MINIX 3 Kernel API

Jorrit N. Herder
<jnherder@cs.vu.nl>

October 20, 2005

Abstract

In general, kernel calls allow system processes to request kernel services, for example, to perform for privileged operations. This document briefly discusses the organization of kernel calls in MINIX 3 and provides an overview of all kernel calls.

Organization of kernel calls

A kernel call means that a request is sent to a kernel where it is handled by one of the kernel tasks. The details of assembling a request message, sending it to the kernel, and awaiting the response are conveniently hidden in a system library. The header file of this library is *src/include/minix/syslib.h* and its implementation is found in *src/lib/syslib*.

The actual implementation of the kernel calls is defined in one of the kernel tasks. In contrast to MINIX 2, the CLOCK task no longer accepts system calls. Instead, all calls are now directed to the SYSTEM task. Suppose that a program makes a sys_call() system call. By convention, this call is transformed into a request message with type SYS_CALL that is sent to the kernel task SYSTEM. The SYSTEM task handles the request in a function named do_call() and returns the result.

The mapping of kernel call numbers and handler functions is done during the SYSTEM task's intialization in *src/kernel/system.c*. The prototypes of the handler functions are declared in *src/kernel/system.h*. Their implementation is contained in separate files in the directory *src/kernel/system/*. These files are compiled into a library *src/kernel/system/system.a* that is linked with the kernel.

The kernel call numbers and their request and response parameters are defined in *src/include/minix/com.h*. Unfortunately, MINIX 2 does not follow a strict naming scheme. Therefore, numerous message types and parameters have been renamed in MINIX 3. Kernel calls now all start with SYS_ and all parameters that belong to the same kernel call now share a common prefix.

Overview of kernel calls in MINIX 3

A concise overview of the kernel calls in MINIX 3 is given in Figure 1. The status of each call compared to MINIX 2 is given in the last column.

Kernel call	Purpose	Status
	PROCESS MANAGEMENT	
SYS_FORK	Fork a process; copy parent process	
SYS_EXEC	Execute a process; initialize registers	
SYS_EXIT	Exit a user process; clear process slot	U
SYS_NICE	Change priority of a user process	N
SYS_PRIVCTL	Change system process privileges	N
SYS_TRACE	Trace or control process execution	
	SIGNAL HANDLING	
SYS_KILL	Send a signal to a process	U
SYS_GETKSIG	Check for pending kernel signals	N
SYS_ENDKSIG	Tell kernel signal has been processed	
SYS_SIGSEND	Start POSIX-style signal handler	
SYS_SIGRETURN	Return from POSIX-style signal	
	MEMORY MANAGEMENT	
SYS_NEWMAP	Install new or updated memory map	
SYS_SEGCTL	Add extra, remote memory segment	N
SYS_MEMSET	Write a pattern into physical memory area	N
	COPYING DATA	
SYS_UMAP	Map virtual to physical address	U
SYS_VIRCOPY	Copy data using virtual addressing	U
SYS_PHYSCOPY	Copy data using physical addressing	U
SYS_VIRVCOPY	Handle vector with virtual copy requests	U
SYS_PHYSVCOPY	Handle vector with physical copy requests	N
	DEVICE I/O	
SYS_DEVIO	Read or write a single device register	N
SYS_SDEVIO	Input or output an entire data buffer	N
SYS_VDEVIO	Process vector with multiple requests	N
SYS_IRQCTL	Set or reset an interrupt policy	N
SYS_INT86	Make a real-mode BIOS call	N
SYS_IOPENABLE	Give process I/O privilege	N
	SYSTEM CONTROL	
SYS_ABORT	Abort MINIX: shutdown the system	U
SYS_GETINFO	Get a copy system info or kernel data	N
	CLOCK FUNCTIONALITY	
SYS_SETALARM	Set or reset a synchronous alarm timer	U
SYS_TIMES	Get process times and uptime since boot	U

Figure 1: This figure provides an overview of the kernel calls in MINIX 3. The legenda for the status is: to be New or Updated (i.e., fully revised—all calls got minor updates) since MINIX 2.

MINIX 3 kernel call interface

MINIX' kernel call interface is detailed below. For each kernel call the message type, the purpose, message type, request and/ or response parameters, and return value are specified. The shorthand Additional remarks about the future status of the call also may be provided.

Legenda

CONSTANT: defined constant; a number indicating the request type or status
PARAMETER: message parameter; a field in the request or response message
void sys_call(arguments): system library function; shorthand to make a kernel call

Alphabetical overview

SYS_ABORT: Shutdown MINIX and return to the boot monitor—if possible. This is used by PM, FS and TTY. Normal aborts usually are initiated by the user, for example, by means of the `'shutdown'` command or typing a `'Ctrl-Alt-Del'`. MINIX will also be taken down if a fatal error occurs in the PM or FS.

request parameters

ABRT_HOW: How to abort. One of the values defined in *src/include/unistd.h*:

- RBT_HALT Halt MINIX and return to the boot monitor.
- RBT_REBOOT Reboot MINIX.
- RBT_PANIC A kernel panic occurred.
- RBT_MONITOR Run the specified code at the boot monitor.
- RBT_RESET Hard reset the system.

ABRT_MON_PROC: Process to get the boot monitor parameters from.

ABRT_MON_LEN: Length of the boot monitor parameters.

ABRT_MON_ADDR: Virtual address of the parameters.

return value

OK: The shutdown sequence was started.

ENIVAL: Invalid process number.

EFAULT: Illegal monitor parameters address.

E2BIG: Monitor parameters exceed maximum length.

library functions

int sys_abort(int shutdown_status, ...);

SYS_DEVIO: Perform device I/O on behalf of a user-space device driver. The driver can request a single port to be read or written with this call. Also see the SYS_SDEVIO and SYS_VDEVIO kernel calls.

request parameters

DIO_REQUEST: Input or output.

- DIO_INPUT Read a value from DIO_PORT.

- DIO_OUTPUT Write DIO_VALUE to DIO_PORT.

DIO_TYPE: A flag indicating the type of values.

- DIO_BYTE Byte type.
- DIO_WORD Word type.
- DIO_LONG Long type.

DIO_PORT: The port to be read or written.

DIO_VALUE: Value to write to the given port. For DIO_OUTPUT only.

response parameters

DIO_VALUE: Value that was read from the given port. For DIO_INPUT only.

return value

OK: The port I/O was successfully done.

EINVAL: An invalid DIO_REQUEST or DIO_TYPE was provided.

library functions

```
int sys_in(port_t port, unsigned long value, int io_type);
int sys_inb(port_t port, u8_t *byte);
int sys_inw(port_t port, u16_t *word);
int sys_inl(port_t port, u32_t *long);
int sys_out(port_t port, unsigned long *value, int io_type);
int sys_outb(port_t port, u8_t byte);
int sys_outw(port_t port, u16_t word);
int sys_outl(port_t port, u32_t long);
```

SYS_ENDKSIG: Finish a kernel signal. The PM uses this call to indicate it has processed the kernel signals in the map obtained through a SYS_GETKSIG kernel call.

response parameters

SIG_PROC: The process that it concerns.

return value

EINVAL: The process had no pending signals or already exited.

OK: The kernel cleared all pending signals.

library functions

```
int sys_endksig(int proc_nr);
```

SYS_EXEC: Update a process' registers after a successfull exec() POSIX-call. After the FS has copied the binary image into memory, the PM informs the kernel about the new register details.

request parameters

PR_PROC_NR: Process that executed a program.

PR_STACK_PTR: New stack pointer.

PR_IP_PTR: New program counter.

PR_NAME_PTR: Pointer to name of program.

return value

OK: This call always succeeds.

library functions

int sys_exec(int proc, char *stack_ptr, char *prog_name, vir_bytes pc);

SYS_EXIT: Clear a process slot. This is usually called by the PM to clean up after a user process exited. System processes, including the PM, can also directly call this function to exit themselves.

request parameters

PR_PROC_NR: Slot number of exiting process if caller is PM. Use SELF to exit the PM.

return value

OK: The cleanup succeeded.

EINVAL: Incorrect process number.

EDONTREPLY: This call does not return if a system process exited.

library functions

int sys_exit(int proc_nr);

SYS_FORK: Allocate a new (child) process in the kernel process table and initialize it based on the prototype (parent) process. The PM has found a free process slot for the child process in its own process table and now requests the kernel to update the kernel's process table.

request parameters

PR_PROC_NR: Child's process table slot.

PR_PPROC_NR: Parent, the process that forked.

return value

OK: A new process slot was successfully assigned.

EINVAL: Invalid parent process number or child slot in use.

library functions

int sys_fork(int parent_proc_nr, int child_proc_nr);

SYS_GETINFO: Obtain a copy of a kernel data structure. This call supports user-space device drivers and servers that need certain system information.

request parameters

I_REQUEST: The type of system information that is requested.

- GET_IMAGE Copy boot image table.
- GET_IRQHOOKS Copy table with interrupt hooks.
- GET_KINFO Copy kernel information structure.
- GET_KMESSAGES Copy buffer with diagnostic kernel messages.
- GET_LOCKTIMING Copy lock times—if DEBUG_TIME_LOCKS is set.
- GET_MACHINE Copy system environment.
- GET_MONPARAMS Copy parameters set by the boot monitor.
- GET_PRIVTAB Copy system privileges table.
- GET_PROCTAB Copy entire kernel process table.
- GET_PROC Copy single process table slot.
- GET_RANDOMNESS Copy randomness gathered by kernel events.

- GET_SCHEDINFO Copy ready queues and process table.

I_VAL_PTR: Virtual address where the information should be copied to.

I_VAL_LEN: Maximum length that the caller can handle.

I_VAL_PTR2: Optional, second address. Used when copying scheduling data.

I_VAL_LEN2: Optional, second length. Overloaded for process number.

return value

OK: The information request succeeded.

EFAULT: An illegal memory address was detected.

E2BIG: Requested data exceeds the maximum provided by the caller.

library functions

```
int sys_getinfo(int request, void *ptr, int len, void *ptr2, int len2);
int sys_getirqhooks(struct irq_hook *ptr);
int sys_getimage(struct boot_image *ptr);
int sys_getkinfo(struct kinfo *ptr);
int sys_getkmessages(struct kmessages *ptr);
int sys_getlocktimings(struct lock_timingdata *ptr);
int sys_getmachine(struct machine *ptr);
int sys_getmonparams(char *ptr, int max_len);
int sys_getprivtab(struct priv *ptr);
int sys_getproctab(struct proc *ptr);
int sys_getproc(struct proc *ptr, int proc_nr);
int sys_getrandomness(struct randomness *ptr);
int sys_getschedinfo(struct proc* ptr, struct proc *ptr2);
```

SYS_GETKSIG: Checks whether there is a process that must be signaled. This is repeatedly done by the PM after receiving a notification that there are kernel signals pending.

response parameters

SIG_PROC: Return next process with pending signals or NONE.

SIG_MAP: Bit map with pending kernel signals.

return value

OK: This call always succeeds.

library functions

```
int sys_getksig(int *proc_nr, sigset_t *sig_map);
```

SYS_INT86: Make a real-mode BIOS on behalf of a user-space device driver. This temporarily switches from 32-bit protected mode to 16-bit real-mode to access the BIOS calls. It is here to support the BIOS_WINI device driver.

request parameters

INT86_REG86: Address of request at the caller.

return value

OK: BIOS call successfully done.

EFAULT: Invalid request address.

library functions

-

SYS_IOPENABLE: Enable the CPU's I/O privilege level bits for to the given process, so that is is allowed to directly perform I/O in user space.

request parameters

PROC_NR: The process to give I/O privileges.

return value

OK: Always succeeds.

library functions

-

SYS_IRQCTL: Set or reset a hardware interrupt policy for a given IRQ line and enable or disable interrupts for this line. This call allows user-space device drivers to grab a hook for use with the kernel's generic interrupt handler. The kernel's interrupt handler merely notifies the driver about the interrupt with a HARD_INT message and reenables the IRQ line if the policy says so. The notification message will contain the 'id' provided by the caller as an argument. Once a policy is in place, drivers can enable and disable interrupts.

request parameters

IRQ_REQUEST: Interrupt control request to perform.

- IRQ_SETPOLICY Set interrupt policy for the generic interrupt handler.
- IRQ_RMPOLICY Remove a previously set interrupt policy.
- IRQ_ENABLE Enable IRQs for the given IRQ line.
- IRQ_DISABLE Disable IRQs for the given IRQ line.

IRQ_VECTOR: IRQ line that must be controlled.

IRQ_POLICY: Bit map with flags indicating IRQ policy.

IRQ_HOOK_ID: When setting a policy this provides index sent to caller upon interrupt. For other requests it is the kernel hook identifier returned by the kernel.

response parameters

IRQ_HOOK_ID: Kernel hook identifier associated with the driver.

return value

EINVAL: Invalid request, IRQ line, hook id, or process number.

EPERM: Only owner of hook can toggle interrupts or release the hook.

ENOSPC: No free IRQ hook could be found.

OK: The request was successfully handled.

library functions

```
int sys_irqctl(int request, int irq_vec, int policy, int *hook_id);
int sys_irqsetpolicy(int irq_vec, int policy, int *hook_id);
int sys_irqrmpolicy(int irq_vec, int *hook_id);
int sys_irqenable(int hook_id);
int sys_irqdisable(int hook_id);
```

SYS_KILL: Signal a process on behalf of a system server. A system process can signal another process with this call. The kernel notifies the PM about the pending signal for further processing. (Note that the kill() POSIX-call is directly handled at the PM.) The PM uses this call to indirectly send a signal message to a system process. This happens when a signal arrives for a system process that set the special SIG_MESS signal handler with the sigaction() POSIX-call.

request parameters

SIG_PROC_NR: Process to be signaled.

SIG_NUMBER: Signal number. Range from 0 to _NSIG.

return value

OK: Call succeeded.

EINVAL: Illegal process or signal number.

EPERM: Cannot send a signal to a kernel task. PM cannot signal a user process with a notification message.

library functions

int sys_kill(int proc_nr, int sig_nr);

SYS_MEMSET: Write a 4-byte pattern into the indicated memory area. The call is used by the PM to zero the BSS segment on an exec() POSIX-call. The kernel is ask to do the work for performance reasons.

request parameters

MEM_PTR: Physical base address of the memory area.

MEM_COUNT: Length in bytes of the memory area.

MEM_PATTERN: The 4-byte pattern to be written.

return value

OK: Call always succeeds.

library functions

int sys_memset(long pattern, phys_bytes base, phys_bytes length);

SYS_NEWMAP: Install a new memory map for a newly forked process or if a process' memory map is changed. The kernel fetches the new memory map from PM and updates its data structures.

request parameters

PR_PROC_NR: Install new map for this process.

PR_MEM_PTR: Pointer to memory map at PM.

return value

OK: New map was successfully installed.

EFAULT: Incorrect address for new memory map.

EINVAL: Invalid process number.

library functions

int sys_newmap(int proc_nr, struct mem_map *ptr);

SYS_NICE: Change a process' priority. This is done by passing a nice values between PRIO_MIN (negative) and PRIO_MAX (positive). A nice value of zero resets the priority to the default.

request parameters

PR_PROC_NR: Process who's priority should be changed

PR_PRIORITY: New nice value for process' priority *return value*

OK: New priority was successfully set.

EINVAL: Invalid process number or priority.

EPERM: Cannot change priority of kernel task.

library functions

int sys_nice(int proc_nr, int priority);

SYS_PHYSCOPY: Copy data using physical addressing. The source and/ or destination address can be virtual like with SYS_VIRCOPY, but in addition an arbitrary physical address is accepted with PHYS_SEG.

request parameters

CP_SRC_SPACE: Source segment.

CP_SRC_ADDR: Virtual source address

CP_SRC_PROC_NR: Process number of the source process.

CP_DST_SPACE: Destination segment.

CP_DST_ADDR: Virtual destination address

CP_DST_PROC_NR: Process number of the destination process.

CP_NR_BYTES: Number of bytes to copy.

return value

OK: The copying was done.

EDOM: Invalid copy count.

EFAULT: Virtual to physical mapping failed.

EINVAL: Incorrect segment type or process number.

EPERM: Only owner of REMOTE_SEG can copy to or from it.

library functions

int sys_abscopy(phys_bytes src_phys, phys_bytes dst_phys, phys_bytes count);

int sys_physcopy(int src_proc, int src_seg, vir_bytes src_vir, int dst_proc, int dst_seg, vir_bytes dst_vir, phys_bytes count);

SYS_PHYSVCOPY: Copy multiple block of data using physical addressing. The request vector is fetched from the caller, and each element is handled like a regular SYS_PHYSCOPY request. Copying continues until all elements have been processed or an error occurs.

request parameters

VCP_VEC_SIZE: Number of elements in request vector.

VCP_VEC_ADDR: Virtual address of request vector at caller.

response parameters

VCP_NR_OK: Number of elements successfully copied.

return value

OK: The copying was done.

EDOM: Invalid copy count.

EFAULT: Virtual to physical mapping failed.

EINVAL: Copy vector too large, incorrect segment or invalid process.

EPERM: Only owner of REMOTE_SEG can copy to or from it.

library functions

int sys_physvcopy(phys_cp_req *copy_vec, int vec_size, int *nr_ok);

SYS_PRIVCTL: Get a private privilege structure and update a process' privileges. This is used to dynamically start system services.

request parameters

CTL_PROC_NR: Process who's privileges should be updated.

return value

OK: The calls succeeded.

EINVAL: Invalid process number.

ENOSPC: No free privilege structure found.

remarks

This system call will be extended to provide both better support and security checks for servers or device drivers that must be dynamically loaded. This is future work.

library functions

-

SYS_SDEVIO: Perform device I/O on behalf of a user-space device driver. Note that this call supports only byte and word granularity. The driver can request input or output of an entire buffer. Also see the SYS_DEVIO and SYS_VDEVIO kernel calls.

request parameters

DIO_REQUEST: Input or output.

- DIO_INPUT Read a value from DIO_PORT.
- DIO_OUTPUT Write DIO_VALUE to DIO_PORT.

DIO_TYPE: A flag indicating the type of values.

- DIO_BYTE Byte type.
- DIO_WORD Word type.

DIO_PORT: The port to be read or written.

DIO_PROC_NR: Process number where buffer is.

DIO_VEC_ADDR: Virtual address of buffer.

DIO_VEC_SIZE: Number of elements to input or output.

response parameters

DIO_VALUE: Value that was read from the given port. For DIO_INPUT only.

return value

OK: The port I/O was successfully done.

EINVAL: Invalid request or port granularity.

EPERM: Cannot do I/O for kernel tasks.

EFAULT: Invalid virtual address of buffer.

library functions

int sys_insb(port_t port, u8_t buffer, int count);
int sys_insw(port_t port, u16_t buffer, int count);
int sys_outsb(port_t port, u8_t buffer, int count);
int sys_outsw(port_t port, u16_t buffer, int count);
int sys_sdevio(int req, long port, int io_type, void *buffer, int count);

SYS_SEGCTL: Add a memory segment to the process' LDT and its remote memory map. The call returns a selector and offset that can be used to directly reach the remote segment, as well as an index into the remote memory map that can be used with the SYS_VIRCOPY kernel call.

request parameters

SEG_PHYS: Physical base address of segment.

SEG_SIZE: Size of segment.

response parameters

SEG_INDEX: Index into remote memory map.

SEG_SELECT: Segment selector for LDT entry.

SEG_OFFSET: Offset within segment. Zero, unless 4K granularity is used.

return value

ENOSPC: No free slot in remote memory map and LDT.

OK: Segment descriptor successfully added.

library functions

int sys_segctl(int *index, u16_t *seg, vir_bytes *off, phys_bytes phys, vir_bytes size);

SYS_SIGRETURN: Return from a POSIX-style signal handler. The PM requests the kernel to put things in order before the signalled process can resume execution. Also see the SYS_SIGSEND kernel call that pushes a signal context frame onto the stack.

request parameters

SIG_PROC: Indicates the process that was signaled.

SIG_CTXT_PTR: Pointer to context structure for POSIX-style signal handling.

response parameters

SIG_PROC: Return next process with pending signals or NONE.

return value

OK: Signal handling action successfully performed.

EINVAL: Invalid process number or context structure.

EFAULT: Invalid context structure address, or could not copy signal frame.

library functions

int sys_sigreturn(int proc_nr, struct sigmsg *sig_context);

SYS_SIGSEND: Signal a process on behalf of the PM by placing the context structure onto the stack. The kernel fetches the structure, initializes it, and copies it to the user's stack.

request parameters

SIG_PROC: Indicates the process that was signaled.

SIG_CTXT_PTR: Pointer to context structure for POSIX-style signal handling.

response parameters

SIG_PROC: Return next process with pending signals or NONE.

return value

OK: Signal handling action successfully performed.

EINVAL: Invalid process number.

EPERM: Cannot signal kernel tasks.

EFAULT: Invalid context structure address, or could not copy signal frame.

library functions

int sys_sigsend(int proc_nr, struct sigmsg *sig_context);

SYS_SETALARM : Set or reset a synchronous alarm timer. When the timer expires it causes a SYN_ALARM notification message with the current uptime as an argument to be sent to the caller. Only system processes can request synchronous alarms.

request parameters

ALRM_EXP_TIME: Absolute or relative expiration time in ticks for this alarm.

ALRM_ABS_TIME: Zero if expire time is relative to the current uptime.

response parameters

ALRM_TIME_LEFT: Ticks left on the previous alarm.

return value

OK: The alarm was successfully set.

EPERM: User processes cannot request alarms.

library functions

int sys_setalarm(clock_t expire_time, int abs_flag);

SYS_TIMES: Get the kernel's uptime since boot and process execution times.

request parameters

T_PROC_NR: The process to get the time information for, or NONE.

response parameters

T_USER_TIME: Process' user time in ticks, if valid number.

T_SYSTEM_TIME: Process' system time in ticks, if valid number.

T_BOOT_TICKS: Number of ticks since MINIX boot.

return value

OK: Always succeeds.

library functions

int sys_times(int proc_nr, clock_t *ptr);

SYS_TRACE: Monitor or control execution of the given process. Handle the debugging commands supported by the ptrace() system call.

request parameters

CTL_REQUEST: The tracing request.

- T_STOP Stop the process.
- T_GETINS Return value from instruction space.
- T_GETDATA Return value from data space.
- T_GETUSER Return value from user process table.

- T_SETINS Set value from instruction space.
- T_SETDATA Set value from data space.
- T_SETUSER Set value in user process table.
- T_RESUME Resume execution.
- T_STEP Set trace bit.

CTL_PROC_NR: The process number that is being traced.

CTL_ADDRESS: Virtual address in the traced process' space.

CTL_DATA: Data to be written.

response parameters

CTL_DATA: Data be returned.

return value

OK: Trace operation succeeded.

EIO: Set or get value failed.

EINVAL: Unsupported trace request.

PERM: Can only trace user processes.

library functions

int sys_trace(int request, int proc_nr, long addr, long *data_ptr);

SYS_UMAP: Map a virtual address to a physical address and return the physical address. The virtual address can be in LOCAL_SEG, REMOTE_SEG, or BIOS_SEG. An offset in bytes can be passed to verify whether this also falls within the segment.

request parameters

CP_SRC_PROC_NR: Process number of the address relates to.

CP_SRC_SPACE: Segment identifier.

CP_SRC_ADDR: Offset within segment.

CP_NR_BYTES: Number of bytes from start.

response parameters

CP_DST_ADDR: Physical address if mapping succeeded.

return value

OK: The copying was done.

EFAULT: Virtual to physical mapping failed.

EINVAL: Incorrect segment type or process number.

remarks

Address zero within BIOS_SEG returns EFAULT, while it the zeroth BIOS interrupt vector in fact is a valid address.

library functions

int sys_umap(int proc_nr, int seg, vir_bytes vir_addr, vir_bytes count, phys_bytes *phys_addr);

SYS_VDEVIO: Perform a series of device I/O on behalf of a user process. The call accepts a pointer to an array of (port,value)-pairs that is to be handeld at once. Hardware interrupts are temporarily disabled to prevented the bactch of I/O calls to be interrupted. Also see SYS_DEVIO and SYS_SDEVIO.

request parameters

DIO_REQUEST: Input or output.

- DIO_INPUT Read a value from DIO_PORT.
- DIO_OUTPUT Write DIO_VALUE to DIO_PORT.

DIO_TYPE: A flag indicating the type of values.

- DIO_BYTE Byte type.
- DIO_WORD Word type.
- DIO_LONG Long type.

DIO_VEC_SIZE: The number of ports to be handled.

DIO_VEC_ADDR: Virtual address of the (port,value)-pairs in the caller's space.

return value

OK: The port I/O was successfully done.

EINVAL: Invalid request or granularity.

E2BIG: Vector size exceeds maximum that can be handled.

EFAULT: The address of the (port,value)-pairs is erroneous.

library functions

```
int sys_voutb(pvb_pair_t *pvb_vec, int vec_size);
int sys_voutw(pvw_pair_t *pvw_vec, int vec_size);
int sys_voutl(pvl_pair_t *pvl_vec, int vec_size);
int sys_vinb(pvb_pair_t *pvb_vec, int vec_size);
int sys_vinw(pvw_pair_t *pvw_vec, int vec_size);
int sys_vinl(pvl_pair_t *pvl_vec, int vec_size);
```

SYS_VIRCOPY: Copy data using virtual addressing. The virtual can be in three segments: LOCAL_SEG (text, stack, data segments), REMOTE_SEG (e.g., RAM disk, video memory), and the BIOS_SEG (BIOS interrupt vectors, BIOS data area). This is the most common system call relating to copying.

request parameters

CP_SRC_SPACE: Source segment.

CP_SRC_ADDR: Virtual source address

CP_SRC_PROC_NR: Process number of the source process.

CP_DST_SPACE: Destination segment.

CP_DST_ADDR: Virtual destination address

CP_DST_PROC_NR: Process number of the destination process.

CP_NR_BYTES: Number of bytes to copy.

return value

OK: The copying was done.

EDOM: Invalid copy count.

EFAULT: Virtual to physical mapping failed.

EPERM: No permission to use PHYS_SEG.

EINVAL: Incorrect segment type or process number.

EPERM: Only owner of REMOTE_SEG can copy to or from it.

library functions

```
int sys_biosin(vir_bytes bios_vir, vir_bytes dst_vir, vir_bytes bytes);
int sys_biosout(vir_bytes src_vir, vir_bytes bios_vir, vir_bytes bytes);
int sys_datacopy(vir_bytes src_proc, vir_bytes src_vir, dst_proc, dst_vir, vir_bytes bytes);
int sys_textcopy(vir_bytes src_proc, vir_bytes src_vir, dst_proc, dst_vir, vir_bytes bytes);
```

int sys_stackcopy(vir_bytes src_proc, vir_bytes src_vir, dst_proc, dst_vir, vir_bytes bytes);
int sys_vircopy(int src_proc, int src_seg, vir_bytes src_vir, int dst_proc, int dst_seg, vir_bytes dst_vir, phys_bytes bytes);

SYS_VIRVCOPY: Copy multiple blocks of data using virtual addressing. The request vector is fetched from the caller, and each element is handled like a regular SYS_VIRCOPY request. Copying continues until all elements have been processed or an error occurs.

request parameters

VCP_VEC_SIZE: Number of elements in request vector.

VCP_VEC_ADDR: Virtual address of request vector at caller.

response parameters

VCP_VEC_OK: Number of elements successfully copied.

return value

OK: The copying was done.

EDOM: Invalid copy count.

EFAULT: Virtual to physical mapping failed.

EPERM: No permission to use PHYS_SEG.

EINVAL: Copy vector too large, incorrect segment or invalid process.

EPERM: Only owner of REMOTE_SEG can copy to or from it.

library functions

int sys_virvcopy(vir_cp_req *copy_vec, int vec_size, int *nr_ok);