COMP3301/COMP7308 2020 Assignment 2 - Process Accounting Pseudo Device Driver

- $Revision: 242 $

1 Process Accounting Pseudo Device Driver

This assignment asks you to write a device driver for the OpenBSD kernel that implements a replacement for the current process accounting facility.

Process accounting is a facility that allows administrators to audit the system resource utilisation of commands run by users.

Process accounting in OpenBSD is currently implemented via the `acct(2)` system call that enables or disables of logging of commands to a file, the `accton(8)` utility for calling the syscall, and the `sa(8)` and `lastcom(8)` utilities for processing said file. The format of the file is described in the `acct(5)` manual page.

A pseudo device driver is an actual driver, but not one that implements support for physical hardware. It provides a virtual, software only, service for user programs to use.

Device special file are entries in a filesystem that refer to a set of functions in the kernel that implement file behaviour such as open, read, write, and close.

Examples of pseudo device drivers that provide device special files are `zero(4)`, `null(4)`, `tun(4)`, and `tap(4)`.

This is an individual assignment. You should feel free to discuss aspects of C programming and the assignment specification with fellow students. You should not actively help (or seek help from) other students with the actual coding of your assignment solution. It is cheating to look at another student's code and it is cheating to allow your code to be seen or shared in printed or electronic form. You should note that all submitted code may be subject to automated checks for plagiarism and collusion. If we detect plagiarism or collusion, formal misconduct proceedings will be initiated against you. If you're having trouble, seek help from a member of the teaching staff. Don't be tempted to copy another student's code. You should read and understand the statements on student misconduct in the course profile and on the school web-site: https://www.itee.uq.edu.au/itee-student-misconduct-including-plagiarism

1.1 Virtual Machine Hard Drive Image Backup

It is recommended that before you begin this assignment, first back up your virtual machine hard drive image file. You can do this buy copying your virtual machine's hard drive image file (.vdmk or .vhi) to another locaiton. If an error occurs and you are unable to reboot your vitural machine, then you can create a new virtual machine, using your backed up virtual machine hard drive image file. To find your virtual machine Hard drive image, using VirtualBox, right click on your virtual machine icon, in the VirtualBox Manager.

1.2 Subversion

It is `highly recommended` that you use subversion to regularly backup your assignment code. Beaware that mishaps with the Virtual Machine kernel do happen and can take a significant amount to recover from. Regular backups with subversion will pervent this.

2 Specifications

This assigment specifies the replacement of the `acct(2)` system call with a pseudo device driver that provides a device special file that produces messages analogous to the entries written to the accounting file.

The `acct(4)` driver will provide a superset of the functionality that is provided by the current system call. The system call only records information about the process when it exits, but the driver will also report information about process forks and execs.

2.1 Code Style

Your code is to be written according to OpenBSD's style guide, as per the `style(9)` man page.

2.2 Compilation

Your code is to be built as part of the kernel on an `amd64` OpenBSD 6.5 or -current system.

The changes to the kernel necessary to configure an `acct(4)` driver so it can be compiled will be supplied as a diff available from Blackboard. The diff can be applied by running the following:

```
dlg@r630 ~$ cd /usr/src/sys
dlg@r630 sys$ patch < /path/to/assignment2-boilerplate.diff
Hmm...  Looks like a unified diff to me...
```

The driver must be implemented in a single file, and placed in `sys/dev/acct.c` next to the `sys/dev/acct.h` provided by the diff described above.

2.3 Messages

The messages that a program reads from the device driver are represented as a set of structs. The kernel driver populates the structs when it is open and the relevant events occur in the kernel, and then makes them available for a program to read.

The structure of the messages the driver should produce is provided in `sys/dev/acct.h`.

2.3.1 Common Fields

All messages from the driver start with a common set of fields that are contained in `struct acct_common`. The other messages all contain `struct acct_common` as their first field.

The first three fields of the common structure refer to the message itself, rather than the process the message is about. The `ac_type` field contains a number representing the type of the current message, eg, a value of 0 or `ACCT_MSG_FORK` states that the message is about a process forking and should be interpreted as the associated message structure.

The `ac_len` field contains the number of bytes used for this message, including the `ac_type` and `ac_len` fields.

`ac_seq` is a simple wrapping counter that increments by one bit shift for every message that the driver generates. E.g. sequence number 3 is 0x04. If the driver receives notification from the rest of the kernel that an event has occurred (eg, `acct_fork()` is called when a process forks), but is unable to generate a message about it, the sequence number should still be incremented by one bit shift, so that the userland consumer of the messages will know that an event has been lost. The counter should be reset to 0 when the `acct(4)` device is opened.

The remaining common fields should be set for the process the message is about.

2.3.2 exit message

The exit message corresponds with `struct acct_exit`. The information in this message corresponds with the information described in `acct(5)`. `acct(2)` may be used as a reference when filling in the information in this message.

2.3.3 fork event

The fork message corresponds with `struct acct_fork`, and is generated when a process forks a new child. The information in the message should be based on the parent of the new process, apart from `ac_cpid` which contains the process ID of the new child. Note that `acct_fork` is given a pointer to the child, not the parent.

2.3.4 exec event

The exec message corresponds with `struct acct_exec`, and is geneated when a process calls `exec()`. It exists to record the new name of the binary the program is executing.

2.4 Driver entry points

`acct.c` must provide the following functions to support the integration into the kernel, and to provide the required interface for userland to access the driver.

2.4.1 Kernel entry points

The kernel is patched to call 3 functions when a process forks, execs, or exits. Those functions are `acct_fork()`, `acct_exec()`, and `acct_exit()` respectively. All these functions take a `struct process *` as their only argument, and do not return anything to the caller.

2.4.2 Userland entry points

`acctattach()` is called when the kernel starts running for the driver to configure any state needed for it to operate.

`acctopen()` is called when a program attempts to open a device file with the corresponding major number to this driver. It should allow only the 0th minor to be opened, opened exclusively, and only opened for reading. Read/write or write only opens of the device should fail with `EPERM`. The sequence number for generated messages should be reset to 0 on every open.

`acctclose()` should clean up any state associated with the open driver.

`acctioctl()` should support `FIONREAD`, and `FIONBIO` as per the `ioctl(2)` manpage. `FIONASYNC` should not be implemented.

`acctread()` dequeues a a single message, and copies as much of that one message as possible to userland.

`acctwrite()` should return `EOPNOTSUPP` as the driver does not support being written to by a userland process.

Only blocking I/O must be supported. Non blocking I/O functions must not be implemented.

3 Submission

You are required to implement the `acct(4)` driver by writing your code in a single file, `sys/dev/acct.c`. This file in the OpenBSD source tree will be committed to your SVN repo as `a2/acct.c`.

Submission must be made electronically by committing to your Subversion repository on `source.eait.uq.edu.au`. In order to mark your assignment the markers will check out `a2/acct.c` from your repository. Code checked in to any other part of your repository will not be marked.

As per the `source.eait.uq.edu.au` usage guidelines, you should only commit source code and Makefiles.

The due date for the code submission is the 6th of October, 2020, at 4pm. Once submitted, no further changes to the code is permitted.

3.1 Demo on Zoom

You are required to demo your assignment 2 via a Zoom desktop screen sharing session, during your assigned Zoom Practical Session in week 9. If you do not demo via the Zoom session, then your assignment 2 will not be marked. You are only permitted to demo the code that was submitted by the due date.

3.2 Recommendations

The following kernel functionality may or may not be useful in the implementation of this assigment:

- `malloc(9)` - kernel memory allocator
- `pool(9)` - resource pool manager
- `TAILQ_INIT(3)` - doubly-linked list macros
- `KASSERT(9)` - kernel assert routines
- `uiomove(9)` - move (copy) data described by a `struct uio`
- `mutex(9)` - kernel mutex implementation
- `rwlock(9)` - kernel read/write lock implementation
- `tsleep(9), and wakeup(9)` - process context sleep and wakeup

The majority of the OpenBSD kernel still runs under a Big Giant lock, known as the kernel lock, which can be used to provide implicit serialisation of code in this driver. The kernel lock is taken and released with `KERNEL_LOCK()` and `KERNEL_UNLOCK()` respectively, or if it is assumed to be held, may be asserted with `KERNEL_ASSERT_LOCKED()`.

4 Testing

A tool will be provided that reads from the special file and parses the messages as per `sys/dev/acct.h`.