

## Modular Code Structure

- module implementation, e.g, `thread.c`
  - contains:
    - \* function implementations
    - \* global variable declarations
    - \* `#includes` of headers needed by implementation
    - \* definitions needed (only) by implementation
    - \* static vs. regular global variables and functions
  - static vs. regular functions and global variables
- module interface, e.g., `thread.h`
  - `#included` by code that needs to use threads
  - contains
    - \* prototypes of (public) thread functions
    - \* declarations of data structures and constants used by the interface

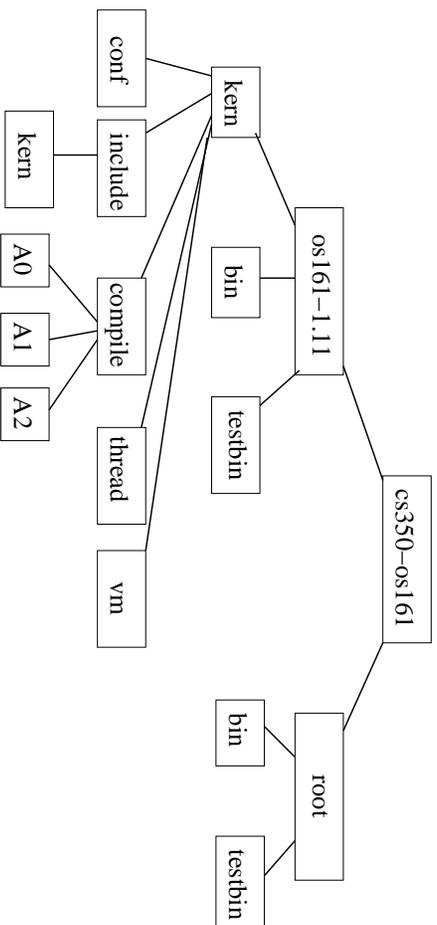
## Other Code Structuring Issues

- some header files are not tied to specific modules
  - example: `include/types.h`
- order of inclusion of header files is significant

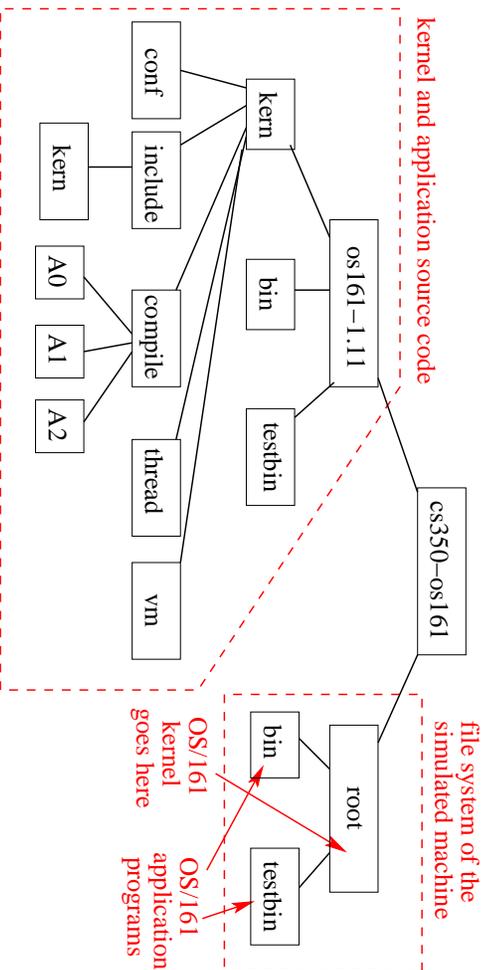
```
#include <types.h>
#include <lib.h>
#include <kern/errno.h>
#include <array.h>
#include <machine/sp1.h>
#include <machine/pcb.h>
#include <thread.h>
...
```

OS/161 convention is that `types.h` should always be included first

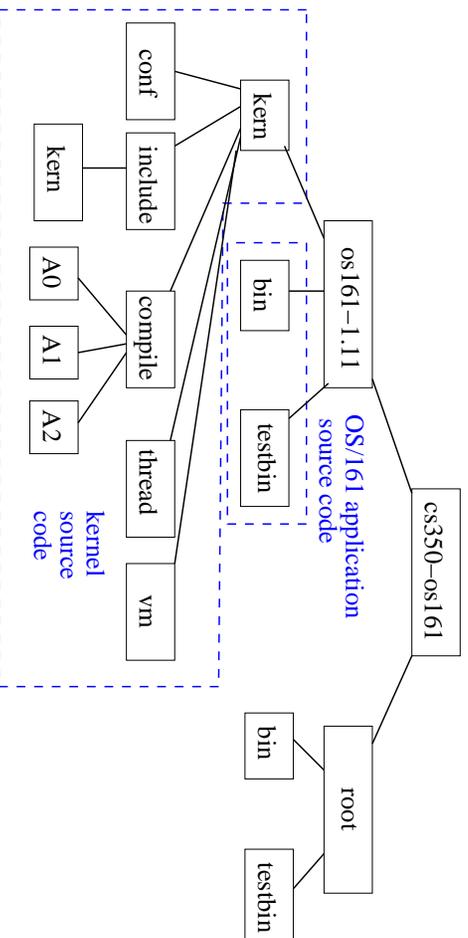
### OS/161 Directory Structure (1 of 4)



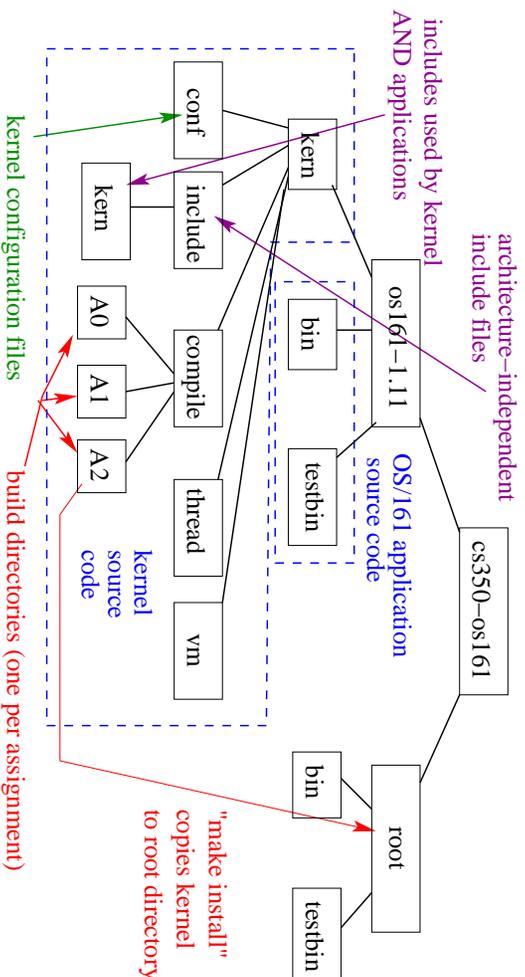
### OS/161 Directory Structure (2 of 4)



### OS/161 Directory Structure (3 of 4)



### OS/161 Directory Structure (4 of 4)



## Kernel's Standard Library

- User-level C applications can use the C standard library
- The OS/161 kernel also has a library, similar to the user-level standard library, e.g.,
  - dynamic memory management (`kmalloc`, `kfree`)
  - string functions
  - input/output (`kprintf`, `kgets`, `putc`)
  - data movement (`copyin`, `copyout`)
- kernel library is not identical to the user-level library, e.g.,
  - `kmalloc` vs. `malloc`
  - `kstrdup` vs. `strdup`
  - `kprintf` vs. `printf`

Why??

## Pointers and Arrays

```
static char *bowls;

int
initialize_bowls(unsigned int bowlcount) {
    unsigned int i;
    bowls = kmalloc(bowlcount*sizeof(char));
    if (bowls == NULL) {
        panic("initialize_bowls: unable to allocate
             space for %d bowls\n", bowlcount);
    }
    /* initialize bowls */
    for (i=0; i<bowlcount; i++) {
        bowls[i] = '-';
    }
    ...
}
```

## Dynamic Memory Allocation

```
struct semaphore *
sem_create(const char *namearg, int initial_count) {
    struct semaphore *sem;
    sem = kmalloc(sizeof(struct semaphore));
    if (sem == NULL) { return NULL; }
    sem->name = kstrdup(namearg);
    if (sem->name == NULL) {
        kfree(sem);
        return NULL; }
    sem->count = initial_count;
    return sem;
}
```

---

What does a semaphore look like?

---

## volatile Variables

```
struct semaphore {
    char *name;
    volatile int count;
};

void P(struct semaphore *sem)
{
    ....
    while (sem->count==0) { thread_sleep(sem); }
    assert(sem->count>0);
    sem->count--;
    ....
}

volatile indicates that the value of a program variable may
change “spontaneously”
```

---

## const Variables

- from kern/lib/kheap.c:

```
#define NSIZES 8
static const size_t
sizes[NSIZES] = {16, 32, 64, 128, 256, 512, 1024, 2048};
```
- from kern/include/lib.h:

```
int strcmp(const char *, const char *);
char *strcpy(char *, const char *);
```

---

---

const indicates that the value of a program variable should never change. In the case of a pointer variable, const indicates that the thing pointed to should never change.

---

---