Development of a Deadlock-Detecting Resource Locking Algorithm for a Kernel Debugging User-space API Library

Timothy Wismer 2004-2005
TJHSST Computer Systems Lab

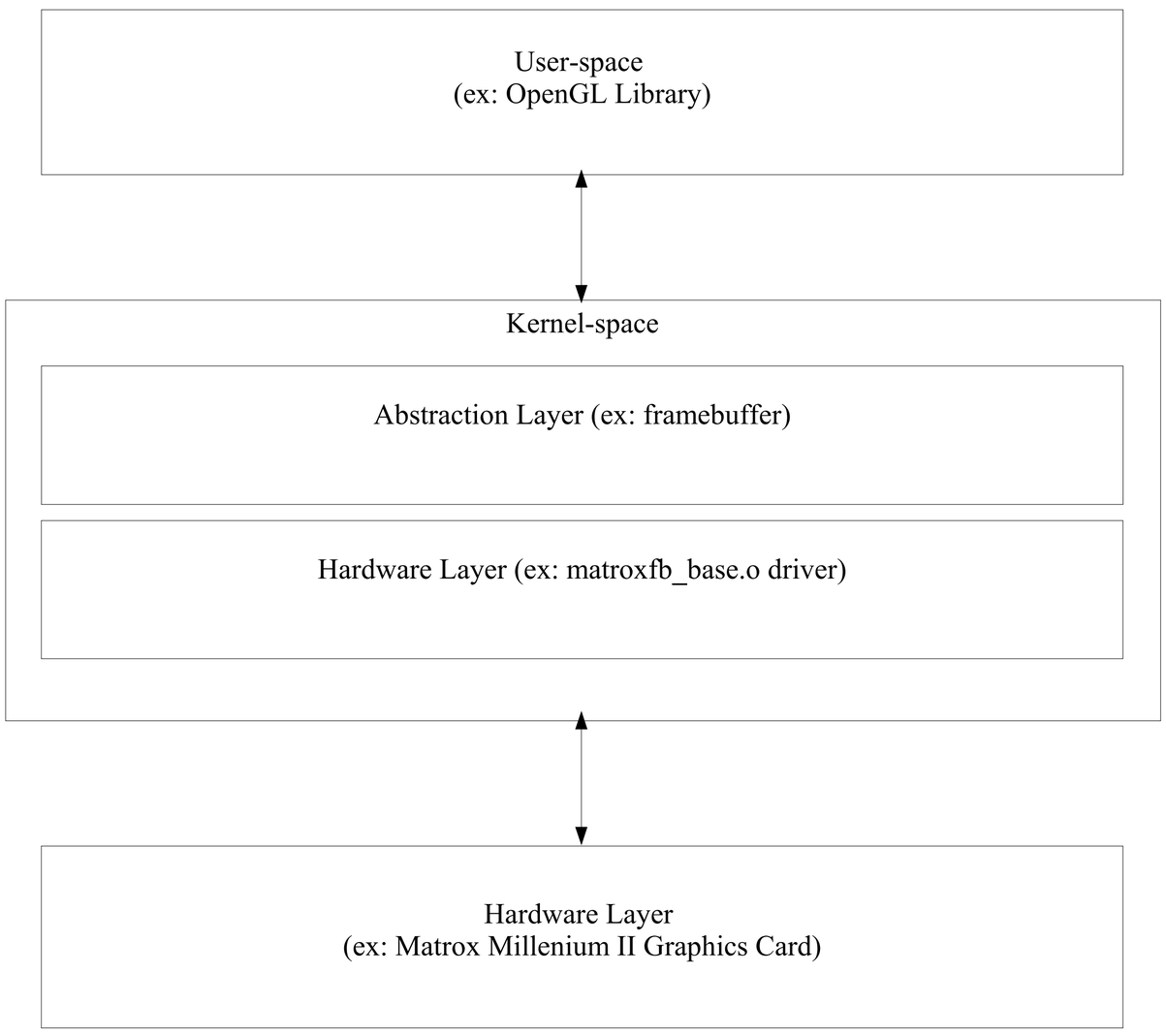
Abstract

The kernel is the heart of an operating system; it is the program that is run when the computer first boots up, and it is responsible for accessing the computer hardware and performing other management tasks on behalf of all other programs. Because of its importance, the Linux Kernel must perform perfectly; any vulnerability, instability, or inefficiency will slow down or threaten the entire system.

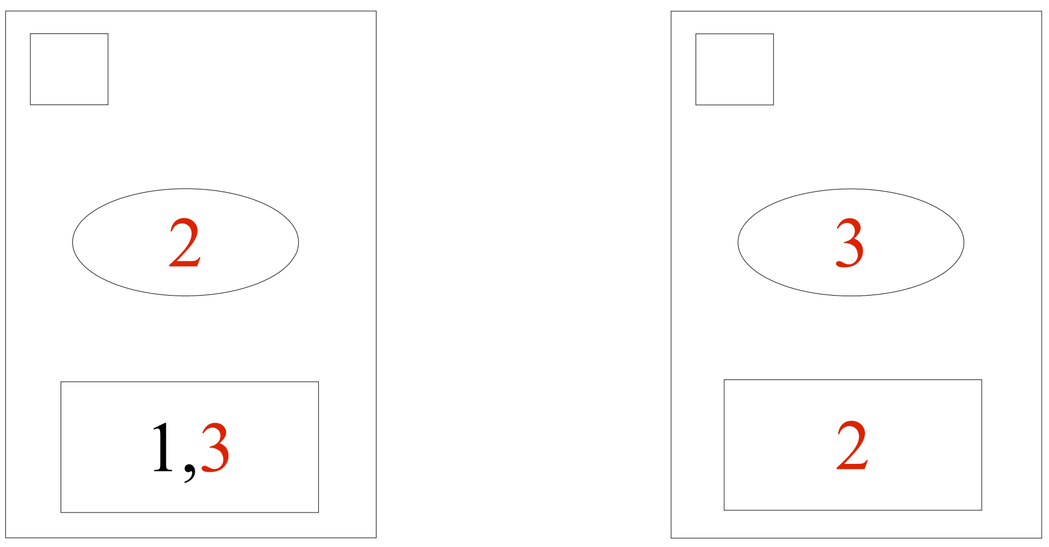
Unfortunately, due to the nature of the program, kernel code cannot be easily debugged. The kernel runs in an environment called *kernel-space*, which is significantly different from the *user-space* environment in which ordinary programs run. The kernel provides user-space as an abstraction to the running programs, masking process scheduling, disk I/O, and so forth. Because the kernel expects to run in kernel-space, it cannot run in userspace. In addition, if there is an error while the kernel is running, it is difficult for the code to provide the tester with useful data about the error (because the kernel itself is responsible for access to the hard drives and monitor). The goal of the KDUAL project is to create a C library which implements the kernel Application Programming Interface (API) *in user-space* and performs automatic debugging. Sections of kernel code can then be compiled against this library and run as ordinary programs for convenient testing.

This particular section of the project aims to implement the kernel's resource locking API, providing the core resource-locking algorithms with debugging code which will provide automatic detection, reporting, and resolution of deadlock situations, thus catching subtle locking errors in early testing and production and easing later debugging. Locking will be implemented in two parts--the core algorithms, with their own API designed to be most convenient for use in development, and simple wrapper code bridging that API to the kernel API. The core algorithms will also be suitable for applications other than kernel programming, e.g. as a generic lock-testing toolkit.

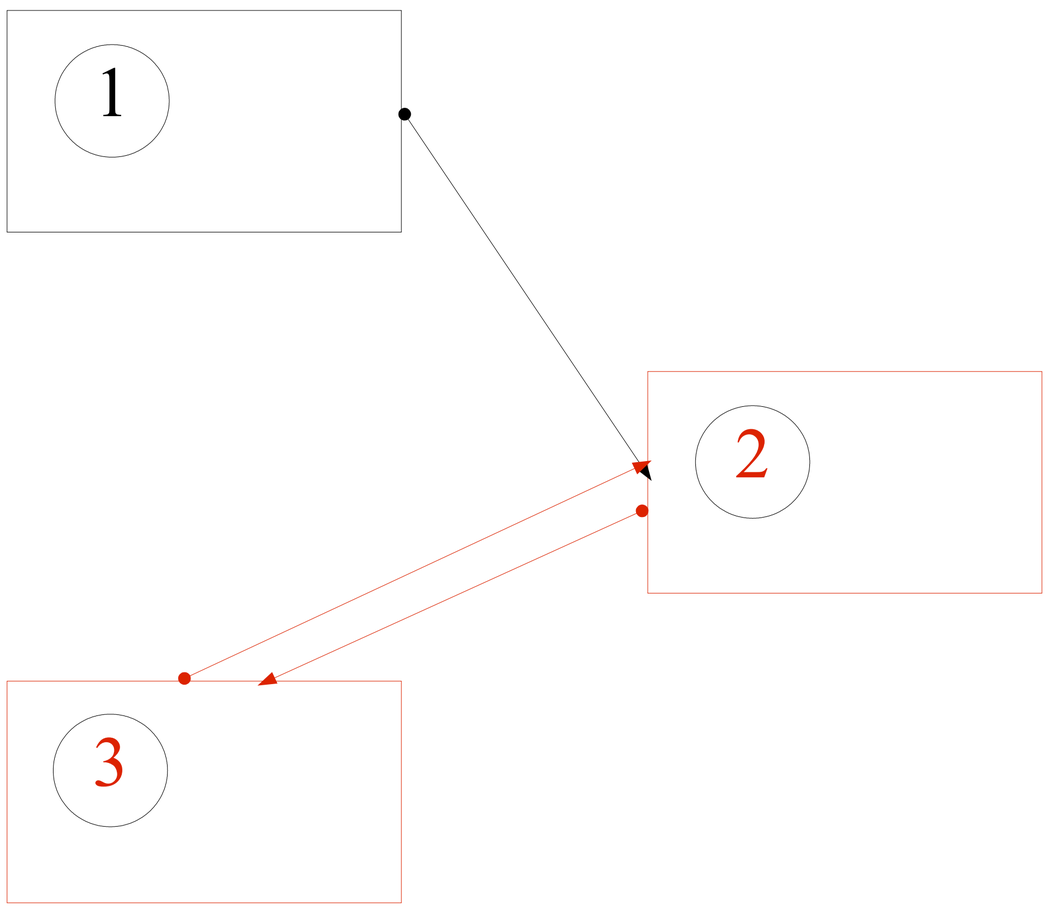
Separate Program Layers: Physical layer at the bottom, low-level hardware interaction and convenient abstractions provided by the kernel, and user code at the top. The middle layers will be implemented by KDUAL.



A deadly embrace: process 2 wants the lock process 3 holds while process 3 wants the lock process 2 holds. Process 1 wants process 2's lock, but is not part of the deadlock.



The Wait-For Graph representing the lock situation above. The deadlock appears as a cycle: a complete path connecting nodes 2 and 3. This will be identified by the Deadlock Detector, which can eliminate the deadlock. Because node 1 is not in the cycle, it will not be affected by the Deadlock Detector.



Locking and Deadlock

Resource-locking is essential for successful *concurrency*: having multiple separate programs running simultaneously and working with the same resources---shared data objects, I/O handles (e.g. sockets), physical devices (reading data from a floppy disk), or any other resource which the processes cannot all use at once. In the absence of locking, such programs would simply "stomp" on each other, accessing the data simultaneously and potentially resulting in data corruption and program failure. Consider a simple example: a number *i* shared by two programs, A and B. At some time during execution, *i* is 0, and each program wants to access it: A wants to set it to 3, and B wants to read its value for later use. If both processes access *i* without coordination, A will succeed in setting the value, but the value B returns is unpredictable. It may be 0 (if the read is completed before the write), 3 (if the write is completed before the read), or some other garbage value (if the read is performed while the write is occuring). To prevent an error situation, a process cannot safely operate on the data until it ensures that no other process is operating on the data. To achieve this, each resource is associated with a "lock" object. Generically, the lock has two states: locked and unlocked. To use a resource, a process must first obtain the associated lock. If the lock is currently in the unlocked state, the process can take the lock and then proceed to manipulate the data as it pleases, returning the lock to the unlocked state when it is done. If the lock is taken (because some other process is using the resource), the current process must wait on the lock until it is returned to the unlocked state (i.e. the resource is no longer in use). The process can then take the lock as before and proceed to use the resource.

Deadlock occurs anytime a process cannot obtain the lock it is waiting for without giving up a lock it already holds. For example, the most trivial case of deadlock occurs when a process attempts to take a lock it already holds; each time the process checks the lock, it is in the locked state, so the process will keep waiting. However, it will never unlock the lock (since it is busy trying to lock it) and so will wait forever. This locking implementation will therefore include built-in testing for deadlock situations, vastly simplifying the debugging process and helping to produce more reliable code. It will also provide other valuable debugging features, e.g. status messages printed by certain segments of the locking code, which will help developers isolate and correct problems.

The detection thread relies on a structure called a Wait-For-Graph (WFG), a common concept in deadlock detection. The *nodes* of the WFG represent processes in the system, and the (directed) *edges* of the graph are dependencies between processes, where an edge $i \rightarrow j$ indicates that process *i* is waiting for a resource currently held by *j*. If there is a deadlock, it will be detectable in the WFG as a *cycle*: a complete path from any node back to itself. The definition of deadlock is that the process is waiting on a resource that it cannot obtain without releasing a resource it already holds; this will necessarly result in a cycle in the WFG, because the process must (through some number of intermediaries) depend on itself. Thus there must be some series of edges leaving from a node and pointing back to the node. The reverse is also true: a cycle in the WFG always indicates a deadlock situation: the cycle indidcates that the process ultimately depends on itself, meaning that it is necessarily involved in deadlock and requires outside intervention. This means that checking for cycles in a WFG will detect all deadlocks *without* identifying non-deadlock situations as deadlocks (false positives); thus it is a reliable method for detection.