Contents

COCONUT-SVSM System Call ABI for the X86-64 Architecture

System Call Architecture

The system call architecture of COCONUT-SVSM defines system call classes. Each system call is identified by a 32-bit class number and a separate 32-bit system call number. Both numbers form a **Global System Call Number** of 64-bit with the class stored in the upper 32 bits and the system call number in the lower 32 bits.

System Call Classes

1. Class `0` maps the basic system calls used to manage virtual memory, thread and process life-times and system call batching.
2. Class `1` contains system calls specific to file objects and file system access in general.

Parameter Types

The COCONUT-SVSM system call ABI uses standard parameter types where appropriate. The common types are listed below.

Primitive Types

Type	Size in Bytes	Description
`s32`	4	Signed 32-bit integer
`u32`	4	Unsigned 32-bit integer
`s64`	8	Signed 64-bit integer
`u64`	8	Unsigned 64-bit integer
`CString`	8	Pointer to a `0`-terminated string
`VirtAddr`	8	Canonical unsigned virtual address
`ObjHandle`	4	Handle for a kernel managed object
`PID`	4	Process ID - Aliases `u32`
`TID`	4	Thread ID - Aliases `u32`

`MMFlags` - Memory Mapping Flags The following flags are defined for the memory mapping system calls.

Flag	Description
`MAP_WRITE`	Writable mapping
`MAP_PRIVATE`	Writes to memory remain private to the process
`MAP_ANONYMOUS`	Mapping is not backed by an object
`MAP_FIXED`	Mapping requires a fixed virtual address

Error Codes

The standard error codes returned by COCONUT-SVSM are listed below. Each error code is of type `u32`. Every value `X` is returned as `0 - X` upon return from a system call.

Error Code	Value	Description
`EINVAL`	`1`	Invalid input parameter
`ENOSYS`	`2`	System call not implemented
`ENOMEM`	`3`	Not enough memory to perform operation
`EPERM`	`4`	Insufficient permissions to execute system call
`EFAULT`	`5`	Fault happened while executing system call
`EBUSY`	`6`	Device or resource busy
`ENOTFOUND`	`7`	Resource not found
`ENOTSUPP`	`8`	Operation not supported

ABI Definition

The control transfer between user-mode and kernel-mode is done via software interrupts. Currently the vector `0x80` is used. The system call ABI described below is designed to allow an easy switch to the `SYSCALL` instruction.

Parameters to system calls are passed via CPU registers. The register assignment is listed in the table below:

Register	In	Out
`RAX`	Global Syscall number	Return code
`RCX`	-	Undefined
`RDI`	Parameter 1	Unmodified
`RSI`	Parameter 2	Unmodified
`R8`	Parameter 3	Unmodified
`R9`	Parameter 4	Unmodified
`R10`	Parameter 5	Unmodified
`R11`	-	Undefined
`RFLAGS`	-	Unmodifed

All other registers remain unmodified from the perspective of the process and have no influence on the execution of the system call.

`Class 0` - Basic System Calls

EXIT

Exits the calling process, including all threads.

System call number: `0`

Parameters

Parameter	Type	Description
1. `exit_code`	`u32`	Exit code

Returns This system call does not return.

MMAP

Creates a new virtual memory mapping in the process address space.

System call number: `1`

Parameters

Parameter	Type	Description
1. `Obj`	`ObjHandle`	Object handle
2. `addr`	`VirtAddr`	Virtual address hint
3. `offset`	`u64`	Object offset
4. `length`	`u64`	Length
5. `flags`	`MMFlags`	Flags

Returns Unsigned virtual address on success, Error code on failure.

MUNMAP

Removes virtual memory mapping from the process address space.

System call number: `2`

Parameters

Parameter	Type	Description
1. `addr`	`VirtAddr`	Virtual address
2. `length`	`u64`	Length to unmap

Returns `0` on success, error code on failure

MRESIZE

Changes the size of a virtual memory mapping.

System call number: `3`

Parameters

Parameter	Type	Description
1. `Base`	`VirtAddr`	Virtual base address
2. `Length`	`u64`	New absolute length

Returns `0` on Success, error code on failure.

EXEC

Create a new process and run it immediately. The new process get a new root directory in the file-system and will only be able to access files in or below that directory. The `root` directory is relative to the calling processes root directory.

System call number: `4`

Parameters

Parameter	Type	Description
1. `file`	`CString`	Path to executable
2. `root`	`CString`	Path do process root directory
3. `flags`	`PFlags`	Execution flags

Returns `PID` of new process on Success, error code on failure.

WAIT

Wait for a child process to exit and return exit code.

System call number: `5`

Parameters

Parameter	Type	Description
1. `pid`	`PID`	PID of the child process

Returns Exit code of process `PID` on Success, error code on failure.

THREAD_CREATE

Creates a new thread associated with the calling process and executes it immediately. The stack and the initial instruction pointer of the thread are passed via the `X86Regs` struct.

System call number: `6`

Parameters

Parameter	Type	Description
1. `regs`	`X86Regs`	Initial register state upon thread launch

Struct `X86Regs`

```
#[repr(C)]
struct X86Regs {
    // General purpose registers
    r15: u64,
    r14: u64,
    r13: u64,
    r12: u64,
    r11: u64,
    r10: u64,
    r9: u64,
    r8: u64,
    rbp: u64,
    rdi: u64,
    rsi: u64,
    rdx: u64,
    rcx: u64,
    rbx: u64,
    rax: u64,

    /// Instruction pointer
    rip: u64,
    /// Reserved MBZ
    rsvd1: u64,
    /// RFlags
    flags: u64,
    /// Stack pointer
    rsp: u64,
    /// Reserved MBZ
    rsvd2: u64,
}
```

Returns `TID` on success, error code on failure.

THREAD_JOIN

Waits for a thread to finish execution.

System call number: `7`

Parameters

Parameter	Type	Description
1. `tid`	`TID`	Thread ID to wait for

Returns `0` on success, error code on failure.

BATCH

Execute multiple system calls at once. The `batch_ptr` parameter points to an array of `struct SysCallIn` in process memory describing all system calls to execute. The COCONUT kernel will start to execute them in order (but does not necessarily complete in order) and writes the results into an array of `struct SysCallOut` pointed to by the `compl_ptr` parameter. The order in which results are written is not guaranteed to match the order in the `struct SysCallIn` array. The arrays pointed to by `batch_ptr` and `compl_ptr` must have at least `entries` number of elements.

None of the system calls can be considered finished before the **BATCH** system call returns, even when another thread observes it as finished in the `compl_ptr` array.

Calling the **BATCH** system call recursively is not supported and considered an error.

System call number: `8`

Parameters:

Parameter	Type	Description
1. `batch_ptr`	`VirtAddr`	Pointer to an array of `struct SysCallIn`
2. `compl_ptr`	`VirtAddr`	Pointer to an array of `struct SysCallOut`
3. `entries`	`u32`	Number of entries in the arrays pointed to by `batch_ptr` and `compl_ptr`
4. `flags`	`BFlags`	Batching flags

Returns The number of executed system calls from the `struct SysCallIn` array on success, error code otherwise. Note that a non-error return code does not indicate all executed system calls where successful. The calling process must evaluate the `struct SysCallOut` array to check for status of individual system calls.

Definition of `struct SysCallIn`

```
#[repr(C)]
struct SysCallIn {
    /// Global SysCall number
    nr: u64,
    /// User data - Used for completion notifcation
    user_data: u64,
    /// Parameters
    parameters: [u64, 5],
}
```

Definition of `struct SysCallOut`

```
#[repr(C)]
struct SysCallOut {
    /// System call result
    result: s64,
    /// User-data from SysCallIn
    user_data: u64,
}
```

Definition of `BFlags` The batching flags, of `BFlags` tell the COCONUT kernel how to execute the system calls. The defined flags are:

Flag	Description
`BF_STRICT_ORDER`	Execute system calls strictly in order, wait for one call to finish before starting the next one
`BF_ERROR_STOP`	Stop execution of further system calls when a call in the array returns an error code

OPEN_PHYSMEM

Get an object handle for physical memory access. The handle returned by this system call can be used with the `MMAP` to map portions of physical memory into the process address space.

System call number: `9`

Parameters This system call has no parameters.

Returns `ObjHandle` on success, error code on failure.

CLOSE

Closes an open `ObjHandle`.

System call number: `10`

Parameters

Parameter	Type	Description
1. `handle`	`ObjHandle`	Object handle to close

Returns Always returns `0`, even if called with an invalid handle.

`Class 1` - File System Calls

OPEN

Opens an existing file in the file-system.

System call number: `0`

Parameters

Parameter	Type	Description
1. `path`	`CString`	Full path of file to open
2. `mode`	`FS_MODE`	Opening mode (read, write, append, ...)
3. `flags`	`FS_FLAGS`	File open flags

Returns `ObjHandle` on success, error code on failure.

Definition of `FS_MODE`

Flag	Description
`FM_READ`	Open file for reading
`FM_WRITE`	Open file for writing
`FM_APPEND`	Place file pointer at end of file
`FM_TRUNC`	Truncate file to zero

Definition of `FS_FLAGS`

Flag	Description
`FF_CREATE`	Create file if it does not exist

READ

Read data from the current position in an open file.

System call number: `1`

Parameters

Parameter	Type	Description
1. `handle`	`ObjHandle`	Handle of open file to read from
2. `buffer`	`VirtAddr`	Pointer to buffer where to write file data
3. `size`	`u64`	Number of bytes to read

Returns Number of bytes written to `buffer` on success, error code on failure. Reading `0` bytes indicates the end of file has been reached.

WRITE

Write data to the current position into an open file.

System call number: `2`

Parameters

Parameter	Type	Description
1. `handle`	`ObjHandle`	Handle of open file to write to
2. `buffer`	`VirtAddr`	Pointer to buffer where to read file data to write
3. `size`	`u64`	Number of bytes to write

Returns Number of bytes written to file on success, error code on failure.

SEEK

Changes the file position pointer.

System call number: `3`

Parameters

Parameter	Type	Description
1. `handle`	`ObjHandle`	Handle of open file
2. `offset`	`s64`	Offset value for changing file position
3. `flags`	`SFLAGS`	Flags indicating how to interpret `offset`

Returns New file position pointer on success, error code on failure.

Definition of `SFLAGS`

Flag	Description
`SK_ABS`	`offset` is an absolute file position
`SK_REL`	`offset` is added to current file position
`SK_END`	`offset` is subtracted from end-of-file postion

OPENDIR

Opens a directory for reading directory entries.

System call number: `4`

Parameters

Parameter	Type	Description
1. `path`	`CString`	Path to directory to open

Returns `ObjHandle` for directory on success, error code on failure.

READDIR

Read directory entries from a directory handle.

System call number: `5`

Parameters

Parameter	Type	Description
1. `handle`	`ObjHandle`	Handle of open directory
2. `dirents`	`VirtAddr`	Pointer to array of `struct DirEnt`
3. `size`	`u64`	Number directory entries to read

Returns Number of directory entries read on success, error code on failure. A return value of `0` means there are no more entries to read.

Definition of `struct DirEnt`

```
/// Maximum length of file name in bytes
static const F_NAME_SIZE: usize = 256;
/// Files
static const F_TYPE_FILE: u8 = 0;
/// Directories
static const F_TYPE_DIR: u8 = 1;

#[repr(C)]
struct DirEnt {
    /// Entry name
    file_name: [u8; F_NAME_SIZE],
    /// Entry type
    file_type: u8,
    /// File size - 0 for directories
    file_size: u64,
}
```

`Class 2` - Event Subsystem Calls

Events are handled via kernel event objects. These are created by other subsystems and have a common interface. Currently only edge-triggered events are supported.

WAIT_FOR_EVENT

Sleep until event is triggered.

System call number: `0`

Parameters

Parameter	Type	Description
1. `handle`	`ObjHandle`	Event handle

Returns `0` on success, error code on failure.

TRIGGER_EDGE

Trigger event and wake up all waiters.

System call number: `1`

Parameters

Parameter	Type	Description
1. `handle`	`ObjHandle`	Event handle

Returns `0` on success, error code on failure.

`Class 3` VMM Subsystem Calls

The system calls in this class are uses to manage state of virtual machines controled by COCONUT-SVSM.

CAPABILITIES

Query values from the features and capabilities array of the VMM subsystem. Each value is of type `u64`.

System call number: `0`

Parameters

Parameter	Type	Description
1. index	u32	Index into capabilites array

The indexes below are specified:

Index	Meaning
`0`	Size of the capabilities array
`1`	Bitmap with available VM indexes
`2`	Global Feature bitmap

Feature bits are TBD.

Returns Capabilities array value at the requested index, `0` if index is invalid.

VM_OPEN

System call number: `1`

Get an object handle for a given VM index. Valid indexes can be obtained via the CAPABILITIES(1) system call. Each index can be opened by exactly one user-space process at any time.

Closing the `ObjHandle` returned by this system call will shut down and destroy all VM state managed by the COCONUT kernel.

Parameters

Parameter	Type	Description
1. index	u32	VM Index to open

Returns `ObjHandle` for VM on success, error code on failure.

VM_CAPABILITIES

Query capabilities array of a virtual machine.

System call number: `2`

Parameters

Parameter	Type	Description
1. `vm_obj`	`ObjHandle`	Object handle for virtual machine
2. `index`	u32	Index into capabilites array

The indexes below are specified:

Index	Meaning
`0`	Size of the capabilities array
`1`	Number of VCPUs for this VM

Returns Capabilities array value at the requested index, `0` if index is invalid.

MMIO_EVENT

Create an event object for MMIO events from a given VM object. The returned `evt_obj` provides an memory mappable area for communicating event details.

System call number: `3`

Parameters

Parameter	Type	Description
1. `vm_obj`	`ObjHandle`	Object handle for virtual machine
2. `pstart`	u64	Physical start address to monitor (inclusive)
3. `pend`	u64	Physical end address to monitor (exclusive)

Returns Event object on success, error code on failure.

IOIO_EVENT

Create an event object for IOIO events from a given VM object. The returned `evt_obj` provides an memory mappable area for communicating event details.

System call number: `4`

Parameters

Parameter	Type	Description
1. `vm_obj`	`ObjHandle`	Object handle for virtual machine
2. `start`	u32	IO port start address to monitor (inclusive)
3. `end`	u32	IO port end address to monitor (exclusive)

Returns Event object on success, error code on failure.

SET_MEM_STATE

Change accessibility state of physical memory for a given `vm_obj`

System call number: `5`

Parameters

Parameter	Type	Description
1. `vm_obj`	`ObjHandle`	Object handle for virtual machine
2. `paddr`	`u64`	Physical address to change state for
3. `page_size`	`u64`	Page-size to assume - must be a supported system page-size
4. `state`	`MSFLAFGS`	New Memory state

Definition of `MSFLAGS`

Flag	Description
`MS_PRIVATE`	Memory is private
`MS_READ`	Memory is readable by VM
`MS_WRITE`	Memory is writable by VM
`MS_EXEC`	Memory is executable by VM

Returns `0` on success, error code on failure.

`Class 4` VCPU System Calls

VCPU_CREATE

Creates a handle for a VCPU in a given VM context. The handle provides a 4 KiB page of memory which can be mapped via `MMAP` into the process memory.

The VCPU can be run with the `WAIT_FOR_EVENT()` system call.

System call number: `0`

Parameters

Parameter	Type	Description
1. `vm_obj`	`ObjHandle`	Object handle for virtual machine
2. `vcpu_idx`	u32	Index of VCPU, must be smaller than the number of VCPUs reported by `VM_CAPABILITIES(1)`

Returns `ObjHandle` for VCPU on success, error code on failure.

SET_STATE

Change VCPU state.

System call number: `1`

Parameters

Parameter	Type	Description
1. `vcpu_obj`	`ObjHandle`	Object handle for VCPU
2. `type`	`u32`	Type of state to change
3. `data`	`VirtAddr`	Virtual address of a `type`-specific data structure in process memory

The defined types are:

Type	Data Structure	Description
`GPR`	`struct X86Regs`	General purpose registers (including `RIP`)
`VMSA_GPA`	struct VmsaGpa	Guest physical address of VMSA (AMD only)

Returns `0` on success, error code on failure

GET_STATE

Query VCPU state.

System call number: `2`

Parameters

Parameter	Type	Description
1. `vcpu_obj`	`ObjHandle`	Object handle for VCPU
2. `type`	`u32`	Type of state to change
3. `data`	`VirtAddr`	Virtual address of a `type`-specific data structure in process memory

The defined types are similar to the `SET_STATE` call.

Returns `0` on success, error code on failure