Linux API for read access to z/VM Monitor Records

Date : 2004-Nov-26

Author: Gerald Schaefer (geraldsc@de.ibm.com)

Description

This item delivers a new Linux API in the form of a misc char device that is usable from user space and allows read access to the z/VM Monitor Records collected by the **MONITOR* System Service of z/VM.

User Requirements

The z/VM guest on which you want to access this API needs to be configured in order to allow IUCV connections to the **MONITOR* service, i.e. it needs the IUCV **MONITOR* statement in its user entry. If the monitor DCSS to be used is restricted (likely), you also need the NAMESAVE <DCSS NAME> statement. This item will use the IUCV device driver to access the z/VM services, so you need a kernel with IUCV support. You also need z/VM version 4.4 or 5.1.

There are two options for being able to load the monitor DCSS (examples assume that the monitor DCSS begins at 144 MB and ends at 152 MB). You can query the location of the monitor DCSS with the Class E privileged CP command Q NSS MAP (the values BEGPAG and ENDPAG are given in units of 4K pages).

See also "CP Command and Utility Reference" (SC24-6081-00) for more information on the DEF STOR and Q NSS MAP commands, as well as "Saved Segments Planning and Administration" (SC24-6116-00) for more information on DCSSes.

1st option:

You can use the CP command DEF STOR CONFIG to define a "memory hole" in your guest virtual storage around the address range of the DCSS.

Example: DEF STOR CONFIG 0.140M 200M.200M

This defines two blocks of storage, the first is 140MB in size an begins at address 0MB, the second is 200MB in size and begins at address 200MB, resulting in a total storage of 340MB. Note that the first block should always start at 0 and be at least 64MB in size.

2nd option:

Your guest virtual storage has to end below the starting address of the DCSS and you have to specify the "mem=" kernel parameter in your parmfile with a value greater than the ending address of the DCSS.

Example:

```
DEF STOR 140M
```

This defines 140MB storage size for your guest, the parameter "mem=160M" is added to the parmfile.

User Interface

The char device is implemented as a kernel module named "monreader", which can be loaded via the modprobe command, or it can be compiled into the kernel instead. There is one optional module (or kernel) parameter, "mondcss", to specify the name of the monitor DCSS. If the module is compiled into the kernel, the kernel parameter "monreader.mondcss=<DCSS NAME>" can be specified in the parmfile.

The default name for the DCSS is "MONDCSS" if none is specified. In case that there are other users already connected to the **MONITOR* service (e.g. Performance Toolkit), the monitor DCSS is already defined and you have to use the same DCSS. The CP command Q MONITOR (Class E privileged) shows the name of the monitor DCSS, if already defined, and the users connected to the **MONITOR* service. Refer to the "z/VM Performance" book (SC24-6109-00) on how to create a monitor DCSS if your z/VM doesn't have one already, you need Class E privileges to define and save a DCSS.

Example:

```
modprobe monreader mondcss=MYDCSS
```

This loads the module and sets the DCSS name to "MYDCSS".

NOTE:

This API provides no interface to control the **MONITOR* service, e.g. specify which data should be collected. This can be done by the CP command MONITOR (Class E privileged), see "CP Command and Utility Reference".

Device nodes with udev:

After loading the module, a char device will be created along with the device node /<udev directory>/monreader.

Device nodes without udev:

If your distribution does not support udev, a device node will not be created automatically and you have to create it manually after loading the module. Therefore you need to know the major and minor numbers of the device. These numbers can be found in /sys/class/misc/monreader/dev.

Typing cat /sys/class/misc/monreader/dev will give an output of the form <major>:<minor>. The device node can be created via the mknod command, enter mknod <name> c <major> <minor>, where <name> is the name of the device node to be created.

Example:

```
# modprobe monreader
# cat /sys/class/misc/monreader/dev
10:63
# mknod /dev/monreader c 10 63
```

This loads the module with the default monitor DCSS (MONDCSS) and creates a device node.

File operations:

The following file operations are supported: open, release, read, poll. There are two alternative methods for reading: either non-blocking read in conjunction with polling, or blocking read without polling. IOCTLs are not supported.

Read:

Reading from the device provides a 12 Byte monitor control element (MCE), followed by a set of one or more contiguous monitor records (similar to the output of the CMS utility MONWRITE without the 4K control blocks). The MCE contains information on the type of the following record set (sample/event data), the monitor domains contained within it and the start and end address of the record set in the monitor DCSS. The start and end address can be used to determine the size of the record set, the end address is the address of the last byte of data. The start address is needed to handle "end-of-frame" records correctly (domain 1, record 13), i.e. it can be used to determine the record start offset relative to a 4K page (frame) boundary.

See "Appendix A: **MONITOR*" in the "z/VM Performance" document for a description of the monitor control element layout. The layout of the monitor records can be found here (z/VM 5.1): https://www.vm.ibm.com/pubs/mon510/index.html

The layout of the data stream provided by the monreader device is as follows:

```
...
<0 byte read>
<first MCE>              \
<first set of records>    |
...                       |- data set
<last MCE>                |
<last set of records>    /
<0 byte read>
...
```

There may be more than one combination of MCE and corresponding record set within one data set and the end of each data set is indicated by a successful read with a return value of 0 (0 byte read). Any received data must be considered invalid until a complete set was read successfully, including the closing 0 byte read. Therefore you should always read the complete set into a buffer before processing the data.

The maximum size of a data set can be as large as the size of the monitor DCSS, so design the buffer adequately or use dynamic memory allocation. The size of the monitor DCSS will be printed into syslog after loading the module. You can also use the (Class E privileged) CP command Q NSS MAP to list all available segments and information about them.

As with most char devices, error conditions are indicated by returning a negative value for the number of bytes read. In this case, the errno variable indicates the error condition:

EIO:
: reply failed, read data is invalid and the application should discard the data read since the last successful read with 0 size.

EFAULT:
: copy_to_user failed, read data is invalid and the application should discard the data read since the last successful read with 0 size.

EAGAIN:
: occurs on a non-blocking read if there is no data available at the moment. There is no data missing or corrupted, just try again or rather use polling for non-blocking reads.

EOVERFLOW:
: message limit reached, the data read since the last successful read with 0 size is valid but subsequent records may be missing.

In the last case (EOVERFLOW) there may be missing data, in the first two cases (EIO, EFAULT) there will be missing data. It's up to the application if it will continue reading subsequent data or rather exit.

Open:

Only one user is allowed to open the char device. If it is already in use, the open function will fail (return a negative value) and set errno to EBUSY. The open function may also fail if an IUCV connection to the **MONITOR* service cannot be established. In this case errno will be set to EIO and an error message with an IPUSER SEVER code will be printed into syslog. The IPUSER SEVER codes are described in the "z/VM Performance" book, Appendix A.

NOTE:

As soon as the device is opened, incoming messages will be accepted and they will account for the message limit, i.e. opening the device without reading from it will provoke the "message limit reached" error (EOVERFLOW error code) eventually.