```
* Debugging, cont.

If stack trace doesn't make any sense, probably something corrupted kernel
state.

If buffers/vars seem to change values unexpectedly, look for mem
corruptions, probably due to your code added and running.

In long running programs, like an OS, can have a temporal difference b/t when
bug took place and when symptoms manifest.
If you suspect a bug happened a long time ago, don't try to "fix" your
current code: reboot, and then retry new code to see if it works.

Some bugs trigger an oops quickly: major buffer overflow.  But a minor
overflow or small memleak may take a long time to detect.

Races, deadlocks, and timing based bugs: everything affects timing.  If you
see good behavior some of the time, and bad behavior the rest of the time,
it could be a race condition.  Worst races manifest themselves a very small
fraction of the time.

To try and reproduce a race bug, it helps to change timing conditions, such
as h/w configuration, speeds, workloads, etc.

Oops messages:

1. "null pointer dereference and 0x00000000" -- NULL ptr dereferenced
2. "null pointer dereference and 0x00000008"

Can happen if you have have code such as:

        struct foo {long i; long j} *ptr; // assume sizeof(long)=8B
        // some code
        try to deref ptr->j but 'ptr' is NULL, and j is 8 bytes into struct

3. "null pointer dereference and 0xfffffff0"

Same as 2 above, dereferencing a NULL ptr, with a negative relative address
to the NULL ptr.

*  Translation b/t virtual and physical addresses

Often use calls like copy_from/to_user.

Suppose you're on a 32-bit system, max 4GB RAM, and you have only 1GB
physical RAM.  Because most processes don't ask for a lot of virtual memory,
let's assume no more than 3GB.  Then Linux can partition the 4GB address
space as follows:

1. Map the physical RAM to the upper 3-4GB
2. Offer virtual addrs from 0-3GB.

Benefit: can figure out if something is virtual or physical by checking if
it's above or below the 3GB number.  So kernel can automatically translate
b/t physical/virtual addrs b/c it knows which is which.  Problem is that if
you assume this, then your code will cause major corruptions on
systems/architectures w/o this optimization.

Some functions assume that what you give them is user memory, and do the
translation to physical mem deep inside the code.

e.g., if 'buf' is kernel mem, must turn off virt-to-phys translation by
setting the kernel data segment (KERNEL_DS):
```

```
    oldfs = get_fs();
    set_fs(KERNEL_DS);
    bytes = filp->f_op->read(filp, buf, len, &filp->f_pos);
    set_fs(oldfs);
```

* bitmaps

```
// Bitmaps

#define FLAG_N  0x01 // decimal 1
#define FLAG_D  0x02 // decimal 2
#define FLAG_V  0x04 // decimal 4
#define FLAG_X  0x08 // decimal 8
#define FLAG_K  0x10 // decimal 16

// to set one or more bits
u_int b1 = (FLAG_N | FLAG_V);
b1 |= FLAG_K;

// to check if a bit is on
if (b1 & FLAG_V) {
  // bit for FLAG_V is on
}

// how to turn off a bit, e.g., turn off bit FLAG_D
b1 &= ~FLAG_D;
b1 &= ~(FLAG_D|FLAG_K); // turn off multiple bits
```

* kmalloc flags

See comment above kmalloc fxn here:

http://lxr.fsl.cs.sunysb.edu/linux/source/include/linux/slab.h#L466

For user level malloc, kernel tries very hard to give user the memory, even a large allocation. Recall it's contiguous in virtual space, but not in physical space.

In kernel space, it's more likely to not have that much physical contiguous memory. That's why kmalloc has different flags to control behavior.

Most useful flags for hw1 are GFP_KERNEL + GFP_WAIT (regular kernel memory, and willing to wait for allocation). But find out what others are doing in similar code (e.g., sys_open/read/write).

* efficiency

First get code to work, worry about efficiency later.

When opening/reading/writing files, what's the right size?

1. naive: alloc big buffer and read whole file in: bad, kmalloc may not have this much ram, puts a lot of stress on mem system. Bad style.

2. read and compare 1 byte at a time: inefficiency, wasteful to call fxn for just 1 byte to read.

3. Best: use the optimal "blocking unit" for the system in question.

- hard disks: usually read/write in terms of 512B "sectors"
- more modern SSDs, may use a 4KB "sector"
- most CPUs divide memory into pages of 4KB, or 8KB, and can even be configurable.

So, use 4KB, or PAGE_SIZE (or PAGE_CACHE_SIZE) in Linux.

In some cases, a small multiple integer of this "optimal" read/write unit may provide extra throughput (but not a large multiple).

* cleanup

Don't leave partially written o/p files: hard for users/programs to handle, can confuse and cause inadvertent data loss. Best to behave "atomically" -- either fail or succeed entirely.

Each time you write to the o/p file, you can get errors like ENOSPC, ENOQUTA, ENOMEM, EIO, etc. So need to cleanup this persistent state.

So, let's delete the partial file use

vfs_unlink(struct inode *dir, struct dentry *dentry)
- dir: inode of parent directory that includes the file name to unlink
- dentry: represents the name and inode of the object to delete/unlink

Note: vfs_unlink (etc.) has some PRE-conditions.
- dir inode must be locked before you call vfs_unlink. careful that 'dir' isn't already locked! Look at sys_unlink and follow it.

In POSIX, you can unlink an open file. last ref disappears after the last close(2) of the file:

```
open()
unlink()
???
close()
```

In kernel, it's best to filp_close the file and THEN call vfs_unlink.

Last Q: if there's already an o/p file X, and you're trying to write new o/p file Y, and write of Y fails mid-way, you don't want to delete Y, b/c you'll have deleted all of the good file X. Solutions;

1. write the o/p to a TEMP name. If not ok: delete it. If OK, delete old, rename new to old.
2. rename existing file. write o/p file. if ok, delete renamed old file, else rename back.