# MOSS
# "A Soft Blanket over the Rocky Slopes of DOS"

A DOS extender based on the Flux OS toolkit

Bryan Ford

`baford@cs.utah.edu`

Sleepless Software
420 South 1100 East
Salt Lake City, UT 84102

and the

Computer Systems Laboratory
Department of Computer Science
University of Utah
Salt Lake City, UT 84112

February 1, 1996

**Abstract**

MOSS is a new DOS extender with the following major features:

- Can be used royalty-free in commercial and non-commercial products.
- Full support for DPMI, VCPI, XMS, and "raw" DOS modes.
- Supports up to 2GB of virtual and physical memory.
- Demand-loading of executables for quick startup times.
- Supports POSIX low-level file I/O as well as ANSI C I/O.
- Processor exceptions can be delivered as POSIX signals.
- Hardware interrupts can be delivered as POSIX.1b real-time queued signals (except under DPMI).
- Traditional-DOS-extender-like interrupt revectoring also available under all environments.
- Supports the POSIX.1b memory locking API (`mlock` et al).
- Remote source-level debugging over a serial line using GDB.
- Full cross-development from Linux and other Unix-compatible OS's.
- Uses i386 ELF object files and executables.
- Allows a program *and* any associated data files to be attached to MOSS, forming one big DOS executable.
- Written almost entirely in easy-to-modify C code.
- Can be compiled *completely* using the freely-available GNU development tools (i.e. doesn't require a commercial compiler/assembler to handle the 16-bit x86 code).

MOSS is available from '`ftp://flux.cs.utah.edu/flux/moss/`'.

# Contents

# 1 Introduction

MOSS is new DOS extender primarily targeted for people who must write DOS programs for some reason or another, but want to have as little to do with DOS as possible. (Like us, for example.) To this end, MOSS is primarily intended for cross-development from "real" OS's rather than for native development under DOS. MOSS's remote source-level debugging support makes it especially convenient to write DOS programs if you have a DOS test machine sitting next to a Un*x-compatible development machine. Furthermore, MOSS's API attempts to hide much of the ugliness of DOS under a Un*x/POSIX veneer, for example by providing POSIX-like file I/O, POSIX signals for intercepting processor exceptions and hardware interrupts, POSIX `mmap()` for mapping device memory such as the video buffer, etc.

We (Sleepless Software) created MOSS for a variety of reasons. One was pragmatic: we had a large Linux program needing to be ported to DOS, and none of the existing DOS extenders (commercial or free) really suited our purposes. They either (a) did not allow unencumbered, royalty-free use with commercial applications, (b) would have required too much porting effort on our part, or (c) did not have good enough cross-development support.

Another reason we created MOSS was to exercise the flexibility of the Flux OS toolkit currently under development at the University of Utah, from which a great deal of MOSS's code is derived. This toolkit is a collection of infrastructure and reusable code for fast and flexible development of operating systems. Since a DOS extender is really just a kind of mini-operating system (though perhaps of a rather unusual variety), we decided that creating a DOS extender based on this toolkit would be a good experiment to see how much functionality could be shared and reused even between such diverse environments. We believe we have succeeded quite well in this experiment: more than 50% of the code implementing MOSS itself is generic code shared by other OS's and OS-like programs that the Flux project has developed, and almost all of the C library is taken directly, unmodified, from FreeBSD. For more information on the Flux project and the Flux OS toolkit, see our web pages at '`http://www.cs.utah.edu/projects/flux/`'.

## 1.1 License

MOSS itself is distributed under the terms of the GNU General Public License. This means that you can use, modify, and redistribute MOSS however you like; however, if you want to distribute a modified version of MOSS, you must release your modified MOSS source code under the same license.

However, distributing MOSS along with a program that uses it is explicitly classified as "mere aggregation" rather than "linking" for licensing purposes, even if MOSS and the program are aggregated into the same executable file. This means that if you distribute commercial software in executable form with MOSS attached to the beginning of the executable, you must ensure that the source to MOSS itself is available (if you have modified it), but you do not need to distribute the source to the attached program.

Furthermore, the C library used by MOSS is based on the BSD C library, and does not contain any GPL code, so linking your application with it will not "contaminate" your application in any way. You only need to meet the conditions of the U.C. Berkeley and the (very similar) University of Utah copyrights under which the C library code is distributed.

## 1.2 Support

Since we're providing this product for free, we don't have the resources to provide a lot of support for it. Certainly we will incorporate any reasonably well-written patches or additions that anyone sends to us, and we will continue to develop MOSS in directions that are useful to us. We will also be glad to hear comments, suggestions, bug reports, etc.; however, if there's a bug you need fixed right away or a new feature you want implemented, you may have to do it yourself.

We have created a mailing list for all discussion related to using and further developing MOSS. To get on the list, send a message to '`moss-request@flux.cs.utah.edu`'. The mailing list itself is at '`moss@flux.cs.utah.edu`'.

# 2 Using MOSS

This section describes briefly how to get started using MOSS.

### 2.1 Tools required

To compile MOSS applications (or MOSS itself), you will need a GCC cross-compiler configured for the target 'i386-moss'. Ready-made MOSS cross-compiler binaries are available at our FTP site for Linux (ELF) and FreeBSD. Simply download the tar files and unpack them from your root directory; they will go into the appropriate places in /usr/local. For example, if your development machine is running Linux:

```
$ cd /
$ tar xvzf /tmp/moss-0.90-bin-linux.tar.gz
```

For other environments, you may have to build your own tools; see Section 3 for detailed instructions.

### 2.2 Building and running MOSS applications

To cross-compile a MOSS program, simply use 'i386-moss-gcc' as the name of the C compiler instead of just 'gcc' or 'cc'. Alternatively, put '/usr/local/i386-moss/bin' at the beginning of your PATH; then you will get the MOSS version of 'gcc', 'ld', 'as', etc., automatically.

To run a program compiled for MOSS, place it in the same directory as MOSS.EXE, give it the name A.OUT, and run 'MOSS'. (Yes, the A.OUT name is currently hard-wired into MOSS; it's just a horrible kludge that we haven't gotten around to fixing yet.) Any command-line options you give will be passed directly to the program, as well as the current environment variables.

### 2.3 Example MOSS application

For example, here is how you can build the ubiquitous "hello world" test program:

```
$ cat >test.c
main()
{
  printf("hello world!\n");
}
^D
$ i386-moss-gcc test.c
```

Now you should have a file called 'a.out' in your current directory. Copy that file to your DOS machine, along with MOSS.EXE, the DOS extender itself, which should be in /usr/local/i386-moss/bin/moss.exe (or the corresponding path if you installed MOSS in a different place). For example, if you can FTP from your DOS machine to your development machine:

```
C:\> ftp develmachine
Username: me
Password:
ftp> get a.out
ftp> get moss.exe
ftp> quit
C:\> moss
hello world!
C:>
```

## 3 Building MOSS

Note that if you download the pre-built MOSS binary distribution, you don't have to build MOSS yourself unless you want to change it. In addition, even if you do want to modify MOSS itself, you can generally still use the pre-built cross-compiler tools (GCC and binutils) without recompiling them as well. This section contains step-by-step instructions to build everything, including GCC, binutils, and MOSS; if you only want to build MOSS and use the pre-built compiler tools, just skip to Section 3.

Besides the basic cross-development tools above, you will also need GNU 'make' in order to compile MOSS yourself. If you are using Linux, you probably already have GNU make; if you are on BSD or another system, you will need to get a copy (e.g. from ftp://prep.ai.mit.edu/pub/gnu). Of course, you can use any 'make' you want (or none at all) to build MOSS applications.

Throughout this process, we assume you want to install the cross-compiler setup in /usr/local. (All of the installed cross-compiler binaries and related utilities will have names prefixed with 'i386-moss-', so they should not conflict with other programs you may have there.) If you would like to install the tools somewhere else, just supply the appropriate --prefix options to the configure commands below.

## 3.1 Building binutils

First, download and unpack the GNU binutils-2.6 distribution, apply the necessary patch available on the MOSS FTP site, configure binutils for the target i386-moss, and build and install it:

```
$ tar xvzf binutils-2.6.tar.gz
$ cd binutils-2.6
$ patch -p1 <../binutils-2.6-pch
$ ./configure --target=i386-moss
$ make
$ make install
$ cd ..
```

## 3.2 Building GCC

Now you'll need to build a GCC cross-compiler for MOSS. Unpack the GCC 2.7.2 distribution, apply the patch for it available on the MOSS FTP site, configure it for the target i386-moss, and build it.

The build will fail while trying to make libgcc. This is normal, and is because libgcc needs some of MOSS's header files which haven't been installed yet; and we can't install MOSS until we have a cross-GCC to compile it with! However, libgcc isn't needed to build MOSS itself, only to build programs that use MOSS. Therefore, at this point, just build and install the compiler and ignore the error produced when GCC tries to build libgcc.

When installing the new compiler, you will need to supply the option INSTALL_LIBGCC= on the make command line; otherwise GCC will try to install the nonexistent libgcc before installing the i386-moss-gcc driver program, and you will be left with an incomplete and nonfunctional compiler setup.

```
$ tar xvzf gcc-2.7.2.tar.gz
$ cd gcc-2.7.2
$ patch -p1 <../gcc-2.7.2-pch
$ ./configure --target=i386-moss
$ make LANGUAGES=c
$ make install LANGUAGES=c INSTALL_LIBGCC=
$ cd ..
```

## 3.3 Building MOSS

You'll need to use GNU make to build MOSS itself. GNU make is usually the standard version of make on Linux machines; on FreeBSD it is usually available as 'gmake'. You can supply the -r option to speed up the build process somewhat by disabling the default rules built into make; however, this isn't required.

Note that the configure script for MOSS is not in the top-level directory of the MOSS source distribution, but instead is one level deep, in the moss subdirectory. Also, you will need to include some additional options on the configure command line, as shown below. This weirdness is a temporary problem and should be fixed soon.

If you plan to do significant development on MOSS itself, you may want to add --enable-debug to the configure command line; this turns various internal assertions and debugging checks. See Section 3.6 below for more information on this and other compile-time options.

Here is a complete example demonstrating how to configure and build MOSS:

```
$ tar xvzf moss-0.90.tar.gz
$ cd moss-0.90
$ moss/configure --target=i386-moss --with-bsdsrc=freebsd/usr/src --prefix=/usr/local/i386-moss
$ gmake -r install
$ cd ..
```

**Note**: GCC 2.7.2 has an optimization bug that makes MOSS malfunction if it is compiled with `-O2`. Therefore, you should always compile MOSS with `-O1` until this GCC bug has been fixed. (The MOSS distribution is set up to use `-O1` by default.)

### 3.4 Building the rest of GCC

Now that MOSS is built and installed, you will need to go back and build the rest of GCC (in particular, `libgcc`, which couldn't be built above). It should succeed now that all the needed `libc` header files are available in the installation directories.

```
$ cd ../gcc-2.7.2
$ make install LANGUAGES=c
```

Now you should be ready to cross-compile real programs for use with MOSS. Just use `i386-moss-gcc` as you would use a native `gcc`.

### 3.5 MOSS directory structure

MOSS currently has a somewhat weird source code organization, and contains a lot of stuff that isn't actually used by MOSS; this is due to our current development environment, and should be fixed shortly. In the top-level directory of the source distribution you will find four subdirectories: `moss`, `mach4`, `mach4-i386`, and `freebsd`. `moss` is the directory containing the source to MOSS itself: `moss/kernel` contains the DOS extender kernel, and `moss/libc` contains the MOSS-specific parts of the C library. The other directories contain other sources needed by MOSS: `mach4` and `mach4-i386` contain the subset of the Flux OS toolkit used by MOSS, and `freebsd` contains a subset of the FreeBSD source distribution from which the MOSS C library is constructed.

### 3.6 Compile-time options

As mentioned above, you can turn on various internal assertions and debugging features by adding `--enable-debug` to MOSS's `configure` command line (which causes the symbol `DEBUG` to be defined while compiling MOSS). However, if you compile with debugging enabled, all the additional assertions and checks compiled in may make the MOSS executable grow too big for the 64K small-model program size limit under DOS (`i386-moss-ld` will report an error message while creating the DOS executable). If this happens, just disable DPMI or VCPI support (or both) as described below, depending on the DOS environment you're testing in, and the executable should then be small enough. (We know of a fairly straightforward way to get rid of the 64K size limit on the MOSS executable, but we haven't implemented it yet; contact us on the MOSS mailing list if you would like to do this.)

There are several other compile-time configuration options you can set; they are located in the source file `moss/kernel/raw/config.h`. The `ENABLE_DPMI` and `ENABLE_VCPI` options enable DPMI and VCPI support, respectively; the `ENABLE_RGDB` option enables remote source-level debugging.

`ENABLE_CODE_CHECK` is an option only used for debugging MOSS itself; it turns on a kludge whereby MOSS copies its entire code segment elsewhere on startup and checks it on exit (and at any other times you call the `code_check()` function) to detect code trashing bugs. This is particularly useful for MOSS itself because it is not possible to unmap page 0 of MOSS's "address space" to catch null pointer references and such, or to make the text segment read-only, as is normally done with Un*x executables and 32-bit programs running under MOSS. You should probably leave this option disabled unless you're into serious MOSS development.

# 4 MOSS system calls

For the most part, MOSS presents a standard ANSI/POSIX application programming interface; since its C library is based on the BSD C library, all of the functions in the BSD C library are available. This document does not attempt to describe these functions; you can find documentation on the ANSI/POSIX interfaces in many easily-available books and on-line repositories. If you're cross-developing MOSS programs from a Un*x-like system, you should be able to get all the documentation you need simply using `man`.

Of course, if you try to use some of the more exotic C library functions that depend on having a "real" Un*x-like OS underneath (which DOS definitely isn't), the program probably won't work quite right. Typically one of two things will happen: either the program will fail to link due to an undefined symbol, or the unsupported function will fail with an error at run-time. Undefined symbols at link time are usually caused by the absence of a MOSS "system call" corresponding to a particular BSD system call that the C library function you are trying to use depends on.

The following sections describe the system calls supported by MOSS. Most of these system calls correspond exactly to BSD system calls and/or standard POSIX functions; for these, only the differences between MOSS's behavior and the "standard" behavior are described.

## 4.1 alloc_dma: allocate DMA memory

SYNOPSIS

`unsigned` **alloc_dma**(`size_t` *size*, `int` *align_bits*, `unsigned` *align_ofs*);

DESCRIPTION

This function can be used to allocate memory from the low 16MB of physical memory accessible to the PC's built-in DMA controller.

This function only allocates the requested physical memory; in order to gain access to it within the application you must map it into the application's address space using `mmap`.

PARAMETERS

*size*: The minimum number of bytes of memory required.

*align_bits*: If greater than zero, the lowest *align_bits* bits of the physical start address of the returned memory will be equal to the specified *align_ofs*.

*align_ofs*: The low-order bits of the required alignment. If *align_ofs* is zero, the returned block will be naturally aligned according to *align_bits*. Must be less than $2^{align\_bits}$.

RETURN VALUE

Returns the physical address of the memory allocated if successful, or 0 on error, in which case `errno` indicates the error.

## 4.2 alloc_dos: allocate low DOS memory

SYNOPSIS

`unsigned` **alloc_dos**(`size_t` *size*, `int` *align_bits*, `unsigned` *align_ofs*);

DESCRIPTION

This function can be used to allocate memory from the low 1MB of physical memory directly accessible to DOS.

This function only allocates the requested physical memory; in order to gain access to it within the application you must map it into the application's address space using `mmap`.

PARAMETERS

*size*: The minimum number of bytes of memory required.

*align_bits*: If greater than zero, the lowest *align_bits* bits of the physical start address of the returned memory will be equal to the specified *align_ofs*.

*align_ofs*: The low-order bits of the required alignment. If *align_ofs* is zero, the returned block will be naturally aligned according to *align_bits*. Must be less than $2^{align\_bits}$.

RETURN VALUE

Returns the physical address of the memory allocated if successful, or 0 on error, in which case `errno` indicates the error.

## 4.3 close: close an open file descriptor

SYNOPSIS

`int` **close**(`int` *fd*);

DESCRIPTION

This system call implements the POSIX `close` function, which closes an open file descriptor.

PARAMETERS

*fd*: The file descriptor to close.

RETURN VALUE

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

## 4.4 _exit: terminate the program without cleanup

SYNOPSIS

`void` **exit**(`int` *rc*);

DESCRIPTION

This system call implements the POSIX `_exit` function, which terminates the program without calling C library cleanup functions such as `atexit()` handlers. However, all memory allocated by the program is still freed, open file descriptors are closed, etc.

PARAMETERS

*rc*: The exit code to return to DOS.

### 4.5 fstat: return statistics on a file descriptor

SYNOPSIS

`int` **fstat**(`int` *fd*, `struct stat` **buf*);

DESCRIPTION

This system call implements the POSIX `_exit` function, which returns various information about a file given an open file descriptor referring to the file.

The most common use of `fstat` is to find the size of a file; the MOSS version of `fstat` works fine for this purpose. In addition, MOSS fills in most of the other fields with values that are "sensible" if not always completely accurate. For example, it provides the file size in blocks assuming a typical MS-DOS block size of 512 bytes. However, fields that have no real meaning on DOS, such as the device and inode numbers, are simply filled with zero.

PARAMETERS

*fd*: File descriptor of the file to examine.

*buf*: Buffer to fill with the file statistics.

RETURN VALUE

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

## 4.6 gettimeofday: read the current system time

SYNOPSIS

`int` **gettimeofday**`(struct timeval *`*tv*, `struct timezone *`*tz*);

DESCRIPTION

This system call implements the BSD `gettimeofday` function, which reads the current system time, and is used by the BSD C library to implement the POSIX `time` function.

Because MOSS must implement this function as two separate DOS calls, one to get the date and the other to get the time, there is a slight chance of a race condition in which the returned value will be a day off if the function is called exactly at midnight. Oh well.

More seriously, the math used to calculate the Un*x-style `timeval` may currently be somewhat broken, producing inaccurate dates; this function is basically untested.

PARAMETERS

*tv*: If non-NULL, the specified structure is filled with the current time of day.

*tz*: Under BSD, if this parameter is non-NULL, the current timezone is stored in the provided structure. However, under MOSS, this paramter is not supported; if the application passes a non-NULL paramter, the `gettimeofday` call will fail.

RETURN VALUE

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

## 4.7 irq_free: free a previously-allocated hardware interrupt

Synopsis

int **irq_free**(int *irq*);

Description

Releases a hardware interrupt that was previously intercepted with `irq_request`, once again allowing the normal DOS handler to service the interrupt. If the interrupt was disabled before `irq_request` was called, then `irq_free` will cause it to be disabled again.

Parameters

*irq*: The hardware IRQ number to release, 0–15.

Return Value

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

### 4.8 irq_request: allocate and handle a hardware interrupt vector

SYNOPSIS

> `int` **irq_request**(`int` *irq*, `int` *pri*, `struct sigevent *`*ev*);

DESCRIPTION

This function allows a MOSS application to intercept one of the PC's hardware interrupts and service it in the application itself.

There are two ways the application can service interrupts, indicated by *ev*`->sigev_notify`:

- `SIGEV_SIGNAL`: Intercepted hardware interrupts will be dispatched to the application as POSIX signals. *ev*`->sigev_signo` specifies the signal number. An opaque value can be provided in *ev*`->sigev_value`; this value will be passed back to the application's handler when it is invoked, identifying to the hardware interrupt that occurred.

  MOSS will automatically clear and disable the hardware interrupt in the PC's programmable interrupt controller (PIC) until the signal handler finishes and returns; then it will automatically re-enable the hardware interrupt. Thus, the interrupt handler should never need to access the PIC registers directly. Of course, the interrupt handler itself must deal with any interrupt enable/disable flags specific to the device actually generating the interrupt.

  Dispatching interrupts as signals is not supported under DPMI. (This is because of a deficiency in the DPMI specification, not a bug in MOSS.)

- `SIGEV_FASTINT`: Intercepted hardware interrupts will be dispatched directly to an application interrupt handler, bypassing the signal mechanism entirely. *ev*`->sigev_handler` must point to the handler to invoke, and *ev*`->sigev_stack` must contain the initial stack pointer value indicating the stack to be used by the interrupt handler. The provided stack must have at least 32 bytes available in addition to the requirements of the handler itself.

  Fast interrupt handlers must run to completion without causing any processor exceptions. All memory touched by the interrupt handler, including its code, stack, and any other data it references, must be locked down using `mlock` or `mlockall`. All interrupts will be disabled on entry to the handler, and the handler must leave interrupts disabled for the duration. MOSS will clear the interrupt in the PIC; the handler should not touch the PIC. Only one fast interrupt handler will ever be invoked at once (they can't be stacked), so multiple interrupt handlers can share the same stack.

The `irq_request()` function causes the specified hardware interrupt to be enabled if it wasn't enabled already. Therefore, the interrupt handler must be ready to service interrupts when this routine is called.

The main advantages of dispatching hardware interrupts as signals are:

- You can call any MOSS system calls, DOS calls, or signal-safe C library functions, from within the signal handler. This means, for example, that you can print messages to the screen from within a signal handler.
- If you are debugging the application remotely, you can set breakpoints in signal handlers, single-step through them, etc.
- It is not necessary to lock down all the memory touched by the signal handler: if a page fault occurs while the handler is running, it will be serviced normally.
- Multiple interrupts can be serviced at once (signal handlers can be stacked).

The advantages of using the `FASTINT` mechanism are:

- Dispatching `FASTINT` handlers is somewhat faster, as you might expect.
- Interrupts can be handled during MOSS or DOS calls, providing a much lower worst-case interrupt latency.

- `FASTINT` handlers work under DPMI as well as the “plain” DOS modes.

In general, you may want to use signal handlers to catch and handle hardware interrupts during debugging, because of their greater convenience, and switch to using `FASTINT` handlers for the “release” version of the program to achieve maximum performance and DPMI compatibility.

### Parameters

*irq*: The hardware IRQ number to intercept, 0–15.

*pri*: Currently unused.

*ev*: A `sigevent` structure describing the action to take when the interrupt occurs.

### Return Value

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

## 4.9 kill: send a signal to a process

SYNOPSIS

`int` **kill**(`pid_t` *pid*, `int` *sig*);

DESCRIPTION

This system call implements the POSIX `kill` function, which sends a signal to a process.

Since MOSS currently handles only one process, the only valid values of *pid* are 0 (the "current" process) and 1 (process 1, which under MOSS is "the" process). Thus, `kill` can only be used by the application to send signals to itself.

However, any valid POSIX or BSD signal can be sent, and they will cause any installed signal handlers to be invoked if appropriate. Since job control doesn't really make sense under MOSS, signals whose default action would normally be to stop the process instead merely terminate the process under MOSS (unless a signal handler has been installed).

PARAMETERS

*pid*: The process to send the signal to.

*sig*: The signal number to send.

RETURN VALUE

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

## 4.10 lseek: reposition current offset in a file

SYNOPSIS

`off_t` **lseek**(`int` *fd*, `off_t` *offset*, `int` *whence*);

DESCRIPTION

This system call implements the POSIX `lseek` function, which changes the current file offset in a file descriptor.

PARAMETERS

*fd*: The file desriptor to reposition.

*offset*: The file offset to seek to relative to the base specified by *whence*.

*whence*: The base from which *offset* is relative:

- `SEEK_SET`: Seek relative to the beginning of the file.
- `SEEK_CUR`: Seek relative to the current position.
- `SEEK_END`: Seek relative to the end of the file.

RETURN VALUE

Returns the new absolute file position if successful, or -1 on error, in which case `errno` indicates the error.

## 4.11 mlock: lock a region of memory

Synopsis

`int` **mlock**(`const void *`*addr*, `size_t` *size*);

Description

This system call implements the POSIX.1b `mlock` function, which locks down a range of a program's memory so that page faults will not occur in that area. This is needed by programs with real-time requirements, or when installing "fast" (non-signal-based) hardware interrupt handlers.

Neither the start address of the specified region nor its length need to be page-aligned; this function will lock all pages overlapping the specified region.

Parameters

*addr*: The start address of the region to lock.

*size*: The size of the region to lock, in bytes.

Return Value

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

### 4.12 mlockall: lock all memory

SYNOPSIS

`int` **mlockall**(`int` *flags*);

DESCRIPTION

This system call implements the POSIX.1b `mlockall` function, which locks down all of a program's memory. The *flags* argument specifies whether to lock down memory the program has already allocated (`MCL_CURRENT`), memory the program allocates in the future (`MCL_FUTURE`), or both (`MCL_CURRENT | MCL_FUTURE`).

MOSS currently only supports `MCL_FUTURE`; attempting to call `mlockall` with `MCL_CURRENT set` will cause the DOS extender to panic. (`MCL_CURRENT` isn't hard to implement; we just haven't done it yet.)

Also, note that `munlockall` is currently unimplemented.

PARAMETERS

*flags*: The flags specifying whether to lock down currently allocated memory, memory allocated in the future, or both.

RETURN VALUE

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

### 4.13 mmap: map a file's contents into memory

SYNOPSIS

`void *`**mmap**(`void *`*addr*, `size_t` *len*, `int` *prot*, `int` *flags*, `int` *fd*, `off_t` *offset*);

DESCRIPTION

This system call implements the POSIX.1b `mmap` function, which maps a file into the program's address space.

Currently, MOSS only supports this function on the "magic" file '`/dev/mem`'; it will fail if used on any other file. You can `mmap /dev/mem` in order to gain direct access to physical memory, which is not normally available to the program running under MOSS.

Note that the `munmap` function is not currently supported. (Actually, it is available, but it simply returns success without doing anything.)

PARAMETERS

*addr*: The desired address at which to map the file. Currently MOSS always ignores this paramter.

*len*: Size in bytes of the file region to map.

*prot*: Permissions to apply to the mapped memory. Currently MOSS simply ignores this parameter and always provides full permissions.

*flags*: Flags describing how to map the file. Currently MOSS simply ignores this parameter and behaves as if `MAP_SHARED` was specified.

*fd*: The file descriptor of the file to map. Currently this must be a file descriptor opened on the special file `/dev/mem`.

*offset*: Starting offset in the file to map. For `/dev/mem`, this is the physical address to map from.

RETURN VALUE

Returns the start address of the mapped region if successful, or `(void*)-1` on error, in which case `errno` indicates the error.

## 4.14 munlock: unlock a region of memory

SYNOPSIS

`int` **munlock**(`const void *`*addr*, `size_t` *size*);

DESCRIPTION

This system call implements the POSIX.1b `munlock` function, which unlocks a range of previously locked memory.

Neither the start address of the specified region nor its length need to be page-aligned; this function will lock all pages overlapping the specified region.

In raw DOS mode and VCPI mode, this function works as the POSIX standard indicates; however, since MOSS currently does not support demand paging (only demand loading of executables and demand zeroing of heap memory), unlocking memory currently has no real effect.

In DPMI mode, `munlock` does not work quite as described by POSIX, because DPMI keeps a "lock count" on locked pages and only unlocks a page when the lock count has returned to zero. Thus, if you unlock a page of memory that was previously locked more than once, whereas POSIX specifies that the page should be unlocked on the first call to `munlock`, under DPMI it will only be unlocked after *two* calls (or however many times the page was locked). (MOSS could emulate true POSIX behavior, but is it worth the trouble?)

PARAMETERS

*addr*: The start address of the region to unlock.

*size*: The size of the region to unlock, in bytes.

RETURN VALUE

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

## 4.15 open: open a file

SYNOPSIS

> `int` **open**(`const char *`*path*, `int` *flags*);
>
> `int` **open**(`const char *`*path*, `int` *flags*, `int` *mode*);

DESCRIPTION

This system call implements the POSIX `open` function, which attempts to open a file given its pathname.

PARAMETERS

*path*: The pathname of the file to open. Under MOSS, there are several different "types" of files that can be opened using this function and accessed with any of the file I/O functions:

- If a DOS file of the specified name can be found, it is opened and associated with the returned file descriptor. However, note that the MOSS file descriptor returned does not necessarily correspond to the DOS file descriptor used by MOSS itself to invoke DOS's file operations.
- If no DOS file of the specified name can be found, but an "embedded file" attached to the MOSS executable exists with a name *exactly* matching the name requested, then the embedded file is opened and associated with the file descriptor. The program can then use the embedded file like any other file, including seeking around in the file; however, embedded files cannot be written to or deleted.
- If you specify the "magic" pathname '`/dev/mem`', you will receive a file handle that you can later use to `mmap` parts of physical memory into the program's directly-accessible address space. Currently the other "normal" file operations, such as read, write, and seek, are not implemented for `/dev/mem`.

*flags*: The `flags` argument works basically as under POSIX or BSD; MOSS will emulate the open modes not supported directly by DOS. However, note that due to this emulation, the `open` operation may not be quite as atomic as it is supposed to be under POSIX systems.

For example, if you use the flags '`O_CREAT | O_EXCL`', MOSS must first check to see if the file exists, and then if it doesn't, create it using DOS's "create file" call. In multitasking environments that support DOS, it is possible that the file could be created after MOSS checks to see if it exists but before the call to create the file. In that case, the open will succeed and the existing file will be truncated, which violates the POSIX atomicity rules for `open`. Well, too bad: this is DOS.

Also, the `O_APPEND` flag is only partly supported. If you specify `O_APPEND`, then the file pointer will be positioned at the end of the file initially during `open`, but it is not re-positioned at the end prior to each write, as specified by POSIX. Again, this is generally good enough for typical situations, but it is "not quite right" in its semantics.

*mode*: When you create a file under MOSS, you may specify a normal POSIX permission mask. However, only the user-writable bit (`S_IWUSR`) will be used to determine whether the DOS file will be read/write or read-only; the other bits in the *mode* are ignored.

RETURN VALUE

Returns the opened file descriptor if successful, or -1 on error, in which case `errno` indicates the error.

## 4.16 read: read from a file

SYNOPSIS

`int` **read**(`int` *fd*, `void *`*buf*, `size_t` *nbyte*);

DESCRIPTION

This system call implements the POSIX `read` function, which reads the next *nbyte* bytes of data into *buf* from file descriptor *fd*.

PARAMETERS

*fd*: The file descriptor to read from.

*buf*: The buffer to read data into.

*nbyte*: The total number of bytes to read.

RETURN VALUE

Returns the actual number of bytes read if successful, or -1 on error, in which case `errno` indicates the error.

## 4.17 rename: rename a file

SYNOPSIS

`int` **rename**(`const char *`*old*, `const char *`*new*);

DESCRIPTION

This system call implements the POSIX `rename` function, which changes the name of an existing file.

Since this function is based on the rather limited DOS rename operation, it can't be used to rename a directory, or to move a file from one directory to another. Furthermore, if the file named by *new* already exists, MOSS will emulate proper POSIX behavior by removing the existing file and replacing it with *old*; however, this operation will not be atomic as it should be under POSIX.

PARAMETERS

*old*: The pathname of the file to rename.

*new*: The new name for the file.

RETURN VALUE

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

## 4.18 sbrk: low-level memory allocator

SYNOPSIS

`char *sbrk(int increment);`

DESCRIPTION

This system call implements the BSD `sbrk` function, which allocates more "bulk" memory from the system. This system call is typically only used by the high-level `malloc` memory allocator implemented in the C library.

Under raw DOS mode and VCPI, the behavior of `sbrk` is generally the same as under BSD: the next chunk of memory returned will fall exactly after the previous chunk returned. The *increment*, and hence the break pointer, doesn't have to be page-aligned.

However, under DPMI 0.9 environments, it is generally impossible to emulate this behavior exactly because the DPMI host does not allow MOSS to request that memory be allocated *at a specific address*. This deficiency is fixed in DPMI 1.0, but there are still very few hosts that support DPMI 1.0 (neither Windows 3.1 nor Windows 95 do, for example). Therefore, an application using `sbrk` must be prepared to be given a chunk of memory that does not precisely follow the previously returned chunk.

Furthermore, even under VCPI and raw DOS, the the application calls `mmap()` to map physical memory, MOSS may map the memory into the program's heap address space and bumps the break pointer appropriately; this is another situation in which consecutive calls to `sbrk` might not return consecutive memory regions.

Fortunately, the `malloc` implementation in the BSD C library does not rely on `sbrk` always returning consecutive chunks; it will still work if the break pointer is thrown off for some reason. However, if this happens, small memory chunks returned by `malloc` may no longer be naturally aligned, as the BSD man page for `malloc` says they are. Thus, your application should not depend on this behavior under MOSS. (In general, most `malloc` packages do not align returned chunks, so depending on this alignment behavior is bad practice anyway.)

PARAMETERS

*increment*: The number of bytes of memory to allocate.

RETURN VALUE

Returns a pointer to the allocated memory if successful, or `(char*)-1` on error, in which case `errno` indicates the error.

## 4.19 sigaction: change the action for a signal

SYNOPSIS

`int` **sigaction**(`int` *sig*, `const struct sigaction *`*act*, `struct sigaction *`*old_act*);

DESCRIPTION

This system call implements the POSIX `sigaction` function, which changes the action to be performed when the specified signal is delivered to the program.

MOSS supports both traditional POSIX.1 signal handlers, as well as POSIX.1b handlers for real-time, queued signals, which take additional arguments. Queued signals are especially useful if you want to service multiple hardware interrupts with a single signal handler, because a value identifying the interrupt can be passed, and multiple generated signals will not be "collapsed" into one as in POSIX.1 and traditional Un*x.

Any processor exceptions the program generates will generate appropriate signals, e.g. `SIGFPE` for divide by zero, `SIGTRAP` for debuggin-related exceptions, `SIGILL` for invalid opcode, and `SIGSEGV` for protection violations and other serious exception conditions. The program can catch these signals with a signal handler if desired.

Of course, processor exceptions that occur in MOSS itself or in DOS do not produce application-visible signals; they merely cause the DOS extender to panic.

MOSS currently does not prevent the application from catching or ignoring `SIGKILL` or `SIGSTOP` signals as specified by POSIX. ("Security" is pretty meaningless under DOS anyway.)

PARAMETERS

*sig*: The signal number whose action is to be changed.

*act*: If non-NULL, must point to the new action to give the signal. *act*`->sa_handler` may be either `SIG_DFL`, `SIG_IGN`, or a pointer to a signal handler function. The `SA_SIGINFO` bit can be set in *act*`->sa_flags` to request that the additional parameters specified by POSIX.1b be supplied to the signal handler function.

*old_act*: If non-NULL, the original action associated with the signal is stored in the provided structure before the signal action is changed.

RETURN VALUE

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

## 4.20 sigpending: find the set of currently pending signals

SYNOPSIS

`int` **sigpending**(`sigset_t *`*set*);

DESCRIPTION

This system call implements the POSIX `sigpending` function, which returns the set of signals currently pending.

PARAMETERS

*set*: A pointer to the `sigset_t` in which to store the set of pending signals.

RETURN VALUE

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

## 4.21 sigprocmask: change the signal mask of the current process

SYNOPSIS

`int` **sigprocmask**(`int` *how*, `const sigset_t` **set*, `sigset_t` **old_set*);

DESCRIPTION

This system call implements the POSIX `sigprocmask` function, which changes the current signal mask of the running program.

Under MOSS, this function can be used to prevent signals generated by hardware interrupts from being dispatched to the process until the appropriate signal is unblocked again. However, note that the signal will automatically remain blocked while the interrupt handler is running, so it is usually not necessary to call this function.

You should not try to block any of the signals that can be generated by processor exceptions, such as `SIGILL`, `SIGSEGV`, etc.

PARAMETERS

*how*: Indicates how to change the signal mask:

- `SIG_BLOCK`: add the signals in *set* to the current signal mask.
- `SIG_UNBLOCK`: unblock the signals in *set*.
- `SIG_SETMASK`: Change the current signal mask to *set*.

*set*: If non-NULL, indicates the signals to change.

*old_set*: If non-NULL, the previous signal mask is stored here.

RETURN VALUE

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

## 4.22 unlink: delete a file

SYNOPSIS

`int` **unlink**(`const char *`*path*);

DESCRIPTION

This system call implements the POSIX `unlink` function, which deletes a file.

PARAMETERS

*path*: The name of the file to remove.

RETURN VALUE

Returns 0 if successful, or -1 on error, in which case `errno` indicates the error.

## 4.23 write: write to a file

SYNOPSIS

`int` **write**(`int` *fd*, `const void *`*buf*, `size_t` *nbyte*);

DESCRIPTION

This system call implements the POSIX `write` function, which writes *nbyte* bytes of data from *buf* into file descriptor *fd*.

Note that you cannot write to an embedded file (a file attached to the MOSS executable).

PARAMETERS

*fd*: The file descriptor to write from.

*buf*: The buffer to write data from.

*nbyte*: The total number of bytes to write.

RETURN VALUE

Returns the actual number of bytes written if successful, or -1 on error, in which case `errno` indicates the error.

# 5 Attaching files to `MOSS.EXE`

Most DOS extenders allow you to attach a 32-bit "extended" program to the DOS extender executable itself, making the DOS extender and program appear to be one big DOS executable. MOSS takes this one step further: it allows you to attach not only the program to be run, but also an arbitrary number of additional files, to the DOS extender executable. The program can then open these files, read them, seek though them, etc., as if they were ordinary DOS files. (These embedded files cannot be written to, however.) Thus, if you want to distribute a program along with a set of tightly-associated data files for its use (e.g. graphics or sound files), you can bind those data files into the DOS executable itself so that users will only see one big DOS executable file.

While this feature is implemented and working in MOSS, generating a combined executable is not entirely trivial, and we haven't automated it yet. If you need to use this feature before we've fixed this problem, send a message to us on the MOSS mailing list and we'll either describe how to do it or provide a script or program to do it for you.

# 6 Remote debugging

MOSS supports full source-level debugging of applications running under MOSS, if you can attach your DOS test machine to a machine running a Un*x-like operating system using a null-modem serial cable. (Null-modem cables can be gotten from any decent computer supply store.) MOSS contains a small piece of *stub* code that allows GDB or another compatible debugger to attach to and control the MOSS application's execution over the serial line.

## 6.1 Preparing to debug

If you are using Linux and have a version of GDB that supports ELF, then you can use your native Linux GDB for remote development of DOS programs running under MOSS: there's no need to build a special GDB cross-debugger. This should also be the case for versions of GDB compiled for other ELF systems such as SVR4. For other OS's that do not use ELF as their primary object file format, you may have to build your own cross-GDB debugger. See the GDB documentation for instructions on how to do this.

You'll also need to turn on the `ENABLE_RGDB` option in '`moss/kernel/raw/config.h` and build a version MOSS that supports remote debugging. In addition, if your null modem cable is connected to a serial port other than COM1 on your DOS machine (MOSS is currently hard-coded to use COM1—gak, I know!), you'll need to edit `moss/kernel/raw/remote-gdb.c` and change the `#define`'d constants '`ser_io_base`' and '`ser_irq`' appropriately. (COM2 is 0x2f8 and 4, respectively.) Also, MOSS is currently hard-coded to use 9600,8,n,1 as its serial parameters; you can similarly change that by hacking the source.

Note that the remote GDB support relies on MOSS's ability to dispatch hardware interrupts as POSIX signals: specifically, a serial line interrupt causes a SIGINT to be sent to the program. Since dispatching hardware interrupts as POSIX signals isn't supported under DPMI, neither is remote debugging. (Actually, it might "sort-of" work: you might be able to debug, but just won't be able to interrupt the program from the remote GDB. We haven't tested this though.)

## 6.2 Running applications under the remote debugger

When you try to run an application under the new debugging-enabled version of MOSS, you'll notice that it appears to hang before the program starts running. MOSS hasn't actually hung (hopefully!); it is simply waiting for instructions to come over the serial line from the remote debugger. At this point, run GDB on your debugging machine, specifying your application's executable as the program to debug. Then, instead of `run`ning the program directly (which wouldn't work anyway since it's a MOSS executable and you'd be running it under a completely different environment), type '`target remote /dev/ttyS0`' (or whatever serial port the null modem cable is connected to).

At this point, if everything worked correctly, the GDB prompt should come back with the program position pointing at the first instruction in the application executable. This will typically be the `crt0` code supplied with MOSS. If you want to proceed immediately to the `main()` routine, just type '`g main`'. At any rate, you can now set breakpoints or otherwise debug the program as you would a program running on a local machine.

### 6.3 Features supported by MOSS's remote GDB stub

MOSS supports most of the features of GDB's remote debugging protocol that are applicable to a single-process environment, such as:

- Inspection and modification of register state.
- Single-stepping and breakpoints.
- Memory examination and modification.
- Interception of signals.
- Generating signals from the remote debugger.
- Interruption from the remote debugger. (Just press ^C when the application is running, and it should stop immediately and return control to the debugger.)

The MOSS remote debugging stub currently does not support compression of register state; this would make remote debugging more responsive, because less information would have to be passed across the wire each time the program stops for instructions from the remote debugger.

Also, currently, MOSS will only notify the remote debugger of signals that have `SIG_DFL` action, i.e. signals that would otherwise cause the program to terminate. There is a trivial change to signal.c that you can make to cause *all* signals, including signals the program has installed handlers for, to be sent to the remote debugger (and the debugger can send them back again if appropriate). This behavior is not enabled by default because it slows things down considerably when lots of signals are occurring (e.g. if you're servicing lots of hardware interrupts with signals), and the "feature" usually only gets in the way anyway. However, if you'd like to turn it on, post a message to the MOSS mailing list and we'll tell you how (or maybe just make it a "proper" option...).