Vectored I/O & Memory-Mapping

CISC 3350 — CUNY Brooklyn College
Lecture Notes

Vectored I/O & Memory-Mapping

This topic covers two aspects of Advanced File I/O: (1) Scatter/gather I/O [= Vectored I/O] and (2) Memory-mapped I/O.

Scatter/Gather I/O, also known as **Vectored I/O**, is a method in which a single syscall reads or writes a **vector** (= array) of buffers from a single data file stream.

The advantages of Scatter/Gather I/O over **linear I/O** (= regular I/O as we've covered so far) are:

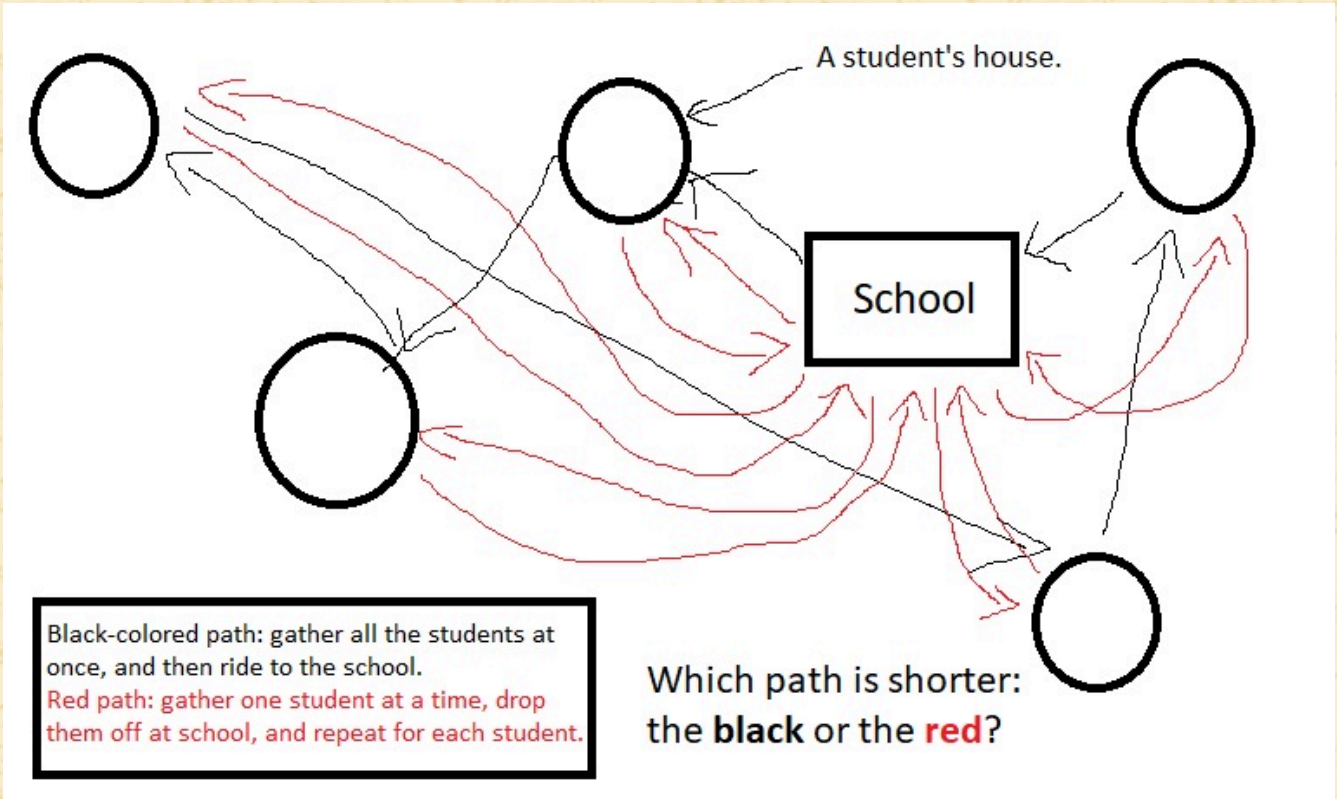
- **More natural coding pattern**: intuitive for naturally segmented data, e.g., multiple lines or strings of text.
- **Efficiency**: a single vectored I/O operation is used instead of multiple linear I/O calls.
- **Performance**: the OS can optimize the I/O operations done via Scatter/Gather I/O, making this method faster than multiple linear I/O calls.
- **Locked**: there is no risk of interleaving (= messing up) I/O from another process or thread.

Vectored I/O: Analogy

Consider the route of some school bus in the morning: having the school bus pick up the kids from their homes, and, only after all the kids are on the bus, driving to the school is going to be considerably faster than picking up one kid, driving to the school and dropping the kid off at the school, picking up the next kid, and then driving to the school again, etc.

A school bus picking up students. Miriam Briskman, CC BY-NC 4.0.

Vectored I/O: Analogy



Which one of the two routes is shorter? Miriam Briskman, CC BY-NC 4.0.

Vectored I/O: The System Calls

<table>
<tr><td colspan="2">Name: <code>readv()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: reads <code>count</code> segments from the file <code>fd</code> into the array of structures <code>iov</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <sys/uio.h> // Defines readv().</code></td></tr>
<tr><td colspan="4">Syntax: <code>ssize_t readv (int fd, const struct iovec *iov, int count);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>count</code>: number of segments (e.g., strings, integers, chars, etc.) that you want to read in.
<code>fd</code>: file descriptor of an open file for reading.
<code>iov</code>: an array of the following structures:
<pre>struct iovec {
    void *iov_base; /* pointer to start of a buffer of any data type */
    size_t iov_len; /* size of the buffer in bytes */
};</pre></td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>ssize_t</code></td><td colspan="2">On success: The overall number of bytes that were read in (= the sum of all <code>iov_lens</code>.)</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>if (readv (fd, arr, 3) == -1) /* Handle the error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Vectored I/O: The System Calls

<table>
<tr><td colspan="2">Name: <code>writev()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: writes at most <code>count</code> segments from the the array of structures <code>iov</code> to the file <code>fd</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <sys/uio.h> // Defines writev().</code></td></tr>
<tr><td colspan="4">Syntax: <code>ssize_t writev (int fd, const struct iovec *iov, int count);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>count</code>: number of segments (e.g., strings, integers, chars, etc.) that you want to write out.
<code>fd</code>: file descriptor of an open file for writing.
<code>iov</code>: an array of the following structures:
<pre>struct iovec {
    void *iov_base; /* pointer to start of a buffer of any data type */
    size_t iov_len; /* size of the buffer in bytes */
};</pre></td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>ssize_t</code></td><td colspan="2">On success: The total number of bytes that were written out (= the sum of all <code>iov_lens</code>.)</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>if (writev (fd, arr, 3) == -1) /* Handle the error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

writev_example.c: Example of Using `writev()`


```c
/* This program exemplifies the vectored I/O
 *    writev() function. It writes 3 lines of text
 *    into a file using the single system call
 *    writev(). Note that writev(), being a system
 *    call, works on file descriptors (as covered
 *    in Topic 4) and not on file pointers.
 *
 *    Miriam Briskman, 3/8/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <stdio.h>     // printf(), perror()
#include <stdlib.h>    // EXIT_SUCCESS, EXIT_FAILURE
```

readv_example.c: Example of Using `readv()`


```
/* This program exemplifies the vectored I/O
 *    readv() function. It assumes the existence
 *    of the file 'seasons.txt' within the current
 *    directory and that it has 3 lines of text in
 *    it. All 3 lines of text will be read in with
 *    the single system call readv(). Note that
 *    readv(), being a system call, works on file
 *    descriptors (as covered in Topic 4) and not
 *    on file pointers.
 *
 *    Miriam Briskman, 3/8/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */
```

Vectored I/O

Any questions?

Memory-Mapped I/O

In **Memory-Mapped I/O**, the contents of a file are copied from disk to main memory and treated as a char array. This way, instead of reading from disk or writing to disk each time, you can access the file contents directly from main memory, which is more than **1000** times faster than accessing a file on disk!

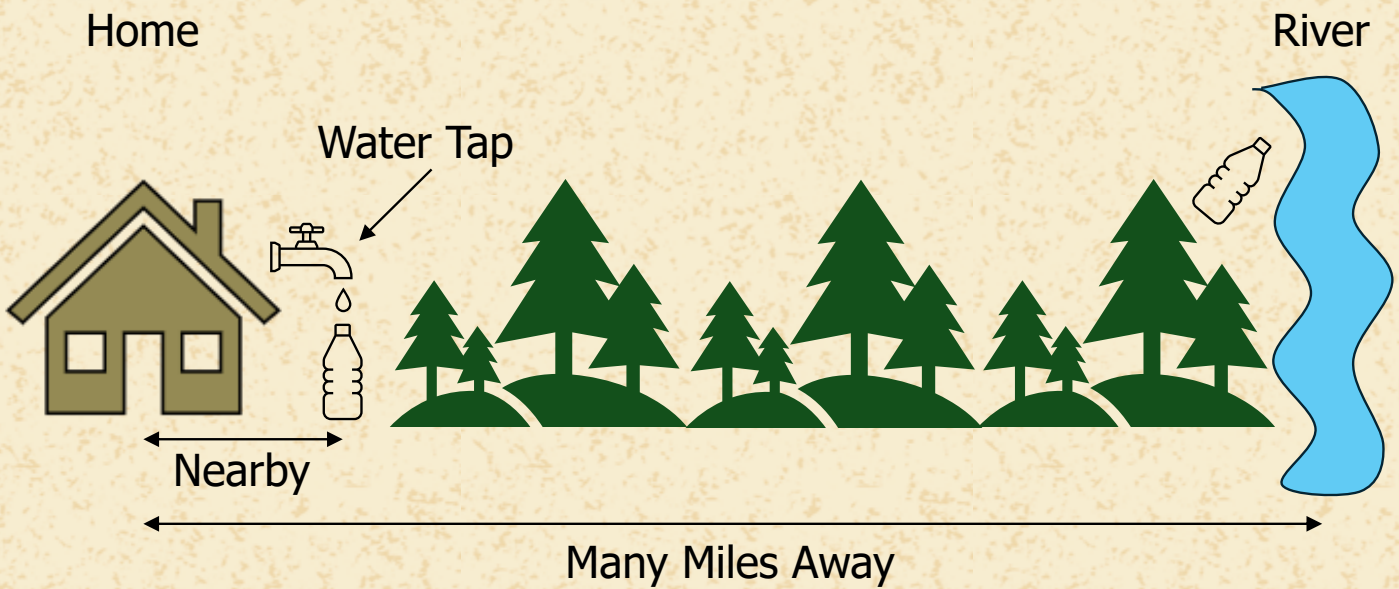
Later, the Linux OS will back up any changes that were made to that char array (in case you used the file for writing) back to disk. No backup to disk is needed when you merely read from a file.

As we learn later this semester, main memory is divided into equally-sized regions of about 4K bytes called **pages**.

The system call that is used to memory-map a file from disk to main memory, `mmap()`, copies the content of the file in terms of the page size: if the length of the file is not equal to a multiple of the page size, an additional page in main memory is used (and usually stays half-empty – just like a page at the end of a book that might be half-empty.)

Memory-Mapped I/O: Analogy

Suppose there are two ways your household can get water: either by walking to the nearby river, which is miles away, or using the water tap installed outside the house. Which of these two ways lets you get water faster?



Which of these two ways lets you get water faster? Miriam Briskman, CC BY-NC 4.0.

Memory-Mapped I/O: The System Calls

Name: `mmap()`	**System call or function:** System Call	**Links:** Online Manual | Course Packet
What it does: maps (copies) the contents of the file `fd` to main memory.		
What libraries you must include: `#include <sys/mman.h> // Defines mmap().`		
Syntax: `void * mmap (void *addr, size_t len, int prot, int flags, int fd, off_t offset);`		
Description of arguments: `addr`: a hint to OS where to map file. Pass `0` for the OS to decide itself. `len`: number of bytes you want to copy from the file represented by `fd`. `prot`: memory protection mode: one or more of `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`. `flags`: type of mapping: one of `MAP_FIXED`, `MAP_PRIVATE`, `MAP_SHARED`, etc. `fd`: file descriptor of an open file for writing. `offset`: offset of bytes (= how deep inside the file you want to start copying.)		
What type it returns: `void*`	**On success:** an address (pointer) in memory to the beginning of the file's content.	
	On failure: `MAP_FAILED` (= `((void*) -1)`)	
	If failure, does it set `errno`? Yes	
Quick example: `if ((content = mmap (0, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)) == MAP_FAILED) /* error */`		
Other variations: N/A		

Memory-Mapped I/O: The System Calls

<table>
<tr><td colspan="2">Name: <code>sysconf()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: Returns some OS configuration info based on what configuration <code>name</code> you passed. Pass <code>_SC_PAGESIZE</code> to find the page size of the OS.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines sysconf().</code></td></tr>
<tr><td colspan="4">Syntax: <code>long sysconf (int name);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>name</code>: the name of the configuration whose info you want to see.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: a non-zero integer representing the info of the configuration.</td></tr>
<tr><td colspan="2">On failure: <code>-1</code>, but it could also be a legit configuration info.</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:
<pre>errno = 0;
long page_size = sysconf (_SC_PAGESIZE);
if (errno != 0)
    /* Handle the error */</pre></td></tr>
<tr><td colspan="4">Other variations: <code>getpagesize()</code></td></tr>
</table>

Memory-Mapped I/O: The System Calls

<table>
<tr><td colspan="2">Name: <code>getpagesize()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: Returns the page size of each page in main memory.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines getpagesize().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int getpagesize (void);</code></td></tr>
<tr><td colspan="4">Description of arguments: – This system call doesn't accept any argument –</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="3">On success: a non-zero integer representing the page size.</td></tr>
<tr><td colspan="3">On failure: – This system call doesn't fail –</td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>printf ("The OS's page size is: %d.\n", getpagesize());</code></td></tr>
<tr><td colspan="4">Other variations: <code>sysconf()</code></td></tr>
</table>

Not all Unix and Linux systems support this function. For portability, using `sysconf()` is recommended over using `getpagesize()`.

Memory-Mapped I/O: The System Calls

<table>
<tr><td colspan="2">Name: <code>munmap()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: unmaps the file that was mapped to main memory at addresses <code>addr</code> down to address <code>addr + len</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <sys/mman.h> // Defines munmap().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int munmap (void *addr, size_t len);</code></td></tr>
<tr><td colspan="4">Description of arguments:
<code>addr</code>: the pointer that you obtained from the call to <code>mmap()</code>
<code>len</code>: the number of bytes used in main memory to store the file's content (you passed this number to <code>mmap()</code>.)</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>if (munmap (content, len) == -1) /* error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

After calling `munmap()`, if you continue accessing the file as a char array, you'll get a segmentation fault.

mmap_example.c: Example of Reading from a Memory-Mapped file


```c
// The following program exemplifies the usage of
//     mmap(), which is used to memory-map a file
//     into main memory to use it as a variable
//     and munmap() which cancels this mapping.

// This program is taken from Linux System Programming:
//     Talking Directly to the Kernel and C Library,
//     2nd Edition, by Love. ISNB: 978-1-44933953-1,
//     pages 109-110.

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
```

Memory Mapped I/O

Any questions?

Memory-Mapped I/O

The advantages of memory-mapped I/O are:

- Reads and writes avoid the extraneous copy into a user-space buffer.
- Avoid incurring more syscalls (though may incur potential page faults.)
- When multiple processes map the same object into memory, the data is shared among all of them.
- Seeking through the content involves trivial memory manipulations (faster than reading from a file.)

The disadvantages of memory-mapped I/O are:

- "Wasted" space between a small file and the integer number of pages (4K).
- Large numbers of mappings may increase fragmentation of (unused holes) the process's address space in main memory.
- Overhead in creating/maintaining mappings and associated data structures in kernel, though the elimination of double copy helps reducing the overhead.

Using memory mapping is, therefore, advised when (1) the mapped file is relatively large, (2) the total size of the mappings is a multiple of the page size, and (3) the file is frequently accessed.

Memory-Mapped I/O: The System Calls

<table>
<tr><td colspan="2">Name: <code>mremap()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: expands or shrinks the mapping at memory address <code>addr</code> from <code>old_size</code> to <code>new_size</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <sys/mman.h> // Defines mremap().</code></td></tr>
<tr><td colspan="4">Syntax: <code>void * mremap (void *addr, size_t old_size, size_t new_size, unsigned long flags);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>addr</code>: the start address to the mapping.
<code>old_size</code>: the current size of the mapped content (in bytes).
<code>new_size</code>: the desired new size of the mapped content (in bytes).
<code>flags</code>: either <code>0</code> or <code>MREMAP_MAYMOVE</code>.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>void*</code></td><td colspan="2">On success: an address (pointer) in memory to the beginning of the file's content.</td></tr>
<tr><td colspan="2">On failure: <code>MAP_FAILED</code> (= <code>((void*) -1)</code>)</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>if ((content = mremap (content, len, len*2, 0)) == MAP_FAILED) /* error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Another Use for `mremap()`

The GNU C Library (= **glibc**) often uses `mremap()` to implement `realloc()`, which resizes a block of memory obtained via `malloc()`.

For example, here is a possible definition that the GNU C Library uses for `realloc()`:

```
void * realloc (void *addr, size_t len)
{
    size_t old_size = look_up_mapping_size (addr); // A function you wrote to find the old size
    void *p = mremap (addr, old_size, len, MREMAP_MAYMOVE);
    if (p == MAP_FAILED)
        return NULL;
    return p;
}
```

Changing the Protection of a Mapping

Name: `mprotect()`	**System call or function:** System Call	**Links:** Online Manual | Course Packet
What it does: changes the protection mode of a memory-mapped region in main memory.		
What libraries you must include: `#include <sys/mman.h> // Defines mprotect().`		
Syntax: `int mprotect (void *addr, size_t len, int prot);`		
Description of arguments: `addr`: any address in main memory to which the program has access. `len`: the number of bytes for which we want to update the protection mode. `prot`: one or more of: `PROT_NONE` (no access allowed), `PROT_READ`, `PROT_WRITE`, or `PROT_EXEC`.		
What type it returns: `int`	**On success:** `0`	
	On failure: `-1`	
	If failure, does it set `errno`? Yes	
Quick example: `if (mprotect (content, len, PROT_READ | PROT_WRITE | PROT_EXEC) == -1) /* error */`		
Other variations: N/A		

Trying to read, for example, from a memory region whose protection is `PROT_WRITE` only, will result in a segmentation fault.

Flushing Mapping's Changes Back to Disk

Name: `msync()`	**System call or function:** System Call	**Links:** Online Manual | Course Packet
What it does: copies all the changes that were made to the mapping back to disk.		
What libraries you must include: `#include <sys/mman.h> // Defines msync().`		
Syntax: `int msync (void *addr, size_t len, int flags);`		
Description of arguments: `addr`: any address inside the mapped region of main memory. `len`: the number of bytes that we want to flush to disk. `flags`: exactly one of `MS_ASYNC` or `MS_SYNC`, plus possibly the flag `MS_INVALIDATE`.		
What type it returns: `int`	**On success:** `0`	
	On failure: `-1`	
	If failure, does it set `errno`? Yes	
Quick example: `if (msync (content, len, MS_SYNC | MS_INVALIDATE) == -1) /* error */`		
Other variations: N/A		

It isn't required to flush back the entire file: you may only flush several bytes from the mapping back to disk, as you choose.

Flushing Mapping's Changes Back to Disk

What each one of the `flags` means:

- `MS_ASYNC`: returns immediately, and simply tells the OS to schedule a sync of the data as soon as the OS can.
- `MS_SYNC`: won’t return (= will start waiting) until all the pages are written back to disk.
- `MS_INVALIDATE`: In case copies of the mapped file are also stored in cache or another place in the OS, this flag tells the OS to discard these copies, and, instead, store the most 'fresh' data that was recently written to main memory.

Note that, in `flags`, you must write either of `MS_ASYNC` or `MS_SYNC`, and, if wanted, also add `MS_INVALIDATE` to one of these. As such, all the possible flag options are:

- `MS_ASYNC`
- `MS_ASYNC | MS_INVALIDATE`
- `MS_SYNC`
- `MS_SYNC | MS_INVALIDATE`

Giving Advice to the OS about the Mapping

<table>
<tr><td colspan="2">Name: <code>madvise()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: hints to the OS about when and how we plan to read data from the mapping, which allows the OS to prepare the memory-mapped region, choose the best caching techniques, and optimize the OS's behavior and performance.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <sys/mman.h> // Defines madvise().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int madvise (void *addr, size_t len, int advice);</code></td></tr>
<tr><td colspan="4">Description of arguments:
<code>addr</code>: the start address to the mapping.
<code>len</code>: the size of the mapped content (in bytes).
<code>advice</code>: exactly one of the following: <code>MADV_NORMAL</code>, <code>MADV_RANDAM</code>, <code>MADV_SEQUENTIAL</code>, <code>MADV_WILLNEED</code>, and <code>MADV_DONTNEED</code>.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example: <code>if (madvise (content, len, MADV_SEQUENTIAL)) == -1) /* error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

Giving Advice to the OS about the Mapping

What each one of the `advice` constants means:

- `MADV_NORMAL`: This advice doesn't indicate anything specific about the expected reading/access to the mapping. As such, the OS will prepare itself for a moderate amount of **readahead** (= reading further, nearby parts of the data inside the mapped region, and storing them in cache.)
- `MADV_RANDOM`: we tell the OS that we don't plan on accessing data in the mapped region in a sequential pattern, and that we will access it sparsely. The OS will disable readahead because reading ahead isn't really useful in this scenario.
- `MADV_SEQUENTIAL`: we expect to read data from the mapped region in sequential order. The OS will perform a lot of readahead in this case.
- `MADV_WILLNEED`: we expect to read data from the mapped region very soon. It might be a good idea for the OS to read ahead a little bit.
- `MADV_DONTNEED`: we don't expect to read data from the mapped region very soon. The OS can, therefore, free up cache for other important data unrelated to our file mapping.

Memory Mapped I/O

Any questions?

Vectored I/O & Memory-Mapping: Sources

Topic 5: Vectored I/O & Memory-Mapping, Miriam. *CISC 3350 OER Course Packet.* 2024, CC BY-NC 4.0.
URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/CISC_3350_OER_Course_Packet.pdf#topic+6:+vectored+i/o+%26+memory-mapping.

Partly based on the teaching of Professor **Yongqing Xiang** of Brooklyn College.