CS2106: Introduction to Operating Systems

Lab Assignment 4: User space Swap

Important

The deadline of submission through LumiNUS: **Fri, 12 Nov, 2pm**.

The total weightage is 8% + [Bonus 2%]:

- Exercise 0: Optional 1% demo
- Exercise 1: 1% if Ex0 is demoed, else 2%
- Exercise 2: 1%
- Exercise 3: 2%
- Exercise 4: 2%
- Exercise 5: 1%
- Exercise 6: 2% bonus

You must ensure the exercises work properly on the SoC Compute Cluster, hostnames: xcne0 – xcne7, Ubuntu 20.04, x86_64, GCC 9.3.0.

1. Introduction

As you know from the lectures, swap, or paging, is a way of "extending" the memory capacity of a computer by moving less-used *pages* to secondary storage, thereby making room for new pages, and moving pages back into main memory on-demand.

Although swap is typically implemented and provided by a kernel transparently to user programs, some operating systems provide features that allow programs to emulate swap in a similarly transparent fashion as well.

On Linux and other POSIX systems, when a process performs an invalid memory access, the kernel sends the `SIGSEGV` signal to the process. The default action for the signal is to terminate the process, and, in most cases, this is the only sensible thing to do: an invalid memory access usually implies a severe bug in the program or that memory corruption has occurred.

However, it is also possible to install a signal handler for `SIGSEGV`. While the C and POSIX specifications both state that the behaviour upon returning from a `SIGSEGV` signal handler is undefined, the behaviour on Linux is clear: it will return to the instruction that caused the invalid memory access and re-try the instruction. This means that by installing a `SIGSEGV` handler and performing actions to move memory, etc., it is possible to implement swap in user space. In effect, our `SIGSEGV` handler is acting as a page fault handler.

In this lab, you will implement a user space swap library, as will be further described below.

Frequently asked questions

Frequently asked questions by students for this assignment will be answered here. The most recent questions will be added at the top; questions will be dated. **Check this file before asking questions.** If you have questions regarding the assignment, please ask on the LumiNUS forum created for this assignment.

Reminder for Windows, macOS, and other non-Linux POSIX OS users

This lab uses Linux-specific behaviours, including the ability to return normally from a `SIGSEGV` handler and certain syscall arguments and operations. Your program may compile successfully on macOS or other POSIX OSes, but is unlikely to behave correctly.

If you are on Windows, the lab should work fine in WSL 2 (given that WSL 2 is just Linux), but it may not work in WSL 1.

Regardless, you are strongly encouraged to test your implementation on the SoC Compute Cluster xcne nodes, as that is where grading will be performed.

2. Overall specifications

The overall specifications for the library are as follows. If you are up for a challenge, you may implement the lab solely based on these specifications; otherwise, they are broken down into exercises below. In either case, please read the specification carefully, as it defines key terms that will be used in the rest of the document.

Controlled memory regions

A controlled memory region is a memory region controlled by the user space swap library.

In the rest of the document, we use **page fault** to denote when a memory access to a controlled memory region causes a `SIGSEGV`. Note that this is **independent** of whether the memory access causes an *actual* page fault in the kernel.

A page of memory in a controlled memory region has these properties:

- **Residency**: A **non-resident** page is not present in memory, and accessing it will cause a page fault. A **resident** page is present in memory, and must not cause a page fault on access.
- **Contents**: Each page of memory has some contents. These contents are independent of the residency of the page, and must be preserved when a page is evicted and then brought back into memory.
- **Dirtiness**: A page is **dirty** if it has been written to since the last time the page was made resident; otherwise, the page is **clean**.

A controlled memory region is allocated as private anonymous pages using `mmap`. The pages should initially be in the **non-resident** state. If the memory region is allocated using `userswap_alloc`, then the pages' contents are initialised to zero. If the region is allocated using `userswap_map`, they should contain the contents of the backing file at the corresponding locations.

Upon a page fault accessing a **non-resident** page, the page should be brought into memory and thereby made resident. If doing so will cause the total size of resident memory in all controlled memory regions to exceed the LORM (Limit Of Resident Memory, defined in a section below), then *before* the new page is brought into memory, the **minimum** number of pages should be evicted according to the page eviction algorithm until the total size after the new page is brought in will be under or equal to the LORM.

If a dirty page in an allocation by `userswap_alloc` is evicted from memory, the contents should be saved in a swap file. Each process should have at most one swap file regardless of the number of allocations made. The swap file should be named `<pid>.swap`, in the current working directory of the process, where `<pid>` is the PID of the process (e.g., `12345.swap` if the PID is 12345). The layout of the swap file is up to you, but its size must not exceed the maximum value since the start of execution of the total amount of memory in all unfreed controlled memory regions that were allocated by `userswap_alloc`. There is no need to shrink swap files after allocations

are freed, **but** the swap file must not be extended if there is sufficient space already, e.g., due to previous allocations that have since been freed. Your implementation should be able to deal with an existing swap file; it is sufficient to just truncate or delete the existing file.

If a dirty page in an allocation by `userswap_map` is evicted from memory, its contents should be updated in the backing file in the corresponding location.

If a **clean page** is evicted from memory, **no file writes should be done due to that eviction.** This is because if the page is already in a swap file or a backing file and it is clean, its contents in the file should be identical to that in memory. Hence no writes are needed. Note that a page that was allocated by `userswap_alloc` and that is read from but never written to, and therefore containing all zeroes, is considered a clean page.

When a page is evicted from memory, the kernel must be advised to actually free the backing physical pages by using `madvise` with `MADV_DONTNEED`; see the man pages for `madvise`.

`SIGSEGV` handler

The signal handler should check if the faulting memory access is to a controlled memory region. If so, the fault should be handled as described above. If not, it should **remove itself as a `SIGSEGV` signal handler**, reset the action taken for a `SIGSEGV` signal, and return immediately, in order to allow the program to crash as it would without the user space swap library.

Page eviction algorithm

A first-in-first-out eviction algorithm should be used. That is, when a page is to be evicted, the page that was least recently made resident should be chosen for eviction. This applies **across all** controlled memory regions i.e., this is a global replacement algorithm.

Limit of resident memory in all controlled memory regions (LORM)

This is a global setting that can be controlled by `userswap_set_size`. The limit applies to all controlled memory regions in **total**, not per-region. The default value of the LORM should be 8,626,176 bytes (that is, 2,106 pages :-)), i.e., this is the value of the LORM prior to any calls to `userswap_set_size`.

Function specifications

- `void *userswap_alloc(size_t size);`

  This function should allocate `size` bytes of memory that is controlled by the swap scheme described above in the "Controlled memory regions" section, and return a pointer to the start of the memory.

  If `size` is not a multiple of the page size, `size` should be rounded up to the next multiple of the page size.

  This function may be called multiple times without any intervening `userswap_free`, i.e., there may be multiple memory allocations active at any given time.

  If the `SIGSEGV` handler has not yet been installed when this function is called, then this function should do so.

- `void *userswap_map(int fd, size_t size);`

  This function should map the first `size` bytes of the file open in the file descriptor `fd`, using the swap scheme described above in the "Controlled memory regions" section.

  If `size` is not a multiple of the page size, `size` should be rounded up to the next multiple of the page size.

  The file shall be known as the **backing file**. `fd` can be assumed to be a valid file descriptor opened in read-write mode using the `open` syscall, but no assumptions should be made as to the current offset of the file descriptor. The file descriptor, once handed to `userswap_map`, can be assumed to be fully controlled by your library, i.e., no other code will perform operations using the file descriptor.

  If the file is shorter than `size` bytes, then this function should also cause the file to be zero-filled to `size` bytes.

  Like `userswap_alloc`, this function may be called multiple times without any intervening `userswap_free`, i.e., there may be multiple memory allocations active at any given time.

  If the `SIGSEGV` handler has not yet been installed when this function is called, then this function should do so.

- `void userswap_free(void *mem);`

  This function should free the block of memory starting from `mem`.

`mem` can be assumed to be a pointer previously returned by `userswap_alloc` or `userswap_map`, and that has not been previously freed.

If the memory region was allocated by `userswap_map`, then any changes made to the memory region must be written to the file accordingly. The file descriptor should **not** be closed.

- `void userswap_set_size(size_t size);`

  This function sets the LORM to `size`.

  If `size` is not a multiple of the page size, `size` should be rounded up to the next multiple of the page size.

  If the total size of resident memory in all controlled regions is above the new LORM, then the **minimum** number of pages should be evicted according to the page eviction algorithm until the total size is under or equal to the LORM.

Assumptions

You may assume that:

- The size of a page is `4096`. You *may*, but are not required to, ignore the high 16 bits of virtual addresses when designing your data structures.
- All arguments passed to the userswap functions are valid.
- All system calls made with valid arguments succeed. Despite this, you should still check the success of every system call, and emit some warning, as they may fail during development due to programmer error, e.g., providing wrong arguments. Omitting error-checking may lead to difficult-to-debug issues.
- There will be no concurrent accesses to controlled memory regions. (Relaxed in the bonus exercise.)
- There will be no attempts to execute memory in a controlled memory region.
- No other code will alter the memory protection of controlled memory regions using `mprotect`, or perform operations on those regions using `madvise`, etc.

Limitations

- Your implementation should have no more than about 128 bytes of overhead (i.e., used for metadata tracking the state of each page) per page of memory in a single allocation. It is okay to exceed this limit for small allocations below 512 pages. This requirement will not be strictly checked, but egregious cases may be penalised.
- You **must not** use the `mmap` syscall to map a file into memory to perform file I/O. That is, when you use `mmap`, the `fd` argument must always be `-1`, even for `userswap_map`.
- Your implementation should work with reasonable performance. We will not be grading based on performance (except for the bonus), but all provided test workloads should still run within reasonable time (i.e., less than 10 seconds).

3. Implementation guide

Please read the overall specifications before reading this section. Each successive exercise builds upon previous exercises; when you complete all exercises, you will have fulfilled the specifications above.

In the lab archive, you will find these files:

- `userswap.c`: A skeleton for you to start from. You should implement everything in this file.
- `userswap.h`: Prewritten header; defines function prototypes.
- `Makefile`: Prewritten Makefile.
- `workload_*.c`: Provided workloads to test your library, as described above.

You should modify **only** `userswap.c`. If you write additional workloads for your own testing, you may wish to add them to the Makefile for your convenience.

Compiling, running and testing

To compile the workloads with your library, simply type `make`. The `-Werror` flag is set. You may remove it while developing, but you are strongly advised to fix all warnings before submitting, as warnings are indicative of undefined behaviour. When grading, we will do so *without* `-Werror`, but all other flags will be kept as-is.

In each exercise below, workloads are suggested to test the functionality in that exercise. All workloads are self-contained and require no input, so you can simply run the workload and inspect the output/result. Note that the suggested workloads are in no way exhaustive tests; passing them does not guarantee full credit.

- workload_readonly: This workload simply allocates an array and then reads each element of the array as an integer, sums all the elements, and then prints the sum.
- workload_wraddr: This workload simply allocates an array of pointers, and then writes the address of the array element to each element itself. It then goes back to read each element of the array, verifying each element as it reads. No output is printed if it succeeds.
- workload_rdfile: This workload creates a file with some data, and then maps the file and verifies that the contents of the mapping are identical to those of the file.
- workload_wrfile: This workload does the same as workload_rdfile, then it writes to the mapping, frees the mapping and then verifies that the file contents are updated.

A tip for debugging: GDB by default breaks on `SIGSEGV` even though there is a signal handler for it. To disable this, use the GDB command `handle SIGSEGV nostop`. You can also make GDB not print on each `SIGSEGV` by `handle SIGSEGV noprint`.

3.1. Exercise 0 [Optional 1% demo]

To get started, in this exercise, you will simply get to a base upon which you can build the rest of the library.

1. Implement `userswap_alloc` to simply allocate the requested amount of memory (rounded up as needed; see the specifications) using `mmap`. Per the specifications, the memory should be initially non-resident and therefore should be allocated as `PROT_NONE`, in order for any accesses to the memory to cause a page fault. `userswap_alloc` should also install the `SIGSEGV` handler, if it has not already been done.
2. Implement `userswap_free`, which should free the entire allocation starting at the provided address using `munmap`.
    - You will need to track the size of each allocation in `userswap_alloc`. A simple linked list of allocations, storing the start address (from mmap) and the size, will be sufficient, but you are free to design and use more performant structures.
3. Write a `SIGSEGV` handler. For this exercise, the handler does not need to check whether the faulting memory address is within a controlled memory region; it can simply call the page fault handler. The page fault handler will need the address to perform `mprotect`, however; the faulting memory address can be found in the `siginfo_t` struct passed to the signal handler.
4. Write a function to handle page faults. This is where the bulk of the logic for the "Controlled memory regions" will reside. For this exercise, the page fault handler only needs to use `mprotect` to make the page containing the accessed memory `PROT_READ`, thereby making the page resident.
    - Note that the specifications say that memory newly allocated by `userswap_alloc` should be initialised to zero. Nothing needs to be done for this, as memory allocated by `mmap` is always initialised to zero.

Suggested test workloads: workload_readonly

Syscall hints: mmap, mprotect, munmap, sigaction

3.2. Exercise 1 [1% if Ex0 is demoed, else 2%]

In this exercise, you will implement more of the basic functionality of the library, stopping short of page eviction.

1. Extend the `SIGSEGV` handler to verify that the faulting memory address is actually in a controlled memory region, and if not, it should reset the action for the `SIGSEGV` signal and return. This completes the functionality for the `SIGSEGV` handler.
2. Extend the page fault handler so that, upon a write, the accessed page is made `PROT_READ | PROT_WRITE`. In the case where non-resident memory is written to, it is acceptable to take two page faults, where the first makes the page `PROT_READ`, and the second makes the page `PROT_READ | PROT_WRITE`. To do this, you will need to track the state of each page; if a page fault happens on a resident page, the only possibility (given the assumptions) is that a write was attempted on a `PROT_READ` page.
    - Pages are only made `PROT_WRITE` on the second fault so that it is possible to tell which pages are dirty, which will be needed for later exercises.
    - It is also possible to directly figure out if a `SIGSEGV` was caused by a read or write, but this is not required.
3. Extend `userswap_free` so that it cleans up the data structures corresponding to the allocation as well.

Here, you will need to design data structures to track the state of pages. You are free to design them as you wish. Remember that there can be multiple controlled memory regions. Keep in mind the overall requirements (i.e., the requirements of subsequent exercises) when designing these data structures. Also take note of the limit on memory overhead as mentioned above, although it is unlikely that you will hit the limit unintentionally.

Note that the data structures you create for this lab will be slightly different from those used by an OS in that they do not map logical or virtual addresses to physical addresses; they just store the state of pages.

Here are some suggestions for data structures:

- A 4-level page table, mirroring the structure used by x86_64. (The assumption above that allows you to ignore the high 16 bits of virtual addresses is relevant for this structure.) This has constant-time lookup and update once every level of page table is initialised for a particular address.
- Some kind of list of allocations that contains a list of pages in each allocation. The lookup and update performance will depend on how the list of allocations is designed.

Suggested test workloads: workload_wraddr

Syscall hints: No new syscalls needed.

3.3. Exercise 2 [1%]

In this exercise, you will implement page eviction, but without swap.

1. Extend the page fault handler so that, if the LORM will be exceeded when making a page resident, a page should be evicted according to the page eviction algorithm *before* the new page is made resident. In this exercise, it will be sufficient to make the page `PROT_NONE`, without performing `madvise(MADV_DONTNEED)`; therefore, a swap file is not needed in this exercise.
2. Implement the `userswap_set_size` function.

You will likely need an additional data structure for the page eviction algorithm. Some form of queue of resident pages, such as a doubly linked list or dynamic array will work well here. Remember to extend `userswap_free` so that it updates that data structure as well when an allocation is freed.

Suggested test workloads: workload_wraddr

Syscall hints: No new syscalls needed.

3.4. Exercise 3 [2%]

In this exercise, you will implement swap. Extend the page fault handler so that:

1. when a page is evicted **and** it has been modified since it was last brought into memory (i.e., the page is **dirty**), its new contents are written to a swap file, and then the physical page is freed by calling `madvise(MADV_DONTNEED)` on the page, in addition to it being set to `PROT_NONE`. The order of `madvise(MADV_DONTNEED)` and `mprotect(PROT_NONE)` does not matter. The swap file should be named and placed accordingly as described in the overall specifications, but the structure of the swap file is up to you to design.
2. when a non-resident page is brought into memory, and it has been previously evicted into a swap file (i.e., not a freshly allocated page), its contents are restored from the swap file.

When designing your swap file format, keep in mind that multiple separate allocations may exist simultaneously, but there may only be one swap file for the entire process.

In general, a swap file does not need to have a particular structure and can simply be a file containing page contents. When a page is evicted from memory, you will need to find a place in the swap file to evict it to, and track the location so that you can read the page back into memory later on. The most straightforward place to record this information is in your page table or similar structure from exercise 1. You will also need to keep track of locations in the swap file corresponding to pages that have been freed, so that those locations can be reused. A simple linked list, or other structure, of free swap file locations will suffice.

You should avoid using buffered I/O for reads and writes to your swap file, as that will defeat the purpose of the library, since memory would be allocated for the buffers and evicted pages stored in those buffers. You *can* use the `stdio` functions if you wish, as long as you disable buffering on the swap file, although we would suggest using the direct syscalls.

There is no need to close the swap file. At the same time, your implementation should be able to deal with an already existing swap file; it is sufficient to just truncate or delete the existing file if one is seen (since the file is uniquely named by PID), or do nothing if your implementation can ignore the existing data.

Suggested test workloads: workload_wraddr

Syscall hints: madvise, open, read/pread, write/pwrite

3.5. Exercise 4 [2%]

In this exercise, you will implement `userswap_map`, i.e., user space memory-mapped files, but for reading only.

1. Implement `userswap_map`, which is mostly the same as `userswap_alloc`, except that it needs to record that this allocation is backed by the file given. It should also length-extend the file as needed, as described in the specifications.
2. Extend the page fault handler so that, upon a read to a non-resident page allocated by `userswap_map`, the page is filled with the contents of the file at the corresponding location. Also, when a page allocated by `userswap_map` is evicted, for this exercise, it is sufficient to just discard the page (i.e., `madvise(MADV_DONTNEED)` and `mprotect(PROT_NONE)`) without storing the contents anywhere.

Suggested test workloads: workload_rdfile

Syscall hints: pread, fstat, ftruncate

3.6. Exercise 5 [1%]

In this exercise, you will implement write support for your user space memory-mapped files functionality.

1. Extend the page fault handler so that, when a page that was allocated by `userswap_map`, and that has been modified since it was last brought into memory, is evicted, its contents are written back to the corresponding location in the backing file. If the page was not modified, it should simply be discarded, as in the previous exercise.
2. Extend `userswap_free` so that any dirty pages are written back to the backing file.

Suggested test workloads: workload_wrfile

Syscall hints: pwrite

3.7. Exercise 6 [Bonus 2%]

In this exercise, you will make this library be able to support **concurrent reads and writes** to controlled memory regions.

You may use any synchronisation mechanism that works on the Compute Cluster. The bonus score you receive will depend on the performance and quality of your synchronisation mechanism; for example, a simple mechanism like a global mutex on the page fault handler may work, but will receive little credit.

Implement this exercise in a copy of `userswap.c` named `bonus_userswap.c`. Ex1–5 will be graded using `userswap.c`, and Ex6 will be graded using `bonus_userswap.c`; of course, your bonus implementation should be able to meet Ex1–5's requirements as well. Include in a text file `bonus_userswap.txt` a short description of the idea behind your synchronisation mechanism.

4. Check your archive before submission

Before you submit your assignment, run the archive check script named `check_zip.sh`. The script checks the following:

1. The name of the archive you provide matches the naming convention prescribed in section 5.
2. The archive can actually be extracted, and the directory structure matches the structure prescribed in section 5.
3. All required files are present.
4. All source files can be compiled.

Once you have created the zip archive, check it by running `bash ./check_zip.sh E0123456.zip`, replacing `E0123456.zip` with the actual name of your zip file.

The script will print errors if your zip file does not meet the requirements. Ensure that no errors are printed. Otherwise, we may have difficulty grading your assignment, which may result in lost credit.

Expected successful output

```
Unzipping file: E0123456.zip
Success!
```

5. Submission through LumiNUS

Zip the following files as `E0123456.zip`, where **E0123456** is your NUSNET ID. Use your NUSNET ID, **not** your matriculation number A0123456X, and use capital E as the prefix.

- `userswap.c`
- `bonus_userswap.c` (only if you are submitting exercise 6)
- `bonus_userswap.txt` (the description of your synchronisation mechanism, only if you are submitting exercise 6)

There should be no folders within the zip archive at all. (macOS users, in particular, should check that the archiver tool does not create an outer directory in the archive.)

Upload your zip file to the "Lab Assignment 4" submission folder on LumiNUS. The deadline for submission is **Fri, 12 Nov, 2pm**.

Please ensure that you follow the instructions carefully. Deviations may be penalised.