Mmap

• Learning Objectives
  • Use mmap (and it’s associated calls).
  • Explain the relationship between mmap and caching.
  • Discuss the pros and cons of using mmap.
What is mmap?

• From the man page:

```c
void *mmap(void *addr,
           size_t len, int prot, int flags,
           int fd, off_t offset);
```

• `mmap()` creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in `addr`. The `length` argument specifies the length of the mapping.

• The `mmap()` system call causes the pages starting at `addr` and continuing for at most `len` bytes to be mapped from the object described by `fd`, starting at byte offset `offset`. 
Let's say I have a file that is 8192 byte long and I want to map the entire file into my address space, what values would I use for length and offset respectively?

Select font size  

- 0, 8192 [✓]
- 2, 4096 [✓]
- 4096, 2 [✓]
- 8192, 0 [✓]
Recall our friend the address space
Mapping into an address space

0xFFFFF0000
Memory mapped region for shared libraries

0x08048000
mmap’s parameters (1)

• The obvious ones:
  • \texttt{fd}: a file descriptor representing the file you are mapping.
  • \texttt{len}: the number of bytes you are mapping.
  • \texttt{offset}: the position in the file at which to begin mapping.
  • Note: \texttt{len} and \texttt{offset} really should be multiples of the system’s underlying page size.
mmap’s parameters (2)

- **flags:**
  - Exactly one of:
    - `MAP_SHARED`: Updates to the mapped file are visible to other processes and are written back to the file.
    - `MAP_PRIVATE`: Updates to the mapped file are not visible to other processes and are not written back to the file.
  - Optionally any of:
    - `MAP_32BIT`: Place the mapping in the first 2 GB of address space.
    - `MAP_ANON/MAP_ANONYMOUS`: Do not associate mapping with a file (`fd` and `offset` are ignored).
    - `MAP_FIXED`: Place the mapping at a specific place in the address space (specified by `addr` argument; else `addr` is ignored).
    - Many other Linux (and other specific) specific ones.
mmap’s parameters (3)

• prot:
  • Either:
    • PROT_NONE: pages may not be accessed
  • Or a bitwise combination of:
    • PROT_EXEC: pages may be executed
    • PROT_READ: pages may be read
    • PROT_WRITE: pages may be written
  • Must be consistent with open mode of fd.
Other Calls

- `int munmap(void *addr, size_t length);`
  - Delete the mapping starting at `addr`

- `int msync (void *addr, size_t length, int flags);`
  - Flushes changes made to the in-memory copy of the file to be reflected back to the disk (persistent store).
```
~ [4] cd
l10 [7]
```
Why \texttt{mmap}? 

- You probably observed that \texttt{mmap} was quite fast, even faster than standard IO. Why?
  
  - Fewer system calls: you make only one system call to map the file; then the rest of the processing that the system has to do is a side-effect of simply touching memory.
  
  - Fewer copies: both standard IO and read/write copy the data out of the operating system into a user buffer. \texttt{mmap} brings the data into memory and lets your application access that data directly (in the read-only case).
Why not `mmap`?

- Why do we ever use read/write if `mmap` is so fast?
  - Can’t really grow files easily using `mmap`, so it’s not create for creating a new files.
  - Although `msync` lets you force data to persistent storage, the application has no control over when data may be flushed back to persistent storage, so it is difficult to maintain on-disk data consistency using `mmap` with updates.
  - Requires block-alignment – not great for small files.
  - Doesn’t work on all file types (just regular files).