Lab: Fork Watcher

Peter Chapin
Vermont State University
CIS-4020, Operating Systems

Purpose

- In this lab you will gather information about each time `fork` is called.
  - The information will be stored in a fixed size circular buffer.
    - Old data will be overwritten as new data is added.
    - Only information on most recent forks preserved.
  - Each record in the buffer is a structure containing interesting fields.
    - What constitutes "interesting" is described later.
  - Module presents data in a `/proc` file.
  - User mode application formats `/proc` data.

Reading

- Resources to review...
  - The `clone` manual page.
    - Read the whole thing.
  - Chapter 3 in Linux Kernel Development by Robert Love.
  - The kernel source.
    - Especially `kernel_clone` and `copy_process`
    - Also study the definition of structure `task_struct`

`current`

- The macro `current` evalutes to a pointer to the `task_struct` of the current thread.
  - Use `current->some_member` to access members of the current thread's `task_struct`.
  - The `copy_process` function allocates and initializes a new `task_struct`.
  - However, what you need to do in this lab is mostly collect information about the process doing the fork.
    - Can gather that in `kernel_clone` before anything else happens.
    - HOWEVER... must also note success/failure... gathered at the end of `kernel_clone`.

What Information?

- What information do we want to collect?
  - Clone flags
  - Parent user ID
  - Parent and child process ID
    - Consider taking into account namespace information as well (not required)
  - Name of the command from which the process was created
    - Must use a helper function to access this properly
  - Return value of the clone operation

Circular Buffer

- You need to do the following...
  - Define a structure (`struct fork_info`) to hold the necessary information.
  - Define an array of such structures at global scope in the same file where `kernel_clone` is defined.
  - Define two global pointers (or index variables):
    - One always points at the next available slot in the buffer (`next_in`)
    - One always points at the next item to remove from buffer (`next_out`)
    - When `next_in == next_out` the buffer is considered empty

put_fork_record

- Signature
  - `void put_fork_record( const struct fork_info *fi );`
- Pseudo-Code
  - Put record into next available slot
  - Advance `next_in` with wrap, if necessary
  - If `next_in` now equals `next_out`, advance `next_out` with wrap, if necessary
    - This policy causes old data to be lost. This is acceptable in this case.
    - Does this work for an empty buffer? How about a full buffer?

get_fork_record

- Signature
  - `int get_fork_record( struct fork_info *fi );`
- Pseudo-code
  - If `next_in == next_out` the buffer is empty
    - Return an error code
  - Otherwise, copy item at `*next_out` and advance `next_out` forward with wrap, if necessary
  - Return the previously copied item
    - Does this work for an empty buffer? How about a full buffer?

Program Structure

- The definition of `struct fork_info` goes in a header file (`fork_info.h`)
  - This allows it to be `#include`d in both the kernel source and the module source
  - Put `fork_info.h` with other kernel headers
- Declaration of `get_fork_record` in the header file
  - So the module code can call it
  - Don't forget to `EXPORT_SYMBOL` that function after it is defined

`/proc` Handling

- Similar to the Counting System Calls lab but trickier. Consider...
  - What happens if new records are added while you are formatting records for display?
    - Do you miss any records?
    - Do you try to display any records twice?
    - Do you display corrupt records?
  - We can deal with some issues using locking techniques, but not all.
    - What happens if the `/proc` reading process takes a "long time" to completely read the `/proc` file?
    - Consider if someone uses `less` to view the first few records and then walks away from the system for an hour.

Formatting Requirements

- When outputting the fork records to the `/proc` file...
  - Okay to output raw numbers without headers in a space-delmited or comma-delimited format
- We can write a user mode application that reads the /proc file and formats it better:
  - Convert clone flags to symbolic names.
  - Convert UID values to real user names.
    - See `getpwuid` in the C library.
  - Convert return values to real error names.
    - `include/asm-generic/errno-base.h`
    - `include/asm-generic/errno.h`

Locking Issues

- Producer consumer problem?
  - Threads that are forking are producers creating `fork_info`s
    - Multiple producers possible (simultaneous forks)
  - Threads reading the `/proc` file are consumers of `fork_info`s.
    - Multiple consumers possible (simultaneous `/proc` file reads).
- Not quite the same...
    - Buffer never "overflows" (old data is overwritten)
    - No need for consumer to sleep on empty buffer (just return end-of-file indication)
  - This simplifies things

Lock Hiding

- Our design is good...
  - We can hide all locking in
    - `put_fork_record`. All producers use this function to install new items in the buffer.
    - `get_fork_record`. All consumers use this function to get items from the buffer.
  - Module authors do not need to worry about getting locks right.
    - This is good; proper locking can be tricky.
    - Question: Will `get_fork_record` ever sleep? Module authors will want to know.

Producer Locking

- Actually simple...
  - Lock a mutex before touching the buffer to avoid corruption
  - No need to reserve a free slot
    - There is always space (in effect)
  - Just need to be sure the nobody else is touching the buffer at the same time. For example, we want to avoid:
    - Thread A: Install new record at `*next_in`
    - Thread B: Install new record at `*next_in` (overwrites Thread A's value!)
    - Thread A: Advance `next_in`
    - Thread B: Advance `next_in` (leaves an "empty" slot behind!)
    - Oops!!

Consumer Locking

- Also simple...
  - Lock mutex before touching the buffer to avoid corruption
  - No need to reserve a used slot
    - If it turns out the buffer is empty, just return error code
  - What kind of problems can occur if there is no locking?

What Kind of Lock?

- We have several to choose from...
  - Spin locks
    - Can only use if critical section is "short" and never sleeps
  - Semaphores, mutexes
    - Required if critical section sleeps. More overhead.
  - Reader/writer locks
    - Appropriate if many threads treat the data as read-only
- And the winner is...
  - Spin locks!
    - Why?