

Karta pracy: FreeRTOS heap_4

model → prawdziwy upstream → RP2350

Klucz	mpabi/inf/rv32i-freertos/01/heap4/v0.2
Wersja / commit	v0.2 / 2b17bb4ff944
Data	14.07.2026
Allocator	FreeRTOS-Kernel V11.3.0, 9b777ae5c5b8
Port RP2350	fork Raspberry Pi, 4f7299d6ea74
Źródło	portable/MemMang/heap_4.c — bez modyfikacji
Punkt wejścia	stemctl

Tylko trzy kroki

1. First-fit + split: skąd bierze się drugi nagłówek.
2. Free-list + coalescing: dlaczego lista jest uporządkowana po adresie.
3. API FreeRTOS: `pvPortMalloc`, `vPortFree`, stats, OOM.

Warunek zaliczenia

- ☐ przewiduję rozmiary nagłówków i bloków na AMD64/RV32;
- ☐ rozpoznaję fragmentację: suma wolnych bajtów $\neq$ największy blok;
- ☐ pokazuję split i dwa kierunki scalania w Memory;
- ☐ odczytuję `xStart`, `pxEnd`, minimum-ever i stack frame;
- ☐ Task 3 daje `pass=1` na host, Hazard3 i RP2350.

1 Start — polecenia i platformy

```
stemctl test native-amd64 freertos heap4 1
stemctl test hazard3-sim freertos heap4 2
stemctl debug hazard3-sim freertos heap4 3
stemctl test rp2350 freertos heap4 3
stemctl deploy rp2350 freertos heap4 3 \
  --device /dev/bus/usb/BBB/DDD
```

	AMD64	Hazard3	RP2350
Task 1-2	model + sanitizery	model na RTL	build
Task 3	V11.3 + adapter 1-thread	pełny V11.3 + port RISC-V	zgodny core+port RP; al-locator V11.3
Debug	GDB	GDB stub RTL	OpenOCD; SRAM
Deploy	—	—	SPI flash + verify

Termdebug

F5	continue	F9	breakpoint
F10	next	F11	step
Shift-F11	finish	F8	stepi
:StudentStack	frame + stos	:StudentLst	listing ELF-a

Strategia obserwacji: breakpoint przed zmianą → Memory → continue/finish → Memory. Nie opieramy karty na data watchpointach.

2 Task 1 — first-fit i split

Policz przed uruchomieniem

	AMD64	RV32
sizeof(ModelBlock)		
request	13 B	13 B
align8(header + request)		
remainder z 256 B		
offset payloadu		

Do wykonania

1. Wskaż nagłówek bloku przydzielonego i nagłówek remainder.
2. Sprawdź: `wanted + remainder == 256`.
3. Zmień request 13 na 17; przewidź wynik bez uruchamiania.
4. W listingu znajdź dodawanie bajtowego offsetu do adresu areny.

```

1 #include "heap4_model.h"
2
3 enum {
4     ARENA_BYTES = 256,
5     REQUEST_BYTES = 13,
6     WANTED_BYTES = (sizeof(ModelBlock) + REQUEST_BYTES + 7U) & ~7U
7 };
8
9 typedef struct {
10     ModelBlock allocated;
11     uint8_t payload[WANTED_BYTES - sizeof(ModelBlock)];
12     ModelBlock remainder;
13     uint8_t free_payload[ARENA_BYTES - WANTED_BYTES - sizeof(ModelBlock)];
14 } SplitArena;
15
16 typedef union {
17     uint64_t force_alignment;
18     SplitArena blocks;
19 } AlignedSplitArena;
20
21 AlignedSplitArena g_split_arena;
22 ModelBlock *g_free_head;
23 volatile size_t g_header_bytes;
24 volatile size_t g_wanted_bytes;
25 volatile size_t g_remainder_bytes;
26 volatile size_t g_payload_offset;
27 volatile int g_split_addresses_aligned;
28 volatile int g_task01_pass;
29
30 __attribute__((noinline)) void task01_debug_checkpoint(void)
31 {
32     #if defined(__GNUC__)
33         __asm__ volatile("" ::: "memory");
34     #endif
35 }
36
37 int main(void)
38 {
39     SplitArena *arena = &g_split_arena.blocks;
40     uintptr_t base = (uintptr_t)&arena->allocated;
41     uintptr_t remainder = (uintptr_t)&arena->remainder;
42
43     arena->allocated.next = NULL;
44     arena->allocated.size = WANTED_BYTES;
45     arena->remainder.next = NULL;
46     arena->remainder.size = ARENA_BYTES - WANTED_BYTES;
47     arena->payload[0] = (uint8_t)'A';
48     g_free_head = &arena->remainder;
49
50     g_header_bytes = sizeof(ModelBlock);

```

```
51 g_wanted_bytes = WANTED_BYTES;
52 g_remainder_bytes = arena->remainder.size;
53 g_payload_offset = (size_t)((uintptr_t)&arena->payload[0] - base);
54 g_split_addresses_aligned =
55     (base % MODEL_ALIGNMENT) == 0U &&
56     (remainder % MODEL_ALIGNMENT) == 0U;
57 g_task01_pass =
58     g_wanted_bytes == model_align_up(sizeof(ModelBlock) + REQUEST_BYTES) &&
59     (size_t)(remainder - base) == g_wanted_bytes &&
60     g_remainder_bytes + g_wanted_bytes == ARENA_BYTES &&
61     g_payload_offset == sizeof(ModelBlock) &&
62     g_split_addresses_aligned;
63
64 task01_debug_checkpoint();
65
66 #if defined(HOST_PRINTF)
67     printf("header=%zu wanted=%zu remainder=%zu payload=%zu aligned=%d pass=%d\n",
68         g_header_bytes, g_wanted_bytes, g_remainder_bytes,
69         g_payload_offset, g_split_addresses_aligned, g_task01_pass);
70 #endif
71     return g_task01_pass ? 0 : 1;
72 }
```

3 Task 2 — lista adresowa i coalescing

Eksperyment

A:48 | B:48 | C:48 $\xrightarrow{\text{free A, free C}}$ A:48 -> C:48 $\xrightarrow{\text{free B}}$ A:144

Do wykonania

1. Zatrzymaj program w `task02_fragmented_checkpoint`.
2. Zapisz adresy i rozmiary dwóch elementów free-list.
3. Wykonaj F5; zatrzymaj w `task02_debug_checkpoint`.
4. Wskaż pierwszy warunek scalania: z blokiem po prawej.
5. Wskaż drugi warunek scalania: z blokiem po lewej.
6. Wyjaśnij, czemu kolejność po adresie redukuje koszt szukania sąsiadów.

Tabela obserwacji

Stan	liczba bloków	suma	największy
przed B	_____	_____	_____
po B	_____	_____	_____

```

1 #include "heap4_model.h"
2
3 enum { BLOCK_BYTES = 48 };
4
5 typedef struct {
6     ModelBlock first;
7     uint8_t first_payload[BLOCK_BYTES - sizeof(ModelBlock)];
8     ModelBlock middle;
9     uint8_t middle_payload[BLOCK_BYTES - sizeof(ModelBlock)];
10    ModelBlock last;
11    uint8_t last_payload[BLOCK_BYTES - sizeof(ModelBlock)];
12 } CoalesceArena;
13
14 typedef union {
15     uint64_t force_alignment;
16     CoalesceArena blocks;
17 } AlignedCoalesceArena;
18
19 AlignedCoalesceArena g_coalesce_arena;
20 ModelBlock *g_free_head;
21 volatile size_t g_before_nodes;
22 volatile size_t g_before_total;
23 volatile size_t g_before_largest;
24 volatile size_t g_after_nodes;
25 volatile size_t g_after_total;
26 volatile size_t g_after_largest;
27 volatile int g_task02_pass;
28
29 static void insert_and_coalesce(ModelBlock *block)
30 {
31     ModelBlock *previous = NULL;
32     ModelBlock *current = g_free_head;
33
34     while (current != NULL && (uintptr_t)current < (uintptr_t)block)
35     {
36         previous = current;
37         current = current->next;
38     }
39
40     block->next = current;
41     if (current != NULL &&
42         (uint8_t *)block + block->size == (uint8_t *)current)
43     {
44         block->size += current->size;
45         block->next = current->next;
46     }

```

```

47
48     if (previous != NULL &&
49         (uint8_t *)previous + previous->size == (uint8_t *)block)
50     {
51         previous->size += block->size;
52         previous->next = block->next;
53     }
54     else if (previous != NULL)
55     {
56         previous->next = block;
57     }
58     else
59     {
60         g_free_head = block;
61     }
62 }
63
64 static void measure(size_t *nodes, size_t *total, size_t *largest)
65 {
66     ModelBlock *block;
67
68     *nodes = 0U;
69     *total = 0U;
70     *largest = 0U;
71     for (block = g_free_head; block != NULL; block = block->next)
72     {
73         (*nodes)++;
74         *total += block->size;
75         if (block->size > *largest)
76         {
77             *largest = block->size;
78         }
79     }
80 }
81
82 __attribute__((noinline)) void task02_fragmented_checkpoint(void)
83 {
84     #if defined(__GNUC__)
85         __asm__ volatile("" ::: "memory");
86     #endif
87 }
88
89 __attribute__((noinline)) void task02_debug_checkpoint(void)
90 {
91     #if defined(__GNUC__)
92         __asm__ volatile("" ::: "memory");
93     #endif
94 }
95
96 int main(void)
97 {
98     CoalesceArena *arena = &g_coalesce_arena.blocks;
99     size_t nodes;
100     size_t total;
101     size_t largest;
102
103     arena->first.size = BLOCK_BYTES;
104     arena->middle.size = BLOCK_BYTES;
105     arena->last.size = BLOCK_BYTES;
106     g_free_head = NULL;
107
108     insert_and_coalesce(&arena->first);
109     insert_and_coalesce(&arena->last);
110     measure(&nodes, &total, &largest);
111     g_before_nodes = nodes;
112     g_before_total = total;
113     g_before_largest = largest;
114     task02_fragmented_checkpoint();
115
116     insert_and_coalesce(&arena->middle);
117     measure(&nodes, &total, &largest);
118     g_after_nodes = nodes;
119     g_after_total = total;
120     g_after_largest = largest;
121     g_task02_pass =
122         g_before_nodes == 2U && g_before_total == 96U &&
123         g_before_largest == 48U && g_after_nodes == 1U &&
124         g_after_total == 144U && g_after_largest == 144U &&

```

```
125     g_free_head == &arena->first && g_free_head->next == NULL;
126
127     task02_debug_checkpoint();
128
129 #if defined(HOST_PRINTF)
130     printf("before_nodes=%zu before_total=%zu before_largest=%zu "
131           "after_nodes=%zu after_total=%zu after_largest=%zu pass=%d\n",
132           g_before_nodes, g_before_total, g_before_largest,
133           g_after_nodes, g_after_total, g_after_largest, g_task02_pass);
134 #endif
135     return g_task02_pass ? 0 : 1;
136 }
```

4 Task 3 — prawdziwy FreeRTOS heap_4.c

Sekwencja testowa

1. init przez pierwsze `pvPortMalloc(1)` i `vPortFree`; zapisz F_0 ;
2. malloc: 24 B, 40 B, 16 B; zapisz F_1 ;
3. free pierwszego i trzeciego; oczekuj 2 wolnych bloków;
4. free środkowego; oczekuj jednego bloku i $F_{final} = F_0$;
5. malloc większy niż heap; oczekuj NULL i jednego hooka.

Inwarianty — zaznacz po teście

- $\square F_0 > F_1$;
- $\square F_{final} = F_0$;
- $\square \text{minimum-ever} \leq F_1$ i nie rośnie po free;
- $\square$ każdy payload ma adres podzielny przez 8;
- $\square$ OOM zwraca NULL;
- $\square$ końcowa free-list ma dokładnie jeden blok.

```

1 #include <stddef.h>
2 #include <stdint.h>
3
4 #include "FreeRTOS.h"
5 #include "task.h"
6
7 #if defined(HOST_PRINTF)
8 #include <stdio.h>
9 #endif
10
11 uint8_t ucHeap[configTOTAL_HEAP_SIZE] __attribute__((aligned(portBYTE_ALIGNMENT)));
12
13 HeapStats_t g_fragmented_stats;
14 HeapStats_t g_final_stats;
15 volatile size_t g_initial_free;
16 volatile size_t g_after_allocations;
17 volatile size_t g_final_free;
18 volatile size_t g_minimum_ever_free;
19 volatile int g_allocations_aligned;
20 volatile int g_oom_is_null;
21 volatile int g_malloc_failed_hooks;
22 volatile int g_assert_failures;
23 volatile int g_task03_pass;
24
25 void vApplicationMallocFailedHook(void)
26 {
27     g_malloc_failed_hooks++;
28 }
29
30 void heap4_lab_assert(const char *file, int line)
31 {
32     (void)file;
33     (void)line;
34     g_assert_failures++;
35 }
36
37 #if defined(HEAP4_HOST_ADAPTER)
38 void vTaskSuspendAll(void)
39 {
40 }
41
42 BaseType_t xTaskResumeAll(void)
43 {
44     return 0;
45 }
46 #endif
47
48 __attribute__((noinline)) void heap4_fragmented_checkpoint(void)
49 {
50 #if defined(__GNUC__)

```

```

51     __asm__ volatile("" ::: "memory");
52 #endif
53 }
54
55 __attribute__((noinline)) void task03_debug_checkpoint(void)
56 {
57     #if defined(__GNUC__)
58         __asm__ volatile("" ::: "memory");
59     #endif
60 }
61
62 int main(void)
63 {
64     void *probe;
65     void *a;
66     void *b;
67     void *c;
68     void *too_large;
69
70     vPortHeapResetState();
71     probe = pvPortMalloc(1U);
72     if (probe != NULL)
73     {
74         vPortFree(probe);
75     }
76     g_initial_free = xPortGetFreeHeapSize();
77
78     a = pvPortMalloc(24U);
79     b = pvPortMalloc(40U);
80     c = pvPortMalloc(16U);
81     g_after_allocations = xPortGetFreeHeapSize();
82     g_allocations_aligned =
83         a != NULL && b != NULL && c != NULL &&
84         ((uintptr_t)a % portBYTE_ALIGNMENT) == 0U &&
85         ((uintptr_t)b % portBYTE_ALIGNMENT) == 0U &&
86         ((uintptr_t)c % portBYTE_ALIGNMENT) == 0U;
87
88     vPortFree(a);
89     vPortFree(c);
90     vPortGetHeapStats(&g_fragmented_stats);
91     heap4_fragmented_checkpoint();
92
93     vPortFree(b);
94     g_final_free = xPortGetFreeHeapSize();
95     g_minimum_ever_free = xPortGetMinimumEverFreeHeapSize();
96     vPortGetHeapStats(&g_final_stats);
97
98     too_large = pvPortMalloc(configTOTAL_HEAP_SIZE * 2U);
99     g_oom_is_null = too_large == NULL;
100     g_task03_pass =
101         probe != NULL && g_allocations_aligned &&
102         g_initial_free > g_after_allocations &&
103         g_final_free == g_initial_free &&
104         g_minimum_ever_free <= g_after_allocations &&
105         g_fragmented_stats.xNumberOfFreeBlocks == 2U &&
106         g_final_stats.xNumberOfFreeBlocks == 1U &&
107         g_final_stats.xSizeOfLargestFreeBlockInBytes == g_initial_free &&
108         g_oom_is_null && g_malloc_failed_hooks == 1 &&
109         g_assert_failures == 0;
110
111     task03_debug_checkpoint();
112
113 #if defined(HOST_PRINTF)
114     printf("initial=%zu after=%zu fragmented=%zu final=%zu min=%zu "
115           "aligned=%d oom=%d hooks=%d asserts=%d pass=%d\n",
116           g_initial_free, g_after_allocations,
117           g_fragmented_stats.xNumberOfFreeBlocks, g_final_free,
118           g_minimum_ever_free, g_allocations_aligned, g_oom_is_null,
119           g_malloc_failed_hooks, g_assert_failures, g_task03_pass);
120 #endif
121     return g_task03_pass ? 0 : 1;
122 }

```

5 Czytanie upstream heap_4.c

Znajdź w pliku — nie czytaj całego od góry

```
K=/opt/FreeRTOS-Kernel/portable/MemMang/heap_4.c
rg -n 'xHeapStructSize|pvPortMalloc|prvHeapInit' "$K"
rg -n 'prvInsertBlockIntoFreeList|vPortFree' "$K"
rg -n 'xFreeBytesRemaining|xMinimumEver' "$K"
rg -n 'heapADD_WILL_OVERFLOW|heapBLOCK_ALLOCATED' "$K"
```

- xHeapStructSize: dlaczego jest wyrównany oddzielnie od sizeof?
- heapBLOCK_ALLOCATED_BITMASK: który bit rozmiaru oznacza właściciela?
- pvPortMalloc: gdzie są overflow, alignment, first-fit i split?
- vPortFree: jak odzyskuje nagłówkę z payloadu?
- prvInsertBlockIntoFreeList: gdzie są dwa kierunki scalania?
- xMinimumEverFreeBytesRemaining: dlaczego free go nie zwiększa?

Breakpointy

```
b pvPortMalloc
b vPortFree
b prvInsertBlockIntoFreeList
p xStart
p pxEnd
p xFreeBytesRemaining
p xMinimumEverFreeBytesRemaining
x/160bx ucHeap
```

6 RP2350 — deploy i dowód sprzętowy

Polecenie

```
stemctl probe list
stemctl deploy rp2350 freertos heap4 3 \
  --device /dev/bus/usb/BBB/DDD
```

Automatyczny test zatrzymuje się dwa razy

1. heap4_fragmented_checkpoint: 2 wolne bloki;
2. task03_debug_checkpoint: heap odtworzony, OOM obsłużony.

Zapis ucznia

Wielkość	Wartość
initial free	_____
after allocations	_____
minimum ever	_____
final free	_____
sp / fp / pc	_____

Kontrakt

- flash: write + verify; ELF: RISC-V;
- breakpointy sprzętowe, nie data watchpoint;
- aligned=1 oom=1 hooks=1 asserts=0 pass=1;
- sp & 15 == 0;
- rp2350-hardware-ok task=task03 heap_4=upstream.

Operacje są rozdzielone: debug ładuje do SRAM; deploy zapisuje trwały obraz do SPI flash.

7 Podsumowanie i oddanie

Odpowiedz jednym zdaniem

1. Dlaczego heap_4 scala, a nie tylko odkłada blok na listę?

2. Dlaczego „wolne razem” nie oznacza „można przydzielić jeden duży blok”?

3. Gdzie leży metadata bloku przydzielonego aplikacji?

4. Co mierzy minimum-ever-free, a czego nie mierzy?

Oddaj

- ☐ wyniki 3 tasków na AMD64;
- ☐ Hazard3: trzy kody wyjścia 0;
- ☐ zrzut Task 2: przed i po scaleniu;
- ☐ zrzut Task 3: source + listing + Memory + Stack;
- ☐ wynik RP2350 **rp2350-hardware-ok**;
- ☐ wypełnione inwarianty i cztery odpowiedzi.

Dalej: taski FreeRTOS, scheduler, osobne stosy, 1 kHz tick; następnie cienki C++ header wrapper i użycie w projekcie hoveboard.