Memory Management Part 3
Unix and Virtual Memory: The *fork/exec* Problem

- Naive implementation:
  - *fork* actually makes a copy of the parent’s address space for the child
  - child executes a few instructions (setting up file descriptors, etc.)
  - child calls *exec*
  - result: a lot of time wasted copying the address space, though very little of the copy is actually used
Quiz 1

How many pages of virtual memory must be copied from the parent to the child in the following code?

```c
if (fork() == 0) {
    close(0);
    dup(open("input_file", O_RDONLY));
    execv("newprog", 0);
}
```

a) 0  
b) 1  
c) lots
vfork

- Don’t make a copy of the address space for the child; instead, give the address space to the child
  - the parent is suspended until the child returns it
- The child executes a few instructions, then does an exec
  - as part of the exec, the address space is handed back to the parent

- Advantages
  - very efficient

- Disadvantages
  - works only if child does an exec
  - child shouldn’t do anything to the address space
Quiz 2

How many pages of virtual memory must be copied from the parent to the child in the following code?

```c
volatile int A = 6;
if (fork() == 0) {
    A = 7;
    exit(0);
}
```

a) 0  
b) 1  
c) lots
Lazy Evaluation

• Always put things off as long as possible

• If you wait long enough, you might not have to do them
A Better *fork*

- Parent and child share the pages comprising their address spaces
  - if either party attempts to modify a page, the modifying process gets a copy of just that page

- Advantages
  - semantically equivalent to the original *fork*
  - usually faster than the original *fork*

- Disadvantages
  - slower than *vfork*
Copy on Write (1)
Copy on Write (2)

Data = 17;
The *mmap* System Call

- L1 Page Table
- L2 Page Tables
- File Pages
- L2 Page Tables
- L1 Page Table
Share-Mapped Files

L1 Page Table → L2 Page Tables → File Pages → L2 Page Tables → L1 Page Table

Data = 17;
Private-Mapped Files

Data = 17;
A Private-Mapped File Changes

L1 Page Table
L2 Page Tables
File Pages
L2 Page Tables
L1 Page Table

Data = 17;
OtherData = 6;
Virtual Copy

- Local RPC
  - “copy” arguments from one process to another
    - assume arguments are page-aligned and page-sized
    - map pages into both caller and callee, copy-on-write
Share Mapping (1)

Process A has share mapped the file object.

File object
Share Mapping (2)

Process A has share-mapped the file object.

A forks, creating B.

Share-mapped file object
Private Mapping (1)

A modifies page x.

Process A

Shadow object

Private-mapped file object

x
y
z
A modifies page x.
**A forks, creating B.**
A modifies page z.
B modifies page y.
Private Mapping (3)

A modifies page x.
A forks, creating B.
A modifies page z.
B modifies page y.
B forks, creating C.
B modifies page x.
C modifies page z.
Share and Private Mapping

A virtual copies $x$, $y$, and $z$ into B.
B modifies $y$.
A modifies $x$. 
Quiz 3

Unix process X has private-mapped a file into its address space. Our system has one-byte pages and the file consists of four pages. The pages are mapped into locations 100 through 103. The initial values of these pages are all zeroes.

1) X stores a 1 into location 100
2) X forks, creating process Y
3) X stores a 1 into location 101
4) Y stores a 2 into location 102
5) Y forks, creating process Z
6) X stores 111 into location 100
7) Y stores 222 into location 103
8) Z sums the contents of locations 100, 101, and 102, and stores them into location 103

What value did Z store into 103?

Answer:

a) 0
b) 3
c) 4
d) 113
Fork Bomb!

```c
int main() {
    while (1) {
        if (fork() <= 0)
            exit(0);
    }
    return 0;
}
```

```c
int main() {
    while (1) {
        if (fork() > 0)
            exit(0);
    }
    return 0;
}
```
Private Mapping (Continued)

Process A

Process B

Process C

Process B exits
Process A exits

x y z

x y z

x y z
Backing Up Pages (1)

- Read-only mapping of a file (e.g. text)
  - pages come from the file, but, since they are never modified, they never need to be written back

- Read-write shared mapping of a file (e.g. via `mmap` system call)
  - pages come from the file, modified pages are written back to the file
Backing Up Pages (2)

- Read-write private mapping of a file (e.g. the data section as well as memory mapped private by the `mmap` system call)
  - pages come from the file, but modified pages, associated with shadow objects, must be backed up in swap space
- Anonymous memory (e.g. bss, stack, and shared memory)
  - pages are created as zero fill on demand; they must be backed up in swap space
Swap Space

- Space management possibilities
  - radical-conservative approach: pre-allocation
    - backing-store space is allocated when virtual memory is allocated
    - page outs always succeed
    - might need to have much more backing store than needed
  - radical-liberal approach: lazy evaluation
    - backing-store space is allocated only when needed
    - page outs could fail because of no space
    - can get by with minimal backing-store space
Space Allocation in Linux

• Total memory = primary + swap space
• System-wide parameter: `overcommit_memory`
  – three possibilities
    - maybe (default)
    - always
    - never
• `mmap` has `MAP_NORESERVE` flag
  – don’t worry about over-committing
Space Allocation in Windows

• Space reservation
  – allocation of virtual memory

• Space commitment
  – reservation of physical resources
    - paging space + physical memory

• `MapViewOfFile` (sort of like `mmap`)
  – no over-commitment

• Thread creation
  – creator specifies both reservation and commitment for stack pages