Module 3 - Memory Management & Queue

By WN

Tujuan Pembelajaran

Setelah menyelesaikan modul ini, praktikan diharapkan mampu:

- Memahami dan dapat mendemonstrasikan jenis-jenis alokasi memory yang terjadi pada suatu sistem.
- Memahami kasus dan cara manajemen heap pada RTOS.
- Memahami definisi data structure Queue dan kepentingannya dalam aplikasi IOT dan Sistem Embedded Multithreaded.

Why Memory Management?

Manajemen memori merupakan hal yang sangat penting dalam aplikasi IoT dan Sistem Embedded. Bayangkan bila sistem menggunakan tipe data yang memerlukan ukuran data yang lebih dari yang dialokasikan, fungsi yang dipanggil pada task tidak diterminasi dengan baik sehingga mengakibatkan stack overflow, atau memory yang dialokasikan pada heap tidak di free sehingga terjadi memory leak.

Hal-hal tersebut Masalah-masalah tersebut dapat berujung pada kegagalan sistem, crash, atau error yang memengaruhi keseluruhan performa perangkat. Oleh karena itu, pemahaman dan penerapan manajemen memori yang baik sangat penting untuk menjaga sistem tetap stabil, efisien, dan andal.

Beberapa masalah seperti memory leak pada dasarnya dapat diatasi dengan teknologi garbage collector seperti pada bahasa pemrograman modern. Namun, ini jarang digunakan pada sistem real-time karena dua alasan utama: keterbatasan resource yaitu RAM dan Processor yang terbatas, serta strict deadline yang harus dipenuhi oleh task.

Tipe Memory Allocation

Dalam sebuah program, baik secara umum maupun pada Sistem Embedded, terdapat beberapa jenis alokasi memory yang dapat dilakukan, diantaranya sebagai berikut: [1]

60c3b8b7-f4af-4a07-b139-0acae5a846fb

Static Variable

Static memory digunakan untuk menyimpan variabel global maupun variabel yang dideklarasikan sebagai static di dalam kode. Berbeda dengan variabel lokal biasa, variabel static tidak hilang setelah fungsi selesai dijalankan, melainkan tetap ada (persist) sepanjang program berjalan. Nilainya akan diingat pada pemanggilan fungsi berikutnya.

Variable-Variable seperti global counter, variable pin / port, merupakan contoh-contoh static variable pada sistem Embedded.

Dalam praktikum ini, karena pemrograman menggunakan bahasa C++, setiap variabel bersifat type-based, artinya ukuran memori sudah ditentukan berdasarkan tipe data yang digunakan. Oleh karena itu, pada static variable, pemilihan tipe dan ukuran variabel harus tepat agar penggunaan memori lebih efisien dan sesuai kebutuhan. Oleh sebab itu, tidak jarang ditemukan fixed-width integer seperti int8_t, int16_t, uint16_t, uint32_t dan sebagainya.

Informasi mengenai fixed width integer dapat dipelajari secara lebih pada link berikut.

Stack

Stack digunakan ketika terjadi alokasi otomatis oleh variabel lokal. Memori stack diatur dengan prinsip Last-In-First-Out (LIFO), di mana variabel dari suatu fungsi akan didorong (push) ke stack saat fungsi dipanggil. Setelah fungsi selesai dan kembali, variabel-variabel tersebut akan dikeluarkan (pop) dari stack, sehingga fungsi dapat melanjutkan eksekusi seperti sebelumnya.

Pada FreeRTOS, ketika sebuah task dibuat menggunakan xTaskCreate(), sistem operasi akan mengalokasikan sebagian memori heap untuk task tersebut. Alokasi ini terdiri dari dua bagian:

- Task Control Block (TCB) Berisi informasi penting tentang task, seperti prioritas, state, dan pointer ke local stack.
- Local Stack Task Digunakan khusus untuk menyimpan variabel lokal dan data saat fungsi dalam task tersebut dipanggil, mirip seperti stack global pada program utama tetapi terbatas hanya untuk task itu.

Setiap variabel lokal yang dibuat dalam sebuah fungsi task akan ditempatkan di local stack milik task tersebut. Oleh karena itu, sangat penting untuk memperkirakan kebutuhan stack sebelum membuat task, lalu menentukan ukurannya pada parameter stack size di xTaskCreate(). Jika ukuran stack terlalu kecil, task bisa mengalami stack overflow yang menyebabkan error atau crash pada sistem.

Heap

Heap adalah area memori yang digunakan untuk alokasi dinamis. Tidak seperti alokasi stack yang dilakukan secara otomatis, heap harus dialokasikan secara eksplisit oleh programmer. Pada bahasa pemrograman C dan C++, proses alokasi memory pada heap dilakukan melalui fungsi berikut.

- malloc() → untuk mengalokasikan memori
- free() → untuk melepaskan kembali memori yang sudah tidak dipakai

Proses ini disebut dynamic allocation. Jika memori heap tidak dibebaskan setelah selesai digunakan, maka akan terjadi memory leak, yang bisa menyebabkan sistem kehabisan memori, crash, atau bahkan korupsi data pada area memori lain.

Dalam FreeRTOS, terdapat potensi dimana dua buah task berbeda mencoba mengakses lokasi memory yang sama, sehingga fungsi malloc() dan free() tidak thread-safe dan tidak aman digunakan antar task. Karena itu, FreeRTOS menyediakan fungsi khusus:

- pvPortMalloc() → untuk mengalokasikan memori dari heap global FreeRTOS
- vPortFree() → untuk mengembalikan memori yang sudah tidak digunakan

Dengan cara ini, alokasi heap lebih aman di lingkungan multitasking, karena RTOS mengatur sinkronisasi dan pengelolaan memori agar tidak saling bertabrakan antar task.

Hubungan Stack dan Heap

Pada kebanyakan sistem, stack dan heap tumbuh saling mendekati dalam ruang memori yang sama. Jika penggunaan keduanya tidak dikontrol, keduanya bisa bertabrakan (stack-heap collision), yang menyebabkan data saling menimpa dan mengakibatkan error serius.

b9d52446-ebca-4ae5-9e8d-9a4a035d1e4d

Demonstrasi Memory Allocation

Berikut merupakan contoh task yang dapat menyebabkan memory leak akibat kurangnya memory deallocation menggunakan vPortFree. Jika dijalankan, sewaktu-waktu jumlah memory yang ada akan habis sehingga sistem akan berhenti membaca sensor.

```
#include <Arduino.h>
#include <FreeRTOS.h>

void SensorTask(void *pvParameters) {
    while (1) {
        // Alokasikan buffer untuk data sensor
        int *sensorData = (int *) pvPortMalloc(50 * sizeof(int)); // buffer 50 data
        if (sensorData == NULL) {
            Serial.println("Error - Heap exhausted.");
        }

        // Pengambilan data sensor
        for (int i = 0; i < 50; i++) {
            sensorData[i] = analogRead(A0);
        }

        // Cetak sebagian hasil
        Serial.print("Sensor[0] = ");
        Serial.print(sensorData[0]);
        Serial.print(", Sensor[49] = ");
        Serial.println(sensorData[49]);

        // Tidak ada vPortFree(sensorData);


        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}

void setup() {
    Serial.begin(115200);
    delay(1000);

    xTaskCreate(
        SensorTask,
        "SensorTask",
        2048,
        NULL,
        1,
        NULL
```

```
    );
}

void loop() {

}
```

Heap Configuration FreeRTOS

FreeRTOS menyediakan beberapa skema pengelolaan heap (memori dinamis), yang berbeda dari segi kompleksitas, fitur, dan trade-off-nya [2].

Saat FreeRTOS membutuhkan memori dinamis (misalnya saat membuat task, queue, atau objek kernel lainnya), ia menggunakan fungsi pvPortMalloc() dan vPortFree() bukan malloc()/free() standar.

Ada lima implementasi contoh heap yang dibawa oleh FreeRTOS yang memiliki karakteristiknya masing-masing:

1. heap_1: sederhana, memori hanya dialokasikan tapi tidak bisa dibebaskan (tidak ada vPortFree()). Cocok jika objek hanya dibuat sekali dan tidak dihapus.
2. heap_2: memungkinkan free, tetapi tidak menggabungkan blok-blok memori bebas yang berdekatan (tidak ada coalescence).
3. heap_3: membungkus (wraps) malloc()/free() standar sehingga bisa memakai alokasi memori dari C library, dengan tambahan pengendalian (misalnya keamanan antar thread) oleh FreeRTOS.
4. heap_4: menggunakan algoritma first-fit, serta mendukung coalescing (menggabungkan blok bebas yang bersebelahan), yang mengurangi fragmentasi memori. Cocok jika sistem sering melakukan alokasi dan pembebasan memori dengan ukuran yang berbeda-beda.
5. heap_5: mirip dengan heap_4, tetapi memiliki kemampuan untuk menggunakan beberapa wilayah memori yang tidak harus kontinu (tidak berurutan) sebagai satu heap kumulatif.

Proses perubahan heap management biasanya dilakukan jika melakukan build FreeRTOS secara mandiri atau pada ESP-IDF dan diluar cakupan modul ini.

Queue

Data structure queue mungkin sudah familiar setelah digunakan pada praktikum pemrograman sebelum sebelumnya. Data structure ini bersifat FIFO dimana data yang masuk pertama kedalam queue akan menjadi data yang pertama keluar dari queue. [3]

Dalam konteks IoT dan FreeRTOS, queue digunakan sebagai medium untuk mengirimkan data antar task atau antar perangkat dan server.

Mengapa queue diperlukan?

Queue diperlukan karena pada sistem multitasking, beberapa task bisa saja mencoba mengakses atau menulis ke memori yang sama secara bersamaan.

Hal ini dapat kita contohkan pada skenario tersebut. Pada skenario ini kita memiliki satu buah global variable (seperti counter) yang akan diakses ataupun diubah oleh task. Task A akan mengubah nilai global variable ini (seperti mengincrement global counter) setelah menjalankan task yang dimiliki. Task C juga akan melakukan hal yang sama setelah menyelesaikan tasknya. Task B akan membaca nilai dari global variable ini pada proses tasknya dan melakukan printing pada serial monitor. Dikarenakan seluruh task berjalan secara bersamaan / paralel, akan terdapat kemungkinan dimana hasil write pada global variable oleh task A akan di overwrite oleh task B. image

Dengan kata lain, jika pengaksesan dan transfer data pada sistem multithreaded / parallel ini dibiarkan tanpa pengaturan, maka akan terjadi memory overwriting atau race condition, di mana data yang sudah ditulis oleh satu task bisa tertimpa oleh task lain sebelum sempat diproses.

Dengan adanya queue, setiap data yang dikirim akan disimpan secara terpisah dalam antrian, sehingga proses tulis dan baca berlangsung secara atomic (tidak bisa diinterupsi oleh task lain di tengah jalan). Selain itu, sifat FIFO dari queue menjamin bahwa urutan data tetap terjaga, sehingga task penerima dapat memproses data sesuai dengan urutan kedatangannya.

88d32074-22e7-4fc3-943f-9e59b06dede9

Dalam konteks IoT, queue juga berperan sebagai buffer, misalnya ketika perangkat ingin mengirimkan data ke server tetapi koneksi sedang terputus. Data dapat terlebih dahulu disimpan di dalam queue untuk kemudian dikirimkan kembali saat koneksi sudah tersedia, sehingga tidak ada data yang hilang.

Menggunakan Queue pada FreeRTOS

Pada FreeRTOS, disediakan API khusus untuk membuat, menulis, dan membaca queue.

Membuat Queue

Queue dibuat dengan fungsi xQueueCreate(), yang membutuhkan dua parameter utama:

- uxQueueLength → jumlah maksimum item yang dapat disimpan dalam queue.
- uxItemSize → ukuran (dalam byte) tiap item yang akan disimpan.

Contoh: membuat queue yang dapat menyimpan 10 integer (int):

```
QueueHandle_t xQueue;
xQueue = xQueueCreate(10, sizeof(int));
if (xQueue == NULL) {
    Serial.println("Error: Queue creation failed.");
}
```

Dapat dilihat bahwa queue memiliki ukuran tetap, dalam hal ini adalah 10 x ukuran byte dari integer.

if (xQueue == NULL) dapat ditambahkan untuk memastikan queue berhasil dibuat, dan mencegah kegagalan pembuatan queue karena keterbatasan memori.

Mengirim Data ke Queue

Task atau ISR (Interrupt Service Routine) dapat mengirim data ke queue menggunakan: xQueueSend() → untuk mengirim dari task biasa. xQueueSendFromISR() → untuk mengirim dari ISR. Contoh: mengirim data sensor ke queue dari sebuah task:

```
int sensorValue = analogRead(A0);
xQueueSend(xQueue, &sensorValue, portMAX_DELAY);
```

Membaca Data dari Queue

Task penerima membaca data dari queue menggunakan xQueueReceive(). Data akan dihapus dari queue setelah berhasil dibaca.

```
int receivedValue;
if (xQueueReceive(xQueue, &receivedValue, portMAX_DELAY) == pdPASS) {
    Serial.print("Received: ");
    Serial.println(receivedValue);
}
```

Parameter-Parameter pada Queue

- Parameter portMAX_DELAY memiliki fungsi berikut:
  - Pada xQueueSend(): task akan menunggu jika queue sedang penuh, sampai ada space kosong untuk memasukkan data.
  - Pada xQueueReceive(): task akan menunggu jika queue kosong, sampai ada data baru masuk.\
- Blocking vs Non-Blocking pada Queue
  - Blocking :Task akan masuk mode blocking ketika kita memberikan timeout > 0 atau portMAX_DELAY pada fungsi queue (xQueueSend() atau xQueueReceive()). Pada saat ini, task akan menunggu hingga block berakhir (dalam hal ini, space tersedia pada queue)
  - Non-Blocking: Task akan masuk mode non-blocking ketika kita memberikan timeout = 0, dimana fungsi queue akan langsung kembali meskipun queue penuh atau kosong. Dengan kata lain, task harus tetap berjalan terus-menerus tanpa tertahan oleh queue.
- Return Value
  - pdTRUE / pdFALSE → digunakan pada fungsi seperti xQueueSend() dan xQueueReceive(), menunjukkan apakah operasi berhasil (pdTRUE) atau gagal (pdFALSE).
  - pdPASS / pdFAIL → digunakan pada operasi yang lebih kompleks atau alokasi memory, menunjukkan keberhasilan (pdPASS) atau kegagalan (pdFAIL).
  - pdTRUE/pdFALSE biasanya dipakai pada fungsi queue (xQueueSend(), xQueueReceive()) untuk menunjukkan status keberhasilan operasi. Sedangkan pdPASS/pdFAIL dipakai pada operasi FreeRTOS yang lebih umum (misal pembuatan queue, semaphore, atau alokasi heap)

Informasi lebih lanjut mengenai parameter-parameter pada API queue dapat dibaca pada dokumentasi FreeRTOS

Contoh Mengirimkan dan Menerima Data Serial Via Queue

```
#include <Arduino.h>
#include <FreeRTOS.h>

// Handle untuk Queue
QueueHandle_t xQueue;
```

```
// Ukuran buffer pesan
#define MSG_MAX_LEN 50

// Task untuk membaca input serial sampai newline
void SerialReadTask(void *pvParameters) {
  while (1) {
    if (Serial.available() > 0) {
      // Baca string sampai newline
      String input = Serial.readStringUntil('\n');

      // Pastikan tidak melebihi buffer
      if (input.length() > 0 && input.length() < MSG_MAX_LEN) {
        char buffer[MSG_MAX_LEN];
        input.toCharArray(buffer, MSG_MAX_LEN);

        // Kirim ke queue
        if (xQueueSend(xQueue, buffer, portMAX_DELAY) == pdPASS) {
          Serial.println("[Sent to Queue]");
        }
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS);
  }
}

// Task untuk membaca dari queue dan menampilkan pesan
void SerialPrintTask(void *pvParameters) {
  char received[MSG_MAX_LEN];

  while (1) {
    if (xQueueReceive(xQueue, &received, portMAX_DELAY) == pdPASS) {
      Serial.print("Message received: ");
      Serial.println(received);
    }
  }
}

void setup() {
  Serial.begin(115200);
  delay(1000);
```

```
  // Buat queue dengan kapasitas 5 pesan
  xQueue = xQueueCreate(5, MSG_MAX_LEN);

  if (xQueue != NULL) {
    xTaskCreate(SerialReadTask, "SerialRead", 4096, NULL, 1, NULL);
    xTaskCreate(SerialPrintTask, "SerialPrint", 4096, NULL, 1, NULL);
  } else {
    Serial.println("Error: Queue creation failed!");
  }
}

void loop() {
}
```

Mengirimkan Struct dengan Queue

Umumnya data tidak dikirimkan secara langsung pada queue, namun terlebih dahulu di masukkan kedalam sebuah struct. Ini berguna ketika data yang dikirimkan berbentuk objek yang memiliki beberapa atribut, contohnya ketika mengirimkan data suhu, kelembapan, beserta waktu saat ini dalam satu buah struct.

Pass Struct by Value

Jika struct dikirimkan by value ke dalam queue, maka FreeRTOS akan menyalin seluruh isi struct ke dalam buffer queue. Cara ini sederhana dan aman, tetapi untuk struct berukuran besar, proses copy bisa memperlambat sistem dan memperboros RAM untuk melakukan proses duplikasi.

```
//Send
Data_t data;
data.tick = xTaskGetTickCount();
snprintf(data.msg, sizeof(data.msg), "Hello Value");
xQueueSend(queue, &data, portMAX_DELAY);

// Receive
Data_t received;
xQueueReceive(queue, &received, portMAX_DELAY);
Serial.println(received.msg);
```

Pass Struct By Reference

Dalam hal ini, pointer memiliki peran penting, dimana dengan pass by reference, kita hanya mengirimkan alamat memori dari struct tersebut ke dalam queue, bukan seluruh isi datanya sehingga data tidak perlu disalin ulang ke dalam buffer queue, sehingga lebih efisien dalam penggunaan memori.

```
// Send
Data_t *data = (Data_t *) pvPortMalloc(sizeof(Data_t));
data->tick = xTaskGetTickCount();
snprintf(data->msg, sizeof(data->msg), "Hello Pointer");
```

```
xQueueSend(queue, &data, portMAX_DELAY);

// Receive
Data_t *received;
xQueueReceive(queue, &received, portMAX_DELAY);
Serial.println(received->msg);
vPortFree(received); // Don't Forget
```

Pass by reference lebih direkomendasikan karena efisiensi dan performanya, namun dalam proses melakukan pass by reference penggunaan pointer perlu diteliti sehingga pengiriman value / memory tidak salah dan memory yang telah dialokasi di free.

Contoh Penggunaan

Berikut merupakan contoh aplikasi Struct dan Queue untuk mengirimkan tick dan pesan dari serial monitor dari suatu task, untuk ditampilkan pada task lainnya dengan format tertentu.

```
/* In this FreeRTOS example, we use xTaskCreatePinnedToCore()
   to show queue communication between tasks.
   We'll send both a serial message and a tick count via a struct pointer. */

#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "Arduino.h"

// Handle untuk Queue
QueueHandle_t xQueue;

// Ukuran buffer pesan
#define MSG_MAX_LEN 50

// Struct untuk data pesan
struct Message {
  TickType_t ticks;
  char text[MSG_MAX_LEN];
};

// Task untuk membaca input serial sampai newline
void SerialReadTask(void *pvParameters) {
  while (1) {
```

```
    if (Serial.available() > 0) {
      // Baca string sampai newline
      String input = Serial.readStringUntil('\n');

      if (input.length() > 0 && input.length() < MSG_MAX_LEN) {
        // Alokasikan memori untuk struct
        Message *msg = (Message *)pvPortMalloc(sizeof(Message));
        if (msg != NULL) {
          msg->ticks = xTaskGetTickCount();
          input.toCharArray(msg->text, MSG_MAX_LEN);

          // Kirim pointer ke queue
          if (xQueueSend(xQueue, &msg, portMAX_DELAY) == pdPASS) {
            Serial.println("[Sent to Queue]");
          } else {
            // Jika gagal, bebaskan memori
            vPortFree(msg);
          }
        }
      }
    }
    vTaskDelay(10 / portTICK_PERIOD_MS);
  }
}

// Task untuk membaca dari queue dan menampilkan pesan
void SerialPrintTask(void *pvParameters) {
  Message *received;

  while (1) {
    if (xQueueReceive(xQueue, &received, portMAX_DELAY) == pdPASS) {
      Serial.print("Message received after ");
      Serial.print((unsigned long)received->ticks);
      Serial.print(" ticks: \"");
      Serial.print(received->text);
      Serial.println("\"");

      // Bebaskan memori setelah dipakai
      vPortFree(received);
    }
```

```
  }
}

void setup() {
  Serial.begin(115200);
  delay(1000);

  // Buat queue untuk menampung 5 pointer ke Message
  xQueue = xQueueCreate(5, sizeof(Message *));
  if (xQueue != NULL) {
    xTaskCreatePinnedToCore(SerialReadTask, "SerialRead", 4096, NULL, 1, NULL, 0);
    xTaskCreatePinnedToCore(SerialPrintTask, "SerialPrint", 4096, NULL, 1, NULL, 1);
  } else {
    Serial.println("Error: Queue creation failed!");
  }
}

void loop() {
}
```

Referensi Lebih Lanjut

- “The FreeRTOSTM Reference Manual.” Available: https://www.freertos.org/media/2018/FreeRTOS_Reference_Manual_V10.0.0.pdf
- “FreeRTOS Memory Management,” Digikey.com, 2021. https://www.digikey.com/en/maker/projects/introduction-to-rtos-solution-to-part-4-memory-management/6d4dfcaa1ff84f57a2098da8e6401d9c
- “FreeRTOS heap memory management - FreeRTOSTM,” Freertos.org, 2024. https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/09-Memory-management/01-Memory-management
- ShawnHymel, “Introduction to RTOS - Solution to Part 5 (FreeRTOS Queue Example),” DigiKey, Feb. 08, 2021. https://www.digikey.com/en/maker/projects/introduction-to-rtos-solution-to-part-5-freertos-queue-example/72d2b361f7b94e0691d947c7c29a03c9
- “C++ Pointers to Structure (With Examples),” Programiz.com, 2025. https://www.programiz.com/cpp-programming/structure-pointer