CS 170 Week 6

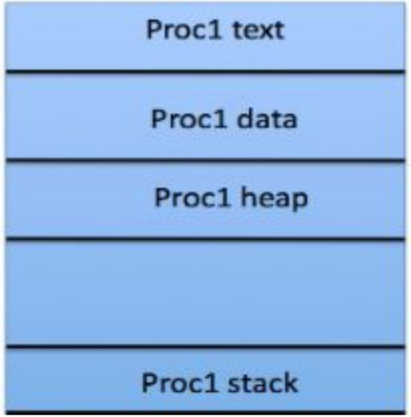
Winter 2026

https://sites.cs.ucsb.edu/~rich/class/cs170/

Lab 2 Topic: Parent-Child Processes

Creating new processes: fork (part 1)

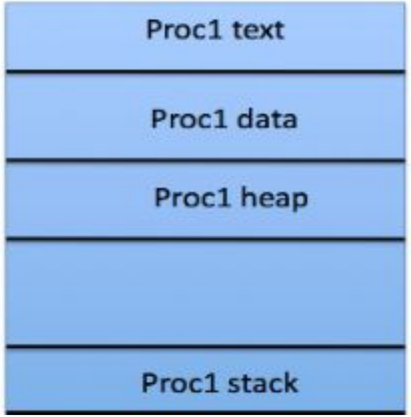
Parent



Register Values
(and more)

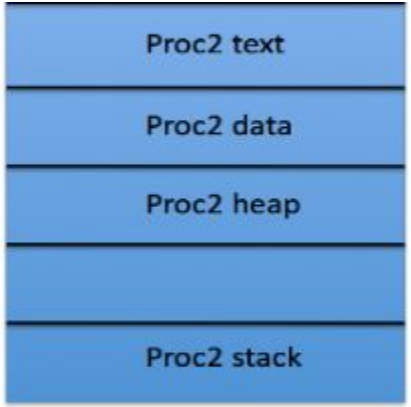
Creating new processes: fork (part 1)

Parent



Register Values
(and more)

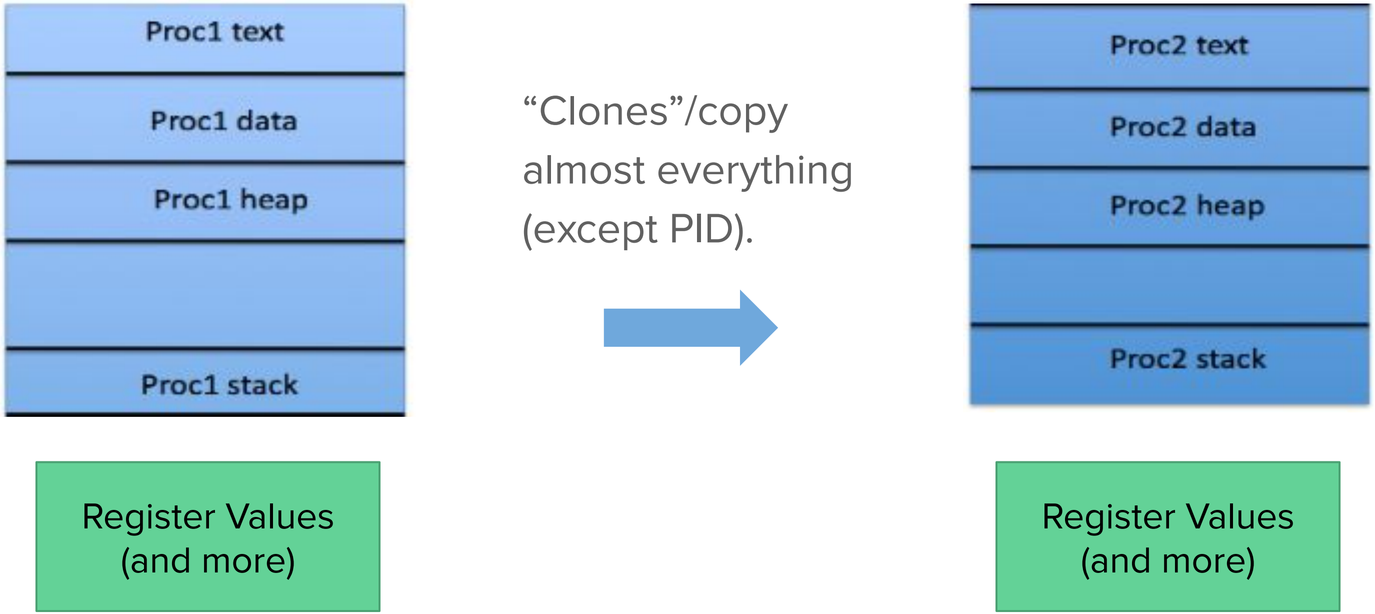
Child



Register Values
(and more)

NOTE: Unix/Unix-like systems use this mechanism (i.e. macOS/OS X, Linux, and KOS). Windows do something different.

Creating new processes: fork (part 1)



Parent

Child

“Clones”/copy almost everything (except PID).

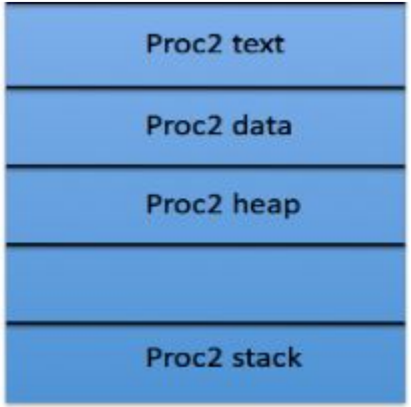
Register Values (and more)

Register Values (and more)

NOTE: Unix/Unix-like systems use this mechanism (i.e. macOS/OS X, Linux, and KOS). Windows do something different.

Creating new processes: exec (optional part 2)

If the intent is to run a *different* program, **exec(ve)** is called after forking.

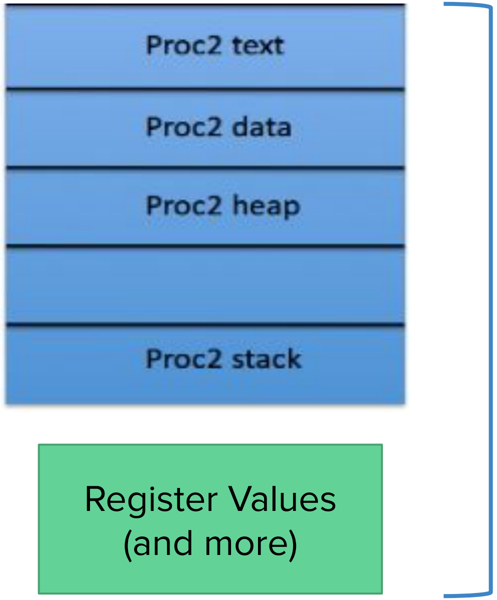


Register Values
(and more)

NOTE: Unix/Unix-like systems use this mechanism (i.e. macOS/OS X, Linux, and KOS). Windows do something different.

Creating new processes: exec (optional part 2)

If the intent is to run a *different* program, **exec(ve)** is called after forking.

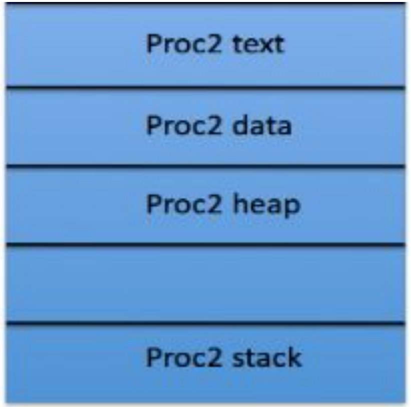


“Everything” is cleared and the program text is replaced by the new program. PID is preserved.

NOTE: Unix/Unix-like systems use this mechanism (i.e. macOS/OS X, Linux, and KOS). Windows do something different.

Creating new processes: exec (optional part 2)

If the intent is to run a *different* program, **exec(ve)** is called ~~after forking~~.



Register Values
(and more)

exec does not require forking to work. You can call exec by itself!

NOTE: Unix/Unix-like systems use this mechanism (i.e. macOS/OS X, Linux, and KOS). Windows do something different.

Parent-child relationship

Because of the fork/exec mechanism, parent-child processes form a “hierarchical tree.”

Parent-child relationship

Because of the fork/exec mechanism, parent-child processes form a “hierarchical tree.”

This is required for the wait syscall. More on this later (or next week)

Parent-child relationship

Because of the fork/exec mechanism, parent-child processes form a “hierarchical tree.”

This is required for the wait syscall. More on this later (or next week)

Question: If the only way a process is created is by a fork (and execve), where does the first process come from? 🤔

Parent-child relationship

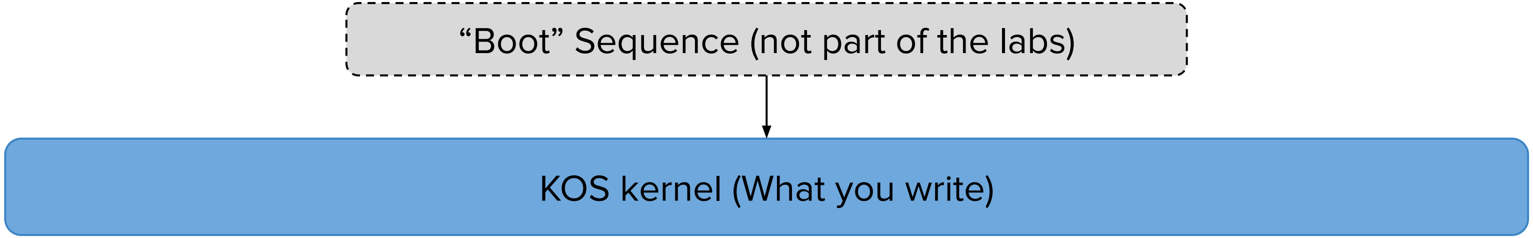
Because of the fork/exec mechanism, parent-child processes form a “hierarchical tree.”

This is required for the wait syscall. More on this later (or next week)

Question: If the only way a process is created is by a fork (and execve), where does the first process come from? 🤔

Answer: The first process (typically PID 1) is spawned by the OS kernel. In KOS, the first process is determined by the “-a” flag.

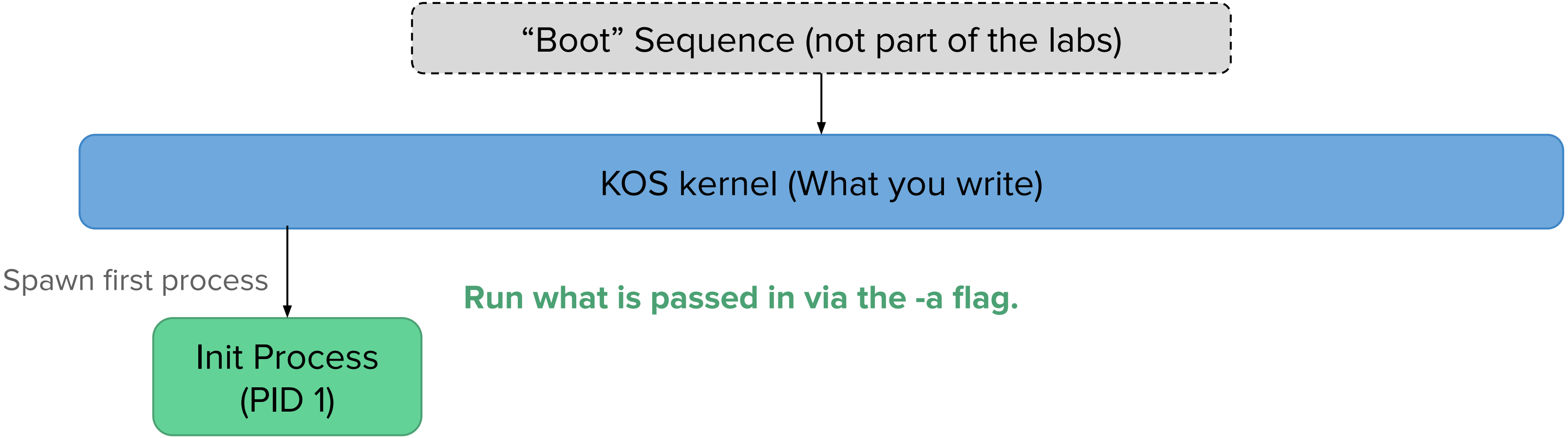
Example Execution Flow



“Boot” Sequence (not part of the labs)

KOS kernel (What you write)

Example Execution Flow



“Boot” Sequence (not part of the labs)

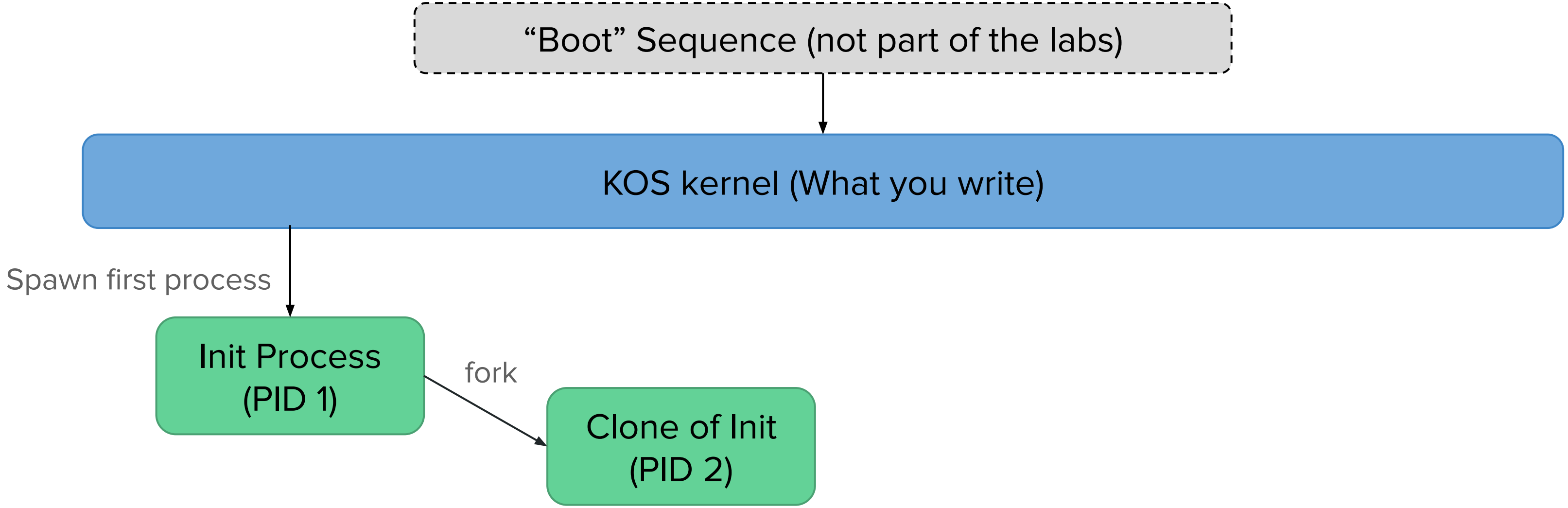
KOS kernel (What you write)

Spawn first process

Init Process (PID 1)

Run what is passed in via the -a flag.

Example Execution Flow



“Boot” Sequence (not part of the labs)

KOS kernel (What you write)

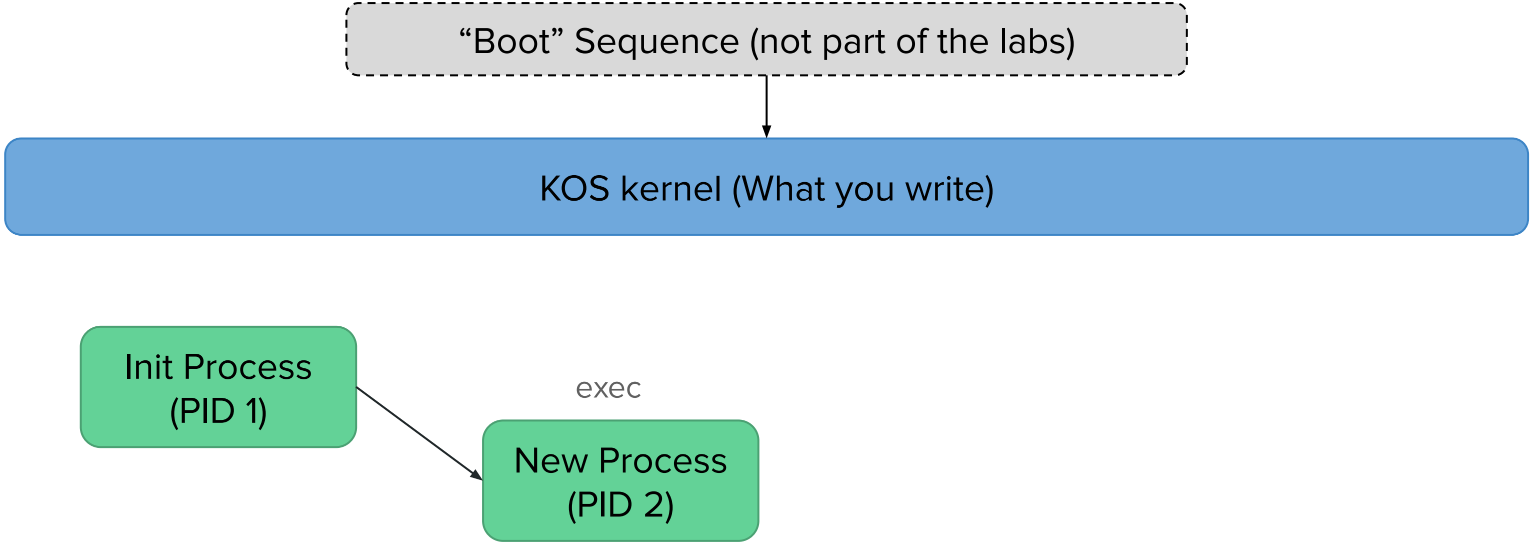
Spawn first process

Init Process (PID 1)

fork

Clone of Init (PID 2)

Example Execution Flow



“Boot” Sequence (not part of the labs)

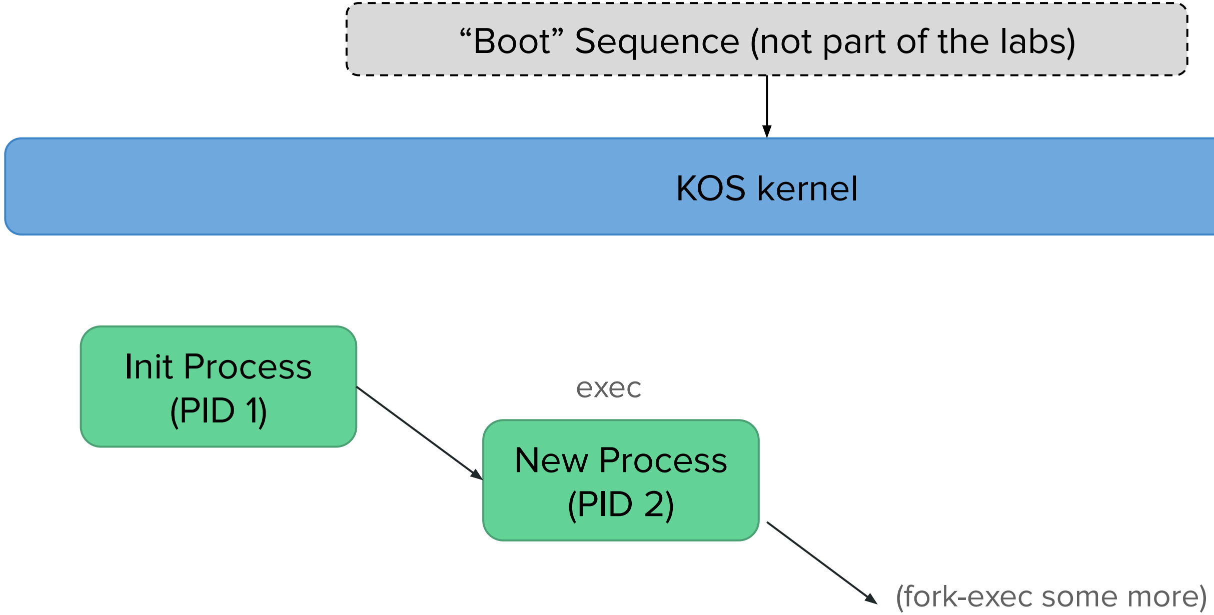
KOS kernel (What you write)

Init Process (PID 1)

exec

New Process (PID 2)

Example Execution Flow



“Boot” Sequence (not part of the labs)

KOS kernel

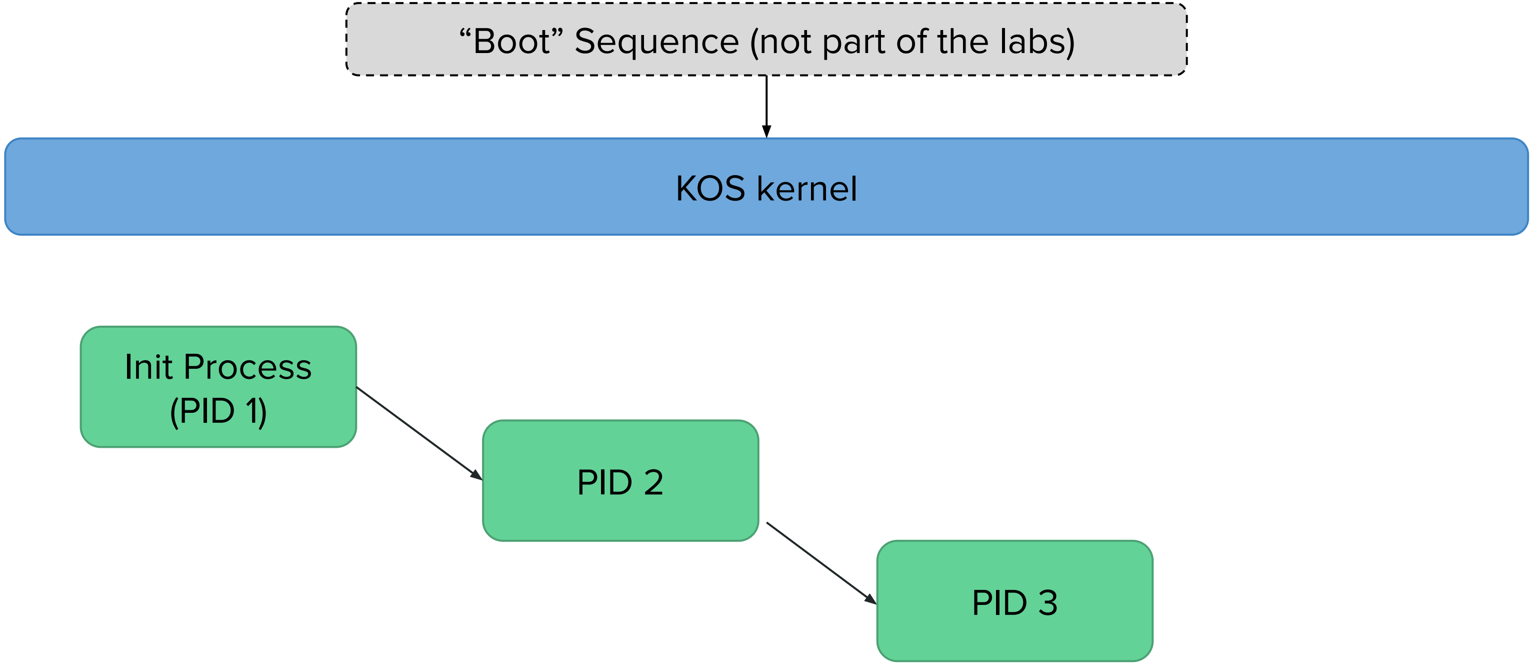
Init Process
(PID 1)

exec

New Process
(PID 2)

(fork-exec some more)

Example Execution Flow



“Boot” Sequence (not part of the labs)

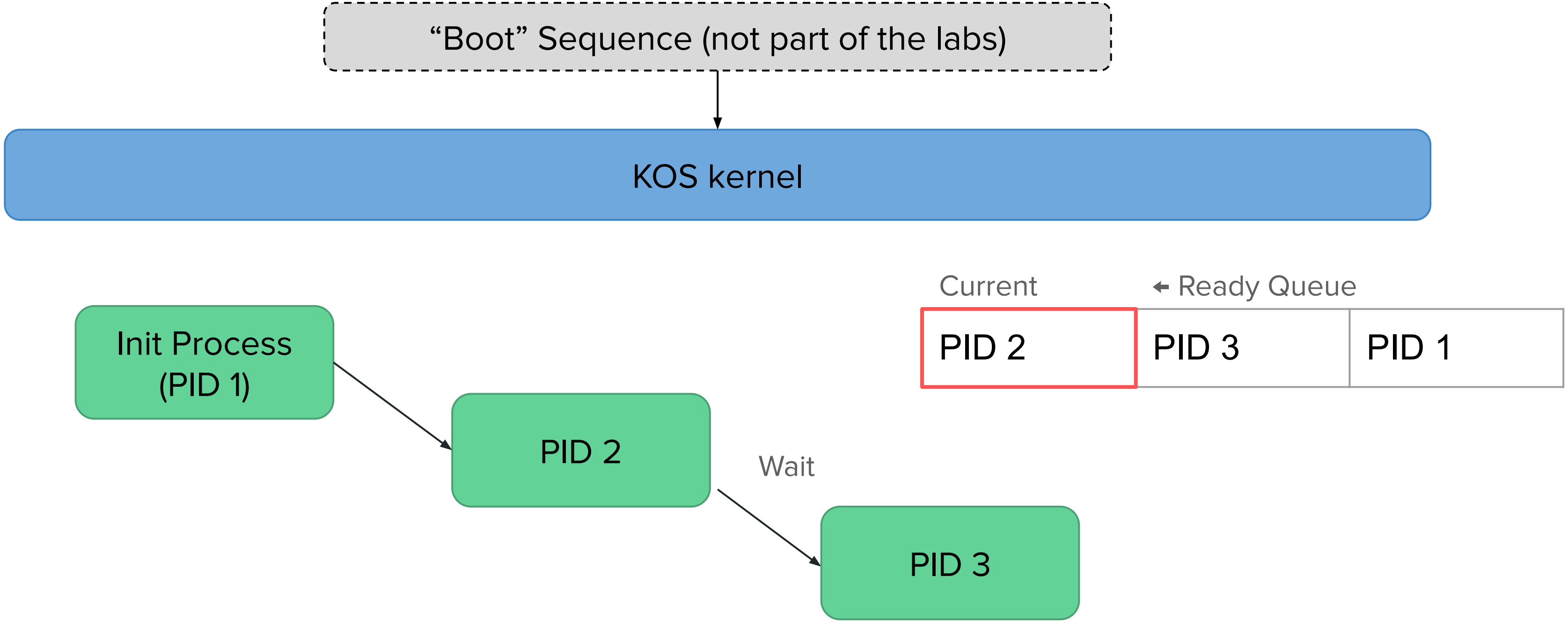
KOS kernel

Init Process
(PID 1)

PID 2

PID 3

Scenario 1: Parent (PID 2) Waits For Child (PID 3)



“Boot” Sequence (not part of the labs)

KOS kernel

Current

← Ready Queue

PID 2	PID 3	PID 1

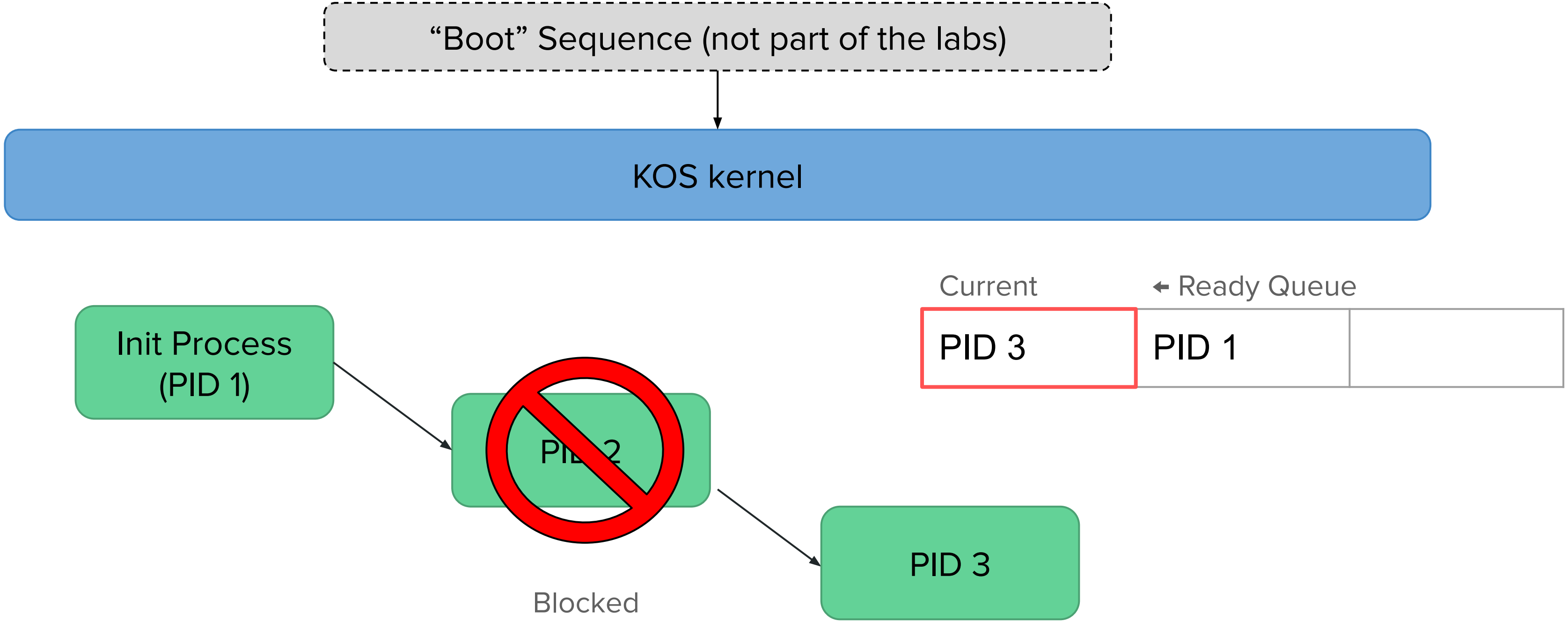
Init Process (PID 1)

PID 2

Wait

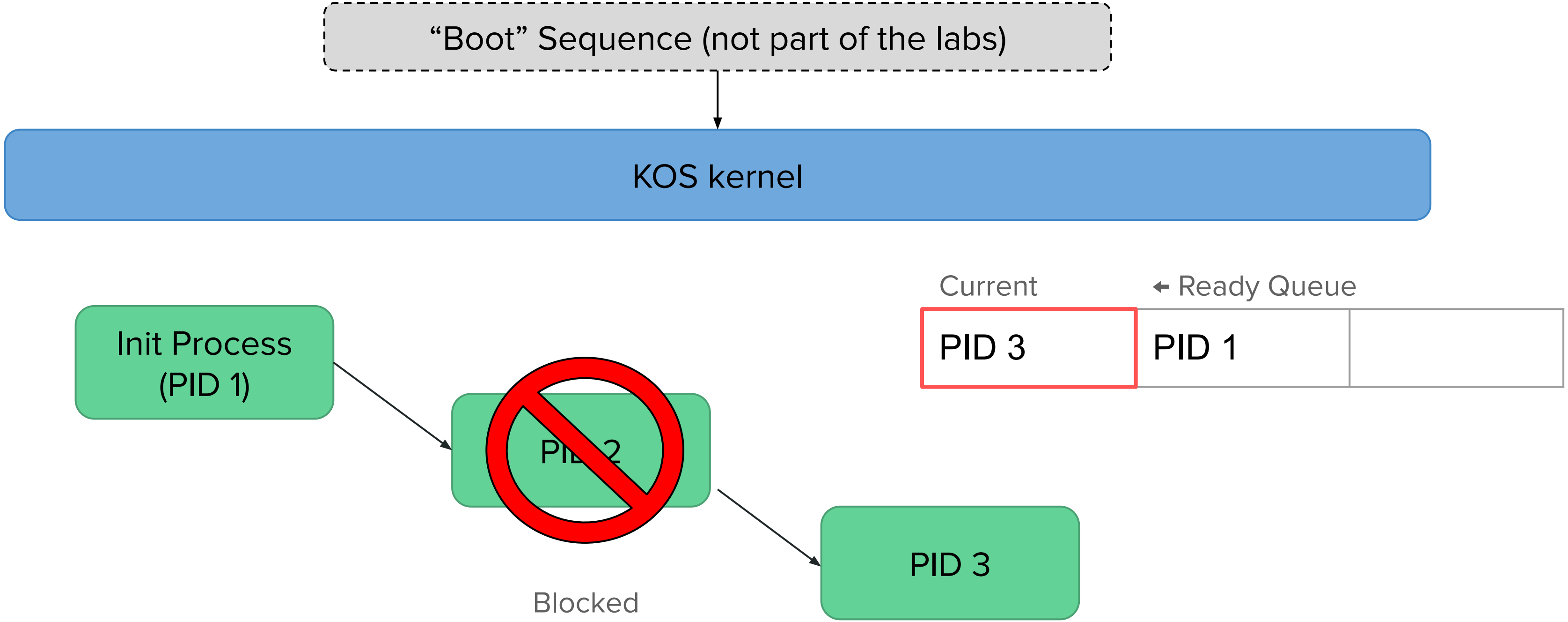
PID 3

Scenario 1: Parent (PID 2) Waits For Child (PID 3)



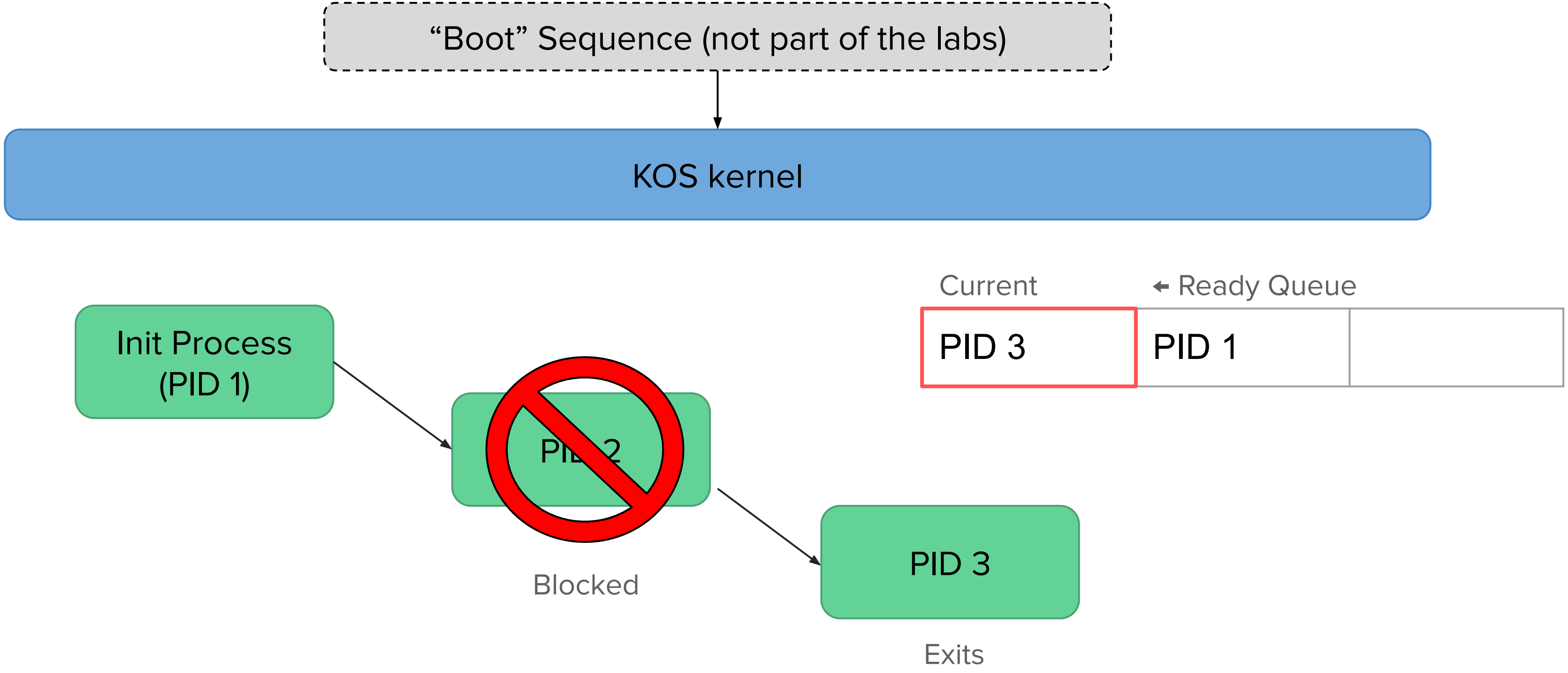
Current	← Ready Queue	
PID 3	PID 1	

Scenario 1: Parent (PID 2) Waits For Child (PID 3)



Current	← Ready Queue	
PID 3	PID 1	

Scenario 2: Child (PID 3) Exit When Parent is Waiting



“Boot” Sequence (not part of the labs)

KOS kernel

Init Process (PID 1)

PID 2

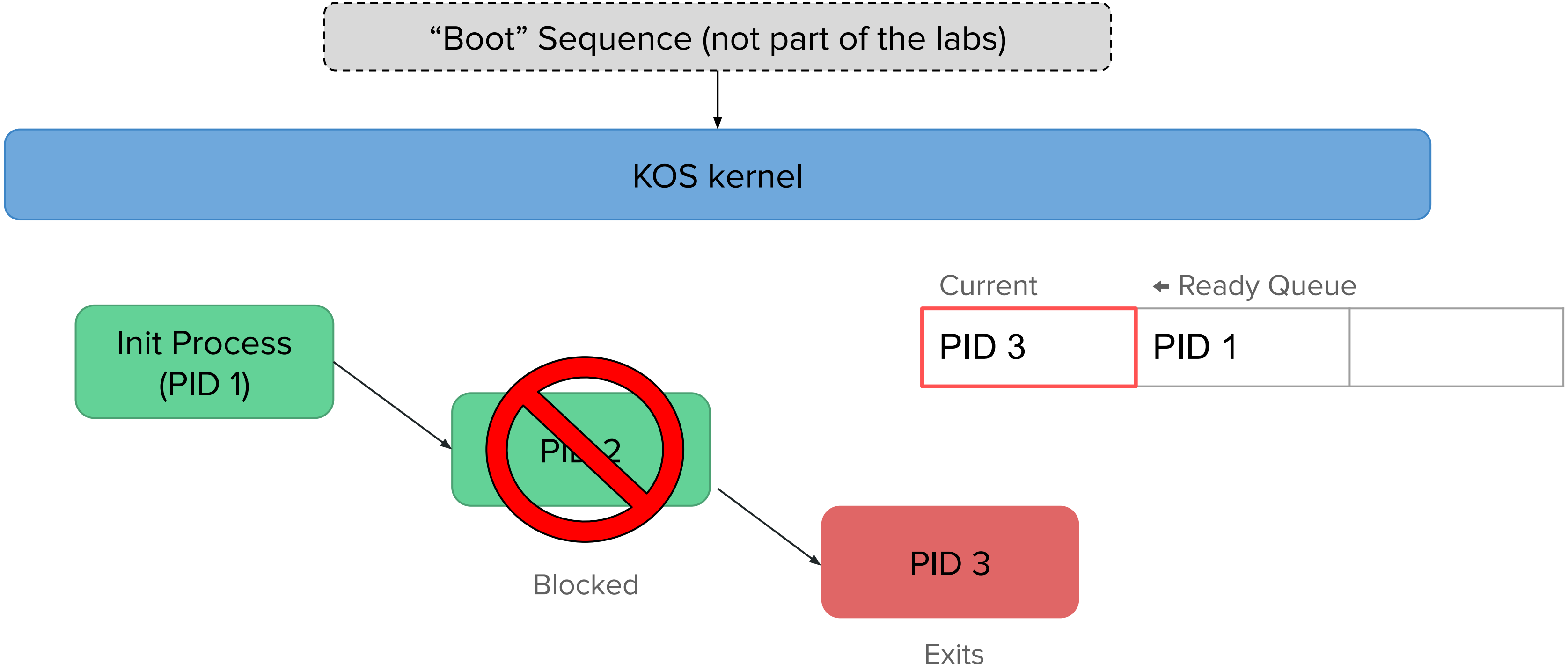
Blocked

PID 3

Exits

Current	← Ready Queue	
PID 3	PID 1	

Scenario 2: Child (PID 3) Exit When Parent is Waiting

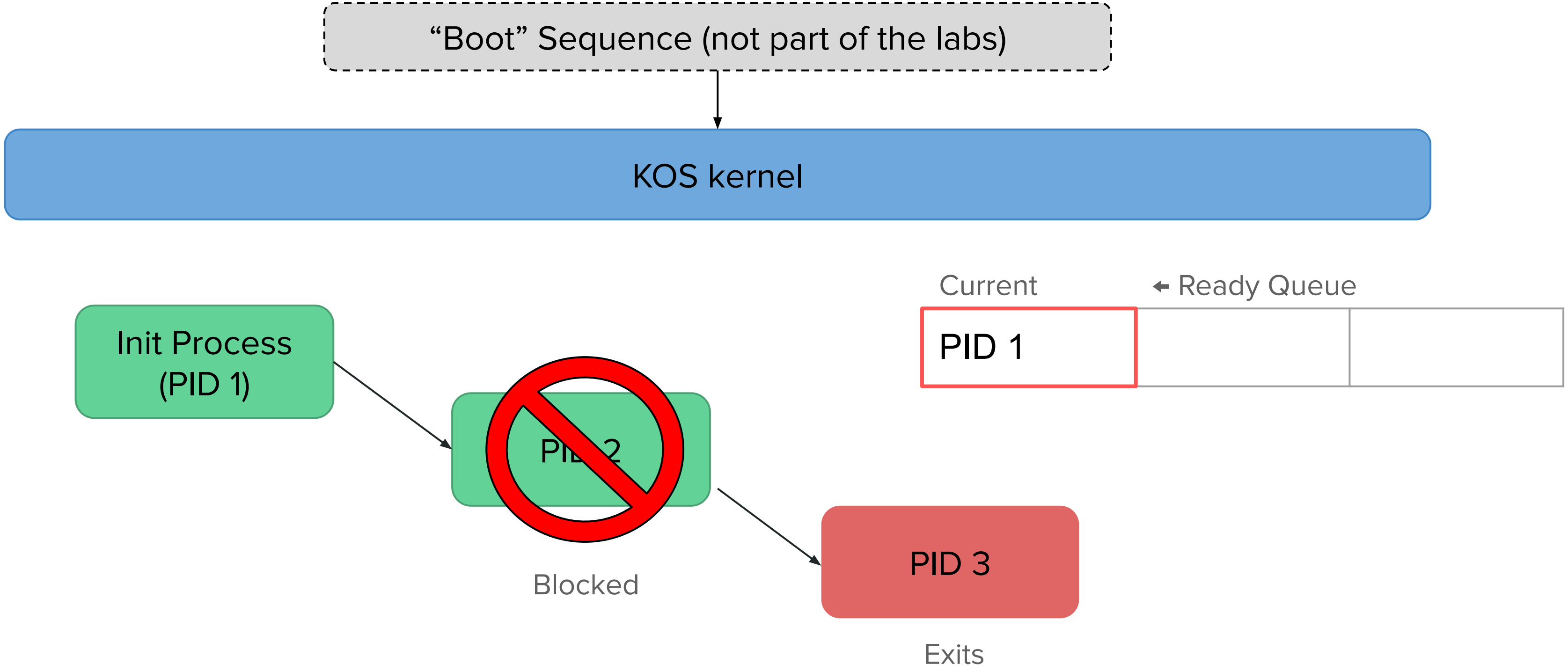


“Boot” Sequence (not part of the labs)

KOS kernel

Current	← Ready Queue	
PID 3	PID 1	

Scenario 2: Child (PID 3) Exit When Parent is Waiting



“Boot” Sequence (not part of the labs)

KOS kernel

Init Process (PID 1)

PID 2

Blocked

PID 3

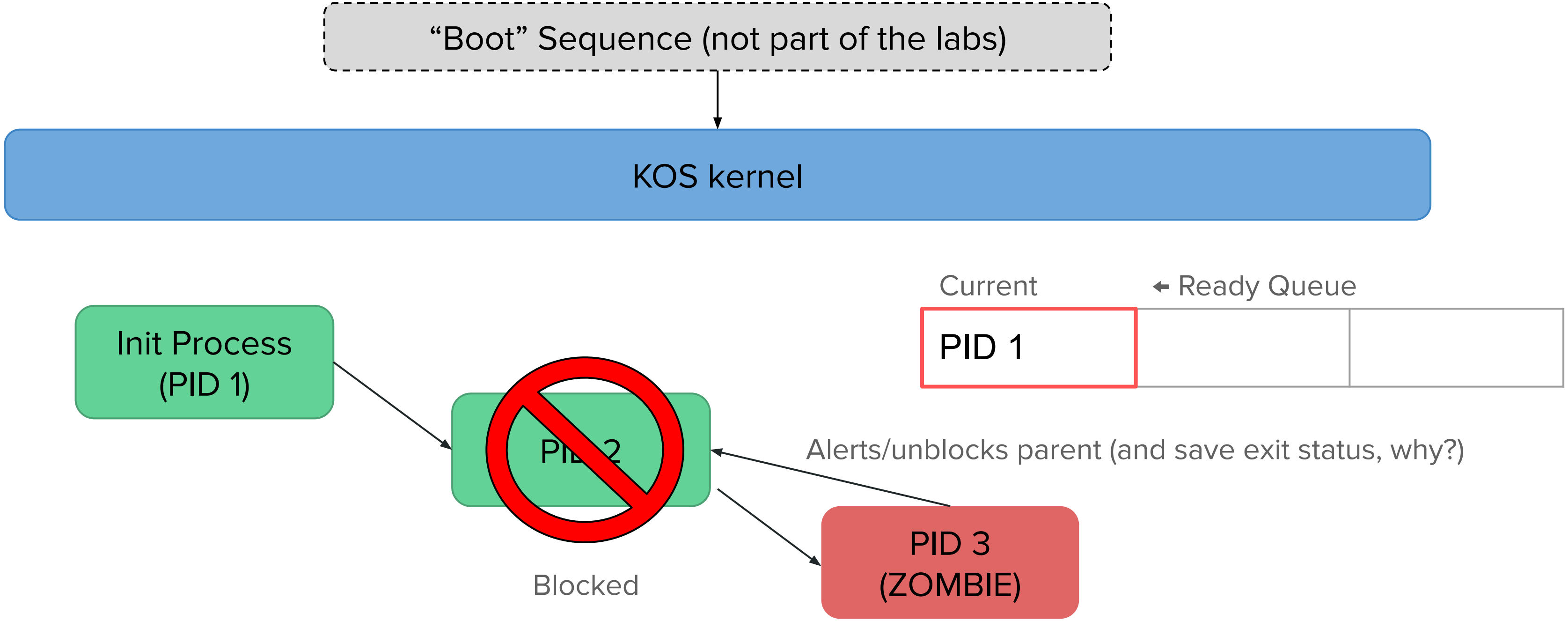
Exits

Current

← Ready Queue

PID 1

Scenario 2: Child (PID 3) Exit When Parent is Waiting



“Boot” Sequence (not part of the labs)

KOS kernel

Current

← Ready Queue

PID 1

Init Process (PID 1)

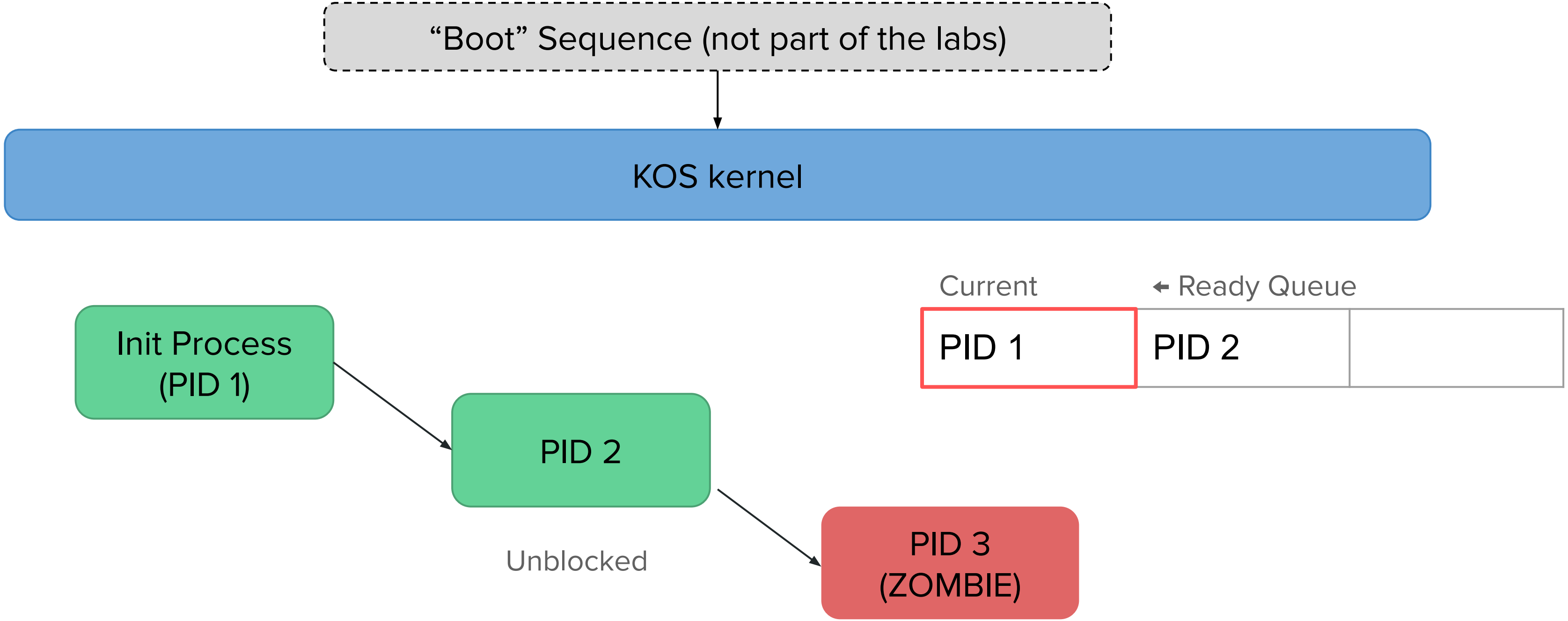
PID 2

Blocked

Alerts/unblocks parent (and save exit status, why?)

PID 3 (ZOMBIE)

Scenario 2: Child (PID 3) Exit When Parent is Waiting



“Boot” Sequence (not part of the labs)

KOS kernel

Init Process (PID 1)

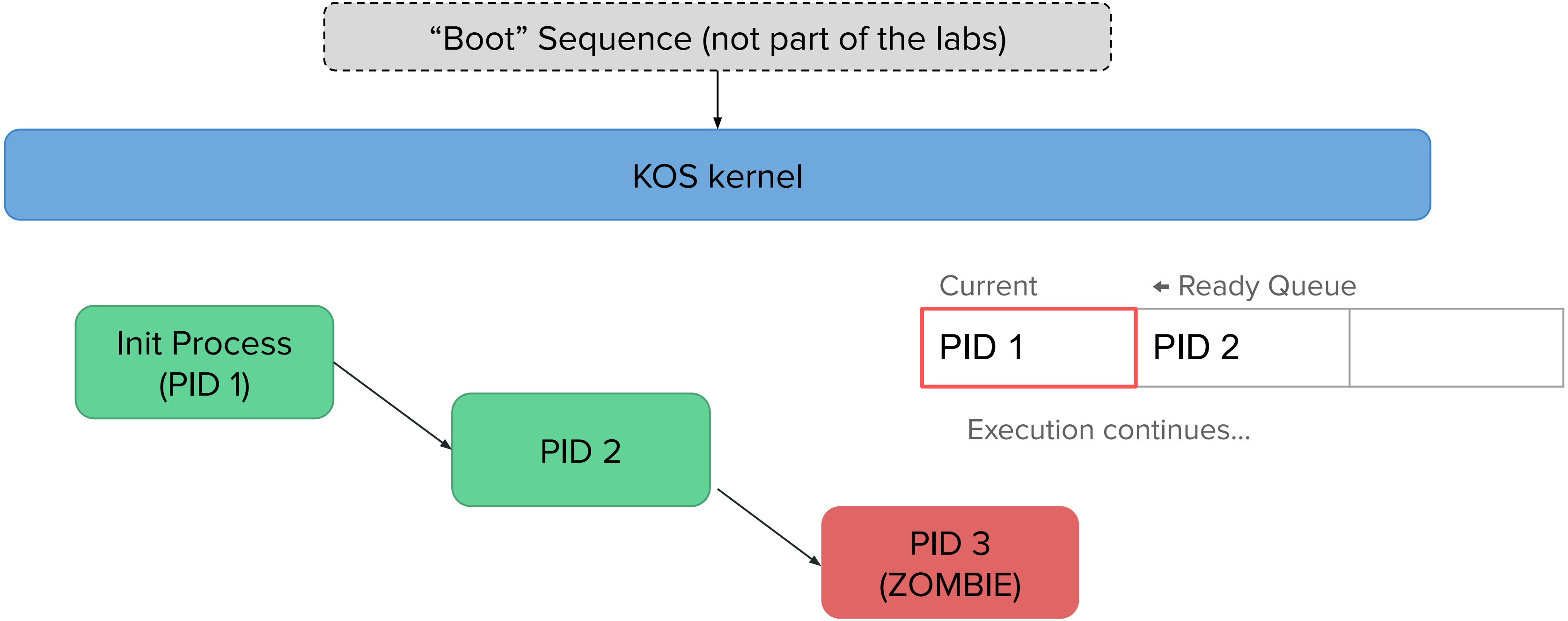
PID 2

Unblocked

PID 3 (ZOMBIE)

Current	← Ready Queue	
PID 1	PID 2	

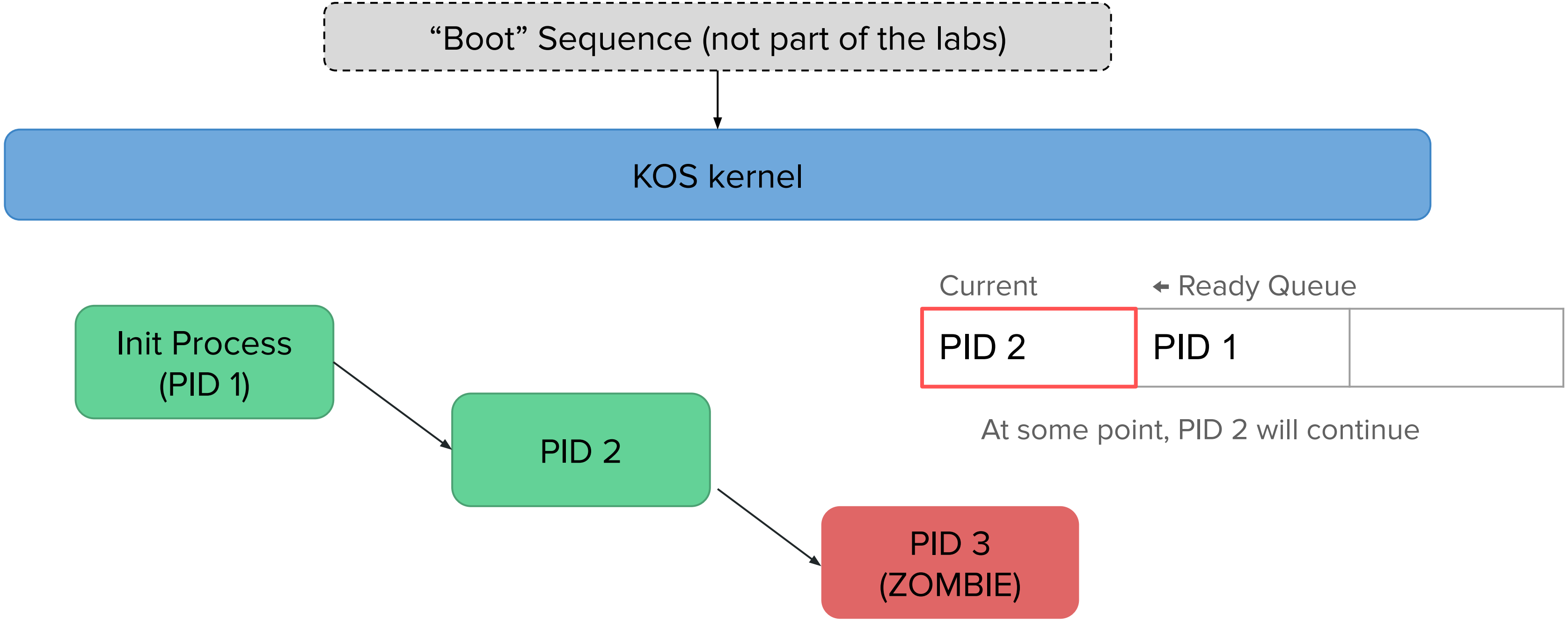
Scenario 2: Child (PID 3) Exit When Parent is Waiting



Current	← Ready Queue	
PID 1	PID 2	

Execution continues...

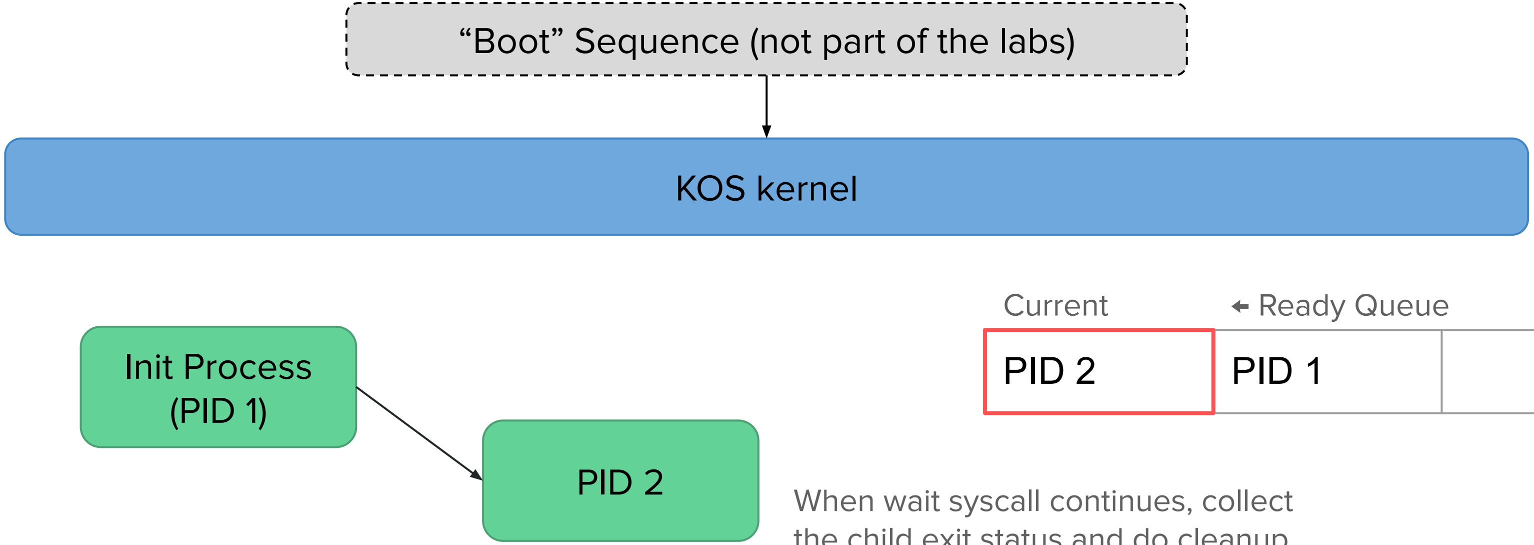
Scenario 2: Child (PID 3) Exit When Parent is Waiting



Current	← Ready Queue	
PID 2	PID 1	

At some point, PID 2 will continue

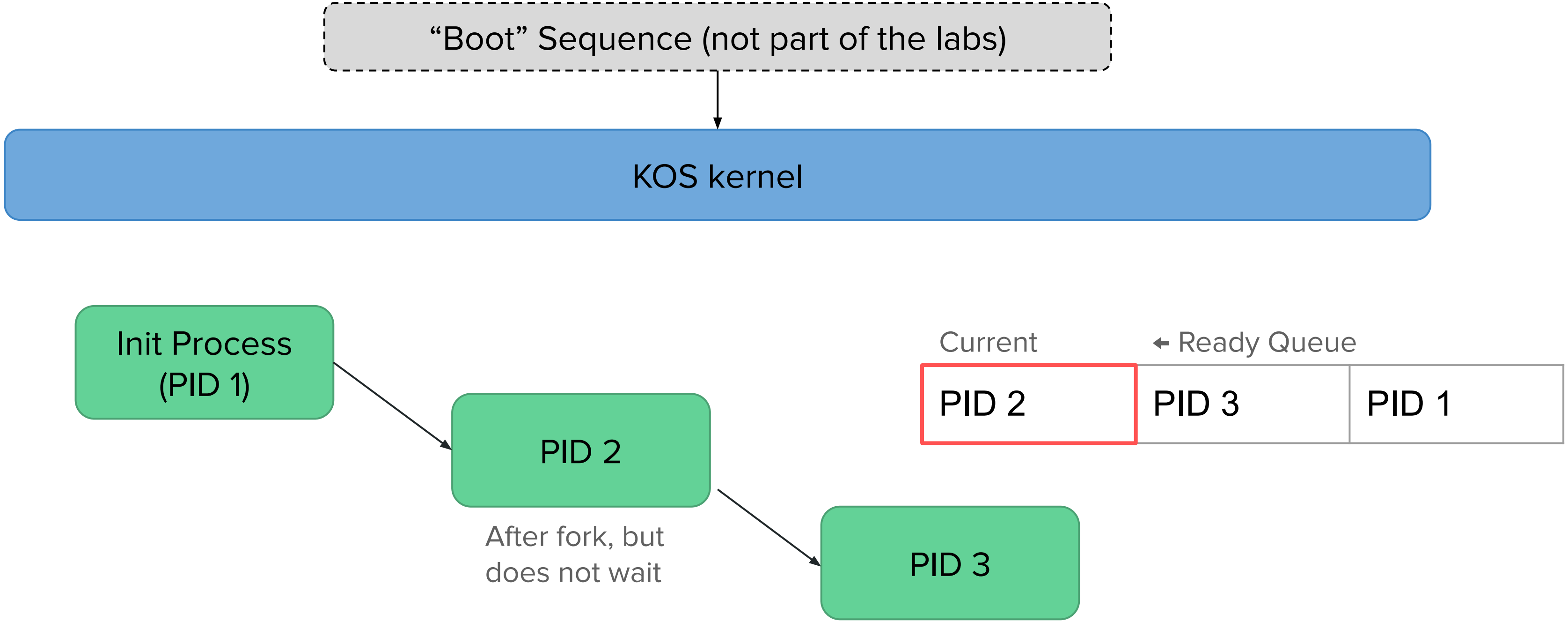
Scenario 2: Child (PID 3) Exit When Parent is Waiting



Current	← Ready Queue	
PID 2	PID 1	

When wait syscall continues, collect the child exit status and do cleanup

Scenario 3: Parent (PID 2) Does Not Wait For Child (PID 3)



“Boot” Sequence (not part of the labs)

KOS kernel

Init Process (PID 1)

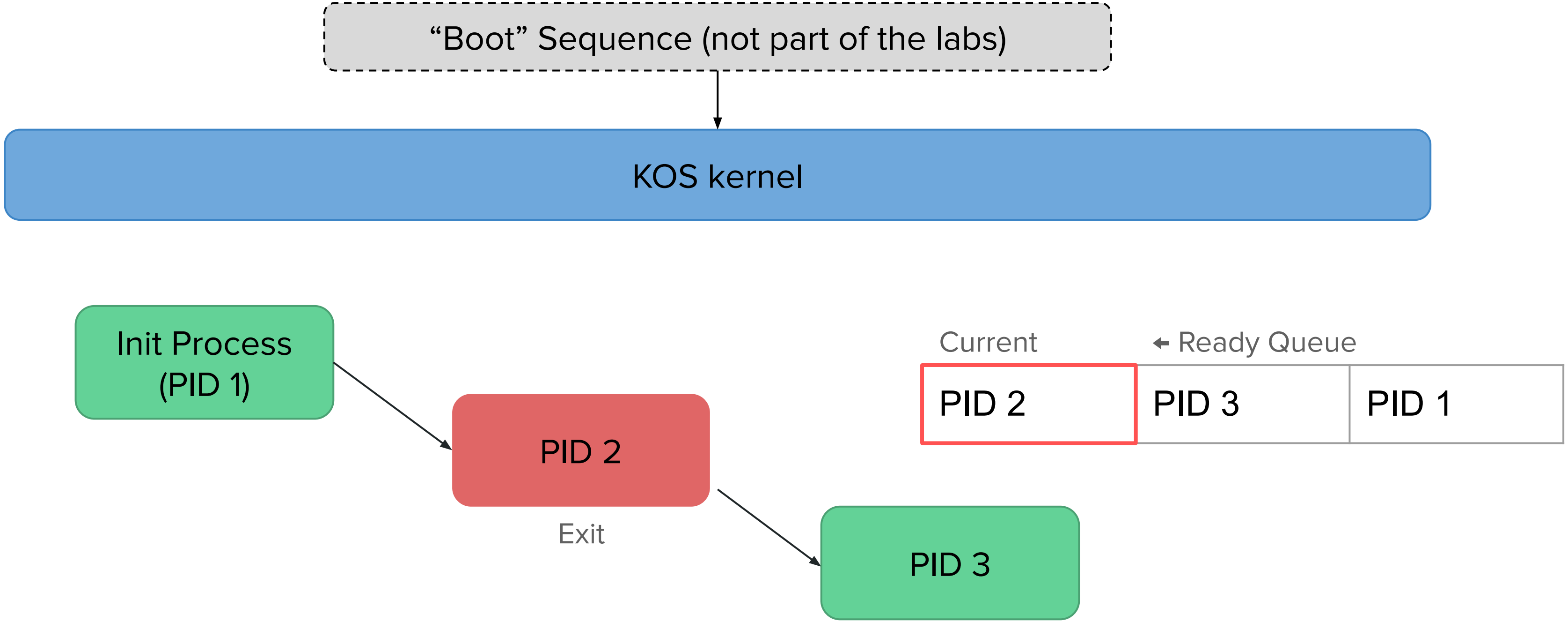
PID 2

After fork, but does not wait

PID 3

Current	← Ready Queue	
PID 2	PID 3	PID 1

Scenario 3: Parent (PID 2) Does Not Wait For Child (PID 3)



“Boot” Sequence (not part of the labs)

KOS kernel

Init Process
(PID 1)

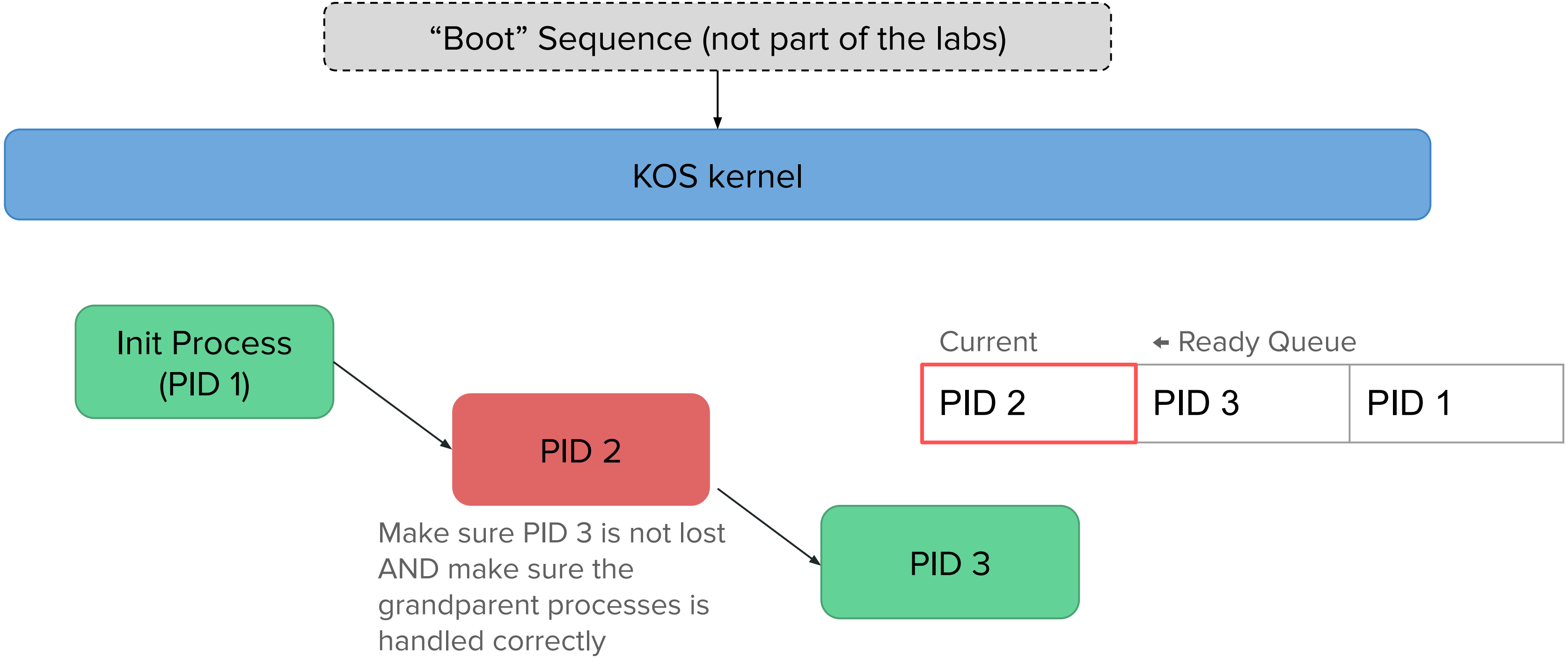
PID 2

Exit

PID 3

Current	← Ready Queue	
PID 2	PID 3	PID 1

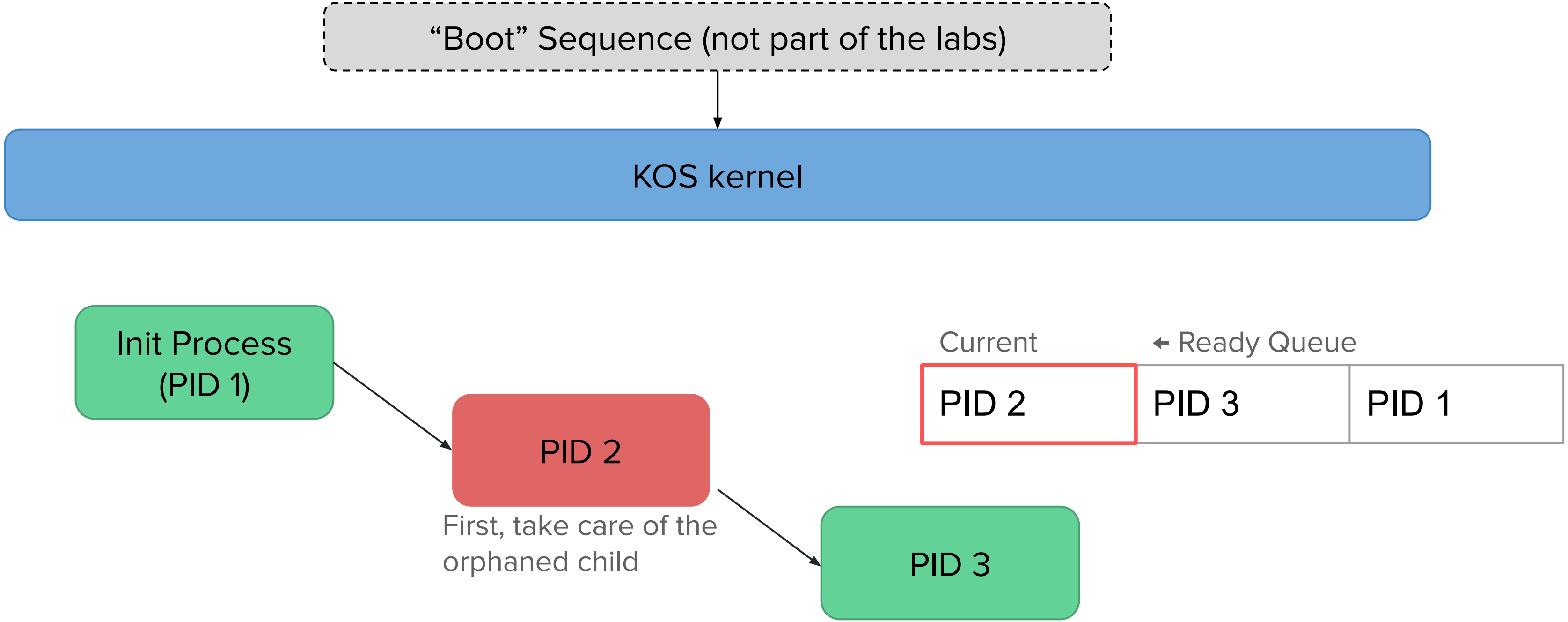
Scenario 3: Parent (PID 2) Does Not Wait For Child (PID 3)



Current	← Ready Queue	
PID 2	PID 3	PID 1

Make sure PID 3 is not lost AND make sure the grandparent processes is handled correctly

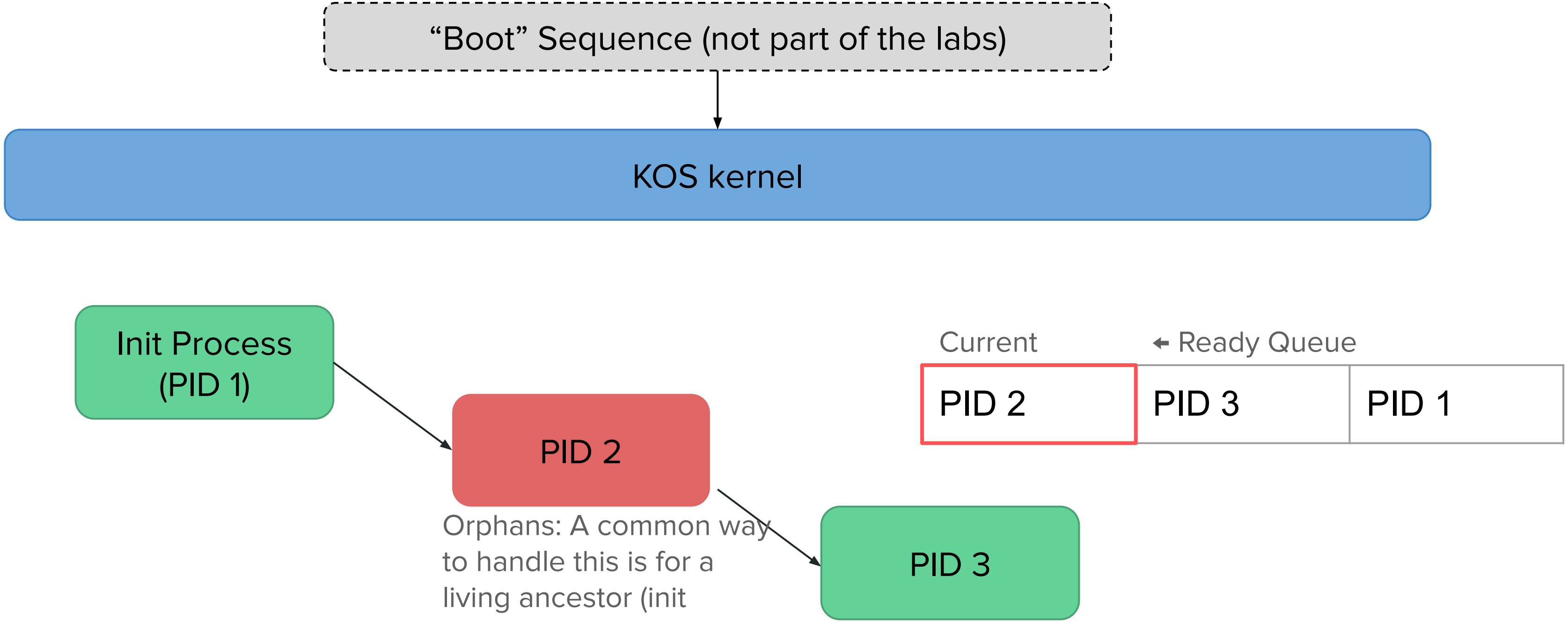
Scenario 3: Parent (PID 2) Does Not Wait For Child (PID 3)



First, take care of the orphaned child

Current	← Ready Queue	
PID 2	PID 3	PID 1

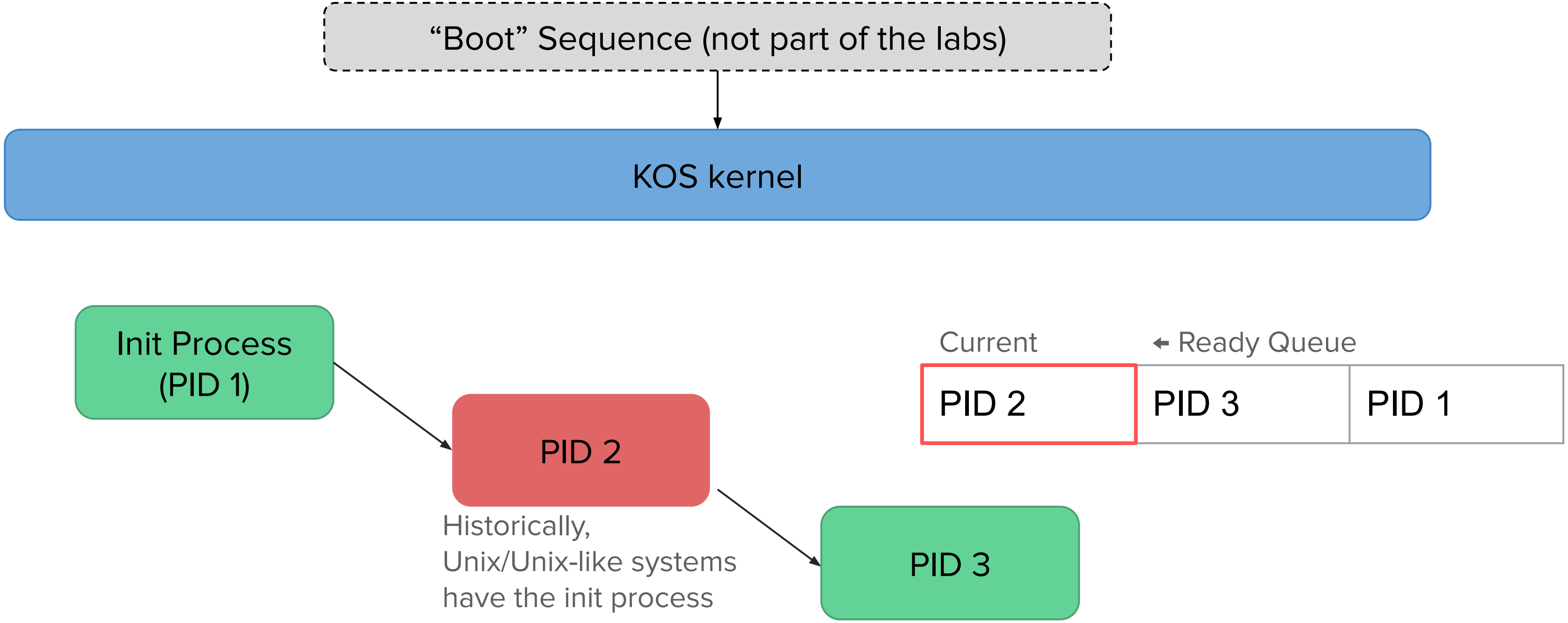
Scenario 3: Parent (PID 2) Does Not Wait For Child (PID 3)



Current	← Ready Queue	
PID 2	PID 3	PID 1

Orphans: A common way to handle this is for a living ancestor (init process) to adopt it!

Scenario 3: Parent (PID 2) Does Not Wait For Child (PID 3)



“Boot” Sequence (not part of the labs)

KOS kernel

Init Process (PID 1)

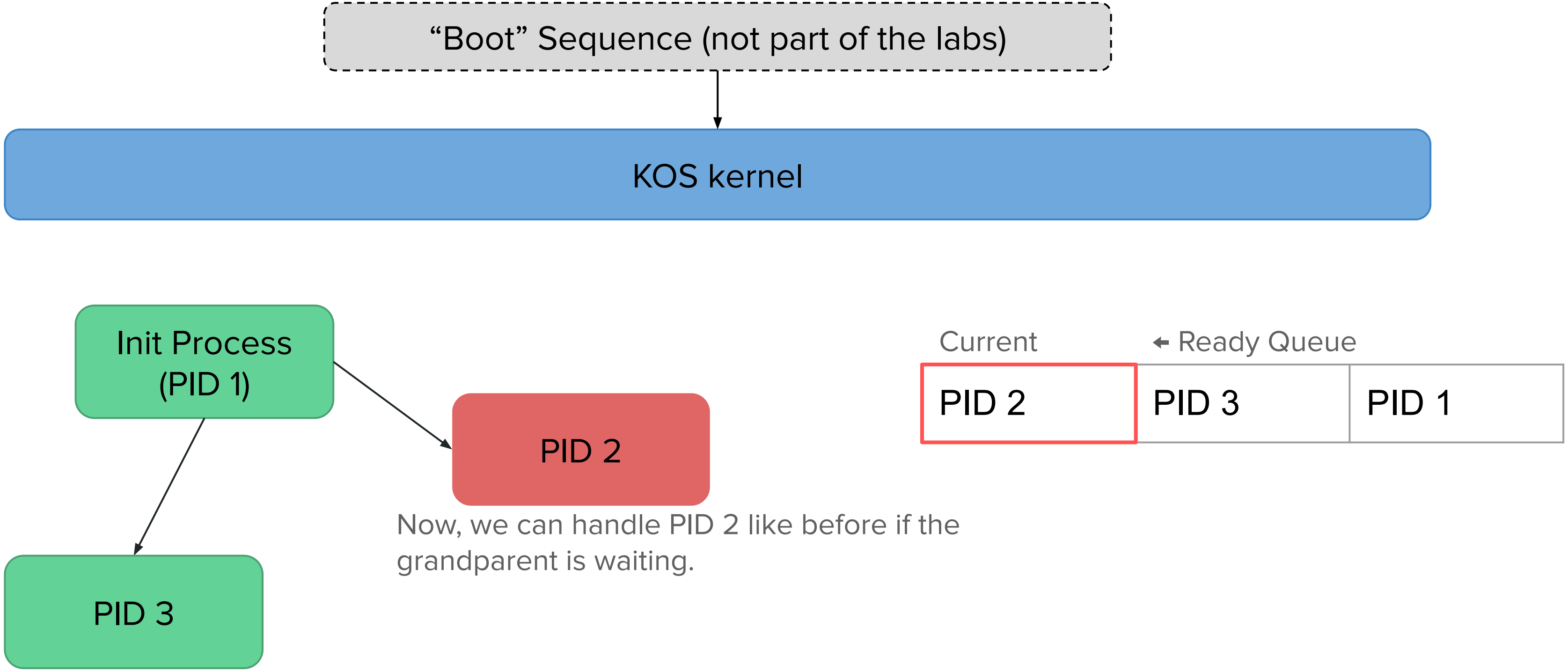
PID 2

PID 3

Historically, Unix/Unix-like systems have the init process adopt orphans

Current	← Ready Queue	
PID 2	PID 3	PID 1

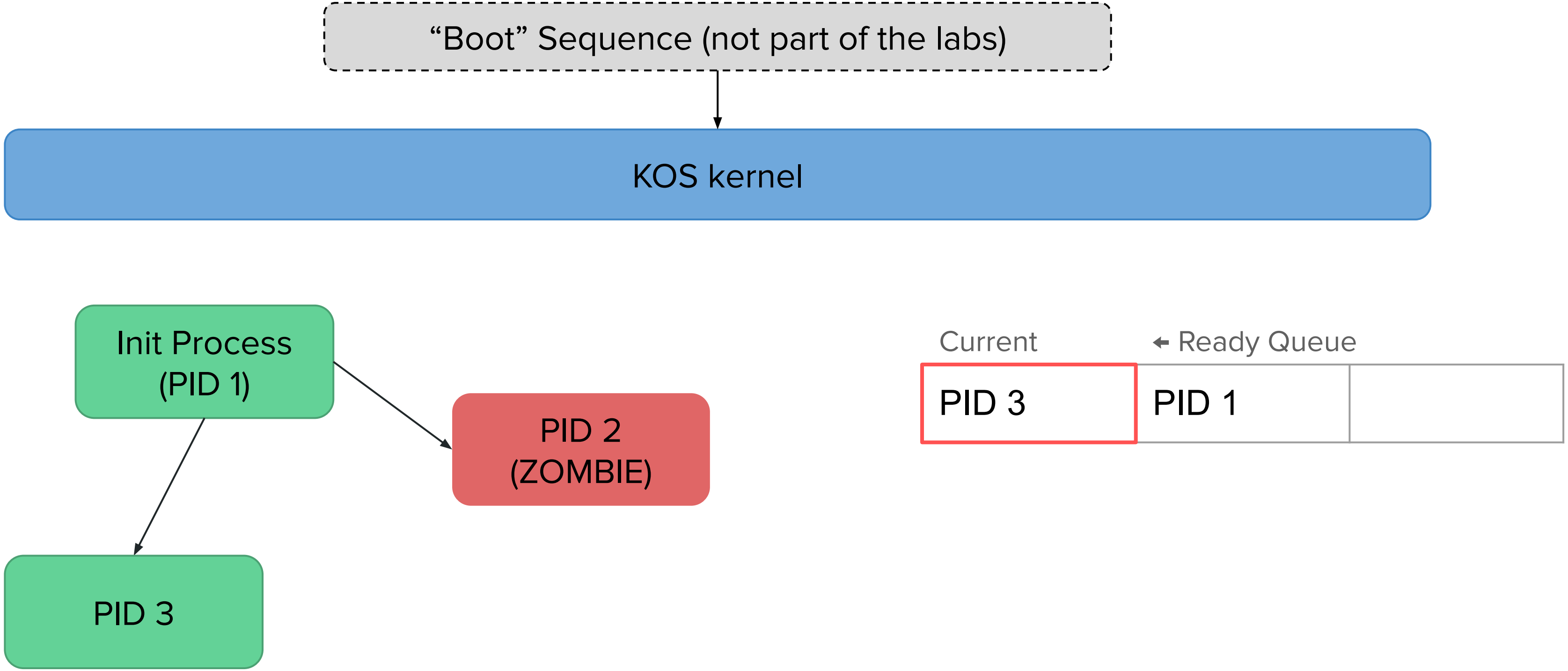
Scenario 3: Parent (PID 2) Does Not Wait For Child (PID 3)



Current	← Ready Queue	
PID 2	PID 3	PID 1

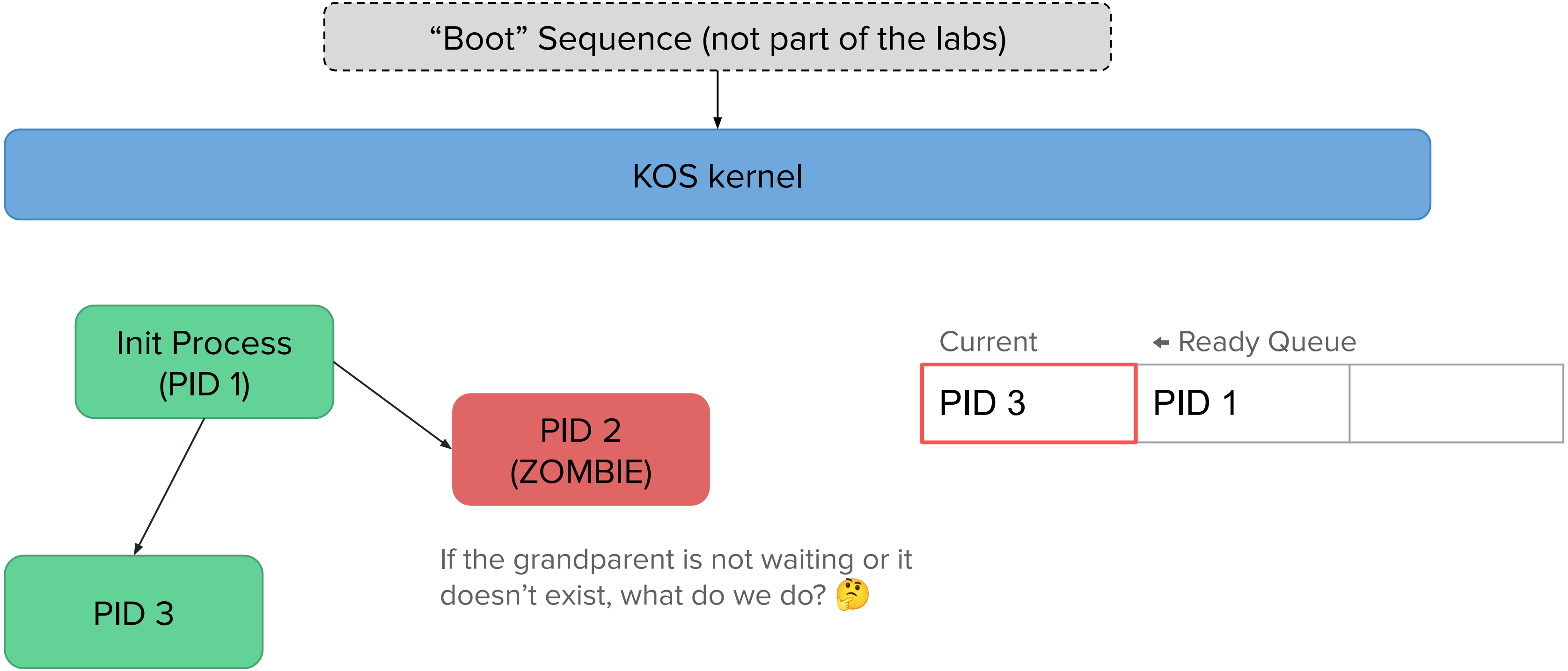
Now, we can handle PID 2 like before if the grandparent is waiting.

Scenario 3: Parent (PID 2) Does Not Wait For Child (PID 3)



Current	← Ready Queue	
PID 3	PID 1	

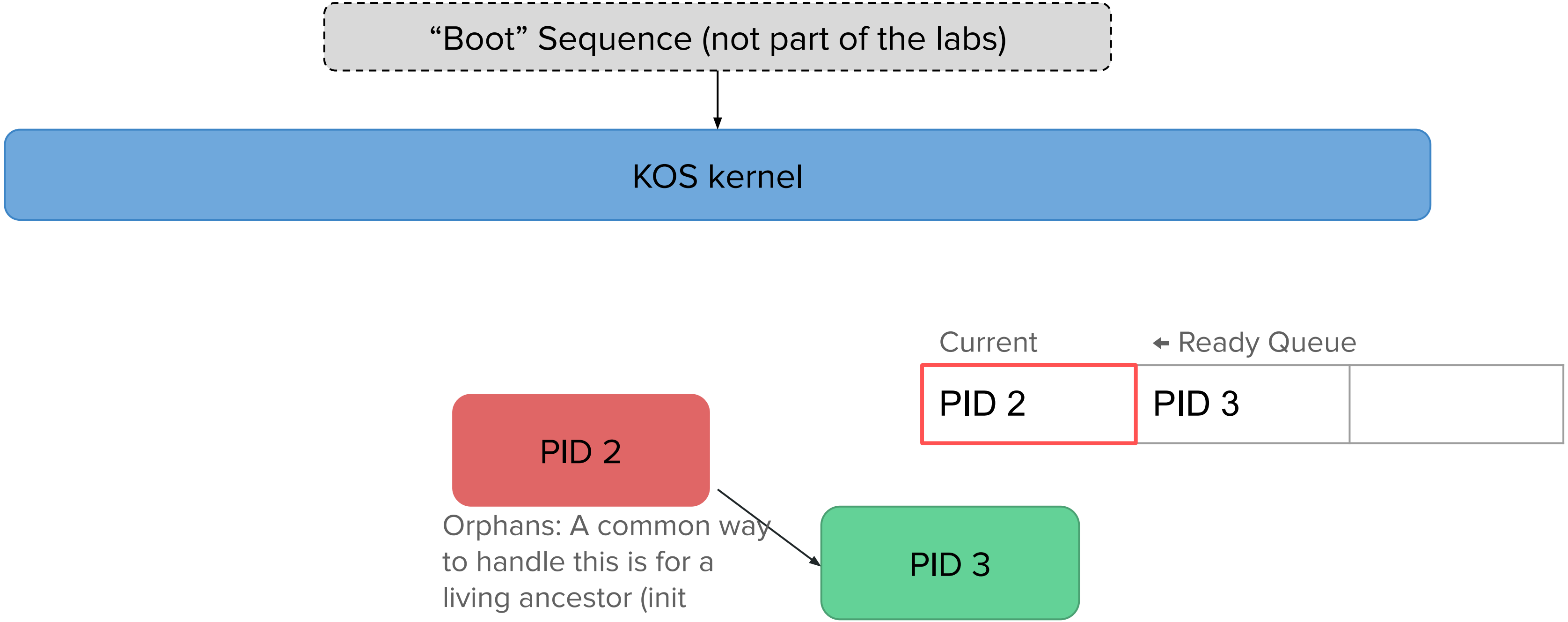
Scenario 3: Parent (PID 2) Does Not Wait For Child (PID 3)



Current	Ready Queue	
PID 3	PID 1	

If the grandparent is not waiting or it doesn’t exist, what do we do? 🤔

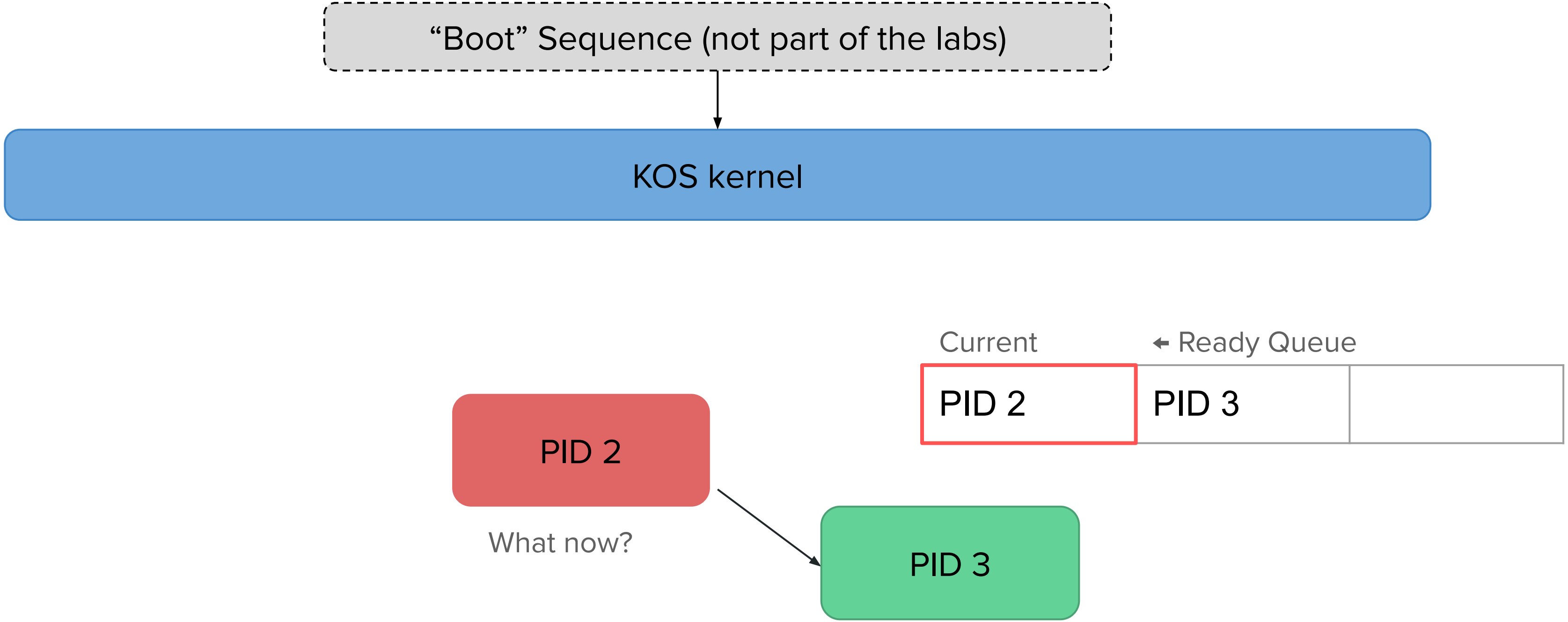
Scenario 3.5: There are no ancestors



Current	← Ready Queue	
PID 2	PID 3	

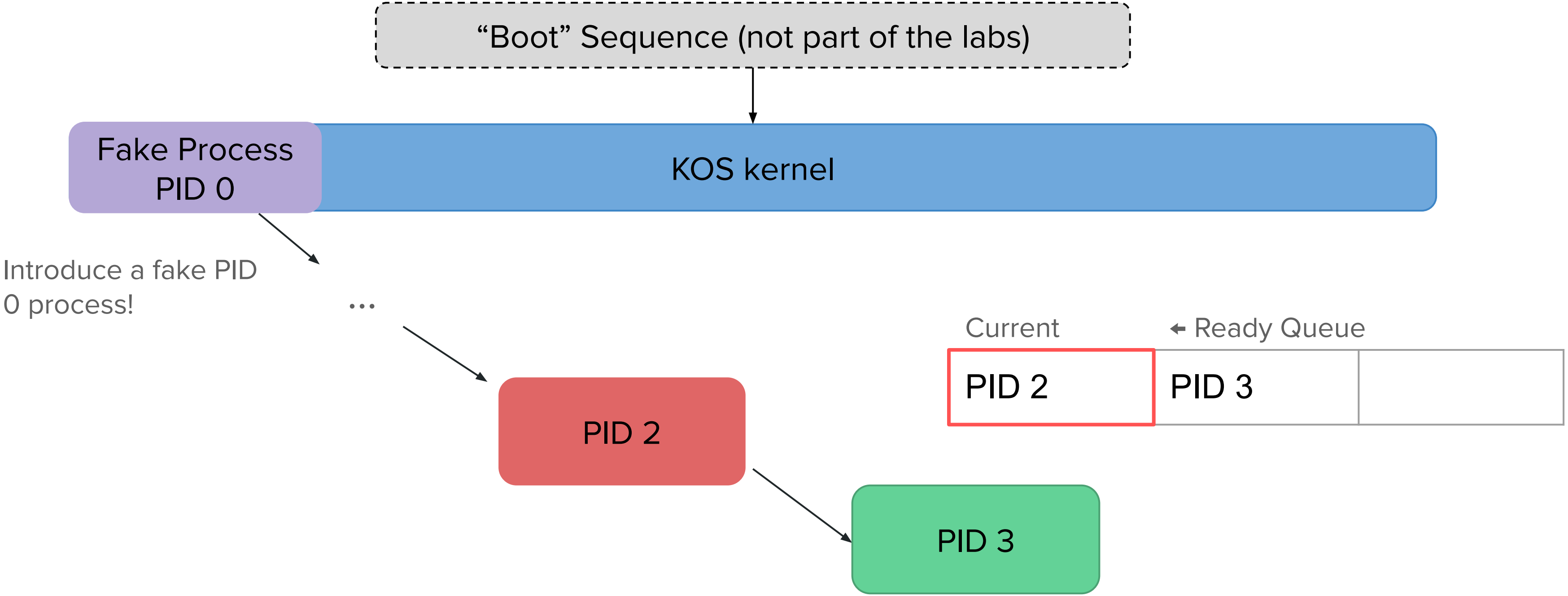
Orphans: A common way to handle this is for a living ancestor (init process) to adopt it!?

Scenario 3.5: There are no ancestors



Current	← Ready Queue	
PID 2	PID 3	

Scenario 3.5: There are no ancestors



“Boot” Sequence (not part of the labs)

Fake Process PID 0

KOS kernel

Introduce a fake PID 0 process!

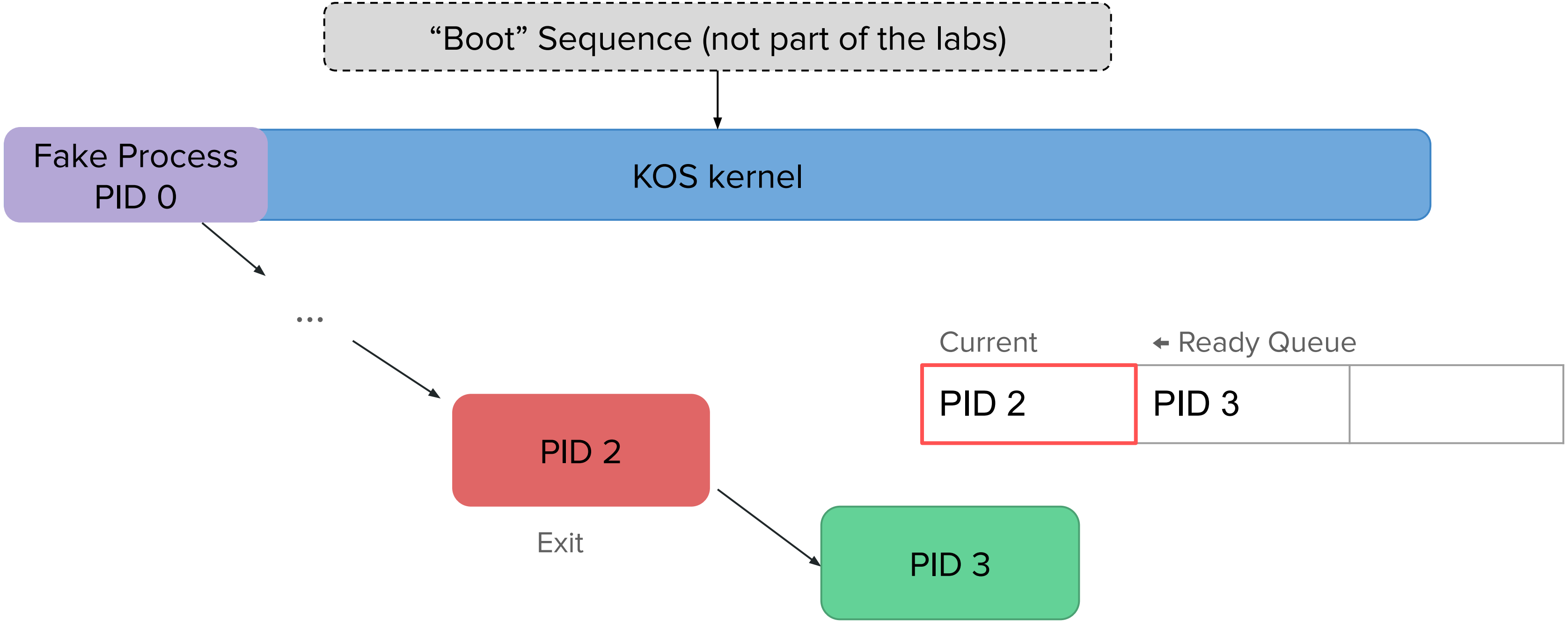
...

PID 2

PID 3

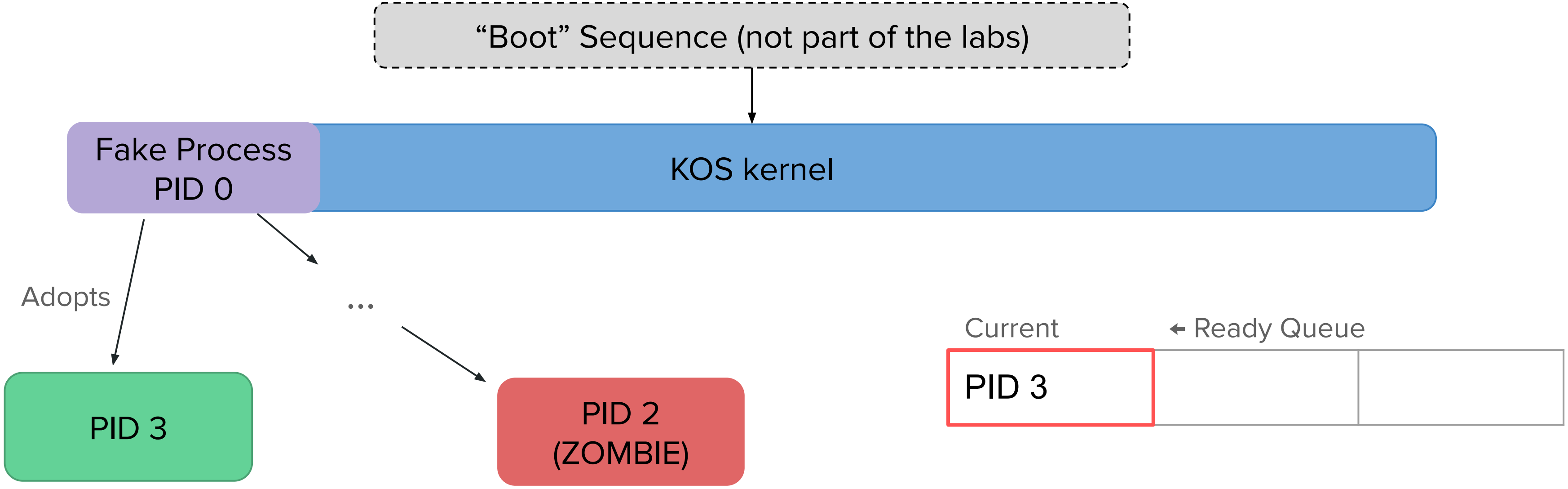
Current	← Ready Queue	
PID 2	PID 3	

Scenario 3.5: There are no ancestors



Current	← Ready Queue	
PID 2	PID 3	

Scenario 3.5: There are no ancestors



“Boot” Sequence (not part of the labs)

Fake Process
PID 0

KOS kernel

Adopts

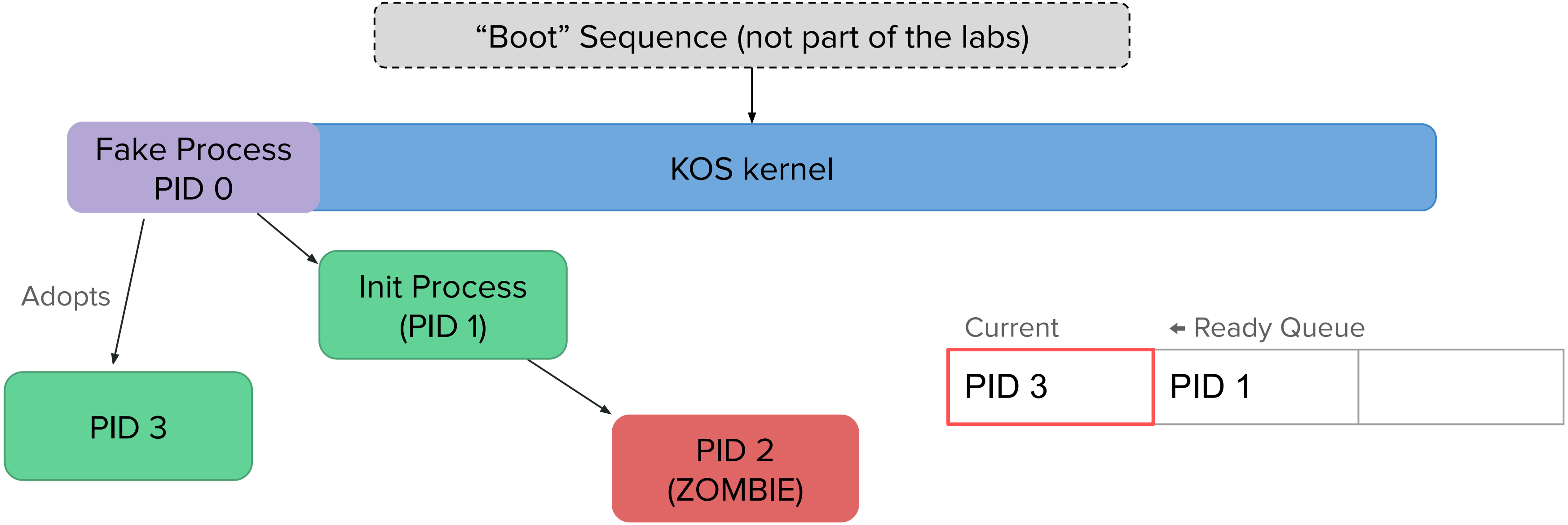
...

PID 3

PID 2
(ZOMBIE)

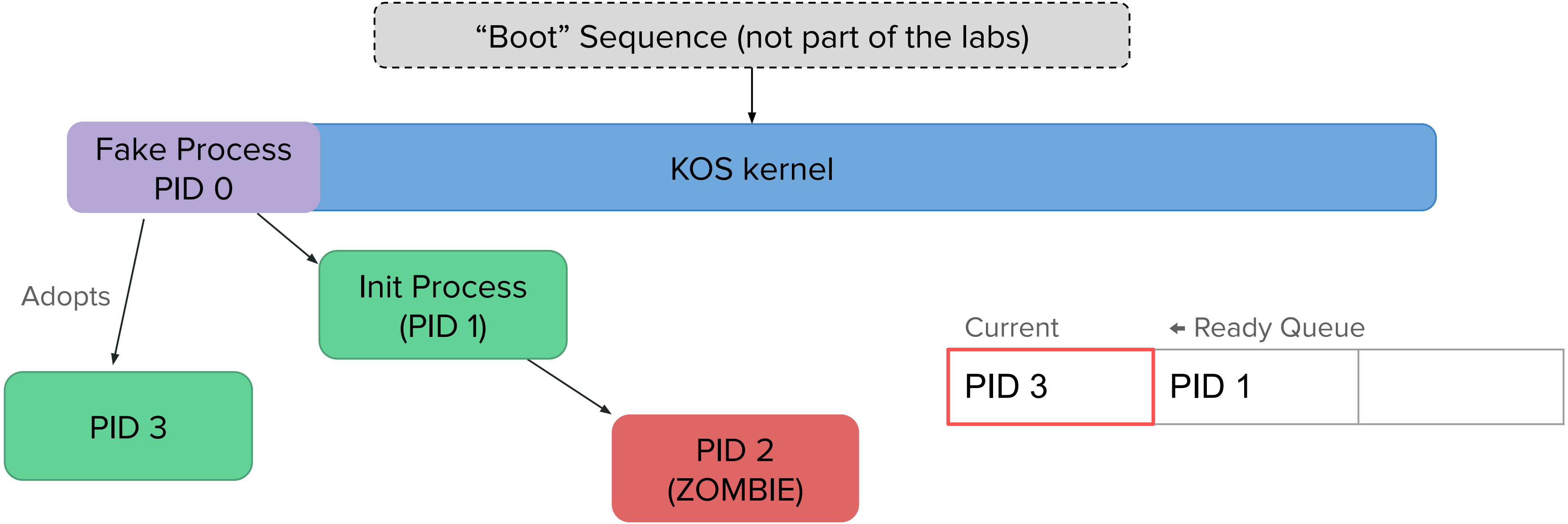
Current	← Ready Queue	
PID 3		

Scenario 3.99: Grandparent is not waiting



Current	← Ready Queue	
PID 3	PID 1	

Scenario 3.99: Grandparent is not waiting



“Boot” Sequence (not part of the labs)

Fake Process PID 0

KOS kernel

Adopts

Init Process (PID 1)

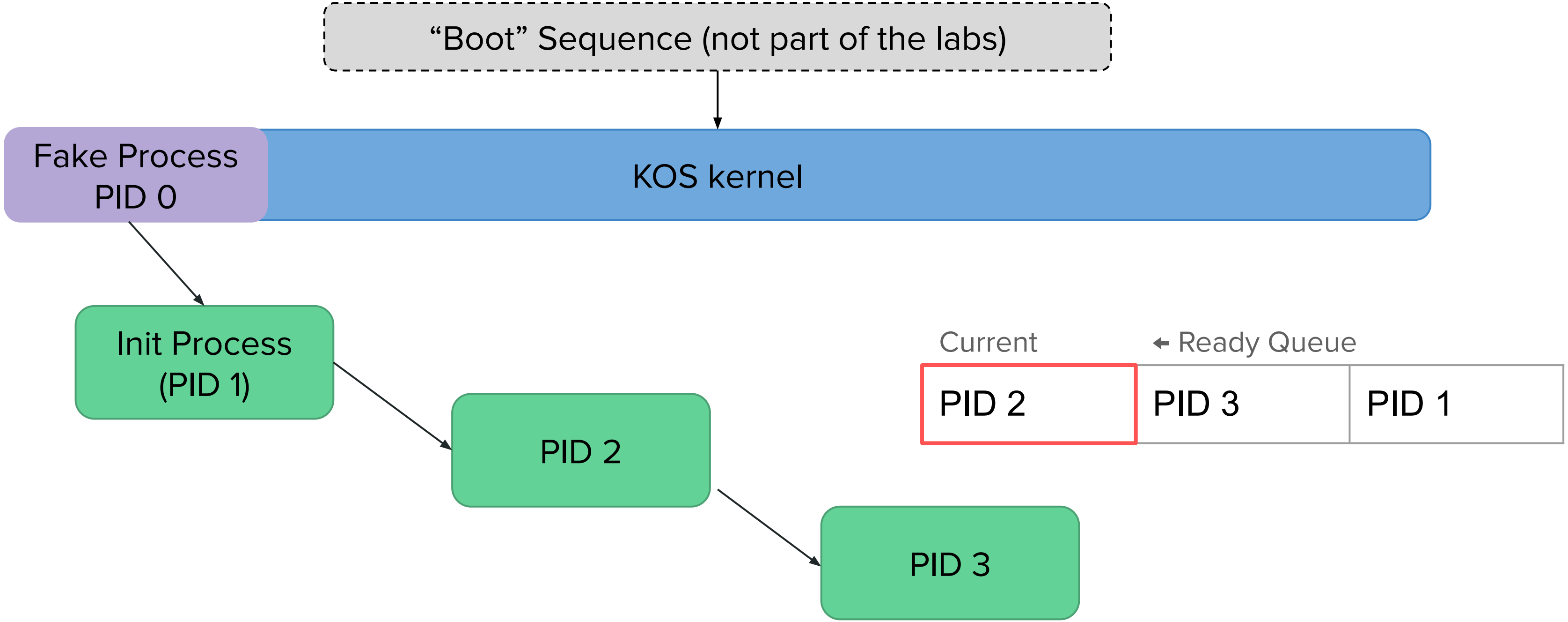
PID 3

PID 2 (ZOMBIE)

Current	← Ready Queue	
PID 3	PID 1	

With this, it doesn’t really matter, because either
1) Grandparent PID 1 exited first. In this case, the “fake process” PID 0 adopted PID 2 already
2) Grandparent PID 1 is still alive, so you have many options (e.g. let grandparent cleanup, clean immediately, etc)

Recap



“Boot” Sequence (not part of the labs)

Fake Process
PID 0

KOS kernel

Init Process
(PID 1)

PID 2

PID 3

Current	← Ready Queue	
PID 2	PID 3	PID 1

Additional Topics/Questions