Relatório VI de Sistemas Operacionais II - INE5424
Heaps múltiplas e Especializadas

Bruno Izaias Bonotto - 16104059

Fabíola Maria Kretzer - 16100725

João Vicente Souto - 16105151

1. Introdução

Diversas máquinas atuais possuem memórias especializadas e dispositivos que são mapeadas no espaço de endereçamento do processador, permitindo acessá-lo diretamente como se fosse um acesso comum a Memória de Acesso Randômico - *Random Access Memory* (RAM). Essas memórias são frequentemente encontradas em máquinas de alto desempenho e utilizam memória compartilhada, tais como os processadores multicore contemporâneos. No entanto, o acesso e manipulação não são frequentemente exploradas pelos sistemas operacionais, demandando grande esforço pelo programador caso queira utilizar tais recursos.

Inicialmente, a versão didática do EPOS possuía uma implementação de acesso a memória que não abstraia as diferentes características das memórias especializadas. Assim, o objetivo deste trabalho consiste em entender o funcionamento da alocação dinâmica de memória e então criar espaços de endereçamento distintos, facilitando seu uso ao fornecer uma abstração da função *new* da linguagem C++.

2. Soluções Utilizadas

Um dos tópicos requisitados para este trabalho é a substituição da função de baixo nível *kmalloc* pelo *operator new*, permitindo que o programador possa especificar, em um parâmetro opcional, em qual dos tipos de mapeamento se deseja alocar a memória. Como solução, foi utilizado o tipo *enum* para cada tipo de *heap* existente. Dessa forma, ao utilizar *operator new* com um dos tipos

especificados é possível a realização de otimizações em tempo de compilação para definir o espaço de endereçamento que será mapeado.

types.h

```
__BEGIN_API
enum Type_System {
        SYSTEM
};

enum Type_Uncached {
        UNCACHED
};
__END_API

void * operator new(size_t, const _API::Type_System &);
void * operator new[](size_t, const _API::Type_System &);

void * operator new(size_t, const _API::Type_Uncached &);
void * operator new[](size_t, const _API::Type_Uncached &);
```

Utilizando-se da nova assinatura do *operator new*, todos os lugares onde usava-se a função *kmalloc()* foram modificados para chamar o alocador de memória do *Kernel*, mantendo a semântica de mapeamento de memória anteriormente implementada.

alarm_init.cc

```
_timer = new (SYSTEM) Alarm_Timer(handler);
```

thread.cc

```
_stack = new (SYSTEM) char[stack_size];
```

thread_init.cc

```
_timer = new (SYSTEM) Scheduler_Timer(QUANTUM, time_slicer);
```

init_first.cc

```
Thread::_running = new (SYSTEM) Thread(Thread::Configuration(Thread::RUNNING,
                                    Thread::MAIN), reinterpret_cast<int (*)()>(__epos_app_entry));
new (SYSTEM) Thread(Thread::Configuration(Thread::READY, Thread::IDLE),
&Thread::idle);
```

A separação das regiões de memória (endereçamento distinto de memória de acordo com o novo parâmetro do *operator new*) ocorre no *system.h*, abstração esta que encapsula uma região de memória que deve ser vinculada ao kernel do sistema, e a *heap* que já existia. Assim, de acordo com a especificação do trabalho, as alterações necessárias foram realizadas também no arquivo *application_scaffold.cc*, onde foi definida a *heap* de segmentos.

system.h

```
class System {
    ...
    static char _preheap[(Traits<System>::multiheap ? sizeof(Segment) : 0) + sizeof(Heap)];
    static Segment * _heap_segment;
    static Heap * _heap;
    ...
}

inline void * operator new(size_t bytes, const EPOS::Type_System & a) {
    return EPOS::System::_heap->alloc(bytes);
}
inline void * operator new[](size_t bytes, const EPOS::Type_System & a) {
    return EPOS::System::_heap->alloc(bytes);
}
```

application_scaffold.cc

```
Segment * System::_heap_segment;
```

Ao inicializar o sistema operacional, as *heaps* serão alocadas através da execução do *init_system.cc*, onde é respeitado o conceito de segmentos

implementado pelo EPOS. Os segmentos, por sua vez, podem existir e conter memória e, ainda assim, serem inacessíveis por não estarem alocados a nenhum espaço de endereçamento, portanto, obriga-se a realização do mapeamento na inicialização o sistema.

init_system.cc

```
if(Traits<System>::multiheap) {
   System::_heap_segment = new (&System::_preheap[0])
                                                Segment(Traits<System>::HEAP_SIZE);
   System::_heap = new (&System::_preheap[sizeof(Segment)])
         Heap(Address_Space(MMU::current()).attach(*System::_heap_segment,
               Memory_Map<Machine>::SYS_HEAP), System::_heap_segment->size());
} else {
   System::_heap = new (&System::_preheap[0])
         Heap(MMU::alloc(MMU::pages(Traits<System>::HEAP_SIZE)),
                                                      Traits<System>::HEAP_SIZE);
}
```

Outro ponto importante sobre a implementação utilizada de base relaciona-se à classe *Heap*, sendo necessário a adaptação das funções *alloc()* e *free()* para suportar o paradigma *multiheap*. Neste ponto, é necessário a reserva de um espaço para o ponteiro da *heap,* permitindo obter informações da mesma após a alocação ser de fato realizada. Também fez-se necessário a implementação de funções específicas para liberação de memória quando utilizado o modo de *multiheap* (*os_free()*) e outra para uso no modo normal (*simple_free()*).

heap.h

```
class Heap: private Grouping_List<char>
{
protected:
   static const bool typed_heap = Traits<System>::multiheap;
    …
    void * alloc(unsigned int bytes) {
         …
         if(typed_heap) { bytes += sizeof(void *); }
```

```
            …
            if(typed_heap) { *addr++ = reinterpret_cast<int>(this); }
            …
        }
    }

    static void os_free(void * ptr) {
        int * addr = reinterpret_cast<int *>(ptr);
        unsigned int bytes = *--addr;
        Heap * heap = reinterpret_cast<Heap *>(*--addr);
        heap->free(addr, bytes);
      }

    static void simple_free(Heap * heap, void * ptr) {
        int * addr = reinterpret_cast<int *>(ptr);
        unsigned int bytes = *--addr;
        heap->free(addr, bytes);
    }
```

Para se fazer uso correto da alocação dinâmica, foram modificados os métodos *malloc()* e *free()* utilizados implicitamente no *operator new* e *delete*, especificados pelo *Standard c*, de modo a tratar o paradigma *multiheaps*.

malloc.h

```
inline void * malloc(size_t bytes) {
    if(Traits<System>::multiheap) {
            return Application::_heap->alloc(bytes);
    } else {
            return System::_heap->alloc(bytes);
    }
}

inline void free(void * ptr) {
    if(Traits<System>::multiheap) {
            Heap::os_free(ptr);
    } else {
            Heap::simple_free(System::_heap, ptr);
```

```
    }
}
```

Conclusão

O pouco tempo estipulado entre os trabalhos E5 e E6 e a falta de implementação de alguns componentes necessários para o correto desenvolvimento deste trabalho (*Segment* e *Address_Space*) fez com que fosse necessária a busca de uma solução já implementada (obtida em: https://github.com/eloisamec/epos) para ser utilizada de guia na solução deste trabalho. Assim. buscou-se identificar os principais pontos modificados na implementação e relacioná-los com os objetivos deste trabalho, tomando o cuidado em explicar o que foi possível interpretar. Porém, pela falta de tempo e complexidade dos conceitos envolvidos, a especificação ficou a desejar.

Notou-se que a solução estudada não implementou *heaps/*segmentos com políticas de cache distintas (*write-back* e *write-through*) assim como solicitado. Entretanto, acredita-se que isso seja possível de se configurar por meio da classe *MMU* (responsável pela paginação e gerenciamento de memória). O que seria viável fazendo-se uso das *flags* (*CWT* - Cache Mode) associadas a uma determinada porção de memória, permitindo que o sistema respeite a semântica atrelada às respectivas políticas de escrita.