

DynamicTask, StaticTask i lifetime

delete logiczny teraz, reclaim dynamiczny przez idle

Karta	K08 / 16	Czas	30 minut
Platforma	Hazard3 / RV32I	Język	freestanding C++17
Stack	256 słów	Polityki	heap kontra caller storage
Wersja	v00.01	UUID	d9e45417-...
		karty	

To samo zachowanie, dwa źródła TCB i stacka

```
xTaskCreate -> TCB + stack from heap_4
xTaskCreateStatic -> caller StaticTask_t + StackType_t[]
```

Oba workery liczą 1..1000 i kończą wynikiem 500500. Różni je miejsce, właściciel oraz moment odzyskania pamięci.

Kontrakt K08. Null handle i stan completed oznaczają zakończenie wrappera, lecz nie dowodzą jeszcze zwolnienia dynamicznego TCB/stacka. To robi idle. Statyczne buforry nigdy nie stają się własnością kernela.

Plan 30 minut

Czas	Tryb	Dowód
0–5	jednostki	256 słów, liczba bajtów i lokalizacje
5–10	dynamic	heap delta, TCB i stack w ucHeap
10–15	static	caller TCB/stack w .bss, heap delta 0
15–20	wykonanie	oba wyniki 500500, completed/null
20–25	delete	dynamic heap nadal zmniejszony przed idle
25–30	cleanup	verifier blokuje; idle odzyskuje dokładny delta

Predykcja

256 słów RV32I × _____ B/słowo = _____ B. Czy vTaskDelete(nullptr) samo natychmiast wywoła dwa free dynamicznego taska? _____

1 Słowa, bajty i właściciel storage

Właściwość	DynamicTask	StaticTask
API	<code>xTaskCreate</code>	<code>xTaskCreateStatic</code>
TCB	przydziela kernel z heapu	caller przekazuje <code>StaticTask_t</code>
stack	kernel: depth w słowach	caller: <code>StackType_t[N]</code>
heap delta przy create	_____	_____
adres TCB	_____	_____
adres stack base	_____	_____
właściciel po delete	idle odzyskuje dynamiczny	caller nadal posiada statyczny

Oblicz przed kompilacją

$$\begin{aligned} \text{stack bytes} &= \text{stack words} \times \text{sizeof}(\text{StackType_t}) \\ &= 256 \times \text{_____} = \text{_____} \text{ B} \end{aligned}$$

Dynamiczny koszt będzie większy od samych stack bytes, bo obejmuje także TCB oraz narzut bloków allocator. Predykcja kosztu: _____ B. Pomiar: _____ B. Różnica: _____ B.

2 Caller-supplied storage musi naprawdę być osobne

```
StaticTaskStorage<256> worker_storage; // .bss
StaticWorker worker{worker_storage};

xTaskCreateStatic(entry, "static", 256, &worker, priority,
                 worker_storage.stack, &worker_storage.tcb);
```

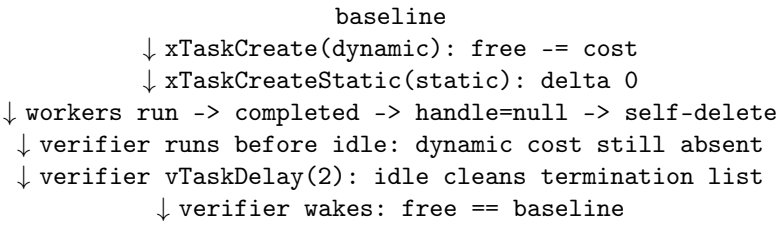
W ELF symbol `g_static_worker_storage` ma typ sekcji _____. Odczyty muszą spełnić:

$$\begin{aligned} \text{observed TCB} &== \text{\&worker_storage.tcb} \\ \text{observed stack base} &== \text{worker_storage.stack} \end{aligned}$$

Dlaczego storage jest oddzielone od handle?

Handle może zostać wyzerowany po completion. Bufor statyczny nadal istnieje i ma własny czas życia. Nie wolno go jednak ponownie użyć, dopóki usunięcie taska nie jest ustalone; sam „request stop” nie wystarcza.

3 Oś czasu delete i idle cleanup



Pomiar	Predykcja	Odczyt
baseline free		
free po dynamic start	mniej	
free po static start	bez zmiany	
free po static verifier create	bez zmiany	
free przed idle cleanup	nadal zmniejszone	
free po idle cleanup	baseline	
recovered by idle	dynamic cost	

Co robi idle?

Self-delete nie może zwolnić własnego stosu, gdy kod nadal na nim wykonuje instrukcje. Kernel odkłada dynamiczny TCB na listę terminacji. Idle działa już na innym stosie i może bezpiecznie wywołać cleanup/free. Dla statycznego taska rozpoznaje caller-owned storage i go nie zwalnia.

Dwa różne zdania o zakończeniu

1. **Logiczne:** task nie jest schedulowany, wrapper ma completed/null.

2. **Pamięciowe:** idle odzyskał dynamiczny TCB/stack; current free wróciło o dokładny koszt.

Czy verifier o prioritycie 1 może sprawdzić reclaim bez blokowania się? _____.

Dlaczego? _____.

Granica destruktorów aplikacyjnych

Wymuszone usunięcie innego taska nie odwija automatycznie jego ramek C++. Workery K08 kończą member body normalnie, a dopiero trampoline self-delete.

4 Hazard3/GDB i zaliczenie

```
make check
riscv64-unknown-elf-gdb build/task01_static_task/prog.elf
b task_storage_debug_checkpoint
```

```
p g_heap_baseline
p g_heap_after_dynamic_start
p g_heap_after_static_start
p g_heap_before_idle_cleanup
p g_heap_after_idle_cleanup
p/x g_dynamic_tcb
p/x g_dynamic_stack_low
p/x g_static_tcb
p/x g_static_tcb_storage
p/x g_static_stack_low
p/x g_static_stack_storage
p g_static_stack_words
p g_static_stack_bytes
p g_task_storage_pass
```

STOP	Stan	Dowód
1	baseline	current free zapisane
2	dynamic created	heap spada; handle/TCB i stack w ucHeap
3–4	static created	caller addresses; heap bez zmiany
5–8	workers	oba wyniki 500500; completion/self-delete
9	before idle	handles null, lecz dynamic cost nadal zajęty
10	after idle	odzyskany dokładny cost; free=baseline
11	final	static storage zachowane; pass=1

Zaliczenie

- ☐ przeliczam 256 słów na 1024 B;
- ☐ pokazuję dodatni dynamic cost i zerowy static delta;
- ☐ klasyfikuję dynamic addresses w heapie i static w .bss;
- ☐ rozróżniam completion/null od idle memory reclaim;
- ☐ pokazuję before-idle, after-idle i exact recovered cost;
- ☐ nie przypisuję kernelowi własności static storage.

Wyjście

Dlaczego self-delete nie może od razu zwolnić własnego stacka?

Kiedy wolno ponownie użyć statycznego TCB/stacka?

Następna karta **K09**: statyczna fasada schedulera, `this_task`, critical section i guard zawieszenia z poprawnym restore.