Lab 4 grading

Testcases

Please read the remarks (the grading result) in conjunction with the grading scheme at the end of this document, and the testcases in the `tc/` folder in `lab4-grader.tar.gz`.

This is the syntax of the testcases:

- size: a decimal (no prefix) or hex (prefixed by `0x`) number; can be multiplied by page size (suffixed by `pg`)
- aname: Allocation name; a name used to refer to an allocation in the testcase script
- fdname: File descriptor name; a name used to refer to a file descriptor in the script
- alloc aname size: allocate memory of size `size` and assign it to the name `aname` (call `userswap_alloc(size)`)
- map aname fdname size: map the file in `fdname` up to size `size` and assign the mapping to the name `aname` (call `userswap_map(fd, size)`)
- read8 aname offset: read the 64-bit value (unsigned long) at offset `offset` into allocation `aname`
- read1 aname offset: read the byte at offset `offset` into allocation `aname`
- write8 aname offset value: write the 64-bit value `value` (unsigned long) at offset `offset` into allocation `aname`
- write1 aname offset value: write the byte `value` at offset `offset` into allocation `aname`
- free aname: free the allocation `aname` (call `userswap_free(address)`)
- lorm size: set the LORM to `size` bytes
- mkfile fdname size: create a temporary file of size `size`, fill it with random data, and open it in the runner; then assign the FD to the name `fdname`
- close fdname: close the FD `fdname`
- mappingassert on|off: enable/disable the checking of the existence of allocations and their permissions (on by default)
- readassert on|off: enable/disable the verification of read results (on by default)
- writeassert on|off: enable/disable the verification of writes (i.e. whether unexpected writes to files results in an error) (off by default)
- checkmapping aname: check the existence (but not the permissions) of the given allocation
- randread aname count: do `count` reads from random locations in the allocation
- randwrite aname count: do `count` writes to random locations in the allocation
- randrw aname count: do `count` reads/writes (in total, randomly decided) from/to random locations in the allocation

- shufread aname startpage count: do one read each to a random location in each page starting from page index `startpage` and going for `count` pages, in a random order
- shufwrite aname startpage count: do one write each to a random location in each page starting from page index `startpage` and going for `count` pages, in a random order
- shufrw aname startpage count: do one read or write (randomly decided) each to a random location in each page starting from page index `startpage` and going for `count` pages, in a random order
- seqread aname startpage count: do one read each to a random location in each page starting from page index `startpage` and going for `count` pages, in sequential order
- seqwrite aname startpage count: do one write each to a random location in each page starting from page index `startpage` and going for `count` pages, in sequential order
- seqrw aname startpage count: do one read or write (randomly decided) each to a random location in each page starting from page index `startpage` and going for `count` pages, in sequential order
- randinit aname: fill the allocation with random bytes
- checkfile fdname: verify the size and contents of the file
- checkhandler: check the presence of the SIGSEGV handler
- expectcrash: expect the testee to crash
- nomadvise: do not check that MADV_DONTNEED has been called on eviction (used primarily for Ex1-2 testcases)
- checkswapfile: verify that the swapfile is not larger than allowed
- randompageoffset: make {shuf,seq}{read,write,rw} use random page offsets (instead of always 0)

Explanation of certain errors

> Requirement(s) Xy not met

Refer to the grading scheme for an explanation of requirements.

> error: exited with signal SIGSYS (Bad system call)

This means that you used `mmap` without specifying `MAP_ANONYMOUS` in the flags.

> error: at addr 0 in alloc a2, expected page permissions X but got Y

At the end of each read/write, the grader checks the mappings in the kernel (using /proc/pid/maps), and expects the following:

- freed page: unmapped
- nonresident page: none
- clean page: readonly
- dirty page: readwrite

error: kernel page at addr 0 in alloc a1 not freed after page evicted; missing MADV_DONTNEED?

This means that the page was evicted, but it was still allocated a physical page in the kernel, likely because your program did not apply MADV_DONTNEED to the page.

error: expected crash but testee did not crash

This means that in testcase 1e, your program did not remove the SIGSEGV handler on a SIGSEGV to an uncontrolled region.

error: testee stopped receiving commands

This means that your program disconnected from the grader before the testcase was complete. Most likely, your program crashed or exited, but we did not see the process exit (due to `wait(WNOHANG)`).

error: testee exited prematurely with exit code 1

The same as above, but we saw the process exit.

error: unexpected write(s) seen

This means your program made write(s) to files or the swapfile when unnecessary (e.g. during an eviction of a clean page).

error: userswap_alloc(1024) returned non page-aligned address 7ffd0ee47f08, unusable

This means that your `userswap_alloc` or `map` returned a non-page-aligned address (last three nybbles nonzero). `mmap` will never return such an address. Either you accidentally returned a pointer to the address, or you might have forgotten to initialise some local variable.

error: expected ... when reading u64 a1+..., got ...

This means that the wrong value was seen when reading from an allocation. Most likely your program did not restore the page's contents correctly.

Using the grader

To use the grader:

1. Place your solution C file in the `s/m/` directory.
2. Run `./run_pretty.sh`.

Although the grader binary is provided (`grader-bin`), the source (in Rust) is available in the `grader/` directory. You will need the Rust toolchain to compile it if you wish to do so.

Grading scheme

CMR = controlled memory region.

When a testcase is said to "require" another testcase, it means that if the latter (requiree) is not met, the former (requirer) is not run at all. The reason is that some later testcases assume some earlier functionality is implemented; for example, it would not make sense to test and award credit for correctly restoring content from a swapfile (3b) if the pages are not removed from memory in the first place (3a).

All testcases are binary i.e. either all of the credit or none of the credit is awarded.

- Exercise 1 (total 2 points) (includes exercise 0)
    - (0.3) 1a: `userswap_alloc` causes a correctly-sized memory allocation to be created
    - (0.1) 1b: `userswap_alloc` causes a correctly-sized memory allocation with the correct permissions (PROT_NONE) to be created. Requires 1a.
    - (0.3) 1c: `userswap_free` correctly removes a memory allocation created by `userswap_alloc`. Requires 1a.
    - (0.2) 1d: A SIGSEGV handler is installed after `userswap_alloc` is called.
    - (0.2) 1e: The SIGSEGV handler is removed on a faulting access not to a CMR (and therefore the program terminates with a SIGSEGV). Requires 1d.
    - (0.2) 1f: On a read to a non-resident page in a CMR (PROT_NONE), it (and only it) should be made PROT_READ. Requires 1a, 1b and 1d.
    - (0.2) 1g: On a write to a non-resident or PROT_READ page in a CMR, it (and only it) should be made PROT_READ | PROT_WRITE. Requires 1a, 1b and 1d.
    - (0.3) 1h: 1a-1c with multiple live allocations
    - (0.2) 1i: 1e-1g with multiple live allocations and random page offsets
- Exercise 2 (total 1 point)
    - (0.2) 2a: Using the default LORM, with a single allocation, when the LORM is exceeded, pages are evicted (set to PROT_NONE) in FIFO order. Requires 1f.
    - (0.1) 2b: After an allocation is made and some pages are made resident, the LORM is raised using `userswap_set_size`. Pages are evicted in FIFO order according to the new LORM. Requires 1f.
    - (0.2) 2c: After an allocation is made and some pages are made resident, the LORM is lowered to a value below the current size of resident memory. Pages are immediately evicted in FIFO order to meet the new LORM. Requires 1f.
    - (0.2) 2d: An allocation is made and some pages are made resident and then evicted. The evicted pages are accessed again. The pages are correctly made resident again (and the

correct other pages are evicted to allow these pages to be made resident again). Requires 2a.
    - (0.3) 2e: 2a-2d with multiple live allocations and random page offsets. Pages are evicted globally.

- Exercise 3 (total 2 points)
    - (0.1) 3a: MADV_DONTNEED is applied to evicted pages.
    - (0.1) 3k: MADV_DONTNEED is applied to all evicted pages (clean or dirty).
    - (0.4) 3b: With a single allocation, pages that are written to, evicted and then made resident again have their contents correctly preserved. Requires 3a.
    - (0.1) 3c: In the test sequence for 3b, the swap file does not exceed the maximum permitted size. Requires 3b.
    - (0.1) 3d: Pages that are only read from and then evicted are not written to the swap file. Requires 3b.
    - (0.1) 3e: Pages that are written to, evicted, restored, and evicted again are not written to the swap file on their second eviction. Requires 3b.
    - (0.2) 3f: Pages that are written to, evicted, restored, evicted again, and restored again have the correct contents. Requires 3a.
    - (0.5) 3g: 3b with multiple live allocations and random page offsets.
    - (0.1) 3h: 3d with multiple live allocations and random page offsets.
    - (0.1) 3i: 3e with multiple live allocations and random page offsets.
    - (0.2) 3j: 3f with multiple live allocations and random page offsets.

- Exercise 4 (total 2 points)
    - (0.2) 4a: `userswap_map` causes a correctly-sized memory allocation to be created
    - (0.3) 4b: Reading from a mapping created by `userswap_map` results in the correct contents corresponding to the file. Requires 4a.
    - (0.1) 4c: `userswap_map` extends the file to the correct length with zeroes.
    - (0.1) 4d: Reading from the region of memory corresponding to the extended length results in zeroes.
    - (0.2) 4e: Pages in a `userswap_map` mapping that are read from, evicted, and then made resident again have the correct contents on the second read. Requires 3a.
    - (0.2) 4f: Pages in a `userswap_map` mapping that are only read from and then evicted do not result in any writes to any files.
    - (0.4) 4g: One `userswap_alloc` allocation and one `userswap_map` mapping is made. Pages from both are read in an random and interleaved manner, and some pages of the `userswap_alloc` allocation are written to. All behaviour is correct (correct pages are evicted, all reads result in correct contents, etc).

    - (0.5) 4h: 4g with multiple `userswap_alloc` allocations and `userswap_map` mappings and random page offsets.

- Exercise 5 (total 1 point)
    - (0.3) 5a: With a single mapping, pages that are written to, evicted, and then made resident again have their contents correctly preserved. Requires 3a.
    - (0.1) 5b: In 5a, when a modified page is evicted, the contents are immediately visible in the backing file. (i.e. no buffered I/O)
    - (0.1) 5c: Pages that are written to, evicted, restored, and evicted again do not result in any writes on their second eviction. Requires 5a.
    - (0.5) 5d: Overall testcase. Multiple allocations and mappings are made and reads and writes are done to all allocations and mappings. Random page offsets. All behaviour is correct.