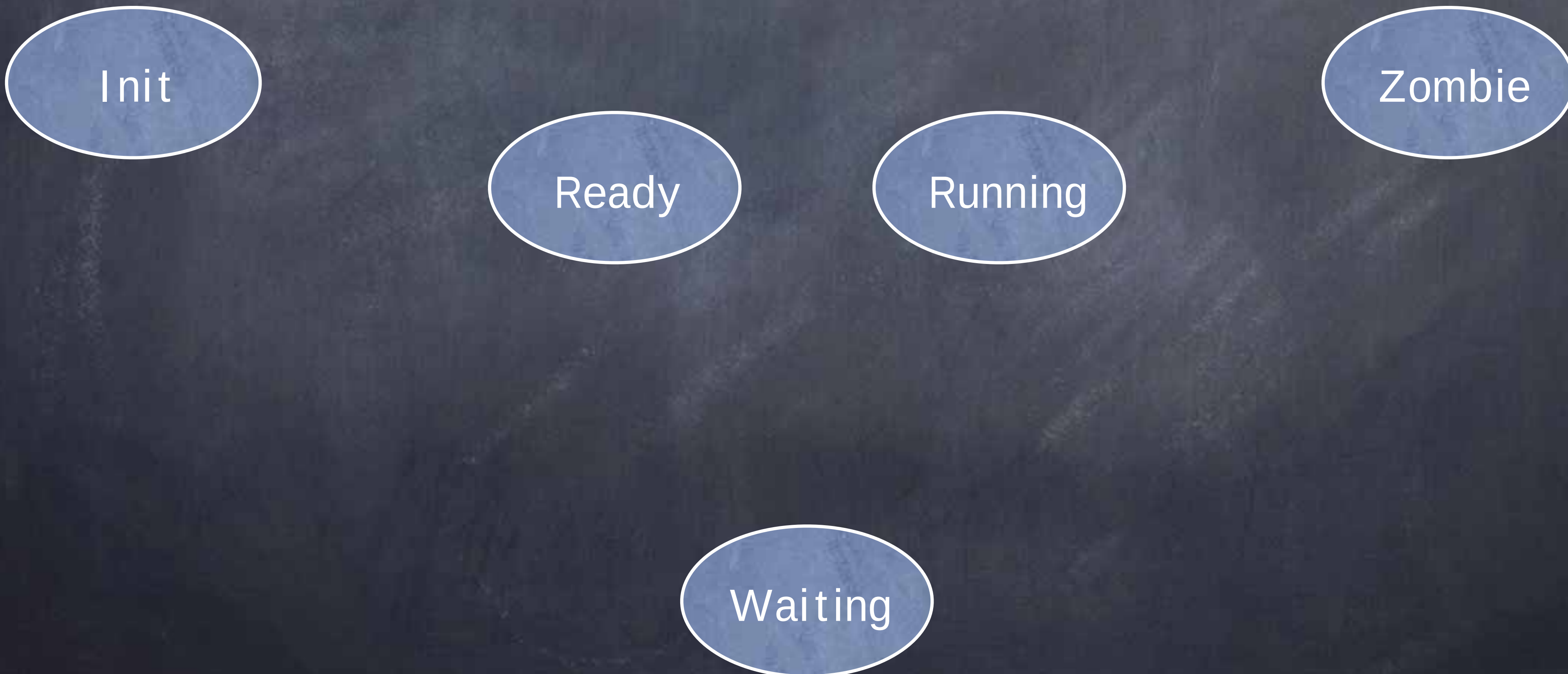
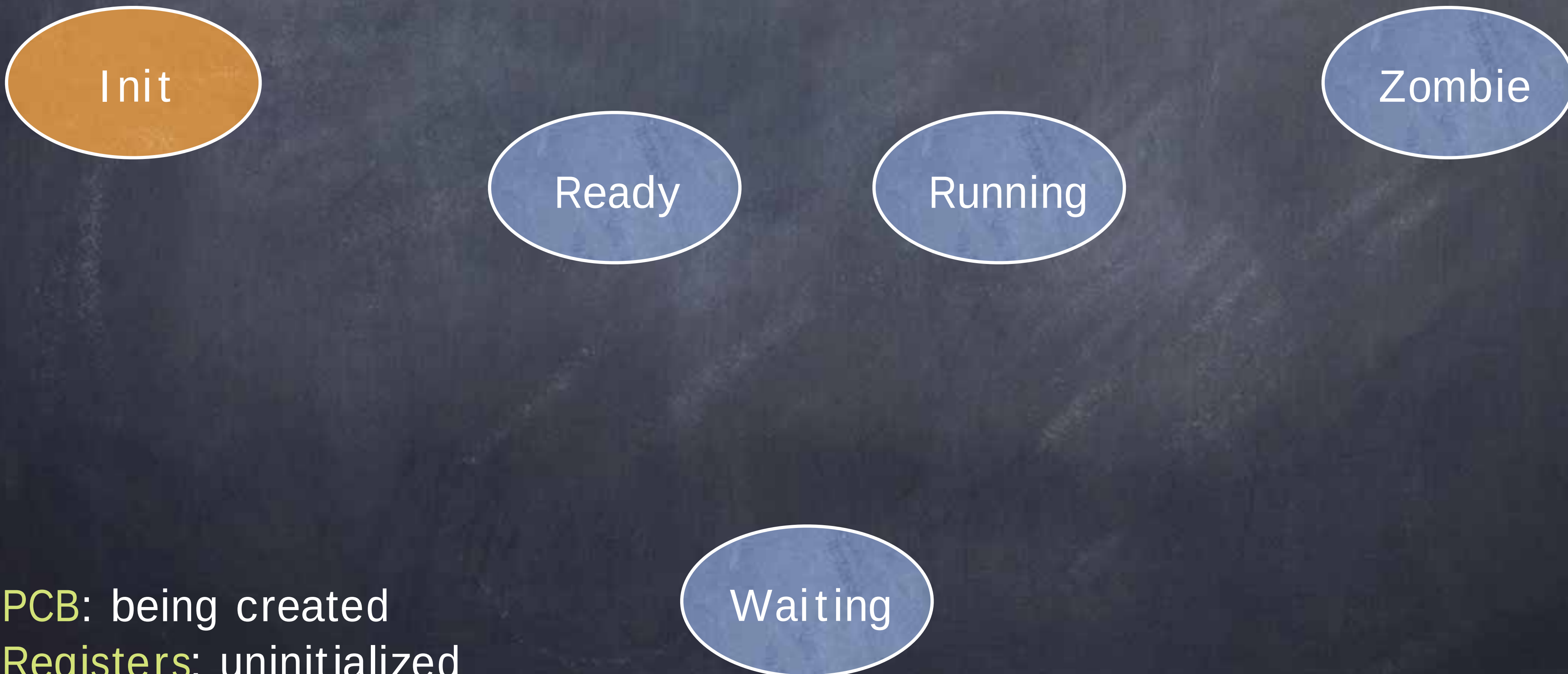
Process Life Cycle

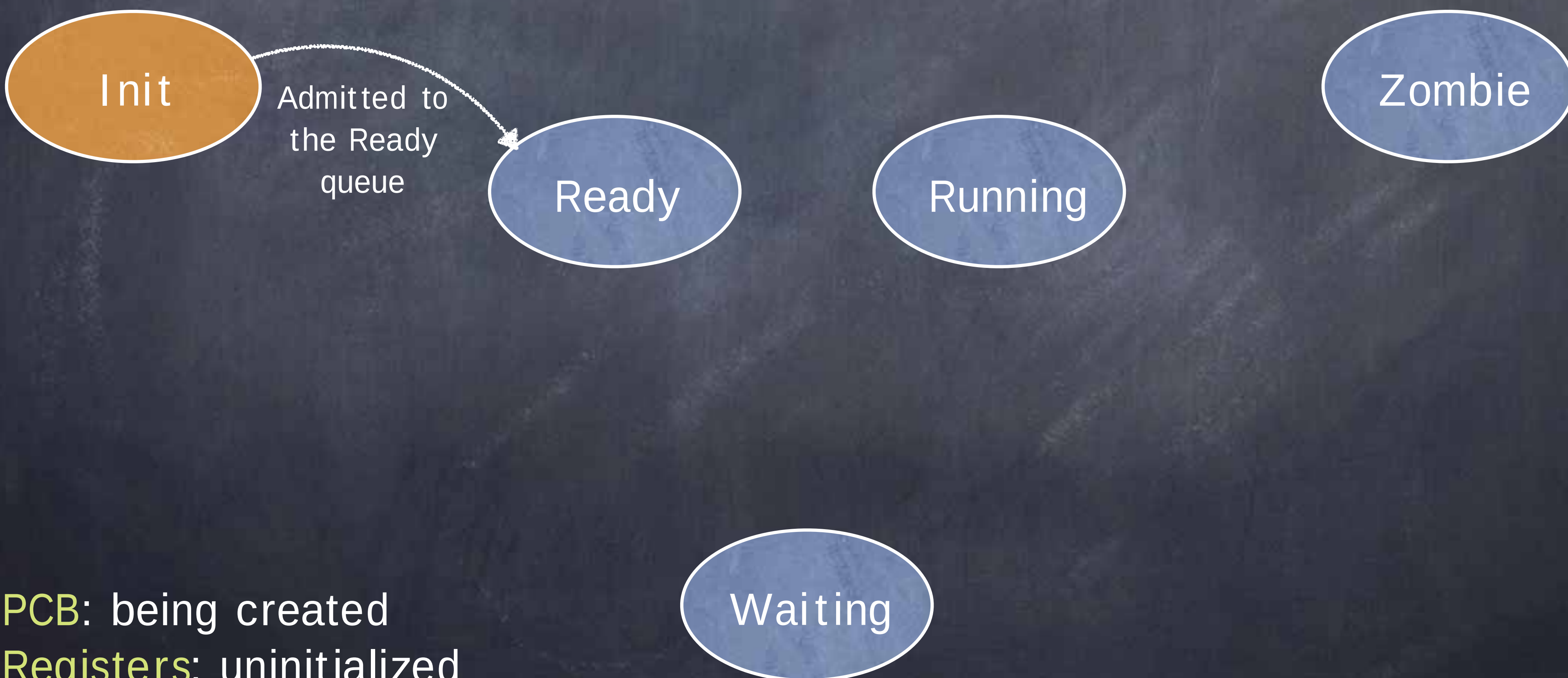


Process Life Cycle



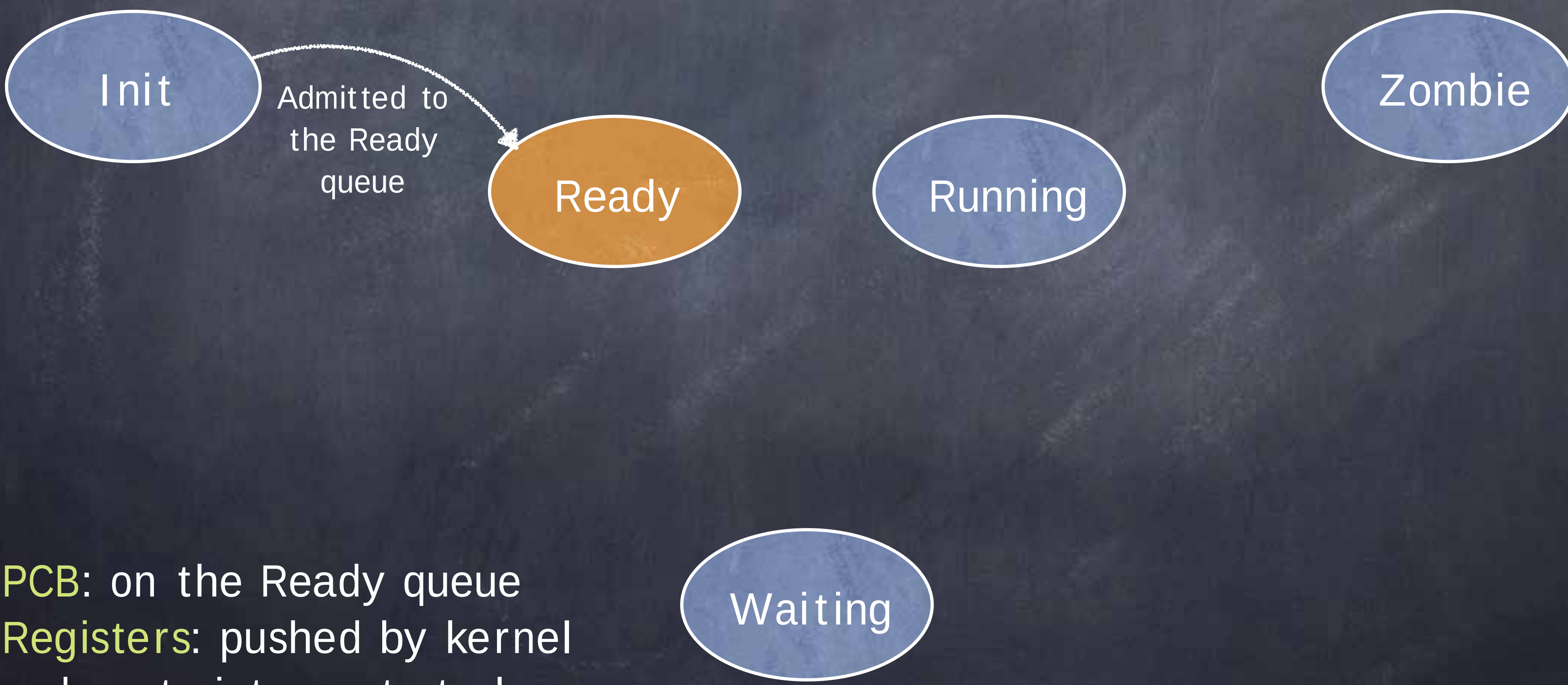
PCB: being created
Registers: uninitialized

Process Life Cycle



PCB: being created
Registers: uninitialized

Process Life Cycle

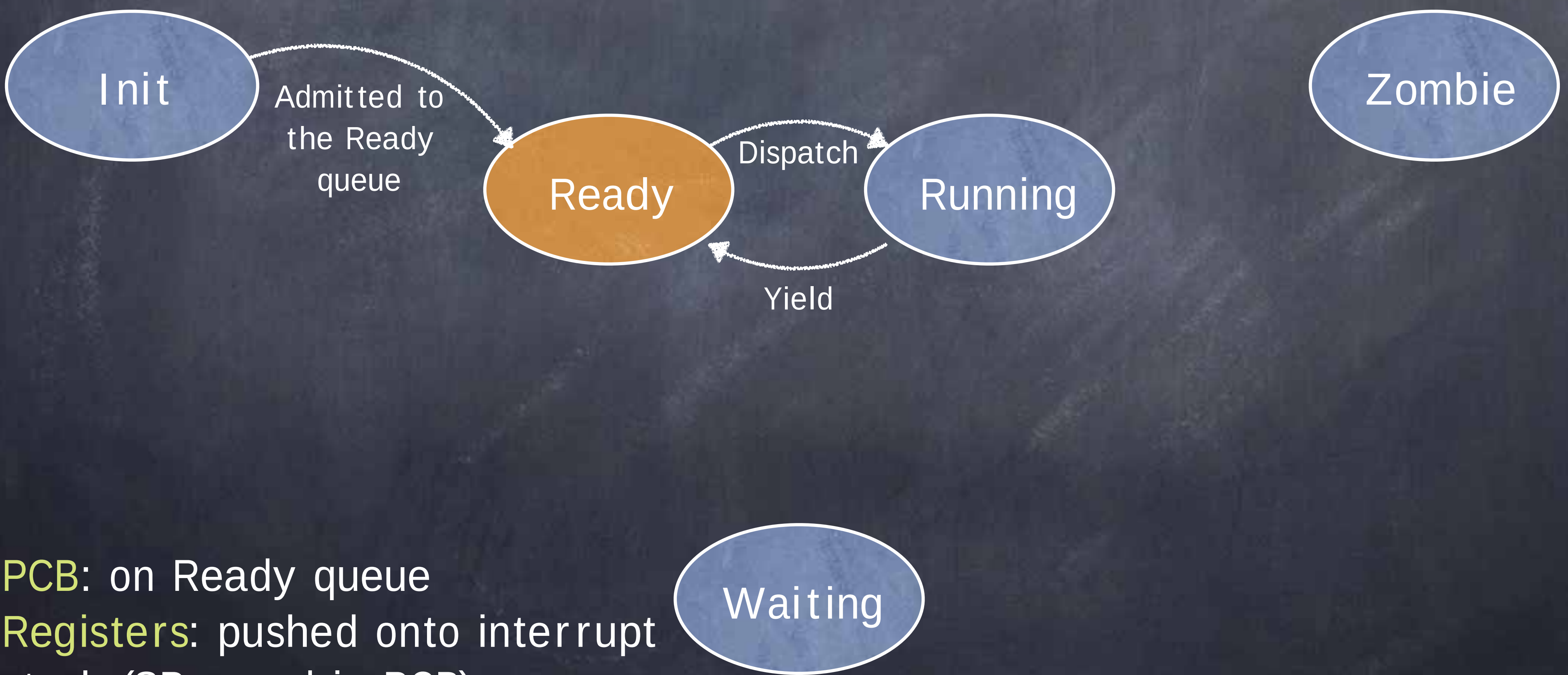


PCB: on the Ready queue
Registers: pushed by kernel code onto interrupt stack

Process Life Cycle

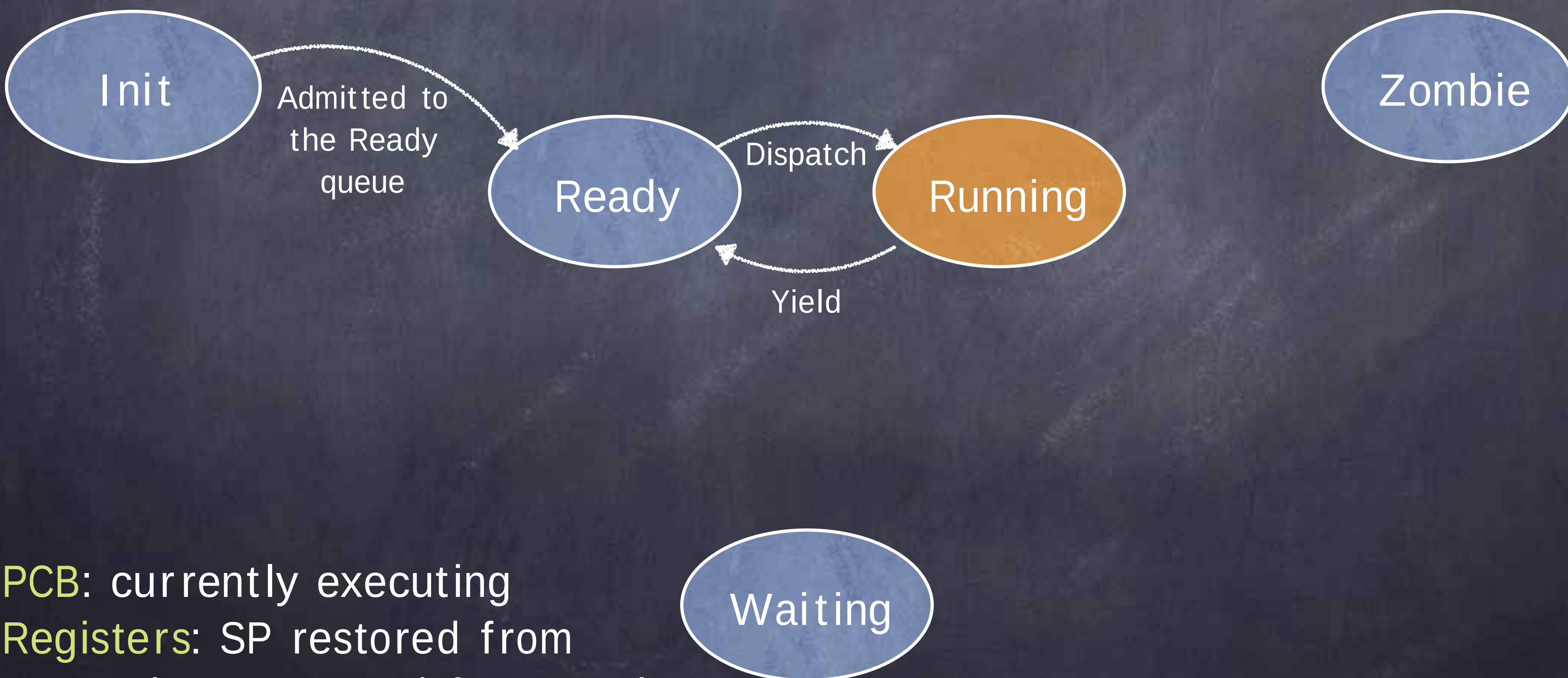
PCB: currently executing
Registers: popped from interrupt stack into CPU

Process Life Cycle



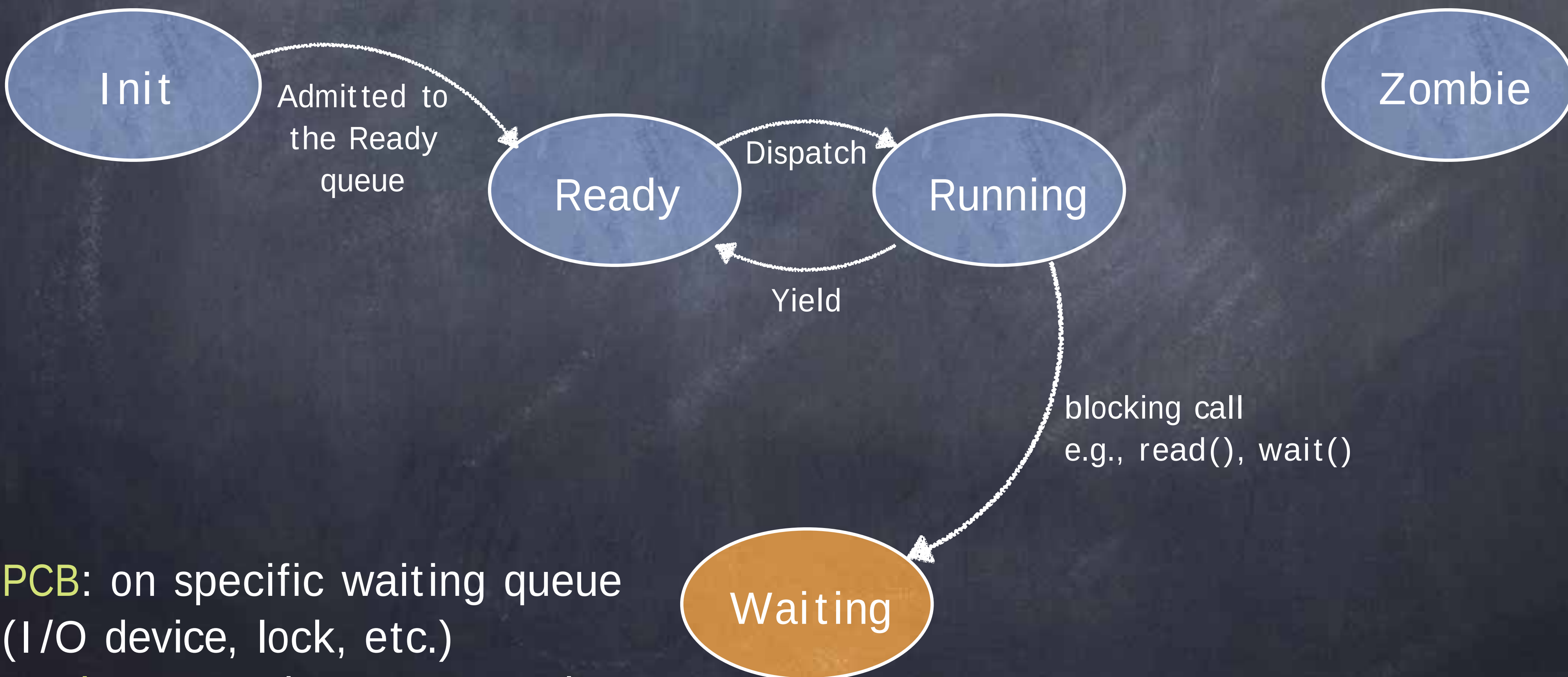
PCB: on Ready queue
Registers: pushed onto interrupt stack (SP saved in PCB)

Process Life Cycle



PCB: currently executing
Registers: SP restored from PCB; others restored from stack

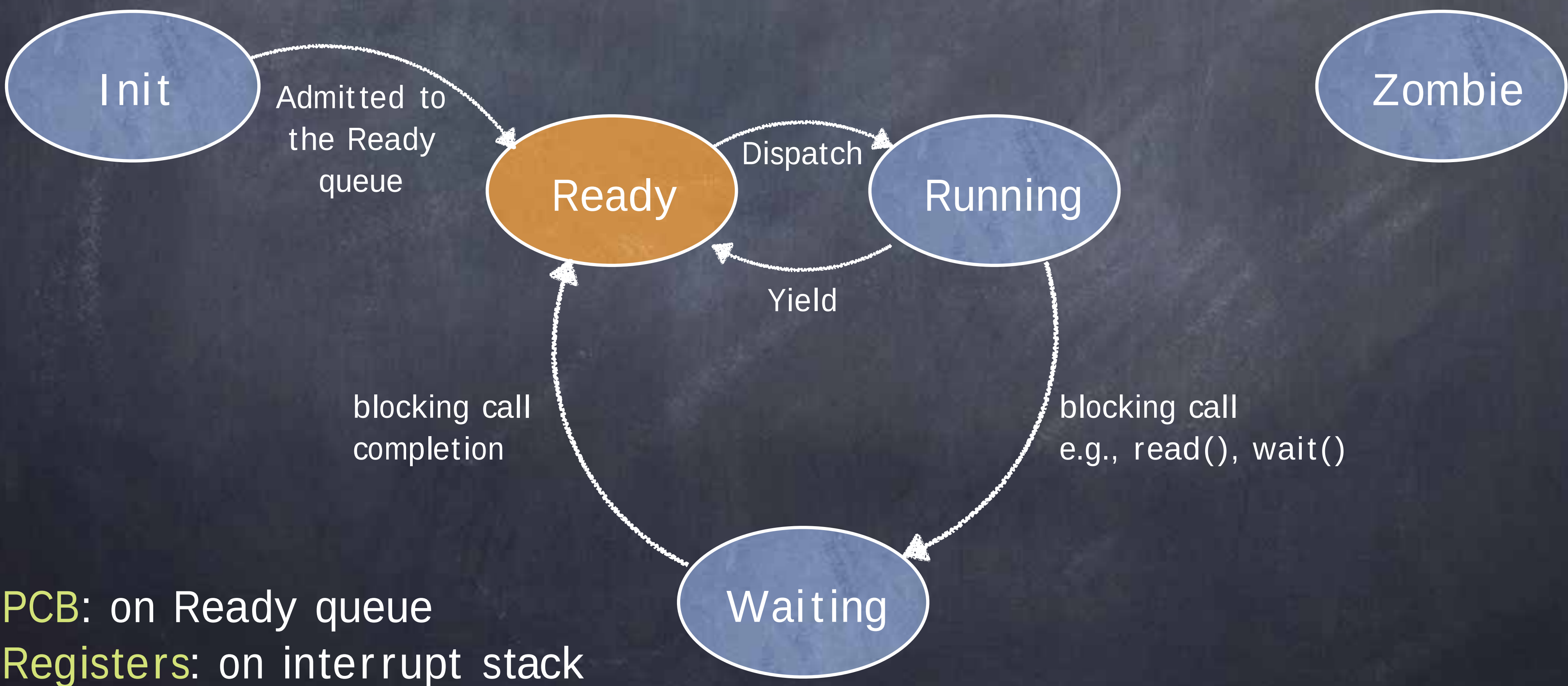
Process Life Cycle



PCB: on specific waiting queue (I/O device, lock, etc.)

Registers: on interrupt stack

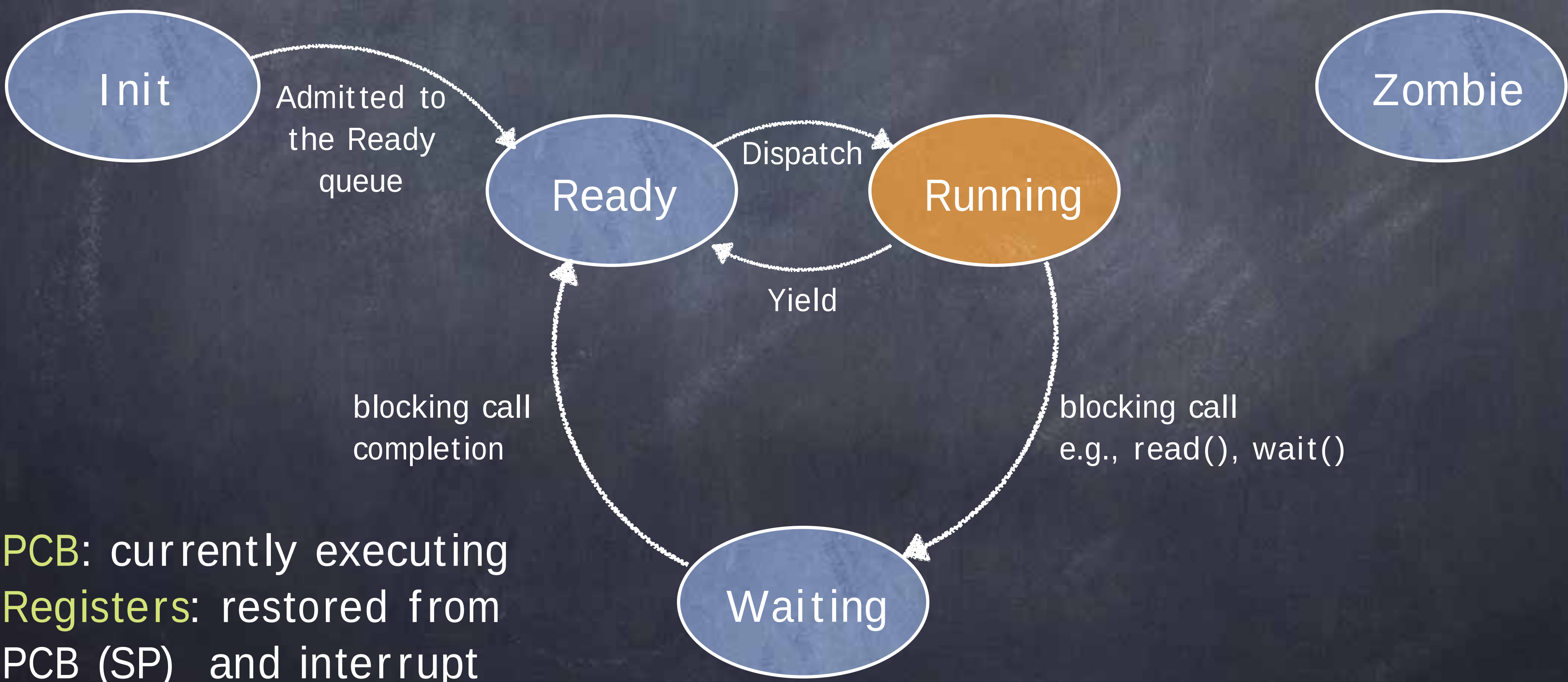
Process Life Cycle



PCB: on Ready queue

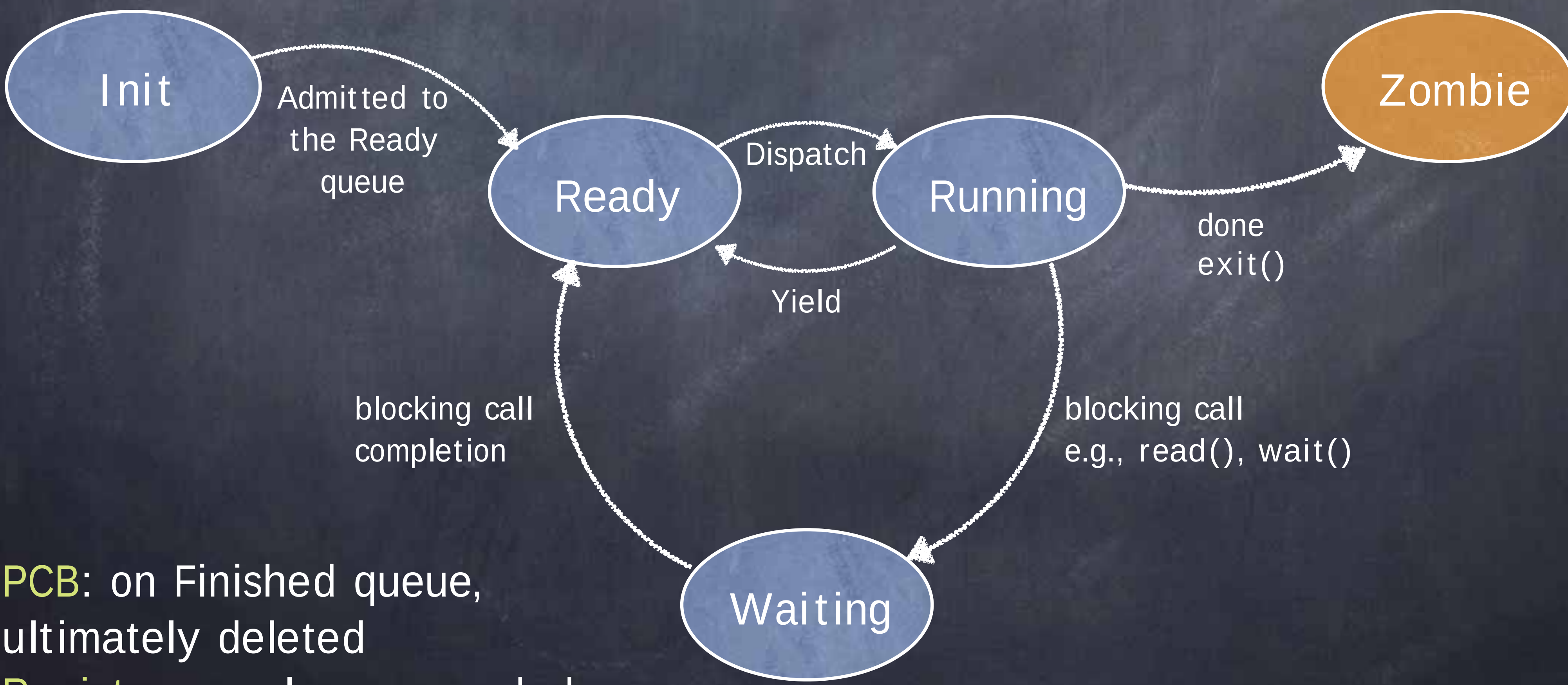
Registers: on interrupt stack

Process Life Cycle



PCB: currently executing
Registers: restored from PCB (SP) and interrupt stack into CPU

Process Life Cycle



PCB: on Finished queue, ultimately deleted
Registers: no longer needed

Invariants to keep in mind

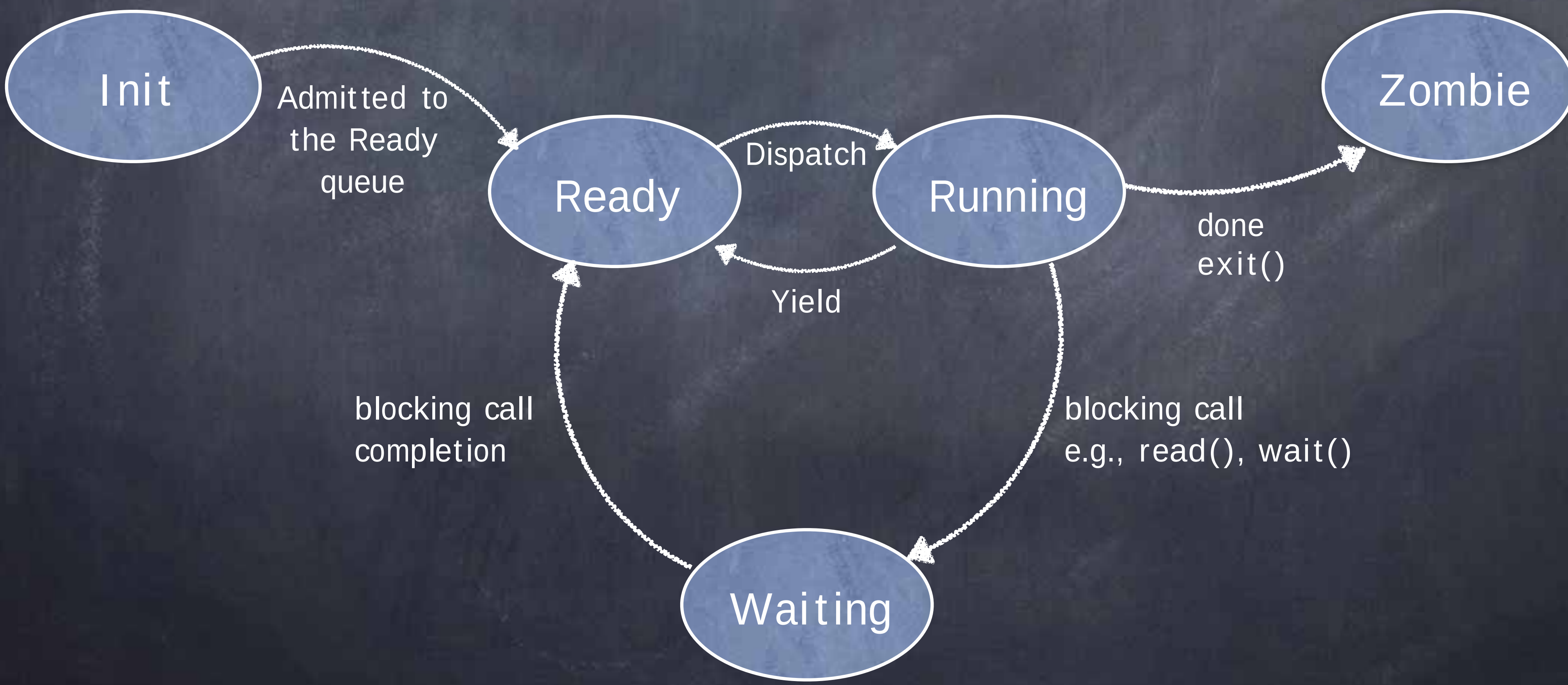
- At most one process/core running at any time
- When CPU in user mode, current process is RUNNING and its interrupt stack is empty
- If process is RUNNING
  - its PCB not on any queue
  - it is not necessarily in USER mode
- If process is READY or WAITING
  - its registers are saved at the top of its interrupt stack
  - its PCB is either
    - on the READY queue (if READY)
    - on some WAIT queue (if WAITING)
- If process is a ZOMBIE
  - its PCB is on FINISHED queue

Cleaning up Zombies

- Process cannot clean up itself (why?)
- Process can be cleaned up
  - by some other process, checking for zombies before returning to RUNNING state
  - or by parent which waits for it
    - but what if parent turns into a zombie first?
  - or by a dedicated “reaper” process

- Linux uses a combination
  - if alive, parent cleans up child that it is waiting for
  - if parent is dead, child process is inherited by the initial process, which is continually waiting

Process Life Cycle



How to Yield/Wait?

- Must switch from executing the current process to executing some other READY process
  - Current process: RUNNING $\rightarrow$ READY
  - Next process: READY $\rightarrow$ RUNNING

1. Save kernel registers of Current on its interrupt stack
2. Save kernel SP of Current in its PCB
3. Restore kernel SP of Next from its PCB
4. Restore kernel registers of Next from its interrupt stack

ctx_switch(&old_sp, new_sp)

```
ctx_switch: //ip already pushed
    pushq   %rbp
    pushq   %rbx
    pushq   %r15
    pushq   %r14
    pushq   %r13
    pushq   %r12
    pushq   %r11
    pushq   %r10
    pushq   %r9
    pushq   %r8
    movq    %rsp, (%rdi)
    movq    %rsi, %rsp
    popq    %rbp
    popq    %rbx
    popq    %r15
    popq    %r14
    popq    %r13
    popq    %r12
    popq    %r11
    popq    %r10
    popq    %r9
    popq    %r8
    retq
```

```
struct pcb *current, *next;

void yield(){
   assert(current->state == RUNNING);
   current->state = READY;
   readyQueue.add(current);
   next = scheduler();
   next->state = RUNNING;
   ctx_switch(&current->sp, next->sp)
   current = next;
}
```

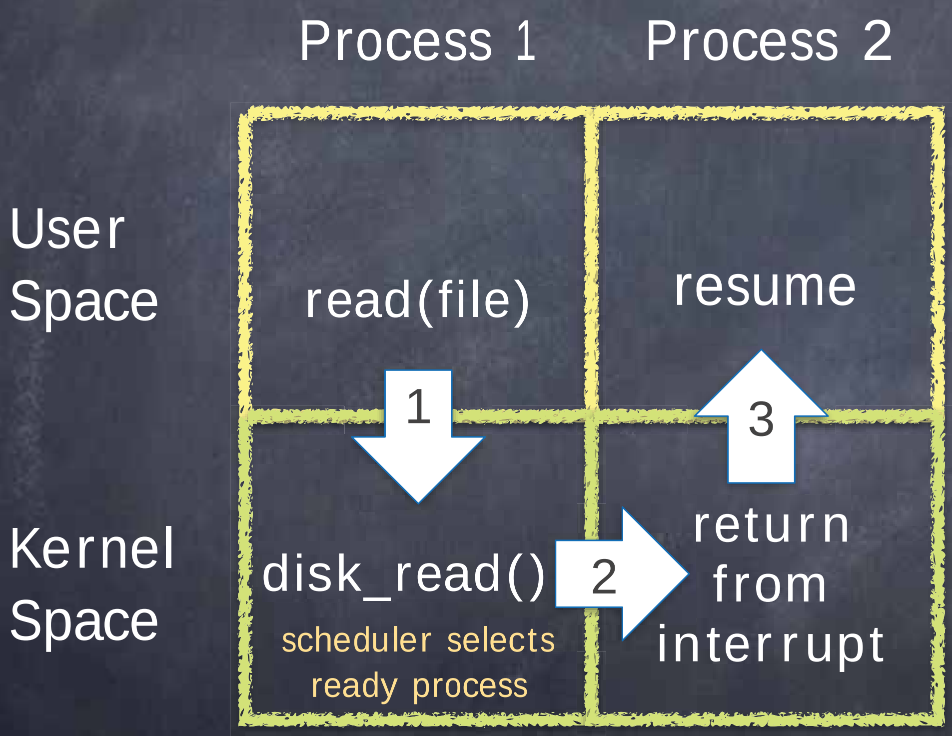
Anybody there?

- What if no process is READY?
  - scheduler() would return NULL – aargh!
- No panic on the Titanic:
  - OS always runs a low priority process, in an infinite loop executing the HLT instruction
    - halts CPU until next interrupt
  - Interrupt handler executes yield() if some other process is put on the Ready queue

Three Flavors of Context Switching

- Interrupt: from user to kernel space
  - on system call, exception, or interrupt
  - Stack switch: P_x user stack $\rightarrow P_x$ interrupt stack
- Yield: between two processes, inside kernel
  - from one PCB/interrupt stack to another
  - Stack switch: P_x interrupt stack $\rightarrow$ P_y interrupt stack
- Return from interrupt: from kernel to user space
  - with the homonymous instruction
  - Stack switch: P_x interrupt stack $\rightarrow P_x$ user stack

Switching between Processes



1. Save Process 1 user registers
2. Save Process 1 kernel registers and restore Process 2 kernel registers
3. Restore Process 2 user registers

System Calls to Create a New Process

- Must, implicitly or explicitly, specify the initial state of every OS resource belonging to the new process.

  - Windows
    - CreateProcess(...);
  - Unix (Linux)
    - fork() + exec(...)

CreateProcess (Simplified)

```
if (!CreateProcess(
    NULL,              // No module name (use command line)
    argv[1],           // Command line
    NULL,              // Process handle not inheritable
    NULL,              // Thread handle not inheritable
    FALSE,             // Set handle inheritance to FALSE
    0,                 // No creation flags
    NULL,              // Use parent's environment block
    NULL,              // Use parent's starting directory
    &si,               // Pointer to STARTUPINFO structure
    &pi )              // Ptr to PROCESS_INFORMATION structure
)
```

[Windows]

fork (actual form)

process identifier

```
int pid = fork();
```

..but needs exec(...)

[Unix]

Kernel Actions to Create a Process

- fork()
  - allocate ProcessID
  - initialize PCB
  - create and initialize new address space
    - identical to the one of the caller, but for the return value of the fork() system call
  - inform scheduler new process is READY
- exec(program, arguments)
  - load program into address space
  - copy arguments into address space's memory
  - initialize h/w context to start execution at ``start”

The rationale for fork() and exec()

- To redirect stdin/stdout:
  - fork, close/open files, exec
- To switch users:
  - fork, setuid, exec
- To start a process with a different current directory:
  - fork, chdir, exec

You get the idea!

But see also:

"A fork() in the road"
A. Baumann et al. (2019)

- A hack to begin with
- No longer simple
- Not composable
- Not thread safe
- Roots for Harvard
- Insecure
- Slow
- Doesn't scale

Creating and managing processes

Syscall	Description
fork()	Create a child process as a clone of the current process. Return to both parent and child. Return child's pid to parent process; return 0 to child
exec (prog, args)	Run application prog in the current process with the specified args (replacing any code and data that was present in process)
wait (&status)	Pause until a child process has exited
exit (status)	Current process is complete and should be garbage collected.
kill (pid, type)	Send an interrupt of a specified type to a process (a bit of an overdramatic misnomer...)

[Unix]

In action

Process 13
Program A

PC

pid
?

```
pid = fork();
if (pid==0)
   exec(B);
else
wait(&status);
```

In action

Process 13
Program A

PC

pid
?

```
pid = fork();
if (pid==0)
   exec(B);
else
wait(&status);
```

Process 13
Program A

PC

pid
14

```
pid = fork();
if (pid==0)
   exec(B);
else
wait(&status);
```

Process 14
Program A B

PC

pid
0

```
pid = fork();
if (pid==0)
   exec(B);
else
wait(&status);
```

```
main() {
  ...
  exit(3);
}
```

In action

Process 13
Program A

PC

pid
?

```
pid = fork();
if (pid==0)
   exec(B);
else
wait(&status);
```

Process 13
Program A

PC

pid
14

```
pid = fork();
if (pid==0)
   exec(B);
else
wait(&status);
```

Status
3

Process 14
Program B

PC

```
main() {
    ...
   exit(3);
}
```

In action (I)

```
#include <stdio.h>
#include <unistd.h>

int main() {

   int child_pid = fork();

   if (child_pid == 0) {          // child process
       printf("I am process %d\n", getpid());
       return 0;
   } else {                       // parent process
       printf("I am the parent of process %d\n", child_pid);
       return 0;
   }
}
```

Possible outputs?

In action (II)

```
#include <stdio.h>
#include <unistd.h>

int main() {

  prinf("I am proud process %d", getpid();)

  int child_pid = fork();

    if (child_pid == 0) {          // child process
        printf("\nI am process %d\n", getpid());
        return 0;
   } else {                        // parent process
        printf("I am process %d, the parent of process %d\n", getpid(), child_pid);
        return 0;
   }
}
```

Possible outputs?

Booting an OS

- ``pull oneself over a fnce by one's bootstraps"
- Steps in booting an O.S.:
  - CPU starts at fixed address
    - in supervisor mode, with interrupts disabled
  - BIOS (in ROM) loads "boot loader" code from specified storage or network device into memory and runs it
  - Boot loader loads OS kernel code into memory and runs it

O.S. initialization

- Determine location/size of physical memory
- Set up initial MMU/page tables
- Initialize the interrupt vector
- Determine which devices the computer has
  - invoke device driver initialization code for each
- Initialize file system code
- Load first process from file system
- Start first process

Review

- A process is an abstraction of a running program
- A context captures the running state of a process:
  - registers (including PC, SP, PSW)
  - memory (including the code, heap, stack)
- The implementation uses two contexts:
  - user context
  - kernel (supervisor) context
- A Process Control Block (PCB) points to both contexts and has other information about the process

Review

- Processes can be in one of the following states:
  - Initializing
  - Running
  - Ready (aka “runnable” on the “ready” queue)
  - Waiting (aka Sleeping or Blocked)
  - Zombie

What is “load”?

- It is the length of the ready queue
- On MacOSX “uptime” at command line reports load averaged over
  - last 1 minute
  - last 5 minutes
  - last 15 minutes
- “top” provides more information about running processes, e.g.,

Processes: 342 total, 2 running
Load Avg: 1.38, 1.64, 1.81

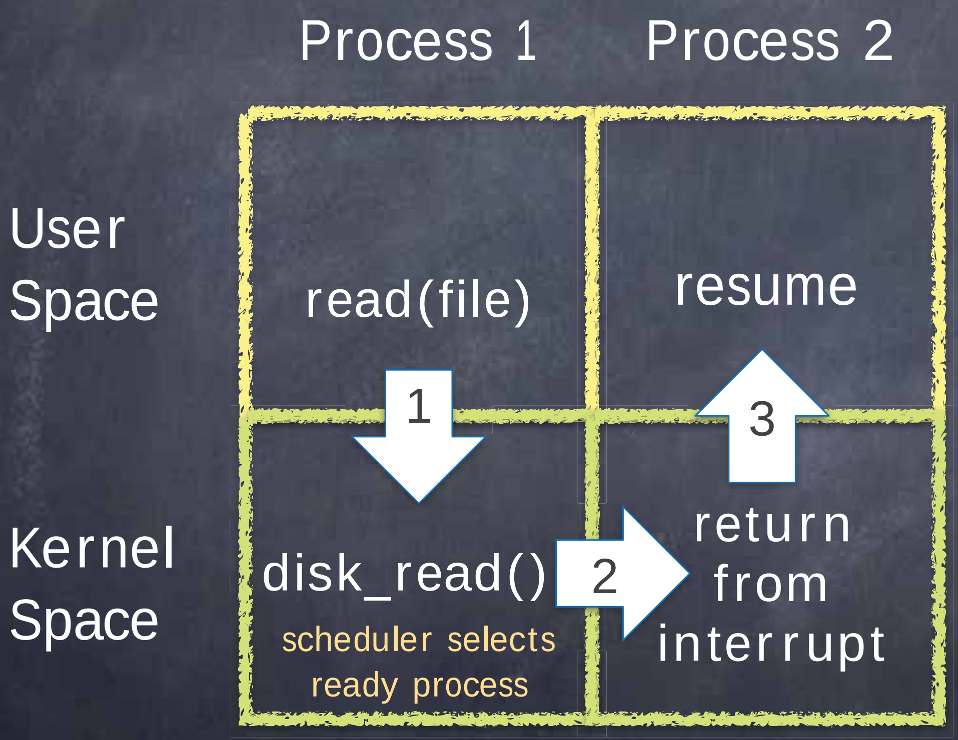
#Processes >> #Processors (cores)

- Solution: time multiplexing
  - Abstractly each processor runs:
    - for ever:
      - NextProcess = scheduler()
      - Copy NextProcess->registers to registers
      - Run for a while
      - Copy registers to NextProcess->registers
  - Scheduler selects process on run queue

Three Flavors of Context Switching

- Interrupt: from user to kernel space
  - on system call, exception, or interrupt
  - Stack switch: P_x user stack $\rightarrow$ P_x interrupt stack
- Yield: between two processes, inside kernel
  - from one PCB/interrupt stack to another
  - Stack switch P_x interrupt stack $\rightarrow$ P_y interrupt stack
- Return from interrupt: from kernel to user space
  - with the homonymous instruction
  - Stack switch: P_x interrupt stack $\rightarrow$ P_x user stack

Switching between Processes



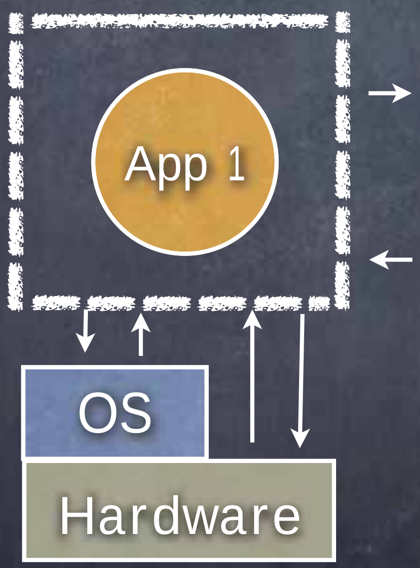
1. Save Process 1 user registers
2. Save Process 1 kernel registers and restore Process 2 kernel registers
3. Restore Process 2 user registers

Threads

An abstraction for concurrency

(Chapters 25-27)

Rethinking the Process Abstraction



- Processes serve two key purposes:
  - defines the granularity at which the OS offers isolation
    - address space identifies what can be touched by the program
  - define the granularity at which the OS offers scheduling and can express concurrency
    - a stream of instructions executed sequentially

Threads: a New Abstraction for Concurrency

- A single-execution stream of instructions that represents a separately schedulable task
  - OS can run, suspend, resume a thread at any time
  - bound to a process (lives in an address space)
  - Finite Progress Axiom: execution proceeds at some unspecified, non-zero speed
- Virtualizes the processor
  - programs run on machine with a seemingly infinite number of processors
- Allows to specify tasks that should be run concurrently…
  - …and lets us code each task sequentially

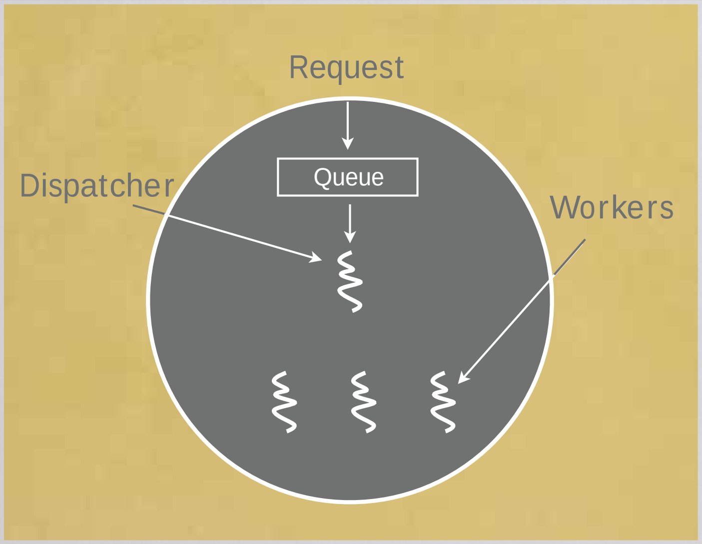
All You Need is Love
(and a stack)

- All threads within a process share
  - heap
  - global/static data
  - libraries
- Each thread has separate
  - program counter
  - registers
  - stack
- Thread stacks are allocated on the heap

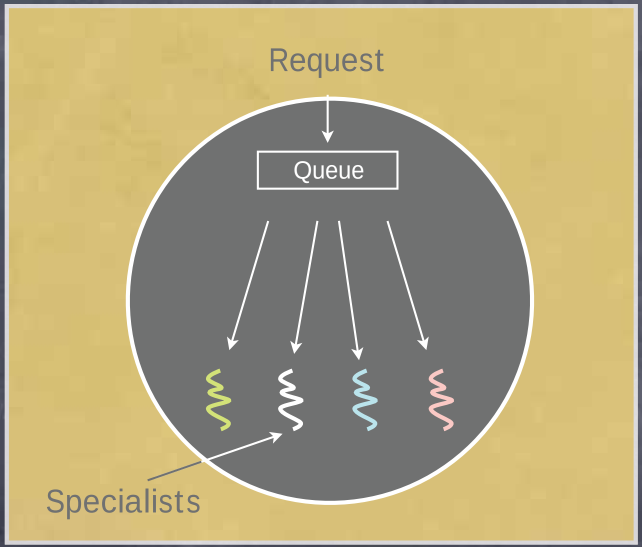
Why Threads?

- To express a natural program structure
  - updating the screen, fetching new data, receiving user input – different tasks within the same address space
- To exploit multiple processors
  - different threads may be mapped to distinct processors
- To maintain responsiveness
  - high priority GUI threads/low priority work threads
- Masking long latency of I/O devices
  - do useful work while waiting

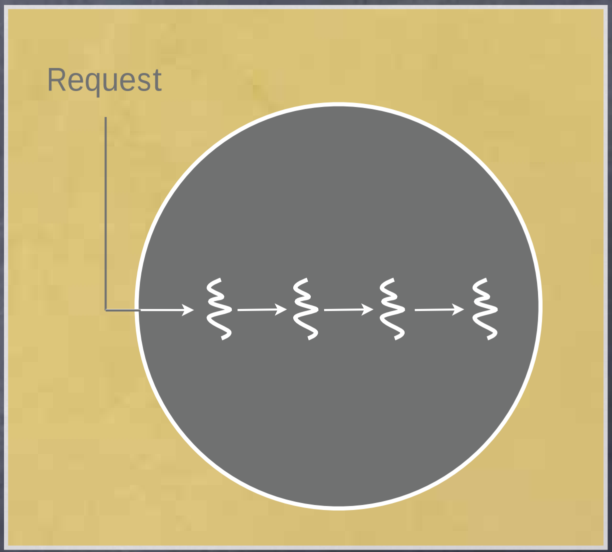
Multithreaded Processing Paradigms



Dispatcher/Workers



Specialists



Pipeline

A simple API

Syscall	Description
void thread_create (thread, func, arg)	Creates a new thread in thread, which will execute function func with arguments arg.
void thread_yield()	Calling thread gives up processor. Scheduler can resume running this thread at any time
int thread_join (thread)	Wait for thread to finish, then return the value thread passed to thread_exit.
void thread_exit (ret)	Finish caller; store ret in caller's TCB and wake up any thread that invoked thread_join(caller).

Preempt or Not Preempt?

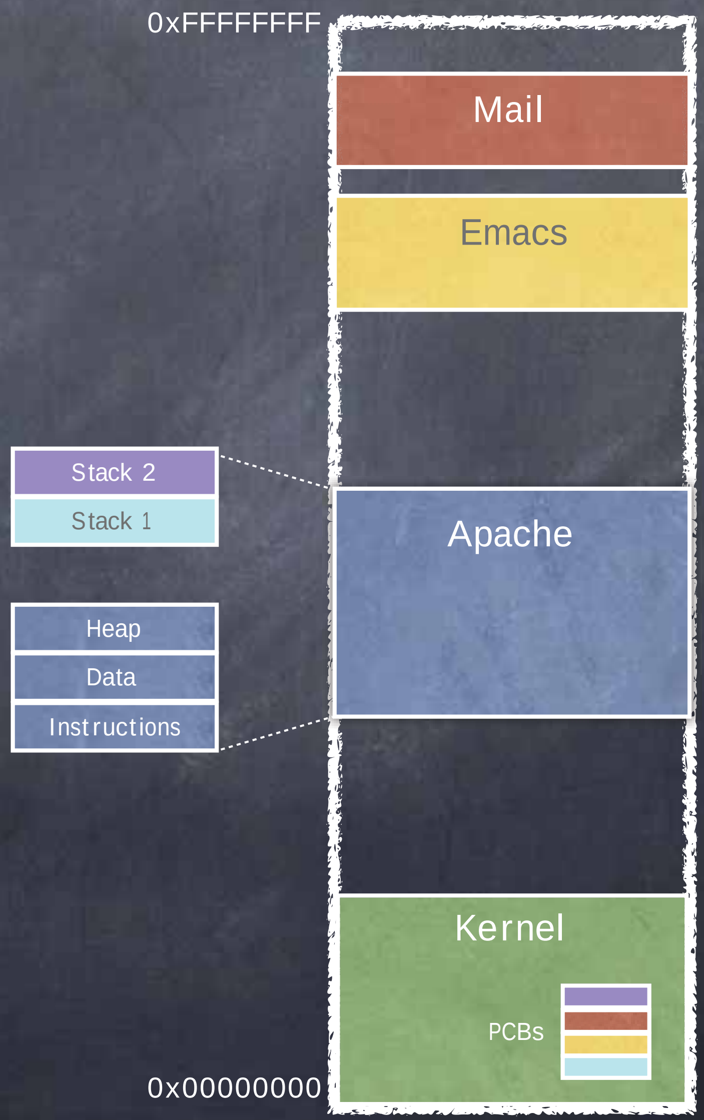
- Preemptive
  - yield automatically upon clock interrupts
  - true of most modern threading systems
- Non-preemptive
  - explicitly yield to pass control to other threads
  - true of CS4411 P1 project

One Abstraction, Two Implementations

- Kernel Threads
  - each thread has its own PCB in the kernel
  - PCBs of threads mapped to the same process point to the same physical memory
  - visible (and schedulable) by kernel
- User Threads
  - one PCB for the process
  - each thread has its own Thread Control Block (TCB) [implemented in the host process’ heap]
  - implemented entirely in user space; invisible to the kernel

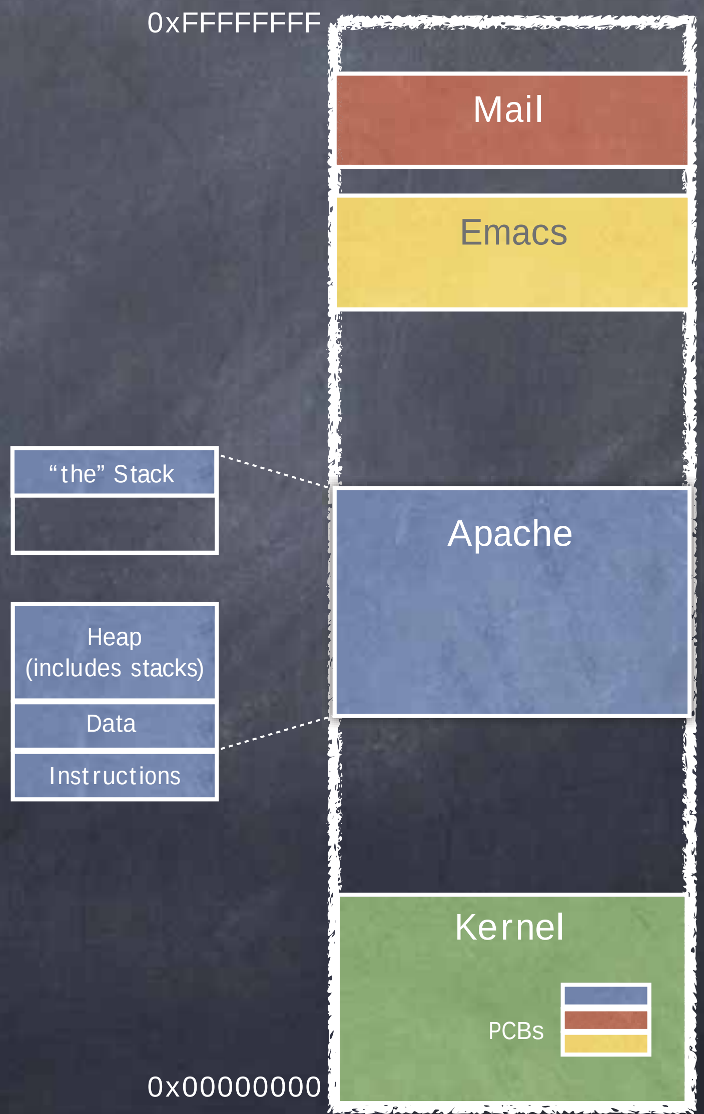
Kernel-level Threads

- Kernel knows about threads existence, and schedules them as it does processes
- Each thread has a separate PCB
- PCBs of threads mapped in the same process have
  - same address space
    - page table base register
  - different PC, SP, registers, interrupt stack



User-level Threads

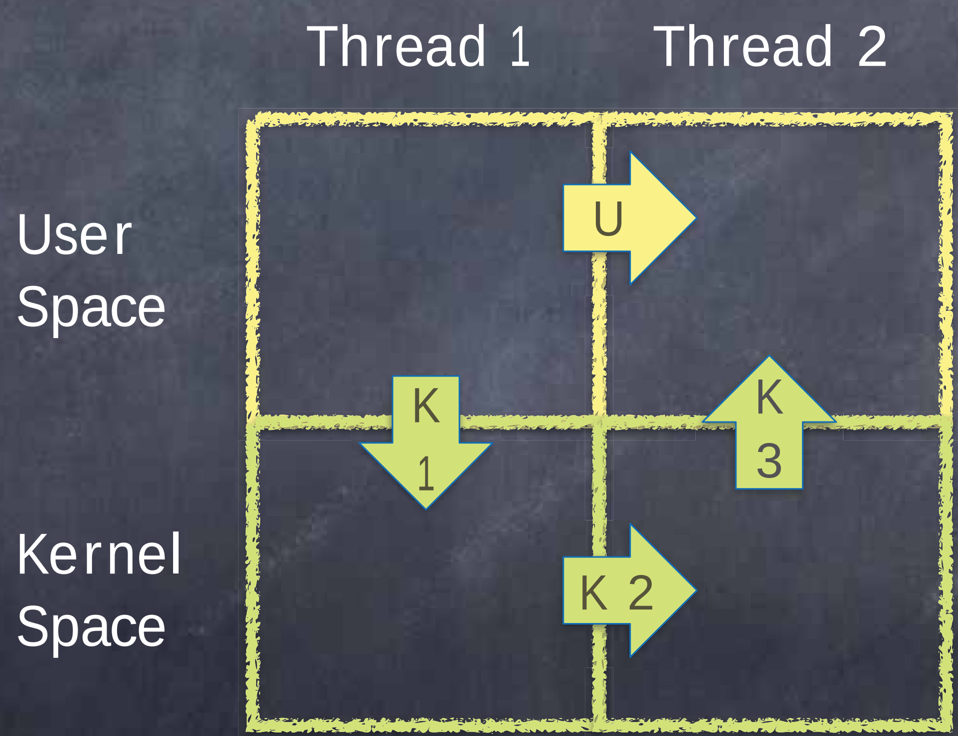
- Run mini-OS in user space
  - real OS is unaware of threads
    - holds a single PCB for all user threads within the same process
  - each thread has associated a Thread Control Block (TCB) kept by process in user space
- User-level threads incur lower overhead than kernel-level threads...
- ...but kernel level threads simplify system call handling and scheduling



Kernel- vs. User-level Threads

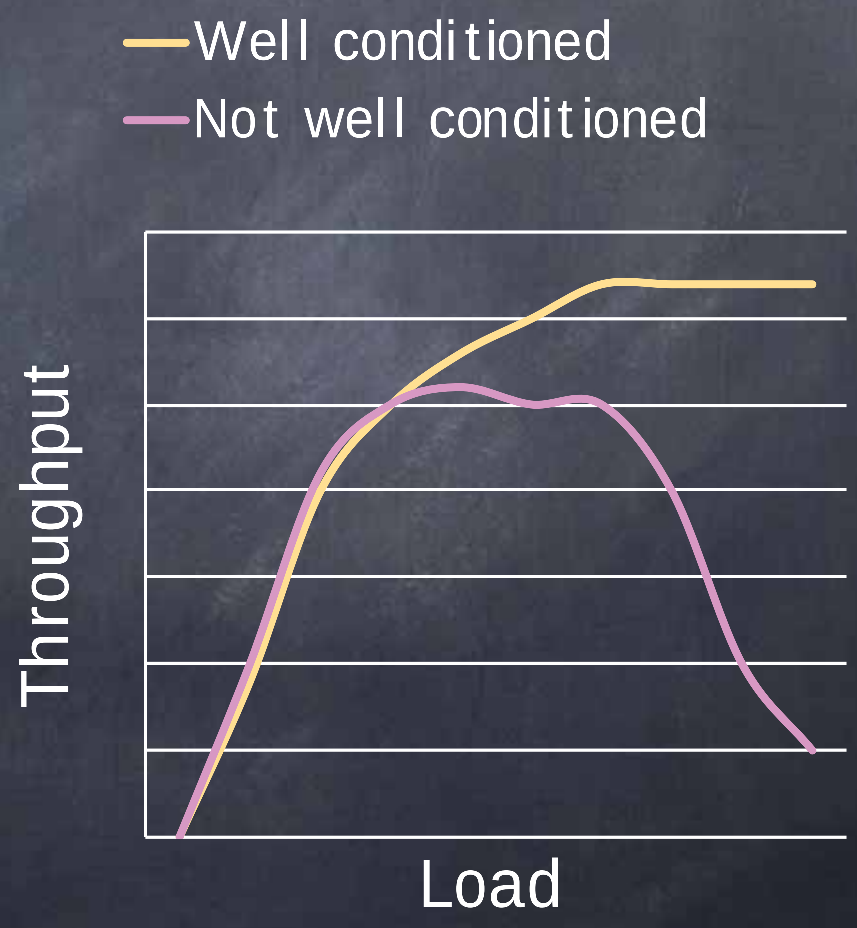
	Kernel-level Threads	User-Level Threads
Ease of implementation	Easy to implement: just like process, but with shared address space	Requires implementing user-level schedule and context switches
Handling system calls	Thread can run blocking systems call concurrently	Blocking system call blocks all threads: needs OS support for non-blocking system calls (scheduler activations)
Cost of context switch	Thread switch requires three context switches	Thread switch efficiently implemented in user space

Kernel- vs. User-level Thread Switching



Threads considered harmful

- Creating a thread or process for each unit of work (e.g., user request) is dangerous
  - High overhead to create & delete thread/process
  - Can exhaust CPU & memory resource
- Thread/process pool controls resource use
  - Allows service to be well conditioned
    - output rate scales to input rate up to saturation
    - excessive demand does not degrade pipeline throughput



Threads

vs

Event-Based Programming

Event-based Programming

- Main loop listens for events; when detected executes corresponding function
- No “blocking” operations
  - No read(), wait(), lock(), etc.
  - I/O is asynchronous
- Code is a collection of event handlers
  - (Similar to I/O interrupt handlers)
  - Invoked when some event happens
  - Run to completion
    - Remember, no blocking operations

Event-Based Web Server

```
handler client_request(client, URI):
    contents := CACHE[URI];
    if contents != None:
        send(client, contents);
    else:
        if PENDING[URI] == { }:
            start_load_file(URI, file_loaded_handler);
        PENDING[URI] ∪= {client };


handler file_loaded (URI, contents):
    CACHE[URI] := contents;
    for each client in PENDING[URI]:
        send(client, contents);
    PENDING[URI] = { };
```

Thread-based Web Server

```
thread client_handler():
    for ever:
        (client, URI) = receive();          # blocks
        CACHE.lock();                       # may block
        while CACHE[URI] == None:
            NEEDED.lock(); NEEDED ∪= {URI};
            NEEDED.notify(); NEEDED.unlock();
            CACHE.wait();                   # blocks
        send(client, CACHE[URI]);
        CACHE.unlock();
```

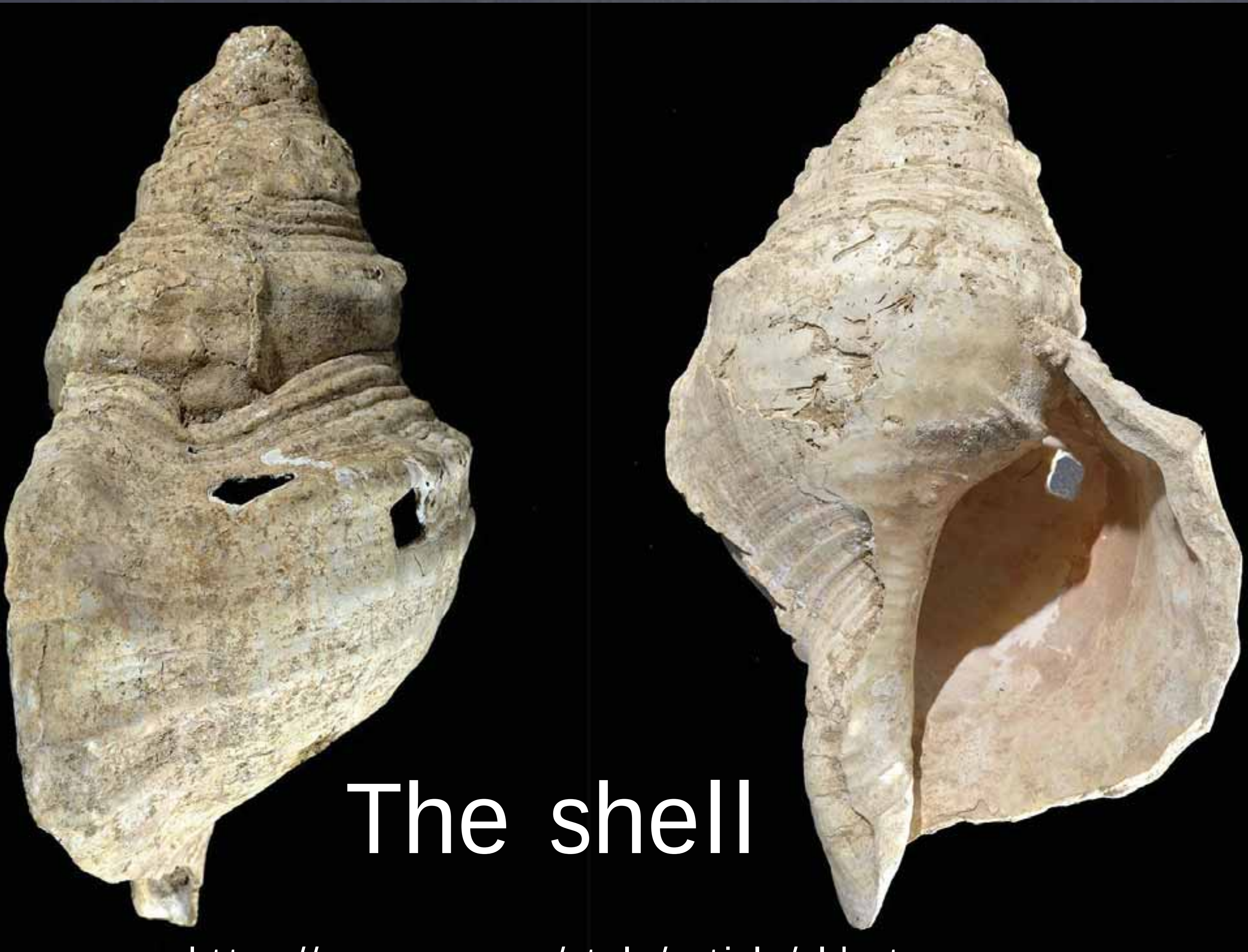
```
thread file_loader(URI, contents):
    for ever:
        NEEDED.lock();                          # may block
        while NEEDED == { }: NEEDED.wait();     # blocks
        uris = NEEDED; NEEDED = { };
        NEEDED.unlock();
        for each URI in uris:
            contents = read(URI);               # blocks
            CACHE.lock(); CACHE[URI] = contents;
            CACHE.notifyAll(); CACHE.unlock();
```

Decades-Old Debate...

- Example debate papers
  - 1995: *Why Threads are a Bad Idea (for most purposes)*
    - J. Ousterhout (UC Berkeley, Sun Labs, now at Stanford)
  - 2003: *Why Events are a Bad Idea (for high-concurrency servers)*
    - R. van Behren, J. Condit, E. Brewer (UC Berkeley)
- But also known to be logically equivalent:
  - 1978, *On the Duality of Operating Systems Structures*
    - H.C. Lauer, R.M. Needham

How They Compare

Event-Based	Thread-Based
good for I/O-parallelism/GUIs	good for any parallelism
no context switch overhead (contexts are short-lived)	keeps track of control flow
does not need locks	needs locks
code becomes spaghetti	code relatively easy to read
deterministic; easy to debug	hard to debug (Harmony to the rescue!)



The shell

https://www.cnn.com/style/article/oldest-wind-instrument-scli-intl-scn/index.html

What is a shell?

An interpreter

- Runs programs on behalf of the user
- Allows programmer to create/manage set of programs
  - sh Original Unix shell (Bourne, 1977)
  - csh BSD Unix C shell (tcsh enhances it)
  - bash "Bourne again" shell
- Every command typed in the shell starts a child process of the shell
- Runs at user-level. Uses syscalls: fork, exec, etc.

The Unix shell (simplified)

```
while(! EOF)
read input
handle regular expressions
int pid = fork()    // create child
if (pid == 0) { // child here
  exec("program", argc, argv0,...);
}
else {  // parent here
...
}
```

Some important commands

- echo [args] # prints args
- pwd # prints working directory
- ls # lists current directory
- cd [dir] # change current directory
- ps # lists your running processes

Commands can be modified with flags

- ls -l # long list of current directory
- ps -a # lists all running processes

Foreground vs Background

- The shell is either
  - reading from standard input or
  - waiting for a process to finish
    - this is the foreground process
    - other processes are background processes
- To start a background process, add &
  - (sleep 5; echo hello) &
  - x & y # runs x in background and y in foreground

Pipes

- x | y
  - runs both x and y in foreground
  - output of x is input to y
  - finishes when both x and y are finished

echo Lorenzo | tr r b | tr n r | tr z t | tr L R