1 CS24 Assignment 8

The files for this assignment are provided in the archive **`cs24hw8.tgz`** on the course website. The archive contains a top-level directory **`cs24hw8`**, in which all other files are contained.

For the questions you need to answer, and the programs you need to write, put them into this **`cs24hw8`** directory. Then, when you have finished the assignment, you can re-archive this directory into a tarball and submit the resulting file:

> *(From the directory containing **`cs24hw8`**; replace **`username`** with your username.)*
> **`tar -czvf cs24hw8-username.tgz cs24hw8`**

Then, submit the resulting file through the course website.

2 Overview

This lab will give you the opportunity to experiment with a simple user-space virtual memory system. Of course, the phrase "*user-space* virtual memory system" should immediately indicate that this is only a toy; the distinction between user-space and kernel-space is, after all, enforced by the processor's MMU and the OS virtual memory system. Nonetheless, it is remarkable that one can implement a simple virtual memory system using only the facilities exposed to UNIX programs. Additionally, it allows us to explore some of the basic characteristics of virtual memory management in a relatively simple environment.

As a UNIX application, we are unable to configure the MMU directly, and we aren't allowed to register a page-fault handler. However, we can use a handful of other UNIX facilities to create a reasonable facsimile of virtual memory, with some simplifications. This will allow us to explore some of the basic concepts that underlie virtual memory systems.

3 UNIX Functionality

We are all familiar with segmentation faults – our programs receive a **`SIGSEGV`** (segmentation violation signal) when our program does something silly like dereference a **`NULL`** pointer, or attempt to write to program code, or any number of other things. As discussed in class, even these situations are caused by page faults and protection faults generated by the processor's MMU; the OS kernel looks at the process' virtual memory descriptors to determine if the fault was caused by an illegal access (e.g. to a region of memory that is unmapped), or if the fault was simply caused by a page needing to be loaded back into RAM (e.g. that page was swapped out to disk).

UNIX gives processes a number of powerful tools to control their own memory mapping, so that programs can use shared memory regions, memory map files, change the permissions of various memory regions, and so forth. The relevant functions are as follows:

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

> This function allows a process to add a region to its virtual address space. It is a very complex function because it can be used to do several different things.

- The **addr** value allows a program to specify the starting address of the memory region, or it can be 0 to allow the kernel to choose the start of the memory region. **This address should be on a hardware page boundary.** On IA32, this means the address should be a multiple of 4096.
- The **length** value specifies the length of the memory region. Again, this length should include a whole number of pages, so on IA32 it should also be a multiple of 4096.
- The **prot** value allows us to set the access permissions for the memory region. For example, we can set **prot** to **PROT_READ**, and only reads will be allowed against the memory region. Attempts to perform writes to the region will cause a segfault.
- The **flags** value specifies other details of the memory region. For example, we can specify **MAP_FIXED** to force **mmap()** to use **addr** as the starting address. We can also specify **MAP_ANONYMOUS** to tell **mmap()** to use the *anonymous file* to populate the memory region; that is, the region will be set to all zeros.
- The **mmap()** function can also be used to memory-map a file into the process' virtual address space; the last two arguments **fd** and **offset** can specify the file to map into the memory region, and the starting offset in the file to use. If **flags** includes **MAP_ANONYMOUS**, it's common to set **fd** to -1; if **fd** refers to a specific file then the flags really shouldn't include **MAP_ANONYMOUS**.

```
int mprotect(void *addr, size_t len, int prot)
```

This function allows a process to change the access permissions of the specified memory region. As before, **addr** and **len** should both conform to hardware page boundaries; on IA32, these values should be a multiple of 4096. The **prot** value can be set to any of the protection values supported by **mmap()**.

The return value simply indicates whether or not the call succeeded; 0 indicates success.

```
int munmap(void *addr, size_t length)
```

This function allows a process to remove a memory region from its virtual address space. The **addr** and **len** values should again conform to hardware page boundaries. It is legal for a program to map a large region of memory, and then unmap smaller portions of it.

3.1 Memory Access Permissions

As stated earlier, both **mmap()** and **mprotect()** allow us to specify access permissions for regions of memory in the process' virtual address space. The relevant permissions are as follows:

- **PROT_NONE** – Even though the memory region is valid, any access, whether a read or a write, will cause a segfault to be generated.
- **PROT_READ** – Only read access is allowed. Writes to the region will cause a segfault.
- **PROT_WRITE** – Write access is allowed. On many systems, **PROT_WRITE** implies **PROT_READ**, but it is most portable to specify **PROT_READ | PROT_WRITE** together when read-write permissions are desired.
- **PROT_EXEC** – The data in the page can be executed as code.

Using these permissions, it should be possible to build everything we need to create a simple virtual memory system in our program. Since violations of these permissions are reported via seg-faults, we will create a **SIGSEGV** handler that handles our virtual memory system operations.

The kernel provides UNIX programs with an advanced signal-handling mechanism, exposed via the **`sigaction()`** kernel function. With this function, a program can register a signal handler that receives very detailed information from the kernel. By using this mechanism, the kernel will inform us of what address caused the seg-fault to occur, and whether the fault was caused by a "mapping error" (the accessed address has no corresponding memory mapping in the process' address space) or an "access error" (the address has a corresponding mapping, but the permissions for the region were violated).

Of course, if we want to use seg-faults for our virtual memory system, we will have to be more clever about debugging our program…

4 Virtual Memory System Design Overview

Our virtual memory system will follow the pattern of all virtual memory systems, but it must work within the constraints of a user-mode UNIX application.

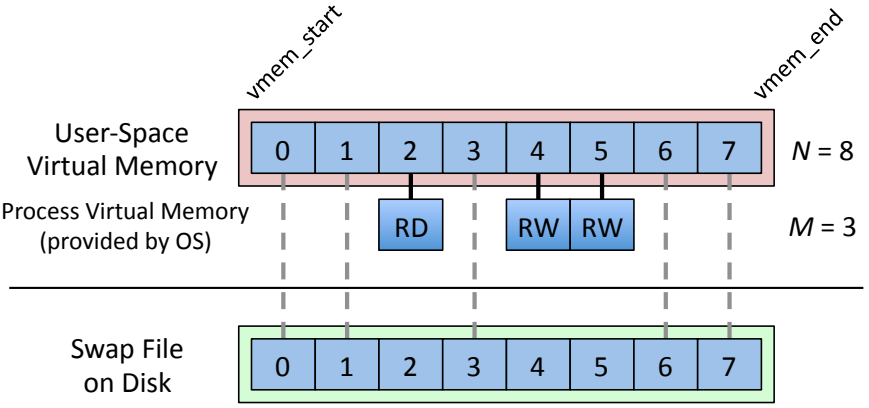
The program will access N virtual-memory pages, which are mapped to M "physical memory page-frames" (with $N \geq M$). We don't have control of physical memory in our user program, so instead we will impose a limit on the maximum number of virtual pages that may be mapped in our virtual address space. For example, we might have 4K virtual pages ($N = 4096$), and an imposed limit of 10 pages that may actually be mapped into our virtual address space at any given time ($M = 10$).

As pages are accessed, we will sometimes have faults (seg-faults in our case), because the virtual page being accessed won't be mapped into the program's address space. To resolve the fault, we must map the virtual page, but to satisfy the "physical memory" constraint, if we already have M pages mapped, we must unmap some other page before we can map the new page into memory. Thus we will never exceed our limit of having M pages mapped at any given time.

Of course, the page being unmapped will contain data, so it must be stored to disk before we can unmap it. Thus, each virtual page is mapped to a *slot* in a *swap file*. (Some operating systems can use a swap partition instead of a swap file; there are benefits and drawbacks of each approach.) Each slot is the size of one page, so if a page is 4KiB, a slot is also 4KiB. (It is very important to understand the distinction between these terms "page," "frame," and "slot." The main difference is simply in where the storage resides.)

There are many ways we can map virtual-memory pages to swap slots, but we will keep it simple: Page i will be stored in slot i. We don't care about using our swap file optimally, since it will only be a few tens of megabytes in size.

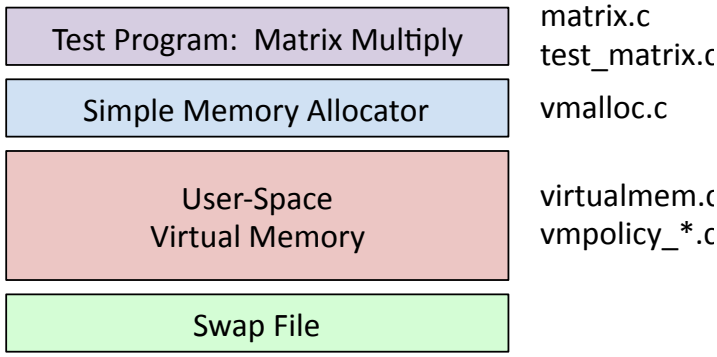
To the right is a simple diagram of the above description, with an 8-page virtual memory that uses 3 "physical page frames." The system will use the various memory-mapping functions described earlier to manage the process' virtual memory. Pages that aren't present in the process' virtual memory are stored in corresponding slots in the swap file on disk.



5 Test Program

If we are going to experiment with virtual memory, we need to have a program that uses a lot of memory, and with access patterns with enough regularity that we will see the benefits and drawbacks of various policies. The provided test program simply generates two *size*×*size* matrices populated with random values, and then multiplies them together into a third *size*×*size* matrix. The matrix dimension *size* is the only required command-line argument for the program.

A diagram of the various components, and the corresponding source files, is to the right. The matrices are allocated from the virtual memory pool using a very simple allocator (virtually identical to the unacceptable allocator in HW3). It doesn't even provide deallocation; our test doesn't require it.



The virtual memory pool is configured to be 16MiB; by varying the matrix size we can vary the allocation requirements of the program. The test uses $3\times size^2\times$ **`sizeof(int)`** $= 12\times size^2$ bytes, which sets an upper bound of $size < 1183$.

Finally, the test program provides a number of command-line options so that we can exercise our virtual memory system in several ways.

`--max_resident` | **`-m`** *integer*

> Specifies the maximum number of pages that may be resident in the virtual memory pool. The default is 64, and the maximum is 4096 (all pages may be resident). The smaller this value, the more faults we will generate.

`--seed` | **`-s`** *long*

> Specifies a random seed for the program to use. If not specified, the program will use the system time as a seed. This doesn't affect program behavior substantially, although you will see some variations in how the first page-replacement policy behaves. (More on that later.)

As an example, to test the program with 1000×1000 matrices and 1024 resident pages, one could type: **`./test_matrix -m 1024 1000`**

6 User-Space Virtual Memory Design and Implementation

The IA32 CPU's Memory Management Unit (MMU) records two important bits in each page's Page Table Entry (PTE) – whether the page has been accessed, and whether the page is dirty. Unfortunately, we don't have access to this functionality, so we must emulate it in software. (Interestingly, the current generation of ARM processors used in the vast majority of mobile phones also doesn't have this capability – their MMUs simply don't support it – and therefore the mobile phone kernel must emulate this functionality in the same way we do in this assignment.)

Answer these questions in the file **`questions.txt`**.

a) Assume that a page is already mapped into the process' virtual address space. Using the permission values supported by **`mmap()`** and **`mprotect()`** (section 3.1), describe a mechanism that will allow us to detect when a page is accessed. (Note: We only care to detect that it is accessed at least once; we don't need to know every time it has been accessed.)

b) Similarly, describe a mechanism using these permission values that will allow us to detect when a page becomes modified (a.k.a. dirty).

Complete the following tasks in the file **`virtualmem.c`**. Note that the program will not run successfully until tasks c-e are completed and debugged. In fact, if you compile and build the initial version, you will end up in a **`SIGSEGV`** infinite-loop since the handler never resolves the fault.

c) Complete the implementation of the **`map_page()`** function, which is responsible for loading a page from the swap file into virtual memory. The operation of this function is straightforward:
   1) Use the **`mmap()`** function to add the page's address-range to the process' virtual memory. Use the flags **`MAP_FIXED | MAP_SHARED | MAP_ANONYMOUS`** to force **`mmap()`** to use the specified address, and to use the anonymous file so that the page will initially be filled with zeros.

   2) Seek to the start of the corresponding slot in the swap file using the **`lseek()`** function, and then read the slot's contents into the page you mapped in step 1, using the **`read()`** function.

      Hint: You will notice that you are writing to the page, so you will want to specify both the read and write permissions in your call to **`mmap()`** in step 1.

   3) Update the page table entry for the page to be resident, and set the appropriate permissions on the page using the **`set_page_permission()`** function (which will also call the **`mprotect()`** UNIX function to set the permissions on the process' virtual memory page you mapped in step 1).

   Make sure you detect and report any errors that can occur on any of the UNIX system calls. Doing so will save you time during debugging; if a UNIX API reports an error then you are probably doing something wrong. You can see the appendix for details on how to do this.

d) Complete the implementation of the **`unmap_page()`** function, which is responsible for removing a page from the process' virtual memory mapping, as well as saving the page's contents to the corresponding slot of the swap file if necessary. The steps will basically be the reverse of the previous function:

   1) If the page is dirty, seek to the start of the corresponding slot in the swap file using **`lseek()`**, then save the page's contents to the slot using the **`write()`** function. If the page is clean, this step should be skipped.

      Hint: Again, since the **`write()`** function will read the page's contents, you should use the **`set_page_permission()`** function to ensure that the page's contents are readable before calling the **`write()`** function. Otherwise you are likely to see an error, particularly when you start implementing various paging policies.

   2) Use the **`munmap()`** function to remove the page's address-range from the process' virtual address space. This call is pretty simple.

3) Finally, update the page table entry for the page to be "not resident" by calling the **`clear_page_entry()`** function.

Again, you must detect and report any errors that can occur on any of the UNIX system calls.

e) Once these two functions are completed, you should be able to complete the implementation of the **`sigsegv_handler()`** function, our seg-fault handler! This function has two important purposes. First, it must resolve "page faults" by loading non-resident pages into memory when they are accessed. Second, it must also properly update the "accessed" and "dirty" bits of pages' page table entries at the appropriate time. This is where you must apply your answers to parts a and b, to ensure that your handler is able to update the "accessed" and "dirty" bits properly.

Here are a few hints for how your implementation will work:

- Your implementation will need to make decisions based on two sources of information, the value of **`infop->si_code`**, and the current state in the page's page-table entry (PTE). The **`si_code`** value may be **`SEGV_MAPERR`**, which means that the faulting address had no corresponding mapping, or it may be **`SEGV_ACCERR`**, which means that the address is valid, but the access violated the current constraints on the memory page. You can use these details along with the page's PTE to determine what to do.

- You should only call **`map_page()`** when a page isn't already resident in virtual memory (i.e. **`si_code`** will be **`SEGV_MAPERR`**). If the page is already resident, you will probably resolve the segfault using the **`set_page_permission()`** function, as well as the helper functions that manipulate page table entries.

- Your handler must respect the constraint of how many pages may be resident at any given time. This is the only situation in which you will call **`unmap_page()`**. A snippet of code will be given to you in the comments for the segfault handler for how to do this.

6.1 Debugging Virtual Memory

Once you have completed the above functions, you should be able to build the code and run the **`test_matrix`** program. At the top of the **`virtualmem.h`** file is a **`VERBOSE`** flag that can be set to 1 for extremely verbose output. This is enabled by default so that you can see if your code works correctly. As you increase the matrix dimensions, the program will require more and more virtual memory pages.

You can use the test program to try out your code incrementally. Once you have implemented **`map_page()`**, you can try using small matrices that will fit entirely within a single page, such as **`test_matrix 10`**. You can also set the "maximum resident" limit to 4096, the total number of pages in the virtual memory pool, to avoid running the page-eviction code at all, e.g. **`test_matrix -m 4096`**.

Once your code seems to work, you should run with a smaller number of virtual pages, e.g. the default of 64 or even fewer, so that you can really give the virtual memory system a workout. Note that it will get very slow as the number of required page-loads increases, but if the program still works properly then you will have confidence that your program is working successfully.

If you need to run the debugger, you will have an interesting challenge in that segmentation faults are expected and normal. Therefore, when you start GDB, you will need to tell the debugger to ignore segfaults, like this:

```
[user]> gdb test_matrix
...
Reading symbols from .../cs24hw8/test_matrix...done.
(gdb) handle SIGSEGV nostop noprint
Signal     Stop   Print   Pass to program   Description
SIGSEGV    No     No      Yes               Segmentation fault
(gdb) run 100
... [program output]
```

You might have noticed that the **`abort()`** function is used liberally within the virtual memory system; this is on purpose. The **`abort()`** function sends a **`SIGABRT`** signal to the calling process, which will stop the debugger and allow you to look at the program's context. If this situation occurs, you will probably not be in your code, so you need to use the **`where`** (or **`bt`**) command to look at the call-stack, and then use the "**`up n`**" command (where ***n*** is the number of frames to step up) so that you can look at the local variables and code where the error occurred.

You may also notice in the segfault handler that we do report segfaults from other parts of the program, if they don't occur within our virtual memory region. Thus, you will be informed if an unexpected segfault occurs, and you can debug it using the described techniques.

7 Paging Policies

The matrix-test program uses a random page-replacement policy. That is, when a page must be evicted, the policy randomly chooses one of the resident pages. This is not ideal, since we would like the policy to respond to program behavior. Thus, you will implement a couple of other policies for this section. The page-replacement policy is implemented in the **`vmpolicy*`** files.

f) Before you write additional policies, record in **`questions.txt`** how many page-loads are generated by running the test program as follows: **`test_matrix -m 1024 1000`**

This value will change due to the random nature of the eviction policy, so feel free to run the program a few times if you want to compute an average. (You aren't required to do so!)

7.1 The FIFO Policy

The FIFO page-replacement policy is a very simple one, which generally does better than the RANDOM policy, but it still makes pretty poor decisions. All resident pages are kept in a FIFO queue. When a new page is loaded (mapped), it is added to the back of the queue. When a page must be evicted, the victim is taken and removed from the front of the queue. Notice that we don't care at all whether the page is being used; we always take the page at the front of the queue.

g) Copy the file **`vmpolicy_random.c`** into a new file **`vmpolicy_fifo.c`**, and modify it to implement the FIFO page-replacement policy. This should be pretty easy, since the random policy implementation already uses a linked list. **Make sure to update all comments!**

Add a build rule to the **`Makefile`** that builds **`test_matrix_fifo`**. You can copy the rule for building **`test_matrix`**, but use **`vmpolicy_fifo.o`** instead of **`vmpolicy_random.o`**.

Also, add **`test_matrix_fifo`** as a prerequisite to the **`all`** target, so that it's easy to build.

Once this is done, run **`test_matrix_fifo -m 1024 1000`** (the same arguments as before), and record the page-load count in **`questions.txt`**. (You should see an improvement.) This value shouldn't change from run to run, since the FIFO policy is definitely not random!

7.2 The CLOCK/LRU Policy

The LRU (Least Recently Used) page-replacement policy is a pretty good policy that always evicts the page that was used (read or written) furthest in the past. (There turn out to be other policies that are better than LRU, but learning about these policies is left as an exercise for the curious.) The main issue with LRU is that it is prohibitively costly to implement in hardware, because *every single memory access* must update some timestamp value associated with each page (e.g. in the page's PTE).

Therefore, a number of page-replacement policies have been created that approximate LRU to varying degrees. A very simple one is known as CLOCK/LRU – it relies on a periodic timer interrupt to approximate LRU. As with FIFO, pages are added to the end of a queue when they are mapped into memory.

- Every time a timer interrupt occurs, CLOCK/LRU traverses all resident pages in the queue. If a page has been accessed since the last timer interrupt, its "accessed" bit is cleared, and the page is moved to the back of the queue. Nothing is done if a page has not been accessed.
- When a page must be chosen for eviction, it is taken from the front of the queue, as with FIFO. However, you will notice that given the behavior of the timer interrupt, pages that have been accessed "recently" will be towards the back of the queue, so the front of the queue will [probably] be pages that haven't been accessed very recently.

In practice, this is a pretty inexpensive algorithm to implement, only slightly more costly than FIFO, and it's significantly more intelligent than FIFO.

h) Copy your **`vmpolicy_fifo.c`** file (or **`vmpolicy_random.c`** if you prefer, but the FIFO implementation will be closer to what you need) into a new file **`vmpolicy_clru.c`**, and modify it to implement the CLOCK/LRU page-replacement policy. Unlike your previous policy, you will need to implement the **`policy_timer_tick()`** function; this is the function called by the periodic timer interrupt. **(Again, update all comments when you copy the source!)**

Note that your timer-tick function won't unmap anything. It should just manipulate pages' page-table entries, and the permissions of pages via the **`set_page_permission()`** function.

Again, add a build rule to the **`Makefile`** that builds **`test_matrix_clru`**, using the object-file **`vmpolicy_clru.o`**. Also, add **`test_matrix_clru`** as a prerequisite to the **`all`** target so that it's easy to build.

Once this is done, run **`test_matrix_clru -m 1024 1000`** to see how your CLOCK/LRU policy does, and record the page-load statistic into **`questions.txt`**. (Note: Even though you will see a *substantial* reduction in page-loads, the CLOCK/LRU code will likely run slower than FIFO implementation, simply because the whole implementation has a lot of overhead. There are many ways to eliminate this overhead, but it's not really the point of the assignment. If you want to try to make it faster, be my guest, but it's not required!)

Appendix A: Useful UNIX Functions

The most important UNIX functions for virtual memory management are described in section 3; they will not be described here.

The following functions are very useful for reading and writing files. In UNIX, open files are represented with an integer *file descriptor*, which is passed to the various functions to represent the file being manipulated. The file descriptor of the swap file in the virtual memory implementation is called fd_swapfile.

Additionally, UNIX maintains a "current position" for every open file; when a read or write is requested, it is performed at the current position, and the position is moved forward by the number of bytes read or written. If we want to perform a read or write at a specific position, we must *seek* to the desired position before performing our read or write.

`off_t lseek(int fd, off_t offset, int whence)`

This function sets the "current position" for the specified file. The whence value indicates the reference point for the offset value:

- **`SEEK_SET`** – the offset is relative to the start of the file
- **`SEEK_CUR`** – the offset is relative to the current file position
- **`SEEK_END`** – the offset is relative to the end of the file

You will want to use **`SEEK_SET`** in your virtual memory implementation.

If **`lseek()`** fails, it will return -1. If it succeeds, it will return the new absolute position relative to the start of the file.

`size_t read(int fd, void *buf, size_t count)`

This function attempts to read up to **`count`** bytes from the specified file, into the buffer starting at **`buf`**. Note that the function is allowed to read fewer than **`count`** bytes, e.g. if the "current position" is near the end of the file and fewer than **`count`** bytes are available.

This function uses the "current position" associated with the file descriptor, and moves the position forward by the number of bytes read.

If **`read()`** fails, it will return -1. If it succeeds, it will return the number of bytes actually read.

`ssize_t write(int fd, const void *buf, size_t count)`

This function attempts to write up to **`count`** bytes from the buffer pointed to by **`buf`**, to the specified file. Note that this function is allowed to write fewer than **`count`** bytes, e.g. if the storage device doesn't have enough space to hold all **`count`** bytes.

This function uses the "current position" associated with the file descriptor, and moves the position forward by the number of bytes written.

If **`write()`** fails, it will return -1. If it succeeds, it will return the number of bytes actually written.

Appendix B: Detecting and Reporting UNIX API Errors

An important aspect of this assignment is that your code must detect and report any errors that are returned by UNIX API calls. This may sound difficult, but it is not that hard to add, and it will make debugging your code much easier. Most of the UNIX API calls return -1 to indicate an error, so most of the time your error handling will look like this:

```
if (lseek(...) == -1) {
    perror("lseek");
    abort();
}
```

The **`perror()`** function is a helpful C library function that will print system error information to **`stderr`**, with a prefix of whatever you specify as the argument to **`perror()`**. Thus, the above pattern makes it easy to tell exactly what system call generated the error.

The calls **`mmap()`**, **`munmap()`** and **`mprotect()`** also return -1 if an error occurs. The **`mmap()`** function is a bit more complex because it also returns a pointer on success, so you may need to use code like this:

```
void *addr;
...
addr = mmap(...);
if (addr == (void *) -1) {
    perror("mmap");
    abort();
}
```

As stated in the previous section, the calls **`read()`** and **`write()`** return -1 if an error occurs, but they also return the number of bytes actually read or written. It is important to also verify that the number of bytes read/written matches our expectation. Therefore, you might do something like this:

```
int rc;
...
rc = read(fd_swapfile, addr, PAGE_SIZE);
if (rc == -1) {
    perror("read");
    abort();
}
if (rc != PAGE_SIZE) {
    fprintf(stderr, "read: only read %d bytes (%d expected)\n",
            rc, PAGE_SIZE);
    abort();
}
```

Note that it's always the best policy to print what happened, and what was expected; it makes debugging sessions a bit easier, and every little bit helps.