Memory Management Part 3
<table>
<thead>
<tr>
<th>Unix and Virtual Memory: The <em>fork/exec</em> Problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Naive implementation:</td>
</tr>
<tr>
<td>– fork actually makes a copy of the parent’s</td>
</tr>
<tr>
<td>address space for the child</td>
</tr>
<tr>
<td>– child executes a few instructions (setting</td>
</tr>
<tr>
<td>up file descriptors, etc.)</td>
</tr>
<tr>
<td>– child calls exec</td>
</tr>
<tr>
<td>– result: a lot of time wasted copying the</td>
</tr>
<tr>
<td>address space, though very little of the</td>
</tr>
<tr>
<td>copy is actually used</td>
</tr>
</tbody>
</table>

An efficient implementation of the *fork* system call was a challenge in the early implementations of Unix with virtual memory.
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) 2  
d) 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

The first approach toward an efficient `fork` was the variant known as `vfork`, which first appeared in late 1979 as part of the third Berkeley Software Distribution (known as 3 BSD). It was remarkably efficient, but was not a complete solution (though good enough for most applications).
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) 2  
d) lots

Assume that the assignment to \( A \) is not optimized out.
The efficient *fork* implementation is a (successful) example of an important OS design principle: *lazy evaluation.*
After the virtual-memory implementation matured, a better approach (in most respects) for a fork implementation became possible, using a technique known as copy on write (described in the next slides).
In the copy-on-write approach, processes share their pages, which are marked read-only.
If a thread in either process attempts to modify a page, the hardware traps it, generating a protection fault. The operating system realizes that an attempt is being made to modify a page marked copy on write, so it makes a copy of that page for the faulting process. The appropriate page table is modified to point to the new page. Note that we could apply copy on write to the L2 page tables as well, so as to avoid having to copy them.

Thus we are postponing copying the address space, in hopes that we might not have to do it.
The *mmap* system call maps a file into a process’s address space. All processes mapping the same file can share the pages of the file.
There are a couple options for how modifications to mmapped files are dealt with. The most straightforward is the *share* option in which changes to mmapped file pages modify the file and hence the changes are seen by the other processes who have share-mapped the file.
The other option is to *private-map* the file: changes made to mmapped file pages do not modify the file. Instead, when a page of a file is first modified via a private mapping, a copy of just that page is made for the modifying process, but this copy is not seen by other processes, nor does it appear in the file.

In the slide, the process on the left has private-mapped the file. Thus its changes to the mapped portion of the address space are made to a copy of the page being modified.
What if a private-mapped file is changed by a process that has share-mapped it? Somewhat surprisingly, the change is seen by the process that has private-mapped the file, as long as the change is not to a page that has been modified in the private-mapped view.

In the slide, while the process on the left has private-mapped the file, the one on the right has share-mapped it. Thus changes by the process on the right to pages that have not been modified by the process on the left are seen by both processes.
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
As an example of how to implement virtual copy and private-mapping operations efficiently, we look at the approach used by the Mach operating system and adopted by Apple in Mac OS X. Suppose process A has a private mapping of the file and has modified page x but neither page y nor page z. Since the file is private-mapped, a copy of x is created in what’s known as a shadow object. If it is necessary to page out x, it will be paged out to storage set up for this shadow object.
Process A forks, creating child process B. Shadow objects are created to contain the further modified pages of A as well as of B. B modifies page y and A modifies page z. If A accesses page y, it will obtain the copy in the original file, not the copy in the shadow object created for B. Similarly, if B accesses z, it will obtain the copy in the original file. If either process accesses x, they will both obtain the copy in the first shadow object.
Process B forks, producing child C. B modifies page x; C modifies page z.
Process A has share-mapped a file, then performs a virtual copy of the mapped portion of its address space into Process B. Thus B now has a private mapping of the file object. If B modifies page y, then, as before, it first makes a copy of y in its shadow object. If A modifies page x, the page is first copied to B’s shadow object and then is modified in the file. (Note: since neither Linux nor Weenix supports the virtual-copy operation, this scenario does not occur in them. However, it does occur in systems that support virtual-copy operations efficiently)
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
These programs should run forever. For them to do so, some trimming must be done of the list of shadow objects.
After processes B and A exit, which shadow objects can be eliminated?
Our next topic is the backing store, i.e., the storage where pages are kept when not in primary memory. As shown in the slide, such storage might be managed by the file system, it might be some otherwise unstructured portion of a disk, or perhaps something else.
This slide and the next list the various possibilities of the backing-store location for Unix.
Back 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
Equally important is when and how backing-store space is allocated.
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

With the “maybe” approach, various heuristics are used to come up with a good guess as to a reasonable level of over-commitment, much like how airlines over-commit seats on planes. And, as is the case with airlines, things don’t always work out. So it’s possible that a process may have to be killed if it cannot continue without additional memory (either primary storage (RAM) or swap space). Note that such a process isn’t necessarily the “culprit”. A large process may have used up most of the available resources. The kernel is trying to write out a page belonging to a smaller process. If there’s no room for it in swap space (on disk), then it’s the smaller process that’s terminated.
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