Universidade Federal de Santa Catarina
Centro Tecnológico
Departamento de Informática e Estatística
INE5424 - Sistemas Operacionais II
Professor: Antônio Augusto Medeiros Fröhlich
Estagiário de Docência: Mateus Krepsky Ludwich
Grupo: 01
Alunos: Alisson Granemann Abreu matrícula: 11100854
Maria Eloísa Costa matrícula: 10200630
Osvaldo Edmundo Schwerz da Rocha matrícula: 10103132

E6: Multiple, Specialized Heaps

Some machines have specialized memories that are mapped in the processor's address space just like ordinary RAM. For instance, some high-performance machines feature low-latency scratchpad memories used to speed up computations. Such memories, however, are seldom properly exported by the operating system, requiring application programs to directly handle them. The case of memory regions with specific caching polices for parallel applications running on shared memory machines, including contemporary multicore processors, fits in the same scenario. A modern operating system could take advantage of the fact that operator new() in C++ has two signatures and override the so called *placement new* to seamlessly export such memories. For example, application programmers could write

```
Type * object = new Type;
```

to create an object of type Type on the ordinary memory, and

```
Type * object = new (UNCACHED) Type;
```

to create the object on uncached memory.

To do

Split the application heap in two and implement the mechanism illustrated above so that programmers will be able to allocate memory that is cached on a write-back policy from one heap and memory that is cached on a write-through policy from the other one. The kmalloc() function shall no longer exist.

Hint: the heap on your current version of OpenEPOS is declared in application.h, defined in application_scaffold.cc (it's a static attribute), and initialized in init_application.cc through the invocation of MMU::alloc() to allocate memory for the heap and Heap::free() to inject that memory into the heap. A system abstraction called Segment does most of what it takes to accomplish the task, but do not forget that segments in EPOS designate memory regions and not address space mappings. A Segment may exist, it may contain memory, and yet it may be inaccessible because it was not attached (using Address_Space::attach()) to any Address_Space

Done

Operador new

O primeiro ponto importante para este trabalho era fazer com que a função kmalloc() não fosse mais chamada em nenhuma parte do sistema. Para tal, fizemos inicialmente um mapeamento de onde essa função era chamada, pois ela terá de ser alterada para um operador new():

system.h

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

...
```

thread.h

```
template<typename ... Tn>
inline Thread::Thread(int (* entry)(Tn ...), Tn ... an)
: _state(READY), _waiting(0), _joining(0), _link(this, NORMAL)
{
    lock();
    _stack = reinterpret_cast<char *>(kmalloc(STACK_SIZE));
```

```
    _context = CPU::init_stack(_stack, STACK_SIZE, &implicit_exit, entry, an
...);
    constructor(entry, STACK_SIZE); // implicit unlock
}

template<typename ... Cn, typename ... Tn>
inline Thread::Thread(const Configuration & conf, int (* entry)(Tn ...), Tn
... an)
: _state(conf.state), _waiting(0), _joining(0), _link(this, conf.priority)
{
    lock();
    _stack = reinterpret_cast<char *>(kmalloc(conf.stack_size));
    _context = CPU::init_stack(_stack, conf.stack_size, &implicit_exit, entry,
an ...);
    constructor(entry, conf.stack_size); // implicit unlock
}
```

alarm_init.cc

```
void Alarm::init()
{
    db<Init, Alarm>(TRC) << "Alarm::init()" << endl;

    _timer = new (kmalloc(sizeof(Alarm_Timer))) Alarm_Timer(handler);
}
```

thread_init.cc

```
void Thread::init()
{
    int (* entry)() = reinterpret_cast<int (*)()>(__epos_app_entry);

    db<Init, Thread>(TRC) << "Thread::init(entry=" << reinterpret_cast<void
*>(entry) << ")" << endl;

    // Create the application's main thread
    // This must precede idle, thus avoiding implicit rescheduling
    // For preemptive scheduling, reschedule() is called, but it will preserve
MAIN as the RUNNING thread
    _running = new (kmalloc(sizeof(Thread))) Thread(Configuration(RUNNING,
MAIN), entry);
    new (kmalloc(sizeof(Thread))) Thread(Configuration(READY, IDLE), &idle);


    if(preemptive)
        _timer = new (kmalloc(sizeof(Scheduler_Timer)))
Scheduler_Timer(QUANTUM, time_slicer);

    db<Init, Thread>(INF) << "Dispatching the first thread: " << _running <<
endl;

    This_Thread::not_booting();
```

```
    _running->_context->load();
}
```

Para fazer a alteração dos kmalloc para new, é necessário primeiramente fazer o overload do operador new para que seja específico de uma determinada classe. No caso, o operador new deverá receber o tipo de alocação, na forma de um enum, para saber qual heap utilizar quando for inicializado. O uso de um enum permite que a decisão de qual heap usar seja feita em tempo de compilação.

types.h

```
...
__BEGIN_API

enum Type_System {
    SYSTEM
};

enum Type_Uncached {
    UNCACHED
};

__END_API

inline void * operator new(size_t s, void * a) { return a; }
inline void * operator new[](size_t s, void * a) { return a; }

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

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

O uso de __BEGIN_API para a definição dos enums define em qual namespace existente em *config.h* o enum será definido. A decisão da criação no namespace API foi por este utilizar todos os outros namespaces (sys e util), ou seja, caso haja a definição do enum em outro namespace (sys ou util), esse será escondido pelo enum definido em api[1].

Com estas alterações, torna-se possível fazer as alterações nos locais mapeados com kmalloc para que seja removido do sistema.

[1] Refs: http://en.cppreference.com/w/cpp/language/namespace

Em thread_init foi comentado o include e alterada a chamada do kmalloc para SYSTEM (enum do new de sistema).

thread_init.cc

```
void Thread::init()
{
    int (* entry)() = reinterpret_cast<int (*)()>(__epos_app_entry);

    db<Init, Thread>(TRC) << "Thread::init(entry=" << reinterpret_cast<void
*>(entry) << ")" << endl;

    // Create the application's main thread
    // This must precede idle, thus avoiding implicit rescheduling
    // For preemptive scheduling, reschedule() is called, but it will preserve
MAIN as the RUNNING thread
    _running = new (SYSTEM) Thread(Configuration(RUNNING, MAIN), entry);
    new (SYSTEM) Thread(Configuration(READY, IDLE), &idle);


    if(preemptive)
        _timer = new (SYSTEM) Scheduler_Timer(QUANTUM, time_slicer);

    db<Init, Thread>(INF) << "Dispatching the first thread: " << _running <<
endl;

    This_Thread::not_booting();

    _running->_context->load();
}
```

Em alarm_init também foi feito o mesmo tipo de alteração.

alarm_init.cc

```
void Alarm::init()
{
    db<Init, Alarm>(TRC) << "Alarm::init()" << endl;

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

A alteração em thread.h é um pouco diferente, visto a forma que é usado o kmalloc para a criação da pilha no construtor da thread. Como o construtor original utiliza `reinterpret_cast<char *>` como forma de definir o tipo do kmalloc, optou-se por utilizar também char para definir o new. Isso se deve ao fato de que char em qualquer arquitetura ser um byte.

thread.h

```
template<typename ... Tn>
inline Thread::Thread(int (* entry)(Tn ...), Tn ... an)
: _state(READY), _waiting(0), _joining(0), _link(this, NORMAL)
{
    lock();
    _stack = new (SYSTEM) char[STACK_SIZE];
    _context = CPU::init_stack(_stack, STACK_SIZE, &implicit_exit, entry, an
...);
    constructor(entry, STACK_SIZE); // implicit unlock
}

template<typename ... Cn, typename ... Tn>
inline Thread::Thread(const Configuration & conf, int (* entry)(Tn ...), Tn
... an)
: _state(conf.state), _waiting(0), _joining(0), _link(this, conf.priority)
{
    lock();
    _stack = new (SYSTEM) char[conf.stack_size];
    _context = CPU::init_stack(_stack, conf.stack_size, &implicit_exit, entry,
an ...);
    constructor(entry, conf.stack_size); // implicit unlock
}
```

Em system.h a princípio, comentamos o kmalloc existente por não ser mais necessário utilizá-lo, mas torna-se necessário fazermos outras alterações, tanto na definição do próprio new de sistema, por causa do overload feito em types.h, como a questão relativa ao delete, visto que o new é outro. O new criado em types torna-se friend na classe System para que possa ser definido inline em system, definindo-se assim o tipo de return deste novo new.

system.h

```
extern "C"
{
    void * malloc(size_t);
    void free(void *);
}

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); }

__BEGIN_SYS

class System
{
    friend class Init_System;
```

```
    friend class Init_Application;
    //friend void * kmalloc(size_t);
    //friend void kfree(void *);
    friend void * ::malloc(size_t);
    friend void ::free(void *);
    friend void * ::operator new(size_t, const EPOS::Type_System &);
    friend void * ::operator new[](size_t, const EPOS::Type_System &);
    friend void ::operator delete(void *);
    friend void ::operator delete[](void *);
...
```

Em thread.cc também existia um kfree que não deveria mais existir, visto que o header kmalloc foi removido dos arquivos do sistema. Ele foi substituído por um delete.

thread.cc

```
    if(_joining)
        _joining->resume();

    unlock();

    delete(_stack);
}
```

Falta no momento o tratamento do operador delete, que depende da heap em que estiver alocada, e a separação das heaps. Como o operador delete chama o método free contido em malloc.h, deixaremos para mais tarde, pois teremos que mexer no tipo de heap.

Algumas alterações de código nos levaram a uma informação contida em traits.h:

```
static const bool multiheap = (mode != Traits<Build>::LIBRARY);
```

Ela determina se a característica de heaps especializadas será utilizada ou não de acordo com o valor setado em *mode*.

A declaração das heaps foram feitas em *system.h*, uma para sistema e uma uncached, mostrado anteriormente. Outra alteração feita foi a declaração de uma heap de segmentos (informação contida na hint do enunciado) e a verificação do tipo de pre_heap que estamos criando:

system.h

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

Para _preheap foi feita uma verificação caso estejamos utilizando multiheap, a preheap terá o tamanho do segmento + o tamanho da heap. Caso contrário será criada a preheap do tamanho da heap normal.

Em system_scaffold.cc é feita a definição da heap de segmentos:

system_scaffold.cc

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

E em init_system.cc é feita a instanciação da nova heap de segmentos. Ao iniciar a heap de sistema é verificado se o sistema está setado para multiheap e dependendo do que estiver, será criada a heap de segmentos ou a heap normal. Como descrito no próprio enunciado, um segmento pode existir, pode conter memória e ainda assim ser inacessível porque não foi alocado usando Address_Space::attach() em nenhum espaço de endereçamento.

init_system.cc

```
	// Initialize System's heap
	  db<Init>(INF) << "Initializing system's heap: " << endl;
	  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);
	  }

	  db<Init>(INF) << "done!" << endl;
```

Também foi necessário adaptar os métodos alloc e free da heap para suportar multiheap. Se o modo multiheap estiver habilitado, se faz necessário a adição de espaço para o ponteiro da heap também, facilitando posteriormente a adição de informações sobre o ponteiro da heap.

heap.h

```
    protected:
        static const bool typed_heap = Traits<System>::multiheap;

    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;

      if(typed_heap) {
           bytes += sizeof(void *);
      }
      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());

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

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

      return addr;
    }
    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);
    }
```

Foram criados também novos métodos free que fossem compatíveis com a necessidade do uso de heaps especializadas ou heaps simples. Elas foram usadas em malloc.h para tratar do uso de 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);
   }
}
```

Ainda resta fazer o tratamento para tipos uncached. Ficou a dúvida sobre criar um arquivo uncached.h ou adicionar as informações diretamente em application.h. Porém adicionar em application.h gerou outra dúvida: Todas as aplicações criadas seriam criadas diretamente com o tipo uncached? Por não termos essa resposta e fazer separado em um uncached.h envolveria mexer em outros pontos do sistema que não temos conhecimento, decimos por não fazer essa alteração.