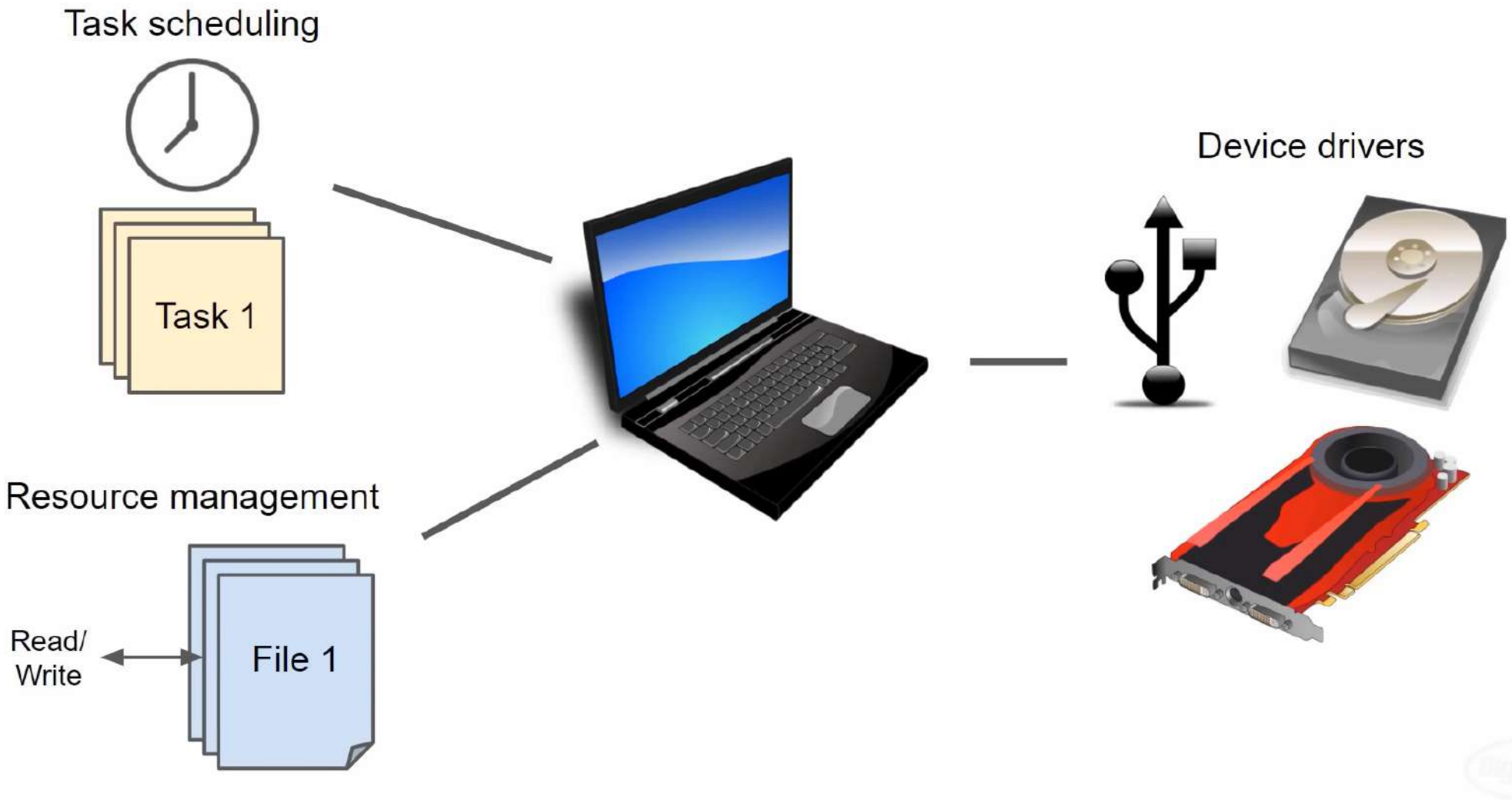
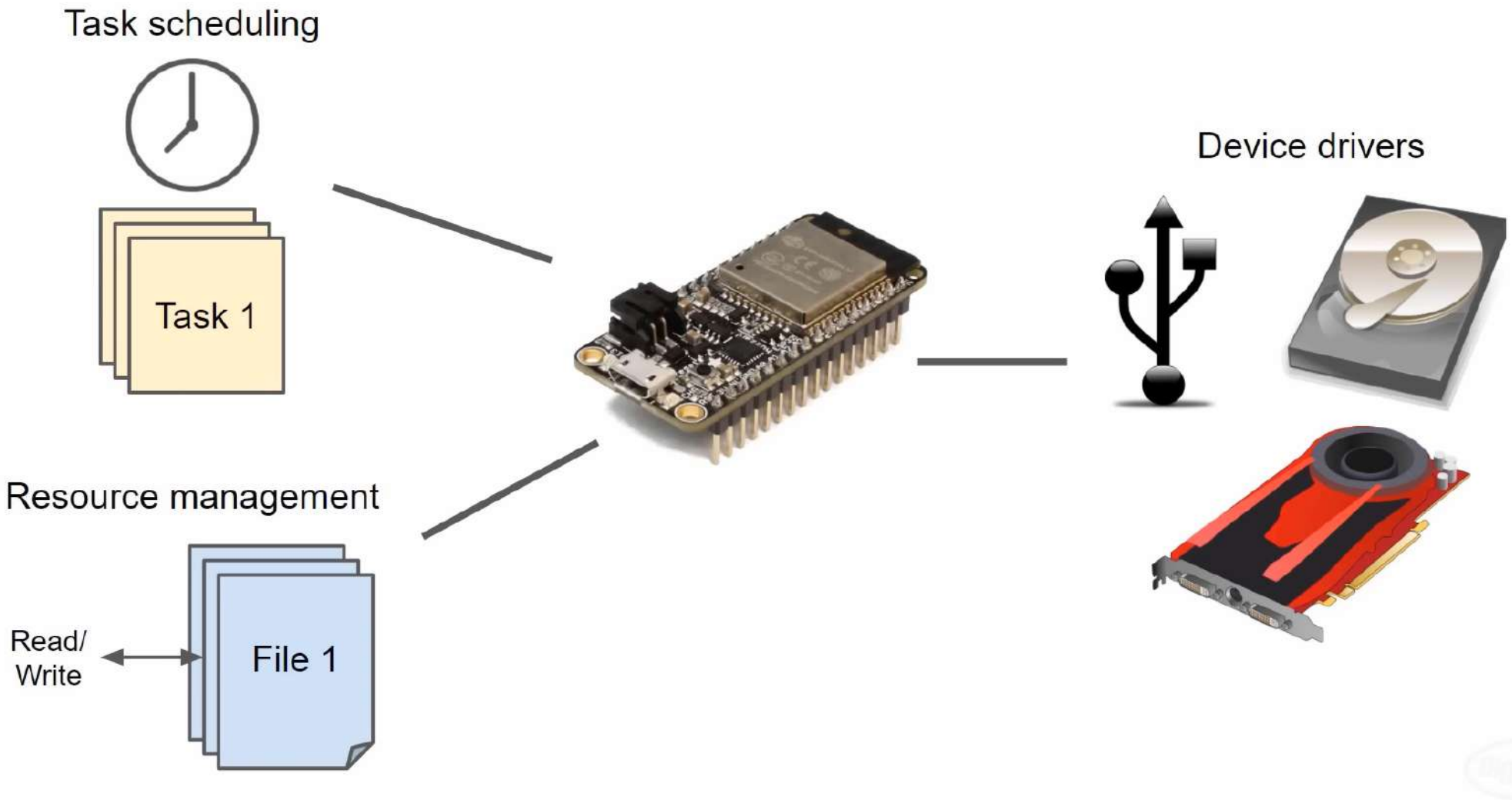
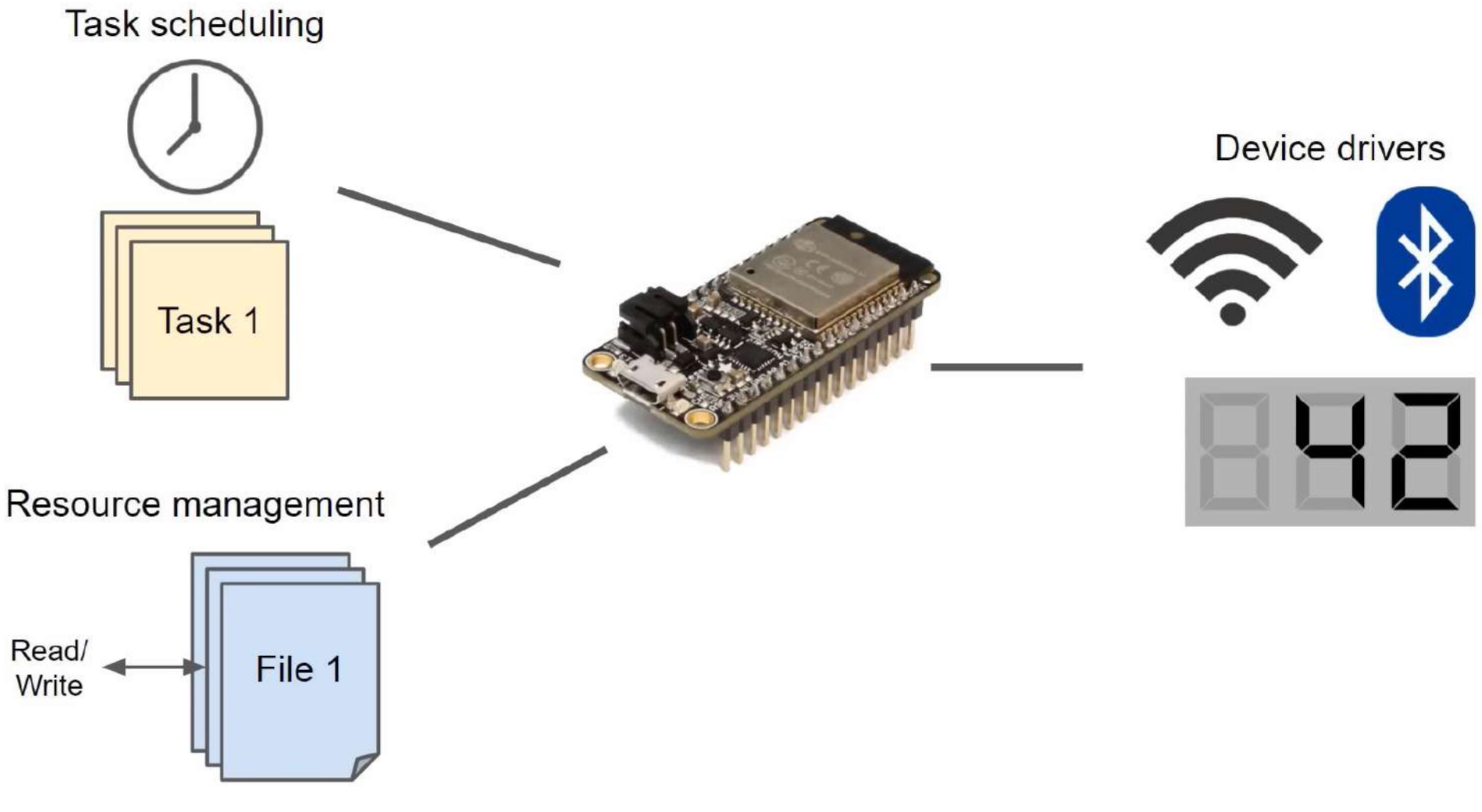
General Purpose Operating System (GPOS)



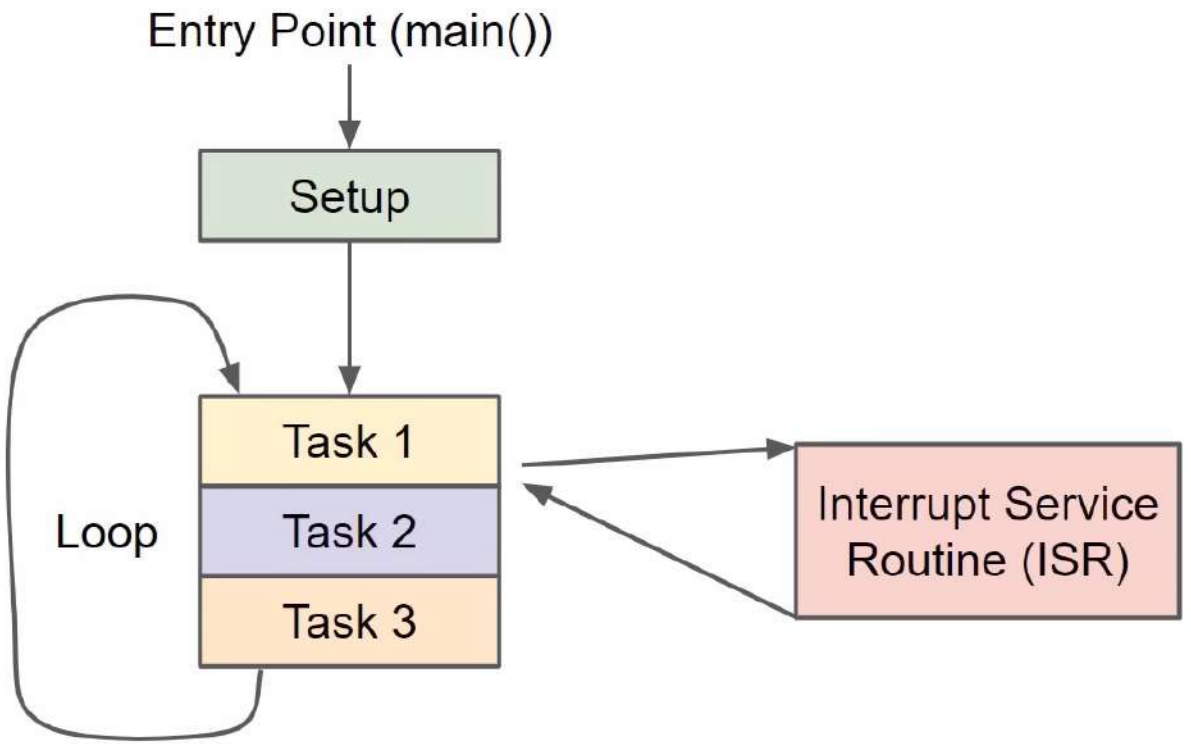
Real-Time Operating System (RTOS)



Real-Time Operating System (RTOS)



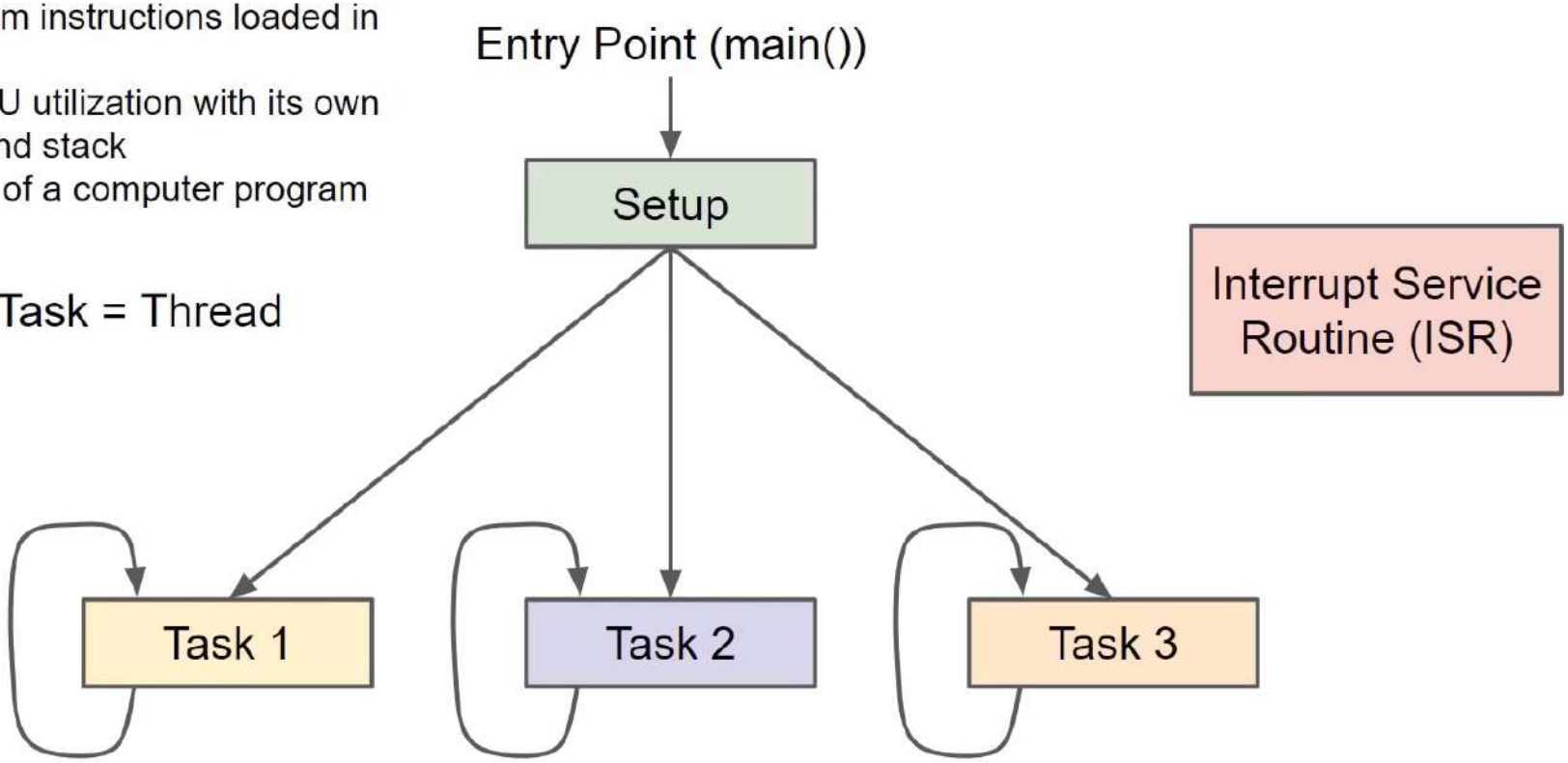
Super Loop


Entry Point (main())
Setup
Task 1
Task 2
Task 3
Loop
Interrupt Service Routine (ISR)

RTOS

- **Task**: set of program instructions loaded in memory
- **Thread**: unit of CPU utilization with its own program counter and stack
- **Process**: instance of a computer program

FreeRTOS: Task = Thread

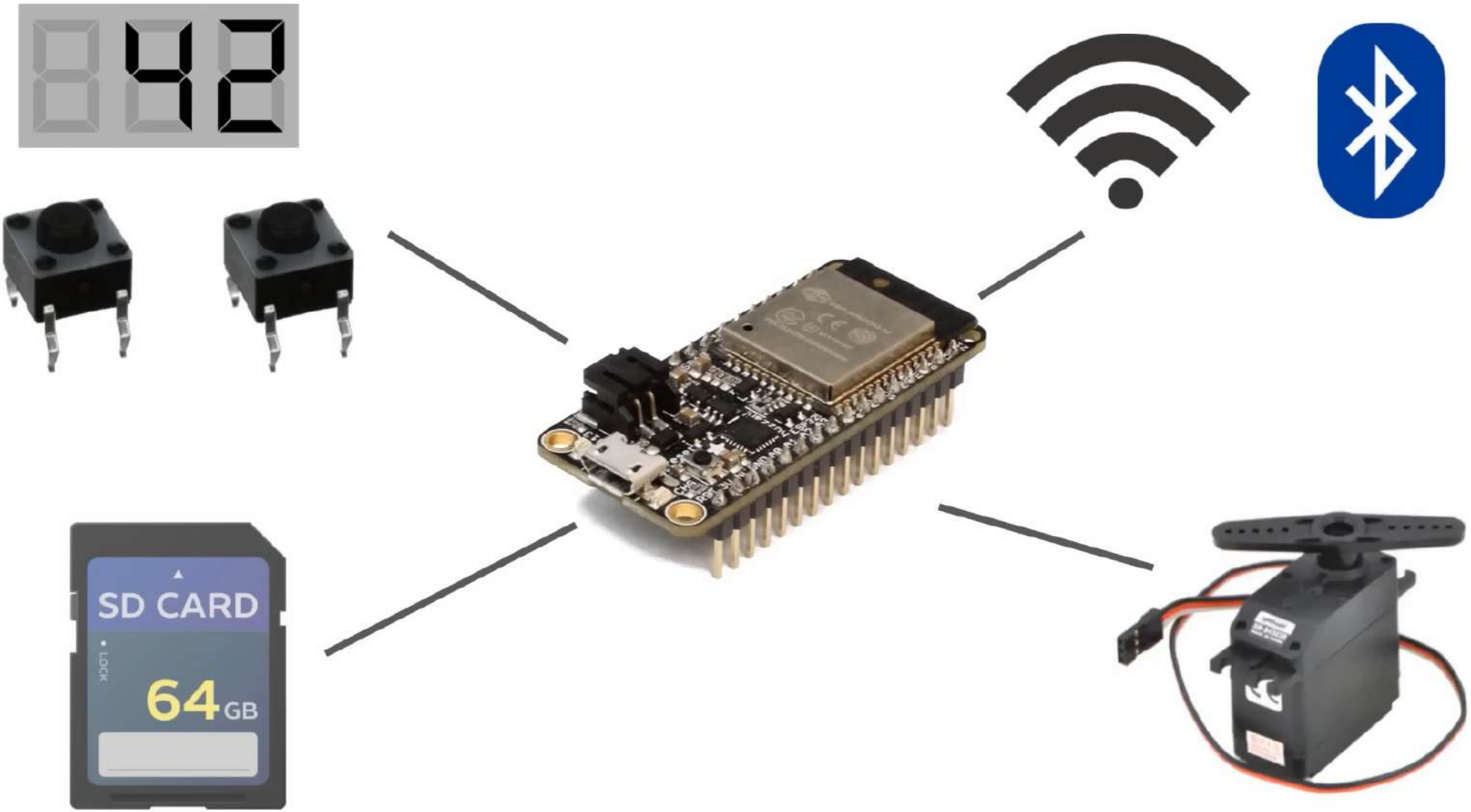


ATmega 328p
- 16 MHz
- 32 kB flash
- 2 kB RAM

STM32L476RG
- 80 MHz
- 1 MB flash
- 128 kB RAM

ESP-WROOM-32
- 240 MHz (dual core)
- 4 MB flash
- 520 kB RAM

Super Loop → RTOS


42
SD CARD
LOCK
64GB





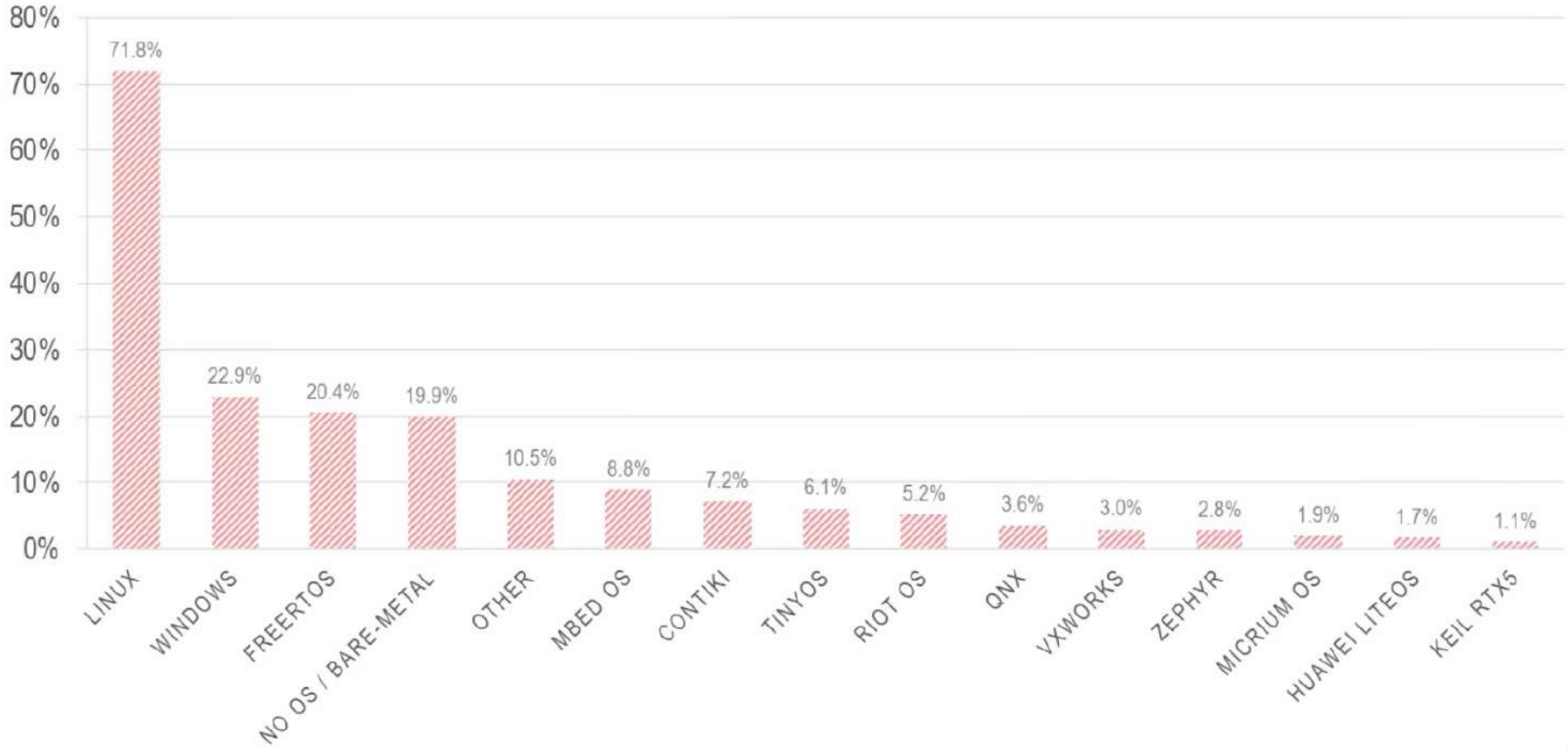

42

SD CARD
64 GB

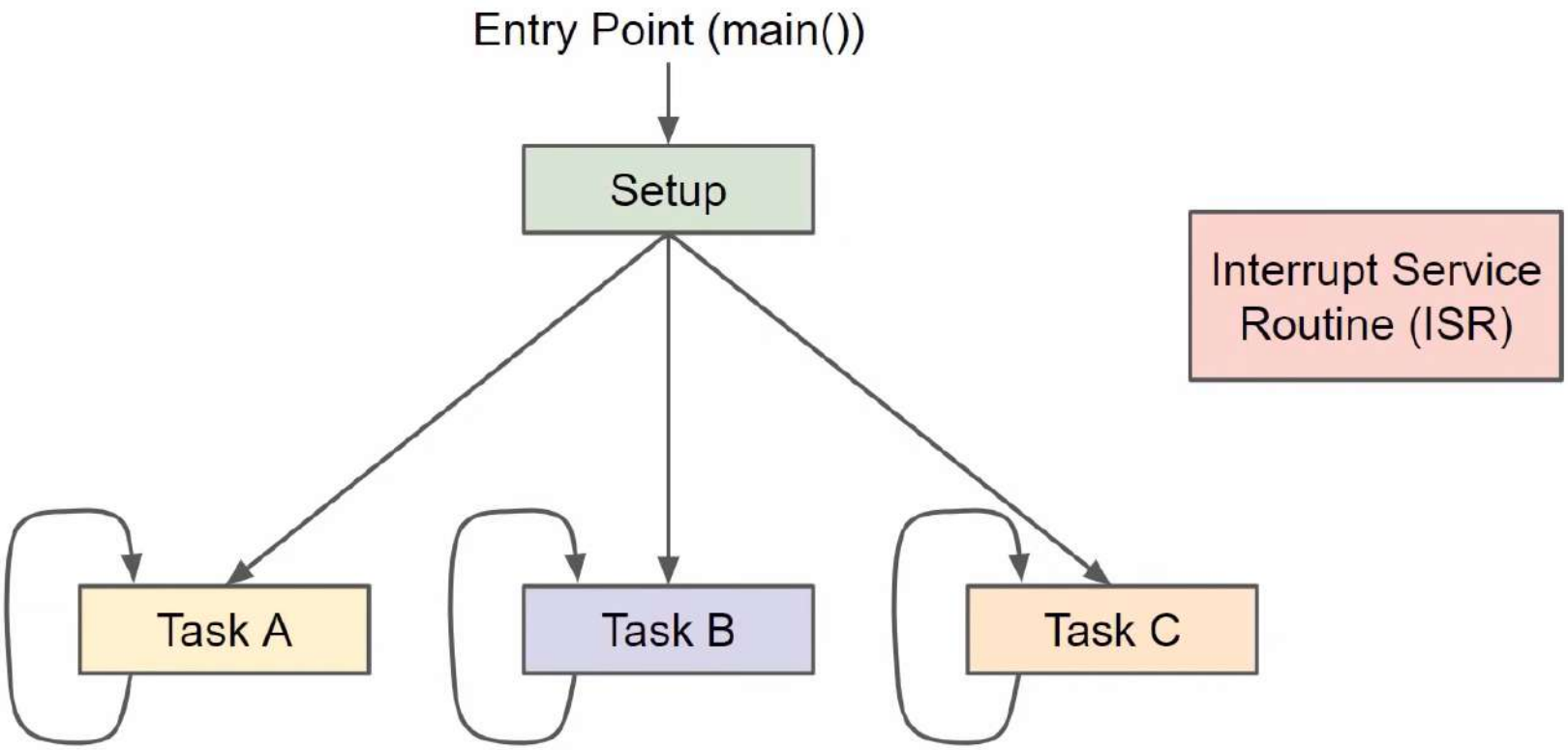


IoT Operating Systems

Which operating system(s) do you use for your IoT devices?

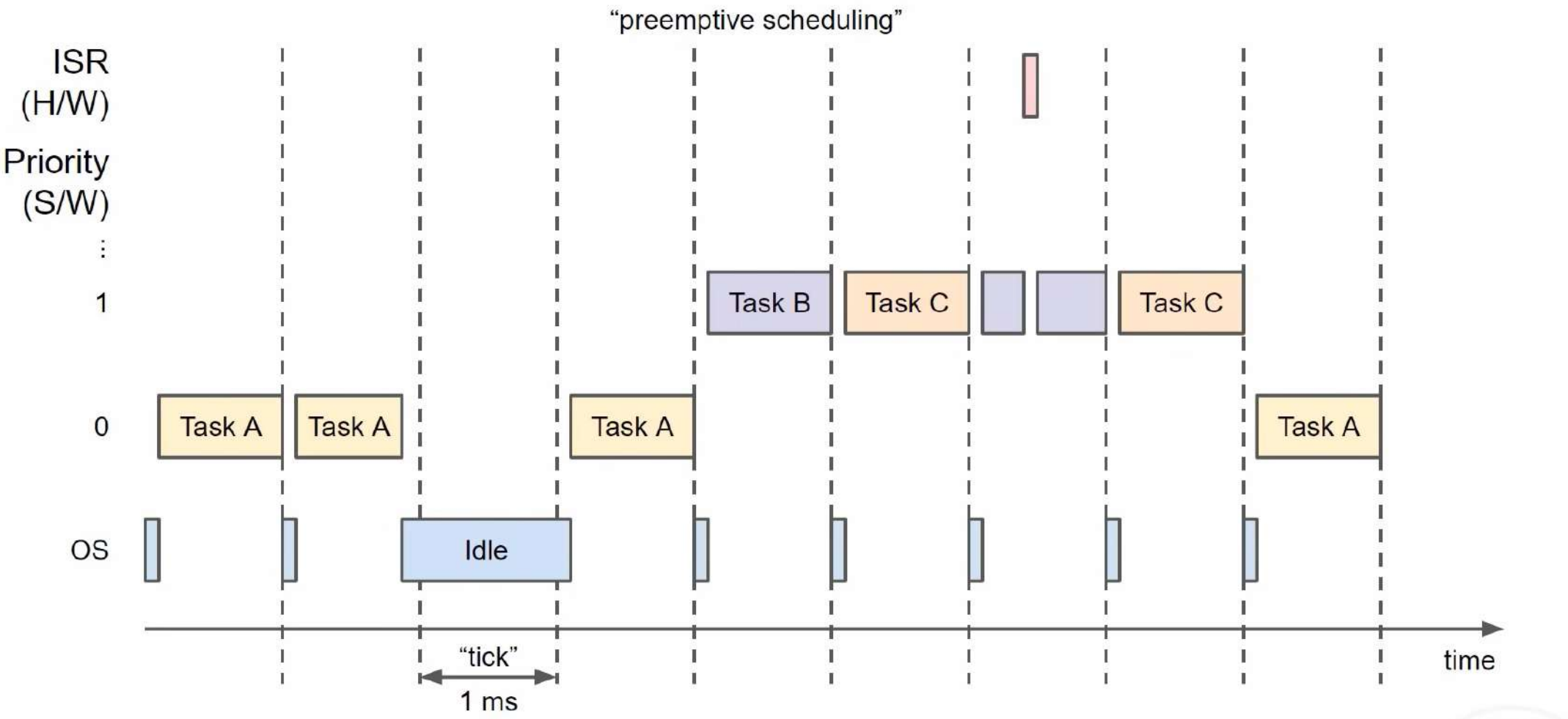


What our code looks like

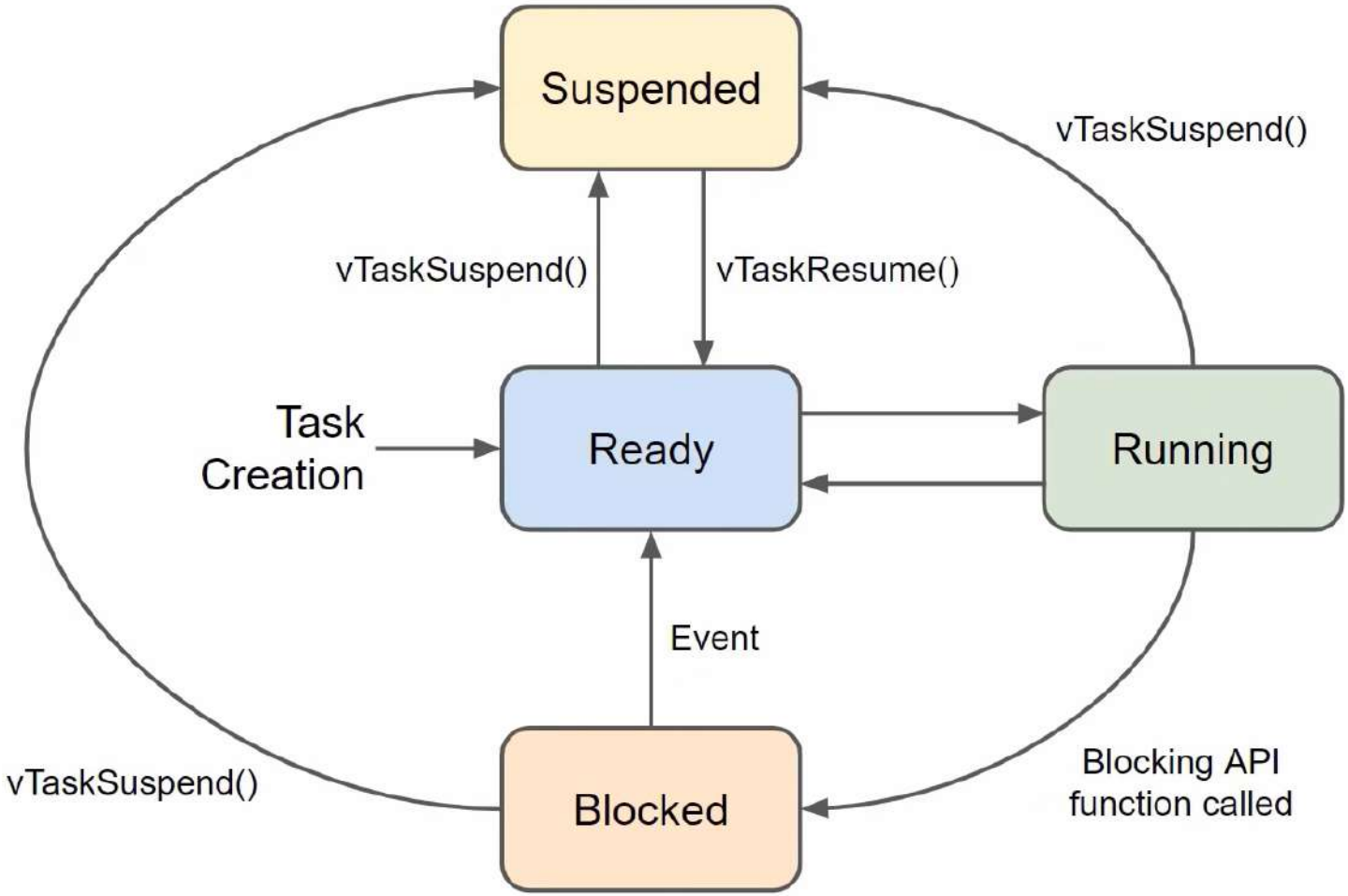


What actually happens*

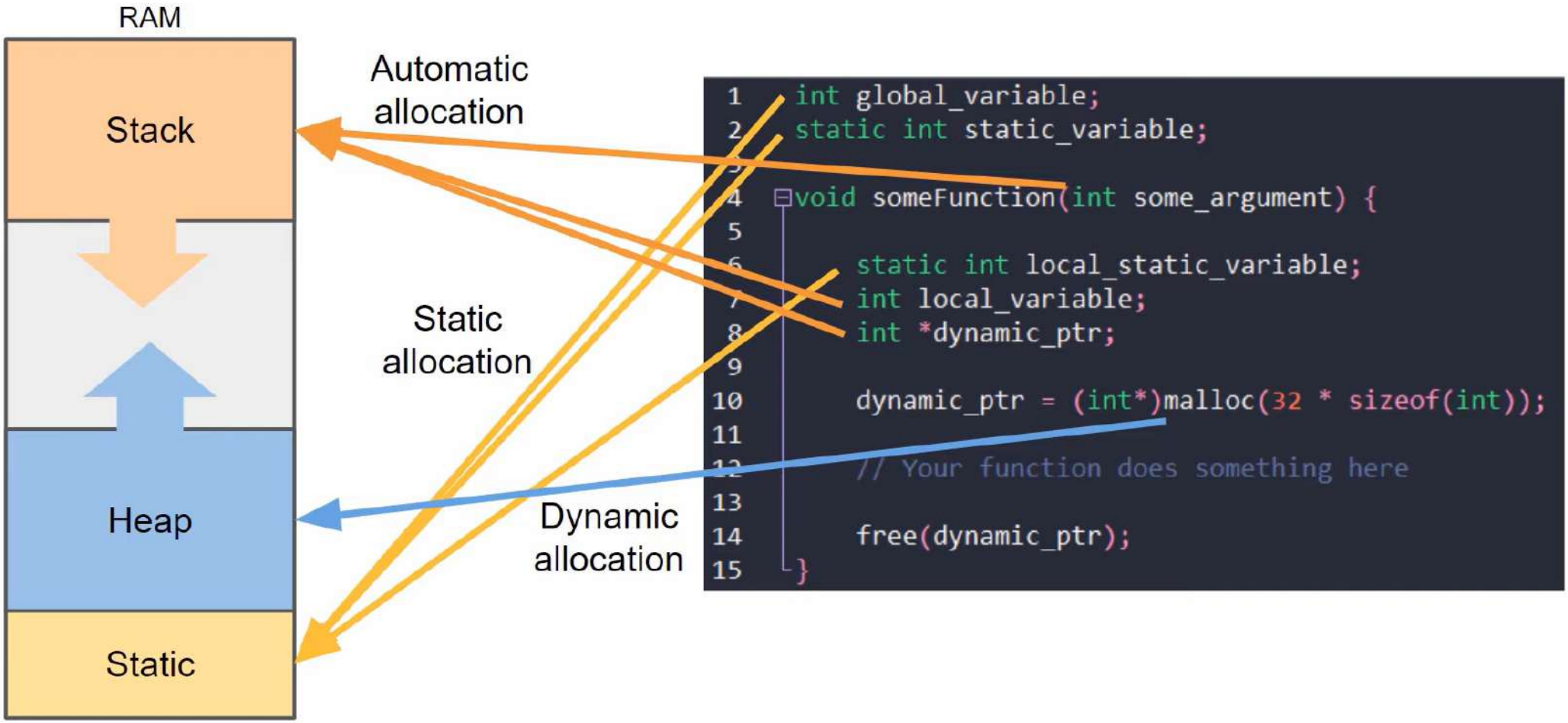
*assuming single-core processor



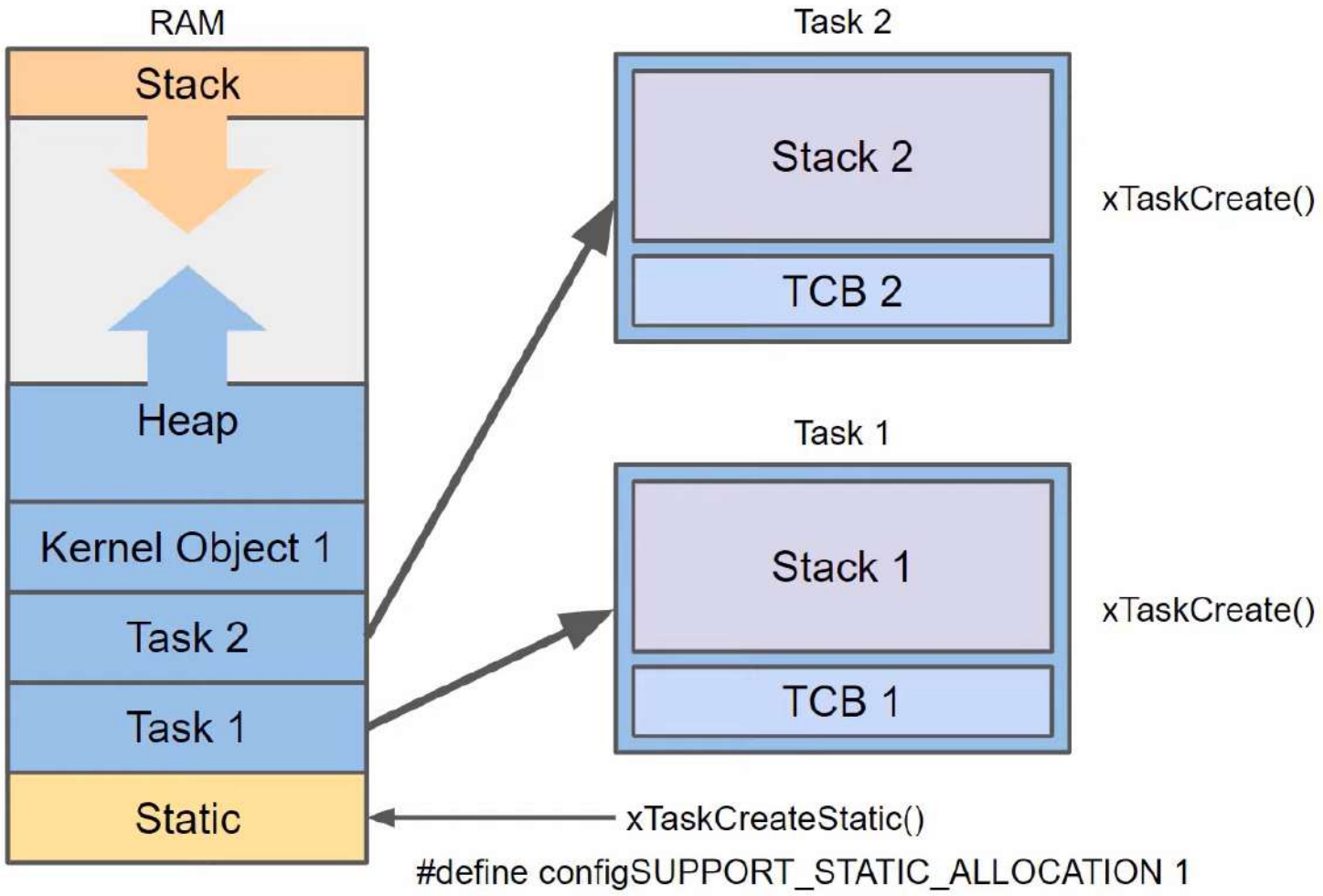
Task States


Suspended
vTaskSuspend()
vTaskSuspend()
vTaskResume()
Task
Creation
Ready
Running
Event
vTaskSuspend()
Blocked
Blocking API
function called

Memory Allocation



RTOS Memory Allocation



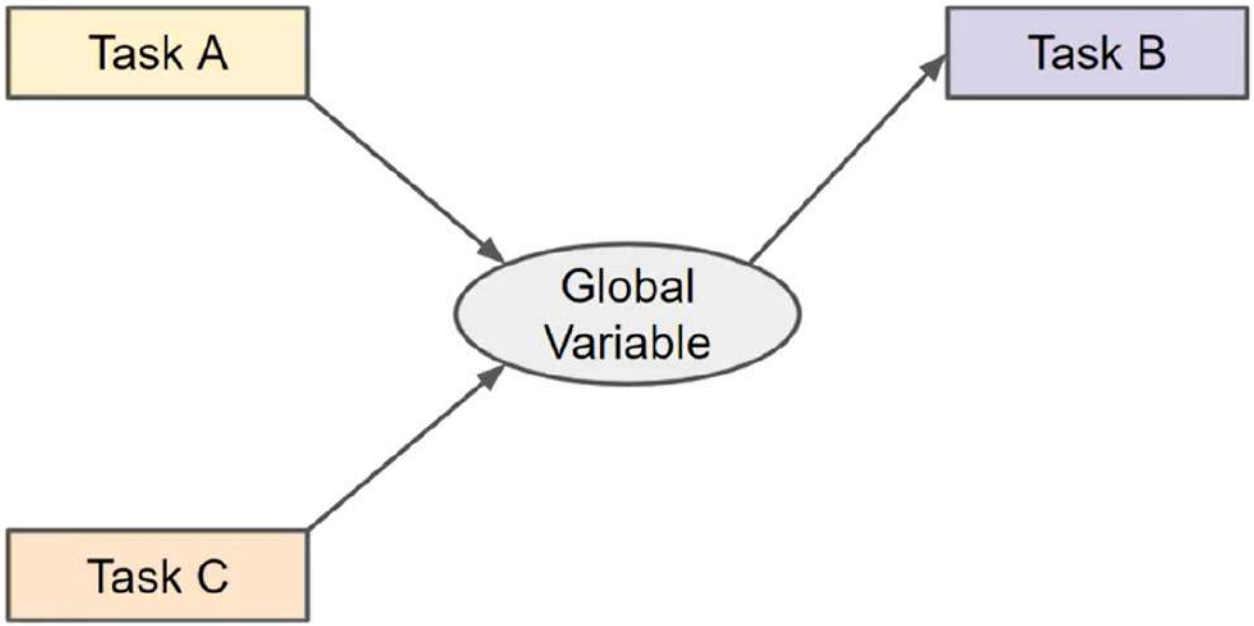
Challenge: Pass a Message

Task A

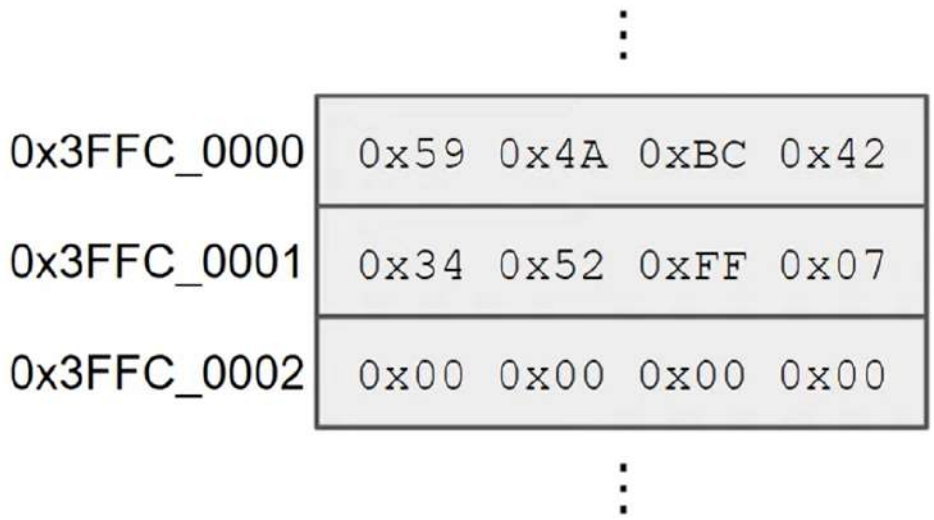
- Listens for input from Serial Monitor
- On newline char ('\n'), stores all chars up to that point in heap memory
- Notifies Task B of new message

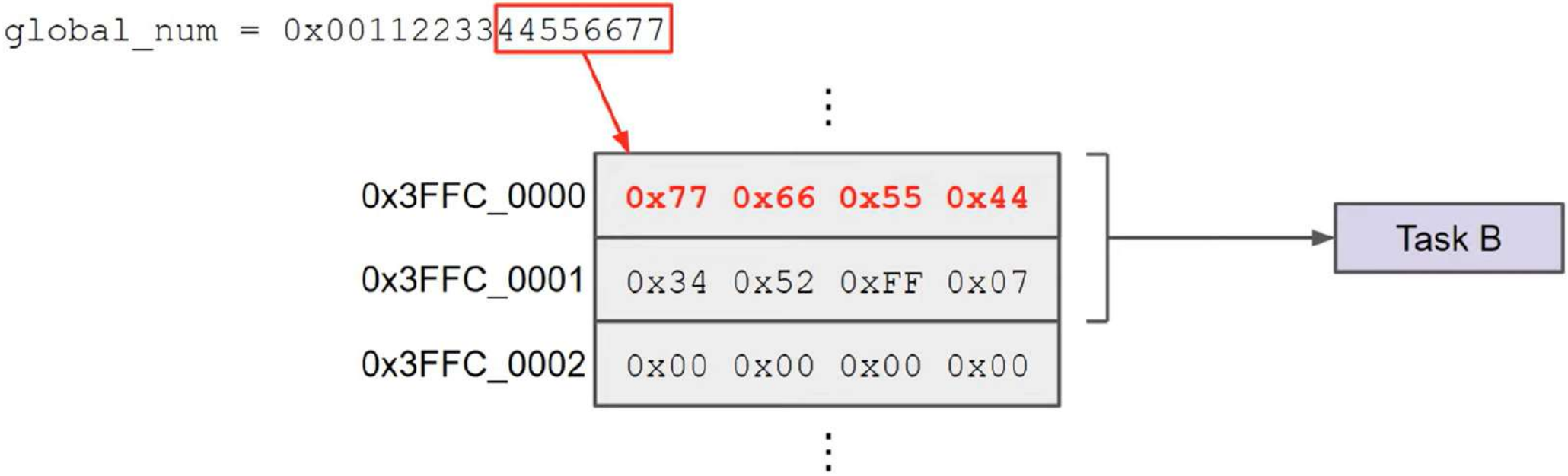
Task B

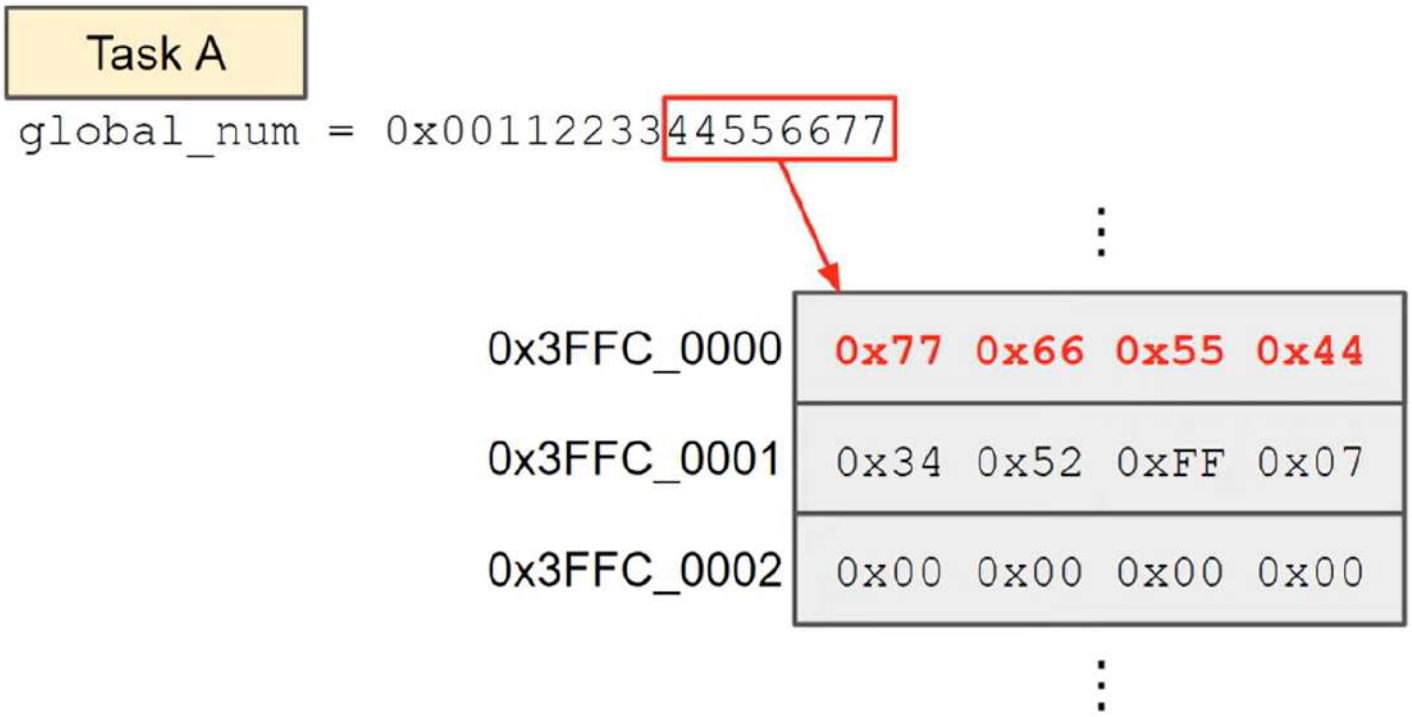
- Waits for notification from Task A
- Prints message found in heap memory to Serial Monitor
- Frees heap memory

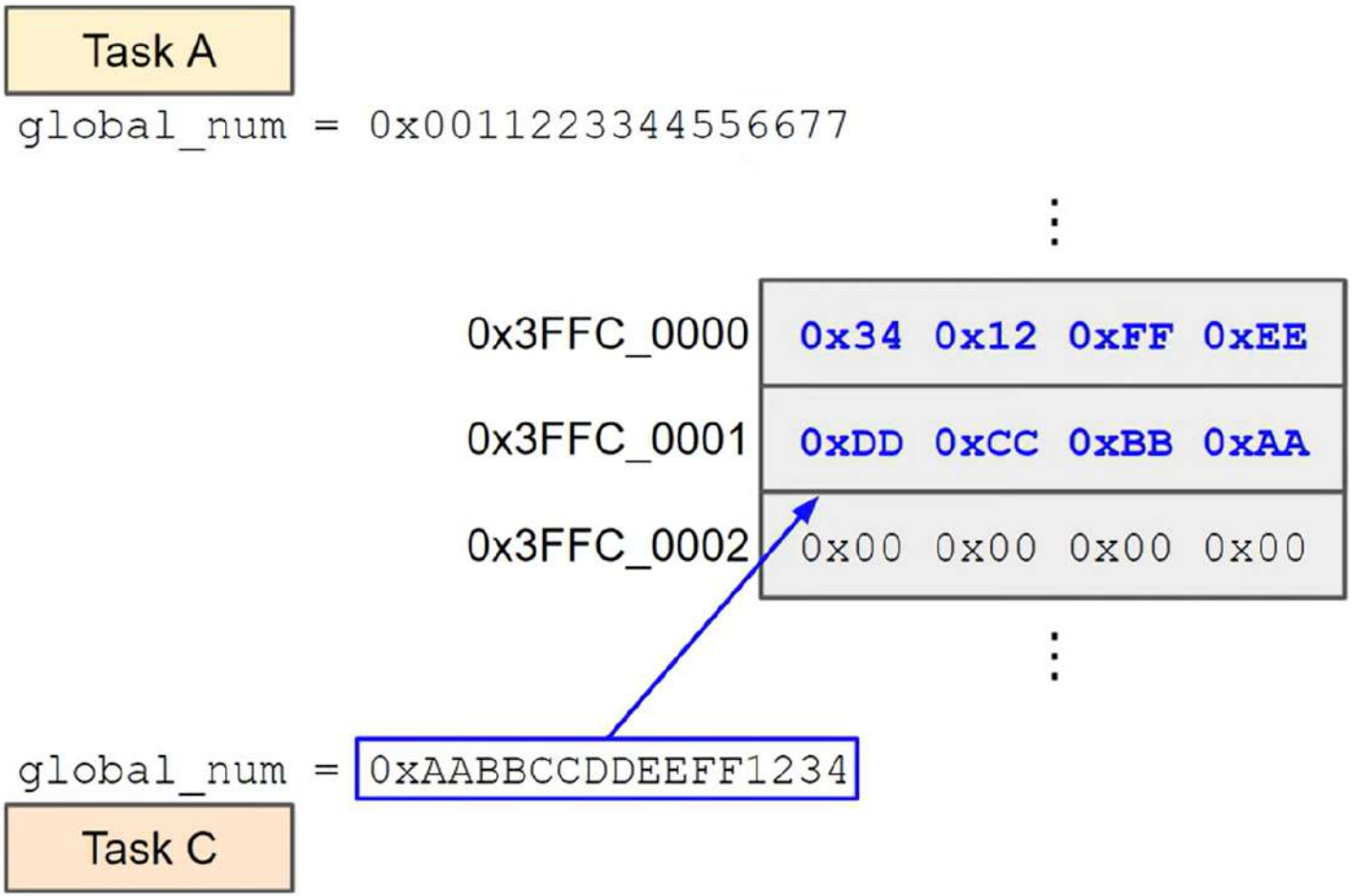

Task A
Task B
Global
Variable
Task C

```
global_num = 0x0011223344556677
```




global_num = 0x0011223344556677
0x3FFC_0000
0x77 0x66 0x55 0x44
0x3FFC_0001
0x34 0x52 0xFF 0x07
0x3FFC_0002
0x00 0x00 0x00 0x00
Task B

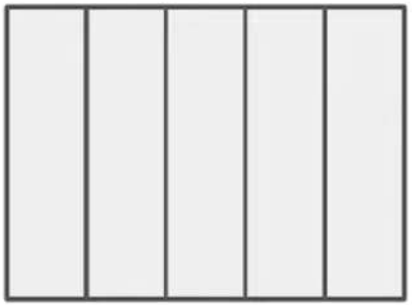

Task A
global_num = 0x0011223344556677
0x3FFC_0000
0x77 0x66 0x55 0x44
0x3FFC_0001
0x34 0x52 0xFF 0x07
0x3FFC_0002
0x00 0x00 0x00 0x00


Task A
global_num = 0x0011223344556677
0x3FFC_0000
0x34 0x12 0xFF 0xEE
0x3FFC_0001
0xDD 0xCC 0xBB 0xAA
0x3FFC_0002
0x00 0x00 0x00 0x00
global_num = 0xAABBCCDDEEFF1234
Task C


Task A
global_num = 0x00112233 44556677
0x3FFC_0000
0x34 0x12 0xFF 0xEE
0x3FFC_0001
0x33 0x22 0x11 0x00
0x3FFC_0002
0x00 0x00 0x00 0x00
global_num = 0xAABBCCDDEEFF1234
Task C

Task A

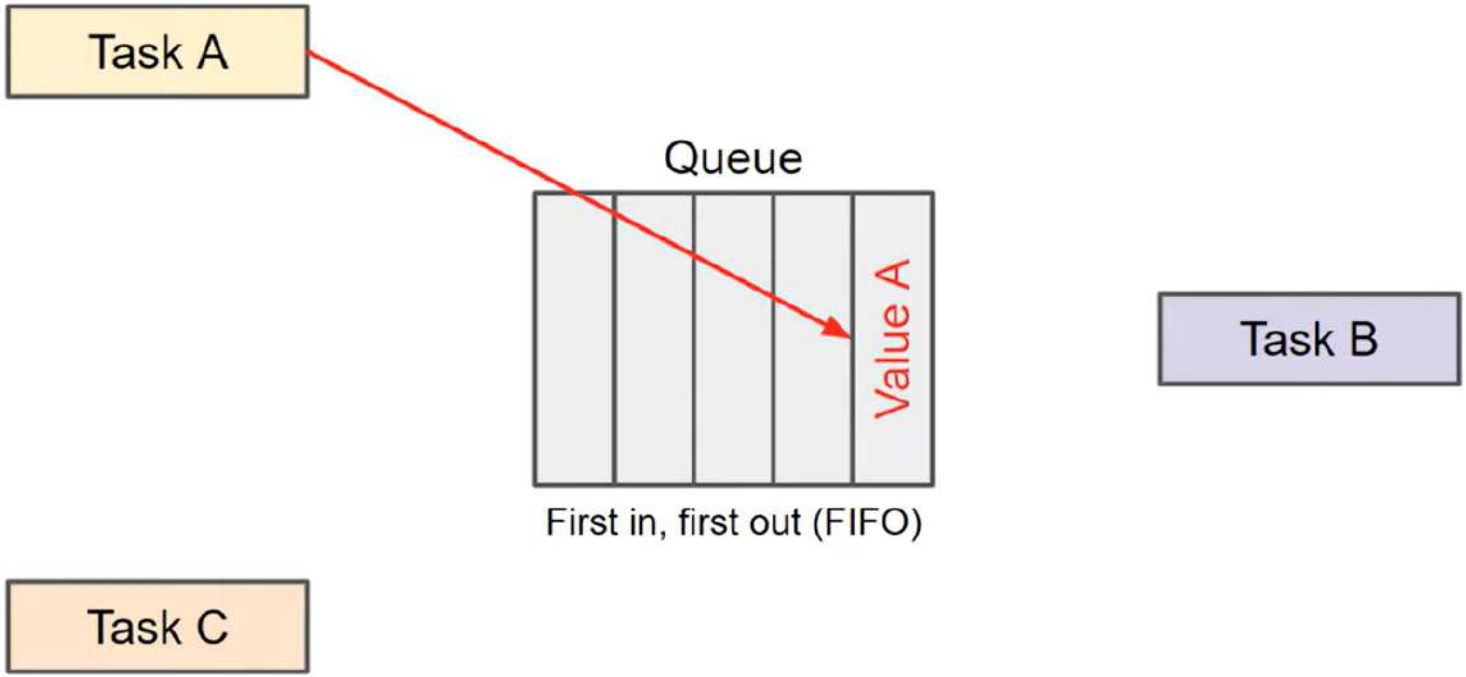
Queue

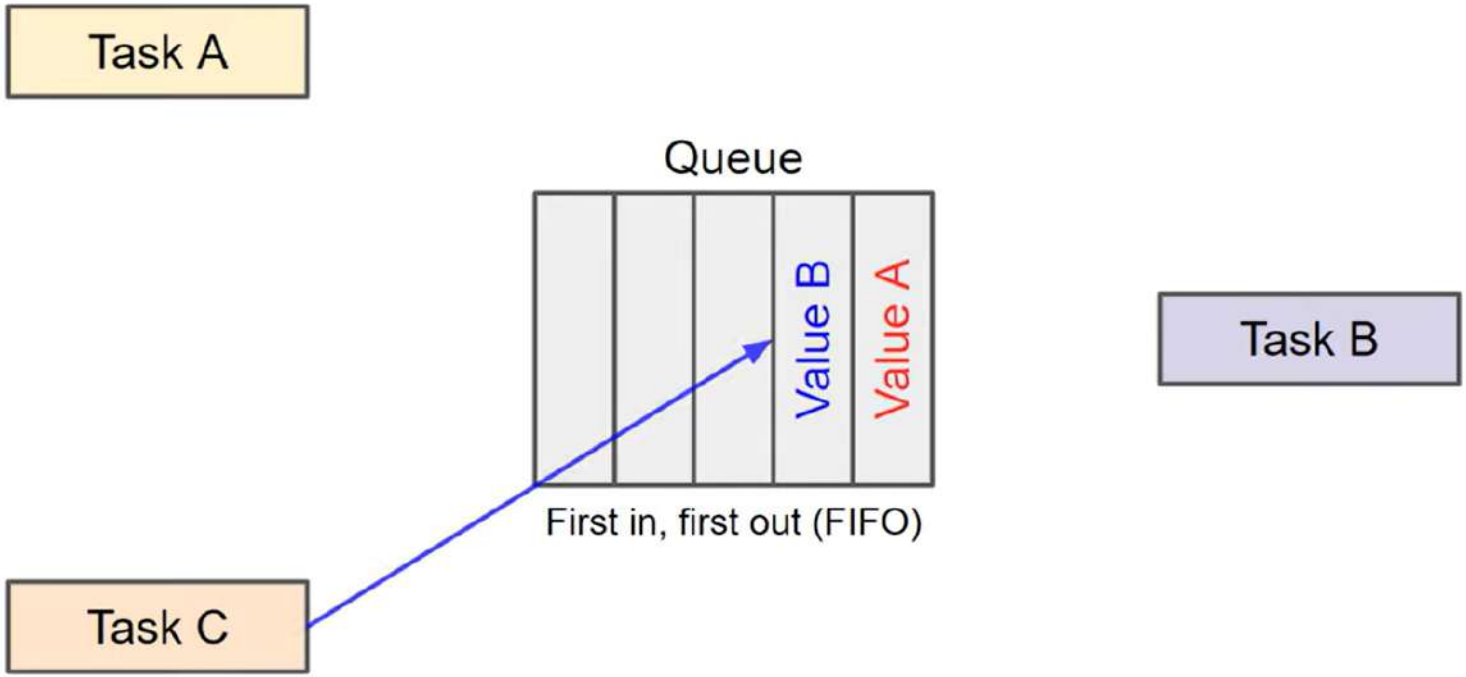


First in, first out (FIFO)

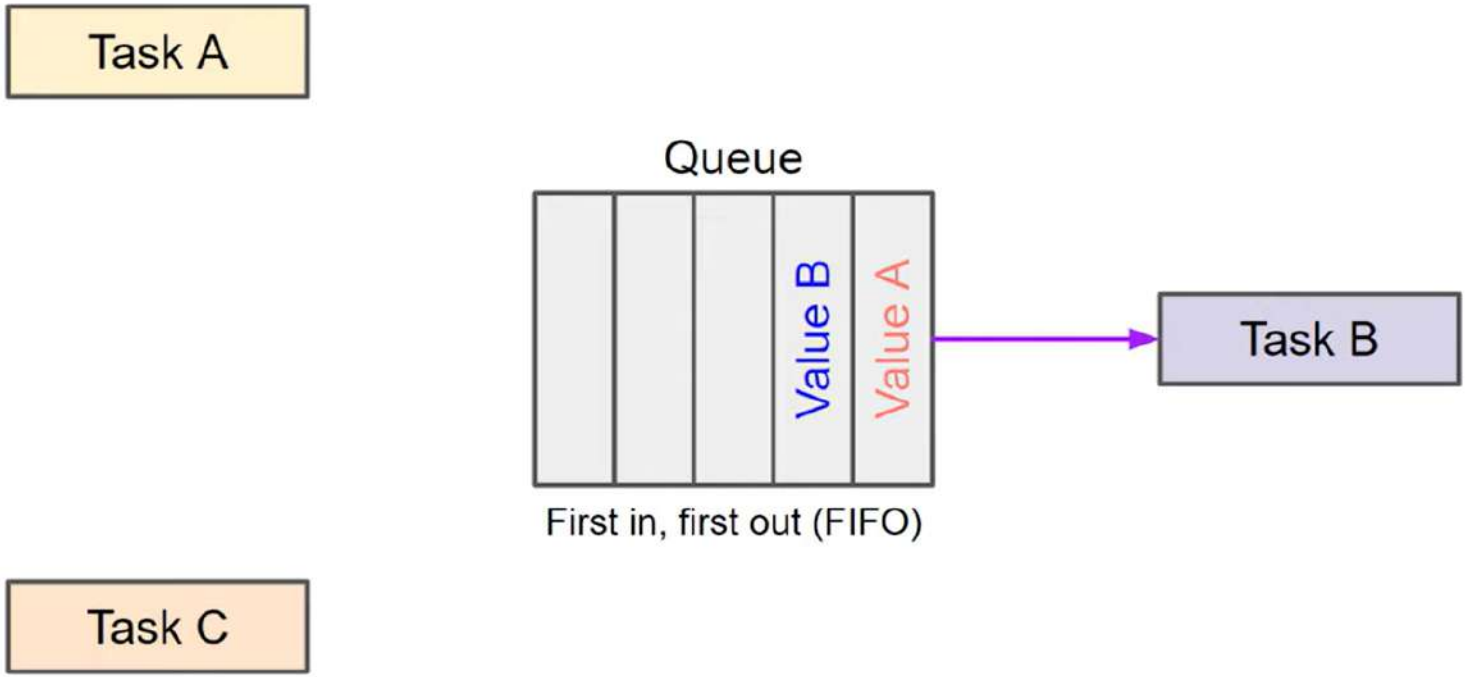
Task B

Task C

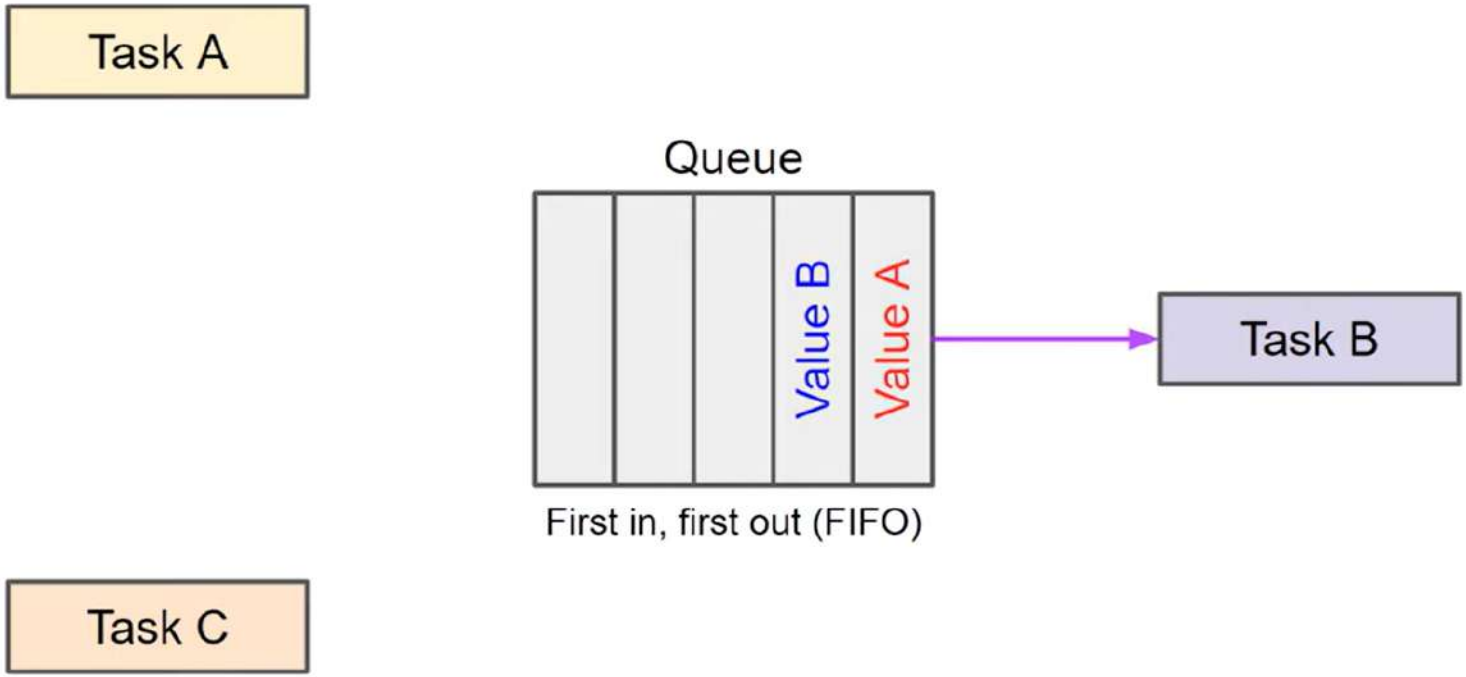

Task A
Queue
Value A
Task B
First in, first out (FIFO)
Task C


Task A
Queue
Value B
Value A
Task B
First in, first out (FIFO)
Task C

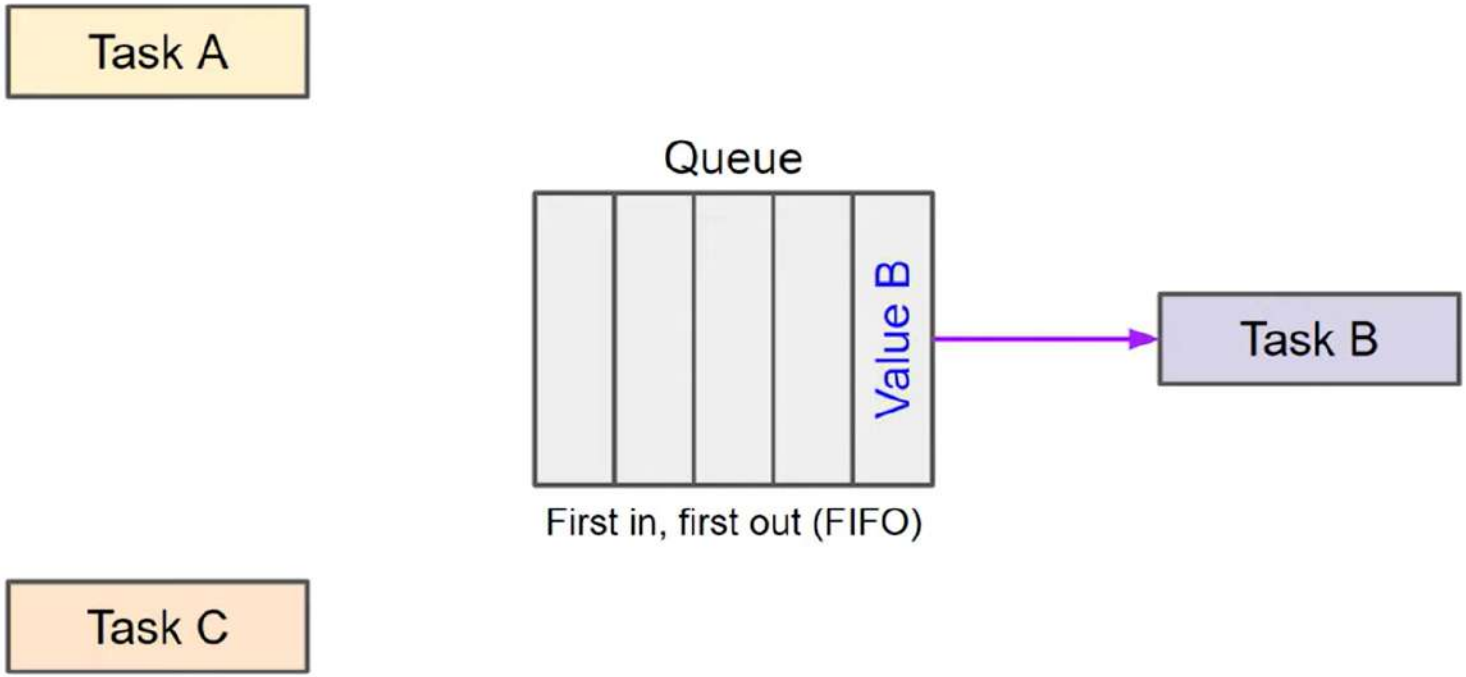
Copy by value, not by reference!



Copy by value, not by reference!

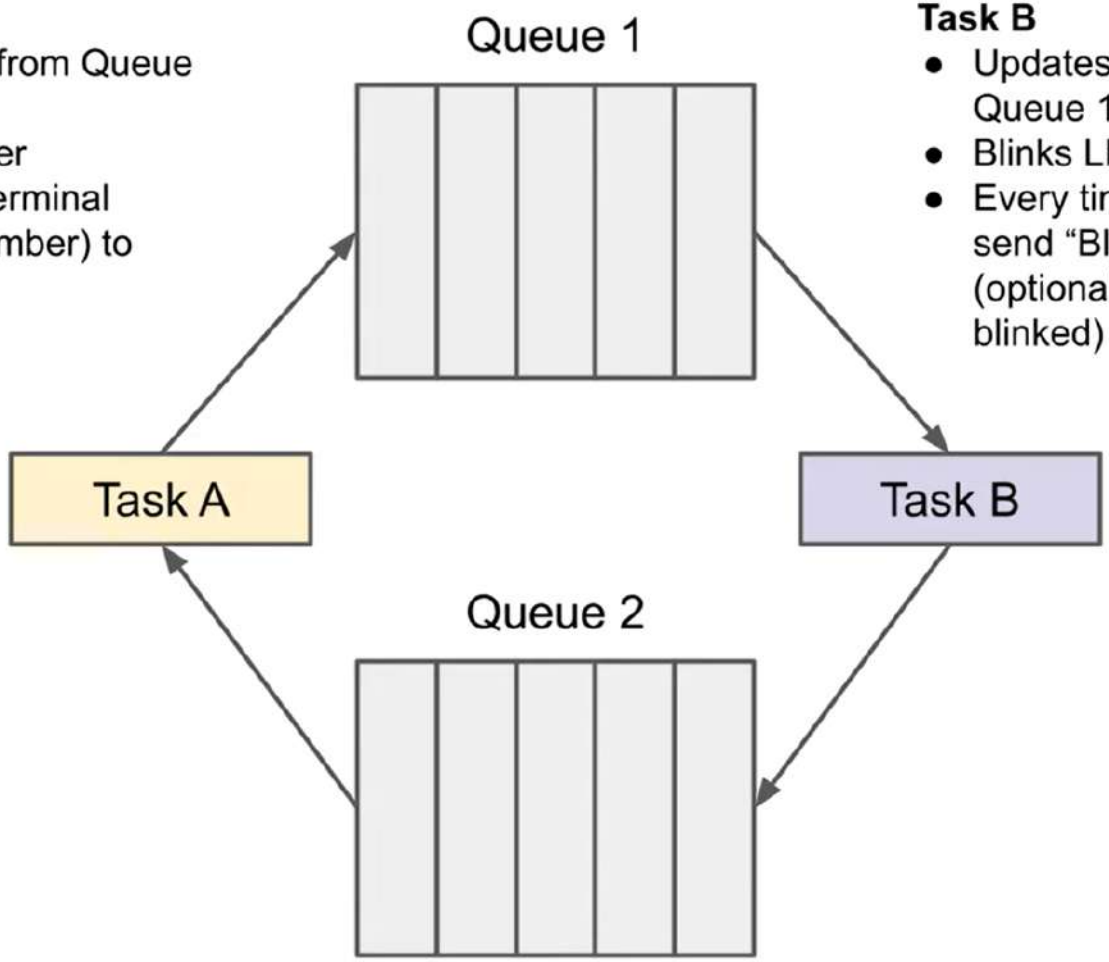


Copy by value, not by reference!



Task A

- Prints any new messages from Queue 2
- Reads serial input from user
- Echo input back to serial terminal
- If “delay xxx” send xxx (number) to Queue 1



Task B

- Updates `t` with any new values from Queue 1
- Blinks LED with `t` delay
- Every time LED blinks 100 times, send “Blinked” string to Queue 2 (optional: also send number of times blinked)

Race Condition

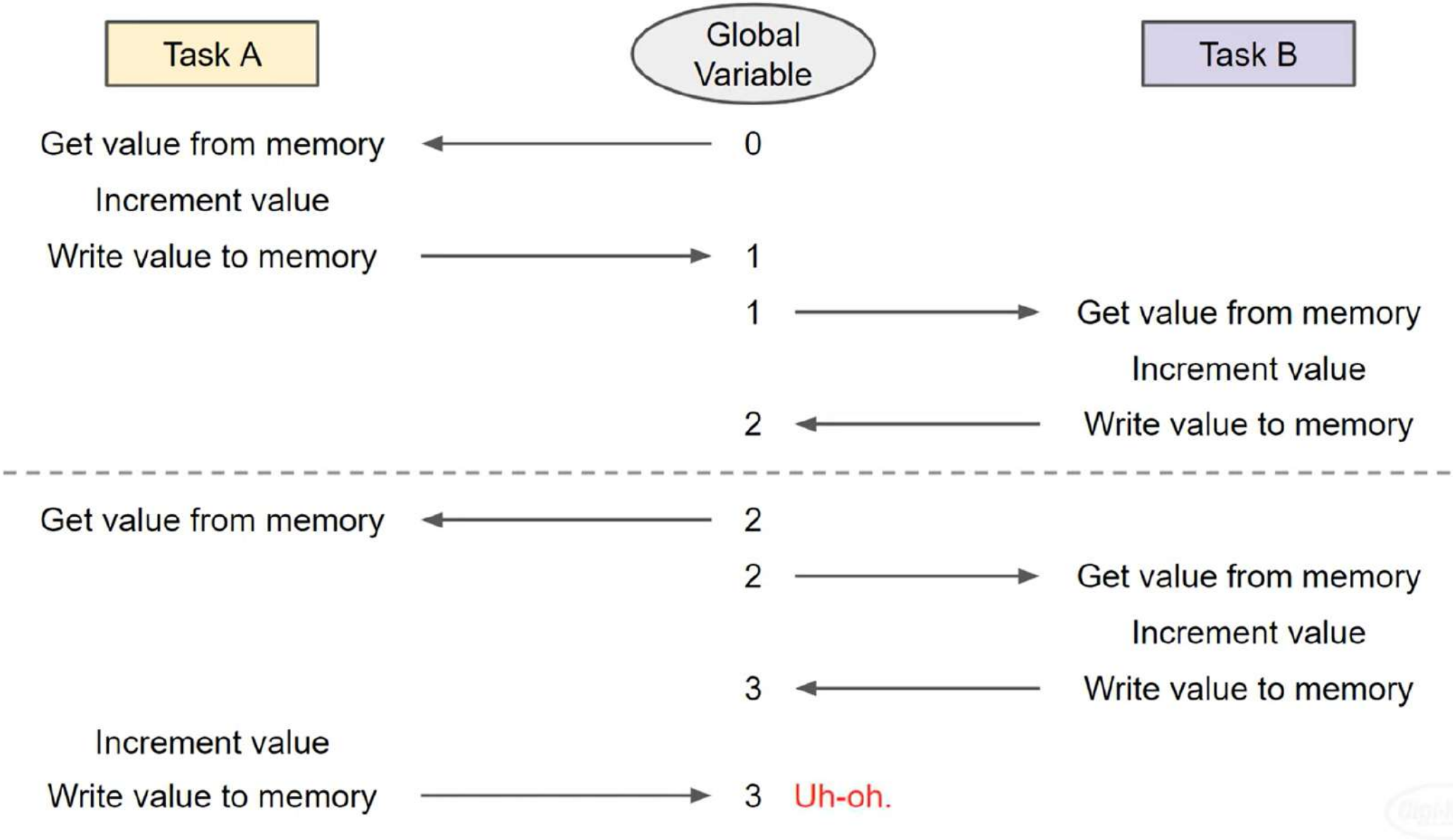
```
int global_var = 0;

void incTask(void *parameters) {
  while(1) {
    global_var++;
  }
}

void main() {
  startTask1(incTask, "Task 1");
  startTask2(incTask, "Task 2");
  sleep();
}
```

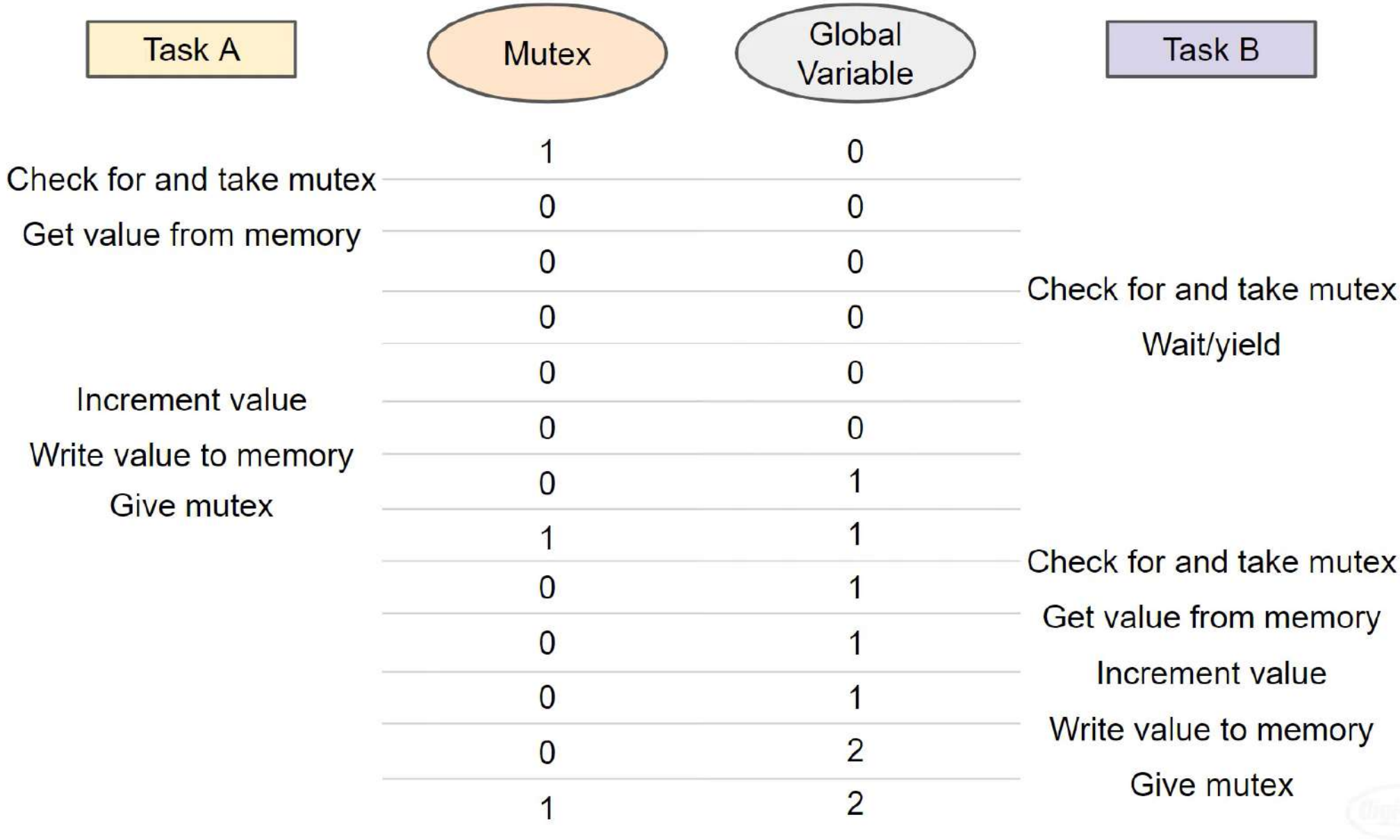
global_var increment can take several instruction cycles!

global_var:
0
1
2
3
3
4
5
5


Task A
Global Variable
Task B
Get value from memory
0
Increment value
Write value to memory
1
1
Get value from memory
Increment value
2
Write value to memory
Get value from memory
2
2
Get value from memory
Increment value
3
Write value to memory
Increment value
Write value to memory
3
Uh-oh.

Protecting Shared Resources and Synchronizing Threads

- Queue: pass messages (data) between threads
- Lock: allows only one thread to enter the “locked” section of code
- Mutex (MUTual EXclusion): Like a lock, but system wide (shared by multiple processes)
- Semaphore: allows multiple threads to enter a critical section of code

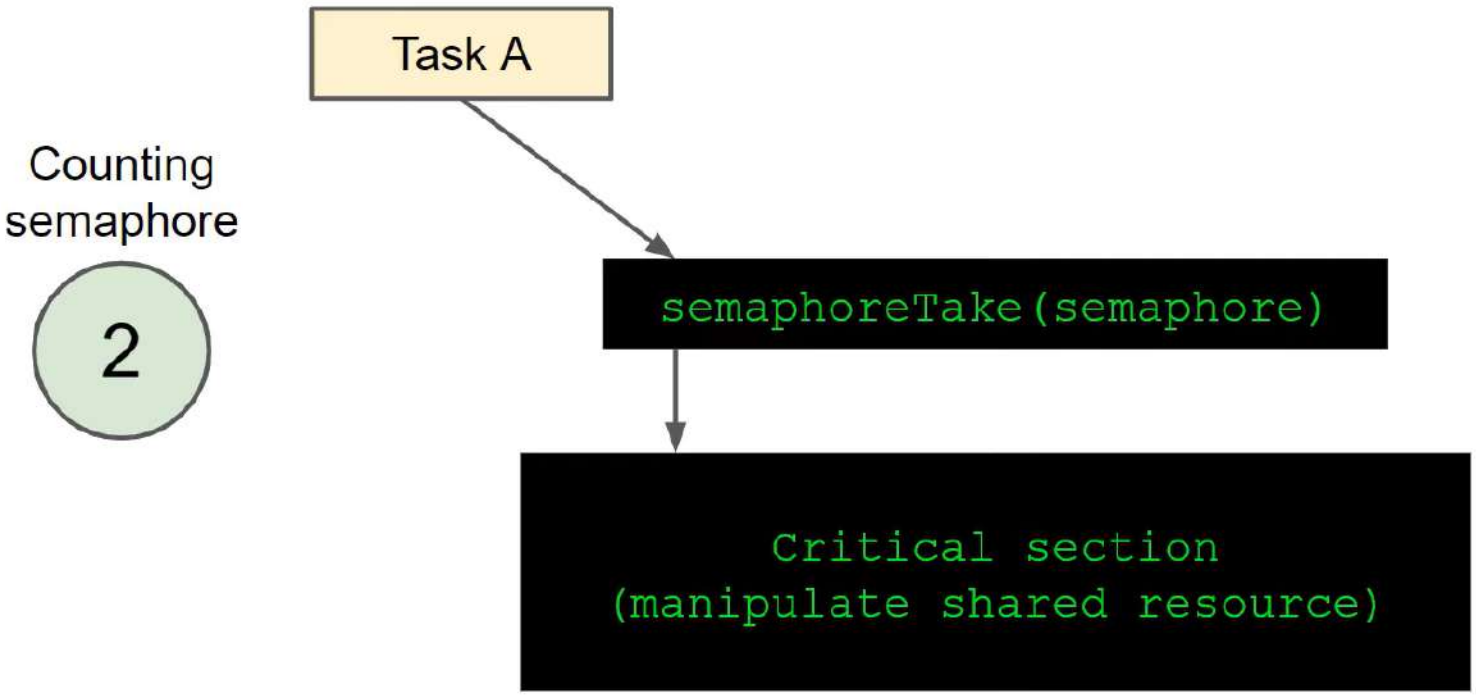

Task A
Mutex
Global Variable
Task B
1
0
Check for and take mutex
0
0
Get value from memory
0
0
Check for and take mutex
0
0
Wait/yield
0
0
Increment value
0
0
Write value to memory
0
1
Give mutex
1
1
Check for and take mutex
0
1
Get value from memory
0
1
Increment value
0
1
Write value to memory
0
2
Give mutex
1
2

Semaphore: The Idea

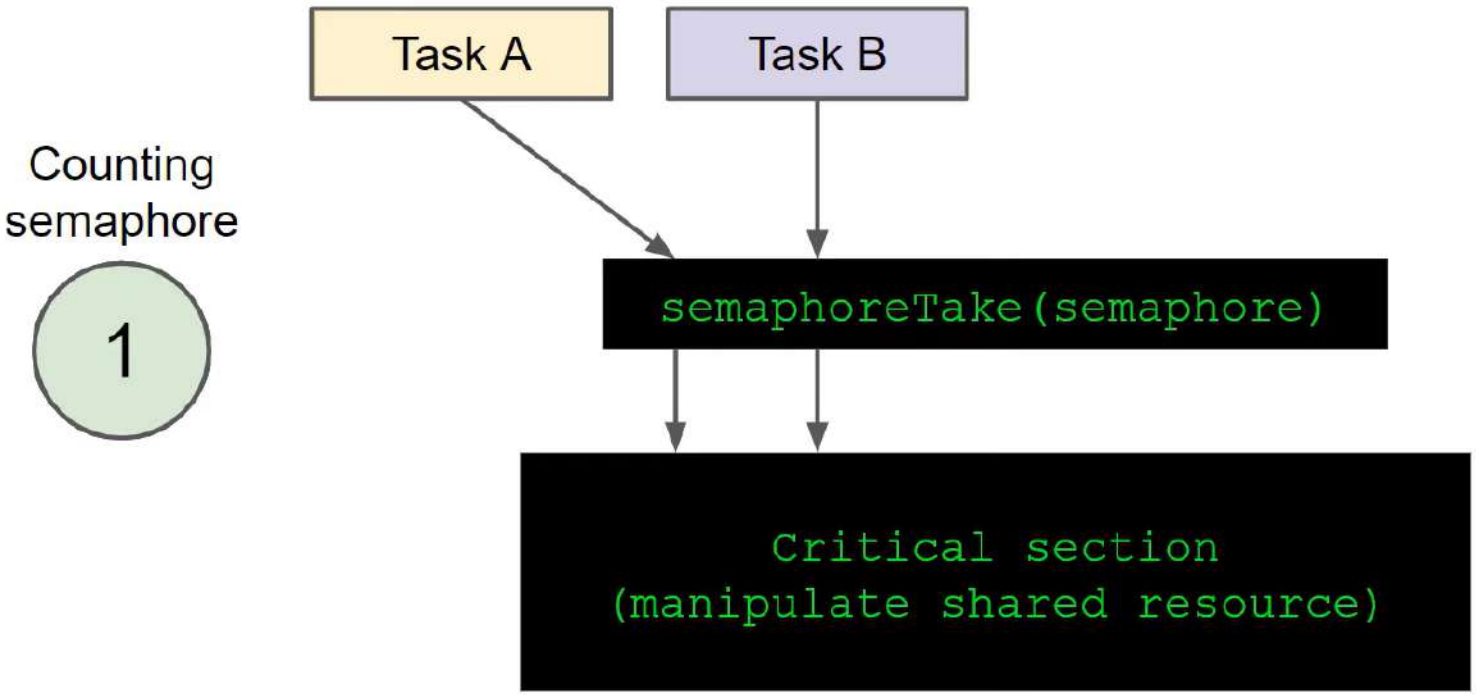
Counting semaphore


```
Critical section
(manipulate shared resource)
```

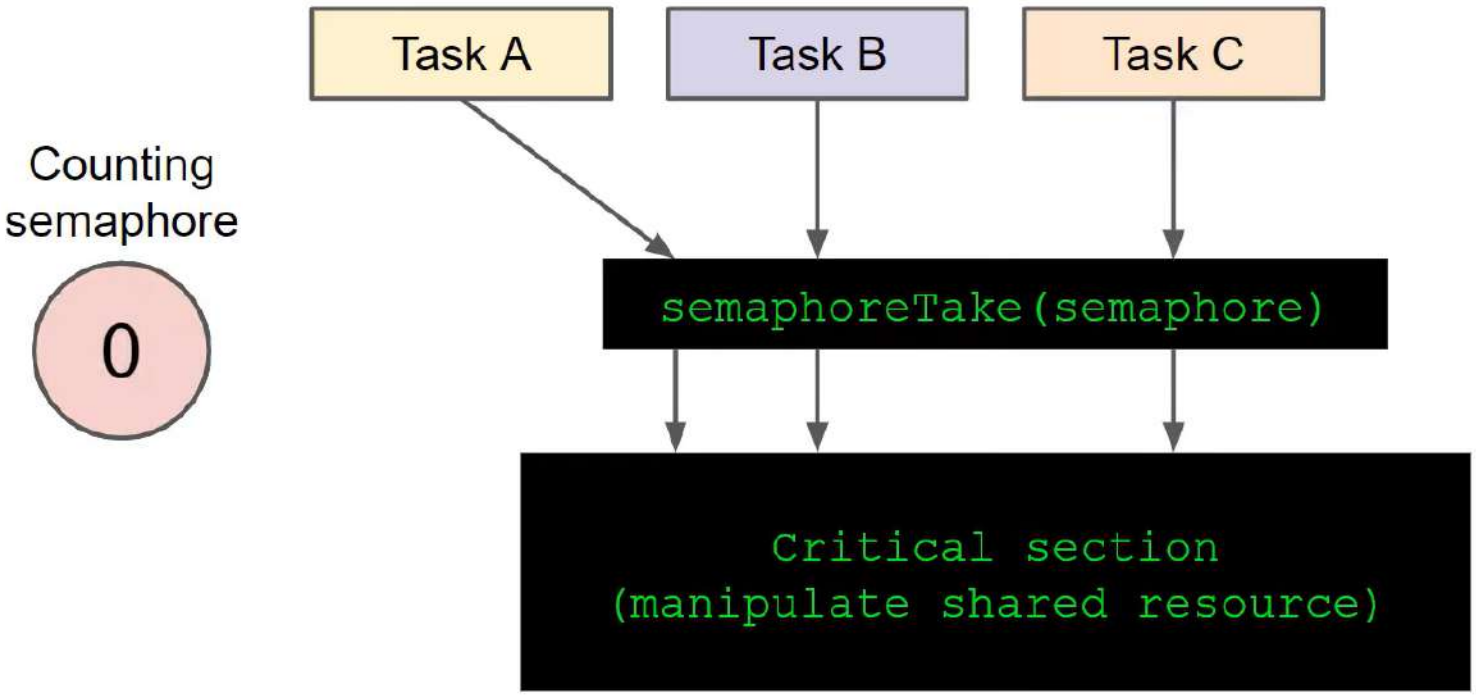
Semaphore: The Idea



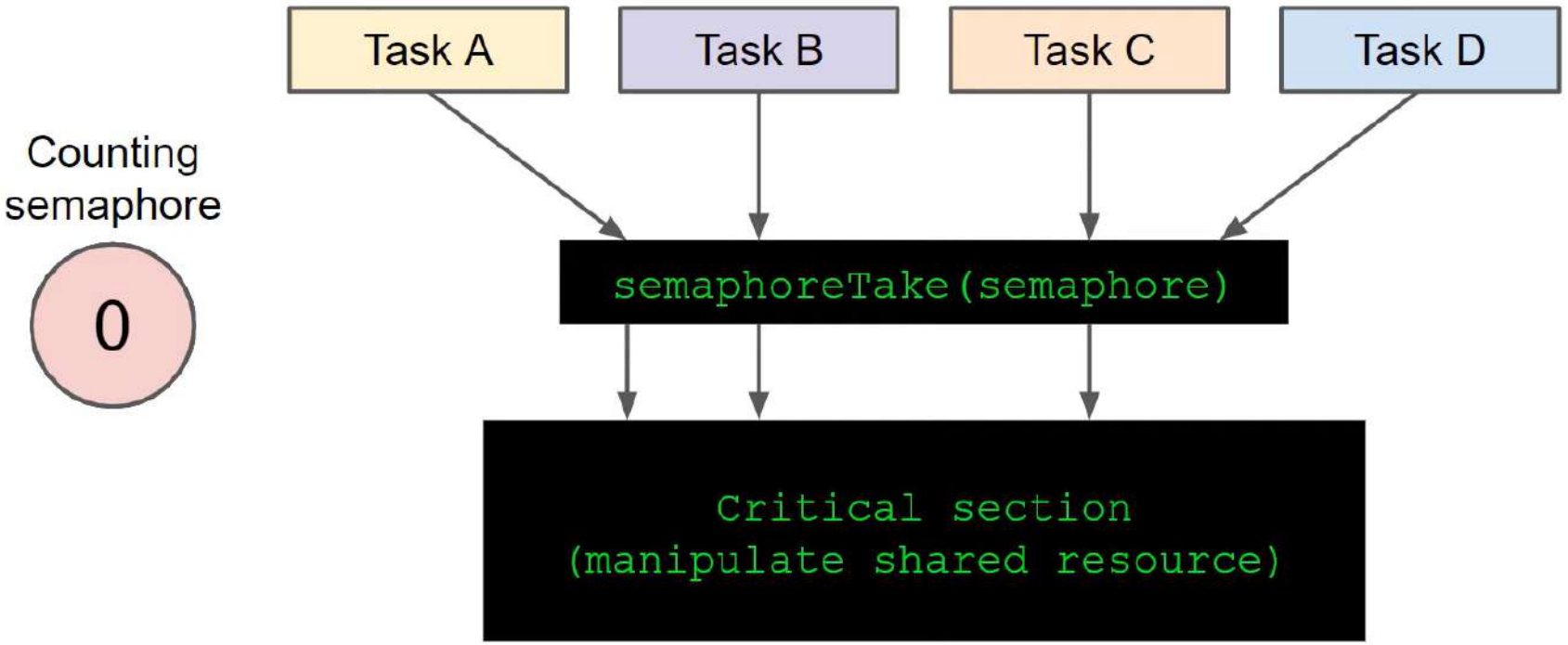
Semaphore: The Idea



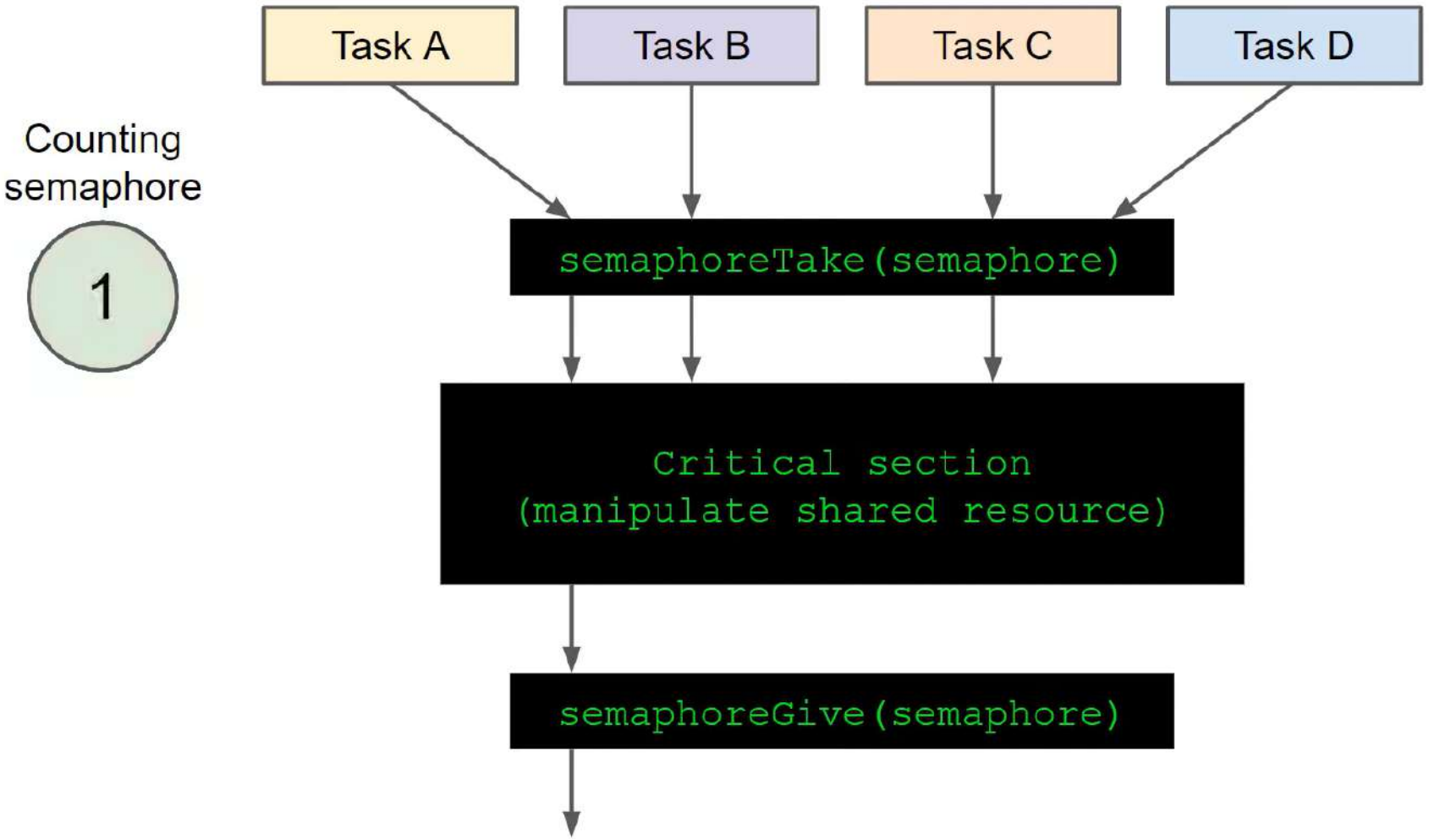
Semaphore: The Idea



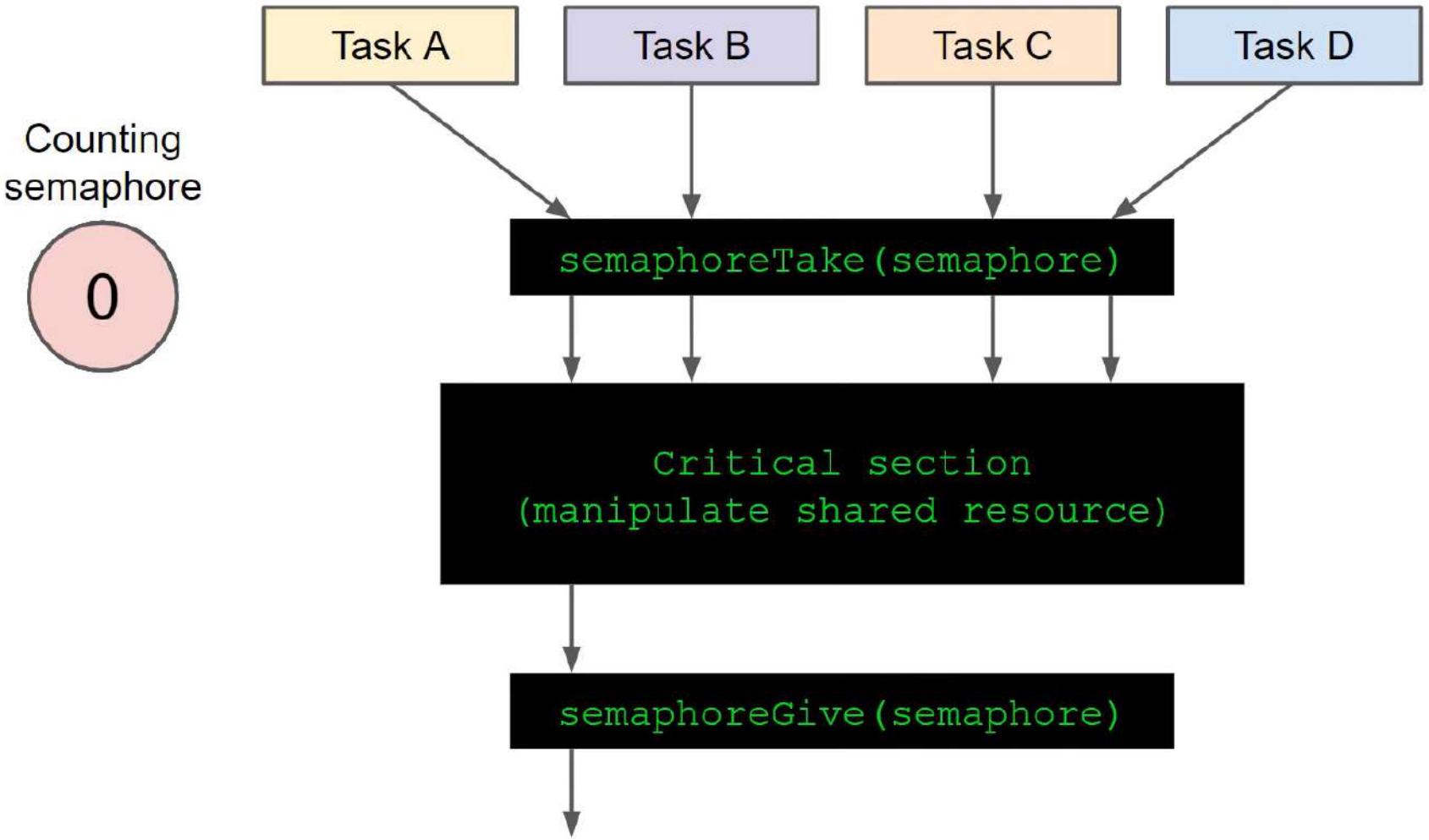
Semaphore: The Idea



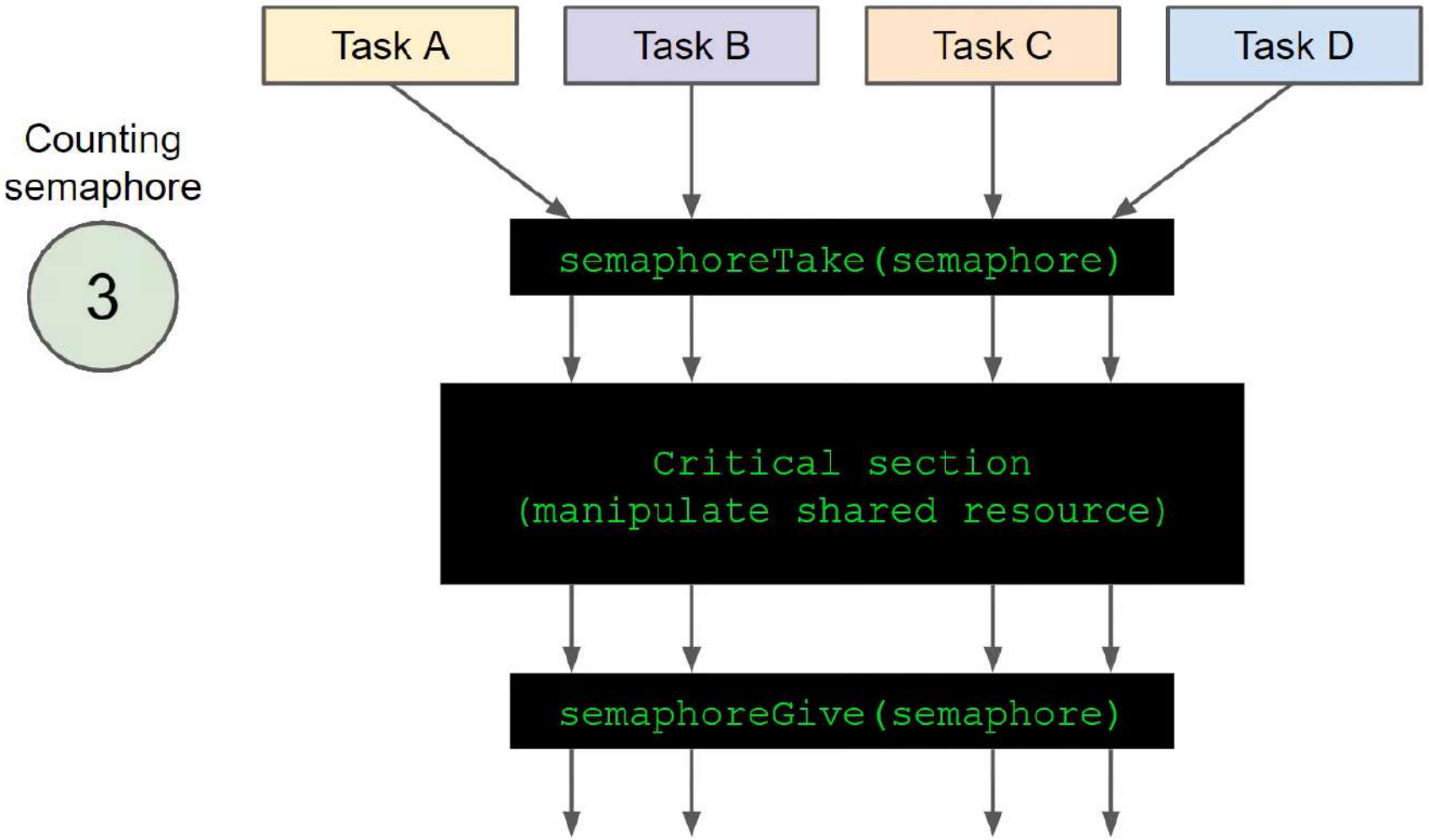
Semaphore: The Idea



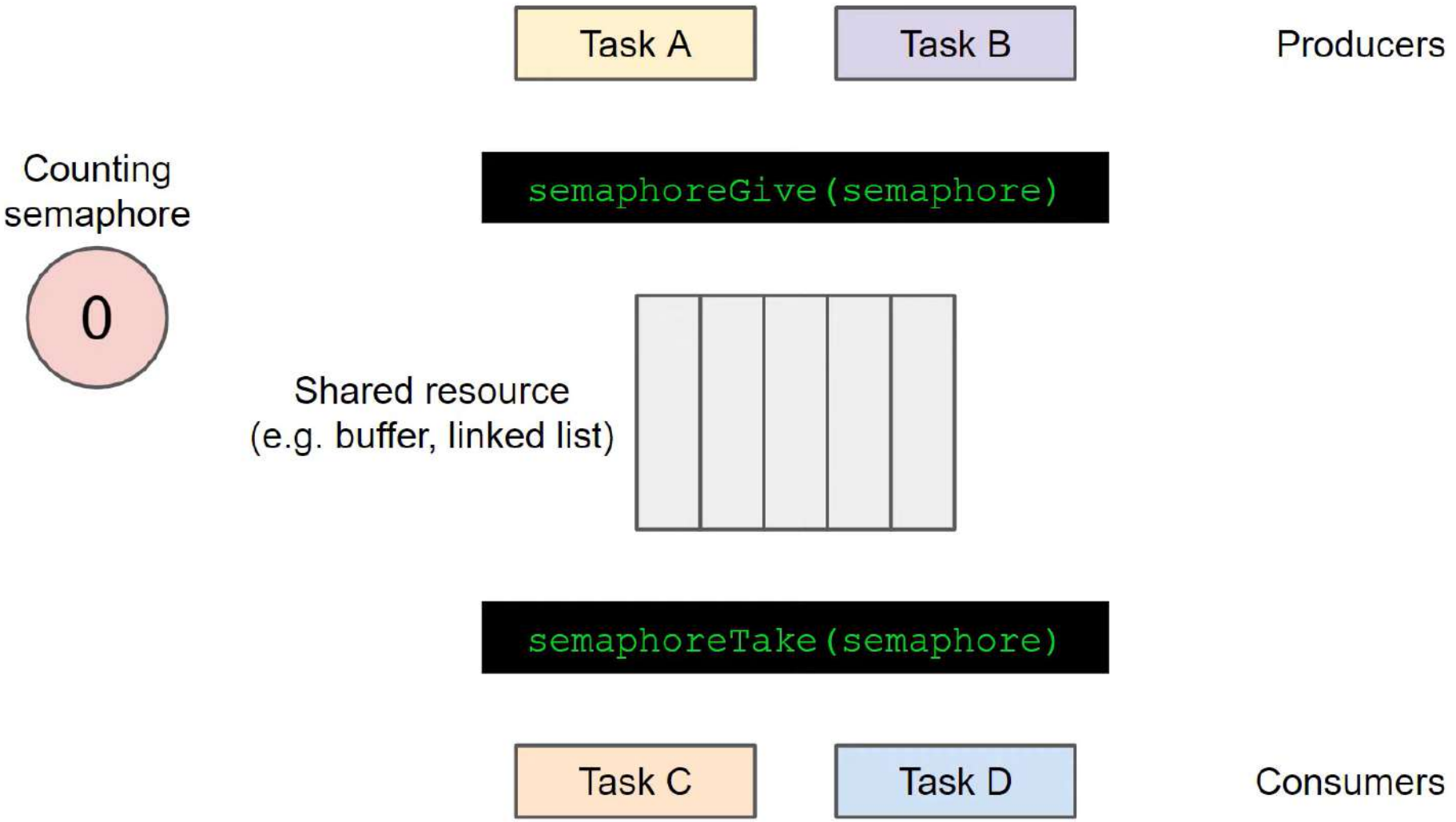
Semaphore: The Idea



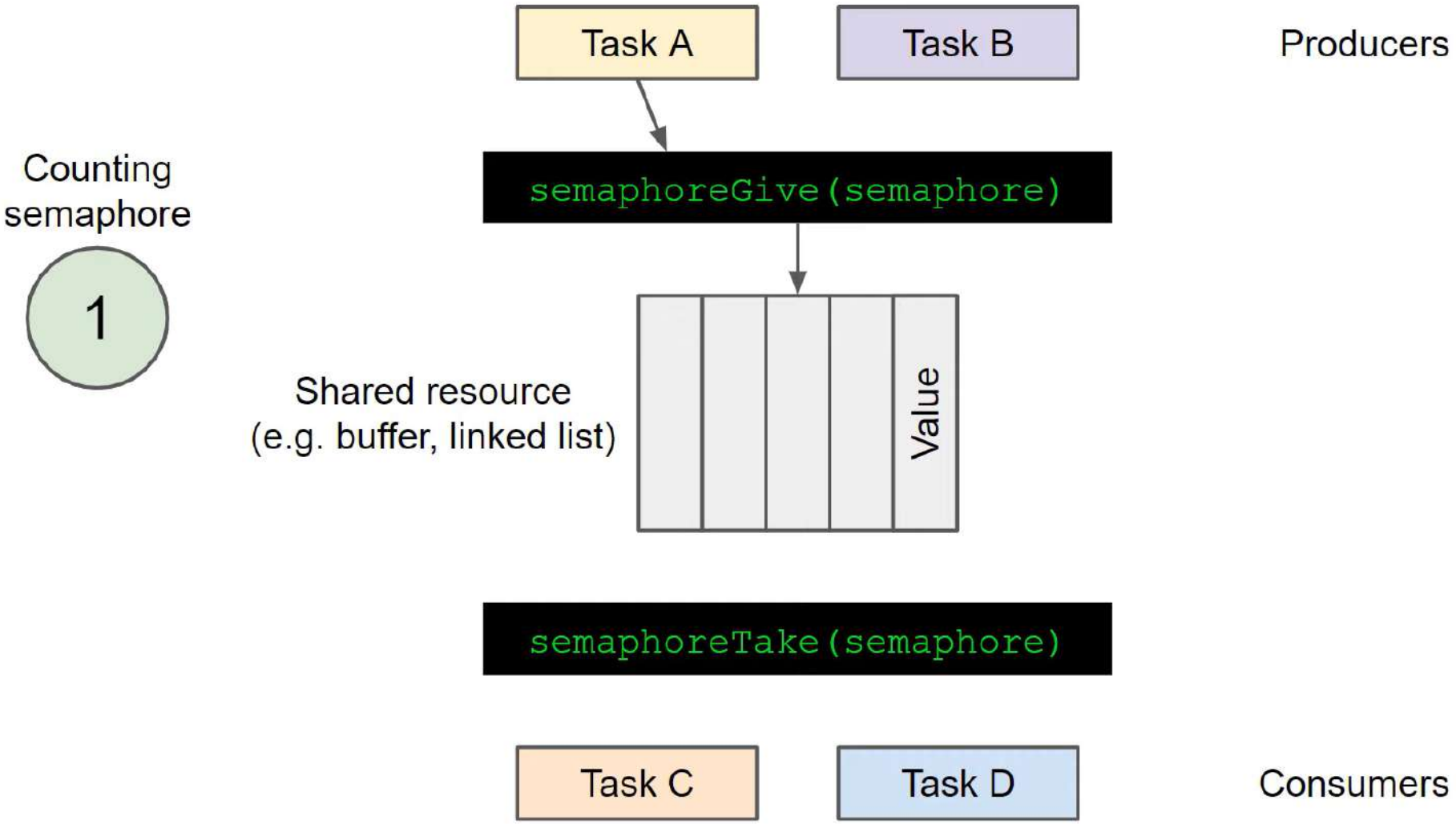
Semaphore: The Idea



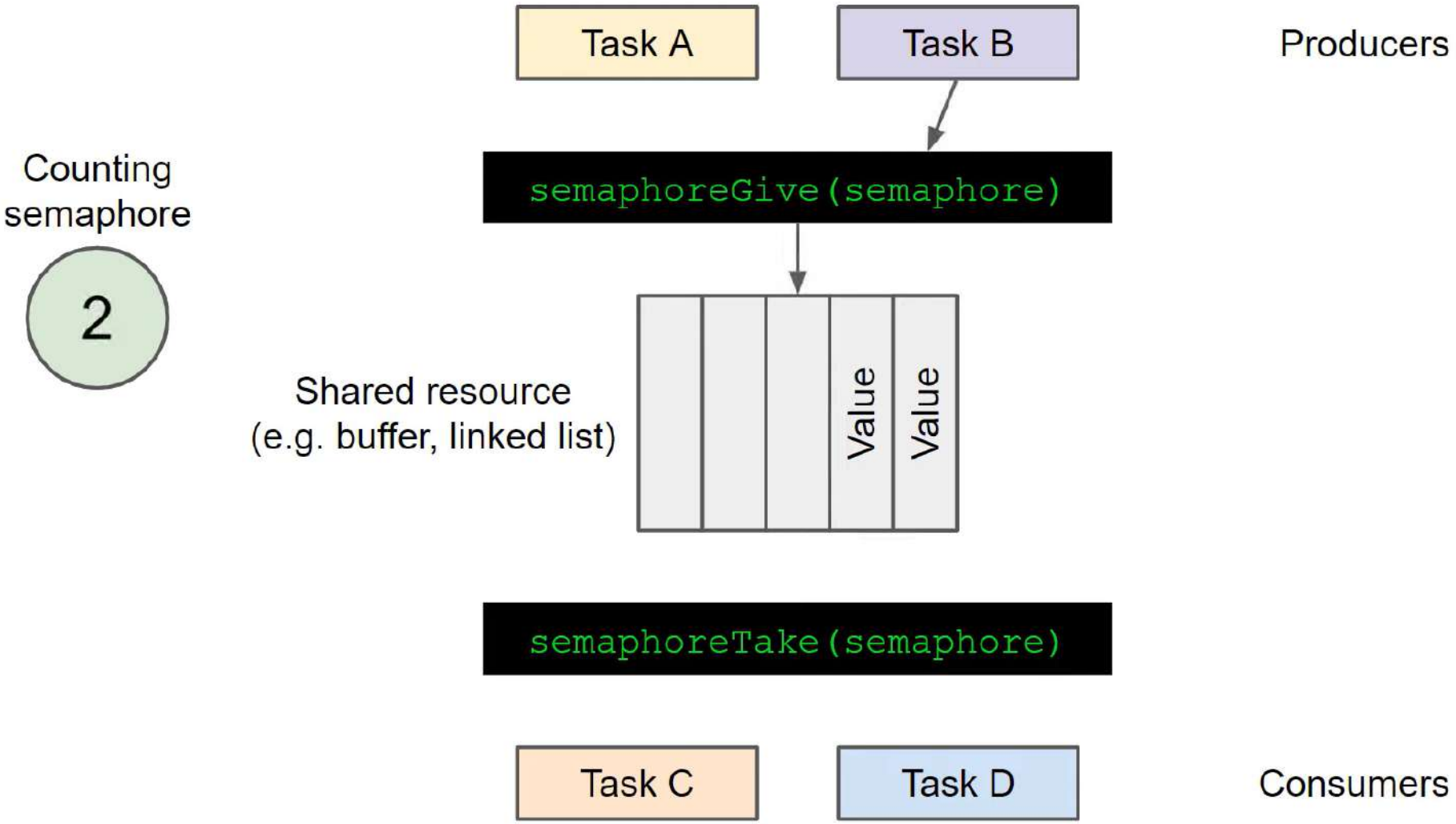
Semaphore: In Practice



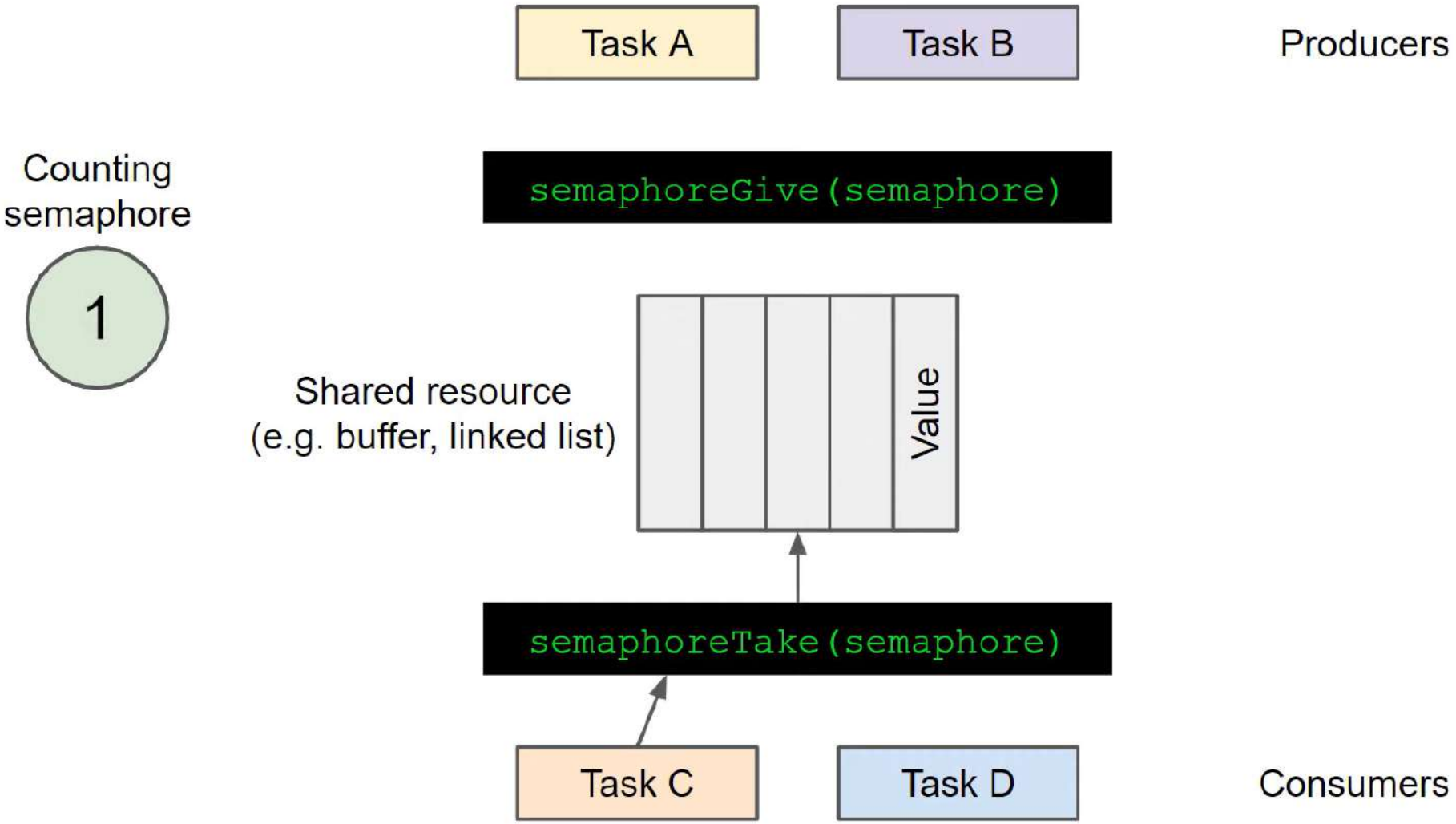
Semaphore: In Practice



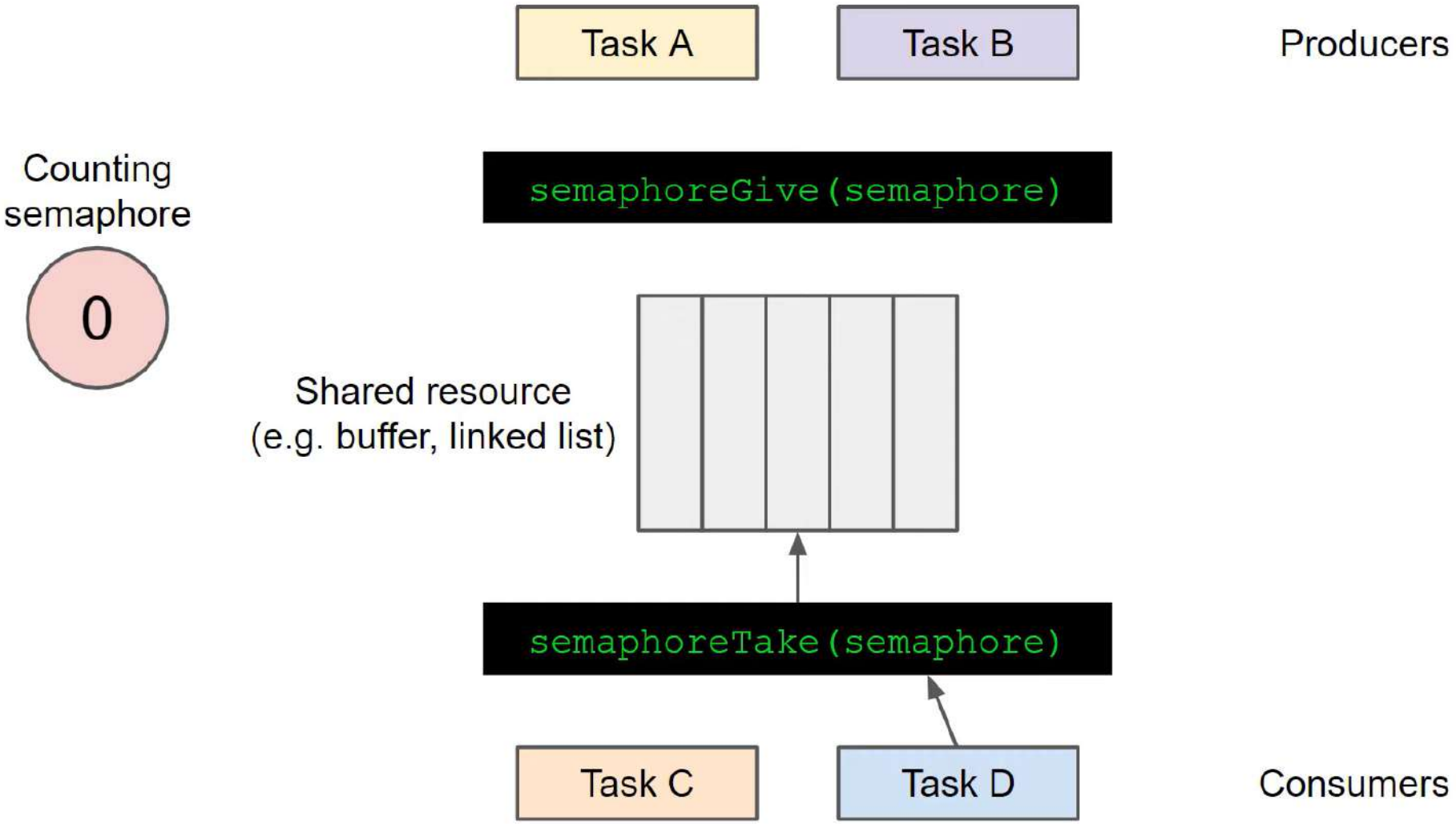
Semaphore: In Practice



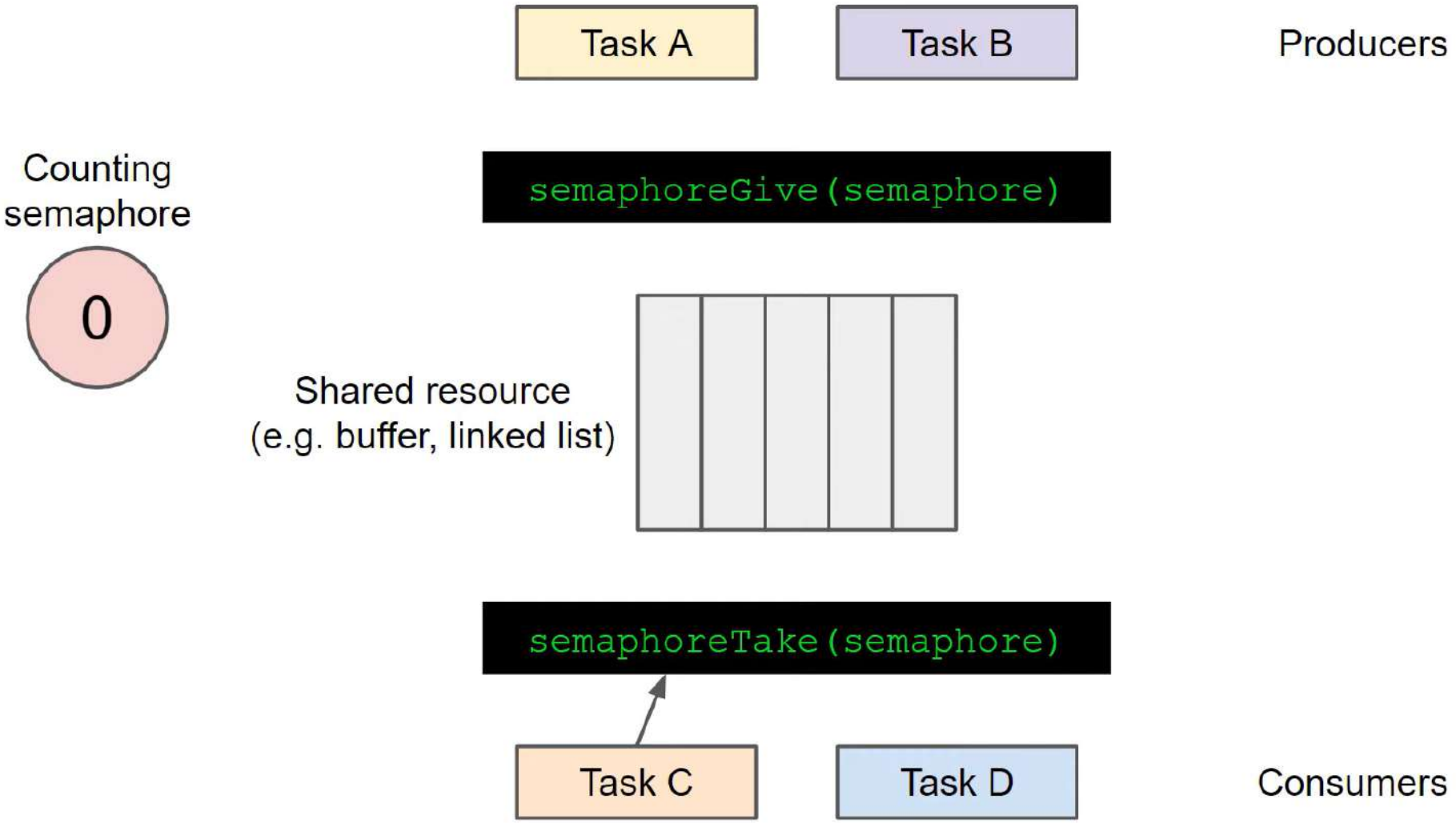
Semaphore: In Practice



Semaphore: In Practice



Semaphore: In Practice



Ownership!

Priority Inheritance!

Mutex

```
// Task 1
semaphoreTake(mutex)
// Use shared resource
semaphoreGive(mutex)


// Task 2
semaphoreTake(mutex)
// Use shared resource
semaphoreGive(mutex)
```

Semaphore

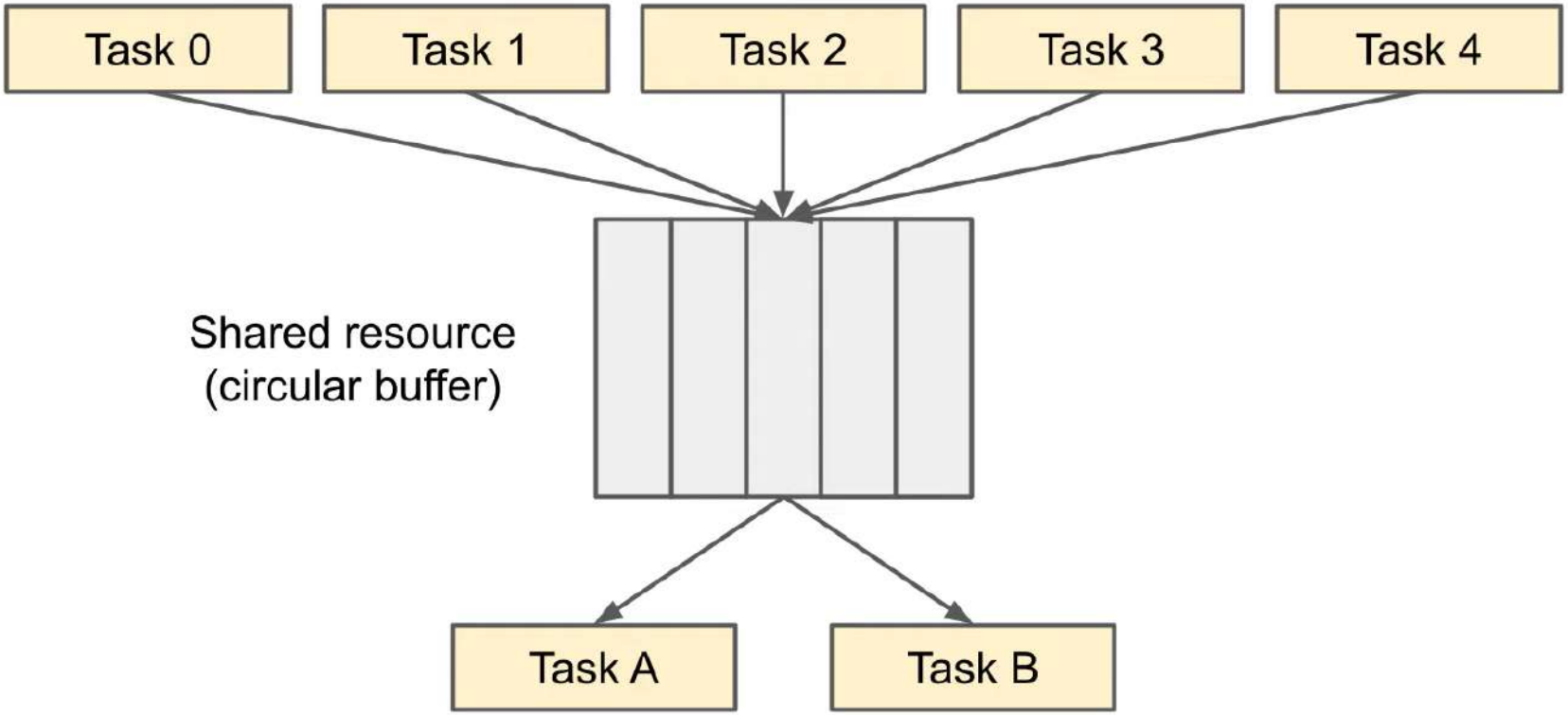
```
// Task 1 (producer)
// Add something to
// shared resource
semaphoreGive(semaphore)


// Task 2 (consumer)
semaphoreTake(mutex)
// Remove something from
// shared resource
```

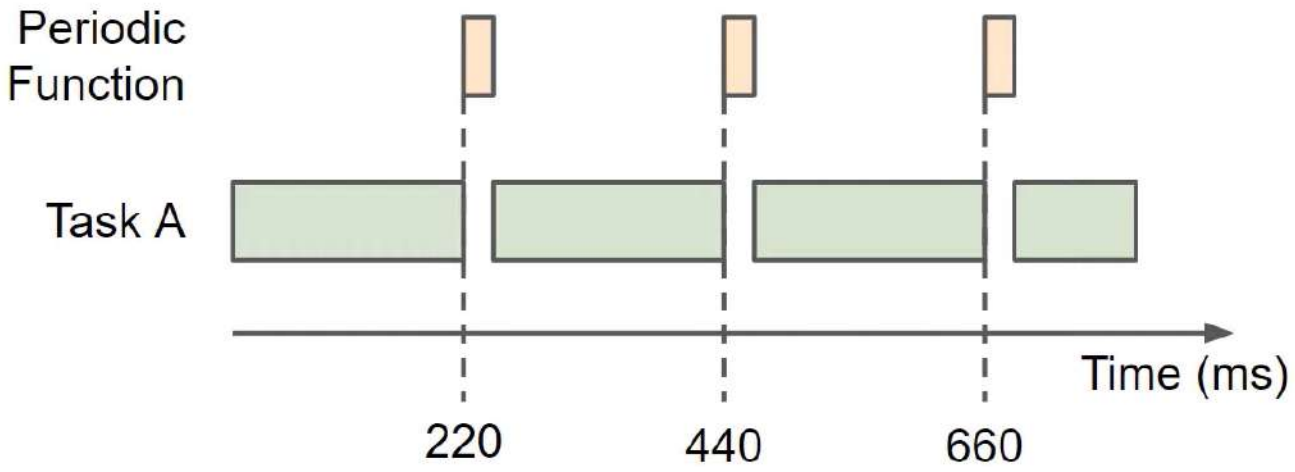
Good for ISR!

FreeRTOS	POSIX
xSemaphoreTake()	sem_wait()
xSemaphoreGive()	sem_post()
uxSemaphoreGetCount()	sem_getvalue()

Semaphore: The Challenge



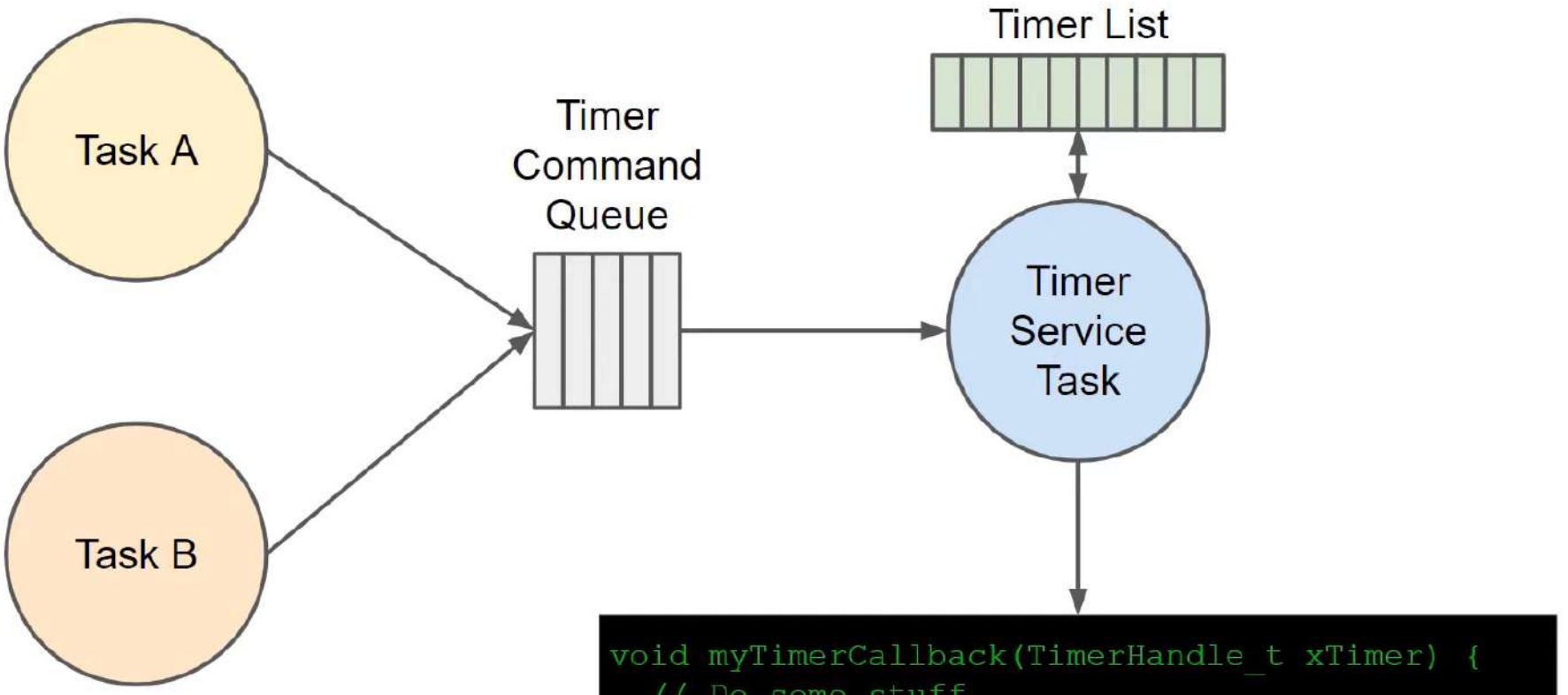
Timers

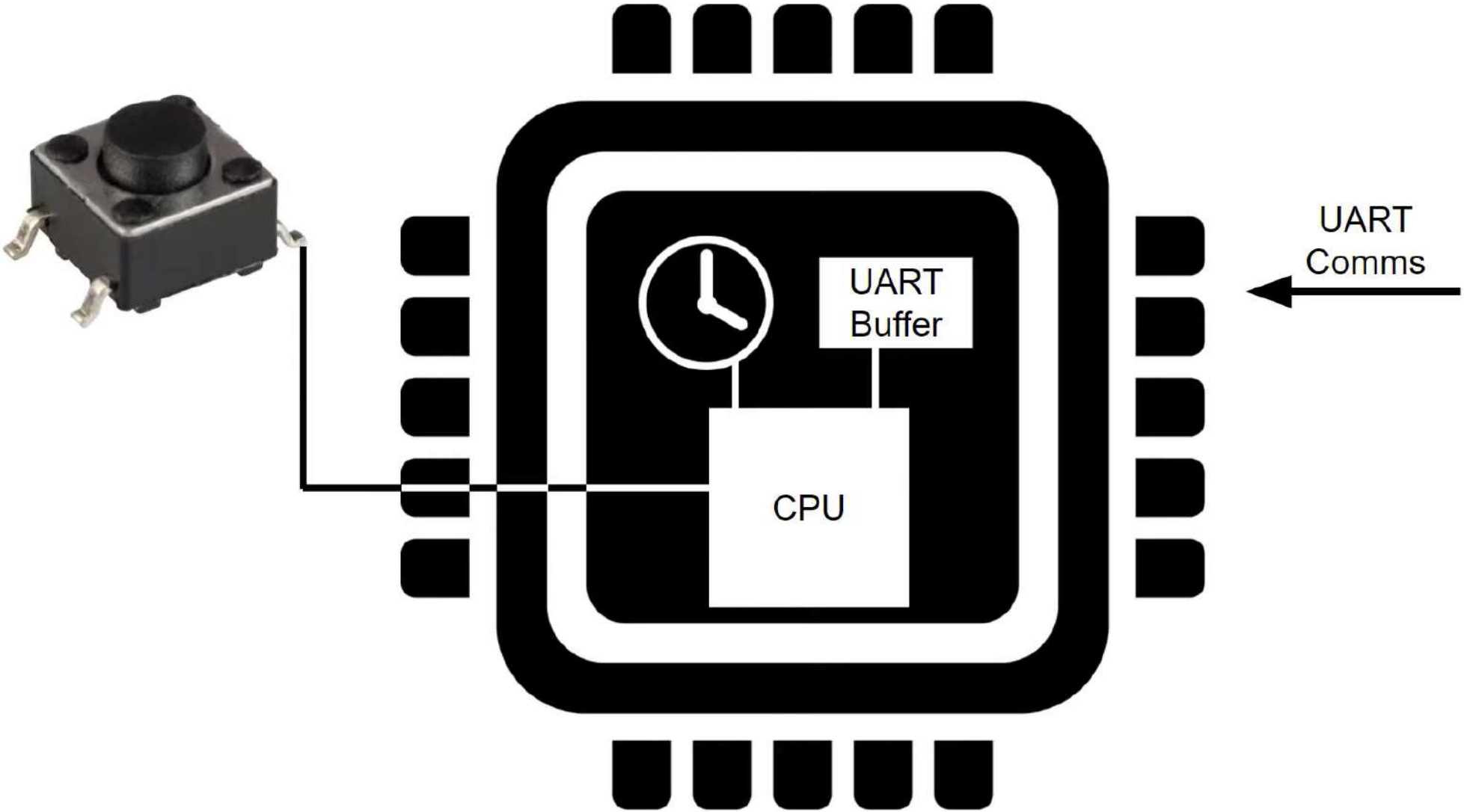


Possible Approaches

- New task with vTaskDelay()
- Task A with xTaskGetTickCount()
- Hardware timer
- Software timer

Software Timers in FreeRTOS




UART
Buffer
CPU
UART
Comms