The Lifetime of Objects

What is object?
Object models ***abstract idea*** as a ***state machine***.

- class data member defines the state space
- class member function defines possible state transition

Object assigns the byte stream in memory with a physical meaning :

- for cpu, there is no object concept, what it sees is just a byte stream of instruction and data
- for compiler, its job is to convert objects into sequence instructions
- for developer, object is a design programming paradigm

Object should go through the following stages :

- object allocation / deallocation (request a predetermined size from physical memory)
- object creation / destruction (start and end of lifetime, *i.e. constructor and destructor in OOP*)
- object interaction with outside world (during object lifetime)

Some codes are important to developer only, while meaningless (no-op) for cpu, such as :

- default constructor or destructor doing nothing
- type casting such as `std::move` and `std::forward`
- type manipulation in template class / traits
- preprocessors, macro, compile time operation, meta template programming ... etc

Object attributes
Object is described by the following attributes :

- physical storage in virtual address space (stack vs heap vs BSS segment etc) ⇐ about allocation
- object size (calculated during compile time, for reserving space in memory) ⇐ about allocation
- object lifetime (time between creation and destruction, valid time of object) ⇐ about creation
- object type (type casting change the interpretation of byte stream) ⇐ about creation
- object state (value of each member) ⇐ about interaction

Object size
Compiler calculates object size by :

- adding all data member together
- including virtual function table
- excluding member function
- excluding static members (as they are stored in BSS segment)
- ► compiler needs to know complete type if raw object is instantiated
- ► compiler needs to know forward declaration (incomplete type) only if reference or pointer is instantiated
- ► ***static variables*** are stored in BSS segment, lives from first encounter, until program terminated

	static variable	automatic variable	dynamic variable
physical memory	BSS segment	stack	heap
allocation	on 1st encounter	on construction	on `new` or `malloc` operator
deallocation	on termination	on destruction	on `delete` or `free` operator
lifetime	program lifetime	scoped	customized by programmer

Object lifetime

- stack memory
  by function call, push local variable to call stack, implicit call constructor
  by function return, pop local variable from call stack, implicit call destructor
- heap memory
  allocate with `malloc` + explicit call constructor = `new`
  deallocate with `free` + explicit call destructor = `delete`
- BSS segment
  static variable first encounter + implicit call constructor
  static variable destructed on program termination

Passing an object

- reference (no real copy)
- move / copy / emplace (i.e. construct inplace)

Physical storage of object

3 essential memory concepts :

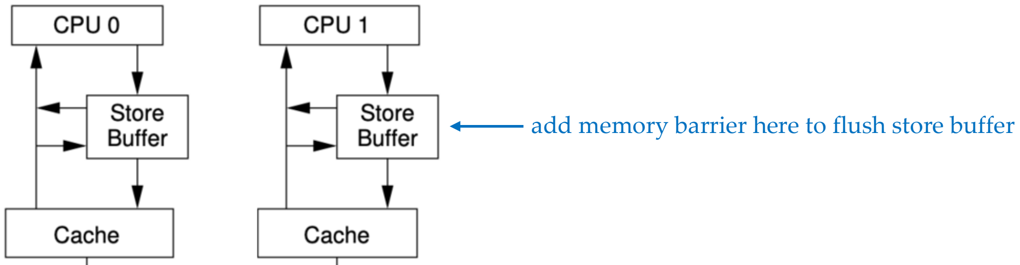
- physical memory *which includes primary and secondary storage*
- virtual address space *which includes stack, heap, free store and BSS segment*
- cache hierarchy inside *CPU* *which includes swap pages*

Some remarks :

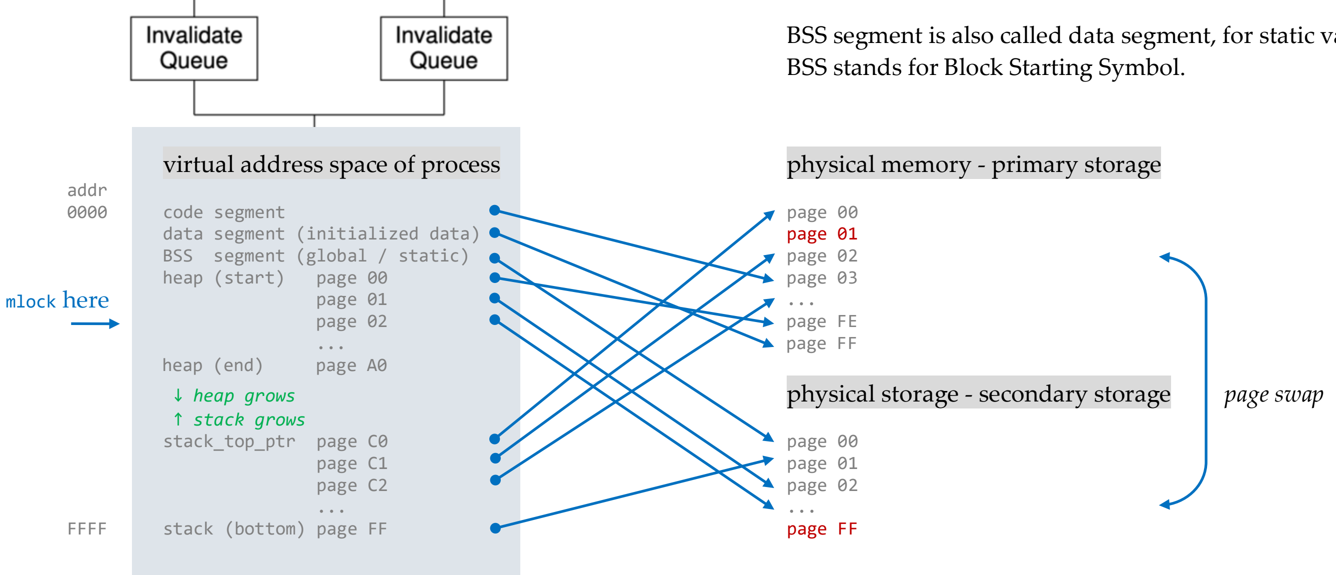
- virtual address space is divided into kernel space and user space
- all threads in one process share same heap
- each thread in one process has its own stack
- heap grows as we call `malloc()`, stack grows as we call functions
- stack memory limit can be adjusted by command `ulimit -s unlimited`

2 useful tools for cache hierarchy :

- memory lock helps to avoid paging and even thrashing
- memory barrier helps to ensure sequence consistency

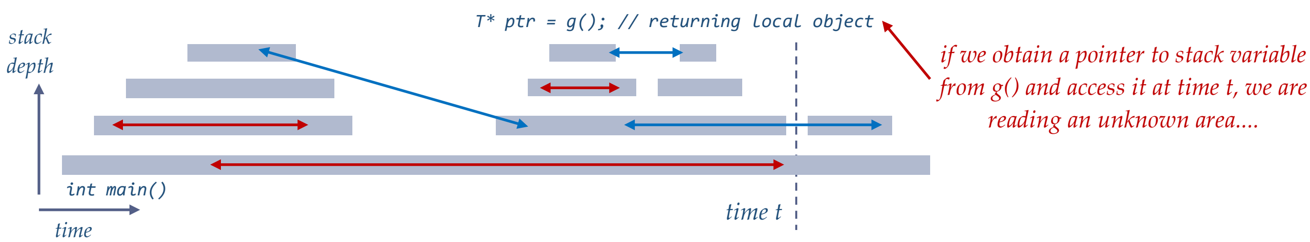


BSS segment is also called data segment, for static variables.
BSS stands for Block Starting Symbol.



Suppose we declare some objects with various lifetimes, the following shows how call stack changes in time :

- red objects stay in same stack frame, hence declaring them in stack memory is good enough
- blue objects live across frames, hence we need declare them in heap memory using `malloc()`



Why do we need complicated memory allocation/deallocation and object lifetime?
Since we are no longer doing procedural programming, sometimes we want to create an object that lives across call-stack :

- object created by factory and pass it elsewhere
- object created by producer and pass it to consumer in *MPMCQ*

The lifetime is no longer confined by callstack, thus we need heap object, hence it comes with lifetime management, we need smart pointers to help us to manage resources with various levels of ownerships. Same thing happens to containers :

- containers with runtime growing capacity, require dynamic allocation
- containers which are node-based, require dynamic allocation for each node
- ► this is why we develop a container library in `YLib` to replace all heap allocations with stack memory for low latency

Using dynamic allocation can extend the lifetime of object but :

- increase complexity as we need to know the number of objects referencing that resource
- increase computation as we need to allocate and deallocate on demand
- multithread safety is not easy for reference count

I get used to implement contiguous memory container (usually) ring buffer by :

```
template<typename T, std::uint32_t N> class lockfree_mpmcq
{
    void push_back(const T& x) { impl[next_write] = x; ++next_write; }
    T impl[n];
};
```

which is not good, it should be done by `char` array instead :

- to avoid double initialization, once by `T::T()` and once by `T::T(const T&)` (what's problem with double initialization?)
- to avoid requiring `T` to be default-constructible (`std::vector` does not have such requirement on `T`)

```
template<typename T, std::uint32_t N> class lockfree_mpmcq
{
    // woo ... everything kicks in : placement new, variadic template, universal reference
    template<typename... ARGS>
    void emplace_back(ARGS&&... args) { new (&(impl[next_write])) T{std::forward<ARGS>(args)...}; ++next_write; }
    char impl[n*sizeof(T)];
};
```

List of topics - Memory allocation for customized container / smart pointer (we do it a lot in `YLib`)

- understand stack / heap / free store *(see c++01.doc, partA)*
- fast access of serialized data using reinterpret cast for *POD* *(see c++01.doc, partB)*
- overload `new` / `delete` operator *(see c++01.doc, partC)*
- overload placement `new` operator *(see c++01.doc, partC)*
- implementation of movable array with `emplace(ARGS&&... args)` *(see c++02.doc, partB2)*
- implementation of smart pointer with `construct(ARGS&&... args)` *(see c++04.doc, partD1)*
- overload deleter for smart pointer *(see c++04.doc, partD6)*
- overload allocator for container *(see c++04.doc, partD7)*
- ► cache coherency, store buffer and memory barrier *(see atomic.doc)*
- ► virtual address space, page swap and memory lock *(see low latency.doc)*
- ► coroutine ?? *(see c++20.doc)*

Allocation functions :

- `malloc` allocates a specific size
- `calloc` allocates a specific size and resets all bytes as zero
- `realloc` reallocates an existing pointer with a new (larger or smaller) size, it may return a new address
- `new` allocates a specific size and invokes constructor
- `new[]` allocates a specific size plus 8 bytes storing the number of elements and invokes constructor repeatedly