Lab: Fork Watcher

CIS-4020, Operating Systems
Peter C. Chapin

Purpose

- In this lab you will gather information about each `fork` system call that is executed.
  - Information stored in a fixed size circular buffer.
    - Old data 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 the book.
  - The kernel source.
    - Especially `do_fork` and `copy_process`

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 Lab #3 is mostly collect information about the process doing the fork.
    - Can gather that in `do_fork` before anything else happens.
    - HOWEVER... must also note success/failure... gathered at the end of `do_fork`.

What Information?

- What information do we want to collect?
  - Clone flags
  - Parent and child process ID.
    - Consider taking into account namespace information as well (not required in the first version).
  - Parent user ID.
  - Name of the command from which the process was created.
    - Must use a helper function to access this properly.
- Most of this information is available in the calling process `task_struct`

Circular Buffer

- You need to do the following...
  - Define a structure to hold the necessary information.
  - Define an array of such structures.
  - Define two pointers (or index variables):
    - One to point at next available slot in the buffer.
    - One to point at next item to remove from buffer.

put_fork_record

- Pseudo-code
  - Put record into next available slot.
  - Bump `next_in` forward (with wrap)
  - If `next_in` equals `next_out`, bump `next_out` forward (with wrap)
    - 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`

- Pseudo-code
  - If `next_in == next_out` the buffer is empty.
    - Return an error code
  - Copy item at `*next_out` and bump `next_out` forward (with wrap).
    - Does this work for an empty buffer? How about a full buffer?

`/proc` Handling

- Similar to Lab #2 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 (to be discussed later), 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.
- 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.
    - Multiple producers possible (simultaneous forks)
  - Threads reading the `/proc` file are consumers.
    - 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.
  - No need to reserve a free slot.
    - There is always space.
  - Just need to be sure the nobody else is touching the buffer at the same time.
    - Thread A: Install new record at `*next_in`
    - Thread B: Install new record at `*next_in`
    - Thread A: Advance `next_in`
    - Thread B: Advance `next_in`
      - Oops!!

Consumer Locking

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

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?