Thread-Specific Storage

Neuwirth István, 2009

Thread-Specific Storage

- Más néven: Thread-Local Storage
  - (TLS)
- Konkurens tervminta
  - Többszálú programozással kapcsolatban fellépő problémákra

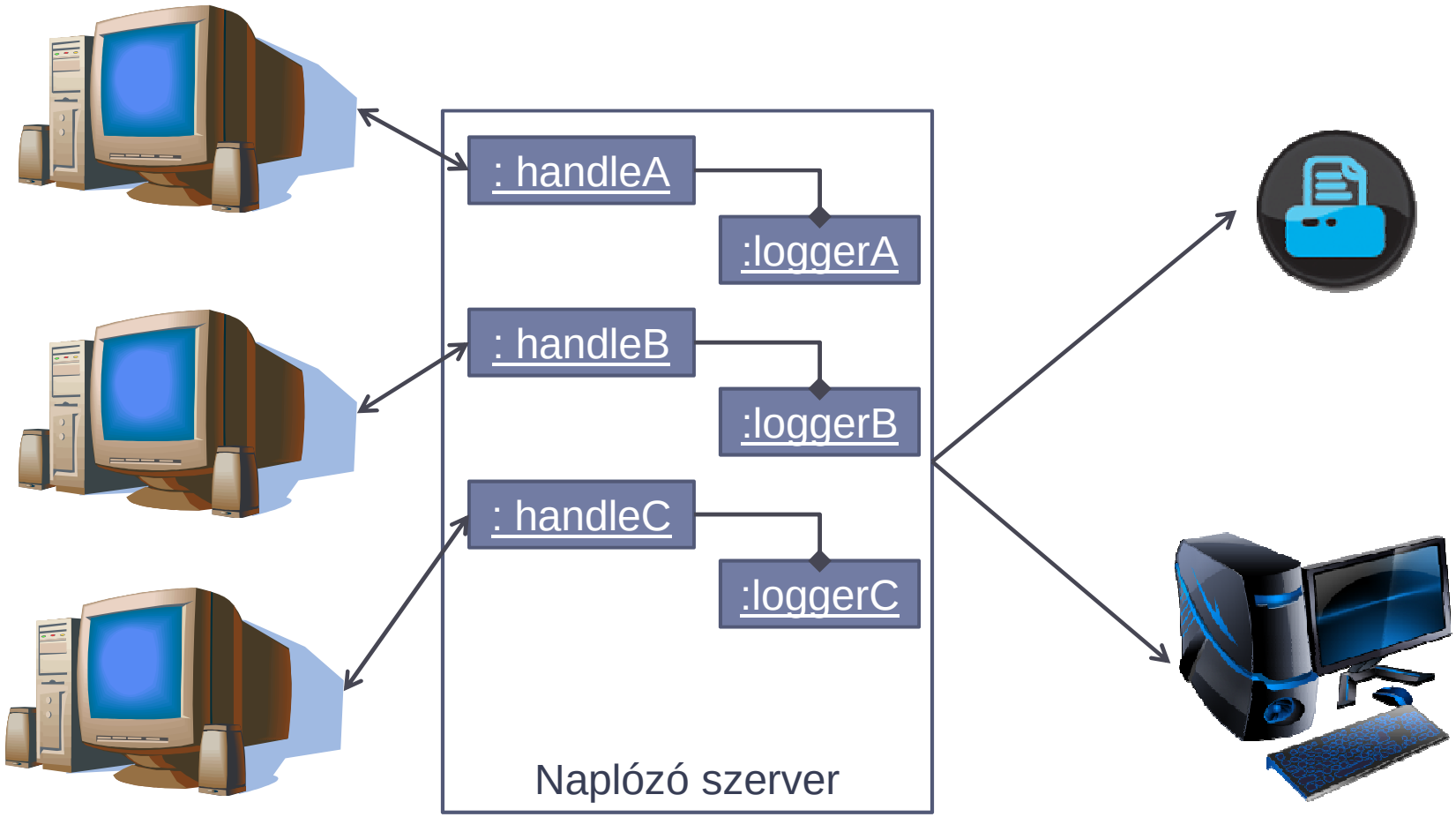
A minta célja

- Lehetővé teszi, hogy több szál használjon egy „logikailag globális” hozzáférési pontot, amelyen keresztül egy szál szempontjából lokális objektumot nyerhet ki, az elérések többletköltsége nélkül.

Egy példa

- Adott egy több szálon futó hálózati naplózó szerver, amely kiszolgálja a klienseket, az állapotaikat naplózza
  - Kapcsolatonként indított szálak vannak
  - Minden szál a hozzá rendelt TCP socket-en keresztül olvassa a naplórekordokat, majd kiírja valamilyen output egységre, legyen az nyomtató, monitor vagy háttértár.
  - Az alacsonyszintű IO műveletekről az operációs rendszerek többségében (Unix és Windows rendszereken is) egy globális hozzáférési ponton keresztül szerezhetünk információt.
  - Ez a pont az *errno*. (Tipikusan egy `int`.)

Egy példa

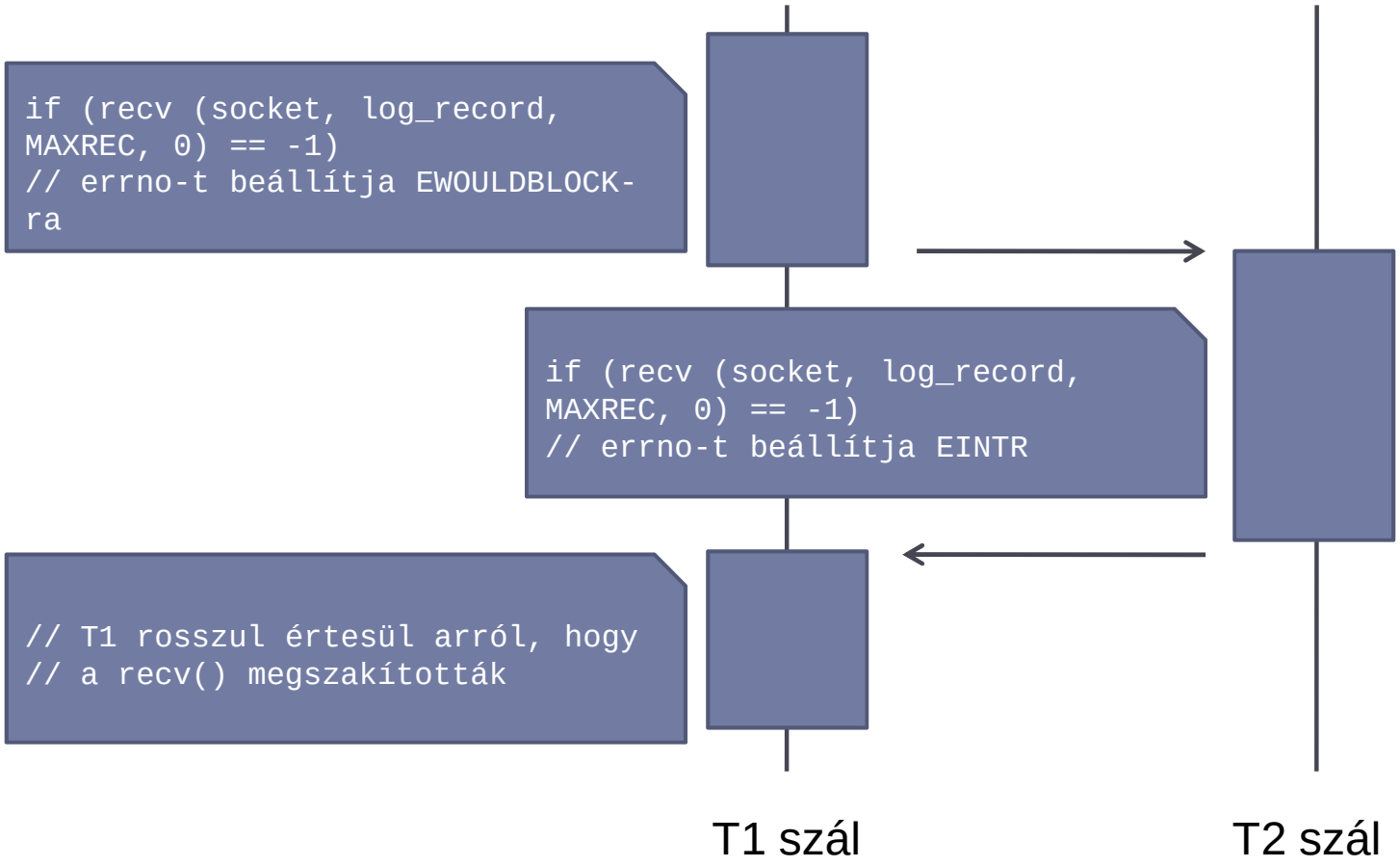

: handleA
:loggerA
: handleB
:loggerB
: handleC
:loggerC
Naplózó szerver

Egy példa

```
// Egy globális <errno> folyamatonként.
extern int errno;

void *logger (HANDLE socket) {
   // Olvassuk a naplórekordokat a kapcsolat lezártáig
   for (;;) {
      char log_record[MAXREC];
      if (recv (socket, log_record, MAXREC, 0) == -1) {
         // Ellenőrizzük, miért nem sikerült a <recv>.
         if (errno == EWOULDBLOCK)
            sleep (1); // Később újra megpróbáljuk.
         else // Megjelenítjük a hiba eredményét.
            cerr << "recv failed, errno=" << errno;
      } else // Normál esetben...
   }
}
```

Egy példa


if (recv (socket, log_record,
MAXREC, 0) == -1)
// errno-t beállítja EWOULDBLOCK-
ra
if (recv (socket, log_record,
MAXREC, 0) == -1)
// errno-t beállítja EINTR
// T1 rosszul értesül arról, hogy
// a recv() megszakították
T1 szál
T2 szál

Megoldás?

- Valamilyen zárolás
  - Versenyhelyzetek
  - Holtponthoz vezethet
  - Gyakori ellenőrzések jelentősen csökkentik a teljesítményt, a zárolási „overhead” miatt akár egy egyszálon futó program is bírhat olyan teljesítménnyel, mint egy többszálú megfelelője.
- Amire szükség van, az valamilyen mechanizmus, amely minden szálnak ad másolatokat (fizikailag lokálisak) a „logikailag globális” objektumokról, mint amilyen az *errno*.

Szerkezet

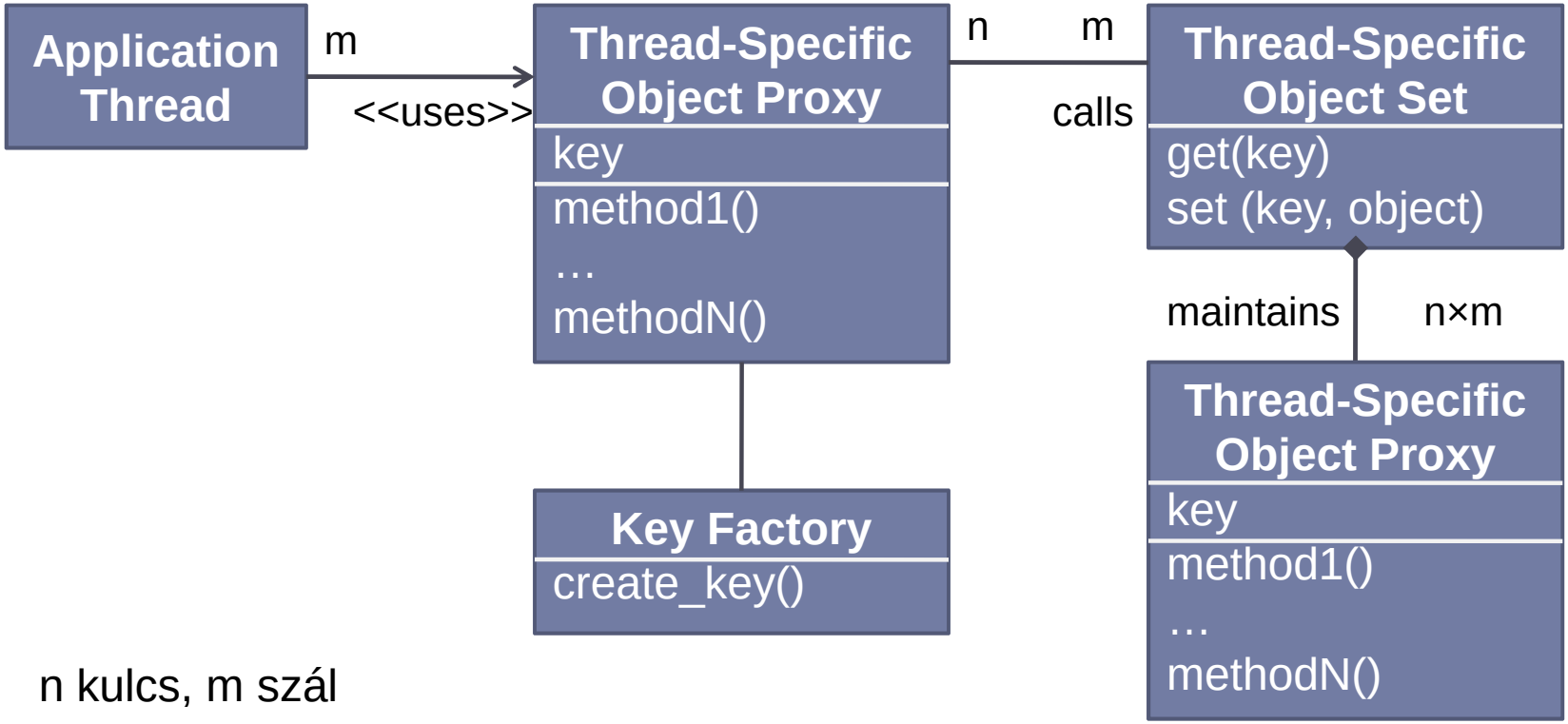
- Hat féle osztály alkotja a mintát
  - Thread-Specific Object
    - Egy objektumpéldány, amelyet csak egy szál tud elérni (például errno int-ként, minden szálnak egy-egy példánya van belőle).
  - Key
    - Azonosítja a szálspecifikus objektumot
  - Key Factory
    - Kulcsokat generál

Például, egy többszálú operációs rendszer implementálja az *errno*-t, egy globálisan egyedi kulcsot létrehozva. Minden egyes szál ezt a kulcsot használja, hogy elérje a saját *errno* példányát.

Szerkezet

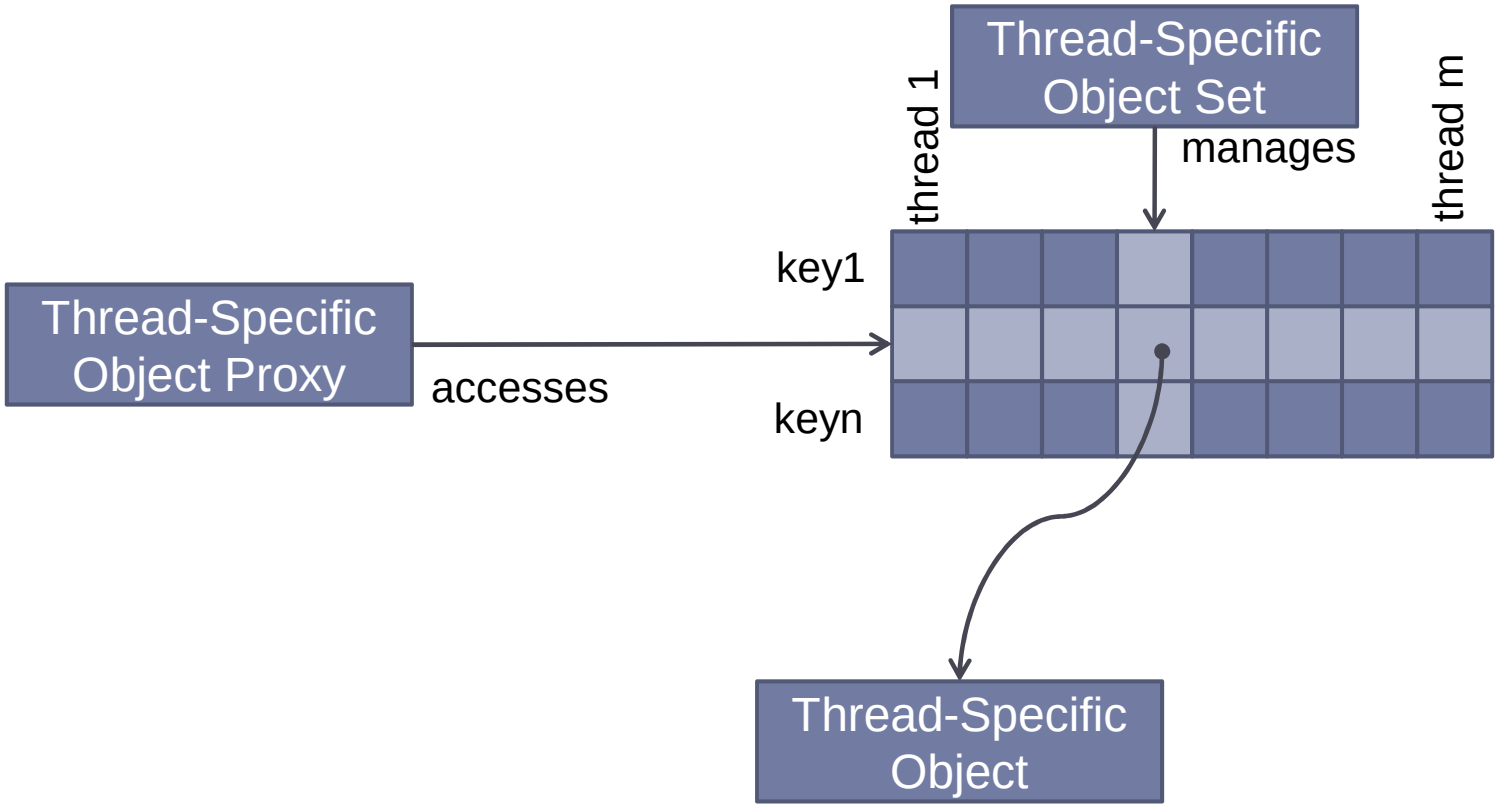
- Thread-Specific Object Set
  - Egy gyűjteményét tartalmazza a különálló szálakhoz rendelt szálspecifikus objektumoknak. Minden egyes szál rendelkezik egy ilyen halmazzal. Metódusai: get()/set().
- Thread-Specific Object Proxy
  - Minden proxy tárol egy kulcsot, amely egyedien azonosít egy szálspecifikus objektumot. Épp ezért, egy ilyen objektum van kulcsonként és szálanként.
- Application Threads
  - Használják a proxy-kat, hogy elérjék a szálspecifikus objektumokat.
  - Mint a példában a logger() függvényt futtató szál.

Szerkezet



n kulcs, m szál

Szerkezet

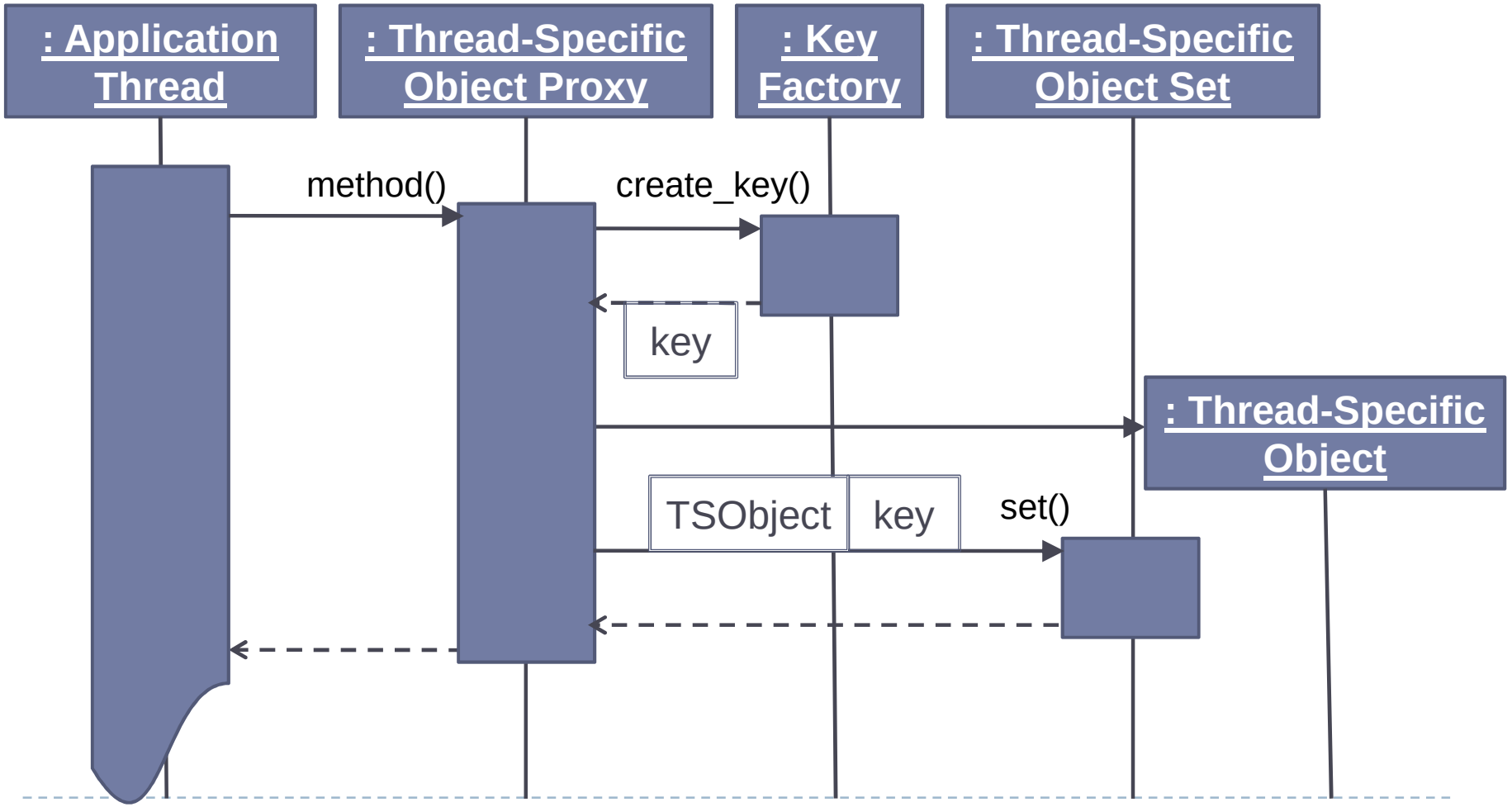

Thread-Specific Object Set
manages
thread 1
thread m
key1
keyn
Thread-Specific Object Proxy
accesses
Thread-Specific Object

Dinamikus viselkedés

- Szálspecifikus objektum létrehozása:
  - Egy alkalmazásszál meghív egy proxy metódust a proxy interfészén.
  - Ha a proxy-nak még nincs társított kulcsa, akkor létrehoz egyet, amit tárol.
  - A proxy létrehoz egy új objektumot dinamikusan. Aztán a halmaz `set()` műveletével társítja a kulcshoz.
  - A következőkben látottak szerint lefut az alkalmazásszál által meghívott metódus.

Dinamikus viselkedés

- Szálspecifikus objektum létrehozása:

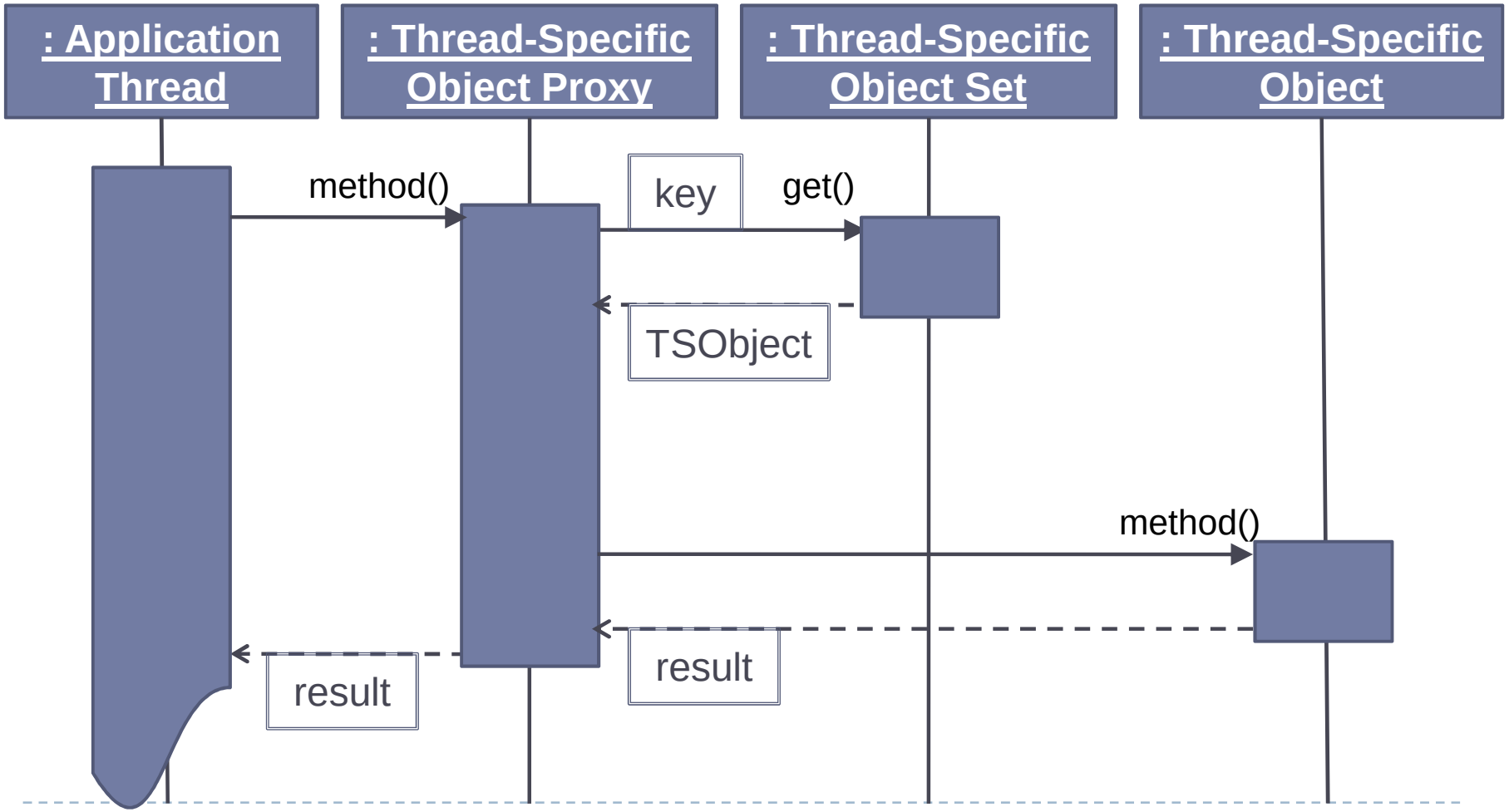


Dinamikus viselkedés

- Létező szálspecifikus objektum elérése:
  - Egy alkalmazásszál meghív metódust a proxy-n.
  - A proxy átadja a kulcsát az alkalmazásszál halmazának `get()` műveletéhez. Ez a függvény visszaad egy mutatót a szóbanforgó szálspecifikus objektumra.
  - A proxy használja ezt a mutatót, hogy meghívja az eredeti metódust az objektumon.

Dinamikus viselkedés

- Létező szálspecifikus objektum elérése:



Implementáció

- Részek az implementációból (POSIX Pthreads könyvtár)

```
int pthread_key_create (pthread_key_t *key, void
(*thread_exit_hook)(void *)) {
   if (total_keys_ >= _POSIX_THREAD_KEYS_MAX) {
      // Use our internal <errno> macro.
      INTERNAL_ERRNO = ENOMEM;
      return -1;
   }
   thread_exit_hook_[total_keys_] = thread_exit_hook;
   *key = total_keys_++;
   return 0;
}
```

Implementáció

- Részek az implementációból (POSIX Pthreads könyvtár)

```
int pthread_setspecific (pthread_key_t key, void *value) {
   if (key < 0 || key >= total_keys) {
      // Use our internal <errno> macro.
      INTERNAL_ERRNO = EINVAL;
      return -1;
   }
   // Store value into appropriate slot in the thread-
   // specific object set.
   pthread_self ()->object_set_[key] = value;
   return 0;
}
```

Implementáció

- Részek az implementációból (POSIX Pthreads könyvtár)

```
int pthread_getspecific (pthread_key_t key, void **value){
   if (key < 0 || key >= total_keys) {
      // Use our internal <errno> macro.
      INTERNAL_ERRNO = EINVAL;
      return -1;
   }
   *value = pthread_self ()->object_set_[key];
   return 0;
}
```

Ismert felhasználások

- Operációs rendszerek (Win32, Solaris, …), az *errno* mechanizmus támogatására.
  - Win32-es API-ban, a szálak rendelkezhetnek ablakkal, amelyeknek van egy-egy message queue-ja. Az, ahogy a hívásokkal leszedjük a sorból a következő elemet, szintén TLS-sel valósul meg.
- OpenGL
  - glVertex() függvény, állapotváltozók befoglalt globális változóként
- Telefonszolgáltatások (valós életbeli példa)

Előnyök

- Négy fő előnye van a Thread-Specific Storage minta használatának:
  - Hatékonyság – a szálspecifikus adat elérése implementálható zárolás nélkül.
  - Újrahasználhatóság – Wrapper Facade minta segítségével.
  - Könnyű használhatóság – csomagolóosztály mögé rejtve eltűnik a forráskódszintű absztrakció (makrók, egyebek).
  - Hordozhatóság – A TLS a legtöbb többszálú operációs rendszer alatt elérhető. A platformfüggő műveletek egy egységes hordozható interfész mögé elrejthetők.

Hátrányok

- Sok alkalmazás nem igényel több szálat a futásához.
  - Használjunk csak abból az egy szálból elérhető adatot.
- Nehezíti a rendszer megértését.
- Megkövetel implementációs opciókat, mint például a paraméteres típusok. Más esetben kevésbé elegánsan és hatékonyan valósítható meg.

Felhasználás különböző nyelvekben

- 1.2-es JDK-tól kezdve a Java támogatja a TLS mintát a java.lang.ThreadLocal osztályon keresztül.
  - Egy példánya ennek az osztálynak egy szálspecifikus objektum proxy.
    - ```
      ThreadLocal<Integer> local = new
      ThreadLocal<Integer>()
      {
         @Override protected Integer initialValue()
         {
            return 1;
         }
      };
      ```
    - ```
      local.set( local.get()+1 );
      ```

Felhasználás különböző nyelvekben

- Visual C++
  - Külön kulcsszóval támogatja:
    - `__declspec(thread) int number;`
- C#
  - Attribútum alkalmazásával:
    - ```
      class FooBar
      {
          [ThreadStatic]
          static int foo;
      }
      ```


      ```

Forrás

- Schmidt, Douglas C.; Michael Stal, Hans Rohnert, Frank Buschmann (2000). *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects.* John Wiley & Sons. ISBN 0-471-60695-2.
- http://en.wikipedia.org/wiki/Thread-Specific_Storage

Köszönöm a figyelmet!