FreeRTOS Architecture Part 1

Name

Universidad Panamericana

Presentation August 29, 2024

Contents

1. Memory Managment
   - Memory Hierarchy
   - Copy
   - Reference
2. Stack and Heap memory
3. Function Pointers
   - Callbacks
4. FreeRTOS
   - Task
   - Scheduler
   - Systick
   - VtaskDelay

Memory Managment

Memory Hierarchy: A Light-Hearted Tour

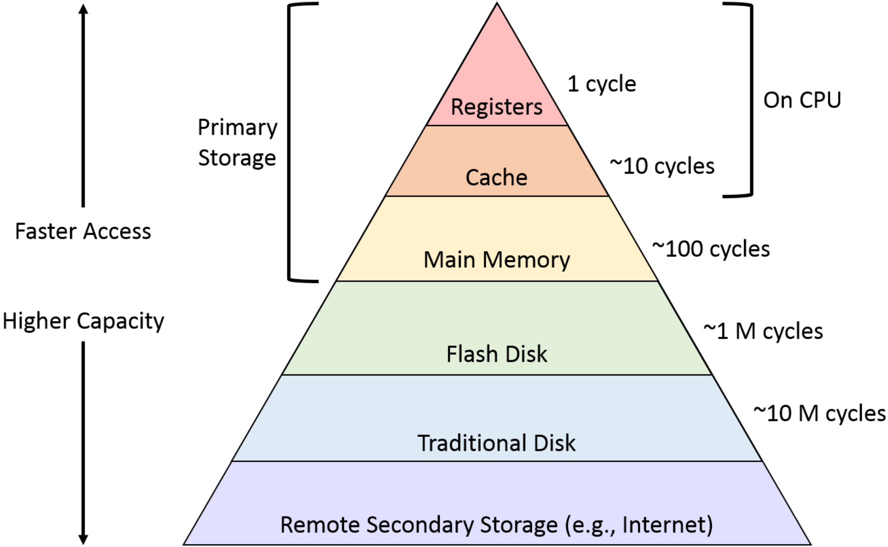
- **Registers:** The speed-demons of memory. Too fast to care, but you really should!
- **Cache:** The backseat driver of computing. It makes decisions you didn't ask for, often with surprising results.

Friendly Reminder

Regularly clearing your cache: not just good practice, it's like digital detox for your devices!

- **RAM (Random Access Memory):** The workaholic of memory. When it runs out, things go south quickly—plan wisely!
- **Storage:** The elephant's graveyard. Where all your code and files go to rest. Yes, your code lives somewhere physical!

Memory Hierarchy



The Memory Hierarchy

How does many values has singles variable?

- One?
- Two?

How does many values has singles variable?

- One?
- Two?

A variable has two values

- One : Its current value
- Two : Its current addres

Passing by Copy

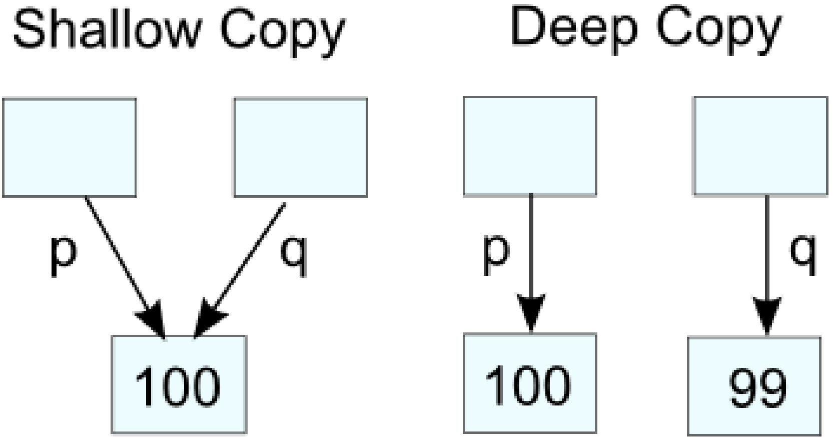
- When parameters are passed by copy, a new instance of the argument is created.
- Modifications within the function do not affect the original variable.
- Best used when you need to ensure the original data remains unchanged.

```cpp
void incrementByCopy(int x) {
    x = x + 1;
    cout << "Inside function: " << x << endl;
}

int main() {
    int a = 5;
    incrementByCopy(a);
    cout << "Outside function: " << a << endl;
}
```

Passing by Reference

- Passing by reference sends a reference to the original variable.
- Any changes inside the function affect the original variable.
- More efficient for large data structures but must be used carefully.

```cpp
void incrementByReference(int& x) {
    x = x + 1;
    cout << "Inside function: " << x << endl;
}

int main() {
    int a = 5;
    incrementByReference(a);
    cout << "Outside function: " << a << endl;
}
```


Shallow Copy
p
q
100
Deep Copy
p
q
100
99

Stack and Heap memory

Stack Memory

- **Definition:** Stack memory is a region of memory where data is added or removed in a last-in-first-out (LIFO) manner.
- **Usage:** Primarily used for static memory allocation, including function call stack (local variables, function parameters).
- **Characteristics:**
  - Fixed size, typically allocated at the start of the program.
  - Automatic management, with variables being pushed { onto the stack and popped off }when no longer needed.
  - Yes the { and } mean something in the code!!! `QuizzSwitchCase.cpp`
  - Fast access due to locality of reference and simplicity of allocation mechanism (moving the stack pointer).
- **Limitations:**
  - Limited space, which can lead to stack overflow if too many function calls or large arrays are declared.
  - No resizing, and not suitable for dynamically allocated data.

Heap Memory

- **Definition:** Heap memory is a region of memory used for dynamic memory allocation where blocks of memory are allocated and freed in an arbitrary order.
- **Usage:** Utilized for allocating memory at runtime when the amount of memory needed cannot be determined at compile time.
- **Characteristics:**
  - Dynamically grows and shrinks based on application needs.
  - Managed through library routines or operating system functions like malloc() and free() in C.
  - Flexible, but with higher overhead and slower access compared to stack memory.
- **Limitations:**
  - Can lead to memory fragmentation.
  - User error for bad manual handling. `BadLinkedList.cpp`

Function Pointers

Function Pointers

- **What Are They?** Variables that store the address of a function.
- **Use Cases:**
  - Modular software design.
  - Passing functions as arguments.
- **Syntax Example:**
  - `void (*funPtr)(int) = &fun;`

Callbacks

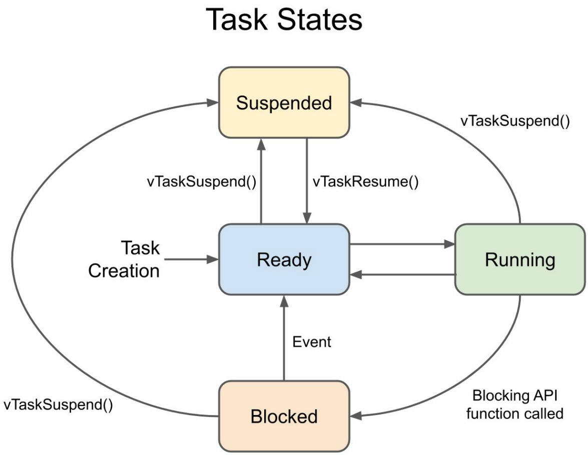
- **Definition:** Functions passed as arguments to other functions.
- **Purpose:**
  - Allow a lower-level software layer to call a function in a higher-level layer.
  - Used extensively for event-driven programming.
- **Example Use:**
  - Asynchronous data processing.
  - Reacting to user inputs or software events.

FreeRTOS

Task similar to Function Pointer

- **Task as Function Pointer:**
  - In FreeRTOS, tasks are defined by function pointers.
  - Function defines task behavior and is invoked when the task runs.
- **How It Works:**
  - `xTaskCreate(pvTaskCode, "TaskName", STACK_SIZE, NULL, Priority, NULL);`
- **Advantages:**
  - Flexibility in task management.
  - Easy integration of different functionalities.

Task States in FreeRTOS



Understanding Task States

- **Task Creation**: A new task starts in the *Ready* state, waiting to be scheduled to run.
- **Ready**: Tasks in this state are ready to run but are currently not being executed by the CPU.
- **Running**: The state of the currently executing task. Only one task can be in this state at a time on single-core systems.
- **Blocked**: A task enters this state if it cannot continue because it is waiting for an event or resource. It will remain blocked until the event occurs or the resource becomes available.
- **Suspended**: Tasks in this state are intentionally suspended by the application, possibly to conserve power or CPU time. They are not schedulable until they are explicitly resumed.
- **Transitions**:

Task States in FreeRTOS

- *vTaskSuspend()* moves a task to *Suspended*.
- *vTaskResume()* moves a task from *Suspended* back to *Ready*.
- An event or the availability of a resource moves a task from *Blocked* to *Ready*.
- Tasks switch from *Ready* to *Running* based on scheduler decisions and priority.

Note

Only the scheduler can move tasks into the *Running* state or handle transitions when a blocking API function is called.

SysTick Timer in FreeRTOS

- **Purpose:** The SysTick timer generates interrupt requests at a selectable interval and is often used to increment the system tick count in RTOS.
- **Function:** Essential for task scheduling, timekeeping, and implementing time delays.
- **System Tick:** Typically configured to tick once per millisecond, which serves as the heartbeat for task switches and timing operations.

Task Management and Scheduler States

- **vTaskDelay:** Delays the execution of the current task, allowing other tasks to run.
- **Scheduler States:**
  - **Running:** The currently executing task.
  - **Ready:** Tasks that are ready to run when given CPU time.
  - **Blocked:** ‘vTaskDelay‘ moves the task to the Blocked state until the delay time expires.
  - **Suspended:** Tasks that have been explicitly suspended, not affected by ‘vTaskDelay‘.
- **Task Switching:** ‘vTaskDelay‘ can trigger a context switch if a higher priority task is ready to run.

Practical Usage of vTaskDelay

```c
#include "FreeRTOS.h"
#include "task.h"

void vTaskCode(void *pvParameters)
{
    for (;;)
    {
        // Perform task operation
        printf("Task is running.\n");
        // Delay the task for 100 ticks
        vTaskDelay(100);
        printf("Task resumes after delay.\n");
    }
}
```