CSE 5433
Lab 4 Documentation
Team #7 (Colin Drake and Ananth Mahadevan)
3/21/2013

Basic Checkpoint System-call Design

Our basic design first requires a quick, small change in the kernel task structure. We added a counter variable, checkpoints, to task_struct which is initialized to 0. This value is incremented during every checkpoint (i.e. every call to the cp_range() system call), and thus can be interpreted as the number of checkpoints run on a given process. In addition, we define a simple function, page_dump(addr), which dumps a given page to a file specified by the PID and number of checkpoints taken already. It does so by looping from addr to addr+PAGE_SIZE (assuming addr is the beginning of a page of memory), and writing the value of each memory address in between to a file located at /tmp/<pid>-cp<n>, where pid = current->pid and n = current->checkpoints. This I/O is supported by the sys_open(), sys_write(), and sys_close() functions.

The actual implementation of the system call is relatively straightforward. To begin with, the vm_area_structs containing *start and *end are found by using vma_find(). These values are clamped from 0GB-3GB to ensure that the checkpoint will only include user code. From here, we loop through the vm_next pointer across all of the found VMAs. For each VMA region, we loop from vm_start to vm_end by increments of PAGE_SIZE if and only if the VMA is not VM_IO or VM_LOCKED and has VM_READ enabled. For each of these addresses, page_dump() is called if and only if the location conditions and page status flags are satisfied. The location conditions are an AND expression that determines if the given page start and end addresses contain the *start and *end addresses. The page status check is implemented in the page_valid() function, which ensures that the page is not locked, NULL, errored, or ZERO_PAGE. This ensures that valid pages with the beginning or tail end of the memory passed in are dumped as well as valid pages that contain all of the data within, covering all of our needed cases. Again, if this condition is true, the page_dump() function then appends the data stored in that page along with the page start address to a file in /tmp. Upon calling the system call cp_range() again, current->checkpoints will be incremented so output will go to the next file.

Incremental Checkpoint System-call Design

Our incremental design is built on top of our implementation of the basic full checkpointing system. Thus, this includes changes made to task_struct, the cp_range(void *start, void *end) function, and the page_dump(addr) function. It will be stored in the same file and implemented in the inc_cp_range(void *start, void *end) function.

The incremental checkpoint process may be split into two cases: the initial checkpoint and the subsequent checkpoints made afterwards. In the case of the initial checkpoint for a process, current->checkpoints is read. If the value is 0, a full checkpoint is needed and is done by simply calling the cp_range(void *start, void *end) function from the original implementation in Part 1 of the lab and then immediately returning. This will write a file to /tmp/<pid>-cp<n> as before. This is due to the fact that the initial checkpoint must always be a full checkpoint.

For each subsequent checkpoint, additional logic is needed to determine which pages to save. To implement the incremental behavior, we check the dirty bit for each page before deciding whether or not to save it. This bit shows whether or not the page has been modified since being paged in. The file include/linux/page-flags.h contains information regarding the flags variable within each page structure. PG_dirty is defined as the bit position that holds the dirty bit and the macro pte_dirty(dirty) tests it by using test_bit(bit, value). Thus, our incremental checkpoint system call can determine if a page has been modified by using this macro. To implement the incremental case, the system call must loop through each VMA from the *start to *end address, just as in the original case. For each vm_area_struct found, the code must also loop through all addresses from vma_start to vma_end by increments of PAGE_SIZE, storing the page start in an unsigned long address variable. This is equivalent to looping over each page in the VMA. From here, the follow_page() function from mm/memory.c may be called, yielding a pointer to a page structure given the address. Finally, with a pointer to the page, pte_dirty(page) may be called. If the result of this macro is true (non-zero valued), the page has been modified since last page-in and must be saved. From here, the page_dump(addr) function (taken from our original implementation) may be called. This opens the file /tmp/<pid>-cp<n> and writes out the contents of all memory addresses from addr to addr+PAGE_SIZE. Thus, only the modified pages will be saved to the output file.

Special Cases

Following were the checks we added for the pages and the VMAs:
PG_LOCKED, PG_ERROR, PG_RECLAIM, PG_MAPPEDTODISK, PG_SWAPCACHE, PG_COMPOUND, PG_NOSAVE, PG_WRITEBACK, PG_PRIVATE, PG_CHECKED, PG_SLAB, PG_ACTIVE, PG_LRU, PG_UPTODATE, PG_REFERENCED, PG_HIGHMEM and PG_RESERVED.

Flags checked for VMA -- VM_LOCKED, VM_IO, VM_READ, VM_DONTCOPY, VM_RESERVED.

As an optimization step, we also check if a pg is a “zero_page” before taking a backup of it.

Bugs/Limitations

While taking the full system dump (from address 0 to 3GB), we encountered this error - “Unable to handle kernel paging request at virtual address 003a6000”. We tried debugging it by printing the VMA values, adding all possible checks for the pages and the VMA areas, but we still could not figure it out. The program takes the backup up until the address 0x003a6000, but stops when it encounters this address.