

Most C++ do heapu, modele i HeapStats

suma wolnych bajtów nie jest największym blokiem

Karta	K05 / 16	Czas	30 minut
Platforma	Hazard3 / RV32I	Język	freestanding C++17
Allocator	prawdziwy heap_4.c	Kernel	V11.3.0, bez zmian
Wersja	v00.01	UUID	c456b505-...
		karty	

Trzy pytania, jeden eksperyment

1. Która polityka heap_1...heap_5 pasuje do topologii pamięci?
 2. Czy każdy wariant new/delete trafia do jednego heapu FreeRTOS?
 3. Czy bieżące free, największy blok i minimum-ever dowodzą tego samego?

Inwariant K05. Każdy żywy blok ma właściciela, a każda decyzja o pojemności używa co najmniej pary: suma wolnych bajtów i największy wolny blok. Minimum-ever jest historią i nie rośnie po zwolnieniu pamięci.

Plan 30 minut

Czas	Tryb	Dowód ucznia
0–5	wybór modelu	heap dla trzech scenariuszy
5–9	most C++	komplet scalar/array, sized/unsized
9–14	predykcja	układ A–B–C i dwie dziury po free
14–21	fragmentacja	largest < request < total, wynik NULL
21–26	koalescencja	jedno free, identyczny request działa
26–30	historia	current wraca, minimum-ever zostaje

Najpierw wybierz

Tylko startup, brak free: _____ Jedna arena + delete: _____ Dwa rozłączne banki RAM: _____

1 Pięć modeli, ale jeden w obrazie programu

Model	Free	Koalescencja	Kontrakt / typowe użycie
heap_1	nie	nie dotyczy	monotoniczna arena; alokuj raz i nie zwalniasz
heap_2	tak	nie	odzyskuje bloki, lecz sąsiednie dziury pozostają osobne
heap_3	libc	zależy od libc	deleguje do <code>malloc/free</code> ; wymaga heapu linkera
heap_4	tak	sąsiednich	jedna arena FreeRTOS; wybór tej serii na Hazard3
heap_5	tak	w regionach	wiele regionów zdefiniowanych przed pierwszą alokacją

Nie łączymy implementacji. Do obrazu linkujemy jeden plik, który definiuje `pvPortMalloc` i `vPortFree`. Most C++ zmienia składnię, nie politykę wybranego allocatora.

Uzasadnij trzy wdrożenia

1. Wszystkie obiekty powstają przed schedulerem i nigdy nie są usuwane: model _____, ponieważ _____.
2. Zadania i kolejki powstają/nikną w jednej arenie RAM: model _____, ponieważ _____.
3. Dwa rozłączne banki RAM mają zasilać jeden allocator: model _____, ponieważ _____.

2 Kompletny most C++

```
void* operator new(size_t n);          void* operator new[](size_t n);
void operator delete(void* p) noexcept;
void operator delete[](void* p) noexcept;
void operator delete(void* p, size_t) noexcept;
void operator delete[](void* p, size_t) noexcept;
```

`new/new[] -> pvPortMalloc` `delete/delete[] -> vPortFree`

W profilu bez wyjątków zwykle `new` jest **fail-fast**. Próba kontrolowanej porażki używa jawnie `pvPortMalloc` i sprawdza `nullptr`; nie zmienia kontraktu ordinary `new`.

Dlaczego `sized delete` też musi istnieć?

Kompilator może wybrać wariant z drugim argumentem rozmiaru. Brak definicji oznacza niekompletny most albo zależność od niedostępnego runtime. W ELF zaznacz sześć znalezionych symboli:

3 Eksperyment: total wystarcza, blok nie

```
baseline: [ FREE ..... ]
allocate: [ A 2K ][ B 4K ][ C 2K ][ FREE ..... ]
free A,C: [ FREE ][ B LIVE ][ FREE + trailing free ..... ]
```

Po zwolnieniu A i C blok B nadal rozdziela wolne obszary. `heap_4` scala C z końcowym free, ale nie może scalić przez żywy B.

Wielkość	Predykcja	Odczyt Hazard3
<code>available_bytes</code>	_____	_____
<code>largest_free_block</code>	_____	_____
<code>free_blocks</code>	_____	_____
<code>request = largest + 1</code>	_____	_____
wynik pierwszej próby	_____	_____

Relacja, nie przypadkowe liczby

largest < request < available i mimo tego result == nullptr

Wyjaśnienie: _____

Zwolnij B i ponów identyczne żądanie

```
free B: [ ONE COALESCED FREE BLOCK ..... ]
```

- Wtedy wymagamy:
- `free_blocks == _____`;
 - `largest == available == baseline: _____`;
 - ponowienie *tego samego* request: _____;
 - adres retry leży w `ucHeap`: _____.

Nagłówek i wyrównanie też kosztują

Statystyka największego wolnego bloku opisuje blok allocatora. Żądany payload potrzebuje jeszcze wyrównanego nagłówka. Dlatego nie zamieniaj pola `largest` w bezwarunkową gwarancję payloadu tej samej wielkości.

4 Siedem checkpointów i historia minimum-ever

```
make check
riscv64-unknown-elf-gdb build/task01_heap_models/prog.elf
b heap_models_debug_checkpoint
```

STOP	Stan	Obowiązkowy dowód
1	baseline	jedno free; zapisz current i minimum-ever
2	A+B+C żywe	trzy adresy; free spada; payload zachowany
3	A/C zwolnione	dwa free; total > largest
4	kontrolowany fail	request między largest i total; alloc count bez zmiany
5	B zwolnione	jeden blok; largest=current=baseline
6	retry żywe	ten sam request działa; min-ever osiąga nowe minimum
7	retry zwolnione	current=baseline, minimum-ever nadal niższe

Minimalny zestaw GDB

```
p g_baseline_stats
p g_after_alloc_stats
p g_fragmented_stats
p g_after_failure_stats
p g_coalesced_stats
p g_after_retry_stats
p g_final_stats
p g_cpp_allocation_addresses
p g_cpp_deallocation_addresses
p g_peak_bytes_used
p g_heap_models_pass
```

Current kontra watermark

Warunek końcowy	Odczyt
final.available ==	_____
baseline.available	_____
final.minimum_ever <	_____
final.available	_____
peak = baseline -	_____
minimum_ever	_____
alloc/free/live == 3/3/0 dla	_____
C++	_____
kolejność zwolnień adresów =	_____
A,C,B	_____
pass == 1	_____

Wyjście

1. Dlaczego 12 KB total free nie obiecuje jednego bloku 10 KB?
2. Co zmieniło zwolnienie B, skoro suma free też tylko wzrosła?
3. Dlaczego minimum-ever nie wróciło wraz z current free?
4. Dlaczego heap_3 nie jest samowystarczalnym wyborem freestanding?

Następna karta **K06**: nie-wirtualny MemoryResource, HeapResource, arena statyczna i typowany FreeRtosAllocator<T>.