

TaskHandle_t, jawny start() i trampoline

jeden stabilny obiekt przez granicę C → C++

Karta	K07 / 16	Czas	30 minut
Platforma	Hazard3 / RV32I	Język	freestanding C++17
Storage	dynamiczny TCB + stack	Dispatch	static CRTP trampoline
Wersja	v00.01	UUID	9bffead0-...
		karty	

Trzy adresy, trzy role

C++ object ≠ TaskHandle_t/TCB ≠ task stack storage

Konstruktor zapisuje tylko konfigurację. Dopiero `start()` woła `xTaskCreate`, a kernel zapamiętuje adres kompletnego obiektu jako `void*`. Statyczny trampoline odzyskuje typ i uruchamia prywatne `run()` bez `vtable`.

Inwariant K07. Adres obiektu od startu do końca taska jest stały. Wrapper nie jest kopiowalny ani przenoszalny. Handle identyfikuje TCB, a nie obiekt aplikacyjny. Destruktor nie wykonuje wymuszonego delete taska.

Plan 30 minut

Czas	Tryb	Dowód
0–5	adresy	object, TCB/handle i stack to różne rzeczy
5–10	explicit start	konstruktor heap delta 0; start tworzy kernel storage
10–16	C API	sześć argumentów <code>xTaskCreate</code>
16–21	trampoline	raw context = object = this in run
21–26	lifecycle	ready, running, completed, null handle, self-delete
26–30	GDB/exit	local na task stack; bez move/vtable/destructor-delete

Predykcja

Który adres znajdzie się w `TaskHandle_t`: obiekt czy TCB? _____ Czy konstruktor może uruchomić task? _____

1 Dlaczego start nie należy do konstruktora

W czasie konstruktora bazowego część derived nie jest jeszcze kompletna. Scheduler mógłby wywołać `run()` natychmiast po `create`. Dlatego:

```
CounterTask worker;           // tylko konfiguracja, heap bez zmian
bool ok = worker.start();      // dopiero teraz xTaskCreate
```

Stan wrappera	Znaczenie
constructed	kompletny obiekt, brak taska kernela
starting	wejście do jawnej operacji start
ready	context i konfiguracja przekazywane do kernela
running	trampoline odzyskał typed object
completed	member body wrócił, handle wyzerowany
start_failed	create nie utworzył taska; handle null

Drugie `start()` musi zwrócić _____, wykonać _____ nowych `xTaskCreate` i pozostawić handle _____.

2 Rozpisz sześć argumentów xTaskCreate

```
xTaskCreate(&Task::trampoline,
            name_, stack_words_,
            static_cast<Derived*>(this),
            priority_, &handle_);
```

Argument	Co niesie / jednostka
&Task::trampoline	_____
name_	_____
stack_words_	_____
typed context: Derived* z this	_____
priority_	_____
&handle_	_____

Pułapka jednostki: stack depth jest liczbą `StackType_t`, nie domyślnie bajtów. Dla RV32I 256 słów to _____ B.

3 Trampoline odzyskuje typ, nie tworzy obiektu

```
static void trampoline(void* context) {
    Derived* derived = static_cast<Derived*>(context);
    Task& base = *static_cast<Task*>(derived);
    base.state_ = TaskState::running;
    derived->run();
    base.state_ = TaskState::completed;
    base.handle_ = nullptr;
    vTaskDelete(nullptr);
}
```

Nie ma `new`, lookupu ani `vtable`. Context jest adresem istniejącego, kompletnego obiektu. Prywatny `run()` jest dostępny dzięki jawnej relacji `friend` z konkretną instancją CRTP.

Trzy domeny adresów

Tożsamość obiektu C++	Tożsamość TCB		Task stack
before start: _____	handle	after _____	start: low: _____
after start: _____	current handle: _____		high: _____
trampoline _____	context: status TCB: _____		local: _____
this in run: _____			watermark: _____
completed _____	object: _____		

object identities equal handle == current == TCB low <= local <= high

Maszyna stanów wrappera

constructed -> starting -> ready -> running -> completed
constructed -> starting -> ready -> start_failed (gdy create fail)

Dlaczego stan `ready` jest ustawiany przed wejściem do `xTaskCreate`? Przy aktywnym schedulerze nowy task może wykonać się, a nawet zakończyć, zanim wywołanie `create` zwróci do startera. Kod po powrocie nie może nadpisać `completed` ponownie stanem `ready`.

Predykcja duplikatu

Drugie `start()` na tym samym obiekcie powinno:

- zwrócić _____;
- wywołać `xTaskCreate` jeszcze raz? _____;
- zmienić `handle`? _____;
- pozostawić stan: _____.

Granica destruktora

Destruktor obiektu na innym stosie nie może bezpiecznie „sprzątnąć” dowolnego żywego taska: wymuszone `vTaskDelete(handle)` nie odwija jego ramek C++. K07 kończy body normalnie i dopiero trampoline usuwa bieżący task.

4 Hazard3/GDB i kontrakt zaliczenia

```
make check
riscv64-unknown-elf-gdb build/task01_task_wrapper/prog.elf
b explicit_task_debug_checkpoint
```

```
p g_checkpoint_trace
p/x g_object_before_start
p/x g_trampoline_context
p/x g_run_this
p/x g_handle_after_start
p/x g_tcb_address
p/x g_stack_low
p/x g_stack_high
p/x g_stack_local
p g_counter_result
p g_task_wrapper_pass
```

STOP	Stan	Obowiązkowy dowód
1	constructed	heap konstrukcji bez zmian, handle null
2–3	start	starting/ready, context wskazuje pełny obiekt
4	create returned	handle wskazuje TCB w ucHeap
5	before scheduler	object address nadal ten sam
6–7	trampoline	context odzyskany, state running
8	private run	handle/current/TCB równe; local w stack; wynik 500500
10	completion	state completed; handle null; self-delete
11–12	verifier	wszystkie relacje i pass=1

Zaliczenie

- ☐ odróżniam object, handle/TCB i task stack;
- ☐ pokazuję zerowy heap delta konstruktora i jawny create;
- ☐ opisuję sześć argumentów `xTaskCreate`;
- ☐ śledzę ten sam context przez trampoline do `run()`;
- ☐ pokazuję local w stack bounds oraz wynik 500500;
- ☐ uzasadniam brak copy/move, vtable i delete w destruktorze.

Wyjście

Dlaczego handle nie może być użyty jako adres obiektu aplikacyjnego?

Co chroni jawny `start()`?

Następna karta K08: dynamiczny i statyczny TCB/stack, słowa kontra bajty, jawne zakończenie i czas odzyskania pamięci.