```
// error return values vs. valid values from functions

// many functions will report in different ways whether they succeeded or failed

// boolean: return true (1, or any non-zero value) or false (0)

// read(2)-like functions: return -errno on error, and >=0 for "number of bytes read"

// k/malloc() like functions: 0 means -ENOMEM (or some other fixed error),
// else valid ptr

// dentry -> revalidate(): -errno (on serious error), 0 (file is valid), 1
// (entry is stale, remove from cache, and re-retrieve entry, say using
// ->lookup)

// Q: but how to handle functions that  need to return both a valid ptr/addr
// and multiple errors?
// A: partition the space to a small part of returning N errors, and the
// rest are valid pointers.
// e.g., 32-bit integer, 2^32 = 4B
// Let's say we want to support a max of 1024 errors.
// then, any number from 0-(2^32-1024) is a valid pointer
// any number above 2^32-1024 is an error.  Get the actual err number by
// subtracting from 2^32-1.  Means: 2^32-1 == error 1, 2^32-2 == errno 2.
// this is called an encoded error.
// example

struct file *fp;
int err;

fp = filp_open(...);
// wrong: if (fp == NULL)...
if (IS_ERR(fp)) // then it's an error
  err = PTR_ERR(fp); // convert an encoded PTR to an errno value
// there's also a macro ERR_PTR(-EPERM) to create an encoded ptr

// Note: this scheme does mean that certain upper addr values can never be
// returned as a valid pointer.

//////////////////////////////////////////////////////////////////////
// kernel stack is limited in size. 4-8kb
// If you overflow the kstack size, bad mem corruptions could happen, as well
// as oopses.
// there's a script in scripts/* to check stack depths.

badcode()
{
  char buf[4096]; // automatic var, 4096B long on stack
  // ...
}

goodcode()
{
  void *buf = kmalloc(4096, ...); // automatic variable, (4-8 bytes) on stack
  // ...
}

######################################################################
# debugging

1. "printf is your friend"

printf is easy to use, but it's a function that takes a certain amount of
processing, hence it can change timing conditions.
```

```
printk in the kernel:
- formats a string
- adds string to a kernel syslog buffer
- separate thread copies buffer to userlevel syslogd daemon
- also display last string on /dev/console
- syslogd in userland would print, or append log to /var/log/*

meaning:

- printks aren't synchronous. delay before you see the message on console or
  /var/log/*

- the printk cyclic buffer is NOT locked and has a limited size.  Messages
  could be interleaved, partially overlap, or even lost if you printk too
  much too fast.

2. assertions

Assertions are very fast ways to check whether a condition is true/false,
and if the assertion fails, then the kernel would panic (possible reboot),
or at least abort the current running module and thread.  Useful when you
suspect a timing bug.

2a. BUG_ON(cond);

will trigger an "oops" message, meaning assertion failed, of "cond" is true

e.g.,  BUG_ON(ptr == NULL)

An "oops" message prints useful info (see below).

2b. WARN_ON(cond)

same as BUG_ON() but continues executing, useful when the condition isn't so
serious.

2c. WARN_ON_ONCE(cond).  Like WARN_ON, but prints msg only once for that
line of code.

2d. just BUG()

Same as BUG_ON(1).  Useful if you're inside a complex branching/looping and
want to trigger a BUG at that point.

######################################################################

# an "oops" message in Linux

Can happen for many reasons, not just BUG/WARN macros (e.g., corrupt
pointers, NULL ptr deref, etc.).

1. message itself "COND failed on file F line N"
2. stack trace: which functions called others in order
3. name of the function inside which the oops took place
4. size of fxn in hex and the instruction number that triggered the oops.
Two numbers separated by a "/"

e.g., 1c/f7d

instruction 1c in function whose code size is f7d instructions, is where
the oops triggered.  Meaning closer to beginning of code.

e.g., f20/f7d
```

Above Meaning closer to end of code.

But: inlining, and CPP macros, and different compiler options could affect this estimate.

Stack trace can also be triggered anywhere by calling dump_stack(). Helps indirectly to determine that some functions got inlined, b/c they don't show up in the stack trace.

Determining functions on the stack isn't easy. Sometimes you'll see a '?' next to a stack function name: meaning kernel isn't sure if that's an actual fxn being called, or just some value on the stack that looks like a function addr ptr.

If kernel stack looks random, most likely you've corrupted the kernel stack (bad ptr, buffer overflow, etc.).

##
when you get an oops

1. kernel state is already bad, need reboot quickly
2. use shell to preserve FIRST oops message, then reboot
3. can use "dmesg" (shows kernel printf buffer), or check /var/log/*
4. reboot. May need a hard reboot, b/c module and even f/s can be blocked.

when system comes back from reboot, verify its integrity (e.g., fsck, git repo)