Intro

In order for users and applications to take advantage of system state information, there must be some way for the kernel to be able to provide it. The Linux kernel uses something called the process file system, also referred to as procfs. Procfs puts information about running processes and the system status on the filesystem itself. This allows for a standardized method for reading system information without the need to write new system calls or interact directly with the kernel. JOS can be extended to have this functionality as well, making it much more powerful by extending the capabilities of users and user applications.

Background

In 1984, Tom Killian presented the paper, “Processes as Files”. This paper described his implementation of a process file system for V8 (Unix 8th edition). As discussed in the paper, this file system was initially implemented with the intent of making debugging simpler with the first program making substantial use of it being an interactive debugger called pi developed by T. A. Cargill. A later iteration of a process file system included in Plan 9, a distributed operating system, improved on V8’s implementation. Rather than having a single file per process, each process was represented by a directory of files that could be manipulated via system I/O calls. The implementation of procfs used by Plan 9 has been mimicked by Linux with a few additions of particular information files. One useful addition was that the PID required to find the particular process directory can be obtained through one of several terminal commands including pgrep, pidof, and ps.

Problem

Both applications and users commonly need access to information about other processes and the state of the system (such as memory or CPU use and system uptime). Since the kernel manages this information and isolates processes to prevent exposing data between one another, this information must be read directly from the kernel. There are a couple of strategies used to accomplish this, the most obvious being system calls. The problem with implementing system calls, however, is that they only add clutter to the existing system call space and don’t allow the user to directly use them from a command-line interface. The implementation of a process file system is a common solution among Unix-like operating systems and will be the solution we implement to address the issue of limited access to information about other processes.

Approach

We will start by to adding hooks to the file system path resolution done in the API we will be using to later implement our file system driver. This will involve creating a graph of nodes representing processes (and other information we want to expose) that implement callbacks for each file system API call (i.e. directory listing, reading, writing). The root node would then override the folder for `/proc`. For example, `cat /proc/120/path` would call the directory listing callback on the virtual “proc” graph node, which would return that it includes a directory “120”. The normal file system path resolver would follow this chain until it hits the file node “path” as it normally would with a file on disk. The read callback on the “path” node would then return the file system path of process id 120, as plaintext.

This would allow us to then model the regular file system as a single graph node that returns itself (with a different spoofed name) when returning directory listings, which would then delegate to the normal disk driver to actually figure out things such as what files can be listed and what data can be read or written. We can maintain a full graph listing that matches the disk (in order to share the file system code between the disk and virtual file system), and keep it up to date with the disk by hooking into the syscalls for manipulating the file system. However, this would be rather memory inefficient when the nodes for actual files/directories can be easily shared, since they should all contain identical callbacks.

The process filesystem node graph will be minimized in memory by generating nodes as they are used and by destroying all created nodes as soon as the syscall ends. The actual node creation could easily be included in the callbacks for the upper level nodes. A similar approach may be taken for the node graph that matches the filesystem, but this could place unneeded overhead on the normal file system driver, which can be expected to be hit regularly. To limit this overhead, we will cache the inode directory structure of the file system.

Potential Problems & Issues

In order for the proc file system to have any real value to users, the kernel has to make sure that it is both accurate and up to date. It could be difficult to have the kernel build and maintain procfs without draining too much of the machine’s valuable resources. The way in which procfs is integrated with the rest of the file system is an important consideration as well, since if done poorly, it is yet another way in which kernel performance could be negatively impacted. The information in procfs also has to be organized in such a way that users can easily locate and access the system information that they require. There are many suitable ways of organizing this information, but it is undoubtedly a challenge to determine which scheme will serve the majority of users best. Also, it is extremely important to make an informed decision concerning what system information will be exposed to the user. If too much information is exposed, this poses a security risk since applications may be able to read an application’s data or perform another kind of attack on running software. If too little information is exposed, then the system becomes useless to both applications and the user.