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: 10
Alunos: André Beims Bräscher matrícula: 12106259
Maria Eloísa Costa matrícula: 10200630

P2: Proxy/Agent

To Do

Projete e implemente um mecanismo para desacoplar os serviços prestados pelo SO de seus clientes. O mecanismo é referenciado na literatura como "stub/skeleton" ou então "proxy/agent". Toda a interação entre a aplicação e o SO deverá ser feita através de mensagens e não mais com chamadas de procedimentos (call).

Done

A solução começou com a elaboração da estrutura da mensagem a ser passada entre aplicação e sistema, obtendo-se a classe "message". Tal mensagem deve conter informações como qual função a ser executada, de qual classe, argumentos, etc. Assim, pode-se ver a implementação da mensagem a seguir:

stub/message.h

```
#ifndef __message_h
#define __message_h

#include <system/config.h>

__BEGIN_API

class Message {

public:
	void class_id(int id);
	int class_id();

	void method_id(int id);
	int method_id();

	void object_id(void * id);
	void * object_id();

	void param1(void * ptr);
	void * param1();

	void param2(void * ptr);
	void * param2();
```

```
        void param3(void * ptr);
        void * param3();

        void param4(void * ptr);
        void * param4();

        void param5(void * ptr);
        void * param5();

        void param6(void * ptr);
        void * param6();

        void param7(void * ptr);
        void * param7();

        void return_value(void * ptr);
        void * return_value();

private:
        int _class_id;
        int _method_id;
        void * _object_id;
        void * _return;
        void * _param1;
        void * _param2;
        void * _param3;
        void * _param4;
        void * _param5;
        void * _param6;
        void * _param7;

};

__END_API
#endif
```

abstraction/message.cc

```
#include <stub/message.h>

__BEGIN_API

        void Message::class_id(int id) { _class_id = id; }
        int Message::class_id() { return _class_id; }

        void Message::method_id(int id) { _method_id = id; }
        int Message::method_id() { return _method_id; }

        void Message::object_id(void * id){ _object_id = id; }
        void * Message::object_id(){ return _object_id; }

        void Message::param1(void * ptr) { _param1 = ptr; }
        void * Message::param1() { return _param1; }
```

```
    void Message::param2(void * ptr) { _param2 = ptr; }
    void * Message::param2() { return _param2; }

    void Message::param3(void * ptr){ _param3 = ptr; }
    void * Message::param3(){ return _param3; }

    void Message::param4(void * ptr){ _param4 = ptr; }
    void * Message::param4(){ return _param4; }

    void Message::param5(void * ptr){ _param5 = ptr; }
    void * Message::param5(){ return _param5; }

    void Message::param6(void * ptr){ _param6 = ptr; }
    void * Message::param6(){ return _param6; }

    void Message::param7(void * ptr){ _param7 = ptr; }
    void * Message::param7(){ return _param7; }

    void Message::return_value(void * ptr){ _return = ptr; }
    void * Message::return_value(){ return _return; }

__END_API
```

Pode-se perceber que a mensagem elaborada contem os campos class_id, method_id, object_id e diversos parâmetros. Como resultado são enviadas todas informações que o sistema precisa para executar uma de suas funções. Porém, para que o sistema seja capaz de efetivamente saber o que deve ser executado, no momento em que deve ser executado é necessário que a aplicação gere tais mensagens e por sua vez o sistema seja capaz de "compreender" tais mensagens. Tais necessidades são atendidas através das estruturas do tipo "stub" e "skeleton", respectivamente.

Estruturas "stub", são as estruturas responsáveis por gerar as mensagens que serão enviadas ao sistema e receber as respostas do sistema. A abordagem do grupo foi elaborar uma stub personalizada para cada classe, atendendo as respectivas assinaturas das funções de tais classes do sistema (incluída ao final do relatório)

Uma vez elaboradas as stubs, foi elaborada a estrutura skeleton a qual é responsável por receber as mensagens, identificar como deve ser a chamada de sistema e chamar adequadamente a função com os argumentos passados, do sistema.

stub/skeleton.h

```
#ifndef __skeleton_h
#define __skeleton_h

#include <stub/message.h>

__BEGIN_SYS

class Skeleton {
```

```
public:
	static void call(Message * m);

private:
	// Address Space
	static void address_space_constructor_1(Message * m);
	static void address_space_constructor_2(Message * m);
	static void address_space_destructor(Message * m);
	static void address_space_attach_1(Message * m);
	static void address_space_attach_2(Message * m);
	static void address_space_detach(Message * m);
	static void address_space_physical(Message * m);

	// Alarm
	static void alarm_constructor(Message * m);
	static void alarm_destructor(Message * m);
	static void alarm_frequency(Message * m);
	static void alarm_delay(Message * m);

	// Condition
	static void condition_constructor(Message * m);
	static void condition_destructor(Message * m);
	static void condition_wait(Message * m);
	static void condition_signal(Message * m);
	static void condition_broadcast(Message * m);

	// Display
	static void display_constructor(Message * m);
	static void display_clear(Message * m);
	static void display_putc(Message * m);
	static void display_puts(Message * m);
	static void display_geometry(Message * m);
	static void display_position_1(Message * m);
	static void display_position_2(Message * m);

	// Handler
	static void handler_constructor(Message * m);
	static void handler_destructor(Message * m);

	// Mutex
	static void mutex_constructor(Message * m);
	static void mutex_destructor(Message * m);
	static void mutex_lock(Message * m);
	static void mutex_unlock(Message * m);

	// Segment
	static void segment_constructor_1(Message * m);
	static void segment_constructor_2(Message * m);
	static void segment_destructor(Message * m);
	static void segment_size(Message * m);
	static void segment_phy_address(Message * m);
	static void segment_resize(Message * m);

	// Semaphore
	static void semaphore_constructor(Message * m);
```

```
        static void semaphore_destructor(Message * m);
        static void semaphore_p(Message * m);
        static void semaphore_v(Message * m);

        // Task
        static void task_constructor(Message * m);
        static void task_destructor(Message * m);
        static void task_address_space(Message * m);
        static void task_code_segment(Message * m);
        static void task_code(Message * m);
        static void task_data_segment(Message * m);
        static void task_data(Message * m);
        static void task_self(Message * m);

        // Thread
        template<typename T>
        static void thread_constructor_1(Message * m);
        template<typename T1, typename T2>
        static void thread_constructor_2(Message * m);
        template<typename T1, typename T2, typename T3>
        static void thread_constructor_3(Message * m);
        template<typename T1, typename T2, typename T3, typename T4>
        static void thread_constructor_4(Message * m);
        template<typename T1, typename T2, typename T3, typename T4, typename T5>
        static void thread_constructor_5(Message * m);
        template<typename T1, typename T2, typename T3, typename T4, typename T5,
typename T6>
        static void thread_constructor_6(Message * m);
        template<typename T1, typename T2, typename T3, typename T4, typename T5,
typename T6, typename T7>
        static void thread_constructor_7(Message * m);
        static void thread_destructor(Message * m);
        static void thread_state(Message * m);
        static void thread_priority_1(Message * m);
        static void thread_priority_2(Message * m);
        static void thread_task(Message * m);
        static void thread_join(Message * m);
        static void thread_pass(Message * m);
        static void thread_suspend(Message * m);
        static void thread_resume(Message * m);
        static void thread_self(Message * m);
        static void thread_yield(Message * m);
        static void thread_exit(Message * m);
};

__END_SYS
#endif
```

abstraction/skeleton.h

```
#include <stub/skeleton.h>
#include <utility/handler.h>
#include <address_space.h>
#include <alarm.h>
#include <app_types.h>
#include <condition.h>
#include <display.h>
#include <mutex.h>
#include <segment.h>
#include <semaphore.h>
#include <task.h>
#include <thread.h>

__BEGIN_SYS

	void Skeleton::call(Message * m){
		switch(m->class_id()){
			case Class::ADDRESS_SPACE:
				switch(m->method_id()){
					case Method::Address_Space::CONSTRUCTOR_1: address_space_constructor_1(m); break;
					case Method::Address_Space::CONSTRUCTOR_2: address_space_constructor_2(m); break;
					case Method::Address_Space::DESTRUCTOR: address_space_destructor(m); break;
					case Method::Address_Space::ATTACH_1: address_space_attach_1(m); break;
					case Method::Address_Space::ATTACH_2: address_space_attach_2(m); break;
					case Method::Address_Space::DETACH: address_space_detach(m); break;
					case Method::Address_Space::PHYSICAL: address_space_physical(m); break;
				}
				break;
			case Class::ALARM:
				switch(m->method_id()){
					case Method::Alarm::CONSTRUCTOR: alarm_constructor(m); break;
					case Method::Alarm::DESTRUCTOR: alarm_destructor(m); break;
					case Method::Alarm::FREQUENCY: alarm_frequency(m); break;
					case Method::Alarm::DELAY: alarm_delay(m); break;
				}
				break;
			case Class::CONDITION:
```

```
        switch(m->method_id()){
                case Method::Condition::CONSTRUCTOR: condition_constructor(m); break;
                case Method::Condition::DESTRUCTOR: condition_destructor(m); break;
                case Method::Condition::WAIT: condition_wait(m); break;
                case Method::Condition::SIGNAL: condition_signal(m); break;
                case Method::Condition::BROADCAST: condition_broadcast(m); break;
        }
        break;
case Class::DISPLAY:
        switch(m->method_id()){
                case Method::Display::CONSTRUCTOR: display_constructor(m); break;
                case Method::Display::CLEAR: display_clear(m); break;
                case Method::Display::PUTC: display_putc(m); break;
                case Method::Display::PUTS: display_puts(m); break;
                case Method::Display::GEOMETRY: display_geometry(m); break;
                case Method::Display::POSITION_1: display_position_1(m); break;
                case Method::Display::POSITION_2: display_position_2(m); break;
        }
        break;
case Class::HANDLER:
        switch(m->method_id()){
                case Method::Handler::CONSTRUCTOR: handler_constructor(m); break;
                case Method::Handler::DESTRUCTOR: handler_destructor(m); break;
        }
        break;
case Class::MUTEX:
        switch(m->method_id()){
                case Method::Mutex::CONSTRUCTOR: mutex_constructor(m); break;
                case Method::Mutex::DESTRUCTOR: mutex_destructor(m); break;
                case Method::Mutex::LOCK: mutex_lock(m); break;
                case Method::Mutex::UNLOCK: mutex_unlock(m); break;
        }
        break;
case Class::SEGMENT:
        switch(m->method_id()){
                case Method::Segment::CONSTRUCTOR_1: segment_constructor_1(m); break;
                case Method::Segment::CONSTRUCTOR_2: segment_constructor_2(m); break;
                case Method::Segment::DESTRUCTOR: segment_destructor(m); break;
                case Method::Segment::SIZE: segment_size(m); break;
```

```
                case Method::Segment::PHY_ADDRESS: segment_phy_address(m); break;
                case Method::Segment::RESIZE: segment_resize(m); break;
        }
        break;
case Class::SEMAPHORE:
        switch(m->method_id()){
                case Method::Semaphore::CONSTRUCTOR: semaphore_constructor(m); break;
                case Method::Semaphore::DESTRUCTOR: semaphore_destructor(m); break;
                case Method::Semaphore::P: semaphore_p(m); break;
                case Method::Semaphore::V: semaphore_v(m); break;
        }
        break;
case Class::TASK:
        switch(m->method_id()) {
                case Method::Task::CONSTRUCTOR: task_constructor(m); break;
                case Method::Task::DESTRUCTOR: task_destructor(m); break;
                case Method::Task::CODE_SEGMENT: task_code_segment(m); break;
                case Method::Task::DATA_SEGMENT: task_data_segment(m); break;
                case Method::Task::DATA: task_data(m); break;
                case Method::Task::CODE: task_code(m); break;
                case Method::Task::ADDRESS_SPACE: task_address_space(m); break;
                case Method::Task::SELF: task_self(m); break;
        }
        break;
case Class::THREAD:
        switch(m->method_id()) {
                case Method::Thread::CONSTRUCTOR_1: thread_constructor_1(m); break;
                case Method::Thread::CONSTRUCTOR_2: thread_constructor_2(m); break;
                case Method::Thread::CONSTRUCTOR_3: thread_constructor_3(m); break;
                case Method::Thread::CONSTRUCTOR_4: thread_constructor_4(m); break;
                case Method::Thread::CONSTRUCTOR_5: thread_constructor_5(m); break;
                case Method::Thread::CONSTRUCTOR_6: thread_constructor_6(m); break;
                case Method::Thread::CONSTRUCTOR_7: thread_constructor_7(m); break;
                case Method::Thread::DESTRUCTOR: thread_destructor(m); break;
                case Method::Thread::STATE: thread_state(m); break;
                case Method::Thread::PRIORITY_1: thread_priority_1(m); break;
                case Method::Thread::PRIORITY_2: thread_priority_2(m); break;
                case Method::Thread::TASK: thread_task(m); break;
                case Method::Thread::JOIN: thread_join(m); break;
```

```
                        case Method::Thread::PASS: thread_pass(m); break;
                        case Method::Thread::SUSPEND: thread_suspend(m); break;
                        case Method::Thread::RESUME: thread_resume(m); break;
                        case Method::Thread::SELF: thread_self(m); break;
                        case Method::Thread::YIELD: thread_yield(m); break;
                        case Method::Thread::EXIT: thread_exit(m); break;
                    }
                    break;
        }
}

// Address Space
void Skeleton::address_space_constructor_1(Message * m) {
        Address_Space * address_space = new (SYSTEM) Address_Space();
        m->return_value(reinterpret_cast<void *>(&address_space));
}

void Skeleton::address_space_constructor_2(Message * m) {
        MMU::Page_Directory pd = *reinterpret_cast<MMU::Page_Directory *>(m->param1());
        Address_Space * address_space = new (SYSTEM) Address_Space(&pd);
        m->return_value(reinterpret_cast<void *>(&address_space));
}

void Skeleton::address_space_destructor(Message * m) {
        Address_Space * address_space = reinterpret_cast<Address_Space *>(m->object_id());
        delete address_space;
}

void Skeleton::address_space_attach_1(Message * m) {
        Address_Space * address_space = reinterpret_cast<Address_Space*>(m->object_id());
        Segment p1 = *reinterpret_cast<Segment*>(m->param1());
        CPU_Common::Log_Addr la = address_space->attach(p1);
        m->return_value(reinterpret_cast<void *>(&la));
}

void Skeleton::address_space_attach_2(Message * m) {
        Address_Space * address_space = reinterpret_cast<Address_Space*>(m->object_id());
        Segment seg = *reinterpret_cast<Segment*>(m->param1());
        CPU_Common::Log_Addr log_addr = *reinterpret_cast<CPU_Common::Log_Addr *>(m->param2());
```

```
	CPU_Common::Log_Addr la = address_space->attach(seg, log_addr);
	m->return_value(reinterpret_cast<void *>(&la));
}

void Skeleton::address_space_detach(Message * m) {
	Address_Space * address_space = reinterpret_cast<Address_Space*>(m->object_id());
	Segment seg = *reinterpret_cast<Segment*>(m->param1());
	address_space->detach(seg);
}

void Skeleton::address_space_physical(Message * m) {
	CPU_Common::Log_Addr log_addr = *reinterpret_cast<CPU_Common::Log_Addr*>(m->param1());
	Address_Space * address_space = reinterpret_cast<Address_Space*>(m->object_id());
	address_space->physical(log_addr);
}

// Alarm
void Skeleton::alarm_constructor(Message * m) {
	const RTC::Microsecond times = *reinterpret_cast<RTC::Microsecond*>(m->param1());
	Handler * handler = reinterpret_cast<Handler*>(m->param2());
	int time = *reinterpret_cast<int*>(m->param3());
	Alarm * alarm = new (SYSTEM) Alarm(times, handler, time);
	m->return_value(reinterpret_cast<void *>(&alarm));
}

void Skeleton::alarm_destructor(Message * m) {
	Alarm * alarm = reinterpret_cast<Alarm *>(m->object_id());
	delete alarm;
}

void Skeleton::alarm_frequency(Message * m) {
	Alarm * alarm = reinterpret_cast<Alarm*>(m->object_id());
	alarm->frequency();
}

void Skeleton::alarm_delay(Message * m) {
	Alarm * alarm = reinterpret_cast<Alarm*>(m->object_id());
	const RTC::Microsecond time = *reinterpret_cast<RTC::Microsecond*>(m->param1());
	alarm->delay(time);
```

```
}

// Condition
void Skeleton::condition_constructor(Message * m) {
	Condition * condition = new (SYSTEM) Condition();
	m->return_value(reinterpret_cast<void *>(&condition));
}

void Skeleton::condition_destructor(Message * m) {
	Condition * condition = reinterpret_cast<Condition*>(m->object_id());
	delete condition;
}

void Skeleton::condition_wait(Message * m) {
	Condition * condition = reinterpret_cast<Condition*>(m->object_id());
	condition->wait();
}

void Skeleton::condition_signal(Message * m) {
	Condition * condition = reinterpret_cast<Condition*>(m->object_id());
	condition->signal();
}

void Skeleton::condition_broadcast(Message * m) {
	Condition * condition = reinterpret_cast<Condition*>(m->object_id());
	condition->broadcast();
}

// Display
void Skeleton::display_constructor(Message * m) {
	Display();
}

void Skeleton::display_clear(Message * m) {
	Display::clear();
}

void Skeleton::display_putc(Message * m) {
	char c = *reinterpret_cast<char*>(m->param1());
```

```
	Display::putc(c);
}

void Skeleton::display_puts(Message * m) {
	const char s = *reinterpret_cast<char*>(m->param1());
	Display::puts(&s);
}

void Skeleton::display_geometry(Message * m) {
	int * lines = reinterpret_cast<int*>(m->param1());
	int * columns = reinterpret_cast<int*>(m->param2());
	Display::geometry(lines, columns);
}

void Skeleton::display_position_1(Message * m) {
	int * line = reinterpret_cast<int*>(m->param1());
	int * column = reinterpret_cast<int*>(m->param2());
	Display::geometry(line, column);
}

void Skeleton::display_position_2(Message * m) {
	int line = *reinterpret_cast<int*>(m->param1());
	int column = *reinterpret_cast<int*>(m->param2());
	Display::geometry(&line, &column);
}

// Handler
void Skeleton::handler_constructor(Message * m) {
	Function * h = reinterpret_cast<Function*>(m->param1());
	Function_Handler * handler = new (SYSTEM) Function_Handler(h);
	m->return_value(reinterpret_cast<void *>(&handler));
}

void Skeleton::handler_destructor(Message * m) {
	Function_Handler * handler = reinterpret_cast<Function_Handler*>(m->object_id());
	delete handler;
}

// Mutex
```

```
void Skeleton::mutex_constructor(Message * m) {
	Mutex * mutex = new (SYSTEM) Mutex();
	m->return_value(reinterpret_cast<void *>(&mutex));
}

void Skeleton::mutex_destructor(Message * m) {
	Mutex * mutex = reinterpret_cast<Mutex*>(m->object_id());
	delete mutex;
}

void Skeleton::mutex_lock(Message * m) {
	Mutex * mutex = reinterpret_cast<Mutex*>(m->object_id());
	mutex->lock();
}

void Skeleton::mutex_unlock(Message * m) {
	Mutex * mutex = reinterpret_cast<Mutex*>(m->object_id());
	mutex->unlock();
}

// Segment
void Skeleton::segment_constructor_1(Message * m) {
	unsigned int bytes = *reinterpret_cast<unsigned int*>(m->param1());
	Flags flags = *reinterpret_cast<Flags*>(m->param2());
	Segment * segment = new (SYSTEM) Segment(bytes, flags);
	m->return_value(reinterpret_cast<void *>(segment));
}

void Skeleton::segment_constructor_2(Message * m) {
	Phy_Addr phy_addr = reinterpret_cast<Phy_Addr*>(m->param2());
	unsigned int bytes = *reinterpret_cast<unsigned int*>(m->param2());
	Flags flags = *reinterpret_cast<Flags*>(m->param3());
	Segment * segment = new (SYSTEM) Segment(phy_addr, bytes, flags);
	m->return_value(reinterpret_cast<void *>(segment));
}

void Skeleton::segment_destructor(Message * m) {
	Segment * segment = reinterpret_cast<Segment*>(m->object_id());
	delete segment;
```

```
}

void Skeleton::segment_size(Message * m) {
	Segment * segment = reinterpret_cast<Segment*>(m->object_id());
	segment->size();
}

void Skeleton::segment_phy_address(Message * m) {
	Segment * segment = reinterpret_cast<Segment*>(m->object_id());
	segment->phy_address();
}

void Skeleton::segment_resize(Message * m) {
	Segment * segment = reinterpret_cast<Segment*>(m->object_id());
	int amount = *reinterpret_cast<int*>(m->param1());
	segment->resize(amount);
}

// Semaphore
void Skeleton::semaphore_constructor(Message * m){
	int v = reinterpret_cast<int>(m->param1());
	Semaphore * sem = new (SYSTEM) Semaphore(v);
	m->return_value(reinterpret_cast<void *>(&sem));
}

void Skeleton::semaphore_destructor(Message * m){
	Semaphore * sem = reinterpret_cast<Semaphore*>(m->object_id());
	delete sem;
}

void Skeleton::semaphore_p(Message * m){
	Semaphore * sem = reinterpret_cast<Semaphore*>(m->object_id());
	sem->p();
}

void Skeleton::semaphore_v(Message * m){
	Semaphore * sem = reinterpret_cast<Semaphore*>(m->object_id());
	sem->v();
}
```

```
// Task
void Skeleton::task_constructor(Message * m) {
       Segment p1 = *reinterpret_cast<Segment*>(m->param1());
       Segment p2 = *reinterpret_cast<Segment*>(m->param2());
       Task * task = new (SYSTEM) Task(p1, p2);
       m-> return_value(reinterpret_cast<void *>(&task));
}

void Skeleton::task_destructor(Message * m) {
       Task * task = reinterpret_cast<Task*>(m->object_id());
       delete task;
}

void Skeleton::task_address_space(Message * m) {
       Task * task = reinterpret_cast<Task*>(m->object_id());
       Address_Space * address_space = task->address_space();
       m->return_value(reinterpret_cast<void *> (address_space));
}

void Skeleton::task_code_segment(Message * m) {
       Task * task = reinterpret_cast<Task*>(m->object_id());
       Segment * cs = const_cast<Segment *>(task->code_segment());
       m->return_value(reinterpret_cast<void *>(cs));
}

void Skeleton::task_code(Message * m) {
       Task * task = reinterpret_cast<Task*>(m->object_id());
       CPU_Common::Log_Addr code = task->code();
       m->return_value(reinterpret_cast<void *>(&code));
}

void Skeleton::task_data_segment(Message * m) {
       Task * task = reinterpret_cast<Task*>(m->object_id());
       Segment * ds = const_cast<Segment *>(task->data_segment());
       m->return_value(reinterpret_cast<void *>(ds));
}

void Skeleton::task_data(Message * m) {
```

```
	Task * task = reinterpret_cast<Task*>(m->object_id());
	CPU_Common::Log_Addr data = task->data();
	m->return_value(reinterpret_cast<void *>(&data));
}

void Skeleton::task_self(Message * m) {
	Task * task = Task::self();
	m->return_value(reinterpret_cast<void *>(task));
}

	// Thread
template<typename T>
void Skeleton::thread_constructor_1(Message * m) {
	T p1 = *reinterpret_cast<T*>(m->param1());
	Thread * thread = new (SYSTEM) Thread(p1);
	m-> return_value(reinterpret_cast<void *>(&thread));
}

template<typename T1, typename T2>
void Skeleton::thread_constructor_2(Message * m) {
	T1 p1 = *reinterpret_cast<T1*>(m->param1());
	T2 p2 = *reinterpret_cast<T2*>(m->param2());
	Thread * thread = new (SYSTEM) Thread(p1, p2);
	m-> return_value(reinterpret_cast<void *>(&thread));
}

template<typename T1, typename T2, typename T3>
void Skeleton::thread_constructor_3(Message * m) {
	T1 p1 = *reinterpret_cast<T1*>(m->param1());
	T2 p2 = *reinterpret_cast<T2*>(m->param2());
	T3 p3 = *reinterpret_cast<T3*>(m->param3());
	Thread * thread = new (SYSTEM) Thread(p1, p2, p3);
	m-> return_value(reinterpret_cast<void *>(&thread));
}

template<typename T1, typename T2, typename T3, typename T4>
void Skeleton::thread_constructor_4(Message * m) {
	T1 p1 = *reinterpret_cast<T1*>(m->param1());
	T2 p2 = *reinterpret_cast<T2*>(m->param2());
```

```
	T3 p3 = *reinterpret_cast<T3*>(m->param3());
	T4 p4 = *reinterpret_cast<T4*>(m->param4());
	Thread * thread = new (SYSTEM) Thread(p1, p2, p3, p4);
	m-> return_value(reinterpret_cast<void *>(&thread));
}

template<typename T1, typename T2, typename T3, typename T4, typename T5>
void Skeleton::thread_constructor_5(Message * m) {
	T1 p1 = *reinterpret_cast<T1*>(m->param1());
	T2 p2 = *reinterpret_cast<T2*>(m->param2());
	T3 p3 = *reinterpret_cast<T3*>(m->param3());
	T4 p4 = *reinterpret_cast<T4*>(m->param4());
	T5 p5 = *reinterpret_cast<T5*>(m->param5());
	Thread * thread = new (SYSTEM) Thread(p1, p2, p3, p4, p5);
	m-> return_value(reinterpret_cast<void *>(&thread));
}

template<typename T1, typename T2, typename T3, typename T4, typename T5, typename T6>
void Skeleton::thread_constructor_6(Message * m) {
	T1 p1 = *reinterpret_cast<T1*>(m->param1());
	T2 p2 = *reinterpret_cast<T2*>(m->param2());
	T3 p3 = *reinterpret_cast<T3*>(m->param3());
	T4 p4 = *reinterpret_cast<T4*>(m->param4());
	T5 p5 = *reinterpret_cast<T5*>(m->param5());
	T6 p6 = *reinterpret_cast<T6*>(m->param6());
	Thread * thread = new (SYSTEM) Thread(p1, p2, p3, p4, p5, p6);
	m-> return_value(reinterpret_cast<void *>(&thread));
}

template<typename T1, typename T2, typename T3, typename T4, typename T5, typename T6, typename T7>
void Skeleton::thread_constructor_7(Message * m) {
	T1 p1 = *reinterpret_cast<T1*>(m->param1());
	T2 p2 = *reinterpret_cast<T2*>(m->param2());
	T3 p3 = *reinterpret_cast<T3*>(m->param3());
	T4 p4 = *reinterpret_cast<T4*>(m->param4());
	T5 p5 = *reinterpret_cast<T5*>(m->param5());
	T6 p6 = *reinterpret_cast<T6*>(m->param6());
	T7 p7 = *reinterpret_cast<T7*>(m->param7());
	Thread * thread = new (SYSTEM) Thread(p1, p2, p3, p4, p5, p6, p7);
```

```
	m-> return_value(reinterpret_cast<void *>(&thread));
}

void Skeleton::thread_destructor(Message * m) {
	Thread * thread = reinterpret_cast<Thread*>(m->object_id());
	delete thread;
}

void Skeleton::thread_state(Message * m) {
	Thread * thread = reinterpret_cast<Thread*>(m->object_id());
	State state = thread->state();
	m->return_value(reinterpret_cast<void *>(&state));
}

void Skeleton::thread_priority_1(Message * m) {
	Thread * thread = reinterpret_cast<Thread*>(m->object_id());
	Priority priority = thread->priority();
	m->return_value(reinterpret_cast<void *>(&priority));
}

void Skeleton::thread_priority_2(Message * m) {
	Thread * thread = reinterpret_cast<Thread*>(m->object_id());
	Priority p = *reinterpret_cast<Priority*>(m->param1());
	thread->priority(p);
}

void Skeleton::thread_task(Message * m) {
	Thread * thread = reinterpret_cast<Thread*>(m->object_id());
	Task * task = reinterpret_cast<Task *>(thread->task());
	m->return_value(reinterpret_cast<void *>(task));
}

void Skeleton::thread_join(Message * m) {
	Thread * thread = reinterpret_cast<Thread*>(m->object_id());
	thread->join();
}

void Skeleton::thread_pass(Message * m) {
	Thread * thread = reinterpret_cast<Thread*>(m->object_id());
```

```
        thread->pass();
    }

    void Skeleton::thread_suspend(Message * m) {
        Thread * thread = reinterpret_cast<Thread*>(m->object_id());
        thread->suspend();
    }

    void Skeleton::thread_resume(Message * m) {
        Thread * thread = reinterpret_cast<Thread*>(m->object_id());
        thread->resume();
    }

    void Skeleton::thread_self(Message * m) {
        Thread * thread = Thread::self();
        m->return_value(reinterpret_cast<void *>(thread));
    }

    void Skeleton::thread_yield(Message * m) {
        Thread * thread = reinterpret_cast<Thread*>(m->object_id());
        thread->yield();
    }

    void Skeleton::thread_exit(Message * m) {
        int status = *reinterpret_cast<int *>(m->param1());
        Thread * thread = reinterpret_cast<Thread*>(m->object_id());
        thread->exit(status);
    }


__END_SYS
```

A função "call" é responsável por determinar qual função do sistema deve ser executada, com base no identificador de classe e de função passado pela mensagem. Uma vez determinada qual deve ser a função do sistema chamada, é passado para a função que efetivamente faz a chamada da função usando o objeto passado pela mensagem bem como os parâmetros.

A fim de evitar problemas de *name clash* devido a escolha de nomes para os stubs, foi criado um novo namespace em *config.h* para resolver a questão. Todos os stubs foram criados dentro do namespace APP e quando cria-se uma aplicação, usa-se o namespace EPOS::APP. Desta forma, o que está dentro de SYS não é visto por APP e vice-versa.

system/config.h

```
namespace EPOS {
    namespace S {
        namespace U {}
        using namespace U;
    }
}
namespace APP {}

#define __BEGIN_API             namespace EPOS {
#define __END_API               }
#define _API                    ::EPOS

#define __BEGIN_APP              namespace EPOS { namespace APP {
#define __END_APP                } }
#define __USING_APP              using namespace EPOS::APP;
#define _APP                            ::EPOS::APP

#define __BEGIN_UTIL            namespace EPOS { namespace S { namespace U {
#define __END_UTIL              }}}
#define __USING_UTIL            using namespace S::U;
#define _UTIL                   ::EPOS::S::U

#define __BEGIN_SYS             namespace EPOS { namespace S {
#define __END_SYS               }}
#define __USING_SYS             using namespace EPOS::S;
#define _SYS                    ::EPOS::S
```

Além disso, foi criado o “app_types” para conter as assinaturas dos métodos das funções do sistema e dessa forma a aplicação trabalhar como se tivesse acesso direto a tais funções ainda que tenha que passar pelo stub. Isso pode ser visto a seguir:

app_types.h

```
#ifndef __app_types_h
#define __app_types_h

#include <system/config.h>
#include <tsc.h>
#include <rtc.h>

__BEGIN_API
```

```
        typedef RTC::Microsecond Microsecond;
        typedef TSC::Hertz Hertz;
        typedef CPU::Phy_Addr Phy_Addr;
    typedef CPU::Log_Addr Log_Addr;
        typedef MMU::Flags Flags;
    typedef CPU::Phy_Addr Phy_Addr;
    typedef void (Function)();

namespace Class {
        enum { ADDRESS_SPACE,
                ALARM,
                CONDITION,
                DISPLAY,
                HANDLER,
                MUTEX,
                SEGMENT,
                SEMAPHORE,
                TASK,
                THREAD };
}

namespace Method {

        namespace Address_Space {
                enum { CONSTRUCTOR_1,
                        CONSTRUCTOR_2,
                        DESTRUCTOR,
                        ATTACH_1,
                        ATTACH_2,
                        DETACH,
                        PHYSICAL };
        }

        namespace Alarm {
                enum { CONSTRUCTOR,
                        DESTRUCTOR,
                        FREQUENCY,
                        DELAY };
        }

        namespace Condition {
                enum { CONSTRUCTOR,
                        DESTRUCTOR,
                        WAIT,
                        SIGNAL,
                        BROADCAST };
        }

        namespace Display {
                enum { CONSTRUCTOR,
                        CLEAR,
                        PUTC,
                        PUTS,
                        GEOMETRY,
                        POSITION_1,
```

```
                POSITION_2 };
}

namespace Handler {
        enum { CONSTRUCTOR,
                DESTRUCTOR };
}

namespace Mutex {
        enum { CONSTRUCTOR,
                DESTRUCTOR,
                LOCK,
                UNLOCK };
}

namespace Segment {
        enum { CONSTRUCTOR_1,
                CONSTRUCTOR_2,
                DESTRUCTOR,
                SIZE,
                PHY_ADDRESS,
                RESIZE };
}

namespace Semaphore {
        enum { CONSTRUCTOR,
                DESTRUCTOR,
                P,
                V };
}

namespace Task {
        enum { CONSTRUCTOR,
                DESTRUCTOR,
                ADDRESS_SPACE,
                CODE_SEGMENT,
                CODE,
                DATA_SEGMENT,
                DATA,
                SELF };
}

namespace Thread {
        enum { CONSTRUCTOR_1,
                CONSTRUCTOR_2,
                CONSTRUCTOR_3,
                CONSTRUCTOR_4,
                CONSTRUCTOR_5,
                CONSTRUCTOR_6,
                CONSTRUCTOR_7,
                DESTRUCTOR,
                STATE,
                PRIORITY_1,
                PRIORITY_2,
                TASK,
```

```
                        JOIN,
                        PASS,
                        SUSPEND,
                        RESUME,
                        SELF,
                        YIELD,
                        EXIT };
        }
}

__END_API

#endif
```

STUBS

```
#ifndef __address_space_h
#define __address_space_h

#include <app_types.h>
#include <stub/message.h>
#include <stub/skeleton.h>

__BEGIN_APP

class Address_Space {

public:

    Address_Space() {
            message = new Message();
       message->class_id(Class::ADDRESS_SPACE);
       message->method_id(Method::Address_Space::CONSTRUCTOR_1);
       Skeleton::call(message);
       _obj_id = message->return_value();
    };

       Address_Space(MMU::Page_Directory * pd) {
            message = new Message();
       message->class_id(Class::ADDRESS_SPACE);
       message->method_id(Method::Address_Space::CONSTRUCTOR_2);
            message->param1((void*) &pd);
       Skeleton::call(message);
       _obj_id = message->return_value();
    };

    ~Address_Space(){
       message->class_id(Class::ADDRESS_SPACE);
       message->method_id(Method::Address_Space::DESTRUCTOR);
       message->object_id(_obj_id);
       Skeleton::call(message);
            delete message;
    };
```

```
    Log_Addr attach(const Segment & seg) {
       message->class_id(Class::ADDRESS_SPACE);
       message->method_id(Method::Address_Space::ATTACH_1);
       message->object_id(_obj_id);
       Skeleton::call(message);
              Log_Addr address = reinterpret_cast<Log_Addr*>(message->return_value());
              return address;
    };

       Log_Addr attach(const Segment & seg, Log_Addr addr) {
       message->class_id(Class::ADDRESS_SPACE);
       message->method_id(Method::Address_Space::ATTACH_2);
       message->object_id(_obj_id);
       Skeleton::call(message);
              Address_Space * address_space = reinterpret_cast<Address_Space
*>(message->return_value());
              return address_space;
    };

    void detach(const Segment & seg){
       message->class_id(Class::ADDRESS_SPACE);
       message->method_id(Method::Address_Space::DETACH);
       message->object_id(_obj_id);
       Skeleton::call(message);
    };

       Phy_Addr physical(Log_Addr address) {
       message->class_id(Class::ADDRESS_SPACE);
       message->method_id(Method::Address_Space::PHYSICAL);
       message->object_id(_obj_id);
       Skeleton::call(message);
              Address_Space * address_space = reinterpret_cast<Address_Space
*>(message->return_value());
              return address_space;
    };

private:
   void * _obj_id;
   Message * message;
};

__END_APP
#endif

#ifndef __alarm_h
#define __alarm_h

#include <app_types.h>
#include <stub/message.h>
#include <stub/skeleton.h>

__BEGIN_APP

class Alarm {
```

```
public:

    Alarm(const Microsecond &time, Handler * handler, int times = 1) {
            message = new Message();
       message->class_id(Class::ALARM);
       message->method_id(Method::Alarm::CONSTRUCTOR);
            message->param1((void*) &time);
            message->param2((void*) &handler);
            message->param3((void*) &times);
       Skeleton::call(message);
       _obj_id = message->return_value();
    };

       ~Alarm(){
       message->class_id(Class::ALARM);
       message->method_id(Method::Alarm::DESTRUCTOR);
       message->object_id(_obj_id);
       Skeleton::call(message);
            delete message;
    };

    static Hertz frequency() {
            Message message = Message();
       message.class_id(Class::ALARM);
       message.method_id(Method::Alarm::FREQUENCY);
       Skeleton::call(&message);
            Hertz frequency = reinterpret_cast<Hertz> (message.return_value());
            return frequency;
    };

       static void delay(const Microsecond & time) {
            Message message = Message();
       message.class_id(Class::ALARM);
       message.method_id(Method::Alarm::DELAY);
       Skeleton::call(&message);
    };

private:
      void * _obj_id;
      Message * message;
};

__END_APP
#endif

#ifndef __address_space_h
#define __address_space_h

#include <app_types.h>
#include <stub/message.h>
#include <stub/skeleton.h>

__BEGIN_APP

class Condition {
```

```
public:

    Condition() {
             message = new Message();
       message->class_id(Class::CONDITION);
       message->method_id(Method::Condition::CONSTRUCTOR);
       Skeleton::call(message);
       _obj_id = message->return_value();
    };

    ~Condition(){
       message->class_id(Class::CONDITION);
       message->method_id(Method::Condition::DESTRUCTOR);
       message->object_id(_obj_id);
       Skeleton::call(message);
             delete message;
    };

    void wait() {
       message->class_id(Class::CONDITION);
       message->method_id(Method::Condition::WAIT);
       message->object_id(_obj_id);
       Skeleton::call(message);
    };

    void signal(){
       message->class_id(Class::CONDITION);
       message->method_id(Method::Condition::SIGNAL);
       message->object_id(_obj_id);
       Skeleton::call(message);
    };

       void broadcast(){
       message->class_id(Class::CONDITION);
       message->method_id(Method::Condition::BROADCAST);
       message->object_id(_obj_id);
       Skeleton::call(message);
    };

private:
   void * _obj_id;
   Message * message;
};

__END_APP
#endif

#ifndef __display_h
#define __display_h

#include <app_types.h>
#include <stub/message.h>
#include <stub/skeleton.h>
```

```
__BEGIN_APP

class Display {

public:
	Display() {
		message = new Message();
		message->class_id(Class::DISPLAY);
	message->method_id(Method::Display::CONSTRUCTOR);
	message->object_id(_obj_id);
	Skeleton::call(message);
	}

    static void clear() {
       Message message = Message();
       message.class_id(Class::DISPLAY);
       message.method_id(Method::Display::CLEAR);
       Skeleton::call(&message);
    }

       static void putc() {
		Message message = Message();
       message.class_id(Class::DISPLAY);
       message.method_id(Method::Display::PUTC);
       Skeleton::call(&message);
    }

       static void puts(const char * s) {
		Message message = Message();
       message.class_id(Class::DISPLAY);
       message.method_id(Method::Display::PUTS);
       message.param1((void*) s);
       Skeleton::call(&message);
       }

       static void geometry(int * lines, int * columns) {
		Message message = Message();
       message.class_id(Class::DISPLAY);
       message.method_id(Method::Display::GEOMETRY);
       message.param1((void*) &lines);
		message.param2((void*) &columns);
       Skeleton::call(&message);
    }

       static void position(int * line, int * column) {
		Message message = Message();
       message.class_id(Class::DISPLAY);
       message.method_id(Method::Display::POSITION_1);
       message.param1((void*) &line);
		message.param2((void*) &column);
       Skeleton::call(&message);
    }

       static void position(int line, int column) {
		Message message = Message();
```

```
        message.class_id(Class::DISPLAY);
        message.method_id(Method::Display::POSITION_2);
        message.param1((void*) line);
               message.param2((void*) column);
        Skeleton::call(&message);
    }

private:
        void * _obj_id;
        Message * message;
};

__END_APP
#endif

#ifndef __handler_h
#define __handler_h

#include <app_types.h>
#include <stub/message.h>
#include <stub/skeleton.h>

__BEGIN_APP

class Function_Handler {

public:
        Function_Handler(Function * h) {
               message = new Message();
               message->class_id(Class::HANDLER);
               message->method_id(Method::Handler::CONSTRUCTOR);
               message->param1((void*) h);
               Skeleton::call(message);
               _obj_id = message->return_value();
        }

        ~Function_Handler(){
               message->class_id(Class::HANDLER);
               message->object_id(_obj_id);
               Skeleton::call(message);
               delete message;
        }

private:
        void * _obj_id;
        Message * message;
};

__END_APP
#endif

#ifndef __mutex_h
#define __mutex_h
```

```
#include <app_types.h>
#include <stub/message.h>
#include <stub/skeleton.h>

__BEGIN_APP

class Mutex {

public:
	Mutex() {
		message = new Message();
	message->class_id(Class::MUTEX);
	message->method_id(Method::Mutex::CONSTRUCTOR);
	Skeleton::call(message);
	_obj_id = message->return_value();
	}

	~Mutex() {
		message->class_id(Class::MUTEX);
	message->method_id(Method::Mutex::DESTRUCTOR);
	message->object_id(_obj_id);
	Skeleton::call(message);
		delete message;
	}

	void lock() {
		message->class_id(Class::MUTEX);
	message->method_id(Method::Mutex::LOCK);
	message->object_id(_obj_id);
	Skeleton::call(message);
	}

	void unlock() {
		message->class_id(Class::MUTEX);
	message->method_id(Method::Mutex::UNLOCK);
	message->object_id(_obj_id);
	Skeleton::call(message);
	}

private:
	void * _obj_id;
	Message * message;
};

__END_APP
#endif

#ifndef __segment_h
#define __segment_h

#include <app_types.h>
#include <stub/message.h>
#include <stub/skeleton.h>

__BEGIN_APP
```

```
class Segment {

public:
       Segment(unsigned int bytes, Flags flags = Flags::APP) {
              message = new Message();
       message->class_id(Class::SEGMENT);
       message->method_id(Method::Segment::CONSTRUCTOR_1);
       message->param1((void*) &bytes);
       message->param1((void*) &flags);
       Skeleton::call(message);
       _obj_id = message->return_value();
    }

       Segment(Phy_Addr phy_addr, unsigned int bytes, Flags flags = Flags::APP) {
              message = new Message();
       message->class_id(Class::SEGMENT);
       message->method_id(Method::Segment::CONSTRUCTOR_2);
       message->param1((void*) &phy_addr);
       message->param1((void*) &bytes);
       message->param1((void*) &flags);
       Skeleton::call(message);
       _obj_id = message->return_value();
    }

    ~Segment(){
       message->class_id(Class::SEGMENT);
       message->method_id(Method::Segment::DESTRUCTOR);
       message->object_id(_obj_id);
       Skeleton::call(message);
              delete message;
    }

       unsigned int size() const{
       message->class_id(Class::SEGMENT);
       message->method_id(Method::Segment::SIZE);
       message->object_id(_obj_id);
       Skeleton::call(message);
              unsigned int size = reinterpret_cast<int*>(message->return_value());
              return size;
    }

    Phy_Addr phy_address() const{
       message->class_id(Class::SEGMENT);
       message->method_id(Method::Segment::PHY_ADDRESS);
       message->object_id(_obj_id);
       Skeleton::call(message);
              Phy_Addr phy_addr = reinterpret_cast<Phy_Addr*>(message-
>return_value());
              return phy_addr;
    }

       int resize(int amount){
       message->class_id(Class::SEGMENT);
       message->method_id(Method::Segment::PHY_ADDRESS);
```

```
        message->object_id(_obj_id);
                message->param1(&amount);
        Skeleton::call(message);
                int size = reinterpret_cast<int*>(message->return_value());
                return size;
    }

private:
        void * _obj_id;
        Message * message;
};

__END_APP
#endif

#ifndef __semaphore_h
#define __semaphore_h

#include <app_types.h>
#include <stub/message.h>
#include <stub/skeleton.h>

__BEGIN_APP

class Semaphore {

public:
    Semaphore(int v = 1){
                message = new Message();
        message->class_id(Class::SEMAPHORE);
        message->method_id(Method::Semaphore::CONSTRUCTOR);
        message->param1((void*) &v);
        Skeleton::call(message);
        _obj_id = message->return_value();
    }

    ~Semaphore(){
        message->class_id(Class::SEMAPHORE);
        message->method_id(Method::Semaphore::DESTRUCTOR);
        message->object_id(_obj_id);
        Skeleton::call(message);
                delete message;
    }

    void p(){
        message->class_id(Class::SEMAPHORE);
        message->method_id(Method::Semaphore::P);
        message->object_id(_obj_id);
        Skeleton::call(message);
    }

    void v(){
        message->class_id(Class::SEMAPHORE);
        message->method_id(Method::Semaphore::V);
        message->object_id(_obj_id);
```

```
      Skeleton::call(message);
    }

private:
      void * _obj_id;
      Message * message;
};

__END_APP
#endif

#ifndef __task_h
#define __task_h

#include <app_types.h>
#include <stub/message.h>
#include <stub/skeleton.h>

__BEGIN_APP

class Task {

public:

    Task(const Segment &cs, const Segment &ds) {
            message = new Message();
      message->class_id(Class::TASK);
      message->method_id(Method::Task::CONSTRUCTOR);
      message->param1((void*) &cs);
      message->param2((void*) &ds);
      Skeleton::call(message);
      _obj_id = message->return_value();
    }

    ~Task(){
      message->class_id(Class::TASK);
      message->method_id(Method::Task::DESTRUCTOR);
      message->object_id(_obj_id);
      Skeleton::call(message);
            delete message;
    }

    Address_Space * address_space() {
      message->class_id(Class::TASK);
      message->method_id(Method::Task::ADDRESS_SPACE);
      message->object_id(_obj_id);
      Skeleton::call(message);
            Address_Space * address_space = reinterpret_cast<Address_Space
*>(message->return_value());
            return address_space;
    }

    const Segment * code_segment(){
      message->class_id(Class::TASK);
      message->method_id(Method::Task::CODE_SEGMENT);
```

```
        message->object_id(_obj_id);
        Skeleton::call(message);
                Segment * cs = reinterpret_cast<Segment *>(message->return_value());
                return cs;
    }

        const Segment * data_segment(){
        message->class_id(Class::TASK);
        message->method_id(Method::Task::DATA_SEGMENT);
        message->object_id(_obj_id);
        Skeleton::call(message);
                Segment * ds = reinterpret_cast<Segment *>(message->return_value());
                return ds;
    }

        Log_Addr code(){
        message->class_id(Class::TASK);
        message->method_id(Method::Task::CODE);
        message->object_id(_obj_id);
        Skeleton::call(message);
                Log_Addr code = reinterpret_cast<Log_Addr*>(message->return_value());
                return code;
    }


        Log_Addr data(){
        message->class_id(Class::TASK);
        message->method_id(Method::Task::DATA);
        message->object_id(_obj_id);
        Skeleton::call(message);
                Log_Addr data = reinterpret_cast<Log_Addr*>(message->return_value());
                return data;
    }

        static Task * self() {
                Message message = Message();
                message.class_id(Class::TASK);
        message.method_id(Method::Task::SELF);
        Skeleton::call(&message);
                Task * task = reinterpret_cast<Task *>(message.return_value());
                return task;
        }

private:
        void * _obj_id;
        Message * message;
};

__END_APP
#endif

#ifndef __thread_h
#define __thread_h

#include <app_types.h>
```

```
#include <stub/message.h>
#include <stub/skeleton.h>
#include <scheduler.h>

__BEGIN_APP

class Thread {

private:
       enum State {
         RUNNING,
         READY,
         SUSPENDED,
         WAITING,
         FINISHING
    };

       typedef Scheduling_Criteria::Priority Priority;

public:
       template<typename T>
       Thread(T t1) {
              message = new Message();
       message->class_id(Class::THREAD);
       message->method_id(Method::Thread::CONSTRUCTOR_1);
              message->param1((void *) t1);
       Skeleton::call(message);
       _obj_id = message->return_value();
       }

       template<typename T1, typename T2>
       Thread(T1 t1, T2 t2) {
              message = new Message();
       message->class_id(Class::THREAD);
       message->method_id(Method::Thread::CONSTRUCTOR_2);
              message->param1((void *) t1);
              message->param2((void *) t2);
       Skeleton::call(message);
       _obj_id = message->return_value();
       }

       template<typename T1, typename T2, typename T3>
       Thread(T1 t1, T2 t2, T3 t3) {
              message = new Message();
       message->class_id(Class::THREAD);
       message->method_id(Method::Thread::CONSTRUCTOR_3);
       message->param1((void *) t1);
              message->param2((void *) t2);
              message->param3((void *) t3);
       Skeleton::call(message);
       _obj_id = message->return_value();
       }

       template<typename T1, typename T2, typename T3, typename T4>
       Thread(T1 t1, T2 t2, T3 t3, T4 t4) {
```

```
		message = new Message();
	message->class_id(Class::THREAD);
	message->method_id(Method::Thread::CONSTRUCTOR_4);
		message->param1((void *) t1);
		message->param2((void *) t2);
		message->param3((void *) t3);
		message->param4((void *) t4);
	Skeleton::call(message);
	_obj_id = message->return_value();
	}

	template<typename T1, typename T2, typename T3, typename T4, typename T5>
	Thread(T1 t1, T2 t2, T3 t3, T4 t4, T5 t5) {
		message = new Message();
	message->class_id(Class::THREAD);
	message->method_id(Method::Thread::CONSTRUCTOR_5);
		message->param1((void *) t1);
		message->param2((void *) t2);
		message->param3((void *) t3);
		message->param4((void *) t4);
		message->param5((void *) t5);
	Skeleton::call(message);
	_obj_id = message->return_value();
	}

	template<typename T1, typename T2, typename T3, typename T4, typename T5,
typename T6>
	Thread(T1 t1, T2 t2, T3 t3, T4 t4, T5 t5, T6 t6) {
		message = new Message();
	message->class_id(Class::THREAD);
	message->method_id(Method::Thread::CONSTRUCTOR_6);
		message->param1((void *) t1);
		message->param2((void *) t2);
		message->param3((void *) t3);
		message->param4((void *) t4);
		message->param5((void *) t5);
		message->param6((void *) t6);
	Skeleton::call(message);
	_obj_id = message->return_value();
	}

	template<typename T1, typename T2, typename T3, typename T4, typename T5,
typename T6, typename T7>
	Thread(T1 t1, T2 t2, T3 t3, T4 t4, T5 t5, T6 t6, T7 t7) {
		message = new Message();
	message->class_id(Class::THREAD);
	message->method_id(Method::Thread::CONSTRUCTOR_7);
		message->param1((void *) t1);
		message->param2((void *) t2);
		message->param3((void *) t3);
		message->param4((void *) t4);
		message->param5((void *) t5);
		message->param6((void *) t6);
		message->param7((void *) t7);
	Skeleton::call(message);
```

```
_obj_id = message->return_value();
}

~Thread() {
	message->class_id(Class::THREAD);
message->method_id(Method::Thread::DESTRUCTOR);
message->object_id(_obj_id);
Skeleton::call(message);
	delete message;
}

const volatile State & state() const {
	message->class_id(Class::THREAD);
message->method_id(Method::Thread::STATE);
message->object_id(_obj_id);
Skeleton::call(message);
	return *reinterpret_cast<State *>(message->return_value());
}

const volatile Priority & priority() const {
	message->class_id(Class::THREAD);
message->method_id(Method::Thread::PRIORITY_1);
message->object_id(_obj_id);
Skeleton::call(message);
	return *reinterpret_cast<Priority *>(message->return_value());
}

void priority(const Priority & p) {
	message->class_id(Class::THREAD);
message->method_id(Method::Thread::PRIORITY_2);
message->object_id(_obj_id);
message->param1((void*) &p);
Skeleton::call(message);
}

Task * task() const {
	message->class_id(Class::THREAD);
message->method_id(Method::Thread::TASK);
message->object_id(_obj_id);
Skeleton::call(message);
	Task * task = reinterpret_cast<Task *>(message->return_value());
	return const_cast<Task*>(task);
}

int join() {
	message->class_id(Class::THREAD);
message->method_id(Method::Thread::JOIN);
message->object_id(_obj_id);
Skeleton::call(message);
	int join = *reinterpret_cast<int *>(message->return_value());
	return join;
}

void pass() {
	message->class_id(Class::THREAD);
```

```
        message->method_id(Method::Thread::PASS);
        message->object_id(_obj_id);
        Skeleton::call(message);
        }

        void suspend() {
                message->class_id(Class::THREAD);
        message->method_id(Method::Thread::SUSPEND);
        message->object_id(_obj_id);
        Skeleton::call(message);
        }

        void resume() {
                message->class_id(Class::THREAD);
        message->method_id(Method::Thread::RESUME);
        message->object_id(_obj_id);
        Skeleton::call(message);
        }

        static Thread & self() {
                Message message = Message();
                message.class_id(Class::THREAD);
        message.method_id(Method::Thread::SELF);
        Skeleton::call(&message);
                return *reinterpret_cast<Thread *>(message.return_value());
        }

        static void yield() {
                Message message = Message();
                message.class_id(Class::THREAD);
        message.method_id(Method::Thread::YIELD);
        Skeleton::call(&message);
        }

        static void exit(int status) {
                Message message = Message();
                message.class_id(Class::THREAD);
        message.method_id(Method::Thread::EXIT);
        message.param1((void*) status);
        Skeleton::call(&message);
        }

private:
        void * _obj_id;
        Message * message;
};

__END_APP
#endif
```