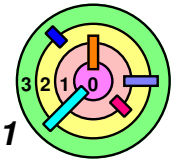
More On System Calls

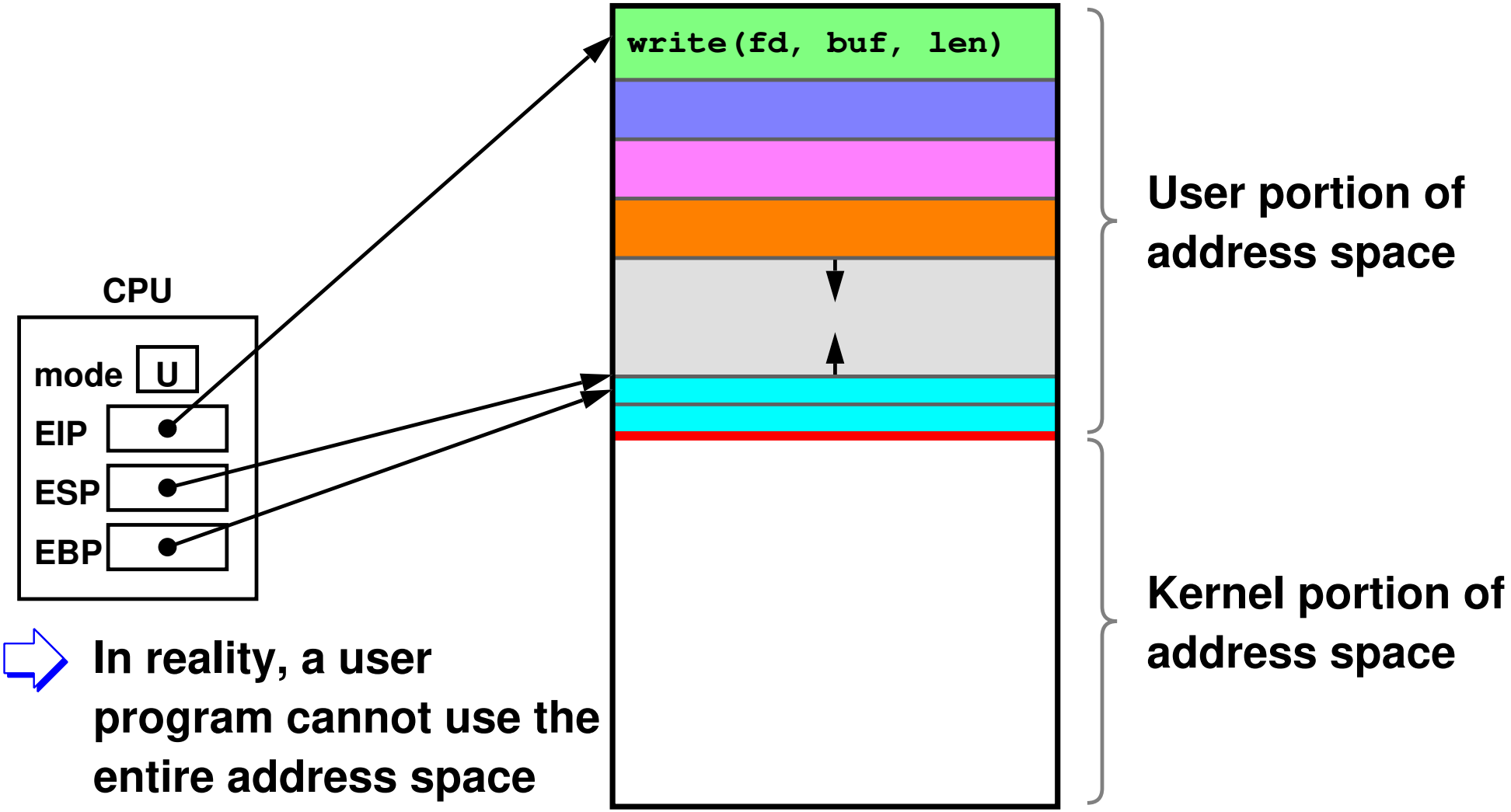
- ***Sole interface*** **between user and kernel**
  - **this interface (definition of system calls) is what distinguishes one OS from another**
    - **in this class, we focus on Sixth-Edition Unix**
- **Implemented as library routines that execute** ***"trap" machine instructions*** **to enter kernel**
- **Errors indicated by returning an invalid value**
  - **error code is in a global variable named** ***errno***

```
if (write(fd, buffer, bufsize) == -1) {
  // error!
  printf("error %d\n", errno);
  // see perror
}
```

  - **on Ubuntu: "`man 2 write`" or "`man -s 2 write`"**
  - **search man pages in all sections: "`man -k ...`"**

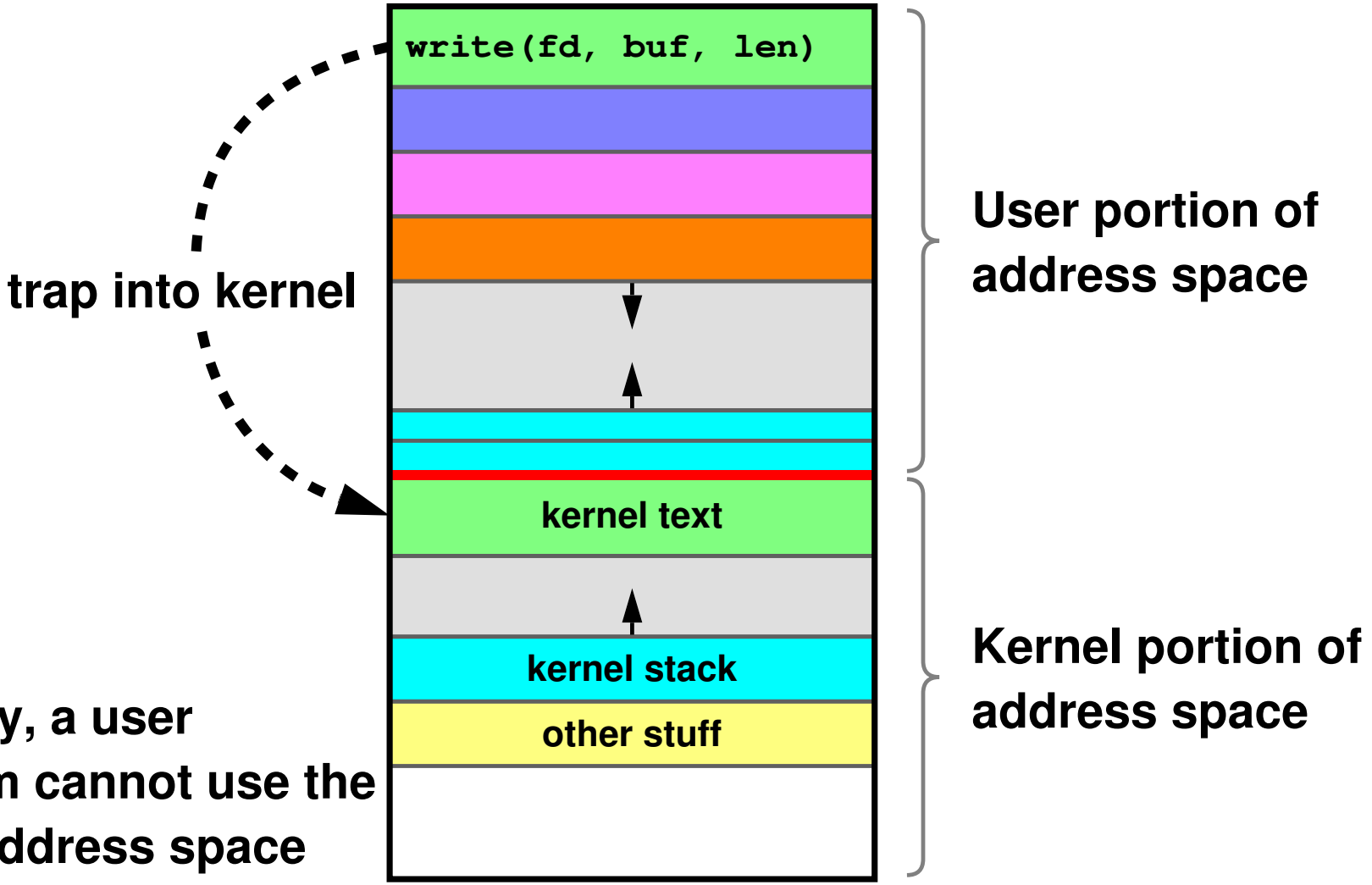


System Calls

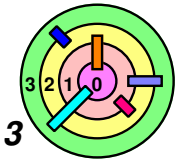


In reality, a user program cannot use the entire address space

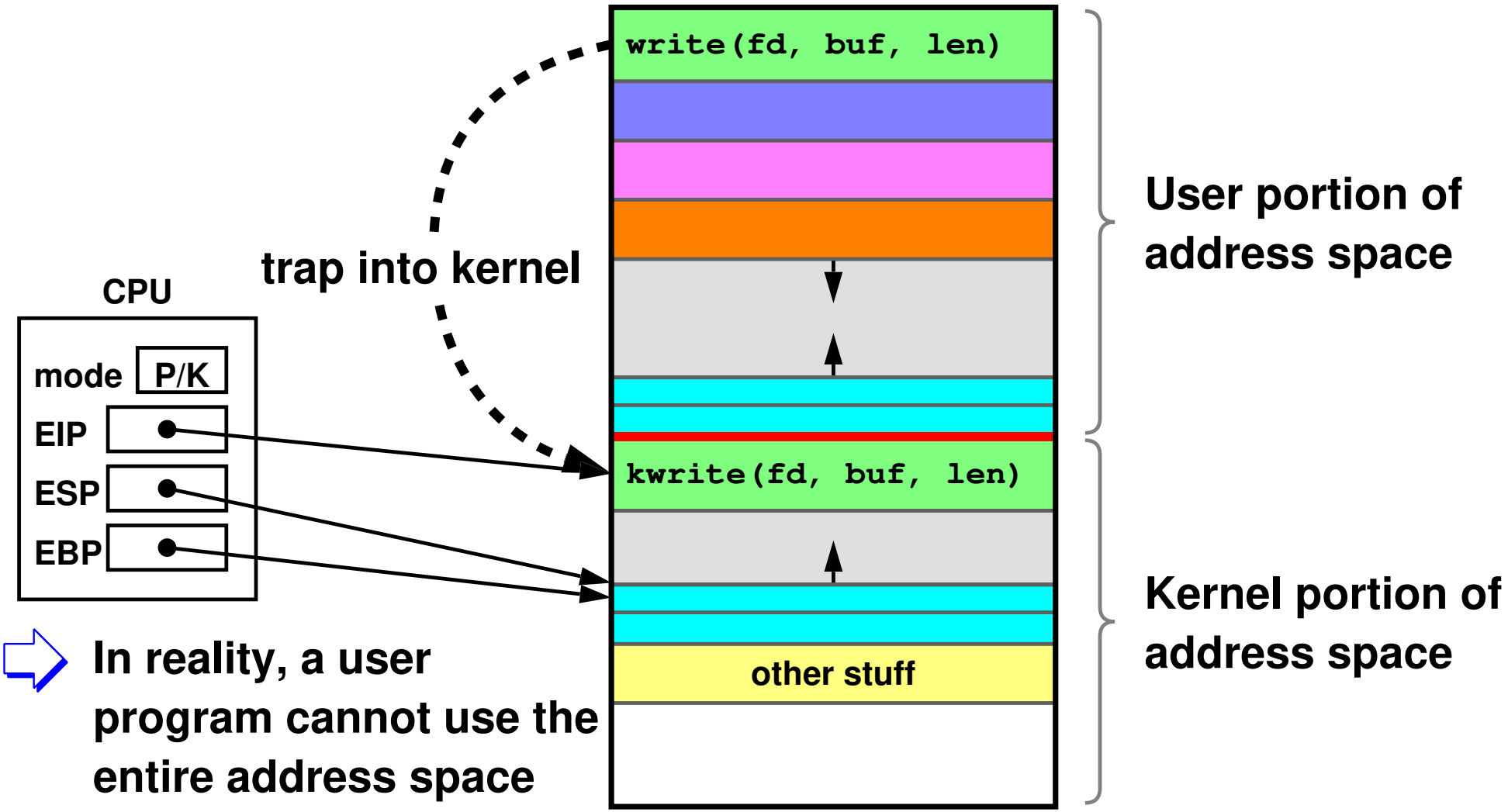
System Calls



- In reality, a user program cannot use the entire address space
- Is this the same *"thread of execution"*?
  - is this the same process?

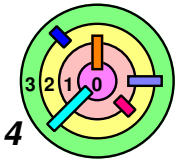


System Calls

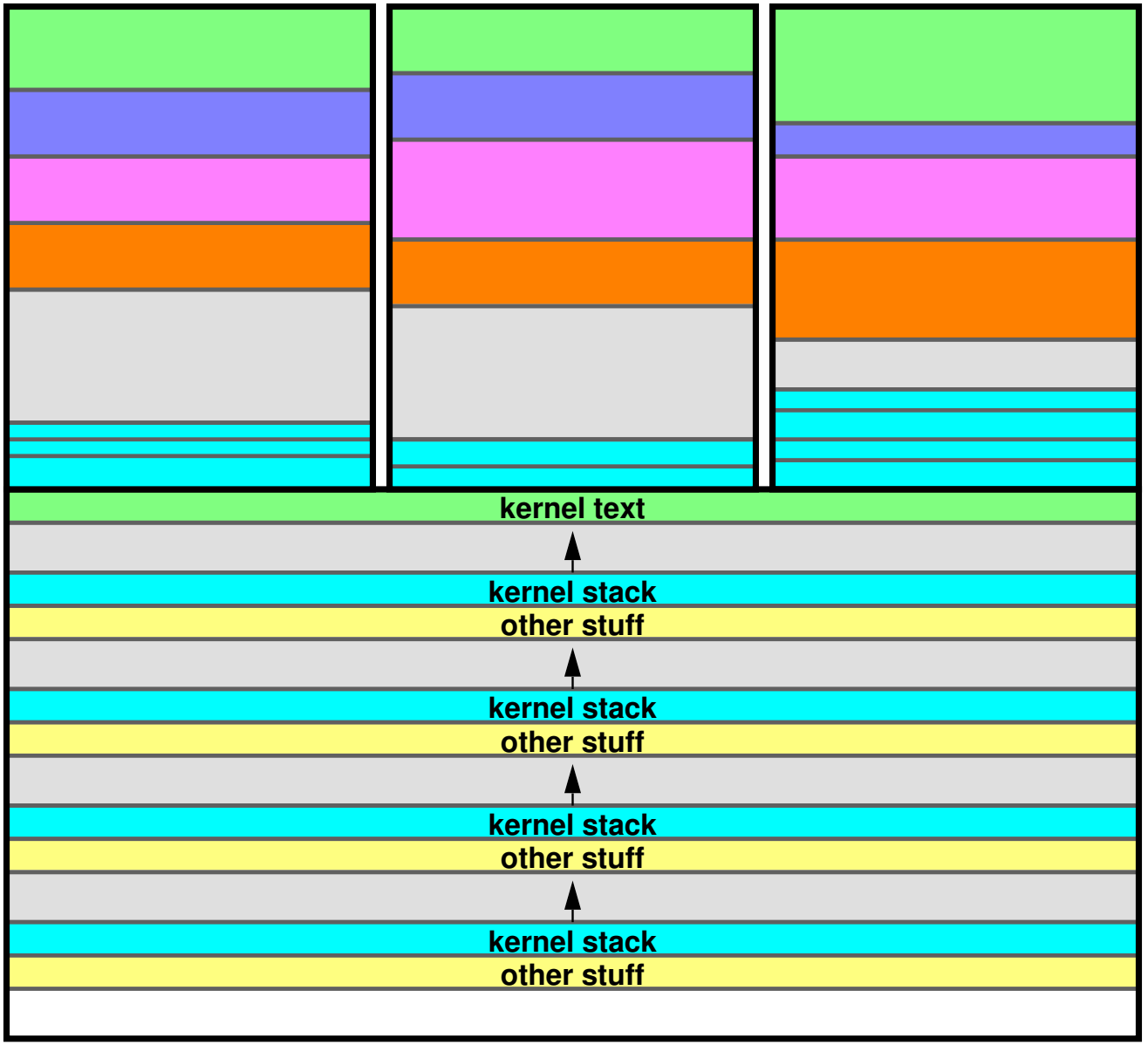


- In reality, a user program cannot use the entire address space

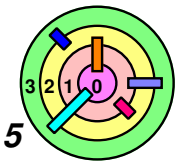
- Is this the same *"thread of execution"*?
  - is this the same process?



Multiple Processes

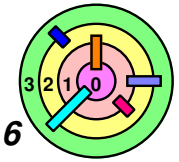


- the same kernel spans across all user processes
  - although there are kernel-only processes as well (and they don't make system calls)
- process is just an abstraction
  - the kernel is very powerful



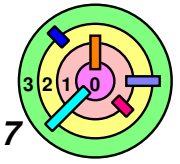
1.3 A Simple OS

- OS Structure
- Processes, Address Spaces, & Threads
- Managing Processes
- Loading Program Into Processes
- *Files*



Files

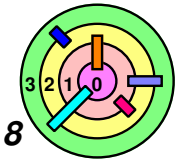
- Our "primes" program wasn't too interesting
  - it has no output!
  - cannot even verify that it's doing the right thing
  - other program cannot use its result
  - how does a process write to someplace *outside the process*?
- Files
  - abstraction of persistent data storage
  - means for fetching and storing data outside a process
    - including disks, another process, keyboard, display, etc.
    - need to *name* these different places
      - hierarchical naming structure
    - part of a process's *extended address space*
      - file "cursor position" is part of "execution context"
- The notion of a *file* is our Unix system's *sole abstraction* for this concept of "someplace outside the process"
  - modern Unix systems have additional abstractions



Naming Files

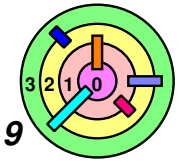
- Directory system
  - shared by all processes running on a computer
    - although each process can have a different view
    - Unix provides a means to restrict a process to a subtree
      - by redefining what "root" means for the process
  - name space is outside the processes
    - a user process provides the name of a file to the OS
    - the OS returns a *handle* to be used to access the file
      - after it has verified that the process is allowed *access* along the *entire path*, starting from root
    - user process uses the handle to read/write the file
      - avoid subsequent access checks

- *Using a handle* (which can be an index into a kernel array) to *refer to an object managed by the kernel* is an important concept
  - *handles* are essentially an *extension* to the process's *address space*
    - can even *survive execs!*



The File Abstraction

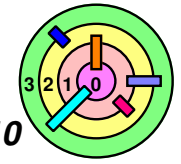
- A file is a simple array of bytes
- Files are made larger by writing beyond their current end
  - although you cannot ***read*** past the current end
- Files are named by paths in a naming tree
- File API
  - `open()`, `read()`, `write()`, `close()`
  - e.g., `cat`
- System calls on files are ***synchronous*** (unfortunately, Computer Science likes to use the same word to mean different things)
  - here, it means that the system call will not return until the operation is considered completed



File Handles (File Descriptors)

```
int fd;
char buffer[1024];
int count;
if ((fd = open("/home/bc/file", O_RDWR) == -1) {
  // the file couldn't be opened
  perror("/home/bc/file");
  exit(1);
}
if ((count = read(fd, buffer, 1024)) == -1) {
  // the read failed
  perror("read");
  exit(1);
}
// buffer now contains count bytes read from the file
```

- what is O_RDWR?
- what does `perror()` do?
- *cursor* position in an opened file depends on what functions/system calls you use
  - what about C++?



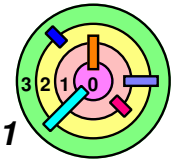
Standard File Descriptors

Standard File Descriptors

- 0 is `stdin` (by default, "map/connect" to the keyboard)
- 1 is `stdout` (by default, "map/connect" to the display)
- 2 is `stderr` (by default, "map/connect" to the display)

```
main() {
   char buf[BUFSIZE];
   int n;
   const char *note = "Write failed\n";

   while ((n = read(0, buf, sizeof(buf))) > 0)
     if (write(1, buf, n) != n) {
        (void)write(2, note, strlen(note));
        exit(EXIT_FAILURE);
     }
   return(EXIT_SUCCESS);
}
```

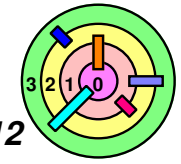


Back to Primes

- Have our primes program write out the solution, i.e., the `primes[]` array

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
   ...
   for (i=1; i<nprimes; i++) {
      ...
   }
   if (write(1, prime, nprimes*sizeof(int)) == -1) {
     perror("primes output");
     exit(1);
   }
   return(0);
}
```

  - the output is not readable by human



Human-Readable Output

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
   ...
   for (i=1; i<nprimes; i++) {
      ...
   }
   for (i=0; i<nprimes; i++) {
      fprintf(stdout, "%d\n", prime[i]);
   }
   return(0);
}
```

- `fprintf(stdout, ...)` is the same as `printf(...)`
  - `stdout` is a pre-defined *file pointer*
  - please see the *Programming FAQ* regarding the difference between a *file descriptor* and a *file pointer*