1 RAID

I think what's key here is that if you're updating a disk, say disk 'B' out of n disks, XOR'ing the parity block by 'B's original value will give the parity of all disks except 'B'. This'll allow you to change the value of 'B', then recompute the parity without reading the values of the blocks from the other disks.

For example, let's say we have 4 disks, with disk 'D' the parity disk:

```
(1)  A = 1000    B = 1101    C = 0111    D = 0010    /* initial state */
```

Now let's say we want to update disk B to hold block 0001. First, XOR the parity block on disk 'D' with 'B's original value:

```
(2)  A = 1000    B = 1101    C = 0111    D = 1111
```

then change 'B's value and recompute the new parity:

```
(3)  A = 1000    B = 0001    C = 0111    D = 1110
```

Given damage to any block now, XOR'ing the non-damaged blocks with the parity block will be enough to uniquely determine the value of the damaged block. Just to test that the new parity in (3) actually works, let's assume disk 'C' is damaged:

```
A:      1000
B:      0001
------------
XOR     1001

XOR:    1001
D:      1110
------------
C:      0111            /* the correct value */
```

2 IA-64

The TLB holds a number of page frame numbers: those that've recently been translated from a virtual address to a real address. If a thread is accessing a region sequentially, one page at a time, there won't be overlap in the pages it's translated. This means there'll be a miss on each access to a new page.

2.1

Since a page is 8k in size with 8 byte entries there're 1024 entries per page table. When trying to make a translation, the TLB will first be checked. If there's a TLB miss and no page tables currently loaded into real memory, it'll have to trap to the OS to bring a new page into memory. Using the above specs, I believe this will happen every 1024 accesses.

2.2

To bring a new page into memory will take 3 memory accesses. Since pages are 8k in size, and you want to reference 1G, you'll need to make:

$$2^{30}/2^{13} * 3 = 393216$$

assuming there's no optimization and you make 3 memory accesses for each page your bring into memory.

3 Mmap

The big problem that might occur is let's say the real memory mmap() maps the file into isn't the same memory used as the buffer cache. If this happens, then writes and reads made via system calls will be accessing different memory from calls using the memory designated via mmap().

What we'd want to happen is for a write() system call to show up in the region of memory mapped via mmap() and for a read() system call to show any changes coming via the mmap() region. If the system calls are using a buffer and mmap() is using a different area of real memory, however, we won't get this kind of behavior.

3.1

If two processes are accessing and modifying the same file via mmap(), then they are each referencing the same area of real memory the file was mapped into. Thus, since they are both dealing with the same memory, changes in one process will be correctly mirrored in the other.

3.2

A quick fix for this would be to have the real memory mmap() maps the file into be the same as the buffer cache used by the read() and write() system calls. Now, since they both reference the same region of real memory, any changes made by system calls will be visible in the mmap()'ed region and vice-versa.

4 Shadow Updates

Let's say you fork a process and privately map its address space. To optimize, the address space of the parent isn't automatically copied into the child's address space. Fork() system calls are usually followed quickly by exec() calls, which generate a new image from disk. It, therefore, doesn't make sense to completely copy over the parent's address space into the child's if the child is just going to shortly discard this address space and start fresh.

What's done, then, is privately mapped processes share the address space of their parent, and only if they end up editing objects are those objects copied into shadow objects, which represent changes between processes. This is a form of lazy evaluation.

You could get an arbitrarily long list of shadow objects, though, if you have processes continually fork() themselves with child processes that are privately mapping the address space, and if these child processes then edit the address space but never call exec() (or another function that uses a new image, fresh from disk).

Concretely, let's say process B forks from process A (privately mapping it's address space) then starts editing objects. Shadow objects will need to be created in B to keep track of these changes. Now, B never calls exec(), but rather forks() another process, C, which is also privately mapping the address space. C, like B, starts changing objects and thus needs shadow objects of its own. This could, hypothetically, go on forever, thus creating a long chain of shadow objects.

4.1

For a process to find the current object associated with a page, it'll first check its newest shadow objects, then the shadow objects of its parent, then the shadow objects of its grandparent, and so on until it gets to the original object.

I don't think this list of shadow objects has to be arbitrarily long, however. Looking in the book at figure 7.25, for example, if process C's shadow object modified page 'y', you could effectively eliminate B's shadow object for 'y', that was copied into C, from the chain of shadow objects for future processes forked from C. Now, process C's shadow object for page 'y' would link to the shadow object for page 'x', eliminating a redundant part of the chain.

This can be done since once a child edits a page that is in a shadow object from one of its ancestors, the ancestor's shadow object becomes obsolete for the child and any of the child's children. You can thus eliminate it from the chain of shadow objects copied into future forked processes.

Additionally, instead of keeping around chains of shadow objects, simply consolidate chains into single, larger shadow objects that can be passed to future children. Taking example 7.25 from the book again, when C forks from B, instead of maintaining the chain of shadow objects [z] $\rightarrow$ [y] $\rightarrow$ [x], consolidate this into a single shadow object keeping track of the changes made to pages x, y, and z.

Lastly, when a process forks() a child process, and the child process privately maps the address space, the child gets a shadow object. Yet, if this child never actually updates any pages before it forks() itself, you don't need to chain together essentially unused shadow objects.