Coyotos Core Domain Interfaces

Version 0.1 (*in progress*)

Jonathan S. Shapiro, Ph.D., Jonathan W. Adams
The EROS Group, LLC

December 8, 2007

Contents

Acknowledgments

Many elements of the utility domain design described here are derived from earlier work on the KeyKOS [1] and EROS [2] operating systems.

Comments and suggestions concerning these interfaces are welcome. They should be filed as bug or enhancement requests in the Coyotos issue tracking system and/or discussed on the `coyotos-dev` electronic mailing list. In order to send, you must be subscribed to the list. The subscription interface may be found at:

> `http://www.coyotos.org/mailman/listinfo/coyotos-dev`.

In order to keep the mail archives readable, we ask that you send only “plain text” emails.

coyotos.Builder

Interface `coyotos.Builder`

Derivation:

```
coyotos.Cap
  coyotos.Verifier
    coyotos.Constructor
      coyotos.Builder
```

Synopsis: Constructor instantiation interface.

The builder interface enables developers and installers to populate new constructors with their initial capabilities. The constructor life cycle begins with a builder capability. When sealed, the builder interface returns a constructor interface, and no longer permits inseration of new capabilities.

As each capability is added to the constructor instance, the constructor computes whether each added capability satisfies the confinement requirements. If all capabilities installed in the constructor satisfy the confinement property, the yield of the constructor is confined and the constructor will certify this to its clients.

The builder capability implements the operations of the constructor capability, allowing the developer to perform test instantiations before sealing the constructor.

Exceptions

Sealed Raised on insert operations when constructor has been sealed.

Operations

setHandler Insert a capability to be used as the yield's external fault handler.

```
void setHandler(
    Cap handler);
```

Raises: `Sealed`

If the installed capability is a constructor capability, a new address space will be fabricated from this constructor for each newly instantiatied process.

setSpace Insert a capability to be used as the yield's initial addres space.

```
void setSpace(
    Cap space);
```

Raises: `Sealed`

If the installed capability is a constructor capability, a new address space will be fabricated from this constructor for each newly instantiatied process.

setPC Set the initial program counter for the yield.

```
void setPC(
    coyaddr_t pc);
```

Raises: `Sealed`

setTool Insert a tool capability for the process.

```
void setTool(
    uint32 slot,
    Cap c);
```

Raises: `Sealed`

seal Seal the constructor, returning a constructor capability.

```
Constructor seal();
```

coyotos.Constructor

Interface `coyotos.Constructor`

Derivation:

```
coyotos.Cap
  coyotos.Verifier
    coyotos.Constructor
```

Synopsis: Process instantiation interface.

The constructor is responsible for fabricating new process instances from application images. It also implements a test of confinement.

A confined process is one that cannot (initially) communicate with any party other than its creator. The constructor tests confinement by examining the capabilities initially held by the process. As each capability is installed, the constructor determines whether this capability is “safe” (meaning that it does not permit outward communication) or a “hole.” Prior to fabricating a new process instance, a client can ask the fabricating constructor whether its yield would be confined.

Constructors also implement a query to determine whether a process is its yield. This allows a client to determine whether a process alleging to be an instance of some binary X was in fact created by the constructor of X instances.

The constructor itself is part of the system-wide trusted computing base. Its determination concerning process confinement can be relied upon *provided* the constructor itself is authentic. The authenticity of the constructor can be determined by asking the metaconstructor (the constructor of constructors) to authenticate its yield. In the normal Coyotos configuration, both the metaconstructor and the space bank verifier are readily available to applications as part of the standard runtime environment.

A constructor lifecycle proceeds in two phases. In the builder phase, it is possible to install new capabilities into the constructor (see `coyotos.builder`). The `coyotos.builder.seal`() operation produces the constructor capability. Once sealed, new capabilities cannot be added to the constructor.

Operations

isYieldConfined Return true exactly if the constructor’s yield is confined.

```
bool isYieldConfined();
```

create Create a new instance of the constructor’s yield.

```
Cap create(
    SpaceBank bank,
    Schedule sched,
    Cap runtime);
```

The created instance is allocated from the provided space bank and executes under the provided schedule.

`bank` should be a newly created bank; upon any failure, the bank will be destroyed.

`runtime` is passed to the newly created process as its runtime information key.

simpleCreate Create a new instance of the constructor's yield, using the default arguments.

```
Cap simpleCreate(
    Cap runtime,
    OUT Cap newBank);
```

Uses `CR_SELF`'s schedule and a new child of `CR_SPACEBANK` to invoke the constructor. `runtime` is passed through unchanged, and the new bank is returned through `newBank`.

Equivalent to:

```
tmp_schedule = coyotos.Process.getSlot(CR_SELF, cslot.schedule);
tmp_fullbank = coyotos.SpaceBank.createChild(CR_SPACEBANK);
try {
  tmp_c_bank = coyotos.SpaceBank.reduce(tmp_fullbank,
                                      restrictions.noRemove);
  ret =
    coyotos.Constructor.create(tmp_c_bank, tmp_schedule, runtime);
  newBank = tmp_fullbank;
  return ret;
} catch (e) {
  try { coyotos.Cap.destroy(tmp_fullbank); } catch () { }
  throw (e);
}
```

getVerifier Get a Verifier for this Constructor

```
Verifier getVerifier();
```

coyotos.ElfSpace

Interface `coyotos.ElfSpace`

Derivation:

```
coyotos.Cap
  coyotos.SpaceHandler
    coyotos.ElfSpace
```

Synopsis: `ElfSpace`

Exceptions

NoSpace No Virtual Address Space is available

Operations

setBreak Set the end of the heap, allowing further allocation

```
void setBreak(
    uint64 newBreak);
```

Raises: `NoSpace`

coyotos.IoStream

Interface `coyotos.IoStream`

Derivation:

```
coyotos.Cap
  coyotos.IoStream
```

Synopsis: Generic bi-directional I/O stream interface.

This is a first version of the I/O stream interface. It doesn't implement any I/O control operations, but it has the basic read write interface.

Constants

Name	Type	Value	Description
bufLimit	uint32	4096	

Type Definitions

chString Maximum length character buffer that can be transmitted or received in a single read/write operation.

```
typedef anon1 chString;
```

Exceptions

RequestWouldBlock

Closed

Operations

doRead

```
void doRead(
  uint32 length,
  OUT chString s);
```

Raises: `RequestWouldBlock`

getReadChannel Return the read-specific channel for this `IoStream`.

```
IoStream getReadChannel();
```

getWriteChannel Return the write-specific channel for this `IoStream`.

```
IoStream getWriteChannel();
```

doWrite
```
uint32 doWrite(
    chString s);
```

Raises: `RequestWouldBlock`

read Read `length` bytes into `chString` from stream.

```
void read(
    uint32 len,
    OUT chString s);
```

This declares a library-supplied wrapper that calls `doRead`() internally, hiding the details of the read blocking protocol.

write Write `chString` to stream.

```
uint32 write(
    chString s);
```

This declares a library-supplied wrapper that calls `doWrite`() internally, hiding the details of the read blocking protocol.

`coyotos.SpaceBank`

Interface `coyotos.SpaceBank`

Derivation:
`coyotos.Cap`
  `coyotos.SpaceBank`

Synopsis: Coyotos system storage allocator.

The space bank implements the storage allocation and deallocation service. To allocate an object, you invoke a space bank and request an object of a specified type. To destroy an object, you invoke the space bank providing a capability to the object and ask that the object designated by that capability be destroyed. The destroy operation requires that the object must currently be "owned" by the bank being invoked.

Space banks also provide bulk destruction services. Destroying a space bank via the `destroyBank`() operation causes all objects allocated from that bank to be destroyed, including any sub-banks. The `removeBank`() operation destroys the space bank itself, but ownership of the objects currently owned by that space bank (including sub-banks) is transferred to the parent of the removed bank.

In a typical running system, there are many space bank capabilities. Typically, all of these capabilities are implemented by a single process known as the "prime bank." Space banks are arranged in a hierarchy rooted at the prime bank. New banks are formed by the `createChild` operation.

Space banks come in various restricted variants.

Exceptions

LimitReached A limit on the amount of usable space has been reached.

Enumerations

restrictions Reduced-authority space bank variants

Name	Type	Value	Description
noAlloc	uint32	1	Capability does not permit object or sub-bank allocation.
noFree	uint32	2	Capability does not permit object destruction.
noDestroy	uint32	4	Capability does not permit space bank destroy() or remove();
noQueryLimits	uint32	8	Capability will not disclose remaining limit bounds.
noChangeLimits	uint32	16	Capability does not permit alteration of bank limits.

Name	Type	Value	Description
noRemove	uint32	32	This bank cannot be destroyed by remove(). This restriction is appropriate when destruction of the bank should not permit allocated storage to survive. If a bank is being used by an untrusted subsystem, the remove() operation would permit the subsystem to conduct denial of resource attacks against its source of storage. Once storage has been inherited by a parent bank, the only means to destroy that storage is to destroy the parent bank. The mechanism of denial is to cause storage to be owned by a bank that contains other state that the holder cannot afford to destroy. This attack is prevented by imposing the `noRemove` restriction on the subsystem's bank.

Structures

limits Structure for limit information.

```
struct limits{
    uint64    byType[Range.obType.otNUM_TYPES];
};
```

Operations

alloc Allocate objects.

```
void alloc(
    Range.obType obType1,
    Range.obType obType2,
    Range.obType obType3,
    OUT Cap c1,
    OUT Cap c2,
    OUT Cap c3);
```

Raises: `LimitReachedNoAccess`

Allocates objects of type `obType1, obType2,` and `obType3,` returning them in `c1, c2, c3.` If `range.obType.otInvalid` is passed as a desired object type, the corresponding return capability will be `cap.null`. The operation is all or nothing; if three objects are requested but only two can be allocated, the `LimitReached` exception is raised.

An allocation performed on any bank is performed in parallel on all of its parent banks up to the prime bank. This means that the effective allocation limit is the *most constraining* limit applied by the invoked bank and all of its parent banks.

Raises `LimitReached` if this allocation would exceed the limits of the current bank or one of its parents.

Raises `NoAccess` if the capability has the `noAlloc` restriction.

free Free objects.

```
void free(
    uint32 count,
    Cap c1,
    Cap c2,
    Cap c3);
```

Raises: `LimitReachedNoAccess`

Releases `count` objects, returning them to the free storage pool. If the passed capabilities are not owned by this space bank, the `cap.RequestError` exception is raised. The passed capabilities must additionally satisfy the constraints of `coyotos.Range.rescind`.

If *any* of the passed capabilities fail to meet these requirements, the operation fails with a `cap.RequestError` exception with no action taken.

If the `count` is not in the range $0 < \text{count} < 4$, `cap.RequestError` is raised.

Raises `NoAccess` if the capability has the `noFree` restriction.

allocProcess Allocate a process.

```
Cap allocProcess(
    Cap brand);
```

Raises: `LimitReachedNoAccess`

Allocates a new process object, installing `brand` in the brand slot of the process. Processes can be allocated using alloc(), but doing so will not permit their brand slot to be populated.

A process allocated using `allocProcess`() should be deallocated using free().

destroyBankAndReturn Destroy the target bank as in destroy(), and reply to the specified `rcvr` capability with result `resultCode.`

```
void destroyBankAndReturn(
    Cap rcvr,
    exception_t resultCode);
```

If `resultCode` is `RC_OK`, the reply to `rcvr` will be a ”normal” (non-error) reply with no payload. Otherwise, it will be an exceptional reply with `resultCode` as the specified exception code.

Note that as with all spacebank replies, the reply will be non-blocking. if `rcvr` is not receiving, the reply will be lost.

If the operation fails, the failure is returned to the **caller**, not `rcvr.`

setLimits Set the limits imposed by this bank.

```
void setLimits(
    limits lims);
```

Raises: `NoAccess`

getLimits Retrieve the current limits imposed by this bank (directly).

```
limits getLimits();
```

Raises: `NoAccess`

getEffectiveLimits Retrieve the most constraining limits that apply to allocations from this bank, including parent limits.

```
limits getEffectiveLimits();
```

Raises: `NoAccess`

getUsage Retrieve the current usage of the bank

```
limits getUsage();
```

Raises: `NoAccess`

destroy *inherited from* **coyotos.Cap.destroy**

Destroy bank and storage.

```
void destroy();
```

In addition to deallocating the space bank itself, the destroy operation deallocates all storage that has been allocated from this bank.

remove Destroy this bank, transfering ownership of storage and sub-banks to the parent bank.

```
void remove();
```

Raises: `NoAccess`

createChild Create a child bank from the current bank.

```
SpaceBank createChild();
```

Raises: `NoAccessLimitReached`

verifyBank Verify that a bank is authentic.

```
bool verifyBank(
    SpaceBank bank);
```

Tests whether a capability references an official space bank. Note that to be able to usefully call this, you must have a capability that you know is a capability to an official space bank. The constructor holds such a capability and uses it to validate the bank that is passed for object construction. Applications can therefore bootstrap bank authentication from the initial bank provided through their constructor.

reduce Return bank capability with reduced authority.

```
SpaceBank reduce(
    restrictions mask);
```

Returns bank capability to the same bank with the additional restrictions specified by `mask.` These restrictions are applied in addition to any pre-existing restrictions on the invoked capability.

coyotos.SpaceHandler

Interface `coyotos.SpaceHandler`

Derivation:

```
coyotos.Cap
  coyotos.SpaceHandler
```

Synopsis: Address space handler.

Operations

getSpace Get the address space root for this space

```
AddressSpace getSpace();
```

coyotos.TermIoStream

Interface `coyotos.TermIoStream`

Derivation:

```
coyotos.Cap
  coyotos.IoStream
    coyotos.TermIoStream
```

Synopsis: Bi-directional I/O stream interface for serial streams.

The intent of this interface is to follow the `TERMIOS` specification of the Single `UNIX` Specification, Version 3, which can be found here. (free to read, registration required).

Constants

Name	Type	Value
NCCS	uint32	32

Type Definitions

tcflag_t `typedef uint32 tcflag_t;`

speed_t `typedef uint32 speed_t;`

Enumerations

	Name	Type	Value
iflag	**IGNBRK**	uint32	1
	BRKINT	uint32	2
	IGNPAR	uint32	4
	PARMRK	uint32	8
	INPCK	uint32	16
	ISTRIP	uint32	32
	INLCR	uint32	64
	IGNCR	uint32	128
	ICRNL	uint32	256
	IUCLC	uint32	512

Name	Type	Value
IXON	uint32	1024
IXANY	uint32	2048
IXOFF	uint32	4096
IMAXBEL	uint32	8192

oflag

Name	Type	Value
OPOST	uint32	1
OLCUC	uint32	2
ONLCR	uint32	4
OCRNL	uint32	8
ONOCR	uint32	16
ONLRET	uint32	32
OFILL	uint32	64
OFDEF	uint32	128
NLDLY	uint32	256
NL0	uint32	512
NL1	uint32	1024
CRDLY	uint32	2048
CR0	uint32	4096
CR1	uint32	8192
CR2	uint32	16384
CR3	uint32	32768
TABDLY	uint32	65536
TAB0	uint32	131072
TAB1	uint32	262144
TAB2	uint32	524288
TAB3	uint32	1048576
BSDLY	uint32	2097152
BS0	uint32	4194304
BS1	uint32	8388608
FFDLY	uint32	16777216
FF0	uint32	33554432
FF1	uint32	67108864

cflag

Name	Type	Value
B0	uint32	0
B50	uint32	1
B75	uint32	2
B110	uint32	3
B134	uint32	4
B150	uint32	5
B200	uint32	6
B300	uint32	7
B600	uint32	8
B1200	uint32	9
B1800	uint32	10
B2400	uint32	11
B4800	uint32	12
B9600	uint32	13

Name	Type	Value
B19200	uint32	14
B38400	uint32	15
B57600	uint32	16
B115200	uint32	17
B230400	uint32	18
B460800	uint32	19
B500000	uint32	20
B576000	uint32	21
B921600	uint32	22
B1000000	uint32	23
B1152000	uint32	24
B1500000	uint32	25
B2000000	uint32	26
B2500000	uint32	27
B3000000	uint32	28
B3500000	uint32	29
B4000000	uint32	30
MAX_BAUD	uint32	
CSIZE	uint32	64
CS5	uint32	0
CS6	uint32	32
CS7	uint32	40
CS8	uint32	48
CSTOPB	uint32	128
CREAD	uint32	256
PARENB	uint32	512
PARODD	uint32	1024
HUPCL	uint32	2048
CLOCAL	uint32	4096

lflag

Name	Type	Value
ISIG	uint32	1
ICANON	uint32	2
ECHO	uint32	4
ECHOE	uint32	8
ECHONL	uint32	16
NOFLSH	uint32	32
TOSTOP	uint32	64
ECHOCTL	uint32	128
ECHOPRT	uint32	256
ECHOKE	uint32	512
FLUSHO	uint32	1024
PENDIN	uint32	2048
ECHOK	uint32	4096

cc

Name	Type	Value
VINTR	uint16	0
VQUIT	uint16	1
VERASE	uint16	2
VKILL	uint16	3

Name	Type	Value
VEOF	uint16	4
VTIME	uint16	5
VSTART	uint16	8
VSTOP	uint16	9
VEOL	uint16	11
VREPRINT	uint16	12
VWERASE	uint16	14
VEOL2	uint16	16

Structures

termios

```
struct termios{
    tcflag_t   c_iflag;
    tcflag_t   c_oflag;
    tcflag_t   c_cflag;
    tcflag_t   c_lflag;
    uint16     c_line;
    uint16     c_cc[NCCS];
    speed_t    c_ispeed;
    speed_t    c_ospeed;
};
```

Operations

makeraw `void makeraw();`

makecooked `void makecooked();`

setattr
```
void setattr(
    termios t);
```

getattr `termios getattr();`

coyotos.Verifier

Interface `coyotos.Verifier`

Derivation:

```
coyotos.Cap
  coyotos.Verifier
```

Synopsis: Constructor Yield Verification Interface

Constructors also implement a query to determine whether a process is its yield. This allows a client to determine whether a process alleging to be an instance of some binary X was in fact created by the constructor of X instances.

In order to verify that a constructor is authentic, there is a constructor verifier, which is the Verifier for the Metaconstructor. In the normal Coyotos configuration, both the metaconstructor and the space bank verifier are readily available to applications as part of the standard runtime environment.

Operations

verifyYield Returns true exactly if `yield` is a child of this constructor.

```
bool verifyYield(
    Cap yield);
```

coyotos.VirtualCopySpace

Interface `coyotos.VirtualCopySpace`

Derivation:

```
coyotos.Cap
  coyotos.SpaceHandler
    coyotos.VirtualCopySpace
```

Synopsis: `VirtualCopySpace`

Operations

freeze `Constructor freeze();`

coyotos.driver.IrqCallback

Interface `coyotos.driver.IrqCallback`

Derivation:

`coyotos.Cap`

  `coyotos.driver.IrqCallback`

Synopsis: Interrupt notification callback interface

This interface is invoked by the `IrqHelper` to advise the receiver that an interrupt has occurred.

Operations

onInterrupt Method invoked by the `IrqHelper` to advise the receiver that an interrupt has arrived.

```
void onInterrupt(
    coyotos.IrqCtl.irq_t irq);
```

coyotos.driver.IrqHelper

Interface `coyotos.driver.IrqHelper`

Derivation:

`coyotos.Cap`
  `coyotos.driver.IrqHelper`

Synopsis: Interrupt demultiplexing helper application.

The interrupt helper is a small process that waits for an interrupt and posts a notice to an interested client program. It exists primarily because the kernel has no means to perform a general up-call.

The helper executes in one of two states. In "loop" state, it waits for an interrupt and calls the callback capability. This state is exited when the callback invocation result is an exception. It also ends after the callback returns if the wait loop has been directed to run only once via `waitOnce`().

In "control" state, it receives on the `IrqHelper` interface and can be instructed to use a different callback capability or re-start the loop state.

A flaw in this design is that there is no straightforward way for an outside party to determine which state the `IrqHelper` is in. One way to *force* the `IrqHelper` to enter the control state is to nullify the process capability in its callback endpoint.

A second flaw in this design is that a callback failure may result in a lost interrupt.

Operations

setCallback Provide the helper with a notification callback capability that should be called from the wait-and-notify loop.

```
void setCallback(
    IrqCallback callbackCap);
```

isDeliveryPending Return true iff an interrupt callback is pending.

```
bool isDeliveryPending();
```

This will return true if an interrupt wait has completed and the associated callback generated an exception. If `runWaitLoop`() or `waitOnce`() are called while a delivery is pending, they will re-try the callback.

cancelPendingDelivery If an interrupt delivery is pending, cancel it, returning true iff cancelation occurred.

```
bool cancelPendingDelivery();
```

It is the responsibility of the caller not to lose track of pending interrupts.

runWaitLoop Start the wait and notify loop, continuing until otherwise instructed by the notification response.

```
void runWaitLoop();
```

The `IrqHelper` returns immediately from this request, and then enters a loop in which it waits for its assigned interrupt, calls the callback capability, and repeats. If the callback generates an exception, the wait loop is terminated and the helper re-enters the control state.

Raises `cap.RequestError` exception if interrupt is not initialized or callback capability is not set.

waitOnce Execute the wait and notify loop exactly once.

```
void waitOnce();
```

coyotos.driver.IrqHelperCtl

Interface `coyotos.driver.IrqHelperCtl`

Derivation:

`coyotos.Cap`
  `coyotos.driver.IrqHelperCtl`

Synopsis: Interrupt demultiplexing helper application.

The interrupt helper is a small process that waits for an interrupt and posts a notice to an interested client program. It exists primarily because the kernel has no means to perform a general up-call.

Operations

setIRQ Tell the helper what interrupt it should wait for.

```
void setIRQ(
    coyotos.IrqCtl.irq_t irq);
```

getHelper `IrqHelper getHelper();`

coyotos.driver.TextConsole

Interface `coyotos.driver.TextConsole`

Derivation:

`coyotos.Cap`
  `coyotos.driver.TextConsole`

Synopsis: ASCII console display interface.

This is the interface specification for a text-oriented display providing minimal color capabilities. Conceptually, it implements the display component of an ASCII terminal, including the display-oriented ANSI terminal escape codes. Note that it does *not* implement the ANSI identify or writeback sequences, mainly because this is an output-only component.

When we have a bidirection stream interface defined, this interface should derive from that and should be cut over to use compatible read/write methods.

The current interface makes no provision for signalling display size changes. It is not immediately clear whether that should be handled here or in a separate process.

This is a very basic text display driver, originally thrown together for the PC display running in MDA mode (mode 25.)

Type Definitions

chString `typedef anon9 chString;`

Operations

clear Clear the display

```
void clear();
```

putChar Write ASSCII character to display.

```
void putChar(
    wchar8 c);
```

putCharSequence Write ASSCII character string to display.

```
void putCharSequence(
    chString s);
```

Bibliography

[1] Norman Hardy. “The KeyKOS Architecture.” *Operating Systems Review*, **19**(4), October 1985, pp. 8–25.

[2] J. S. Shapiro, J. M. Smith, and D. J. Farber. “EROS, A Fast Capability System” *Proc. 17th ACM Symposium on Operating Systems Principles*. Dec 1999. pp. 170–185. Kiawah Island Resort, SC, USA.