActiveRecord::Base.verify_active_connections!

      if(@running && (job_id = (read_pipe.readline rescue nil)))
        job = Job.find(job_id.chomp.to_i)
        $0 = "stablemaster: #{job.class}##{job.id}"
      end

    end
    $0 = old_0
  end
end
```

- similar status messages for stable master and idle workers
- can easily identify and kill troublesome jobs
- look around in stable_master.rb for more

CASE STUDY 1.2: BRIDGE

stable master / horses:

```
# lib/stable_master.rb
class StableMaster
  def run_jobs
    old_0 = $0

    while @running
      ActiveRecord::Base.verify_active_connections!

      if(@running && (job_id = (read_pipe.readline rescue nil)))
        job = Job.find(job_id.chomp.to_i)
        $0 = "stablemaster: #{job.class}##{job.id}"
      end

    end
    $0 = old_0
  end
end
```

ProTip: AR tends to lose db connections, so reconnect in child processes

CASE STUDY 2: RESQUE

Redis based processing queue

- forks a child per job to contain memory bloat
- communicates runtime status via process names

CASE STUDY 2: RESQUE

Redis based processing queue

- forks a child per job to contain memory bloat
- communicates runtime status via process names

```
def process_job(job, &block)
  # ...
  @child = fork(job) do
    reconnect
    run_hook :after_fork, job
    unregister_signal_handlers
    perform(job, &block)
    exit! unless options[:run_at_exit_hooks]
  end

  if @child
    wait_for_child
    job.fail(DirtyExit.new($?.to_s)) if $?.signaled?
  # ...
  end
  done_working
end
```

CASE STUDY 2: RESQUE

Redis based processing queue

- forks a child per job to contain memory bloat
- communicates runtime status via process names

```
  def process_job(job, &block)
    # ...
    @child = fork(job) do
      reconnect
      run_hook :after_fork, job
      unregister_signal_handlers
      perform(job, &block)
      exit! unless options[:run_at_exit_hooks]
    end

    if @child
      wait_for_child
      job.fail(DirtyExit.new($?.to_s)) if $?.signaled?
    # ...
    end
    done_working
  end
```

forking a child to run the job

CASE STUDY 2: RESQUE

Redis based processing queue

- forks a child per job to contain memory bloat
- communicates runtime status via process names

```
  def process_job(job, &block)
    # ...
    @child = fork(job) do
      reconnect
      run_hook :after_fork, job
      unregister_signal_handlers
      perform(job, &block)
      exit! unless options[:run_at_exit_hooks]
    end

    if @child
      wait_for_child
      job.fail(DirtyExit.new($?.to_s)) if $?.signaled?
    # ...
    end
    done_working
  end
```

reconnecting to the server

CASE STUDY 2: RESQUE

Redis based processing queue

- forks a child per job to contain memory bloat
- communicates runtime status via process names

```
def process_job(job, &block)
  # ...
  @child = fork(job) do
    reconnect
    run_hook :after_fork, job
    unregister_signal_handlers
    perform(job, &block)
    exit! unless options[:run_at_exit_hooks]
  end

  if @child
    wait_for_child
    job.fail(DirtyExit.new($?.to_s)) if $?.signaled?
  # ...
  end
  done_working
end
```

and executing the job

CASE STUDY 2: RESQUE

Redis based processing queue

- forks a child per job to contain memory bloat
- communicates runtime status via process names

```
def process_job(job, &block)
  # ...
  @child = fork(job) do
    reconnect
    run_hook :after_fork, job
    unregister_signal_handlers
    perform(job, &block)
    exit! unless options[:run_at_exit_hooks]
  end

  if @child
    wait_for_child
    job.fail(DirtyExit.new($?.to_s)) if $?.signaled?
  # ...
  end
  done_working
end
```

parent process waits for child to finish

CASE STUDY 2: RESQUE

Redis based processing queue

- forks a child per job to contain memory bloat
- communicates runtime status via process names

```
def process_job(job, &block)
  # ...
  @child = fork(job) do
    reconnect
    run_hook :after_fork, job
    unregister_signal_handlers
    perform(job, &block)
    exit! unless options[:run_at_exit_hooks]
  end

  if @child
    wait_for_child
    job.fail(DirtyExit.new($?.to_s)) if $?.signaled?
  # ...
  end
  done_working
end
```

and marks the job as failed if it's been killed with a signal

CASE STUDY 2: RESQUE

Redis based processing queue

- forks a child per job to contain memory bloat
- communicates runtime status via process names

```
    perform(job, &block)
```

```
  # Processes a given job in the child.
  def perform(job)
    procline "Processing #{job.queue} since #{Time.now.to_i} [#{job.payload_class_name}]"
    begin
      run_hook :before_perform, job
      job.perform
      run_hook :after_perform, job
    rescue Object => e
      job.fail(e)
      failed!
    else
      Resque.logger.info "done: #{job.inspect}"
    ensure
      yield job if block_given?
    end
  end
```

and executing the job

CASE STUDY 2: RESQUE

Redis based processing queue

- forks a child per job to contain memory bloat
- communicates runtime status via process names

```
      perform(job, &block)

    # Processes a given job in the child.
    def perform(job)
      procline "Processing #{job.queue} since #{Time.now.to_i} [#{job.payload_class_name}]"
      begin
        run_hook :before_perform, job
        job.perform
        run_hook :after_perform, job
      rescue Object => e
        job.fail(e)
        failed!
      else
        Resque.logger.info "done: #{job.inspect}"
      ensure
        yield job if block_given?
      end
    end
```

displaying job details via process name

CASE STUDY 2: RESQUE

Redis based processing queue

- forks a child per job to contain memory bloat
- communicates runtime status via process names

more source code:

https://github.com/resque/resque/blob/master/lib/resque/worker.rb

COLLECTING CHILDREN

`Process.wait`

- waits for the next child to exit
- sets `$?` with Process::Status

`Process::Status`

pid	child's process id
exited?	true if exited normally
exitstatus	byte-sized exit status
signalled?	true if interrupted by a signed (kill'ed)
success?	true if exited with an exit code of 0

COLLECTING CHILDREN

Finer-grade control

```
Process.wait(pid=-1, flags=0)
Process.waitpid(pid=-1, flags=0)
  returns pid of exited child
```

```
include Process
fork { exit 99 }                    #=> 27429
wait                                #=> 27429
$?.exitstatus                       #=> 99

pid = fork { sleep 3 }              #=> 27440
Time.now                            #=> 2008-03-08 19:56:16 +0900
waitpid(pid, Process::WNOHANG)      #=> nil
Time.now                            #=> 2008-03-08 19:56:16 +0900
waitpid(pid, 0)                     #=> 27440
Time.now                            #=> 2008-03-08 19:56:19 +0900
```

```
Process.wait(pid=-1, flags=0)
Process.waitpid(pid=-1, flags=0)
  returns pid and Process::Status of exited child
```

```
Process.fork { exit 99 }    #=> 27437
pid, status = Process.wait2
pid                         #=> 27437
status.exitstatus           #=> 99
```

REAP YOUR ZOMBIES

- Exit status of a process is available until collected
- Exited process becomes a zombie until reaped

```
puts fork { exit 0 }
sleep
```

```
> ps axf | grep ruby
10828 pts/2  S+   0:00  |   \_ ruby tmp.rb
10829 pts/2  Z+   0:00  |       \_ [ruby] <defunct>
```

DEAD CHILDREN = ZOMBIES

- Dead children become zombies (even if for a short time)
- Zombies can't be killed
- Lots of zombies - something's wrong somewhere

DEAD CHILDREN = ZOMBIES

- Dead children become zombies (even if for a short time)
- Zombies can't be killed
- Lots of zombies - something's wrong somewhere

Make sure to always use Process.wait et al to reap child processes!

CASE STUDY 3: UNICORN

"I like Unicorn because it's Unix"
(c) Ryan Tomayko / GitHub

- Mongrel minus threads plus Unix processes
- leans heavily on OS kernel to balance connections and manage workers

UNICORN AS A REAPER

https://github.com/defunkt/unicorn/blob/master/lib/unicorn/http_server.rb#L380

```ruby
class Unicorn::HttpServer
  def join
    begin
      reap_all_workers
      case SIG_QUEUE.shift
      when nil
        master_sleep(sleep_time)
      # when ... <- handling signals here
      end
    end while true
    stop # gracefully shutdown all workers on our way out
  end

  def reap_all_workers
    begin
      wpid, status = Process.waitpid2(-1, Process::WNOHANG)
      wpid or return
      if reexec_pid == wpid
        logger.error "reaped #{status.inspect} exec()-ed"
        self.reexec_pid = 0
        self.pid = pid.chomp('.oldbin') if pid
        proc_name 'master'
      else
        worker = WORKERS.delete(wpid) and worker.close rescue nil
        m = "reaped #{status.inspect} worker=#{worker.nr rescue 'unknown'}"
        status.success? ? logger.info(m) : logger.error(m)
      end
    rescue Errno::ECHILD
      break
    end while true
  end
end
```

UNICORN AS A REAPER

https://github.com/defunkt/unicorn/blob/master/lib/unicorn/http_server.rb#L380

```ruby
class Unicorn::HttpServer
  def join
    begin
      reap_all_workers
      case SIG_QUEUE.shift
      when nil
        master_sleep(sleep_time)
      # when ... <- handling signals here
      end
    end while true
    stop # gracefully shutdown all workers on our way out
  end

  def reap_all_workers
    begin
      wpid, status = Process.waitpid2(-1, Process::WNOHANG)
      wpid or return
      if reexec_pid == wpid
        logger.error "reaped #{status.inspect} exec()-ed"
        self.reexec_pid = 0
        self.pid = pid.chomp('.oldbin') if pid
        proc_name 'master'
      else
        worker = WORKERS.delete(wpid) and worker.close rescue nil
        m = "reaped #{status.inspect} worker=#{worker.nr rescue 'unknown'}"
        status.success? ? logger.info(m) : logger.error(m)
      end
    rescue Errno::ECHILD
      break
    end while true
  end
end
```

master loop reaps exited workers until stopped

UNICORN AS A REAPER

https://github.com/defunkt/unicorn/blob/master/lib/unicorn/http_server.rb#L380

```ruby
class Unicorn::HttpServer
  def join
    begin
      reap_all_workers
      case SIG_QUEUE.shift
      when nil
        master_sleep(sleep_time)
      # when ... <- handling signals here
      end
    end while true
    stop # gracefully shutdown all workers on our way out
  end

  def reap_all_workers
    begin
      wpid, status = Process.waitpid2(-1, Process::WNOHANG)
      wpid or return
      if reexec_pid == wpid
        logger.error "reaped #{status.inspect} exec()-ed"
        self.reexec_pid = 0
        self.pid = pid.chomp('.oldbin') if pid
        proc_name 'master'
      else
        worker = WORKERS.delete(wpid) and worker.close rescue nil
        m = "reaped #{status.inspect} worker=#{worker.nr rescue 'unknown'}"
        status.success? ? logger.info(m) : logger.error(m)
      end
    rescue Errno::ECHILD
      break
    end while true
  end
end
```

collecting status of the next exited worker

UNICORN AS A REAPER

https://github.com/defunkt/unicorn/blob/master/lib/unicorn/http_server.rb#L380

```
class Unicorn::HttpServer
  def join
    begin
      reap_all_workers
      case SIG_QUEUE.shift
      when nil
        master_sleep(sleep_time)
      # when ... <- handling signals here
      end
    end while true
    stop # gracefully shutdown all workers on our way out
  end

  def reap_all_workers
    begin
      wpid, status = Process.waitpid2(-1, Process::WNOHANG)
      wpid or return
      if reexec_pid == wpid
        logger.error "reaped #{status.inspect} exec()-ed"
        self.reexec_pid = 0
        self.pid = pid.chomp('.oldbin') if pid
        proc_name 'master'
      else
        worker = WORKERS.delete(wpid) and worker.close rescue nil
        m = "reaped #{status.inspect} worker=#{worker.nr rescue 'unknown'}"
        status.success? ? logger.info(m) : logger.error(m)
      end
    rescue Errno::ECHILD
      break
    end while true
  end
end
```

renaming the master process when doing zero-downtime deploys

UNICORN AS A REAPER

https://github.com/defunkt/unicorn/blob/master/lib/unicorn/http_server.rb#L380

```ruby
class Unicorn::HttpServer
  def join
    begin
      reap_all_workers
      case SIG_QUEUE.shift
      when nil
        master_sleep(sleep_time)
      # when ... <- handling signals here
      end
    end while true
    stop # gracefully shutdown all workers on our way out
  end

  def reap_all_workers
    begin
      wpid, status = Process.waitpid2(-1, Process::WNOHANG)
      wpid or return
      if reexec_pid == wpid
        logger.error "reaped #{status.inspect} exec()-ed"
        self.reexec_pid = 0
        self.pid = pid.chomp('.oldbin') if pid
        proc_name 'master'
      else
        worker = WORKERS.delete(wpid) and worker.close rescue nil
        m = "reaped #{status.inspect} worker=#{worker.nr rescue 'unknown'}"
        status.success? ? logger.info(m) : logger.error(m)
      end
    rescue Errno::ECHILD
      break
    end while true
  end
end
```

wpid is PID of the reaped worker - close communication pipe and remove its data

UNICORN AS A REAPER

https://github.com/defunkt/unicorn/blob/master/lib/unicorn/http_server.rb#L380

```ruby
class Unicorn::HttpServer
  def join
    begin
      reap_all_workers
      case SIG_QUEUE.shift
      when nil
        master_sleep(sleep_time)
      # when ... <- handling signals here
      end
    end while true
    stop # gracefully shutdown all workers on our way out
  end

  def reap_all_workers
    begin
      wpid, status = Process.waitpid2(-1, Process::WNOHANG)
      wpid or return
      if reexec_pid == wpid
        logger.error "reaped #{status.inspect} exec()-ed"
        self.reexec_pid = 0
        self.pid = pid.chomp('.oldbin') if pid
        proc_name 'master'
      else
        worker = WORKERS.delete(wpid) and worker.close rescue nil
        m = "reaped #{status.inspect} worker=#{worker.nr rescue 'unknown'}"
        status.success? ? logger.info(m) : logger.error(m)
      end
    rescue Errno::ECHILD
      break
    end while true
  end
end
```

generate log message and put it into normal or error log depending on exit status

UNICORN AS A REAPER

https://github.com/defunkt/unicorn/blob/master/lib/unicorn/http_server.rb#L380

```
class Unicorn::HttpServer
  def join
    begin
      reap_all_workers
      case SIG_QUEUE.shift
      when nil
        master_sleep(sleep_time)
      # when ... <- handling signals here
      end
    end while true
    stop # gracefully shutdown all workers on our way out
  end

  def reap_all_workers
    begin
      wpid, status = Process.waitpid2(-1, Process::WNOHANG)
      wpid or return
      if reexec_pid == wpid
        logger.error "reaped #{status.inspect} exec()-ed"
        self.reexec_pid = 0
        self.pid = pid.chomp('.oldbin') if pid
        proc_name 'master'
      else
        worker = WORKERS.delete(wpid) and worker.close rescue nil
        m = "reaped #{status.inspect} worker=#{worker.nr rescue 'unknown'}"
        status.success? ? logger.info(m) : logger.error(m)
      end
    rescue Errno::ECHILD
      break
    end while true
  end
end
```

keep doing it until all exited workers are reaped

READ THE CODE

Ryan Tomayko, "I like Unicorn because it's Unix"

http://tomayko.com/writings/unicorn-is-unix

Official site

http://unicorn.bogomips.org/

Source code

HTTP_Server: https://github.com/defunkt/unicorn/blob/master/lib/unicorn/http_server.rb

MORE IPC

- Signals
- Pipes
- Sockets (TCP, Unix)

but there's enough to make a whole separate talk

MORE ABOUT PROCESSES

Jesse Storimer, "Working with Unix processes"

http://www.workingwithunixprocesses.com/

Eric Wong, "Unix System Programming in Ruby"

http://librelist.com/browser/usp.ruby/

THREADS AREN'T PROCESSES

- a process has many threads (at least two in Ruby)

- each thread has its own:

  - stack
  - registers
  - execution pointer

THREADS AREN'T PROCESSES

- a process has many threads (at least two in Ruby)

- each thread has its own:

  - stack
  - registers
  - execution pointer

but the memory is shared between all threads!

WHAT IS "SHARED MEMORY"

Processes

```
counter = 0

worker = Proc.new do
  1.upto(1000) { counter += 1 }
  worker_id = $$
  puts "worker #{worker_id}: #{counter}"
end

fork(&worker)
fork(&worker)

Process.waitall
puts "parent counter = #{counter}"
```

```
% ruby tmp.rb
process 23307: 1000
process 23308: 1000
parent counter = 0
```

Threads

```
counter = 0

worker = Proc.new do
  1.upto(1000) { counter += 1 }
  worker_id = Thread.current.object_id
  puts "worker #{worker_id}: #{counter}"
end

[
  Thread.new(&worker), Thread.new(&worker)
].each(&:join)

puts "counter = #{counter}"
```

```
% ruby tmp.rb
thread 70336137980280: 1774
thread 70336137980180: 2000
counter = 2000
% ruby -v
ruby 2.0.0p0 (2013-02-24 revision 39474)
```

WHAT IS "SHARED MEMORY"

Processes

```ruby
counter = 0

worker = Proc.new do
  1.upto(1000) { counter += 1 }
  worker_id = $$
  puts "worker #{worker_id}: #{counter}"
end

fork(&worker)
fork(&worker)

Process.waitall
puts "parent counter = #{counter}"
```

```
% ruby tmp.rb
process 23307: 1000
process 23308: 1000
parent counter = 0
```

counter var is separate

Threads

```ruby
counter = 0

worker = Proc.new do
  1.upto(1000) { counter += 1 }
  worker_id = Thread.current.object_id
  puts "worker #{worker_id}: #{counter}"
end

[
  Thread.new(&worker), Thread.new(&worker)
].each(&:join)

puts "counter = #{counter}"
```

```
% ruby tmp.rb
thread 70336137980280: 1774
thread 70336137980180: 2000
counter = 2000
% ruby -v
ruby 2.0.0p0 (2013-02-24 revision 39474)
```

counter var is shared

IMPLICATIONS

- context switch can occur at any time in any thread
- make sure your threads are not read-writing common data
- compound operations (like ||= or +=) can be interrupted in the middle!
- use `Thread.current[:varname]` if you need to

IMPLICATIONS

- context switch can occur at any time in any thread
- make sure your threads are not read-writing common data
- compound operations (like ||= or +=) can be interrupted in the middle!
- use `Thread.current[:varname]` if you need to

Example: memoization

bad

```
class Client
  def self.channel
    @c ||= Channel.new
  end
end
```

better

```
class Client
  def self.channel
    Thread.current[:channel] ||= Channel.new
  end
end
```

GREEN THREADS

- Ruby < 1.9 - Ruby-owned thread management
- Invisible to and unmanageable by OS kernel
- OS still runs a single thread by process
- Concurrency, but not parallelization
- Green threads are NOT UNIX

GREEN THREADS

- Ruby < 1.9 - Ruby-owned thread management
- Invisible to and unmanageable by OS kernel
- OS still runs a single thread by process
- Concurrency, but not parallelization
- Green threads are NOT UNIX

SUCK!

NATIVE THREADS IN MRI

- Ruby 1.9 and 2.0
- allow for truly parallel execution
- but only on blocking IO
- Global Interpreter Lock (GIL) on Ruby code execution

GIL

- Protects MRI internals from thread safety issues
- Allows MRI to operate with non-thread-safe C extensions
- Isn't going away any time soon

GIL

- Protects MRI internals from thread safety issues
- Allows MRI to operate with non-thread-safe C extensions
- Isn't going away any time soon

Makes sure your Ruby code will NEVER run in parallel on MRI

PURE RUBY CODE AND GIL

```ruby
require 'benchmark'
require 'digest/sha2'

worker = Proc.new do
  200_000.times { Digest::SHA512.hexdigest('DEADBEEF') }
end

Benchmark.bm do |bb|
  bb.report 'single' do
    5.times(&worker)
  end

  bb.report 'multi' do
    5.times.map { Thread.new(&worker) }.each(&:join)
  end
end
```

PURE RUBY CODE AND GIL

```ruby
require 'benchmark'
require 'digest/sha2'

worker = Proc.new do
  200_000.times { Digest::SHA512.hexdigest('DEADBEEF') }
end

Benchmark.bm do |bb|
  bb.report 'single' do
    5.times(&worker)
  end

  bb.report 'multi' do
    5.times.map { Thread.new(&worker) }.each(&:join)
  end
end
```

```
% ruby-2.0.0-p0 tmp.rb
            ...         real
single ...   (  1.935500)
multi  ...   (  2.093167)
```

PURE RUBY CODE AND GIL

```ruby
require 'benchmark'
require 'digest/sha2'

worker = Proc.new do
  200_000.times { Digest::SHA512.hexdigest('DEADBEEF') }
end

Benchmark.bm do |bb|
  bb.report 'single' do
    5.times(&worker)
  end

  bb.report 'multi' do
    5.times.map { Thread.new(&worker) }.each(&:join)
  end
end
```

```
% ruby-2.0.0-p0 tmp.rb
            ...       real
single      ...   (  1.935500)
multi       ...   (  2.093167)
```

GIL doesn't allow pure Ruby code to run in parallel,
thus the same time as sequential code

PURE RUBY CODE AND GIL

```ruby
require 'benchmark'
require 'digest/sha2'

worker = Proc.new do
  200_000.times { Digest::SHA512.hexdigest('DEADBEEF') }
end

Benchmark.bm do |bb|
  bb.report 'single' do
    5.times(&worker)
  end

  bb.report 'multi' do
    5.times.map { Thread.new(&worker) }.each(&:join)
  end
end
```

```
% ruby-2.0.0-p0 tmp.rb
            ...         real
single ...    (  1.935500)
multi  ...    (  2.093167)
```

```
% jruby-1.7.3 tmp.rb
            ...         real
single  ...   (  2.450000)
multi   ...   (  1.089000)
```

jRuby doesn't have GIL, so fully parallel execution

PURE RUBY CODE AND GIL

```
% jruby-1.7.3 tmp.rb
            ...       real
single  ... (  2.450000)
multi   ... (  1.089000)
```

PURE RUBY CODE AND GIL

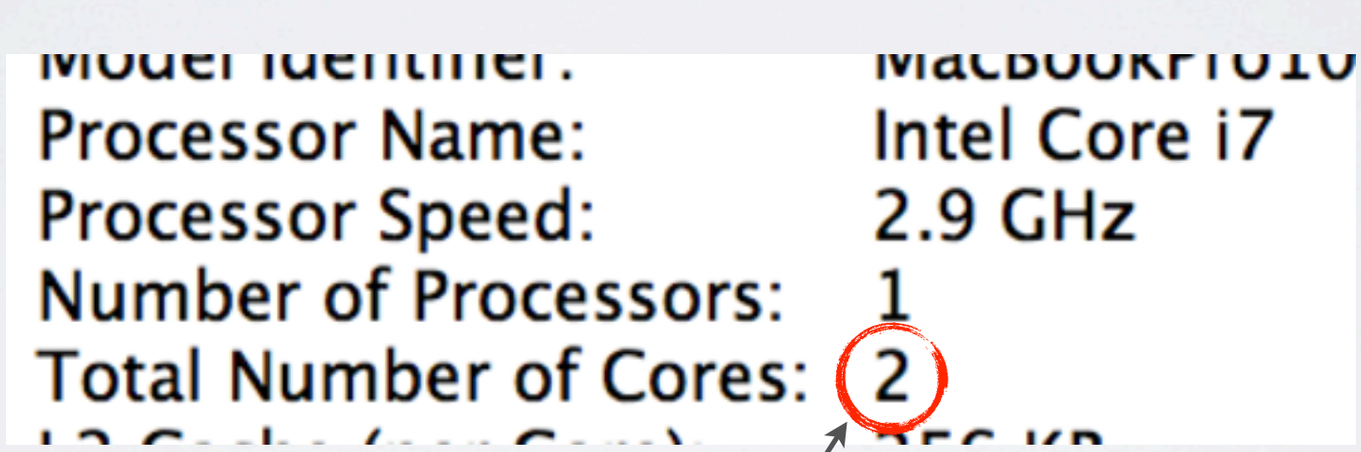
```
% jruby-1.7.3 tmp.rb
          ...       real
single  ... (  2.450000)
multi   ... (  1.089000)
```

why only 2x faster, if we're running 5 threads?

PURE RUBY CODE AND GIL

```
% jruby-1.7.3 tmp.rb
          ...       real
single  ... (  2.450000)
multi   ... (  1.089000)
```

why only 2x faster, if we're running 5 threads?



one thread per physical CPU core

CPU-BOUND THREADS

- run mostly Ruby code and calculations
- with GIL - execute concurrently, but not in parallel
- for parallel processing - must be split into processes, or run on non-MRI implementation
- scaling limited to the number of physical cores

IO-BOUND THREADS

- spend most of their time waiting for blocking IO operations

- can run in parallel on Ruby 1.9

- scale beyond physical cores, but still up to a limit

IO-BOUND THREADS

```ruby
require 'benchmark'
require 'open-uri'

worker = Proc.new do
  5.times { open 'http://google.com' }
end

Benchmark.bm do |bb|
  bb.report 'single' do
    5.times(&worker)
  end

  bb.report 'multi' do
    5.times.map { Thread.new(&worker) }.each(&:join)
  end
end
```

IO-BOUND THREADS

```ruby
require 'benchmark'
require 'open-uri'

worker = Proc.new do
  5.times { open 'http://google.com' }
end

Benchmark.bm do |bb|
  bb.report 'single' do
    5.times(&worker)
  end

  bb.report 'multi' do
    5.times.map { Thread.new(&worker) }.each(&:join)
  end
end
```

```
% ruby-2.0.0-p0 tmp.rb
            ...         real
single      ...   ( 13.714725)
multi       ...   (  3.539165)
```

```
% jruby-1.7.3 tmp.rb
            ...       real
single      ... ( 15.273000)
multi       ... (  3.820000)
```

- not affected by GIL
- scale proportionally to the number of threads

FIBERS ARE NEW GREEN

- user-space threads introduced in Ruby 1.9
- scheduled by Ruby, not OS kernel
- very useful for event-based code
- need to behave cooperatively to be efficient

THREAD EXEC CONTROLS

- mutexes
- semaphores
- conditional variables
- thread-safe data structures

IN THE WILD: SIDEKIQ

http://sidekiq.org

- Efficient messaging processing
- Unlike Resque, utilizes threads instead of processes
- Shines with IO-heavy workers

IN THE WILD: CELLULOID

http://celluloid.io

- Framework to work with threads and build concurrent apps
- Allows to work with threads as with regular app objects
- Easy async calls to other threads

MORE ON THREADS ET AL

Jesse Storimer, "Working with Ruby threads"
http://www.workingwithrubythreads.com

yours truly,
"Concurrent programming and threads in Ruby - reading list"
http://speakmy.name/2013/04/02/concurrent-programming-and-threads-in-ruby-reading-list/

"Those who don't understand Unix
are condemned to reinvent it, poorly"

Henry Spencer