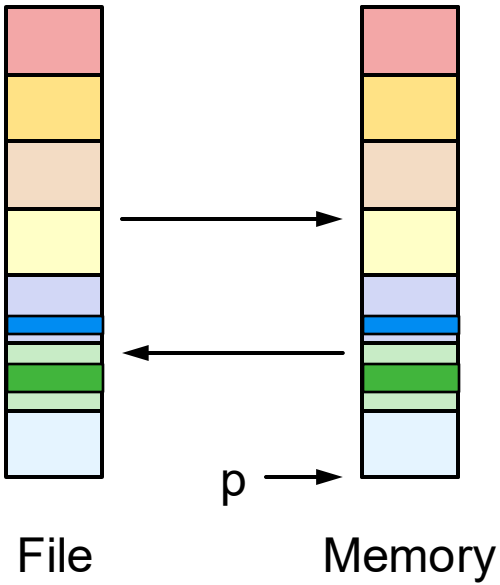
CS 111 Assignment 5:
Memory-Mapped Encrypted Files

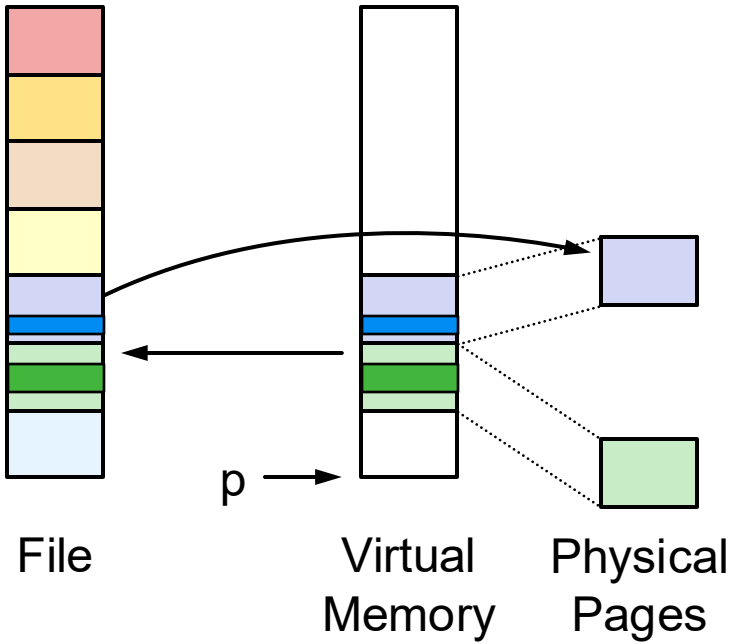
Accessing a File in Memory



1. **Get file size**
2. **Allocate memory:**
   `char *p = new char[size];`
3. **Read file into memory**
4. **Read and write file in memory:**
   `p[10000] = 'x';`
5. **When finished, write modified data back to file**

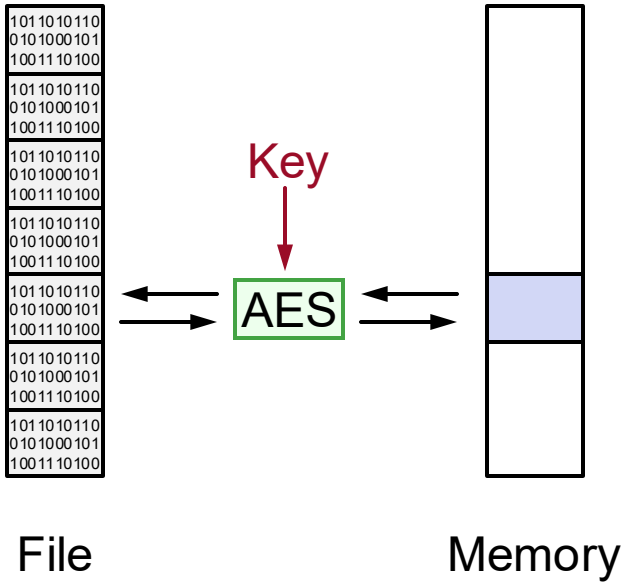
If file is large and most of it is never touched, this is inefficient

A Better Way: Mapped File



1. **Get file size**
2. **Allocate virtual memory region (no physical pages yet!)**
3. **Pretend file is in memory and start reading and writing**
4. **On page fault:**
   - Allocate physical page
   - Read part of file into page
   - Associate page with region (“map” it)
5. **Load other pages as accessed**
6. **When finished, only write back dirty pages**

Extra Twist: Encryption



1. **Data is stored on disk in encrypted form (cypher text: unrecognizable)**
2. **Decrypt when reading in (clear text in memory)**
3. **Encrypt again when writing out**
4. **Encryption/decryption controlled by a key**
   - Must keep key secret
   - Same key for reading and writing

MCryptFile

```
MCryptFile(Key key, std::string path)
```

```
char *map(size_t min_size = 0)
```

Sets up virtual memory region but doesn't allocate physical pages or read from file; pages will be read in as memory is accessed

```
void flush()
```

Write all dirty pages back to the file

```
void unmap()
```

Write back dirty pages and tear down the virtual memory region

```
char *map_base()
```

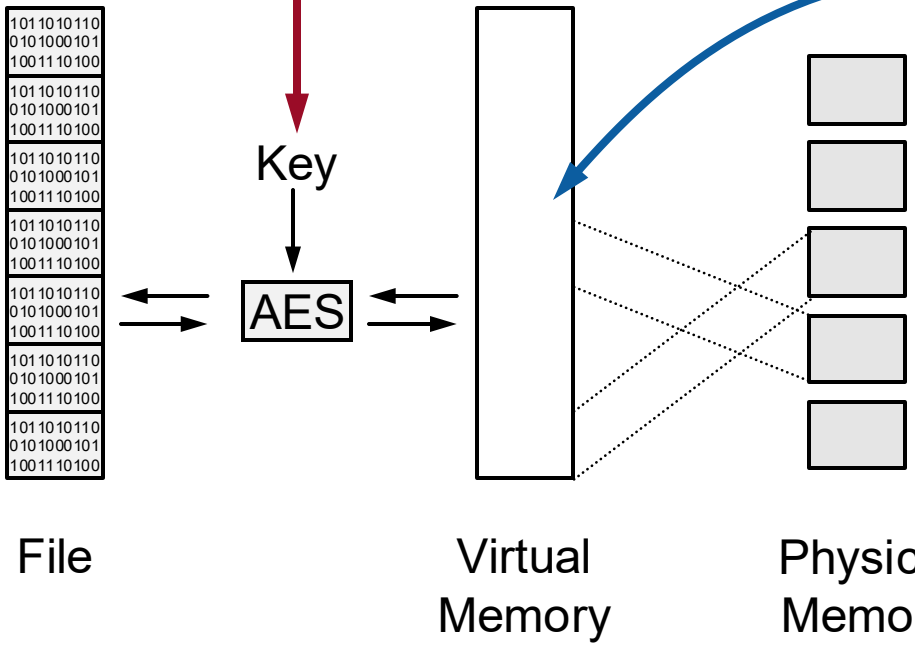
```
std::size_t map_size()
```

```
static void set_memory_size(size_t npages)
```

Set size of physical memory pool

Supporting Classes

`CryptFile`: reads/writes encrypted files



`VMRegion`:

- Allocates virtual memory
- Catch page faults
- Map/unmap physical pages

`PhysMem`:

- Manages pool of physical pages
- Allocates/frees pages

CryptFile

```
CryptFile(Key key, std::string path)

int aligned_pread(void *dst, size_t len, size_t offset)

int aligned_pwrite(const void *src, size_t len, size_t offset)

size_t file_size()
```

MCryptFile: Subclass of CryptFile

```
class MCryptFile : public CryptFile { ... }
```

Declare parent class

```
MCryptFile::MCryptFile(Key key, std::string path)
    : CryptFile(key, path)
{ ... }
```

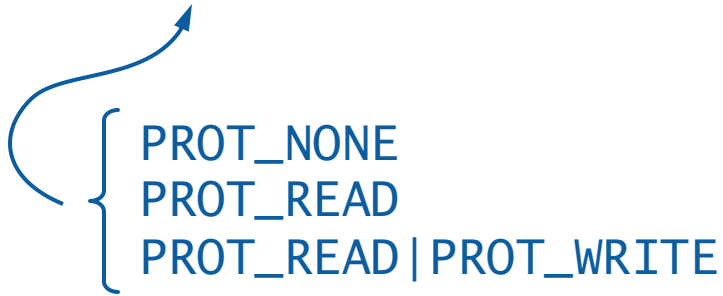
Constructor delegation

```
aligned_pread(...)
```

Methods of CryptFile are methods of MCryptFile

VMRegion

```
VMRegion(size_t nbytes, std::function<void(char *)> handler)
VPage get_base()
std::size_t get_size()
static void map(VPage va, PPage pa, Prot prot)
static void unmap(VPage va)
size_t page_size
```



- VPage is a char *
- How to compute a VPage for page 2 of a VMRegion?

PhysMem

```
PhysMem(size_t npages)
PPage page_alloc()
void page_free(PPage p)
size_t npages()
size_t nfree()
PPage pool_base()
```

- PPage is a char *
- Can use PPage to read and write the contents of the page
- Aliasing: same information accessible at two different addresses

vm_demo

```
int main(int argc, char **argv)
{
    VPage vp = nullptr;
    PPage pp = nullptr;
    VMRegion region(5*page_size, [&vp, &pp, &region](char *fault_addr) {
        std::cout << "Page fault at 0x" << std::hex
                << std::uintptr_t(fault_addr) << std::endl;
        region.map(vp, pp, PROT_READ|PROT_WRITE);
    });
    vp = region.get_base() + 2*page_size;
    std::cout << "VMRegion is at 0x" << std::hex
            << std::uintptr_t(region.get_base()) << ", VPage 2 is at 0x"
            << std::hex << std::uintptr_t(vp) << std::endl;

    PhysMem mem(10);
    pp = mem.page_alloc();
    std::cout << "PPage is at 0x" << std::hex << std::uintptr_t(pp) << std::endl;

    strcpy(pp, "data written to PPage");
    std::cout << "Initial contents of VPage: " << vp << std::endl;
    strcpy(vp, "new info written to VPage");
    std::cout << "Contents read from PPage: " << pp << std::endl;
    region.unmap(vp);
    mem.page_free(pp);
}
```

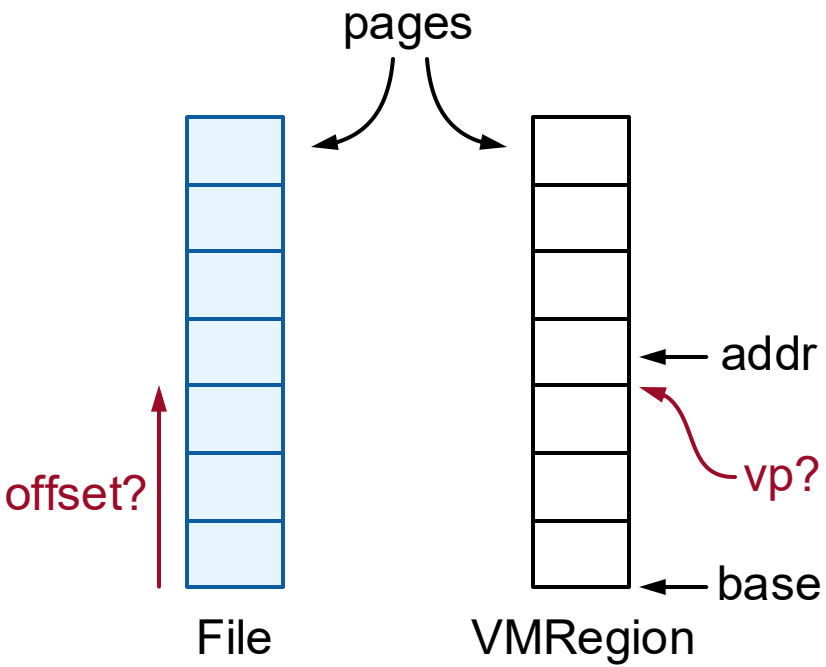
Dirty Pages

- **flush method: should only write back pages that have been modified**
- **How to tell which pages have been modified?**
  - Real page tables usually have “dirty” bits, but this assignment has no way to access them
- **If we can detect the first time a page is written, can record a dirty bit in software**

Supplemental Page Map

- **The `VMRegion` `map` and `unmap` methods modify the hardware page maps.**
- **But, no way to read info back from the page maps:**
  - Can't ask whether a particular page is mapped
  - Can't find out current permissions on page
- **You will need information like this:**
  - What info is needed to `flush` and `unmap` a `VMRegion`?
- **Solution: supplemental page map**
  - Information kept by `MCryptFile` for each `VPage` that has been mapped
  - Should be able to look up info with `VPage`
  - Best to have one supplemental page map, shared by all VMRegions
- **Don't implement supplemental page map right away**
  - Wait until you need it (Milestone 3: when you write `unmap`)
  - Add more info as needed

More Address Manipulation



- **Suppose page fault occurs at addr**
- **Where is the beginning of the VPage (vp)?**
- **What is the corresponding offset in the file?**