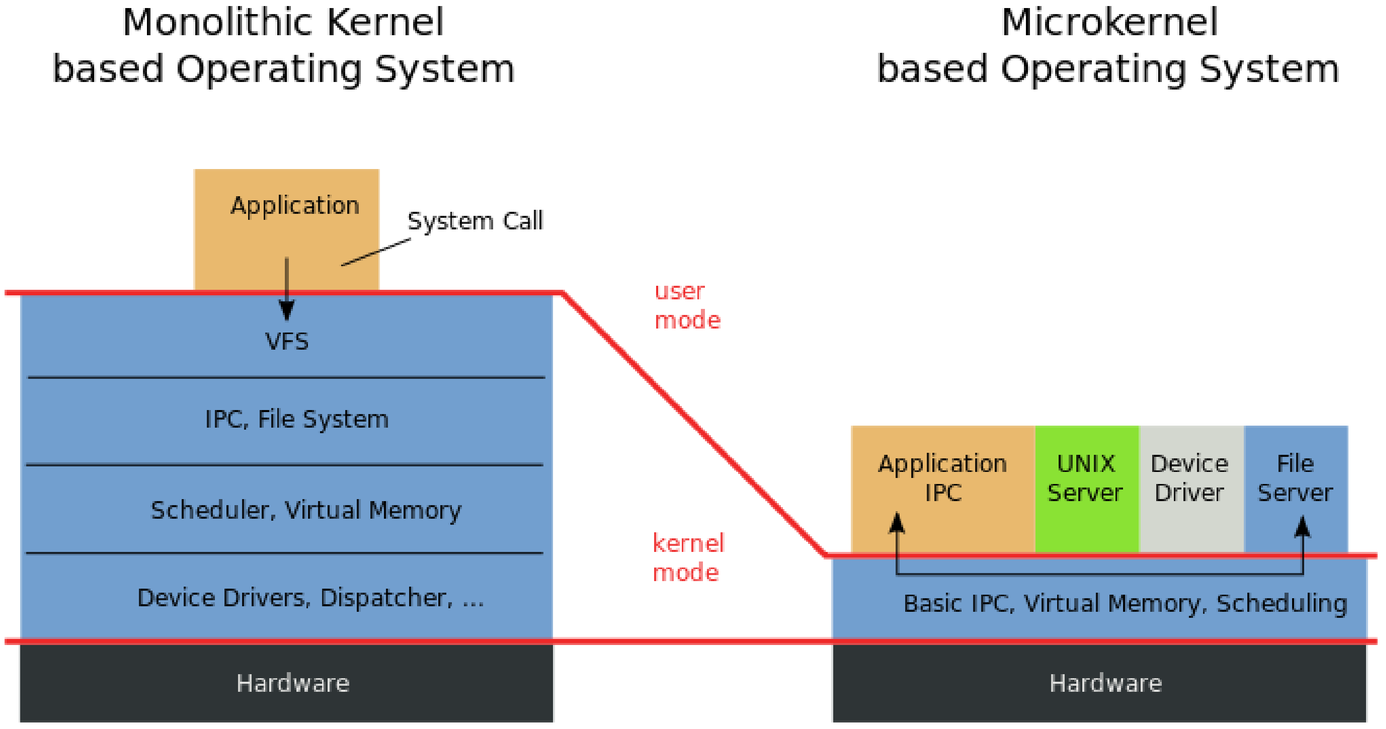
ESPRESSIF

Operating Systems and FreeRTOS

Operating Systems

- An OS (Operating System) manages computer hardware and software.
- OS's with Monolithic Kernels typically offer services such as Scheduling, Drivers, File System etc.
- The Application/Users utilizes OS services via system calls.
- Micro-kernels only offer the bare minimums of an OS (Tasks, scheduling, IPC, virtual memory). Other services are usually implemented on top of the Micro-kernel



Real Time Operating Systems

- A Real Time System is a system where the correctness of the system is dependent on the correctness and timeliness of the result

Hard Real Time	Firm Real Time	Soft Real Time
Missing a deadline results in a total system failure	Infrequent missed deadlines are permissible but a late result is useless	A missed deadline results in degraded performance

- An RTOS (Real Time Operating System) is an OS intended to serve real time applications. Real time systems must guarantee a response within specified time constraints (deadlines).

- An RTOS differs from a non real time OS generally aims to maximize throughput by maintaining scheduling fairness

- ESP-IDF currently uses a modified version of FreeRTOS

FreeRTOS

- An RTOS popular with embedded systems. Designed to be small, simple, and free.

- FreeRTOS is a micro-kernel offering the bare minimum services of an RTOS
  - Tasks (Threads)
  - Scheduling
  - Queues, Semaphores, and Mutexes
  - Software Timers
  - Event Groups

- Other services such as drivers and file systems are provided in the form of IDF components. These components often built using FreeRTOS API

- FreeRTOS is designed for single core, but has been modified to be SMP compatible for the ESP32

FreeRTOS Task Structure

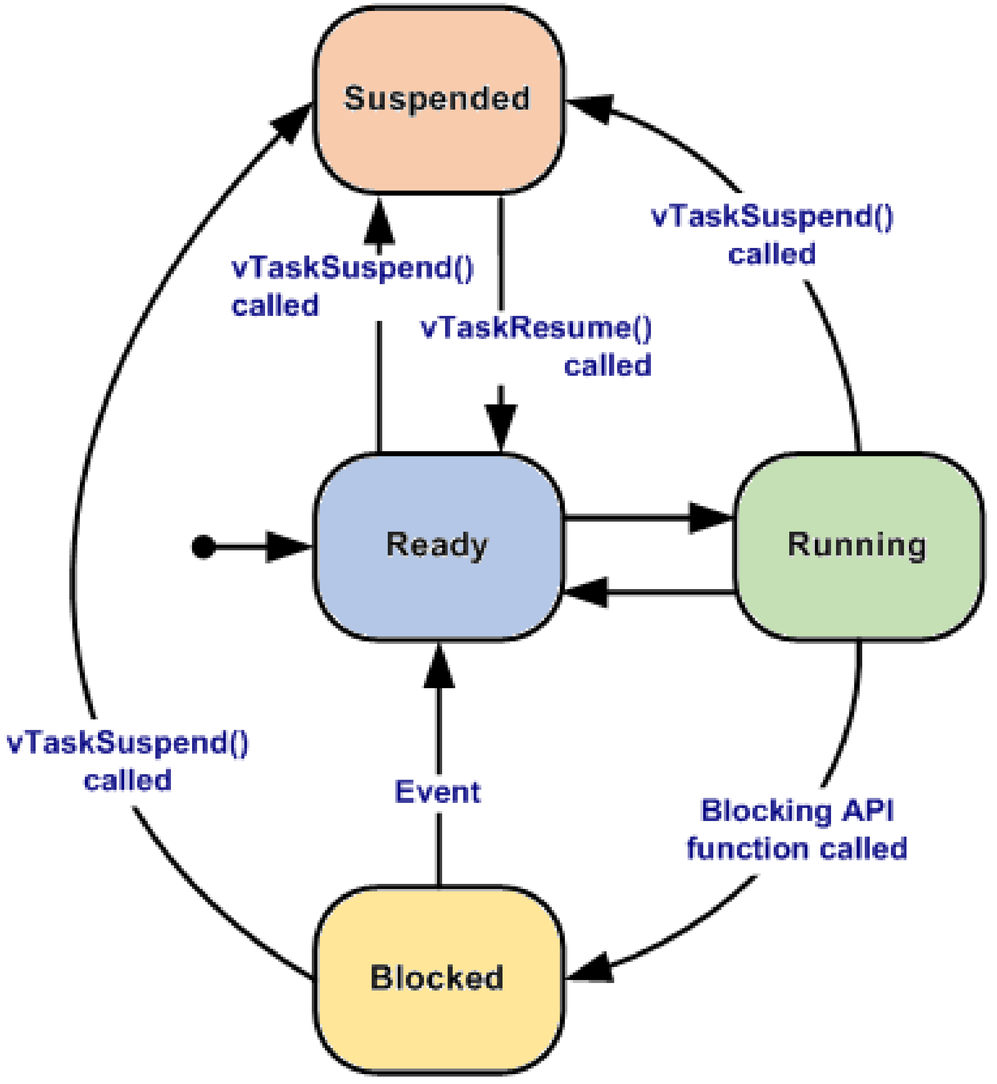
FreeRTOS tasks are implemented as C functions and are structured as follows

```
void example_task(void *pvParameters)
{
    //Initialize local variables here

    while (1) {
        //Task main loop
    }

    //Delete task here
    vTaskDelete(NULL);
}
```

- A task function must be a function that accepts a void pointer parameter and returns void.
- User can use the parameter as a pointer to a structure containing parameter values, or cast the pointer directly as a value.
- Main body of task is usually implemented as an infinite loop.
- To prevent the loop constantly consuming CPU time, a task will usually **block** (calling a delay or waiting for an event).
- Breaking out of the loop usually means the task has run to completion.
- A task should **never return or execute past the end of the function**, doing so will likely lead to an unrecoverable error.
- A task should exit by deleting itself.

FreeRTOS Task States

Task State	Description
Running	The task is currently being executed
Ready	The task is able to execute but is not currently executing
Blocked	The task is waiting for a temporal or external event. The task will not be scheduled until it is unblocked by the event or times-out
Suspended	Task will not be scheduled until explicitly unsuspended



What Is Scheduling?

- A single CPU core can only execute one instruction at a time
- A CPU switches between the execution of multiple tasks rapidly to create the illusion of multitasking.
- Multicore CPUs can achieve true multitasking, as they can physically run multiple instructions simultaneously
- The methods and algorithms used to select which task(s) to execute is known as scheduling.
- Scheduling algorithms vary greatly depending on the goals of the scheduler
  - Maximizing throughput
  - Maximizing fairness
  - Minimizing latency
  - Minimizing wait time

FreeRTOS Scheduling

FreeRTOS uses a Fixed Priority Preemptive scheduler with Time Slicing

Task blocked

Task unblocked

Preempted by tick

Preempted by higher priority task

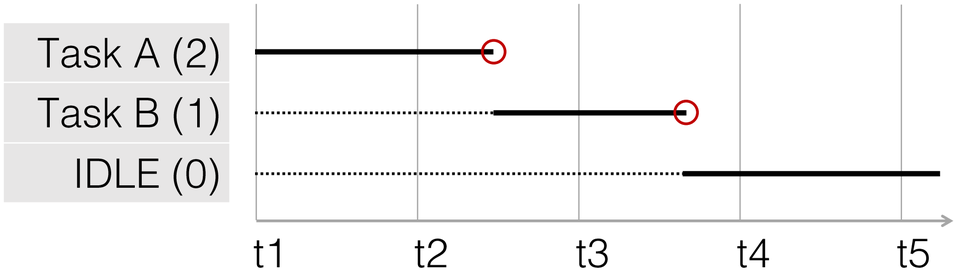
Task Running

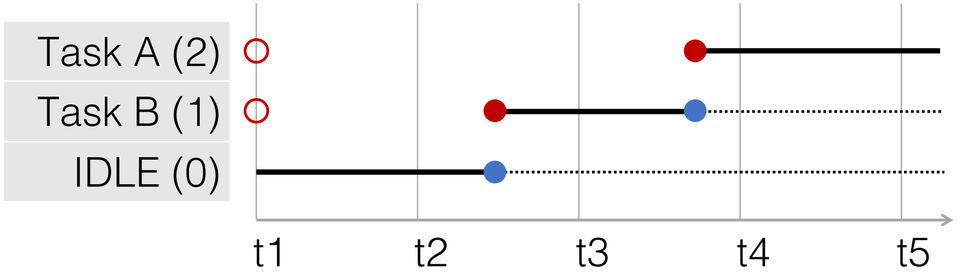
Task Ready

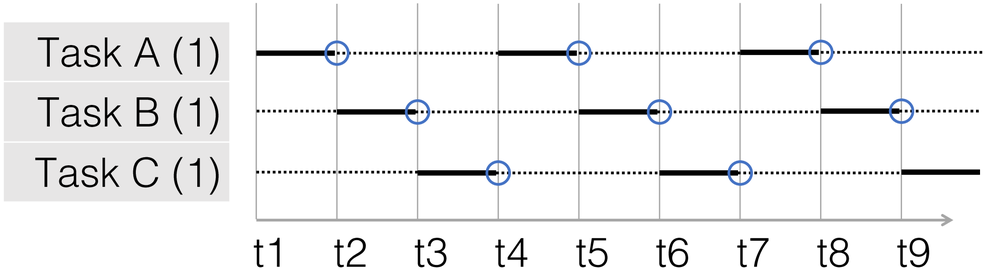
Fixed Priority: Each tasks is given a constant priority upon creation. The scheduler executes highest priority ready state task

Preemptive: The scheduler can switch execution to another task without the cooperation of the currently running task

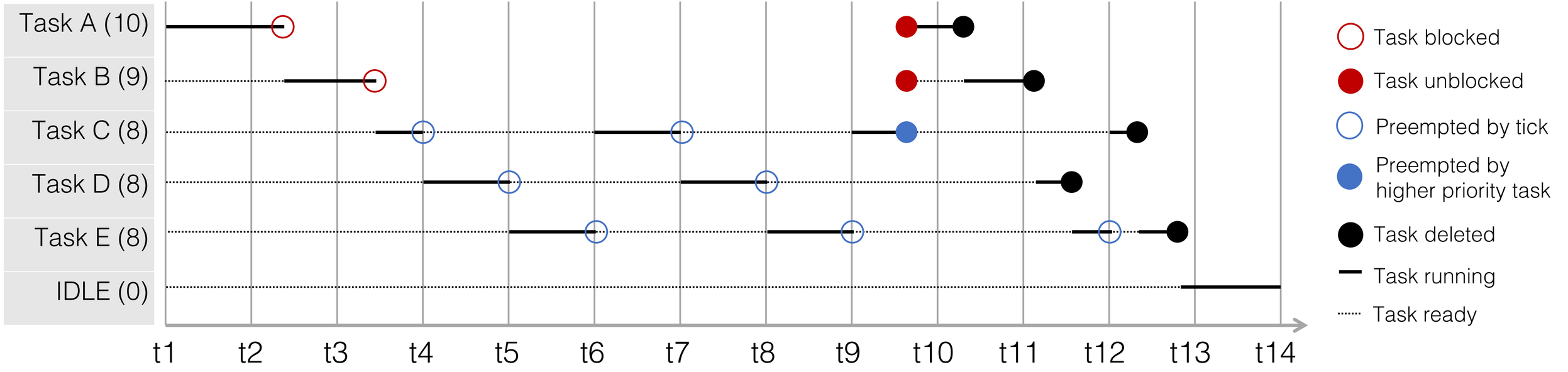
Time Slicing: The scheduler will periodically switch execution between ready state tasks of the same priority (Round Robin scheduling between tasks of the same priority). Time slicing is governed by a tick interrupt.







Scheduling Example



t1 – t2	Scheduler selects highest priority ready task (Task A)
t2 – t3	Task A blocks, scheduler selects next highest priority (Task B)
t3 - t9	Task B blocks. Task C, D, E have the same priority. Scheduler round robins between Tasks C, D, E switching on each tick
t9 - t10	Task A and B unblock. Task A preempts Task C
t10 – t12	Task A and B delete. Notice Task D is scheduled next due to round robin.
t12 – t15	IDLE is scheduled when no other task can run

Symmetric Multiprocessing

- SMP (Symmetric Multiprocessing) is a computing architecture where two or more identical processors (cores) are connected to a single shared main memory and controlled by a single operating system.

- In SMP, both cores view the memory identically meaning running the same piece of code on either core will have the same effect.

- This allows the running of tasks to be switched between cores without adverse affects.

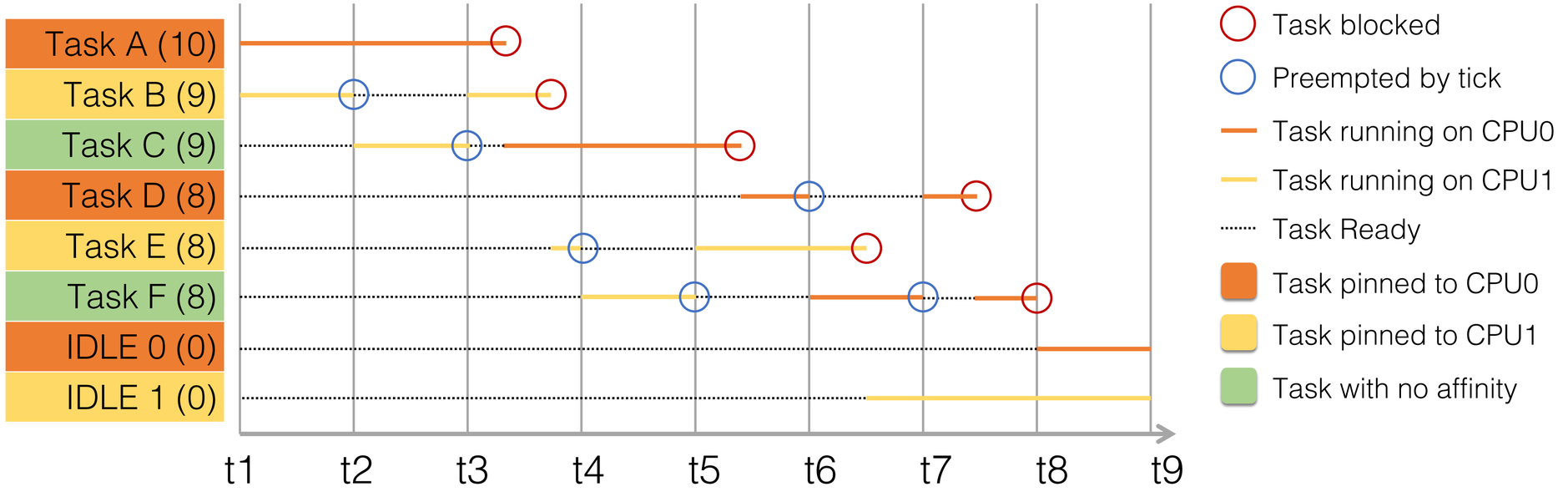
- Prevents application code from the needing to consider which core it will run on (since the cores are identical)

- Note that there are some special exceptions to the symmetry of SMP (interrupt allocation, and CPU specific memory regions such as instruction cache)

SMP Scheduling

- The ESP32 is an SMP chip containing two CPU cores (CPU0 and CPU1)
  - CPU0 is also known as PRO_CPU. Originally meant to handle protocol related processing
  - CPU1 is also known as APP_CPU. Originally meant to handle application related processing
- The FreeRTOS tasks and scheduler have been modified in ESP-IDF to be SMP compatible.
- Each task must have a core affinity. This specifies which core a task is allowed to run on
  - 0 means task can only run on CPU0
  - 1 means task can only run on CPU1
  - tskNO_AFFINITY means task can run on both cores
- Each scheduler executes the highest priority ready state tasks each core can run at the current time
  - A core can only execute a task with compatible affinity (regardless of its priority)
  - If the highest priority task is already being run by the other core, the current core cannot run it

SMP Scheduling Example



t1	Each CPU picks the highest ready state task that it can run (Task A for CPU0, Task B for CPU1)
t2 – t3	Task C has no affinity, therefore the scheduler time slices between Task B and C on CPU1
t3 – t4	Task A blocks. Next highest ready state task that can run on CPU0 is Task C. Task B blocks. Task C is already running. The next highest ready state task that can run on CPU1 is Task E
t5 – t6	Task C blocks. Both Task D and Task F can run on CPU0. Task D is selected (depends on task list ordering)
t6 – t7	Task D blocks, Task F runs. Task E blocks. Task F is already running therefore IDLE1 runs on CPU1
t7 – t8	Both Tasks D and F block. Only IDLE0 can run on CPU0

FreeRTOS Task Creation

FreeRTOS tasks can be created by calling xTaskCreatePinnedToCore()

```
void app_main()
{
    int task_param = 5;
    TaskHandle_t task_handle;
    xTaskCreatePinnedToCore(example_task, "Example", 2048, (void *)task_param, 10, &task_handle, 0);
}                                          1          2      3            4          5        6      7
```

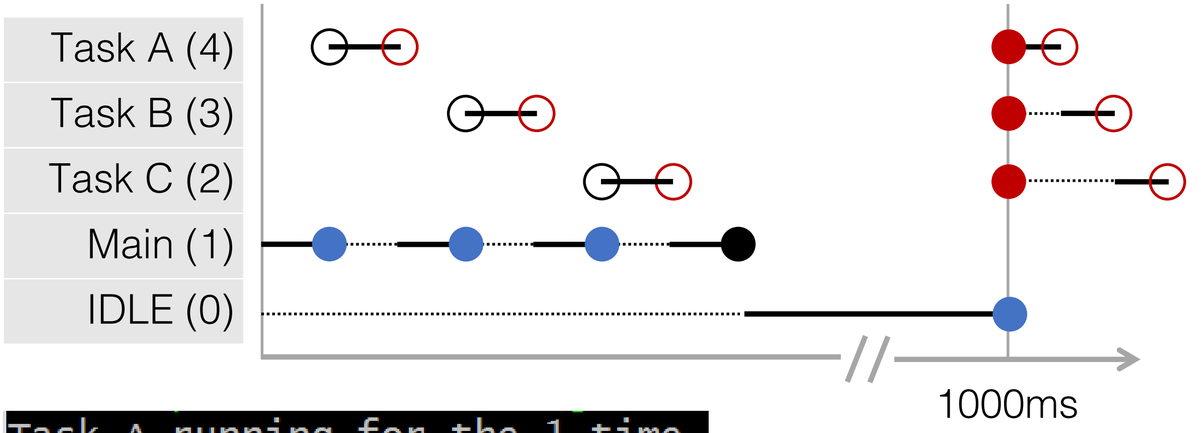
pvTaskCode	1	Pointer to the C function that implements the task
pcName	2	Name of the task (for debugging purposes)
usStackDepth	3	Size of the task's stack in bytes
pvParameters	4	Parameter passed to the task
uxPriority	5	Priority of the task
pxCreatedTask	6	Pointer to variable to store the handle of task
xCoreID	7	CPU affinity of task (0, 1 or tskNO_AFFINITY)

Notes on Task Usage

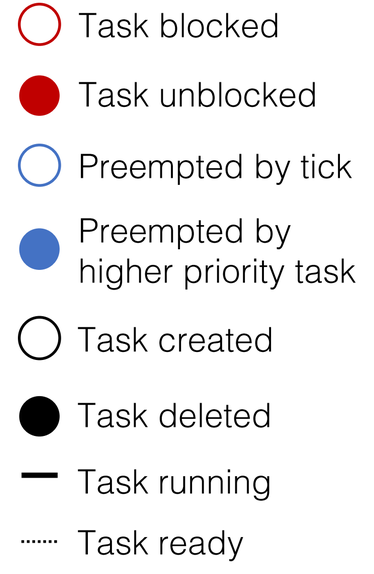
- When calling xTaskCreatePinnedToCore() the newly created task can preempt the current running task if it has a higher priority.
- ESP-IDF will automatically create a Main Task pinned to CPU0 with priority 1
- The Main Task calls a user defined function app_main() which acts as the entry point for ESP-IDF applications
- Beware of using interrupts from unpinned tasks. Interrupts cannot be allocated across CPUs
- Make sure tasks are created with sufficient stack size

Coding Example (pinned to CPU0)

```c
#include <stdio.h>
#include <stdlib.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

void example_task(void *arg)
{
    //Argument is task number
    char task_identifier = (char)arg;
    //Loop a fixed number of times
    for (int i = 1; i <= 3; i++) {
        printf("Task %c running for the %d time\n",
                task_identifier, i);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
    //Delete the task
    vTaskDelete(NULL);
}

void app_main()
{
    xTaskCreatePinnedToCore(example_task, "task", 2048, (void *)'A', 4 , NULL, 0);
    xTaskCreatePinnedToCore(example_task, "task", 2048, (void *)'B', 3 , NULL, 0);
    xTaskCreatePinnedToCore(example_task, "task", 2048, (void *)'C', 2 , NULL, 0);
    printf("All tasks created\n");
}
```

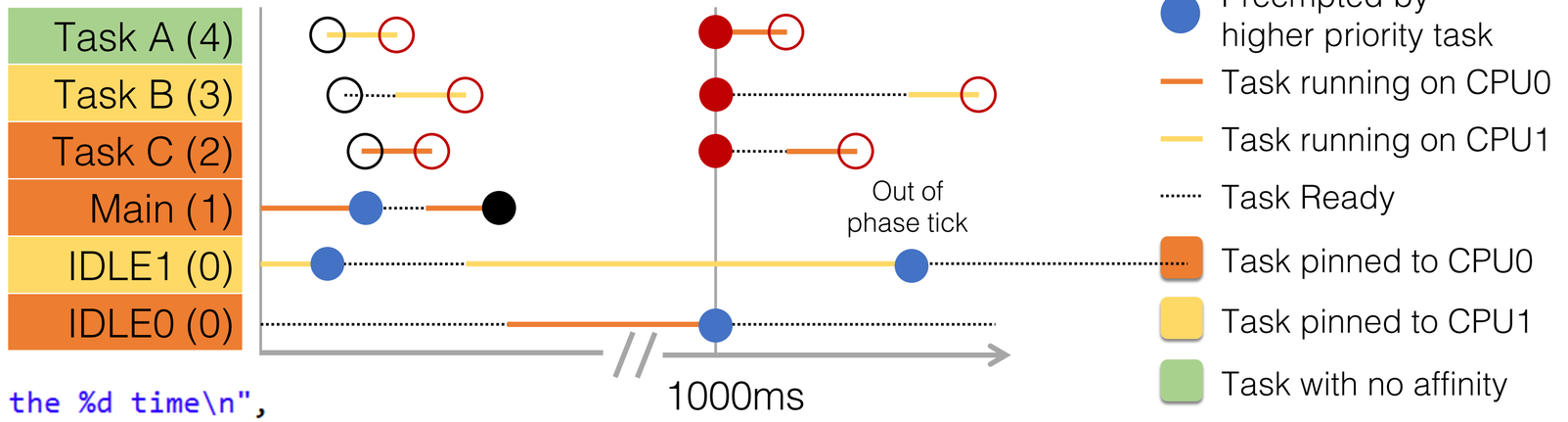



```
Task A running for the 1 time
Task B running for the 1 time
Task C running for the 1 time
All tasks created
Task A running for the 2 time
Task B running for the 2 time
Task C running for the 2 time
Task A running for the 3 time
Task B running for the 3 time
Task C running for the 3 time
```



Coding Example (SMP)

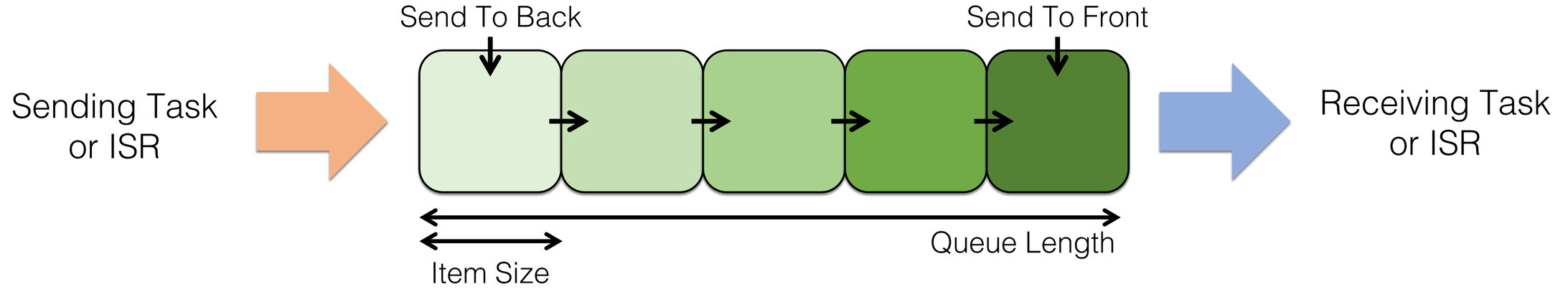
```c
#include <stdio.h>
#include <stdlib.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

void example_task(void *arg)
{
    //Argument is task number
    char task_identifier = (char)arg;
    //Loop a fixed number of times
    for (int i = 1; i <= 3; i++) {
        printf("Task %c running on core %d for the %d time\n",
                task_identifier, xPortGetCoreID(), i);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
    //Delete the task
    vTaskDelete(NULL);
}

void app_main()
{
    vTaskDelay(10);     //Let initialization finish on other CPU
    xTaskCreatePinnedToCore(example_task, "task", 2048, (void *)'A',
                            4 , NULL, tskNO_AFFINITY);
    xTaskCreatePinnedToCore(example_task, "task", 2048, (void *)'B',
                            3 , NULL, 1);
    xTaskCreatePinnedToCore(example_task, "task", 2048, (void *)'C',
                            2 , NULL, 0);
    printf("All tasks created\n");
}
```




```
Task A running on core 1 for the 1 time
Task C running on core 0 for the 1 time
Task B running on core 1 for the 1 time
All tasks created
Task A running on core 0 for the 2 time
Task C running on core 0 for the 2 time
Task B running on core 1 for the 2 time
Task A running on core 0 for the 3 time
Task C running on core 0 for the 3 time
Task B running on core 1 for the 3 time
```

Queues

- Queues provide a way to send data between multiple tasks, and between tasks and interrupts
- Each queue can hold a fixed number of items, each item is of fixed size, and items are passed by copy
- Tasks can block when attempting to send/receive to/from a full/empty queue. When multiple tasks block on the queue, the highest priority task will unblock first.
- An ISR will not block but will return immediately if it fails to send/receive from a queue
- Queues are FIFO by default, but offer write to front or overwrite capabilities



Queue API

Sending Functions	
xQueueSendToBack() xQueueSendToBackFromISR()	Copy an item to the back of the queue from a task/ISR. ISR version will not block.
xQueueSendToFront() xQueueSendToFrontFromISR()	Copy an item to the front of the queue from a task/ISR. ISR version will not block.
xQueueOverwrite() xQueueOverwriteFromISR()	Copy and item to the back of the queue from a task/ISR but overwrite if there is not space available. Only call on queues of length one.

Receiving Functions	
xQueueReceive() xQueueReceiveFromISR()	Receive and remove an item from the front of the queue. ISR version will not block.
xQueuePeek() xQueuePeekFromISR()	Receive but don't remove an item from the front of the queue. ISR version will not block.

Utility Functions	
xQueueCreate()	Creates a queue
vQueueDelete()	Deletes a queue. Only call when no tasks are blocked on the queue.

Queue Example

```
7  void send_task(void *arg)
8  {
9      QueueHandle_t handle = (QueueHandle_t)arg;
10     uint32_t data = 0;
11     printf("Send task running\n");
12     //Send data
13     for (int i = 0; i < 9; i++) {
14         xQueueSendToBack(handle, &data, portMAX_DELAY);
15         printf("Sent %d\n", data);
16         data++;
17     }
18     //Delete task
19     printf("Sending completed\n");
20     vTaskDelete(NULL);
21 }
```

```
23 void rec_task(void *arg)
24 {
25     QueueHandle_t handle = (QueueHandle_t)arg;
26     uint32_t data;
27     printf("Receive task running\n");
28     //Receive data
29     for (int i = 0; i < 9; i++) {
30         xQueueReceive(handle, &data, portMAX_DELAY);
31         printf("Received %d\n", data);
32     }
33     //Delete task
34     printf("Received completed\n");
35     vTaskDelete(NULL);
36 }
```

```
38 void app_main()
39 {
40     //Create Queue
41     QueueHandle_t handle = xQueueCreate(3, sizeof(uint32_t));
42     //Create Tasks
43     xTaskCreatePinnedToCore(rec_task, "rec", 2048,
44                             (void *)handle, 2, NULL, 0);
45     xTaskCreatePinnedToCore(send_task, "send", 2048,
46                             (void *)handle, 3, NULL, 0);
47 }
```

```
Receive task running
Send task running
Sent 0
Sent 1
Sent 2
Sent 3
Received 0
Sent 4
Received 1
Sent 5
Received 2
Sent 6
Received 3
Sent 7
Received 4
Sent 8
Sending completed
Received 5
Received 6
Received 7
Received 8
Received completed
```

Queue is empty, Receiving Task blocks.
Sending Task sends items to queue but Receiving Task does not run immediately as Sending Task has a higher priority. Sending Task blocks when the queue is full.
Receiving Task can finally run because Sending Task has blocked. Receiving Task reads an item from the queue and unblocks the Sending Task.
Sending Task has sent all its data. Receiving task reads until the queue is empty

Semaphores

- A Semaphore is a data type used to synchronize multiple tasks, or control access to a common resource between tasks. Semaphores can be thought of as a simple variable containing a count value.
  - Every time the semaphore is taken, the count is decremented
  - Every time the semaphore is given, the count is incremented
  - When a task attempts to take to take a semaphore with count == 0, it will block

	Binary Semaphore	Counting Semaphore
Description	Binary Semaphores are implemented as queues of length one and zero item size. Therefore the count value can only be 1 or 0.	Counting Semaphores are implemented as queues of length one or larger and zero item size.
Use case	• Commonly used to synchronize another task. A task will block on an empty semaphore, and wait for another task to give the semaphore to trigger it to run.	• Can be used to limit concurrent access to a resource e.g. UART channels. Each task/ISR using the resource must take the semaphore, and give when done. • Can be used to synchronize multiple tasks
Creation API	xSemaphoreCreateBinary()	xSemaphoreCreateCounting()
Take API	xSemaphoreTake() or xSemaphoreTakeFromISR()	
Give API	xSemaphoreGive() or xSemaphoreGiveFromISR()	
Deletion API	vSemaphoreDelete()	

Semaphores Example

Binary Semaphore

```
void sync_task(void *arg)
{
    SemaphoreHandle_t handle = (SemaphoreHandle_t)arg;
    //Block on binary semaphore
    printf("Blocking on semaphore\n");
    xSemaphoreTake(handle, portMAX_DELAY);
    printf("Semaphore taken\n");
    vTaskDelete(NULL);
}

void app_main()
{
    //Create binary semaphore
    SemaphoreHandle_t handle = xSemaphoreCreateBinary();
    //Create task to synchronize on
    xTaskCreatePinnedToCore(sync_task, "sync", 2048,
                            (void *)handle, 2, NULL, 0);
    //Unblock task by giving semaphore
    printf("Giving Semaphore\n");
    xSemaphoreGive(handle);
}
```

```
Blocking on semaphore
Giving Semaphore
Semaphore taken
```

Counting Semaphore

```
void sync_task(void *arg)
{
    SemaphoreHandle_t handle = (SemaphoreHandle_t)arg;
    printf("Running task\n");
    vTaskDelay(pdMS_TO_TICKS(10));
    //Signify task completion by giving semaphore
    printf("Signal completion\n");
    xSemaphoreGive(handle);
    vTaskDelete(NULL);
}
void app_main()
{
    //Create counting semaphore
    SemaphoreHandle_t handle = xSemaphoreCreateCounting(4, 0);
    //Create four tasks
    for (int i = 0; i < 4; i++) {
        xTaskCreatePinnedToCore(sync_task, "sync", 2048,
                                (void *)handle, 2, NULL, 0);
    }
    //Wait for tasks to finish by taking semaphore
    printf("Waiting for tasks to complete\n");
    for (int i = 0; i < 4; i++) {
        xSemaphoreTake(handle, portMAX_DELAY);
    }
    printf("All tasks complete\n");
}
```

```
Running task
Running task
Running task
Running task
Waiting for tasks to complete
Signal completion
Signal completion
Signal completion
Signal completion
All tasks complete
```

Mutexes

- Mutexes are a data type used for **Mut**ual **Ex**clusion. In other words ensuring that only a single task/ISR can access a particular resource at any one time. Mutexes are similar to binary semaphores in FreeRTOS.

- Mutexes differ from semaphores in that a task/ISR that takes a mutex must also be the same task/ISR that returns it.

- Mutexes contain a priority inheritance mechanism. If a lower priority Task A has taken a mutex and a higher priority Task B attempts to take the same mutex, Task A will inherit the priority of Task B.

- A recursive mutex can be taken repeatedly by the same owner. Therefore if Task A takes a mutex, it can take it again. However it must return the mutex the same number of times it has taken it.

- Use xSemaphoreCreateMutex() to create a mutex
- Use xSemaphoreCreateRecursiveMutex() to create a recursive mutex

Critical Sections and Spinlocks

- Mutexes prevent concurrent access from tasks, but not from ISRs
- Critical sections are regions of code which needs protected access to a shared resource
- Single Core FreeRTOS implements critical sections by disabling interrupts
- SMP FreeRTOS implements critical sections by disabling interrupts and using a spinlock
  - When a spinlock is taken, and another task/ISR attempts to take the same lock, it will spin (busy-wait) until the lock is released.
- Beware of deadlock when using spinlocks

Extra Topics

Read about these features in your own time

- Queuesets allows a task to simultaneously block on multiple queues
- Software Timers allows small snippets of code to run periodically
- Task Notifications can unblock a task and are a light weight version of binary semaphores
- Event Groups allow tasks to block until a certain combination of flags are set
- Ring Buffers are like queues but allows unfixed item size.