CSCI 4061: Virtual Memory

Chris Kauffman

Last Updated:
Thu Dec 7 12:52:03 CST 2017
Logistics: End Game

<table>
<thead>
<tr>
<th>Date</th>
<th>Lecture</th>
<th>Outside</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mon 12/04</td>
<td>Lab 13: Sockets</td>
<td></td>
</tr>
<tr>
<td>Tue 12/05</td>
<td>Sockets</td>
<td></td>
</tr>
<tr>
<td>Thu 12/07</td>
<td>Virtual Memory</td>
<td></td>
</tr>
<tr>
<td>Mon 12/11</td>
<td>Lab 14: Review</td>
<td></td>
</tr>
<tr>
<td>Tue 12/12</td>
<td>Review</td>
<td></td>
</tr>
<tr>
<td>Wed 12/13</td>
<td>Classes End</td>
<td></td>
</tr>
<tr>
<td>Wed 12/20</td>
<td>10:30am-12:30pm</td>
<td>Final Exam</td>
</tr>
</tbody>
</table>

Reading

- Stevens/Rago: Ch 16 Sockets
- mmap(): Linux System Programming, 2nd Edition By: Robert Love (library site link)

Goals: Finish Sockets

Lab13: Client Sockets

How did it go?

Project 2

Updates and Questions
Addresses are a Lie

- Operating system uses tables and hardware to translate every program address.
- Processes know virtual addresses which are translated via the memory subsystem to physical addresses in RAM and on disk.
- Contiguous virtual addresses may be spread all over physical memory.

Source: WikiP Virtual Memory
Address Translation

- OS maintains tables to translate virtual to physical addresses
- This needs to be FAST so usually involves hardware: Memory Manager Unit (MMU) and Translation Lookaside Buffer (TLB)
- Address translation is NOT CONSTANT $O(1)$, has an impact on performance of real algorithms

Source: John T. Bell Operating Systems Course Notes

*See: On a Model of Virtual Address Translation (2015)
Pages and Mapping

- Memory is segmented into hunks called **pages**, 4Kb is common (use `pagesize.c` to see your system’s page size)
- OS maintains tables of which pages of memory exist in RAM, which are on disk
- OS maintains tables per process that translate process virtual addresses to physical pages
- **Shared Memory** can be arranged by mapping virtual addresses for two processes to the same memory page

<table>
<thead>
<tr>
<th>Proc</th>
<th>VirtPage</th>
<th>PhysPage</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>0</td>
<td>1046</td>
<td>Shared</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>900</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2032</td>
<td></td>
</tr>
<tr>
<td>456</td>
<td>0</td>
<td>800</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>400</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1046</td>
<td>Shared</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3040</td>
<td></td>
</tr>
</tbody>
</table>
Exercise: Process Memory Image and Libraries

How many programs on the system need to use malloc() and printf()?

Where is the code for malloc() or printf() in the process memory?
Shared Libraries: *.so Files

- Code for libraries can be shared
- libc.so: shared library with malloc(), printf() etc in it
- OS puts into one page, maps all linked procs to it

Source: John T. Bell Operating Systems Course Notes
Exercise: Recall `fork()`

- What does `fork()` do?
- What does the result of a `fork()` look like?
- What *seems* to need to happen for this to work
Fork and Shared Pages

- `fork()`'ing a process creates a nearly identical copy of a process
- Might need to copy all memory form parent to child pages
- Can save a lot of time if memory pages of child process are shared with parent - no copying needed (initially)
- What’s the major danger here?

Source: John T. Bell Operating Systems Course Notes
Fork, Shared Pages, Copy on Write (COW Pages)

- If neither process writes to the page, sharing doesn’t matter
- If either process writes, OS will make a copy and remap addresses to copy so it is exclusive
- Fast if hardware Memory Management Unit and OS know what they are doing (Linux + Parallel Python/R + Big Data)

Source: John T. Bell Operating Systems Course Notes
Shared Memory

Most Unix Systems provide System V and POSIX means for a program to explicitly create shared memory.

// SYSTEM V SHARED MEMORY
int shmget(key_t key, size_t size, int shmflg);
// create/acquire a segment of shared memory associated with given key
// and size, returns id associated with segment

void *shmat(int shmid, const void *shmaddr, int shmflg);
// attach to shared memory with given id, return address of shared
// memory, may specify preferred address or NULL

// POSIX SHARED MEMORY
int shm_open(const char *name, int oflag, mode_t mode);
// get an id (file descriptor) for segment of shared memory, similar
// to open() system call but memory only

void *mmap(void *addr, size_t len, int prot, int flags,
           int fd, off_t off);
// map given file descriptor to a memory address. Reads/writes
// associated with address are reflected into the contents of the file
// descriptor potentially resulting in reads/writes to backing files.
mmap(): Mapping Addresses is Ammazing

- ptr = mmap(NULL, size,...,fd,0) arranges backing entity of fd to be mapped to be mapped to ptr
- fd might be shared memory created with shm_open()
- fd might be a file opened with open()...
  - Wait, what?

```c
int fd = open("gettysburg.txt", O_RDONLY); // open file to get file descriptor
char *file_chars = mmap(NULL, size, PROT_READ, MAP_SHARED, fd, 0); // pointer to file contents call mmap with given size and file descriptor read only, potentially share, offset 0
printf("%c",file_chars[0]); // print 0th char
printf("%c",file_chars[5]); // print 5th char
```
Exercise: Examine `mmap-demo.c`

- Determine what it does
- Are there any limits to the information that is produced by the program
- How might one modify the program to accommodate arbitrarily sized files?
- Answer in `mmap-print-file.c`
mmap() allows file reads/writes without read()/write()

- Memory mapped files are not just for reading
- With appropriate options, writing is also possible
  ```c
  char *file_chars =
      mmap(NULL, size, PROT_READ | PROT_WRITE,
           MAP_SHARED, fd, 0);
  ```
- Amazing stuff: assign to memory, OS reflects change into the file
- Example: mmap-tr.c to transform one character to another
mmap() Flexibility is complete

- mmap() just gives a pointer: can assert that it points to binary data like structs as well
- See example: mmap-specific-stock.c for an example of this
- Multiple processes can map files to shared memory to communicate, read/write same files, cooperate
- IPC control mechanisms such as semaphores, message queues, mutexes should be used to control shared files to prevent read/write conflicts
mmap() Comparisons

Benefits

- Avoid `read()` into memory, change, `write()` cycle
- Saves memory and time
- Many Linux mechanisms backed by `mmap()` like shared memory

Drawbacks

- Always maps pages of memory \( \sim 4096\text{b} \) (4K)
- For small maps, lots of wasted space
- No bounds checking, just like everything else in C