UFSC – CTC – INE

INE5424 - Sistemas Operacionais II

RELATÓRIO DO EXERCÍCIO VI

Multiple, Specialized Heaps

Carlos Bonetti - 12100739

Thiago Senhorinha Rose - 12100774

Rodrigo Aguiar Costa - 12104064

Sobrecarga do operador *new*

Para utilizarmos a nova sintaxe desejada (*new (SYSTEM) Obj*) com diferentes heaps, optamos por sobrecarregar o operador *placement new* com diferentes tipos, representados por enums. Desta forma, em tempo de compilação é realizado um match de cada chamada *new* com a assinatura correspondente ao tipo passado como parâmetro.
Os tipos que representam as diferentes heaps foram declaradas no namespace do EPOS, enquando os operadores no namespace global.
As declarações e implementações dos alocadores da heap SYSTEM e UNCACHED foram declarados e implementados em *system.h*:

system.h

```
// EPOS Global System Abstraction Declarations

#ifndef __system_h
#define __system_h

#include <utility/heap.h>

// Specialized system allocators
__BEGIN_API
enum Heap_Type_System { SYSTEM };
enum Heap_Type_Uncached { UNCACHED };
__END_API

inline void * operator new(size_t bytes, const EPOS::Heap_Type_System
&);
ineline void * operator new[](size_t bytes, const
EPOS::Heap_Type_System &);

inline void * operator new(size_t bytes, const EPOS::Heap_Type_Uncached
&);
inline void * operator new[](size_t bytes, const
EPOS::Heap_Type_Uncached &);

__BEGIN_SYS

class System
{
    friend class Init_System;
    friend class Init_Application;
    //friend void * kmalloc(size_t);
    //friend void kfree(void *);

    friend void * ::operator new(size_t bytes, const
EPOS::Heap_Type_System &);
    friend void * ::operator new[](size_t bytes, const
EPOS::Heap_Type_System &);
```

```
    friend void * ::operator new(size_t bytes, const
EPOS::Heap_Type_Uncached &);
    friend void * ::operator new[](size_t bytes, const
EPOS::Heap_Type_Uncached &);

...
};

__END_SYS

// System heaps allocator definition

// SYSTEM heap allocator
inline void * operator new(size_t bytes, const EPOS::Heap_Type_System &
heap) {
    return EPOS::System::_heap->alloc(bytes);
}
inline void * operator new[](size_t bytes, const EPOS::Heap_Type_System
& heap) {
    return EPOS::System::_heap->alloc(bytes);
}

// UNCACHED heap allocator
inline void * operator new(size_t bytes, const EPOS::Heap_Type_Uncached
& heap) {
    return EPOS::System::_uncached_heap->alloc(bytes);
}
inline void * operator new[](size_t bytes, const
EPOS::Heap_Type_Uncached & heap) {
    return EPOS::System::_uncached_heap->alloc(bytes);
}

#endif
```

Neste caso, os operadores *new* foram declarados como *friend* na classe System, para estes terem acesso a suas variáveis protegidas.

Definição do Operador new Padrão

O operador *new* padrão é definido em malloc.h. Desta forma, ao usar um alocador da forma `new Type;` comum, uma chamada a `malloc` acontece que por sua vez chama `Application::_heap->alloc`. Este é exatamente o comportamento que queremos, então não precisamos mexer aqui.

Sobrecarga do operador *delete*

Ao alocar um espaço de memória para um objeto via Heap, além de alocar um espaço "*fake*" destinado a salvar o número de bytes alocados para este objeto, modificamos o

método para também alocar o ponteiro para a heap no qual o objeto foi salvo. Assim, ao chamar delete de um ponteiro qualquer, podemos carregar o ponteiro da heap no qual este objeto foi alocado e chamar o método *free* correspondente àquela heap.

heap.h

```
void * alloc(unsigned int bytes) {
        db<Heaps>(TRC) << "Heap::alloc(this=" << this << ",bytes=" <<
bytes;

        if(!bytes)
            return 0;

        if(!Traits<CPU>::unaligned_memory_access)
            while((bytes % sizeof(void *)))
                ++bytes;

        // add room for heap pointer
        bytes += sizeof(Heap *);

        bytes += sizeof(int);          // add room for size
        if(bytes < sizeof(Element))
            bytes = sizeof(Element);

        Element * e = search_decrementing(bytes);
        if(!e) {
            out_of_memory();
            return 0;
        }

        int * addr = reinterpret_cast<int *>(e->object() + e->size());

        *addr++ = reinterpret_cast<int>(this); // add heap pointer

        *addr++ = bytes;

        db<Heaps>(TRC) << ") => " << reinterpret_cast<void *>(addr) <<
endl;

        return addr;
    }
```

O novo delete padrão também foi definido em heap.h:

heap.h

```
// Delete cannot be declared inline due to virtual destructors
void operator delete(void * ptr);
void operator delete[](void * ptr);
```

```
__BEGIN_UTIL

// Heap
class Heap: private Grouping_List<char>
{ ...
```

E implementado em heap.cc:

heap.cc

```
void operator delete(void * ptr) {
    int * addr = reinterpret_cast<int *>(ptr);
    unsigned int bytes = *(--addr);
    EPOS::Heap * heap = reinterpret_cast<EPOS::Heap *>(*--addr);
    heap->free(addr, bytes);
}
void operator delete[](void * ptr) {
    int * addr = reinterpret_cast<int *>(ptr);
    unsigned int bytes = *(--addr);
    EPOS::Heap * heap = reinterpret_cast<EPOS::Heap *>(*--addr);
    heap->free(addr, bytes);
}
```

Operador *delete* padrão

O operador delete padrão era antes definido em malloc.h. Sua implementação, que ocorria em malloc.cc realizava um free(ptr) que por sua vez chamava `Application::_heap->free(ptr)`. Como queremos que toda chamada a delete caia no método free correspondente à heap utilizada para alocar o objeto, podemos remover as definições de delete de malloc.cc e malloc.h, excluindo o arquivo malloc.cc que já não tem mais utilidade. O delete genérico definido em heap.h agora realiza esta tarefa.

malloc.cc (arquivo removido)

```
-// EPOS Application-level Dynamic Memory Utility Implementation

-#include <system/config.h>
-#include <utility/malloc.h>

-// C++ dynamic memory deallocators
-void operator delete(void * object) {
-    return free(object);
-}

-void operator delete[](void * object) {
-    return free(object);
-}
```

malloc.h (declarações de delete removidas)

```
...
-// Delete cannot be declared inline due to virtual destructors
-void operator delete(void * ptr);
-void operator delete[](void * ptr);

#endif
```

Eliminação de kmalloc

Após a inclusão da operação *new (SYSTEM)* a função *kmalloc* não se faz mais necessária. Portanto, removemos o arquivo *kmalloc.h* e alteramos as referências de *kmalloc()* para *new (SYSTEM)* e *kfree()* para *delete*, nos seguintes arquivos:

- thread.cc - constructor_prolog e destrutor
- alarm_init.cc - Alarm::init()
- thread_init.cc - Thread::init()
- init_first - Init_First()

Também removemos as assinaturas de *kmalloc* e *kfree* da lista *friend* da classe *System*.

Criação da Heap Uncached

A declaração da nova Heap Uncached foi feita em *system.h*, de forma similar à SYSTEM heap:

system.h

```
class System {
...
private:
    static System_Info<Machine> * _si;
    static char _preheap[sizeof(Heap)];
    static Heap * _heap;
    static char _preheap_uncached[sizeof(Heap)];
    static Heap * _uncached_heap;
}
```

E definida em *system_scaffold.cc*:

system_scaffold.cc

```
// System class attributes
...
char System::_preheap_uncached[];
Heap * System::_uncached_heap;
```

Tornando a Heap um Segment e anexando ao Address Space

Como cada Heap agora deverá ter um Segment que a referencie, optamos por fazer a Heap extender Segment, assim cada Heap criada automaticamente terá uma tabela Segment que referencia seu espaço. Esse Segment, porém, deve ser anexado ao Address Space do sistema. Fazemos isso no construtor da Heap/Segment:

heap.h

```
class Heap: private Grouping_List<char>, public Segment
{
public:
    using Grouping_List<char>::empty;
    using Grouping_List<char>::size;

    Heap(void * addr, unsigned int bytes, Flags flags = Flags::APP) :
Segment(bytes, flags) {
        db<Init, Heaps>(TRC) << "Heap(addr=" << addr << ",bytes=" <<
bytes << ") => " << this << endl;

        Address_Space(MMU::current()).attach(this);

        free(addr, bytes);
    }
...
```

Adicionamos o parâmetro opcional flags para alterar as políticas do Segment. Também criamos uma nova policy de flags para o segmento de memória UNCACHED, ativando o bit de write-through:

mmu.h

```
class Flags
{
public:
    enum {
            PRE = 0x001, // Presence (0=not-present, 1=present)
            RW  = 0x002, // Write (0=read-only, 1=read-write)
            USR = 0x004, // Access Control (0=supervisor, 1=user)
            CWT = 0x008, // Cache Mode (0=write-back, 1=write-through)
            CD  = 0x010, // Cache Disable (0=cacheable,
1=non-cacheable)
            CT  = 0x020, // Contiguous (0=non-contiguous, 1=contiguous)
            IO  = 0x040, // Memory Mapped I/O (0=memory, 1=I/O)
            SYS = (PRE | RW ),
            APP = (PRE | RW | USR),
            UNC = (PRE | RW | CWT) // New UNCACHED policy
        };
```

Inicialização da nova Heap Uncached

A inicialização da nova heap foi incluída em Init_System(), logo após a inicialização da SYSTEM heap.

init_system.cc

```
Init_System() {
    ...
    // Initialize System's heap
    ...

    // Initialize System's UNCACHED heap
    db<Init>(INF) << "Initializing system's uncached heap: " << endl;
    System::_uncached_heap = new (&System::_preheap_uncached[0])
Heap(MMU::alloc(MMU::pages(HEAP_SIZE)), HEAP_SIZE, Heap::Flags::UNC);
    db<Init>(INF) << "done!" << endl;
    ...
}
```

A Heap é inicializada com a Flag UNC, que ativa o bit de *write-through* para o Segmento correspondente à Heap criada.