How to run multiple processes

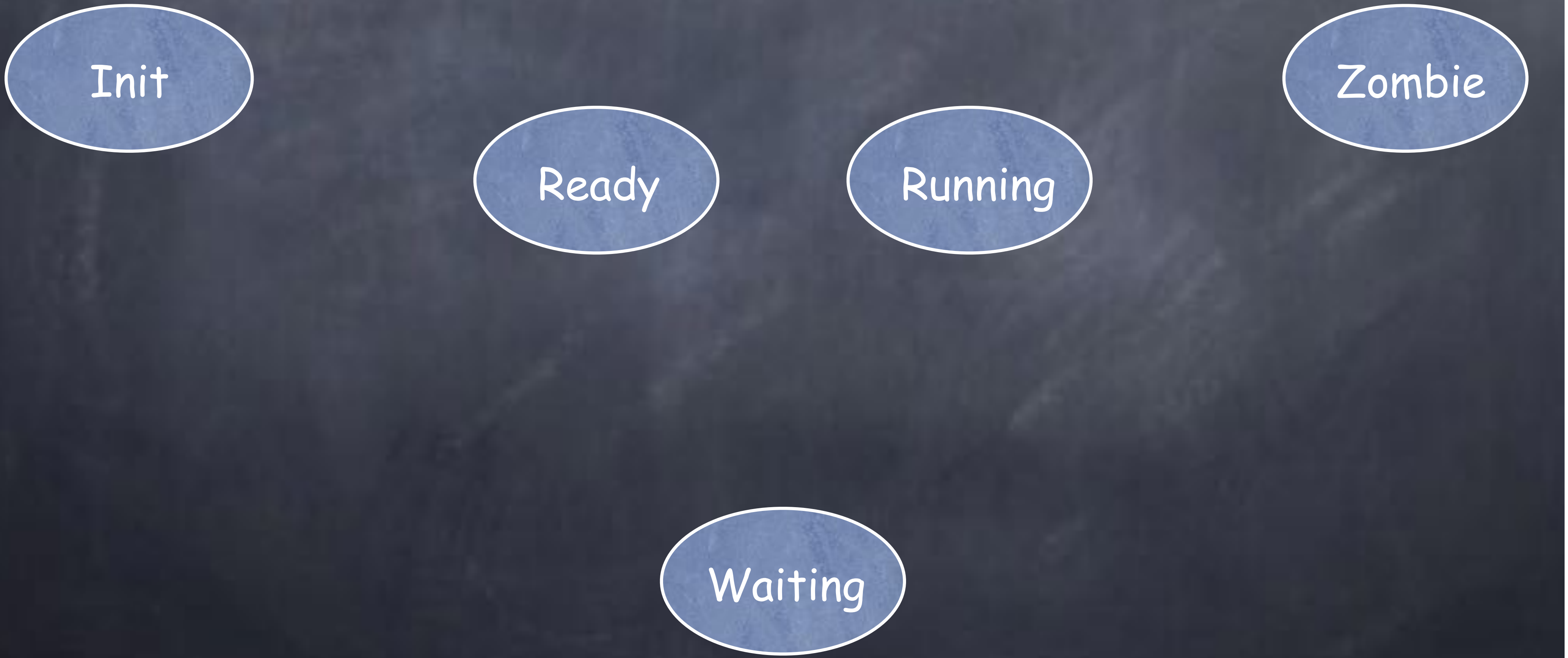
The Problem

- Say (for simplicity) we have a single core CPU
- A process physically runs on the CPU
- Yet each process somehow has its own
  - Registers
  - Memory
  - I/O Resources
- Need to multiplex/schedule to create virtual CPUs for each process

Our friend, the Process Control Block

- A per-process data structure held by OS, with
  - location in memory (page table)
  - location of executable on disk
  - id of user executing this process (uid)
  - process identifier (pid)
  - process status (running, waiting, etc.)
  - scheduling info
  - interrupt stack
  - saved kernel SP (when process is not running)
    - points into interrupt stack
    - interrupt stack contains saved registers and kernel call stack for this process
  - ...and more

Process Life Cycle



Init

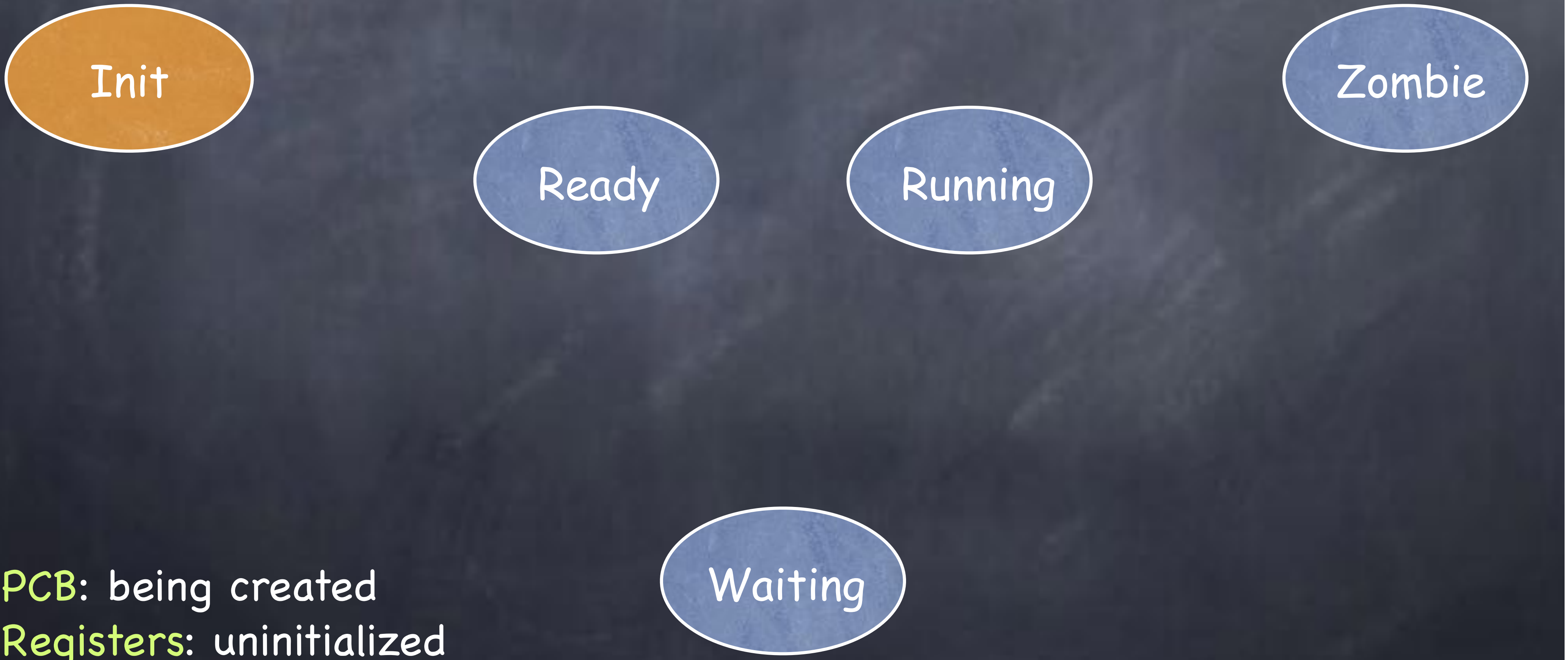
Ready

Running

Zombie

Waiting

Process Life Cycle



Init

Ready

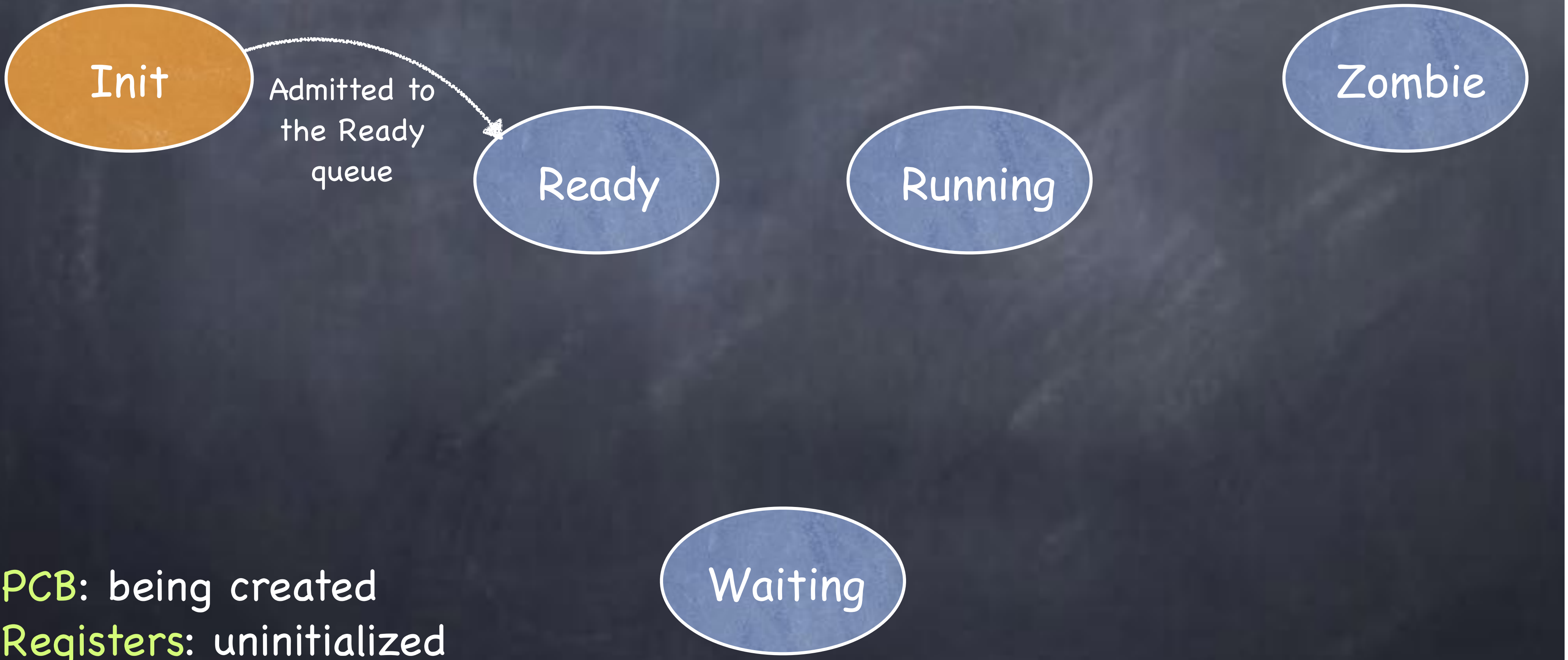
Running

Zombie

Waiting

PCB: being created
Registers: uninitialized

Process Life Cycle



Init

Admitted to the Ready queue

Ready

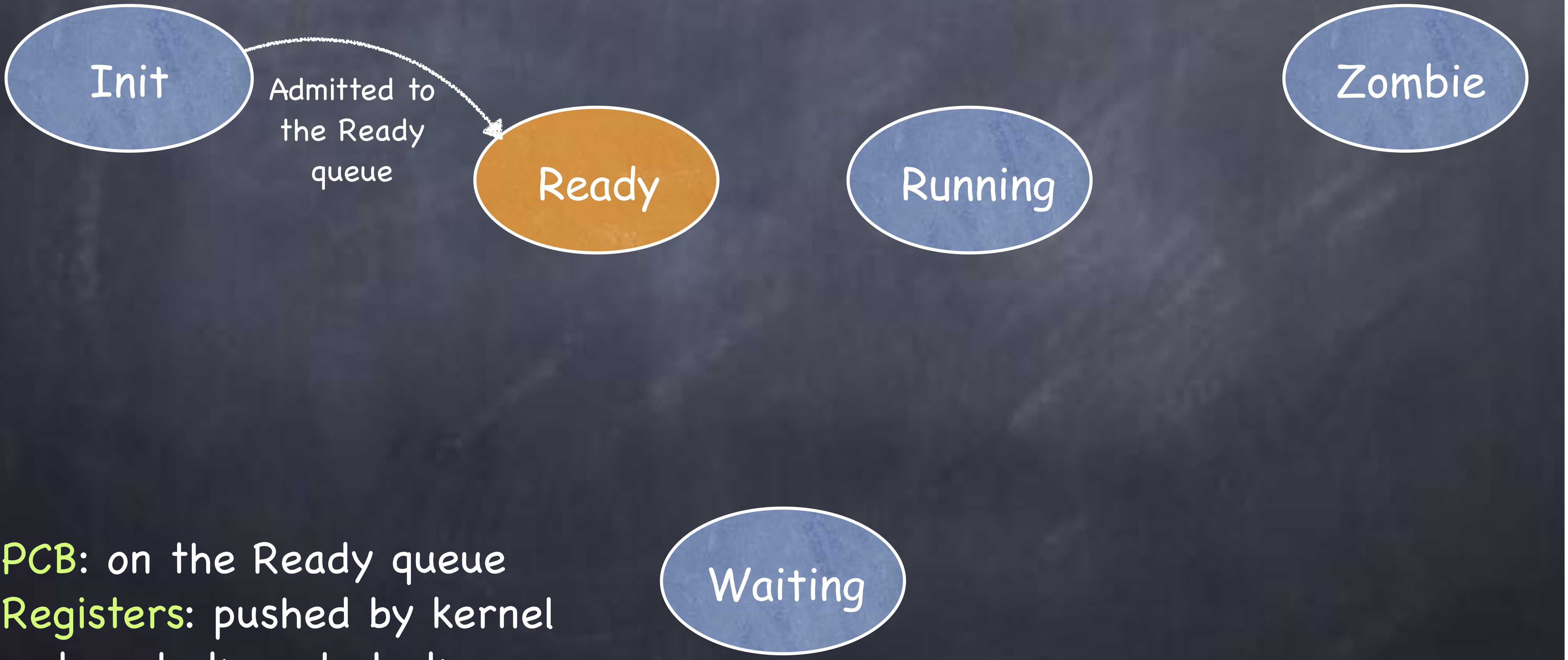
Running

Zombie

Waiting

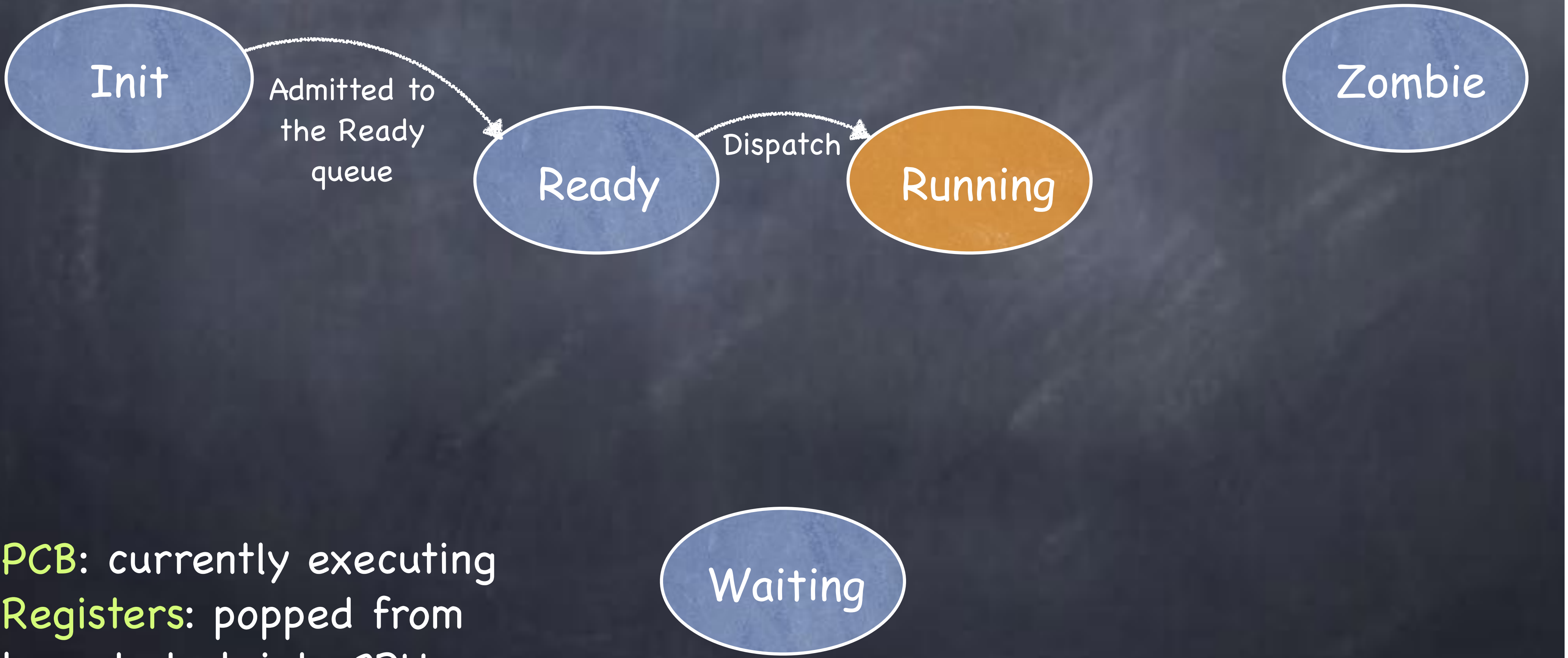
PCB: being created
Registers: uninitialized

Process Life Cycle



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

Process Life Cycle



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

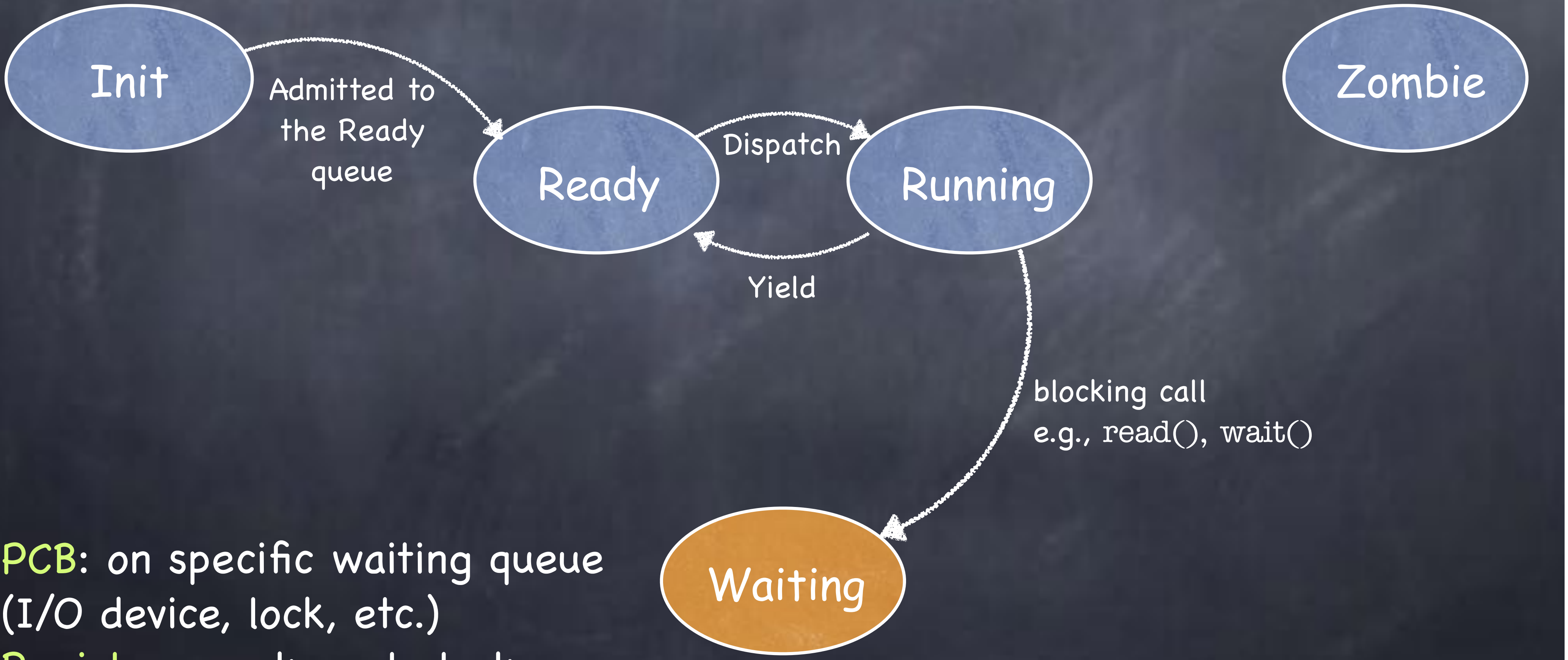
Process Life Cycle

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

Process Life Cycle

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

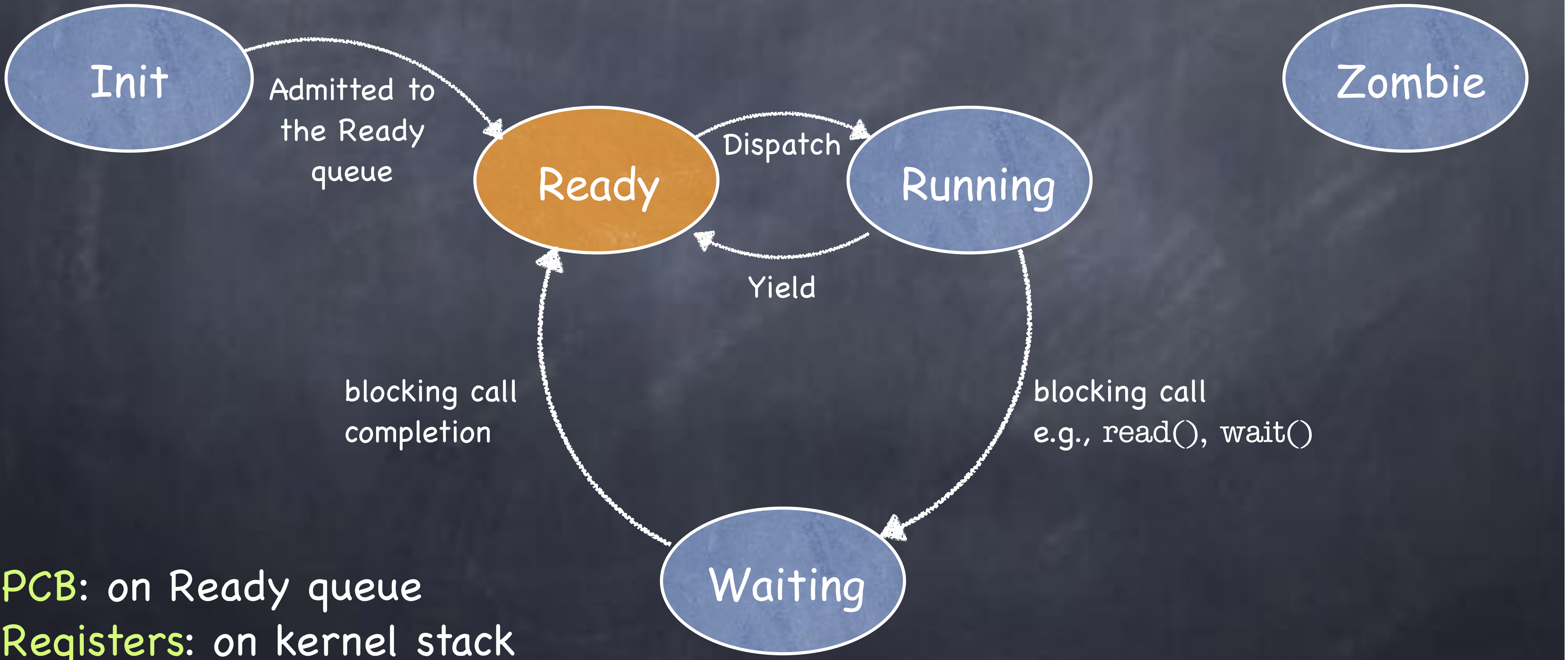
Process Life Cycle



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

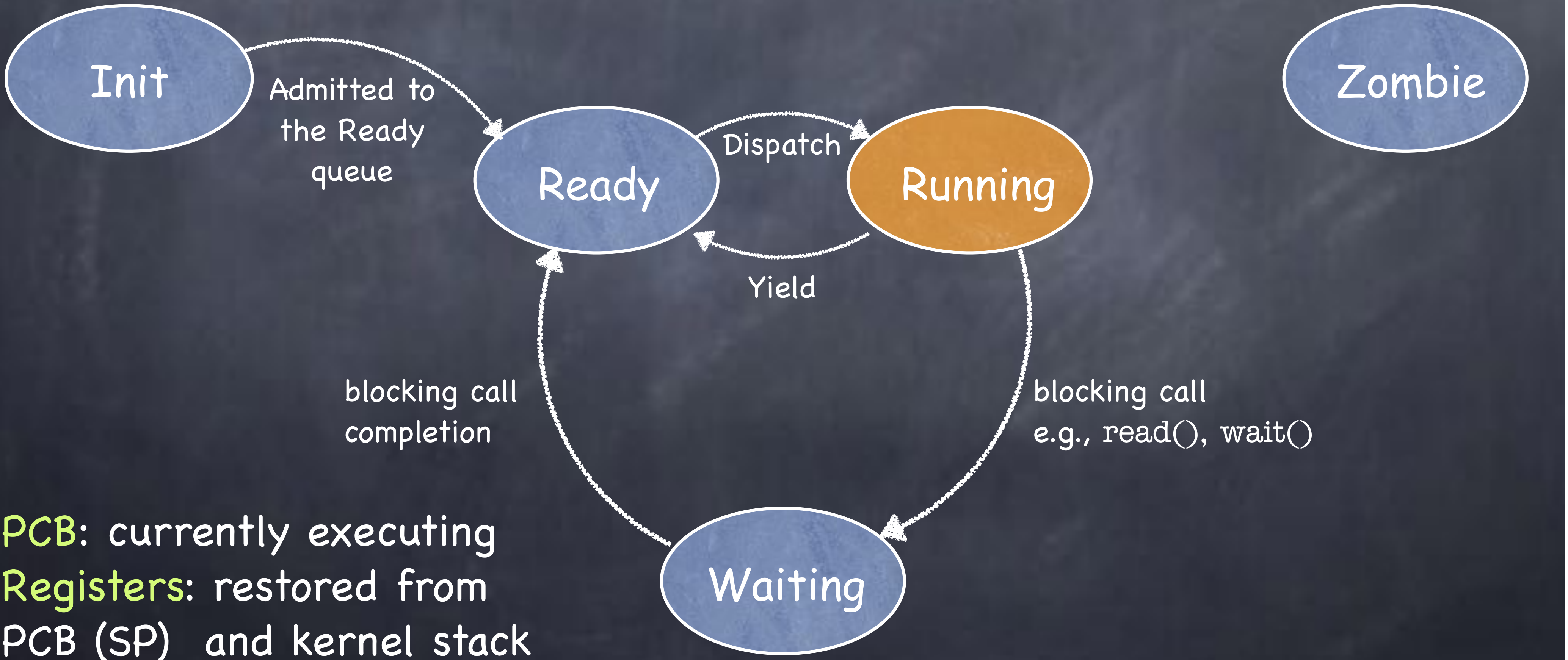
Registers: on kernel stack

Process Life Cycle



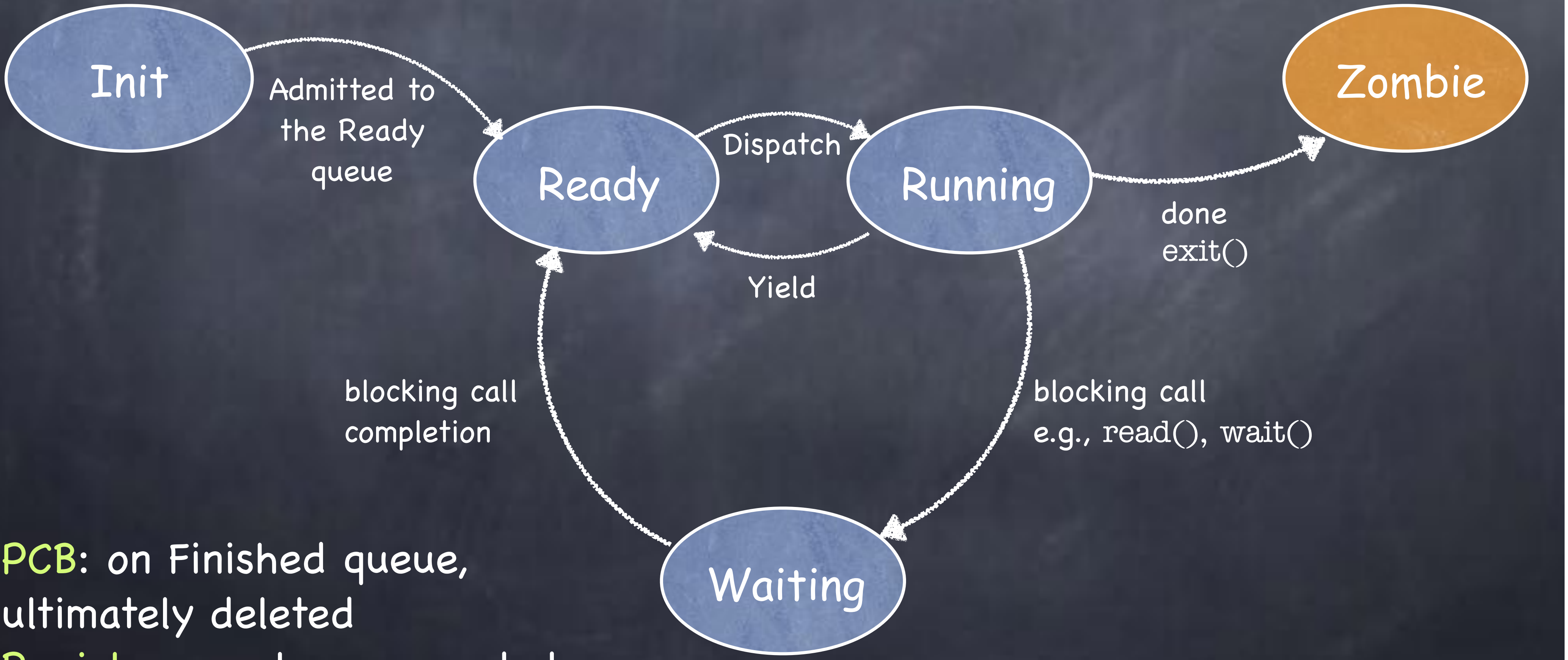
PCB: on Ready queue
Registers: on kernel stack

Process Life Cycle



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

Process Life Cycle



PCB: on Finished queue, ultimately deleted

Registers: no longer needed

Invariants to keep in mind

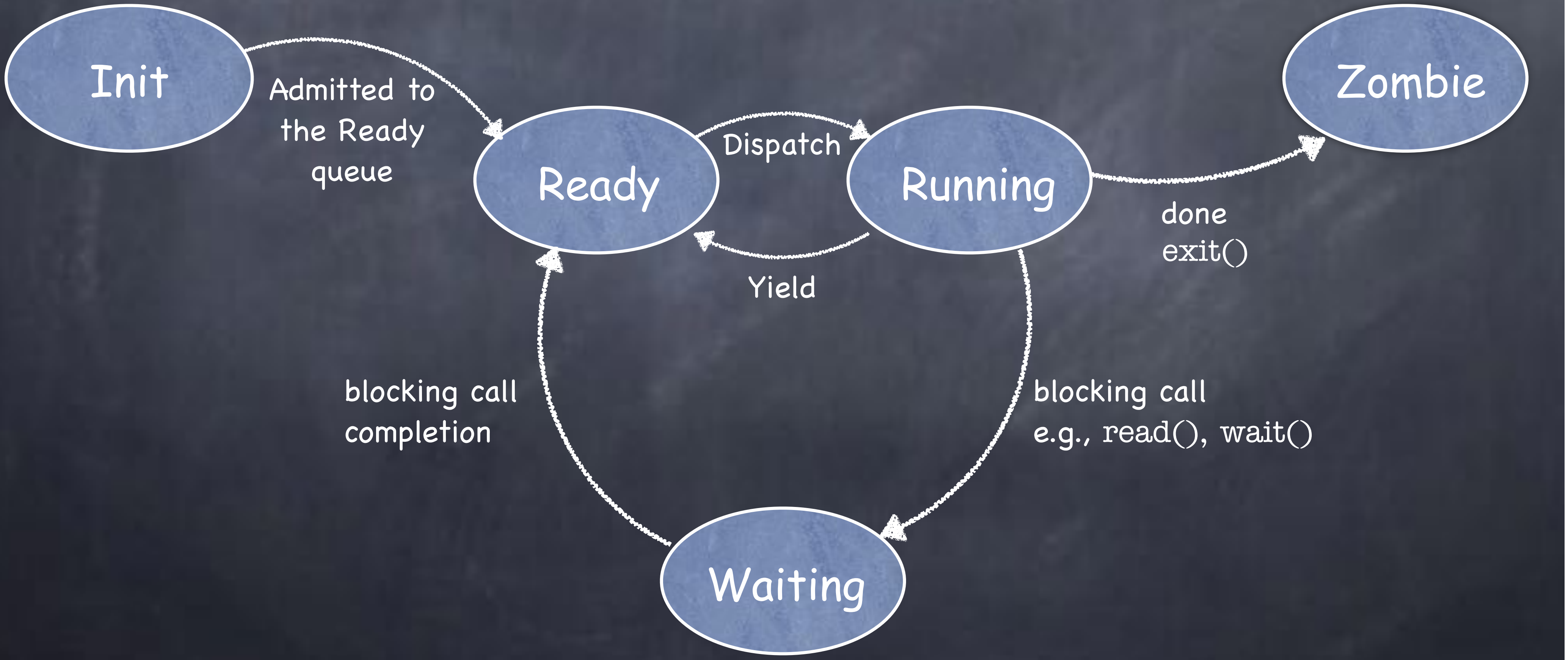
- At most one process/core running at any time
- When CPU in user mode, current process is RUNNING and its kernel stack is empty
- If process is RUNNING
  - its PCB not on any queue
  - it is not necessarily in USER mode
- If process is READY or WAITING
  - its registers are saved at the top of its kernel/interrupt 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
  - hard to clean up and switch without a stack!
- 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



Init

Admitted to the Ready queue

Ready

Dispatch

Running

Yield

Zombie

done
exit()

blocking call
completion

blocking call
e.g., read(), wait()

Waiting

How to Yield/Wait?

- Must switch the "CPU state" (the context) captured in its registers and PSW
- Must switch from executing the current process to executing some other READY process
  - ☐ Current process: RUNNING → READY
  - ☐ Next process: READY → RUNNING

1. Save kernel registers of Current on its kernel 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 kernel 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;
}
```

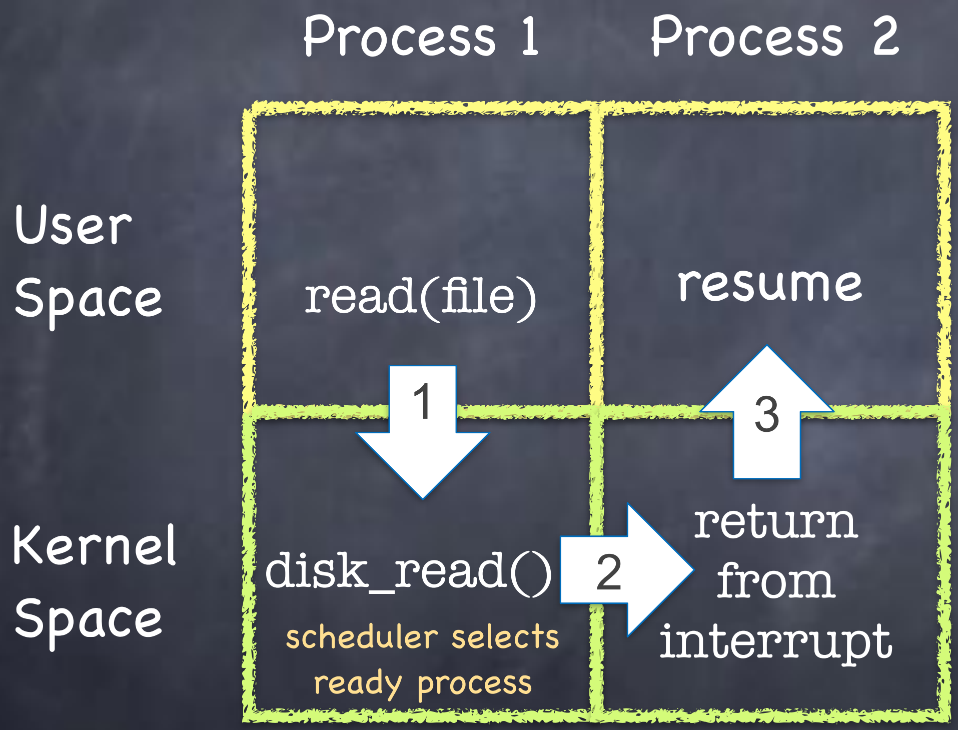
Anybody there?

- 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 → P_x interrupt stack
- Yield: between two processes, inside kernel
  - from one PCB/interrupt stack to another
  - Stack switch: P_x interrupt stack → P_y interrupt stack
- Return from interrupt: from kernel to user space
  - with the homonymous instruction
  - Stack switch: P_x interrupt stack → 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
    - returns twice, (!), to both the parent and the child process, setting pid to different values
  - 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"