Effortless Parallelism

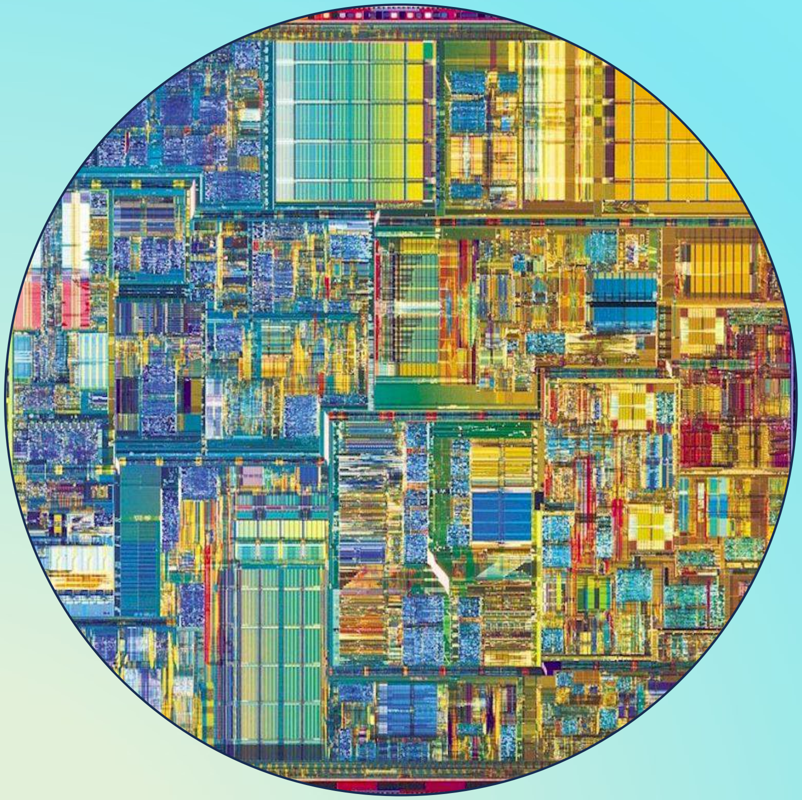
Leveraging Julia Threads for High-Performance Scientific Computing

Ed Armstrong
University of Guelph
SHARCNET

Effortless?

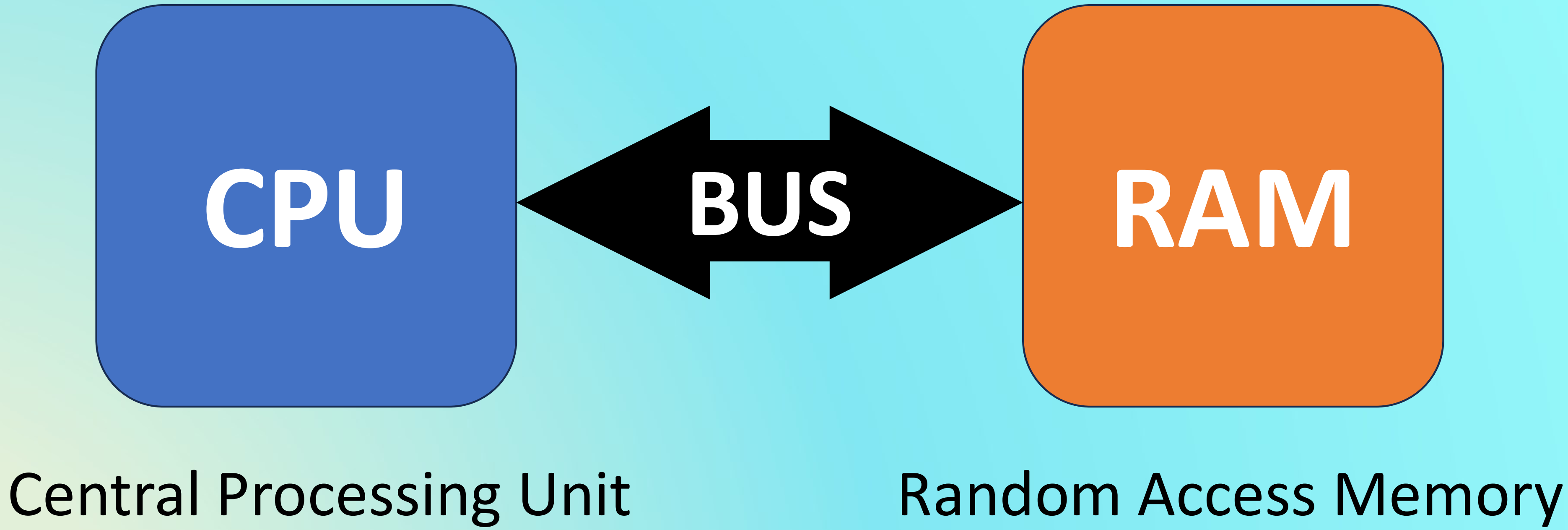
```julia
t1 = @spawn myfunction()
result = fetch(t1)
```

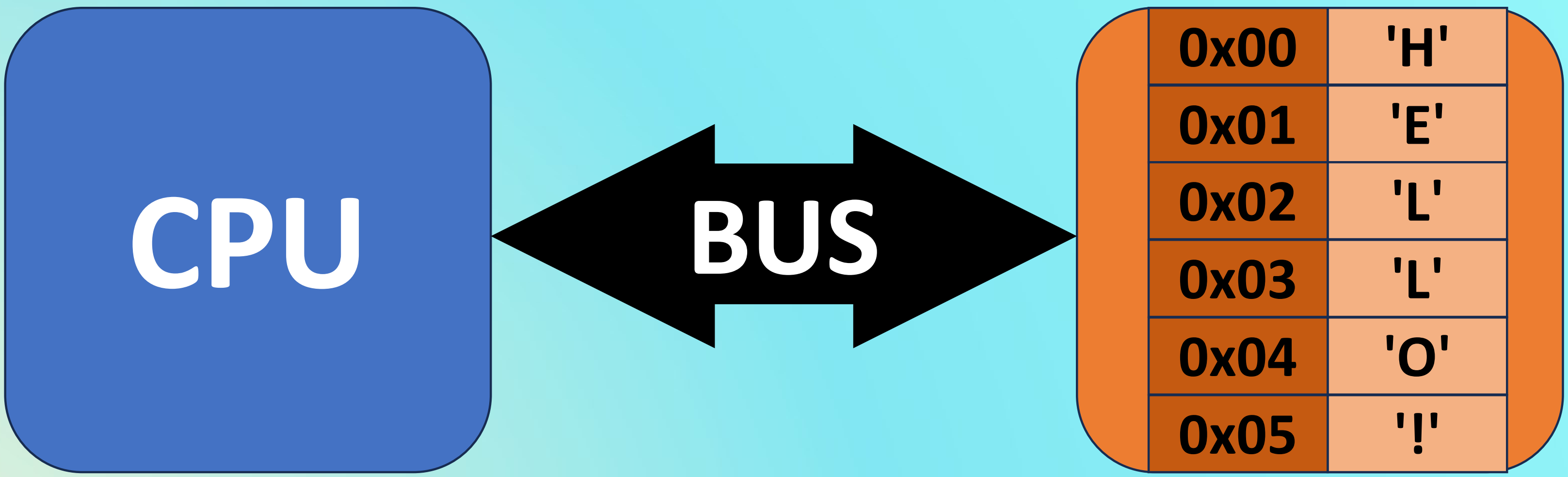
- A thread is a lightweight process.
- Shares memory with its parent process.
- Retains a separate CPU state.

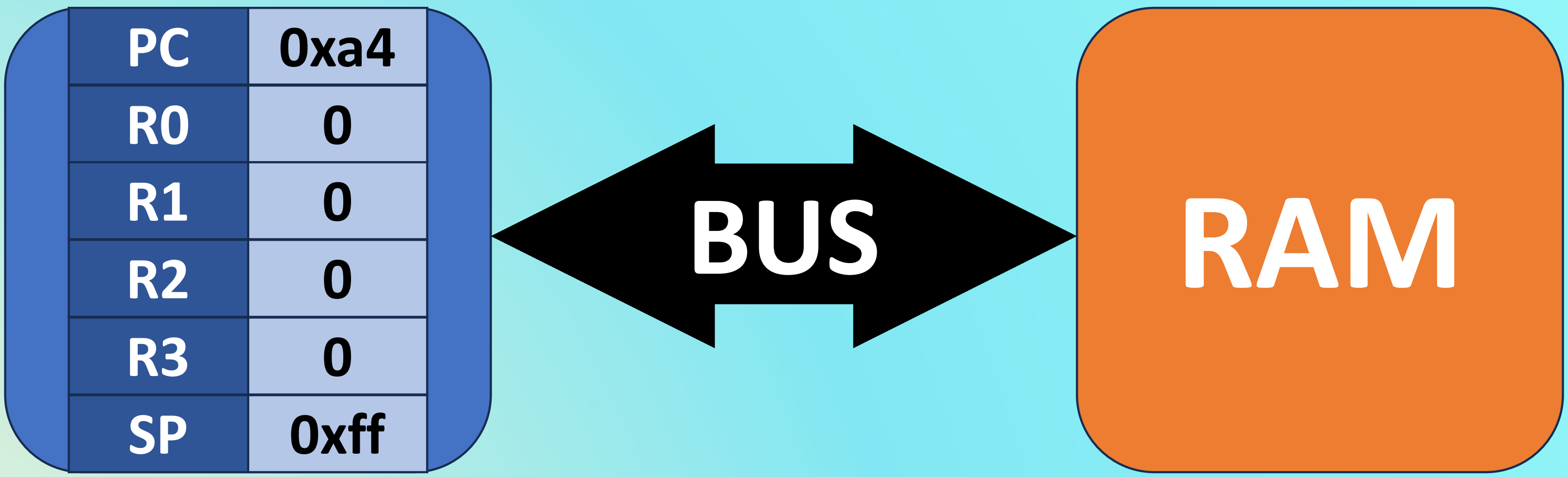


How Computer Do

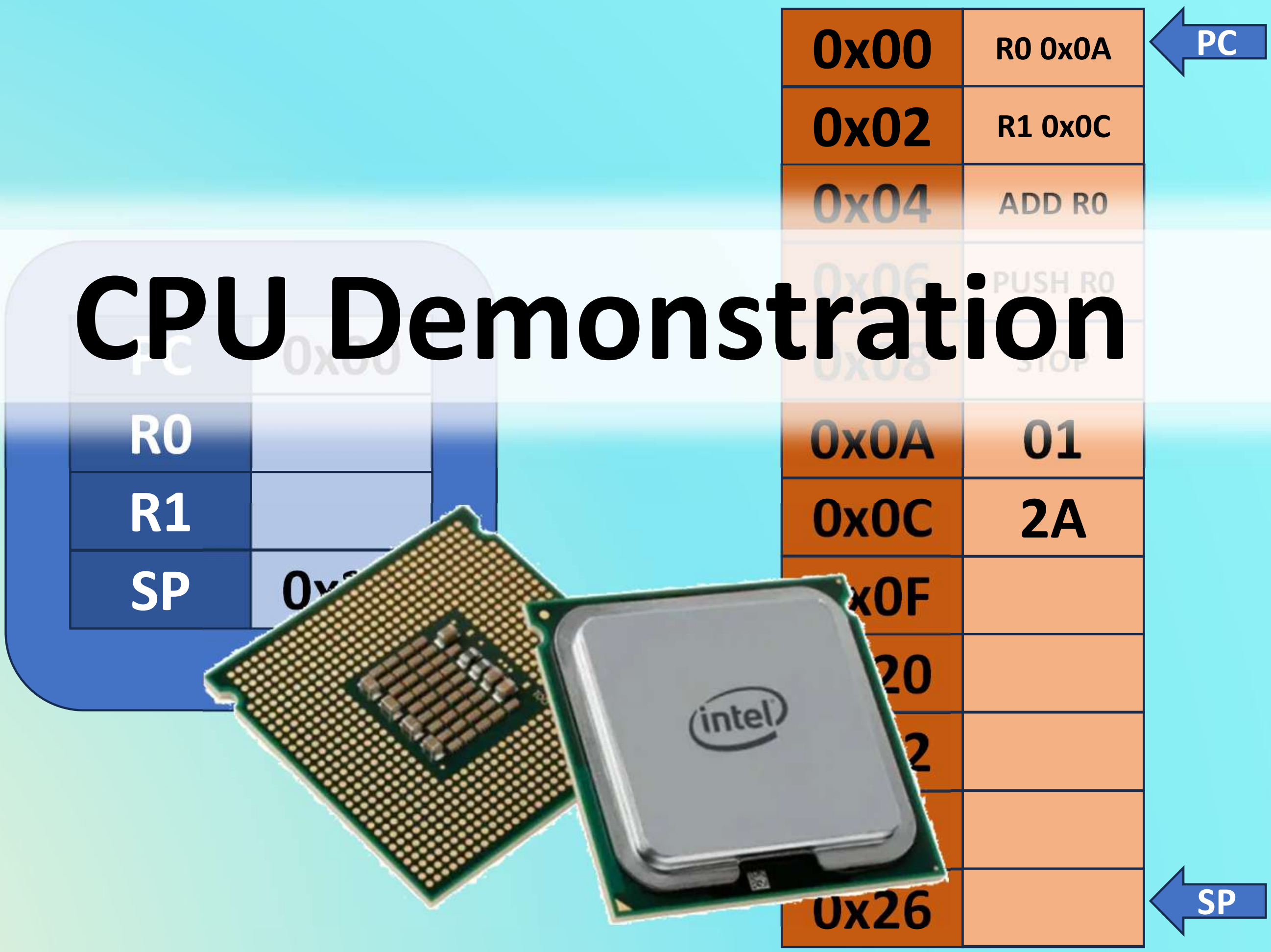
A brief description of computer architecture


CPU
BUS
RAM
Central Processing Unit
Random Access Memory


CPU
BUS
0x00 'H'
0x01 'E'
0x02 'L'
0x03 'L'
0x04 'O'
0x05 '!'


PC
0xa4
R0
0
R1
0
R2
0
R3
0
SP
0xff
BUS
RAM

CPU Demonstration

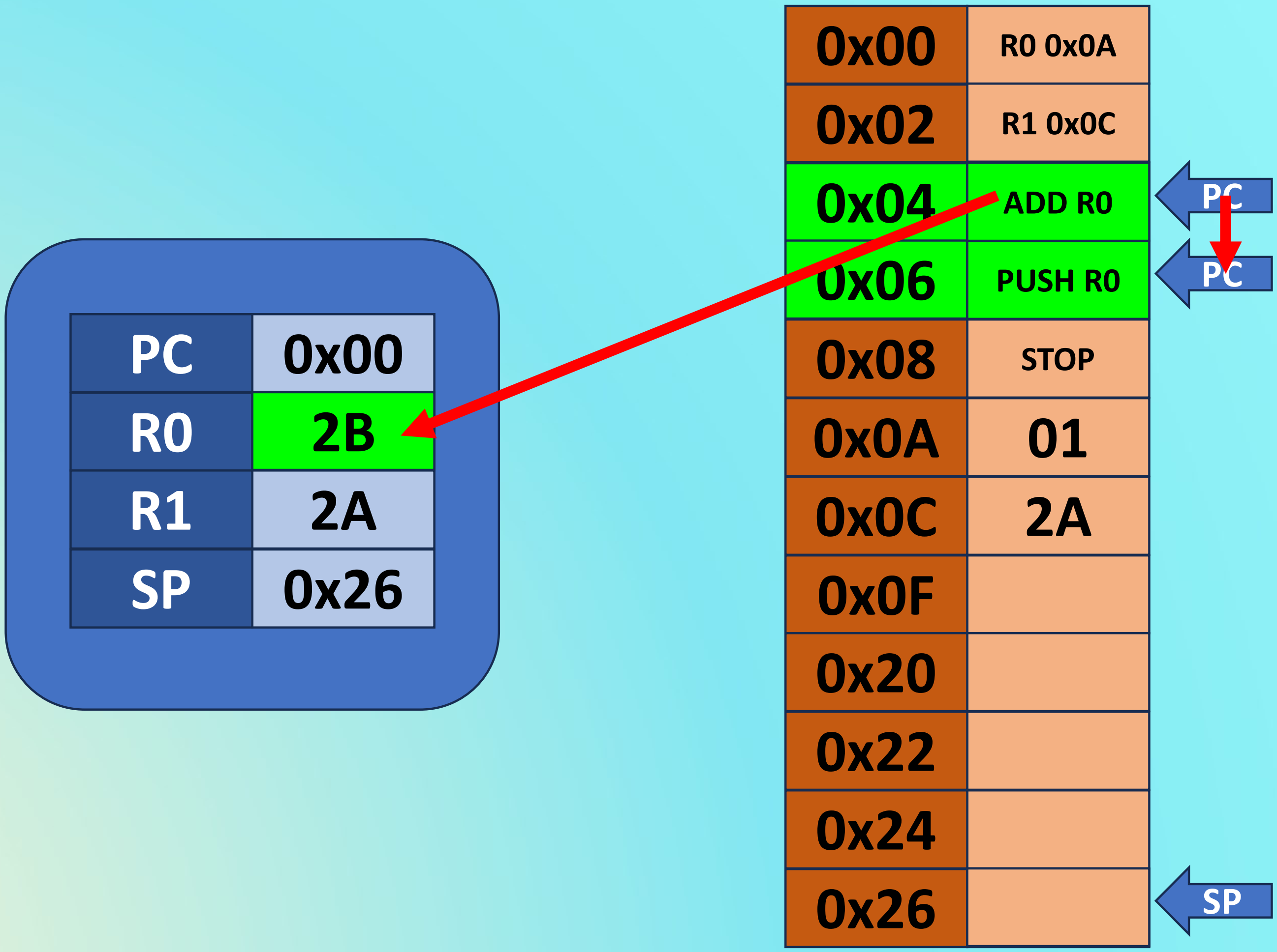


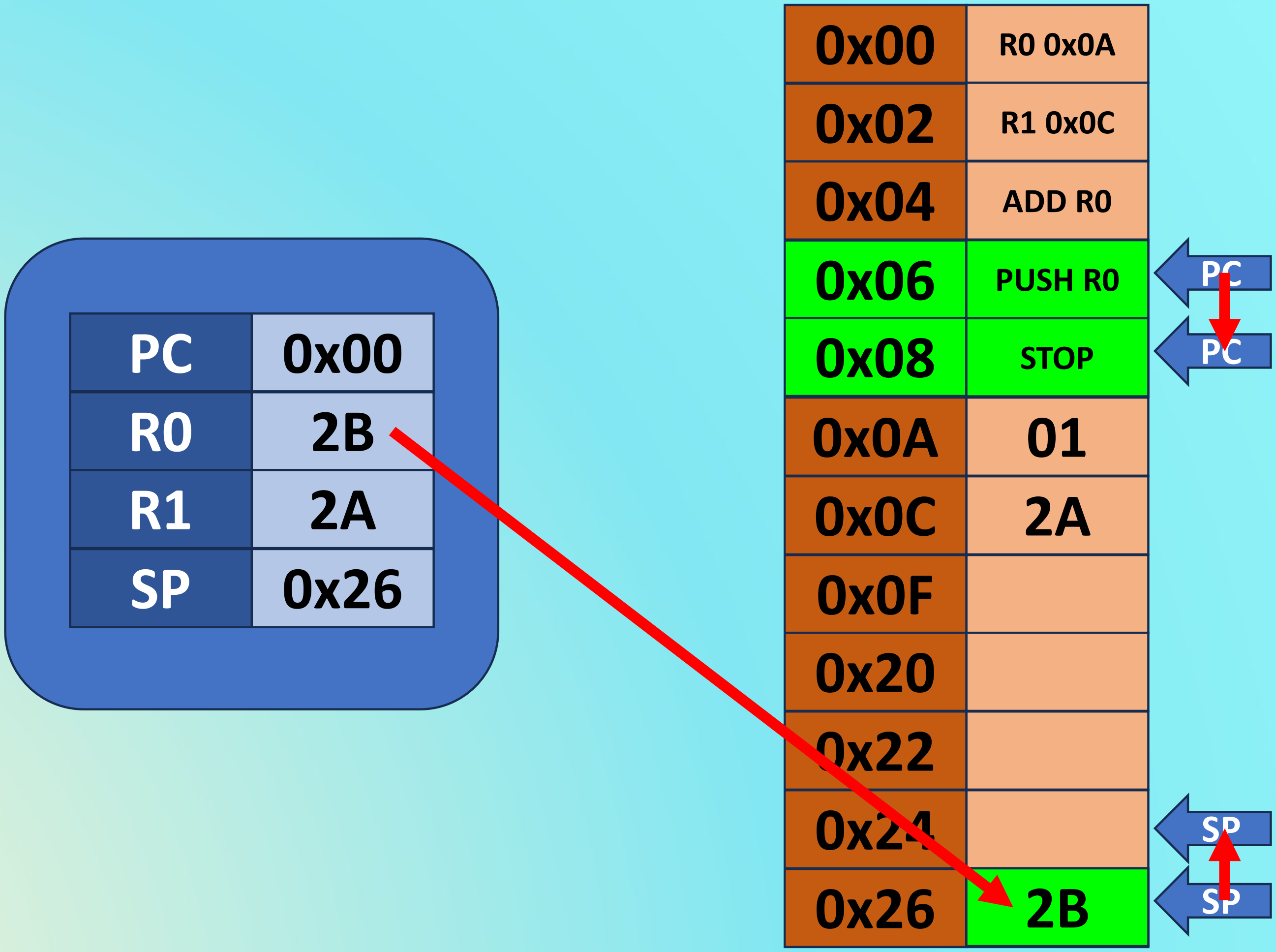
PC	0x00
R0	
R1	
SP	0x26

0x00	R0 0x0A	PC
0x02	R1 0x0C	
0x04	ADD R0	
0x06	PUSH R0	
0x08	STOP	
0x0A	01	
0x0C	2A	
0x0F		
0x20		
0x22		
0x24		
0x26		SP


PC
0x00
R0
01
R1
SP
0x26
0x00
R0 0x0A
0x02
R1 0x0C
0x04
ADD R0
0x06
PUSH R0
0x08
STOP
0x0A
01
0x0C
2A
0x0F
0x20
0x22
0x24
0x26
PC
PC
SP

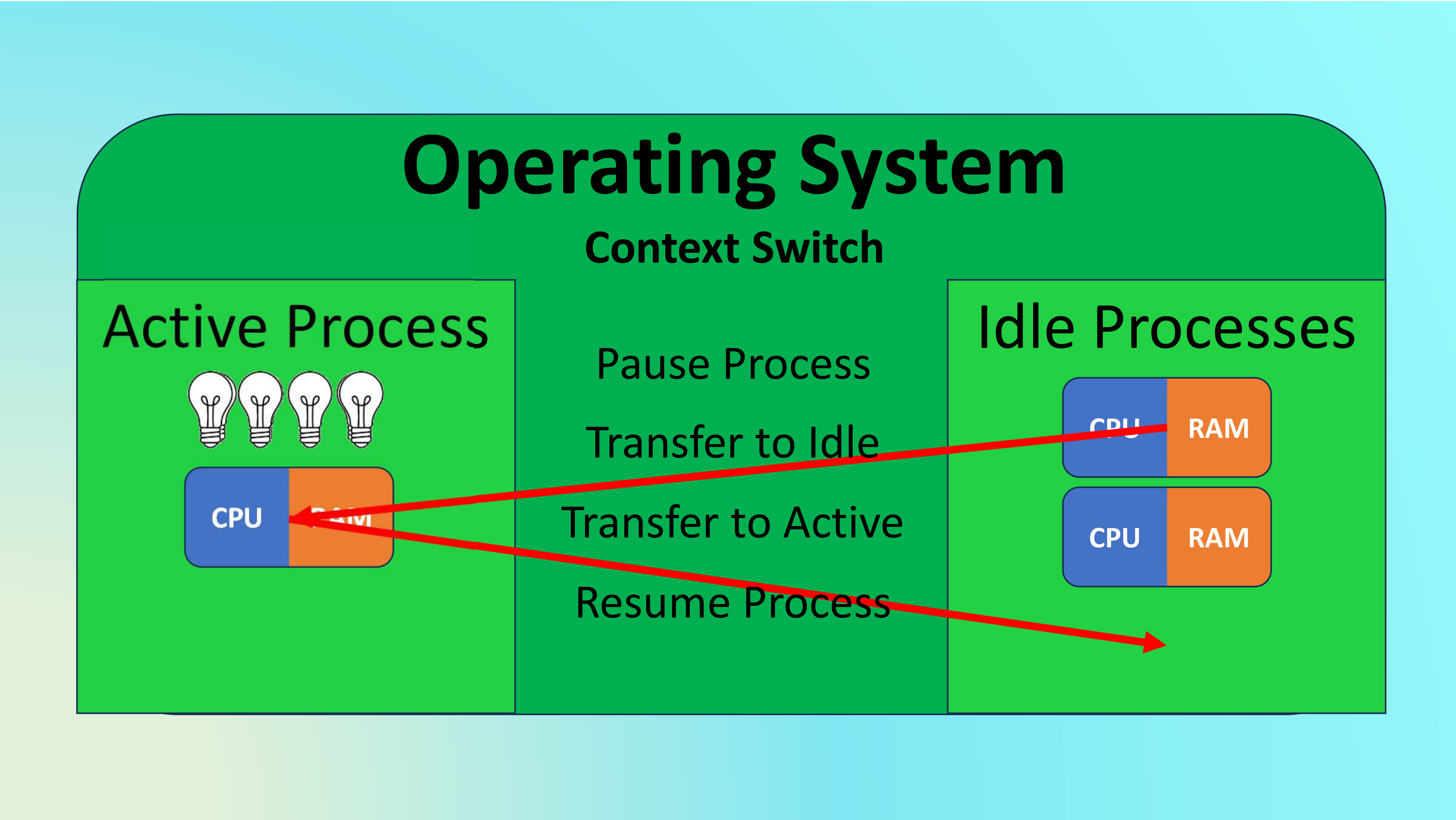

PC
0x00
R0
01
R1
2A
SP
0x26
0x00
R0 0x0A
0x02
R1 0x0C
0x04
ADD R0
0x06
PUSH R0
0x08
STOP
0x0A
01
0x0C
2A
0x0F
0x20
0x22
0x24
0x26
PC
PC
SP


PC
0x00
R0
2B
R1
2A
SP
0x26
0x00
R0 0x0A
0x02
R1 0x0C
0x04
ADD R0
0x06
PUSH R0
0x08
STOP
0x0A
01
0x0C
2A
0x0F
0x20
0x22
0x24
0x26
PC
PC
SP


PC
0x00
R0
2B
R1
2A
SP
0x26
0x00
R0 0x0A
0x02
R1 0x0C
0x04
ADD R0
0x06
PUSH R0
0x08
STOP
0x0A
01
0x0C
2A
0x0F
0x20
0x22
0x24
0x26
2B
PC
PC
SP
SP

PC	0x00
R0	2B
R1	2A
SP	0x26

0x00	R0 0x0A	
0x02	R1 0x0C	
0x04	ADD R0	
0x06	PUSH R0	
0x08	STOP	PC
0x0A	01	
0x0C	2A	
0x0F		
0x20		
0x22		
0x24		SP
0x26	2B	


Operating System
Context Switch
Active Process
CPU
Pause Process
Transfer to Idle
Transfer to Active
Resume Process
Idle Processes
RAM
CPU
RAM

Context Switch

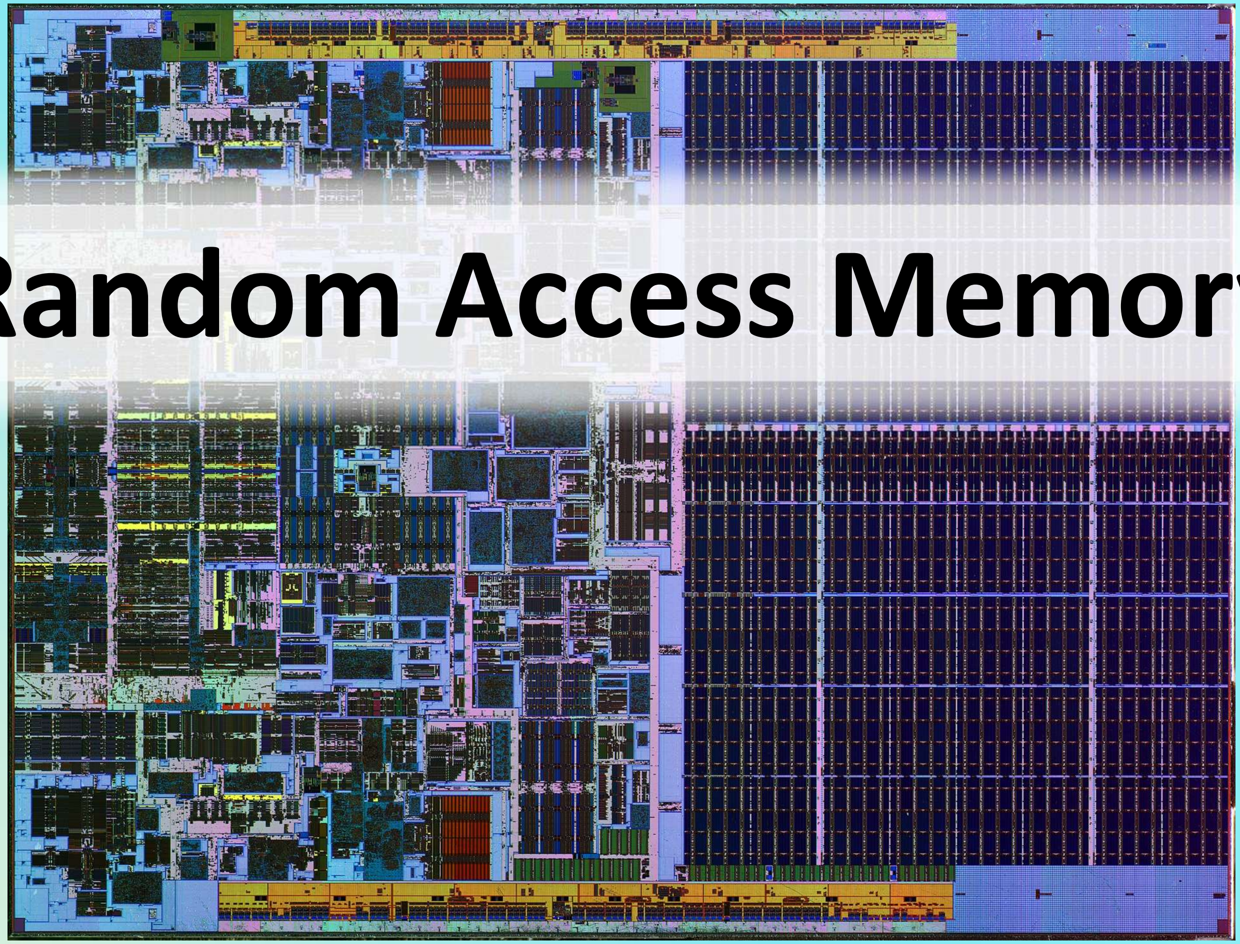
Process swapping <=> Context Switch

Creates CPU and memory overhead

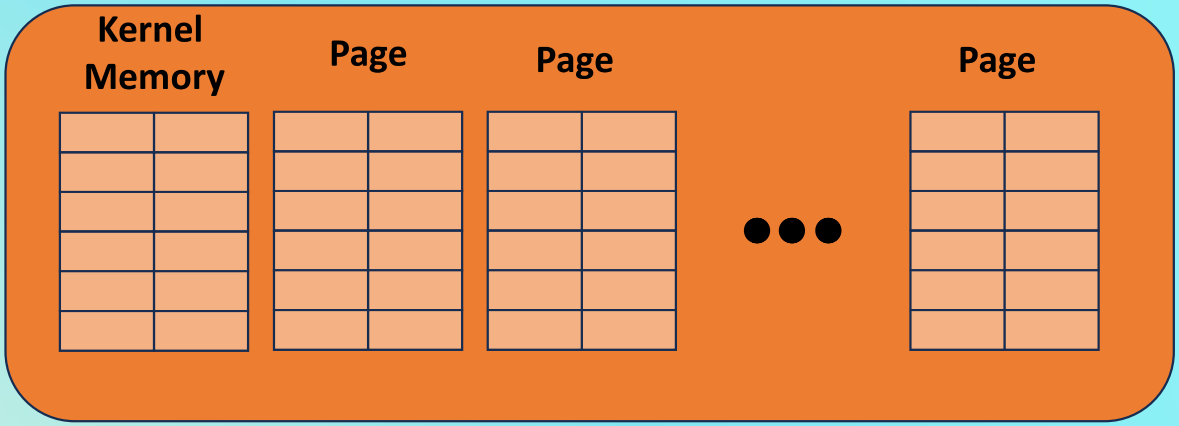
I/O

During I/O, yields and preemption

Random Access Memory

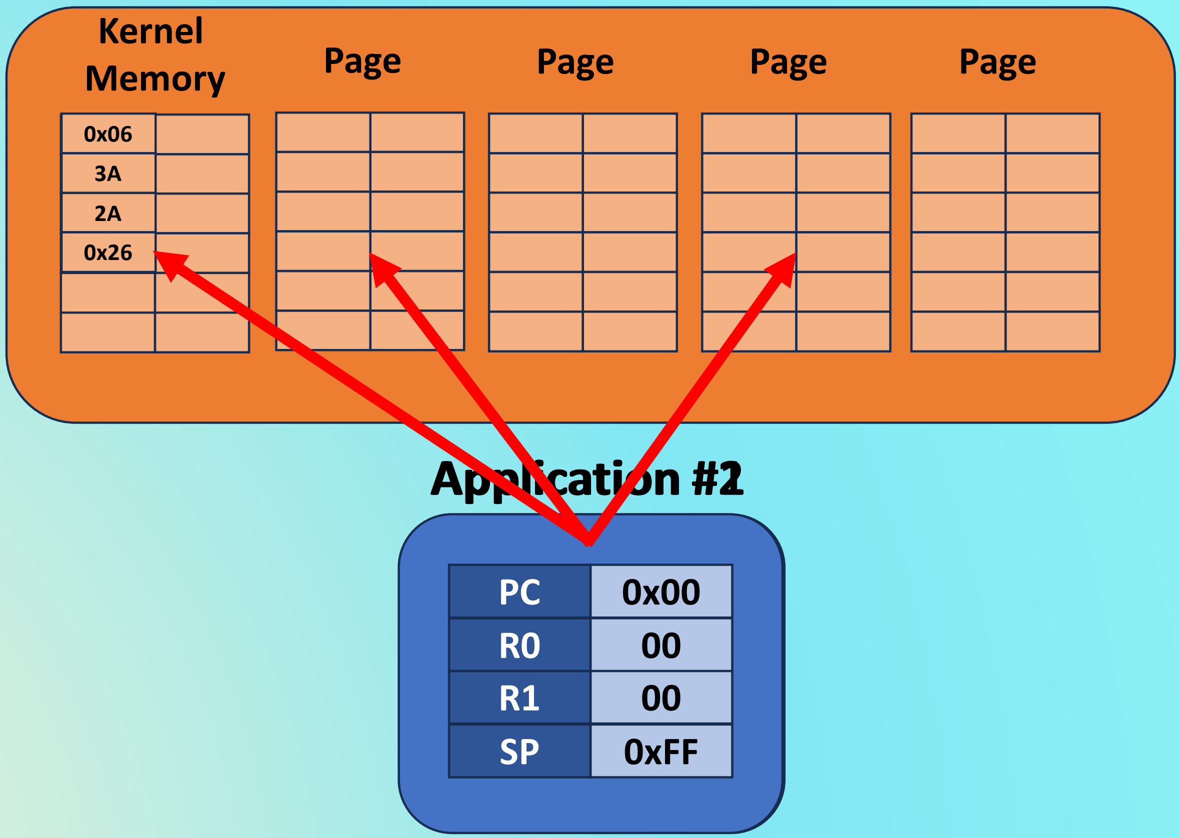


A Closer Look at RAM

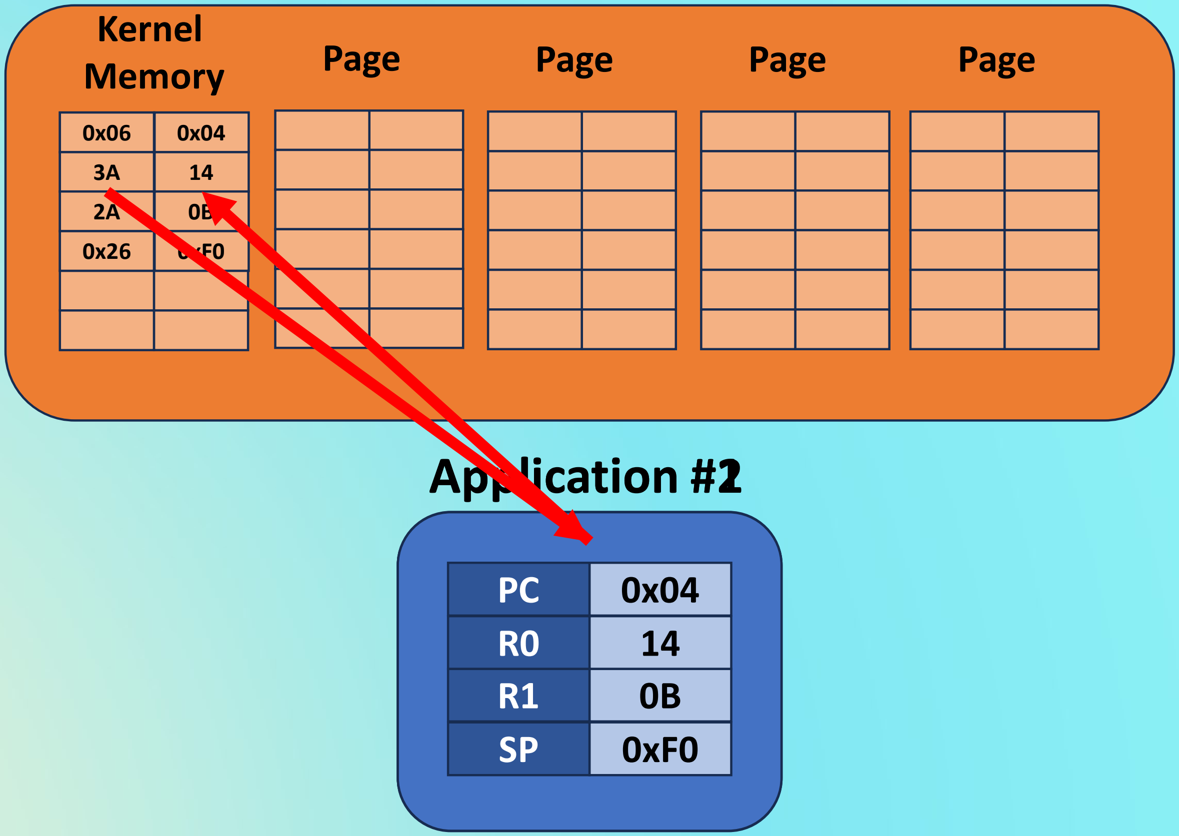


- **Pages are fixed-size memory blocks**, typically 4 KB.
- Each process gets its own set of pages.

Context Switch



Context Switch



Spawning Tasks

Creating Threads in Julia

@task

- **Coroutine**: They can pause and resume.
- **Green threads**: Managed by Julia, not the OS.
- **Lightweight threads**: Much cheaper to run than OS threads.

@task

```
julia> a = @task some_function(100)
Task (runnable) @0x000072e745614330
```

- **Wraps a block into a zero-arg anonymous function.**
- Syntax sugar for Task(() -> ...)
- Creates a Task object but does not schedule it.
- Captures the surrounding scope via closure.

@task

```
julia> a = @task some_function(100)
Task (runnable) @0x000072e745614330
```

- Wraps a block into a zero-arg anonymous function.
- **Syntax sugar for Task(() -> ...)**
- Creates a Task object but does not schedule it.
- Captures the surrounding scope via closure.

@task

```
julia> a = @task some_function(100)
Task (runnable) @0x000072e745614330
```

- Wraps a block into a zero-arg anonymous function.
- Syntax sugar for Task(() -> ...)
- **Creates a Task object but does not schedule it.**
- Captures the surrounding scope via closure.

@task

```
julia> a = @task some_function(100)
Task (runnable) @0x000072e745614330
```

- Wraps a block into a zero-arg anonymous function.
- Syntax sugar for Task(() -> ...)
- Creates a Task object but does not schedule it.
- **Captures the surrounding scope via closure.**


Demo
Time

schedule

```
my_task = @task some_function(100)
schedule(my_task)
wait(my_task)

print(Threads.nthreads())
print(Threads.threadid)

~> julia --threads=4 my_program.jl
```

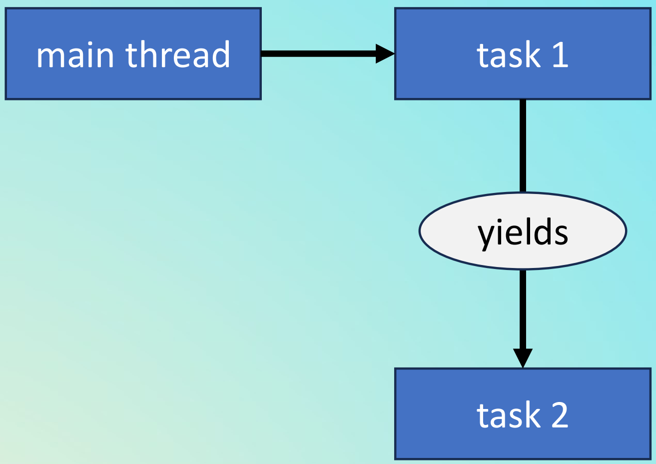
@spawn

```julia
task = @spawn some_function()
result = fetch(task) # or wait(task)
```

- Creates a **Task** that will run some_function()
- Schedules it to run on **any** available Julia thread.
- You use **fetch**(t) to get the result.

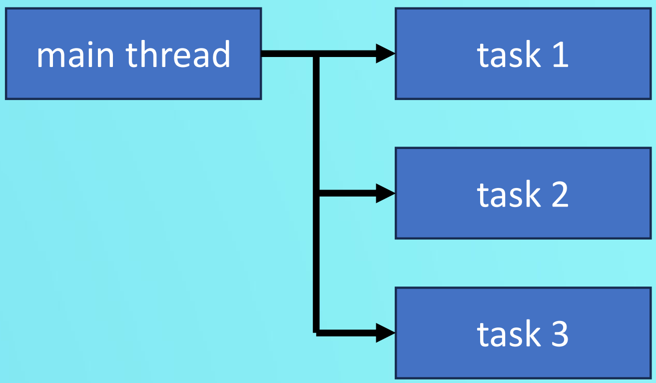
@task vs @spawn

@task with schedule



all tasks run on one thread
without parallelism

Threads.@spawn



all tasks run concurrently
on multiple threads

@threads

```
Threads.@threads for i = 1:10
    a[i] = Threads.threadid()
end
```

- Create parallel loops using the Threads.@threads macro.
- The iteration space is split among the threads.
- Fast, easy, and ideal for embarrassingly parallel tasks.

When not to use loop parallelism

- When loop iterations **depend on each other**.
- For **very small loops** — overhead can dominate
- In **distributed-memory environments** use @distributed *beyond our scope.


Demo
Time

Thread Safety and Synchronization

Race Condition

A race condition occurs when two threads use or change the same data at the same time, and the result depends on which one finishes first.

Race Condition

Bank Account

$100

$20

Transaction A

Lookup Balance ($100)

Calculate $100 - $80 = $20

Update Balance to $20

Race Condition

Bank Account

$100 $20 $30

Transaction A

Lookup Balance ($100)

Calculate $100 - $80 = $20

Update Balance to $20

Transaction B

Lookup Balance ($100)

Calculate $100 - $70 = $30

Update Balance to $30

Race Condition

A race condition occurs when two threads use or change the same data at the same time, and the result depends on which one finishes first.

- Shared resource
- Concurrent access
- At least one access must write

Race Conditions

```julia
using .Threads
counter = 0
function race_increment()
    Threads.@threads for i in 1:1000
        global counter += 1
    end
    println("Final counter value: $counter")
end
race_increment()
```


Demo
Time

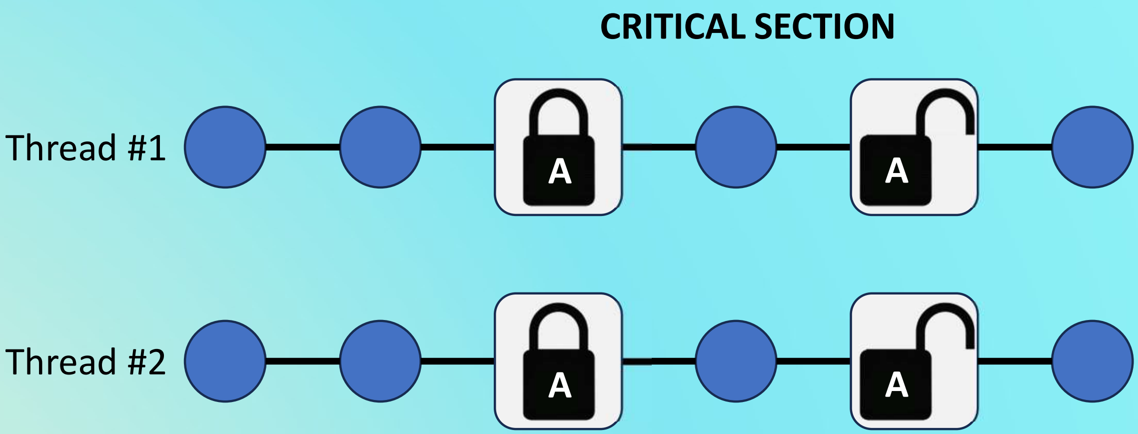
Mutex

Mutual Exclusion

```julia
mutex = ReentrantLock()
lock(mutex) do
    global counter += 1
end
```

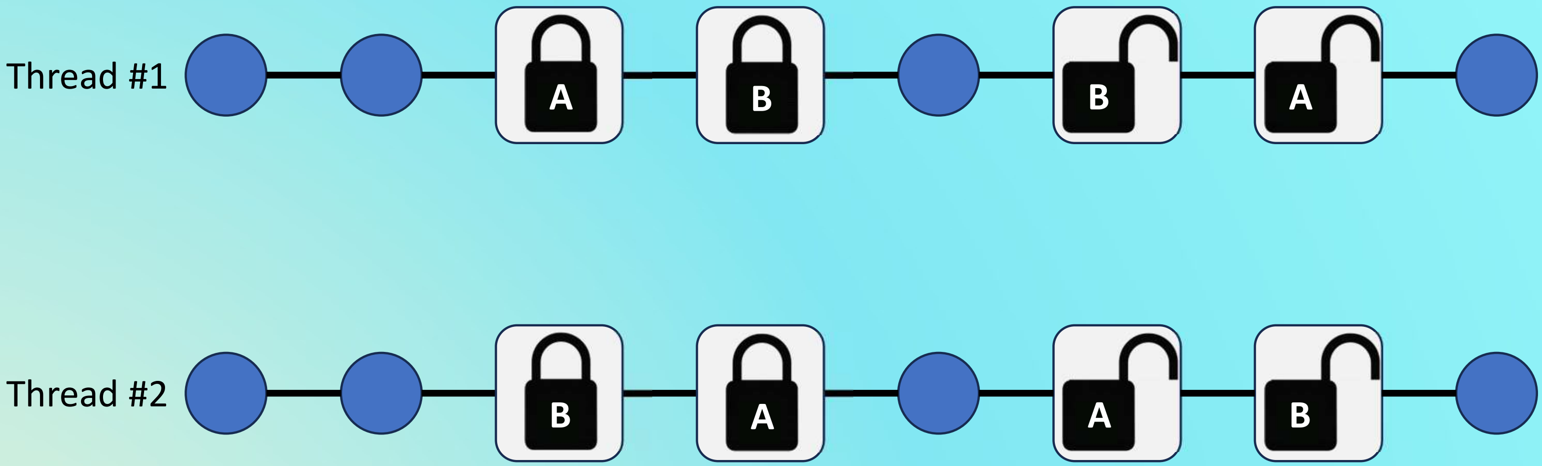
A synchronization primitive used to prevent multiple threads from accessing a shared resource at the same time.

Single Lock




Demo
Time

Multiple Locks

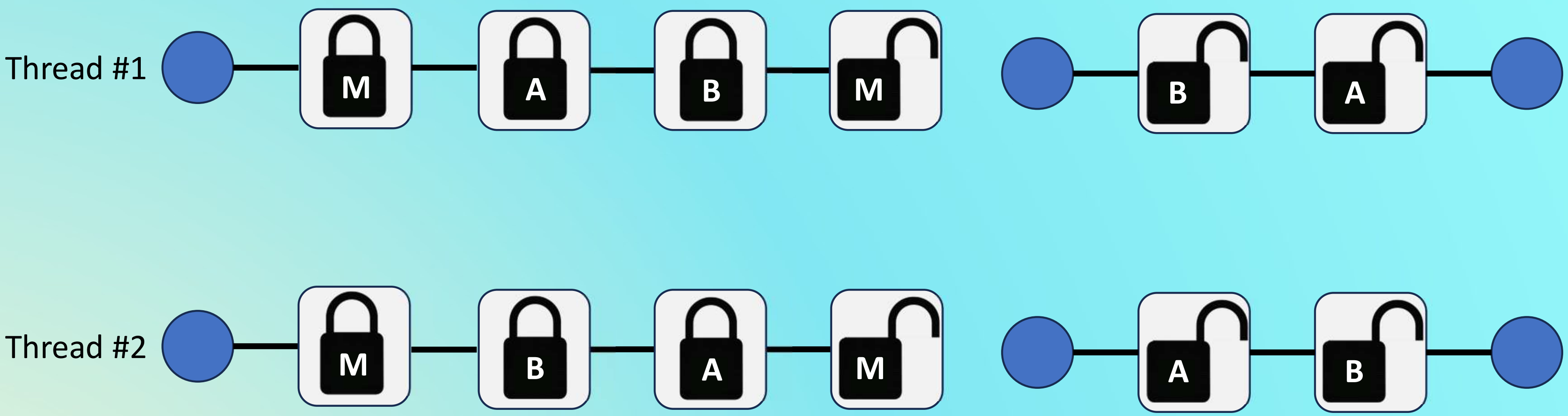


A deadlock occurs when two threads try to acquire two locks out of order.

Best Practices

- Try not to hold locks across long operations like I/O
- Start with a coarse-grained approach.
- Acquire locks in the same order.
- Use a lock manager to enforce order/queuing.
- Limit concurrent access & minimize threads.

Hierarchal Locks



A deadlock occurs when two threads try to acquire two locks out of order.

Atomic

```julia
x = Atomic{Int}(0)
```

Low overhead: No context switches or blocking — just one atomic CPU instruction.
Ideal for simple updates: Perfect for counters, flags, and single step operations.
Safe parallelism: Prevents data races without the complexity of lock/unlock logic.

Only for simple operations: Can't safely coordinate read-modify-write.
Harder to reason about: Subtle memory ordering and no built-in blocking or waiting.
Limited type support: Only works for primitive types.

Atomic Operations

Atomic{T}(value)

- <T> Int*, Uint*, Bool, Ptr (*Only available in system word size).

atomic_add!(x, value)
atomic_sub!(x, value)
atomic_xchg!(x, value)
atomic_cas!(x, expected, value)

Atomic Operations cont.

atomic_xchg!(x, value)

- Atomically replaces x with value and returns the original value.

atomic_cas!(x, expected, value)

- Atomically sets x to value if x currently equals expected; returns the old value.

x[]

- Retrieves the value in the atomic x.


Demo
Time

Fin