Other Uses of Paging Support

COMP 321

Dave Johnson

1

Some Advantages of Paging (So Far)

Remapping

- Can map ***any*** physical page to appear ***anywhere*** in process's address space
- Can make ***any*** physical pages ***appear*** to be contiguous in virtual memory
- Thus, no ***external*** fragmentation
- Memory allocation is ***easy***, just allocate ***any*** available physical page
- Can easily grow a process's virtual address space
- Can make, e.g., stack and heap ***very, very*** far apart at almost no cost

Demand Paging

- Can make process's virtual address space appear very large
- Can run programs (much) larger than physical memory

2

Other Uses of Paging Support

Using same hardware and building on same kernel software

Faster, more efficient operation of the existing system

- Reducing consumption of physical memory
- Reducing CPU time to effectively implement some operations
- Reducing disk storage and disk I/O in implementation of some operations

Provide new facilities for user processes to make use of

- New memory management operations
- More flexible, more efficient memory management
- New memory management functionality

Loading Text Pages on execve()

The old way:

- The process's page table is built, include PTEs for all pages of text
  - Allocate a new physical page for each text page
  - For each page, PTE = valid, protection = read/execute
- Read all of the text into this new area of virtual address space
- And, finally, return from execve() to run the new program

Problems:

- Requires physical memory, even for parts of text that never get executed
- Requires disk read for all of that text, too
- Slows the startup of the new program

Faster, More Efficient Loading of Text on execve()

Load each page of text <u>on demand</u> from the program file

- The process's page table is built, include PTEs for all pages of text
  - Do not allocate any physical pages for any of the text
  - For each page, valid = 0, protection = read/execute
  - Kernel must remember ***why*** it set valid = 0
  - Kernel must remember the executable file and where each text page is
  - Return from execve() and start the new program running

Each individual text page gets loaded on demand in response to page fault

- Allocate a new physical page, PTE valid = 1, read in ***only*** that one text page
- If text page is chosen as victim, no need to write out the page, and it will be reloaded later on demand the same way when (if) it is next accessed again

What if the Text File Is Modified While Running?

Need to protect against the text file changing while we're running it

- The old way, we no longer need the original text file, since we've read it all into physical memory already
- The new way, the program could run some pages of the old version of the text, some pages of the new version of the text
  - Almost certainly not going to work correctly (probably crashes)

Solution

- The kernel keeps track of which program files are currently running
- If new open() for O_WRONLY or O_RDWR, or O_TRUNC, return ETXTBSY
- If new execve() while some file descriptor open for O_WRONLY or O_RDWR, return ETXTBSY

Sharing Text Pages Between Processes

Since a text page is read-only, can easily share it between all processes currently running that same program

- Only one copy of that page is needed in physical memory, rather than (up to) one copy per process running it
  - Example: Consider the number of processes running the shell!
- This is an "easy" extension
  - The kernel already needs to keep track of which program files are currently running – to prevent modification to one while it is running and demand loading from it
  - On an execve() for a program already being run by one or more other process – the kernel sets up calling process's PTEs to use the same PFNs

Supporting Procedure Libraries

The old way:

- The library is an "archive" of individual ".o" object files
- The linker embeds each of those needed object files statically in the executable program (e.g., most programs use printf)

Problems:

- Every program on disk has embedded its own separate copy of many of those object files
  - Wasting disk space
- Every program in memory has a separate copy, too
  - Wasting physical memory

Improvement: Shared Libraries

Make the library shared in memory among all processes using it

- Link the library now instead as a ***single combined*** object file
  - Example on CLEAR: libc.so.6 ≈ 2MB with 2344 external text symbols
- Map the entire library into a process's virtual address space
  - The kernel keeps a list of objects mapped into memory (an extension of the list of programs running for demand loading and sharing text)
  - If that library is already mapped into memory, new process's PTEs use the same PFNs, so the pages are shared in physical memory
  - The virtual address of the library may (but doesn't have to be) the same in each process, since the library is compiled and linked as "position independent code" (PIC)

Setting Up bss Pages on execve()

The old way:

- The process's page table is built, including PTEs for all pages of bss
  - Allocate a new physical page for each text page
  - For each page, PTE = valid, protection = read/write
- Initialize all of this new area to be full of 0's (e.g., memset())
- And, finally, return from execve() to run the new program

Problems:

- Requires physical memory, even for parts of bss that never get used
- Requires CPU time to zero it all out, too
- Slows the startup of the new program

Faster, More Efficient Setting up bss on execve()

Allocate and zero out each page of bss <u>on demand</u>

- The process's page table is built, including PTEs for all pages of bss
  - Do not allocate any physical pages for any of the bss
  - For each bss page, PTE valid = 0, protection = read/write
  - Kernel must remember ***why*** it set valid = 0
  - Return from execve() and start the new program running

Each individual bss page gets handled on demand in response to page fault

- Allocate a new physical page, PTE valid = 1, zero out ***only*** that one bss page
- If such a bss page is chosen as victim, it gets paged out to the ***paging file***, the same as the old way, as if it had not been set up on demand
- Technique is called "demand zero" or "zero-fill on demand"

Copying Memory from Parent to Child on fork()

The old way:

- The ***child***'s page table is built, including PTEs for ***all*** pages as in the ***parent***
  - Allocate a new physical page for each ***valid*** page
  - For each page, child PTE valid = 1, protection = same as in parent
  - Copy that page contents from parent's address space into child's
- And, finally, return from fork() to allow child to run as a new process

Problems:

- Requires physical memory, even for pages never used by the child
- Requires CPU time for copying all of that, too
- Slows the startup of the new process
- Extreme (but common) example: fork() "immediately" followed by execve()

Faster, More Efficient fork()

Allocate physical memory for and copy each page only <u>on demand</u>

- The ***child***'s page table is built, including PTEs for ***all*** pages as in the ***parent***
  - Do not allocate any new physical pages for any of the child's pages
  - The child's PTEs ***share*** physical pages with the parent (same PFNs)
  - Return from fork() to allow child to run as a new process

The technique is called "copy-on-write" (COW)

- As long as those shared pages aren't modified, they can be shared
- OK for text pages, but what about data pages, which might get modified?
  - Kernel turns off PTE write protection bit when setting up copy-on-write
  - Attempt to modify page causes hardware exception, and the kernel makes a copy of just that single page and reenables write in PTE protection

Copy-on-Write Details

For each copy-on-write shared physical page

- The kernel keeps track of the number of times that page is currently shared
  - After a fork, each page is now shared two times (parent and child)
  - But if either process forks again, the count is now 3, and so on
- The kernel disables write access to the page by turning off bit in all PTEs
  - Each sharing process has its own page table and thus its own PTE for it
- If some write is attempted, that will cause a hardware exception
  - If the page is still being shared (count is > 1), the kernel allocates a new physical page, copies the shared page into it, and decrements count
  - The kernel reenables write access in PTE for that page in that process
  - The kernel returns from exception and hardware re-executes instruction

The mmap() Kernel Call

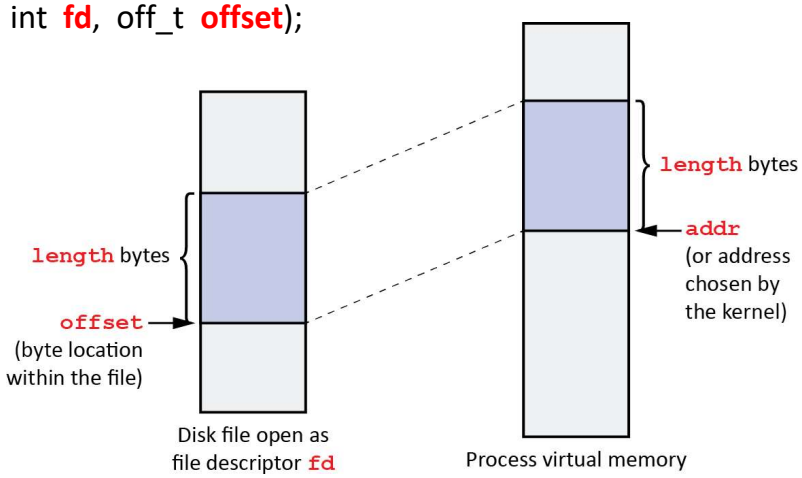
```
void *mmap(void addr[.length], size_t length, int prot, int flags,
           int fd, off_t offset);
```

Map (part of) a file (or something else) into your virtual address space

- "fd" must be open to the file (or whatever) you want to map in from
- "addr" and "offset" must be multiple of PAGESIZE, but "length" need not be
- All pages covered by any part of [addr, addr+length) are mapped
  - if addr == NULL, the kernel chooses the virtual address
- "prot" should be PROT_NONE (no access) or the logical "or" of
  - PROT_READ – allow read (e.g., LOAD) accesses from the page
  - PROT_WRITE – allow write (e.g., STORE) accesses to the page
  - PROT_EXEC – allow CPU execution of instructions from the page
- mmap() returns the virtual address at which the memory was mapped

The mmap() Kernel Call

void *mmap(void **addr**, size_t **length**, int prot,
int flags, int **fd**, off_t **offset**);



The mmap() Kernel Call

The "flags" argument can specify many different things as the "or" of

- The sharing of the mapping with other processes, as either of
  - MAP_SHARED – Modifications to the memory ***are*** visible to other processes mapping this file and ***are*** eventually reflected in the file
  - MAP_PRIVATE – Modifications to the memory ***are not*** visible to other processes mapping to this file, and the file contents ***is not*** changed
- What to map to (the default is to map to a file)
  - MAP_ANONYMOUS (or MAP_ANON) – instead of a file, map to newly created "anonymous" memory (i.e., map to "nothing")
- Restrictions on the virtual address to map to
  - MAP_FIXED – "addr" is normally treated as a "hint" by the kernel, but instead, the mapping must be exactly at virtual address addr

A Simple (Sort of Silly) mmap() Example Program

```
int main(void)
{
    char  *addr1,  *addr2;
    int  fd1,  fd2;
    struct stat  sb;
    size_t   i,   len;

    fd1 = open("input.txt", O_RDONLY);
    fd2 = creat("output.txt", 0666);

    fstat(fd1, &sb);
    len = sb.st_size;

    addr1 = mmap(NULL,
        len,  PROT_READ,
        MAP_PRIVATE,  fd1,  0);
    addr2 = malloc(len);

    for (i = 0; i < len; i++)
        addr2[i] = toupper(addr1[i]);

    write(fd2, addr2, len);

    exit(0);
}
```

The munmap() Kernel Call

```
int  munmap(void addr[.length],  size_t  length);
```

Remove a mapping from your virtual address space

- Make virtual memory starting at "addr" for length of "length" invalid
  - Any future access will cause a SIGSEGV
- "addr" must be a multiple of PAGESIZE, but "length" need not be
- All pages covered by any part of [addr, addr+length) are unmapped

The mprotect() Kernel Call

```
int  mprotect(void  addr[.len],  size_t  len,  int  prot);
```

Change the protection on a mapping in your virtual address space

- Change page protection starting at "addr" for length of "length" invalid
- "addr" must be a multiple of PAGESIZE, but "length" need not be
- All pages covered in any part by [addr, addr+length) are changed
- "prot" should be PROT_NONE (no access) or the logical "or" of
  - PROT_READ – allow read (e.g., LOAD) accesses from the page
  - PROT_WRITE – allow write (e.g., STORE) accesses to the page
  - PROT_EXEC – allow CPU execution of instructions from the page

The msync() Kernel Call

```
int msync(void addr[.length], size_t length, int flags);
```

Synchronize the file with the memory mapping

- Make sure the disk copy is up-to-date with changes in memory copy
- "addr" must be a multiple of PAGESIZE, but "length" need not be
- All pages covered in any part by [addr, addr+length) are updated
- "flags" should be either
  - MS_ASYNC – ***asynchronous*** update, returns immediately
  - MS_SYNC – ***synchronous*** update, ***does no return until updated on disk***
- Happens implicitly when munmap() is used on these pages
  - And thus happens implicitly on _exit()

The madvise() Kernel Call

```
int madvise(void addr[.length], size_t length, int advice);
```

Allows the process to advise the kernel on virtual memory page accesses

- Use of madvise() is never required
- It is optional and just gives "advice" to the kernel
  - "The great thing about advice is you don't have to take it"
  - But if used carefully, madvise() can improve performance for your process and/or for the system as a whole
- "addr" must be a multiple of PAGESIZE, but "length" need not be
- The advise covers all pages covered in any part by [addr, addr+length)

The madvise() Kernel Call

There are <u>many</u> (mostly non-standard) options for "advice", but mainly

- MADV_NORMAL – no special treatment (the default)
- MADV_RANDOM – expect page references to be in "random" order
  - Page read-ahead by the kernel may not be helpful
- MADV_SEQUENTIAL – expect page references to be in sequential order
  - Page read-ahead by the kernel may be particularly helpful
  - And/or consider preceding page on a page fault to be a good victim
- MADV_WILLNEED – expect references in the near future
  - It may be a good idea for the kernel to pre-page in some pages
  - And/or make these pages less likely to be chosen as a victim
- MADV_DONTNEED – do not expect references in the near future
  - These pages may be good to use as a victim on page replacement