Difference Point for system calls

Revision 2.8

Revision History

Rev.	Date	Prepared by	Updates / Remarks
2.2	28-February-18	NEC	the first version
2.3	23-March-18	NEC	Updated section-6 for the signal related difference point
2.4	23-July-18	NEC	Updated section 2.9 for “mmap()” for file backed mmap() Updated section 6 for effect of set-user-id and set-group-id bit of a VE executable
2.5	15-August-18	NEC	Added section 2.19 for clock_gettime() difference. Added section 6.18 for /proc/self difference.
2.6	14-September-18	NEC	Added section 6.23 for write() series system call difference.
2.7	29-November-18	NEC	Updated section 2.12 regarding the argv/envp limit for execve() system call. Added differences in relation to glibc.
2.8	08-February-19	NEC	This revision covers VEOS v2.0.3 or later. Changed the format of top page. Added section 6.24 for Non-atomic I/O Added section 6.25 for prlimit system call and RPM command behavior for VE.

1. Introduction

This document aims at listing down all the difference between Linux system calls and the VEOS specific system call implementation.

All the system calls are categorized as follows:

1. Supported system calls
   Here the list of supported system calls are provided which have complete support in VEOS. Also the differences with respect to Linux system call is also provided.

2. Partially supported system calls
   Here the list of partially supported (supported with limitations) are provided. Also the differences with respect to Linux system call is also provided.

3. Not supported system calls
   Here the list of system calls which are not supported in VEOS is provided.

2. Supported System Calls

Following is the list of system calls which are completely supported in VEOS.

SL No	System calls	Differences (Yes / No)
1.	fork	Yes
2.	waitid	Yes
3.	sched_getaffinity	Yes
4.	sched_setaffinity	Yes
5.	sched_yield	No
6.	getpgrp	No
7.	getpid	No
8.	getpgid	No
9.	getppid	No
10.	gettid	No
11.	getsid	No
12.	setsid	No
13.	setpgid	No
14.	time	No
15.	gettimeofday	No
16.	clock_getres	No
17.	vfork	No
18.	exit	No
19.	execve	Yes
20.	sysinfo	No
21.	sched_rr_get_interval	Yes
22.	acct	No
23.	clock_gettime	Yes
24.	kill	No
25.	tkill	No
26.	tgkill	No
27.	rt_sigqueueinfo	No
28.	rt_tgsigqueueinfo	No
29.	sigaction	Yes
30.	sigprocmask	Yes
31.	sigreturn	No
32.	sigsuspend	No
33.	sigaltstack	yes
34.	sigpending	No

35.	signalfd	Yes
36.	signalfd4	Yes
37.	rt_sigtimedwait	No
38.	lookup_dcookie	No
39.	semtimedop	No
40.	recvmmsg	Yes
41.	timer_getoverrun	No
42.	sendmsg	yes
43.	name_to_handle_at	No
44.	mq_getsetattr	No
45.	open_by_handle_at	No
46.	inotify_add_watch	No
47.	timerfd_settime	No
48.	timerfd_gettime	No
49.	newfstatat	No
50.	inotify_rm_watch	No
51.	ioprio_set	No
52.	ioprio_get	No
53.	ppoll	No
54.	getsockopt	No
55.	poll	No
56.	epoll_ctl	No
57.	getgroups	No
58.	socketpair	No
59.	fanotify_mark	No
60.	readlink	No
61.	epoll_create1	No
62.	fanotify_init	No
63.	semctl	No
64.	recvmsg	No
65.	writev	No
66.	msgctl	No
67.	msgrcv	No
68.	recvfrom	Yes
69.	mount	No
70.	truncate	No
71.	getpeername	No
72.	mq_timedreceive	Yes

73.	accept4	No
74.	sendto	No
75.	accept	No
76.	mq_timedsend	Yes
77.	utimensat	No
78.	epoll_pwait	No
79.	splice	No
80.	getresgid	No
81.	utime	No
82.	mq_open	No
83.	symlink	No
84.	statfs	No
85.	renameat	No
86.	epoll_wait	No
87.	utimes	No
88.	symlinkat	No
89.	flock	No
90.	futimesat	No
91.	connect	No
92.	msgsnd	No
93.	readlinkat	No
94.	setdomainname	No
95.	getdents	No
96.	mq_notify	No
97.	uname	No
98.	setsockopt	No
99.	fcntl	No
100.	setgroups	No
101.	syslog	No
102.	access	No
103.	openat	No
104.	write	No
105.	pwritev	No
106.	pwrite64	No
107.	sethostname	No
108.	creat	No
109.	fstatfs	No
110.	open	No

111.	stat	No
112.	bind	No
113.	setuid	No
114.	fstat	No
115.	getcwd	No
116.	timer_gettime	No
117.	setgid	No
118.	ftruncate	No
119.	close	No
120.	pause	No
121.	socket	No
122.	eventfd2	No
123.	fdatasync	No
124.	vhangup	No
125.	fadvise64	No
126.	inotify_init	No
127.	epoll_create	No
128.	select	No
129.	unlink	No
130.	pselect6	No
131.	dup	No
132.	dup2	No
133.	pipe	No
134.	nanosleep	No
135.	chown	No
136.	lchown	No
137.	fchown	No
138.	lseek	No
139.	mkdir	No
140.	tee	No
141.	chroot	No
142.	ioperm	No
143.	alarm	No
144.	mknodat	No
145.	setreuid	No
146.	sync	No
147.	getgid	No
148.	sync_file_range	No

149.	mknod	No
150.	fsync	No
151.	rename	No
152.	dup3	No
153.	faccessat	No
154.	lstat	No
155.	readahead	No
156.	getsockname	No
157.	preadv	No
158.	pread64	No
159.	read	No
160.	mq_unlink	No
161.	semget	No
162.	linkat	No
163.	setresuid	No
164.	eventfd	No
165.	fchmodat	No
166.	umask	No
167.	fchmod	No
168.	fchownat	No
169.	readv	No
170.	link	No
171.	rmdir	No
172.	setfsgid	No
173.	setfsuid	No
174.	chmod	No
175.	chdir	No
176.	geteuid	No
177.	pipe2	No
178.	unlinkat	No
179.	setregid	No
180.	msgget	No
181.	listen	No
182.	fchdir	No
183.	semop	No
184.	getresuid	No
185.	inotify_init1	No
186.	iopl	No

187.	fallocate	No
188.	getegid	No
189.	mkdirat	No
190.	setresgid	No
191.	getuid	No
192.	getdents64	No
193.	timerfd_create	No
194.	umount2	No
195.	timer_delete	No
196.	shutdown	No
197.	syncfs	No
198.	pivot_root	No
199.	mmap	Yes
200.	munmap	No
201.	mprotect	No
202.	msync	Yes
203.	shmget	Yes
204.	shmat	No
205.	shmctl	Yes
206.	process_vm_readv	No
207.	process_vm_writev	No
208.	grow	Yes
209.	getrusage	Yes
210.	sendfile	No
211.	timer_settime	No
212.	Sendmmsg	Yes
213.	brk	No
214.	shmdt	No
215.	fgetxattr	Yes
216.	flistxattr	Yes
217.	fremovexattr	Yes
218.	fsetxattr	Yes
219.	getxattr	Yes
220.	lgetxattr	Yes
221.	listxattr	Yes
222.	llistxattr	Yes
223.	lremovexattr	Yes
224.	lsetxattr	Yes
225.	removexattr	Yes

226.	setxattr	Yes
227.	sysve	VEOS specific

1. waitid()

1. If a child is sent a terminating signal, example SIGFPE, SIGTERM, etc., SIGKILL is sent instead of the actual one. This leads to WTERMSIG(status) value = SIGKILL (9) instead of the actual one when the child process is waited upon in the parent.

2. fork()

1. Copy-on-write is not supported in VEOS. When a child process is created, it is allocated new memory.
2. After a process exhausts its maximum limit of open file descriptors, subsequent invocation of fork() system call will fail with errno set to EAGAIN.
3. After changing the default root directory of the calling process to that specified in path with chroot() system call, subsequent invocation of fork() will fail with errno set to EAGAIN.

3. sched_getaffinity()

1. If pid 1 is given as an argument it will return -1 and errno ESRCH will be set.

4. sched_setaffinity()

1. If pid 1 is given as an argument it will return -1 and errno ESRCH will be set.

5. sched_rr_get_interval()

1. If pid 1 is given as an argument it will return -1 and errno ESRCH will be set.

6. sigaction()

1. In VEOS signal 35 is reserved as SIGTIMER and signal number 34 is reserved by musl-libc. If an attempt is made to register handler for the same then sigaction() will fail with EINVAL error. There is no such limitation for glibc.

7. sigaltstack()

1. Minimum alternate stack size will be VE_MINSIGSTKSZ (533400). If user give alternate stack size less than 512KB ENOMEM will be returned.
2. If an attempt is made to register alternate stack with invalid stack pointer then sigaltstack() will fail with EFAULT.

8. Signalfd()/signalfd4

If an attempt is made to invoke signalfd() with invalid “mask” argument then it return EFAULT instead of EINVAL.

9. mmap()

1. Following flags of mmap() are not supported in VE and would return EINVAL
    a. MAP_GROWSDOWN
    b. MAP_HUGETLB
    c. MAP_LOCKED
    d. MAP_NONBLOCK
    e. MAP_POPULATE
2. In VEOS huge page mappings are not configured via huge tlb file system.
3. Only two page size are supported for mmap in which smallest mmap page size is 2MB and largest page size is 64MB.
4. New flags are added MAP_2MB, MAP_64MB to get memory mapping over specific page size.

5. If VE process does not specify page size in mmap flags then default page size considered depends upon executable page size.
6. MAP_STACK is supported and will be used with grow() system call else the behavior is undefined. No physical mappings are done in this flag.
7. If user tries to do mmap() with MAP_FIXED flag in between the range of 96TB-97TB the it would fail as this is reserved for VE process address space.
8. In case of file backed mmap() with MAP_SHARED flag, VE memory is shared by VE processes of the same VE node. VE memory contents with underline file are synced using msync(), munmap() system call or during VE process exit. Because of this architecture, the change of VE memory contents is not visible from a VE process of another VE node or a VH process, until VE process invokes msync() or munmap(), or terminates.
9. In case of file backed mmap() with MAP_SHARED flag, the contents of the file are transferred to VE memory when mmap request comes firstly. The contents of the file are not synced to VE memory if mmap request for already mapped region comes again. So, the change of the underline file by a VE process of another VE or a VH process is not visible from the VE process.
10. In case of file backed mmap() with MAP_SHARED flag, if VE process has mapped the file(say mapping 1) and then it invokes ftruncate() to decrease the file size then while accessing mapping 1, SIGBUS will not be generated. But new mapping to same file will give SIGBUS if user tries to access beyond file size.
11. Even if MAP_NORESERVE flag is provided by user then VE physical pages are allocated as of size inputted based upon the available VE memory.
12. If the file is mapped with different page size, then VE memory is not shared.
13. VEOS handles A POSIX shared memory object created by shm_open() in the same manner with a file backed memory. So, it can be used as a shared memory in one VE node. A swap area corresponding to it is required to preserve the contents.
14. In VE accessing memory mapped with file "/dev/zero" will give SIGBUS but in Linux it is successful. So for all type of file whose size is zero will always result in SIGBUS when mapped and accessed in VE.
15. In case of file backed mmap() the contents of the file are not synced to VE memory if write request for the mapped file comes. So, the change of the underline file is not visible, even if the VE process mapping the file writes the same file using write() system call.

10.shmget()

1. SHM_HUGETLB flag is not supported in VEOS and if VE process use this flag in system call then EINVAL will be returned
2. Only two page size are supported for shmget in which smallest shm page size is 2MB and largest page size is 64MB.
3. New flags are added SHM_2MB, SHM_64MB to create shared memory segment with specific page size.
4. Even if SHM_NORESERVE flag is provided by user then VE physical pages are allocated as of size of segment based upon the available VE memory.
5. Minimum size alignment of shared memory segment is not as per SHMLBA (4KB). The size alignment depends upon SHM_2MB/SHM_64MB flags.

6. The VE memory is shared by VE processes of the same VE node. The VE memory and VH memory are not synced. So, the contents of VE memory are not visible from a VE process of another VE node or VH process.
7. If a process 1 creates a shared memory with 64 MB page size and other process 2 invokes shmget() with 2 MB page size then shmget() would success as segment is already created. But while attaching this segment in process 2 address returned would be 64 MB aligned (as per the segment page size defined while creation).

11.grow()

grow() system call is used to extend the stack size of VE process or thread.

1. There are two input arguments in this system call.
2. System call returns EINVAL if invalid address is specified.
3. This system call will be invoked via function epilogue/prologue. It is not recommended that VE process will explicitly invoke this system call otherwise behavior is undefined.

12.execve()

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

1. After successful `execve()` system call, argv[0] in the new loaded VE binary will always contains the absolute path of VE binary. If caller of `execve()` provide some data in argv[0] then it will be lost. So, the caller of `execve()` have to strictly follow the convention of passing binary name in argv[0]. Further arguments to VE program in argv[1] to so on.

   Note: This is applicable to all the exec() family functions.

2. VE process can execute new VE program or VH program. When VE process executes new VE program, VE process needs to specify VE program with the first argument of execve() system call. When VE process executes VH program, VE process's information such as resource limit is discarded, even if VH program executes VE program, again.

3. execve() system call's second argument "**argv**" is an array of argument strings passed to the new program. In VE the maximum number of command line arguments which can be passed are 512. When execve() is invoked with "argv" greater than 512 strings, system call will fail with E2BIG errno.

   And execve() system call's third argument "**envp**" is an array of strings, conventionally of the form key=value, passed to the new program as environment. In VE the maximum number of environment variable strings which can be passed are 508. When execve() is invoked with "envp" greater than 508 strings, system call will fail with E2BIG errno. Additional 4 environment variables VE_EXEC_PATH, LOG4C_RCPATH, HOME and PWD are always passed to the new program.

4. Passing an invalid file having executable permission to execve() system call will result in termination of whole thread group.

13.msync()

1. In VE flag MS_INVALIDATE is not supported and return error as EINVAL. We don't have any support to invalidate other mappings of the same file.

14.recvfrom() / recvmmsg() / sendmmsg() / mq_timedreceive() / mq_timedsend()

In VE environment behavior of system call may differ for the system calls that takes size/length/count `size_t (unsigned int)` as an argument for buffer length. For example in the above series of system call `recvfrom` takes `size_t len` as an input argument to system call

```
ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                        struct sockaddr *src_addr, socklen_t *addrlen);
```

Now if any user application while invoking `recvfrom(2)` passes a negative value say “-1” ((this value will be converted into huge positive value) for `len` argument.
In this scenario (negative value converted to huge positive) while handing system call, Linux kernel truncate the huge positive value to `MAX_RW_COUNT` and system call return success.
However in case of VE these system call may fail (for handling the negative value). Since in VEOS implementation of system call, when invalid negative value will be passed as an argument to system call (considered as a huge positive value) even after truncating this huge positive value to `MAX_RW_COUNT` system call may return failure, because in VEOS implementation system call handler need to allocate local buffer and send/receive data from VE memory based on the value of len (which happens to be `MAX_ROW_COUNT`) hence It may happen that allocating local buffer (through `malloc`) may fail or send/receive data from VE memory may fail and system call may return `EFAULT/ENOMEM/ENOSPC` as an error code.

Similar deviation in system call behavior may be observed in below system calls

- recvmmsg()/sendmmsg()
- mq_timedreceive()/mq_timedsend()
- lookup_dcookie()
- getsockopt()/setsockopt()
- readv()/writev()
- sendto()
- epoll_pwait()
- epoll_wait()
- setgroups()
- read()/write()
- getcwd()
- pread64()/pwrite64()
- getdents64()

15.shmclt()

1. Following flags of shmctl() are not supported in VE and would return EINVAL
    a. SHM_LOCK
    b. SHM_UNLOCK

16.getrusage()

Following fields of rusage structure are not maintained in VE and would return 0.

- struct timeval ru_stime;
- long ru_minflt;
- long ru_majflt;

17.chroot()

Note

- After changing the default root directory of the calling process to that specified in path with chroot() system call, subsequent invocation of fork(), vfork() and clone() system calls will fail with errno set to EAGAIN.

-

18.sigprocmask()

In VEOS SIGCONT signal cannot be masked. The request to mask SIGCONT signal through sigprocmask(2) will be ignored and sigprocmask(2) will return success to user program.

19. clock_gettime()

If argument "clock_id" is the ID of INIT process CPU-time clock, then failure is returned with errno set to EINVAL, as VEOS does not have an executing INIT process.

3. Partially Supported System Calls

Following is the list of system calls which are partially supported in VEOS.

SL No	System calls	Differences (Yes / No)
1	Clone	Yes
2	futex	Yes
3	prlimit	Yes
4	getrlimit	Yes
5	setrlimit	Yes
6	wait4	Yes
7	clock_nanosleep	Yes
8	timer_create	Yes
9	getitimer	Yes
10	madvise	Yes
11	mlock	Yes
12	munlock	Yes
13	mlockall	Yes
14	munlockall	Yes
15	setitimer	Yes
16	Ioctl	Yes
17	exit_group	Yes
18	getcpu	Yes
19	quotactl	Yes
20	set_tid_address	Yes
21	ustat	Yes

Note

❖ Partially supported syscalls exit_group(), futex(), getcpu(), quotactl(), set_tid_address() and ustat() are supported only for calling by library that is provided as a part of VEOS, such as glibc or musl-libc. Direct invocation by user program is not supported.

1. clone()

1. Clone() has a partial support in VEOS.

2. In clone() only the following combination of flags are supported:

SL No	Flags
1.	SIGCHLD
2.	CLONE_PARENT_SETTID | SIGCHLD
3.	CLONE_CHILD_SETTID | CLONE_CHILD_CLEARTID | SIGCHLD
4.	CLONE_VM | CLONE_VFORK | SIGCHLD
5.	CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SYSVSEM | CLONE_SIGHAND | CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID | 0

3. Other than the above mentioned flags, none of the other flags mentioned in the man-page of clone() is supported. Clone() will return <EINVAL> for all unsupported flags.

4. In the flags, no signal can be specified other than SIGCHLD. As mentioned in point-2 and point-3, clone will return <EINVAL> is any other signal is specified.

5. The maximum number of threads created for a process is 64 (including main process). Clone() will return < EAGAIN> if attempt is made to create more than 64 threads.

6. The maximum number of threads supported by VEOS is 1024. Clone() will return < EAGAIN> if attempt is made to create more than 1024 threads.

7. The raw system call for clone() in VEOS is as follows:

```
int clone(int flags, void *stack, pid_t *ptid, pid_t *ctid, void *tls,
void *guard_ptr)
```

As such, the fn and arg arguments of the clone() wrapper function are omitted.

8. clone() is VEOS-specific and should not be used in programs intended to be portable.

Note

❖ Clone() libc wrapper and raw system call is meant to be used by the library.

❖ Clone() libc wrapper and raw system call is not supposed to be used directly be the end-user. The end-user needs to use the high-level system calls and APIs available like fork(), vfork() and pthread_create() etc.

9. Clone() system calls returns the following errors
    a. EAGAIN: If more than 64 threads per process are created.
    b. EAGAIN: If more than 1024 threads (VEOS system wide) are created.
    c. EAGAIN: If more than 256 process (VEOS system wide) are created.

Note

❖ Refer the limitations of getrlimit() / setrlimit() for RLIMIT_NPROC handling in VEOS

10. After a process exhausts its maximum limit of open file descriptors, subsequent invocation of clone() system call will fail with errno set to EAGAIN.
11. After changing the default root directory of the calling process to that specified in path with chroot() system call, subsequent invocation of clone() will fail with errno set to EAGAIN.

2. futex()

1. futex() has a partial support in VEOS.

2. The futex() system call only supports the following futex operations.

SL No	Flags
1.	FUTEX_WAIT
2.	FUTEX_WAKE
3.	FUTEX_REQUEUE
4.	FUTEX_CMP_REQUEUE
5.	FUTEX_WAIT_BITSET
6.	FUTEX_WAKE_BITSET
7.	FUTEX_PRIVATE_FLAG
8.	FUTEX_CLOCK_REALTIME

3. Rest of the futex operations mentioned in the futex man-page are not supported by the futex system call. Futex() will return <EINVAL> for all unsupported futex operations.

4. Robust futex operations are not supported. Only normal futex calls are supported with above mentioned set of operations.

5. Priority inheritance futex (PI-futex) are not supported in VEOS.

6. futex() is VEOS-specific and should not be used in programs intended to be portable.

Note

❖ Libc does not provide a wrapper for this system call
❖ Bare futexes are not intended as an easy-to-use abstraction for end-users.
❖ Users of futex system call are assumed to be assembly literate and are aware of the source of the futex user space library and kernel space implementation.

- ❖ To achieve process and thread synchronization and locking, use higher-level programming abstractions implemented via futexes including POSIX semaphores and various POSIX threads synchronization mechanisms (mutexes, condition variables, read-write locks, and barriers).

3. prlimit() / getrlimit() / setrlimit()

1. prlimit() has a partial support in VEOS.
2. The following flags have different behavior in VEOS:

SL No	Flags	Behavior
1.	RLIMIT_CPU	As per the manpage, If the process continues to consume CPU time, it will be sent SIGXCPU once per second until the hard limit is reached. In VEOS, SIGXCPU will be sent only once.
2.	RLIMIT_NPROC	In VEOS hard limit and soft limit is not maintained for RLIMIT_NPROC. Prlimit() will show the values of the VH host kernel and will also will set the value to the VH host kernel. However in VEOS, we have the following limit for process and threads - Max number of process = 256 - Threads per process = 64 - Max number of threads = 1024 VEOS do not consider RLIMIT_NPROC value during creation of tasks (process/threads). For all privileged process (with CAP_SYS_RESOURCE capability) or unprivileged process, the limit will be as per the above defined values (i.e. 256, 64 and 1024).
3.	RLIMIT_NICE	This is not supported and EINVAL is returned
4.	RLIMIT_RTPRIO	This is not supported and EINVAL is returned
5.	RLIMIT_RTTIME	This is not defined in musl-libc and usage of this flag in VE program will result in compilation error, where as it is defined in glibc and available for user programs in (include/bits/resource.h) as: #define RLIMIT_RTTIME 15
6.	RLIM_NLIMITS	Macro RLIM_NLIMITS is defined as : While in case of musl-libc : #define RLIM_NLIMITS 15 While in case of glibc : #define RLIM_NLIMITS 16

3. If pid 1 is given as an argument it will return -1 and errno ESRCH will be set.
4. In getrlimit() / setrlimit(), errno EFAULT is never returned. EFAULT is not set when a pointer argument points to a location outside the accessible address space (limitation from MUSL LIBC 1.1.14, where getrlimit() / setrlimit() internally invoke prlimit), where as in case of glibc EFAULT is returned when pointer argument points to a location outside the accessible address space.

4. wait4()

1. wait4() has a partial support in VEOS.

2. Following flags are not supported in wait4

SL No	Flags
1.	__WCLONE
2.	__WALL

3. wait4 is not supported with above flags. This is due to limitation of clone() system call due to which we cannot create "clone" children. [A "clone" child is one which delivers no signal, or a signal other than SIGCHLD to its parent upon termination].
4. If a child is sent a terminating signal, example SIGFPE, SIGTERM, etc., SIGKILL is sent instead of the actual one. This leads to WTERMSIG(status) value = SIGKILL (9) instead of the actual one when the child process is waited upon in the parent.

5. clock_nanosleep ()

1. clock_nanosleep() has a partial support in VEOS.

2. In clock_nanosleep() only the following flags are supported:

SL No	Flags
1.	CLOCK_REALTIME
2.	CLOCK_MONOTONIC

3. CLOCK_PROCESS_CPUTIME_ID flag is unsupported. Clock_nanosleep () will return EINVAL for CLOCK_PROCESS_CPUTIME_ID flag.

6. timer_create()

1. timer_create() has a partial support in VEOS.

2. The timer_create() system call only supports the following flags.

SL No	Flags
1.	CLOCK_REALTIME
2.	CLOCK_MONOTONIC

3. Rest of the flags mentioned in the timer_create() man-page are not supported by the timer_create() system call. Timer_create() will return EINVAL for all unsupported flags.

7. getitimer()

1. getitimer() has a partial support in VEOS.

2. The getitimer() system call only supports the following flag.

SL No	Flags
1.	ITIMER_REAL

3. Rest of the flags mentioned in the getitimer() man-page are not supported by the getitimer () system call. Getitimer() will return EINVAL for all unsupported flags.

8. madvise()

1. madvise() system call will always return success in VEOS because paging is not supported in VEOS, but ported application may invoke madvise() system call.

9. mlock()

1. mlock() system call will always return success in VEOS because paging is not supported in VEOS, but ported application may invoke mlock() system call.

10. munlock()

1. munlock() system call will always return success in VEOS because paging is not supported in VEOS, but ported application may invoke munlock() system call.

11. mlockall()

1. mlockall() system call will always return success in VEOS because paging is not supported in VEOS, but ported application may invoke mlockall() system call.

12. munlockall()

1. munlockall() system call will always return success in VEOS because paging is not supported in VEOS, but ported application may invoke munlockall() system call.

13. setitimer()

1. setitimer() has a partial support in VEOS.

2. The setitimer() system call only supports the following flag.

SL No	Flags
1.	ITIMER_REAL

3. Rest of the flags mentioned in the setitimer() man-page are not supported by the setitimer() system call. Setitimer() will return EINVAL for all unsupported flags.

14. ioctl()

1. ioctl() has a partial support in VEOS.
2. On VEOS any non-tty request using ioctl() system call will not be served. In this case ioctl() will fail with errno set to EINVAL.

4. Not Supported System Calls

Following is the list of system calls which are not supported in VEOS.

SL No	System calls	Error returned upon system call invocation
1	get_robust_list	ENOTSUP
2	set_robust_list	ENOTSUP
3	unshare	ENOTSUP
4	set_thread_area	ENOTSUP
5	get_thread_area	ENOTSUP
6	prctl	ENOTSUP
7	setpriority	ENOTSUP
8	getpriority	ENOTSUP
9	sched_get_priority_max	ENOTSUP
10	sched_get_priority_min	ENOTSUP
11	sched_setparam	ENOTSUP
12	sched_getparam	ENOTSUP
13	sched_setscheduler	ENOTSUP
14	sched_getscheduler	ENOTSUP
15	clock_settime	EPERM
16	settimeofday	EPERM
17	add_key	ENOTSUP
18	request_key	ENOTSUP
19	keyctl	ENOTSUP
20	reboot	ENOTSUP
21	personality	ENOTSUP
22	sysfs	ENOTSUP
23	setns	ENOTSUP
24	io_setup	ENOTSUP
25	io_destroy	ENOTSUP
26	io_getevents	ENOTSUP
27	io_submit	ENOTSUP
28	io_cancel	ENOTSUP
29	perf_event_open	ENOTSUP
30	ptrace Note: This is different from ve_ptrace(). See section-5	ENOTSUP
31	remap_file_pages	ENOTSUP
32	set_mempolicy	ENOTSUP

33	get_mempolicy	ENOTSUP
34	migrate_pages	ENOTSUP
35	kcmp	ENOTSUP
36	fexit_module	ENOTSUP
37	mremap	ENOTSUP
38	times	ENOTSUP
39	adjtimex	ENOTSUP
40	clock_adjtime	EPERM
41	mbind	ENOTSUP
42	move_pages	ENOTSUP
43	uselib	ENOTSUP
44	_sysctl	ENOTSUP
45	create_module	ENOTSUP
46	get_kernel_syms	ENOTSUP
47	query_module	ENOTSUP
48	nfsservctl	ENOTSUP
49	getpmsg	ENOTSUP
50	putpmsg	ENOTSUP
51	afs_syscall	ENOTSUP
52	tuxcall	ENOTSUP
53	security	ENOTSUP
54	epoll_ctl_old	ENOTSUP
55	epoll_wait_old	ENOTSUP
56	vserver	ENOTSUP
57	swapon	ENOTSUP
58	swapoff	ENOTSUP
59	capget	Compilation Error when header file “capability.h” is used
60	capset	Compilation Error when header file “capability.h” is used
61	vmsplice	ENOTSUP

Note:

- sched_setparam() will return ENOSYS when called using musl-libc wrapper and will return ENOTSUP when called using sycall() or glibc wrapper.
- sched_getparam () will return ENOSYS when called using musl-libc wrapper and will return ENOTSUP when called using sycall() or glibc wrapper.
- sched_setscheduler () will return ENOSYS when called using musl-libc wrapper and will return ENOTSUP when called using sycall() or glibc wrapper.
- sched_getscheduler () will return ENOSYS when called using musl-libc wrapper and will return ENOTSUP when called using sycall() or glibc wrapper.

5. VE Ptrace System Call

1. For VE program ptrace() system call is not supported it will return –ENOTSUP.
2. Only VE debugger can use ptrace() system call by invoking ve_ptrace() instead of ptrace().
3. Traced VE processes should be present on a single node.
4. Single VE debugger can’t trace multiple VE processes of different VE nodes.
5. New ptrace request “PTRACE_STOP_VE” has to be invoked by VE debugger to stop the VE process/thread when debugger comes out of any wait() system call family.

```
ve_ptrace(PTRACE_STOP_VE, pid, 0, 0);
```

6. Followings are the requests that are not supported:

SL No	Ptrace Request
1.	PTRACE_SYSEMU
2.	PTRACE_SYSEMU_SINGLESTEP
3.	PTRACE_O_TRACEEXEC
4.	PTRACE_O_TRACEVFORKDONE

7. Ptrace request PTRACE_GETFPREGS/ PTRACE_SETFPREGS will get/set the vector registers.
8. VE Debugger can’t invoke PTRACE_TRACEME as it has special handling in VE environment.

6. Generic VEOS Difference Points / Limitations

1. Consider the following difference point if trap corresponding to FE_DIVBYZERO is disabled.
   When a child process does an integer divide-by-zero computation, example (5/0), the Floating point exception is not generated, and hence child does not get killed. Due to this, correct termination status of child is not received in parent while doing wait4 (or any other wait family system call).

   However in Linux, integer divide-by-zero computation raises floating point exception even if the trap corresponding to FE_DIVBYZERO is disabled, therefore if child process terminated by such exception will always returns expected termination status.

2. In VE architecture it may happen that VE task will be terminated while executing more than 3 signal handler in nested manner.

3. If VE process explicitly sends signal 32 to itself using tkill()/tgkill(), then whole thread group will be terminated rather than target thread. This is in compliance with musl libc but with glibc only target thread will be deleted.

4. In VEOS, when core pattern (/proc/sys/kernel/core_pattern) contains pipe (|) as first symbol then core file will be created at current working directory of VE process. The file name of core file will be "core.xxxx.ve" if the pid is xxxx.

5. In VEOS, while creating coredump only patterns “%”, “p”, ”h” are supported if mentioned in core_pattern file. Symbols other than this will be ignored.

6. If a VE process is getting traced and an attempt is made to read si_code after setting breakpoint etc. si_code will be set to TRAP_BRKPT always (si_code populated for SIGTRAP signals like TRAP_TRACE, TRAP_BRANCH, TRAP_HWBKPT will not be set.

7. If user has performed integer divide by zero or floating-point divide by zero, while signal handler is registered for SIGFPE signal then si_code will be set to FPE_FLTDIV for both cases. For Linux, si_code FPE_INTDIV is set when integer divide by zero is performes and si_code FPE_FLTDIV is set when floating-point divide by zero is performed.

8. When a VE process receives any terminating signal then to end user it will appear that process is terminated using SIGKILL because for every terminating signal we terminate the pseudo process using SIGKILL signal.

9. In multiprocess environment where parent sends terminating signal to child and wait to get the status using WIFSIGNALED() then parent will always receive SIGKILL as terminating signal when it gets the status as pseudo child process is terminated with SIGKILL(as mentioned in above point) and also wait() syscall is offloaded to kernel.

10. In VEOS when signal information is received by a signal handler (invoked due to some exception) then “si_addr” filed if siginfo struct always stores some relevant instruction address (ICE register value). However in linux, for some exceptions si_addr stores the address of instruction where the exception occurred while for some it stores the address where fault has occurred.

According to VE HW spec, depending on the exception cause, this register may hold the address of the instruction which caused the exception, or the address of a branch instruction lastly executed before the exception was reported.

In case of non-masked arithmetic exception as mentioned below, ICE saves the address of the instruction to cause the exception

- Divide by zero
- floating point overflow exception
- Floating-point underflow exception
- Fixed-point overflow exception
- Invalid operation exception
- Inexact exception,

In case of following exception, ICE saves the address of a branch instruction lastly executed before the exception was reported.

- Memory protection exception
- Missing page exception
- Missing space exception
- Memory access exception
- Host memory protection exception
- Host missing page exception
- Host missing space exception
- Host memory access exception
- I/O Access Exceptions
- Illegal data format exception
- Illegal instruction format exception

11. In VEOS syscalls read() and pread64(), futex(),recvfrom(),recvmsg(), recvmmsg(), sendmsg(), sendmmsg(), sendto(), accept(), accept4(), connect() can never be restarted automatically after the interruption with signal.

12. In VEOS if a VE process receives unrecoverable h/w exception, and VE process has installed a signal handler for signal mapped to h/w exception then signal handler will be invoke once and later VE process will be terminated with the signal to which exception has been mapped. In Linux process generates exception and user has installed handler for the same, then after signal handler is executed same instruction which caused fault will be executed and hence signal handler will be invoked indefinitely.

13. In VEOS with musl libc, signal number 34 and 35 are reserved. If user program tries to register signal handler for these signal it will receive EINVAL. There is no such limitation for glibc.

14. In VEOS `SIGCANCEL` signal generated for a thread through `pthread_cancel()` API may not deliver instantly, if the thread is executing the blocking system call like `sleep(2)` etc. (means thread may not cancel/exit instantly) . The delivery of `SIGCANCEL` will be deferred until the execution of blocking system call etc.

15. The maximum number of requests that can be handled by VEOS concurrently is 1056. The requests comprises of:
    - Requests from VE tasks (process / threads)
    - Requests from ported RPM commands
    - Requests from GDB

Note

- ❖ Maximum number of VEOS worker threads = 1056
- ❖ Maximum number of VE tasks = 1024

16. If logging (log4c) is enabled, VE process gets 6 as the first file descriptor number when allocated using open() / socket() like system calls. FDs from 3 to 5 are reserved for VEOS. If logging (log4c) is disabled, VE process gets 5 as the first file descriptor number when allocated using open() / socket() like system calls. FDs from 3 to 4 are reserved for VEOS.

17. In VEOS architecture, if user demands or try to fetch the current state of any task, then user is required to use the ve rpm specific commands rather than using proc fs interface as on Linux environment. Similarly in order to fetch VE process execution information, user need to use VE specific rpm commands(ps, etc) rather than VH rpm commands.

18. In VE architecture information regarding the current executing task (self) by accessing `/proc/self/` directory is not supported. For e.g. in VE architecture accessing soft link `/proc/self/exe` will not return binary path for current executing VE task.

19. If the system call is interrupted by a signal handler, like nanosleep/pselect veos returns -1, sets errno to EINTR, and writes the remaining time into the structure pointed to by “rem” unless “rem” is NULL But due to off-loading & context switch overheads in veos design the the "rem" precision in microseconds will vary on VEOS.

20. VE architecture supports Large page(2MB) and Huge Page(64MB) whereas corresponding VH is having a page size of 4KB.

21. In VEOS, SIGCONT signal is non-maskable (Like SIGKILL and SIGSTOP). Any request to mask SIGCONT signal through system call (that updated signal mask set of task) like `sigprocmask(2)`, `sigaction(2)`, `pselect(2)/pselect6(2)`,`ppoll(2)`,`epoll_pwait(2)` etc. will be simply ignored and success will be returned to the user program.

22. set-user-id bit and set-group-id bit of VE programs do not take effect i.e. the effective user ID is not changed to the owner of the VE program file even if set-user-id bit is set. Similarly the effective group ID is not changed to the group of the VE program file even if set-group-id bit is set.

23. In VE architecture a program can write maximum up to 2GB-4KB buffer data through write series system calls like write(2), writev(2) etc. Hence maximum return value of write series system calls on VE architecture would be 2GB-4KB. Maximum value will not be dependent on page size of VE architecture.

24. According to POSIX standards I/O is intended to be atomic to ordinary files and pipes and FIFOs. Atomic means that all the bytes from a single operation that started out together end up together, without interleaving from other I/O operations. However VE architecture supports both ATOMIC & NON-ATOMIC I/O mode. By default NON-ATOMIC mode is enabled on VE in order to reduce memory consumption at VH side.
    - ATOMIC I/O mode can be enabled by exporting VE_ATOMIC_IO=1.

25. For VE, the specification about resource limits of VE and corresponding pseudo process with ve_exec command, VH prlimit command, VE prlimit() system call and ported prlimit command is mentioned in below table:

Categories	**'ve_exec' command**	**VH prlimit command**	**VE prlimit command**	**VE prlimit system call**
Category-1: **FSIZE, LOCKS, MSGQUEUE, NPROC, MEMLOCK, RTTIME, NOFILE**	When we run a new VE process it would inherit all the resource limit of the corresponding pseudo process (ve_exec).	Suppose VE process is running and we change the resource limit of pseudo process by VH prlimit command, then it is also reflected for corresponding VE process.	Suppose VE process is running and we change the resource limit of VE process by VE prlimit command, then it is reflected also for corresponding pseudo process.	Suppose VE process is running and we change the resource limit of VE process by VE prlimit system call, then it is also reflected for corresponding pseudo process.
Category-2: **AS, CPU, CORE, DATA, RSS, SIGPENDING**	When we run a new VE process it would inherit the resource limit of the corresponding pseudo process.	Suppose VE process is running and we change the resource limit of pseudo process by VH prlimit command, then it is **not** reflected for corresponding VE process.	Suppose VE process is running and we change the resource limit of VE process by VE prlimit command, then it is **not** reflected for corresponding pseudo process.	Suppose VE process is running and we change the resource limit of VE process by VE prlimit system call, then it is **not** reflected for corresponding pseudo process.
Category-3: **STACK**	STACK limit for new VE process will either be set as "unlimited" or the values passed through "VE_STACK_LIMIT" environment variable.	Suppose VE process is running and we change the resource limit of pseudo process by VH prlimit command, then it is **not** reflected for corresponding VE process.	Suppose VE process is running and we change the resource limit of VE process by VE prlimit command, then it is **not** reflected for corresponding pseudo process.	Suppose VE process is running and we change the resource limit of VE process by VE prlimit system call, then it is **not** reflected for corresponding pseudo process.

- NICE and RTPRIO resource limits are not supported for VE.
- RSS limit has no effect on VH & VE with ulimit, prlimit command or prlimit system call.