Andrew Hanushevsky:
Basic I/O API's

Goals

- Overview of basic I/O API's
- Explain some confusing I/O terminology
  - Blocking and non-blocking I/O
- Explain performance oriented I/O open options
  - 32 and 64 bit I/O
- I/O peculiarities
  - Threading implications
- How to get the most performance

Basic Read API's

- `#include <unistd.h>`
  - `ssize_t  read(int fd, void *buf, size_t count);`
  - `ssize_t pread(int fd, void *buf, size_t count, off_t offset);`
- `#include <sys/uio.h>`
  - `ssize_t readv(int fd, const struct iovec *iov, int iovcnt);`
    - struct iovec

```
{void  *iov_base; /* Buffer address */
 size_t iov_len;  /* Number of bytes*/
};
```

Basic Write API's

- **#include <unistd.h>**
  - **ssize_t write(int** *fd*, **void** **buf*, **size_t** *count***);**
  - **ssize_t pwrite(int** *fd*, **void** **buf*, **size_t** *count*, **off_t** *offset***);**
- **#include <sys/uio.h>**
  - **ssize_t writev(int** *fd*, **const struct iovec** **iov*, **int** *iovcnt***);**
    - struct iovec

```
{void  *iov_base; /* Buffer address */
 size_t iov_len;  /* Number of bytes*/
};
```

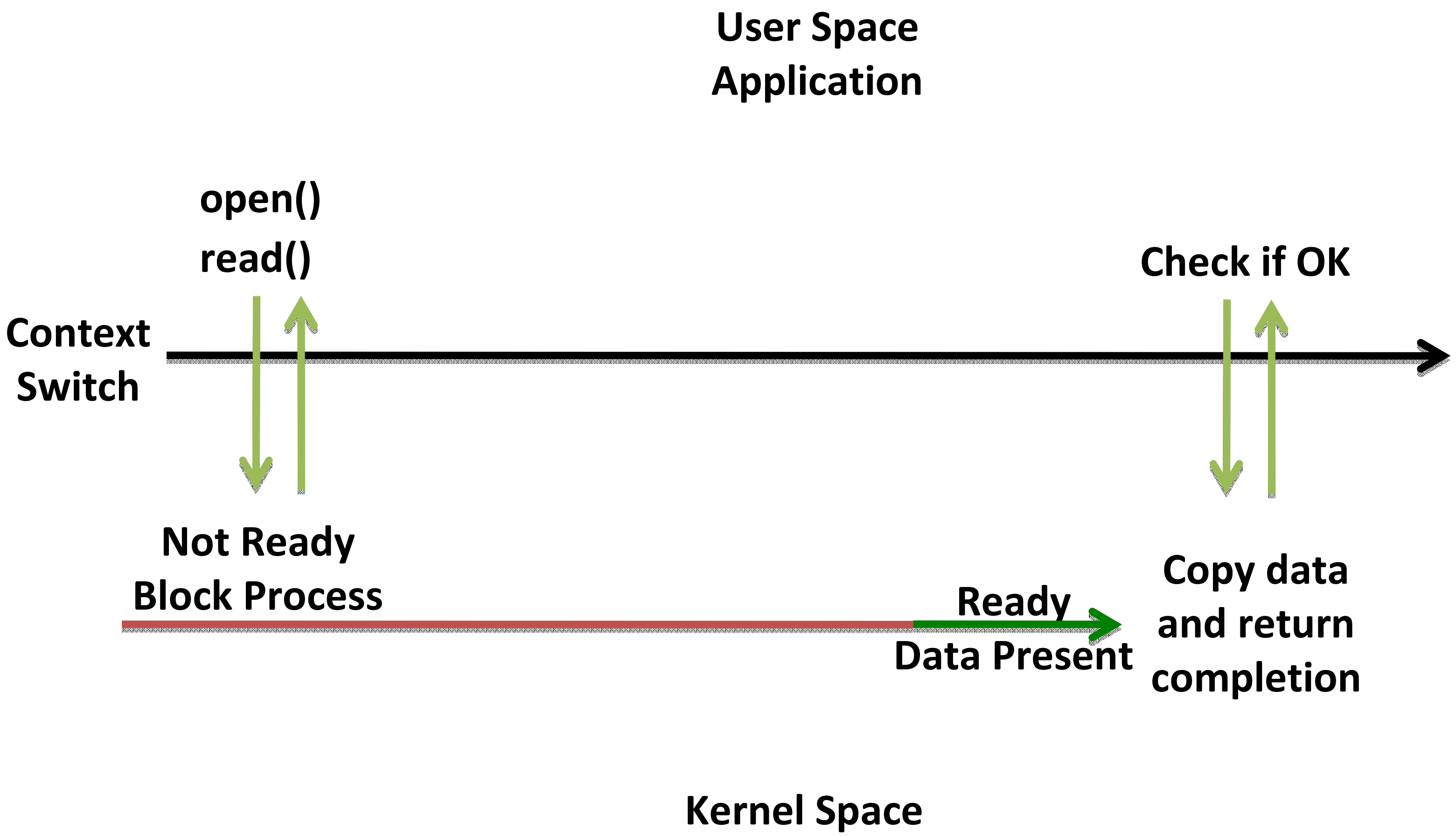
Basic I/O API's

- Can use API for *any* type of device
  - Synchronous
    - I/O occurs only when thread is suspended
  - Handles blocking and non-blocking I/O
    - Selected with open() flags
    - Special errno value indicates blocking state
  - Now to explain blocking vs non-blocking I/O

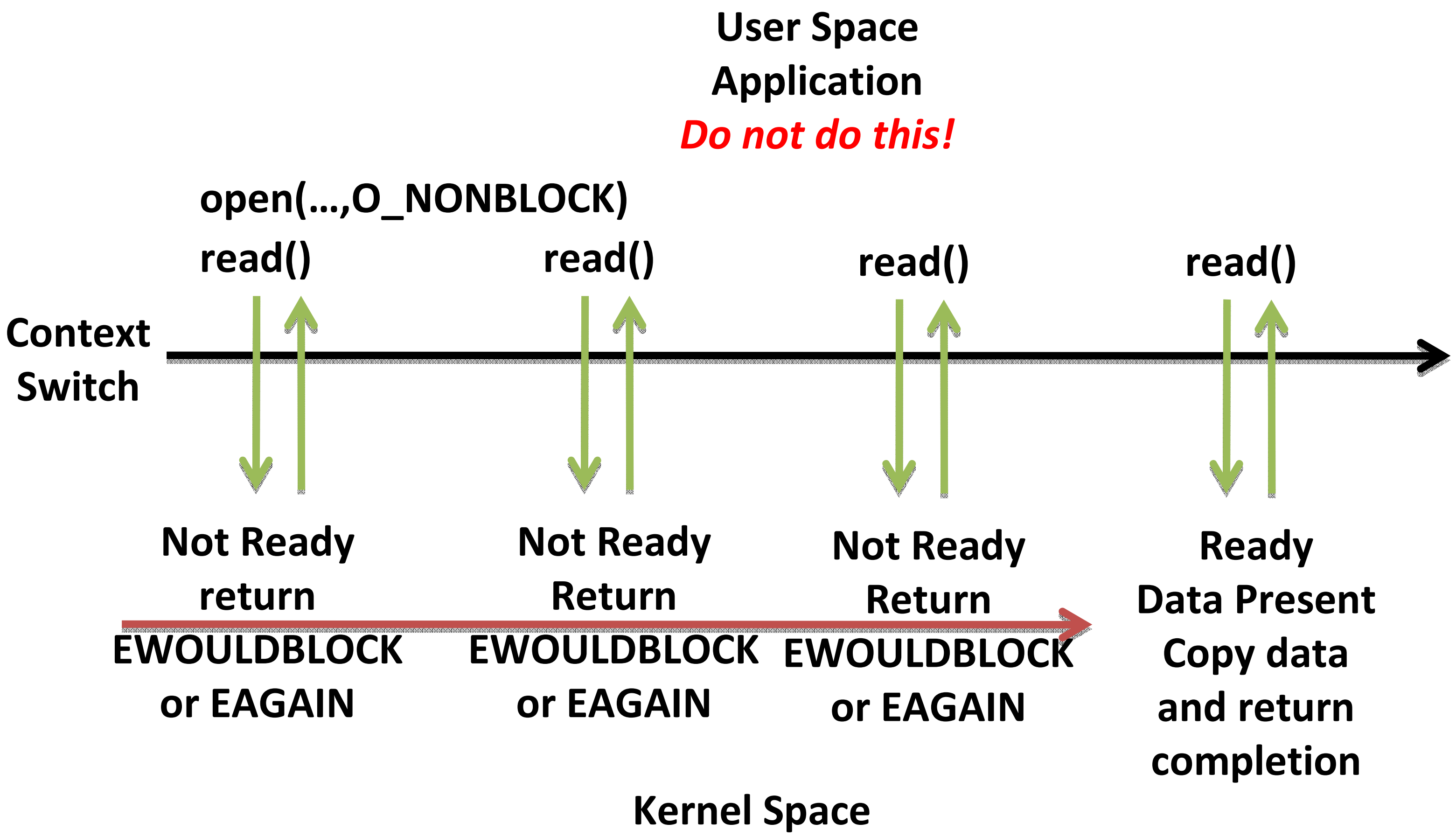
Blocking vs Non-Blocking I/O

- A device is considered blocking if it toggles between ready and not ready states
  - Read: no data present so not ready
  - Write: data cannot be accepted so not ready
    - I/O to a not ready device blocks the process
    - I/O to a ready device suspends the process in I/O wait
  - Reads and writes either complete to the extent possible, never start, or end with an error
- Devices that are always ready are non-blocking
  - Reads and writes either fully complete, never start, or end with an error

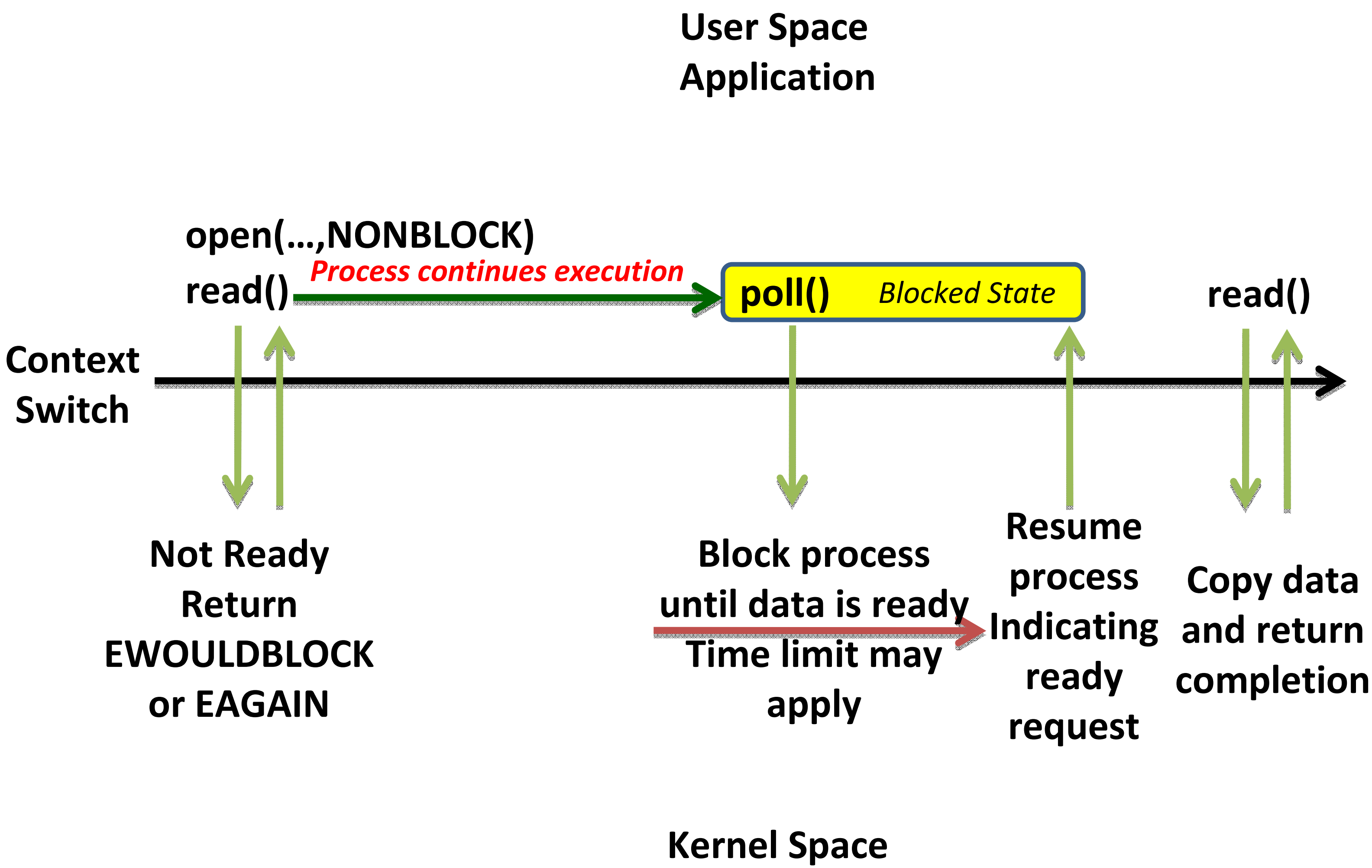
Blocking I/O


User Space
Application
open()
read()
Check if OK
Context
Switch
Not Ready
Block Process
Ready
Data Present
Copy data
and return
completion
Kernel Space

Blocking I/O Without Blocking I



Blocking I/O Without Blocking II



Blocking Devices

- Simulated and network devices are blocking
  - Pipes, fifo’s, sockets, streams, and terminals
- I/O occurs when device is ready
  - Process may or may not block as per **open()** options
- Not all requested data may be read or written
  - Reads transfer data that is immediately available
    - Many times this is less than what was requested
  - Writes transfer data until device becomes not ready
    - Usually because some resource becomes unavailable

Non-Blocking Devices

- By definition, disks are always ready
  - So, they are non-blocking
    - Implies I/O to regular files should be non-blocking
- However, I/O occurs through a file system
  - POSIX compliant file systems are non-blocking
    - They must adhere to the non-blocking nature of disks
  - Not all file systems are POSIX compliant
    - Typically, network based ones may not be fully compliant

Implications

- File system I/O might not be fully non-blocking
  - While relatively rare, this may happen
    - Usually in the area of incomplete I/O requests
- Something to worry about?
  - Usually not with commonly used file systems
    - But, easy to program around
- This section concentrates on non-blocking I/O
  - With accommodations for blocking devices

Starting With Open

- **open()** opens *any* Unix named device
  - Normally, files but can be FIFO's and pipes
    - As long as it has a file system path it's OK
- Always returns and integer
  - File descriptor or -1 on error
    - Check **errno** variable for actual reason when -1
- Many options exist
  - We will cover the more important ones

The Open API

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags,
        [mode_t mode]);
```

mode required
if ***flags*** contain **O_CREAT**

Commonly Used Open Flags

- **O_RDONLY**, **O_WRONLY**, or **O_RDWR**
  – How the file will be accessed
- **O_CREAT**, **O_EXCL**, and **O_TRUNC**
  – File creation disposition
- Read man page for the gory details

Esoteric Open Flags

- **O_NOATIME** (Since Linux 2.6.8)
  - Linux specific, don't update access time in the inode.
    - Can significantly improve performance for *some* applications
- **O_CLOEXEC** (Since Linux 2.6.23)
  - Linux specific, but important threading flag!
- **O_DIRECT** (Since Linux 2.4.10)
  - Generic, bypass file system cache
    - Can significantly improve performance in *isolated* cases
    - Not supported by all file systems and *may* return error if specified
- **O_NONBLOCK**
  - Enable non-blocking I/O
- **O_SYNC**
  - Make sure data written to disk before returning

Obsolete Open Flags

- **O_LARGEFILE**
  - Obsolete for 64-bit systems
    - **CC -D_FILE_OFFSET_BITS=64** preferred
- **O_NDELAY**
  - Obsolete in POSIX conforming systems
    - **O_NONBLOCK** preferred
      - **O_NDELAY** causes read/write to return 0 if blocked

        POSIX defines -1 with **EWOULDBLOCK**

Basic I/O API Parameter Types

- **ssize_t** is *signed*
  - This way -1 can be returned to indicate error
- **size_t** is *unsigned*
  - Maximum size defined by **SSIZE_MAX**
    - 2^{31}-1 for 32 bit architectures (2147483647)
    - 2^{63}-1 for 64 bit architectures (9223372036854775807)
- **off_t** is *signed* (Historical reasons)
- All automatically defined as 32 or 64 bits
  - Depending on target architecture

Read Peculiarities

- Returns bytes read or -1
  - Bytes read can be 0 to amount wanted
    - 0 → end of file for regular files o/w nothing available
    - When less than requested → all that is available
- -1 indicates error
  - Check **errno** variable for actual value
    - The most common ones are
      - **EINTR** call interrupted by a signal, nothing read
      - **EWOULDBLOCK** or **EAGAIN** for non-blocking I/O
        - » You will rarely program non-blocking I/O

Bullet Proof Read

```
ssize_t rc;
   do {rc = read(fd, buff, blen);}
      while(rc < 0 && EINTR == errno);
   if (rc < 0) {handle error}
```

Regular POSIX Files — **Other Devices**

```
ssize_t rc;
do{do {rc = read(fd, buff, blen);}
      while(rc < 0 && EINTR == errno);
   if (rc < 0) {handle error}
   if (!rc)    {handle EOF (e.g., ^D)}
   blen -= rc; buff += rc;
  } while(blen > 0);
```

Write Peculiarities

- Returns bytes written or -1
  - Bytes written can be 0 to amount wanted
    - When less than requested $\rightarrow$ all that could be written
- -1 indicates error
  - Check **errno** variable for actual value
    - The most common ones are
      - **EINTR** call interrupted by a signal
      - **EWOULDBLOCK** or **EAGAIN** for non-blocking I/O
        - » You will rarely program non-blocking I/O
      - **ENOSPC** for regular files

Bullet Proof Write

```
ssize_t rc;
   do {rc = write(fd, buff, blen);}
      while(rc < 0 && EINTR == errno);
   if (rc < 0) {handle error}
```

Regular POSIX Files

Other Devices

```
ssize_t rc;
do{do {rc = write(fd, buff, blen);}
      while(rc < 0 && EINTR == errno);
   if (rc < 0) {handle error}

   blen -= rc; buff += rc;
  } while(blen > 0);
```

read/write vs pread/write

- **read()** and **write()** use the current offset
  - Maintained per file pointer per process
  - Incremented on each **read()** and **write()**
  - Can use **lseek()** to change it
- This is difficult for multi-threaded apps
  - Especially ones sharing the same file pointer
- **pread()** and **pwrite()** solve this problem
  - You specify the offset on each invocation
    - Does not affect the current file offset pointer

lseek() & write() vs pwrite()

```
lseek(fd, offset, SEEK_SET);
do {rc = write(fd, buff, blen);}
   while(rc < 0 && EINTR == errno);
if (rc < 0) {handle error}
```

logically equivalent to

```
do {rc = pwrite(fd,buff,blen,offset);}
   while(rc < 0 && EINTR == errno);
if (rc < 0) {handle error}
```

In practice, these are *not equivalent* in multi-threaded applications
if the underlying file is referenced by more than one thread!

read/write vs readv/writev

- **read()** and **write()** only reference single buffer
- **readv()** and **writev()** reference one or more
- Use the latter to efficiently scatter/gather data
  - Better than multiple read/write calls
- Note OS's have limits on the number of buffers
  - IOV_MAX defines the limit
    - E.g., 1024 for Linux but 16 for Solaris

Performance Options

- API's themselves offer no performance options
- How you use them matters
  - E.g., using readv/writev when appropriate
- Only one practical possibility
  - Page aligned buffers
    - Allows some file system to use copy on write
    - Avoids extra page reference
      - Need not be page aligned; merely aligned within a page
    - Always required if you use **O_DIRECT** open flag

Page Aligning Buffers

- **`int posix_memalign(`**

  `void **`*`memptr`*

  `size_t `*`alignment,`*

  `size_t `*`size`*`);`

  – On success, zero returned with . . .
    - *memptr* holding pointer to allocated memory
      – Will be at least the size of *size* and start at an address that is a multiple of *alignment* which must be a power of 2 and here should be the page size.

  – On failure, **errno** value is returned
    - **errno** variable is not set

posix_memalign() Issues

- Not all platforms support **posix_memalign()**
  - Linux supports it since glibc 2.1.91
    - Use **-D_GNU_SOURCE** or **-D_XOPEN_SOURCE=600**
- Most systems support **memalign()**
  - `void *memalign(size_t boundary, size_t size);`
  - Not necessarily equivalent
    - Area allocated *might* not be used with **free()**
      - Not true if you use **glibc** (i.e., g++ or gcc)

posix_memalign() Example

```
#include <stdlib.h>
#include <unistd.h>

static int PageSize = sysconf(_SC_PAGESIZE);
void *Buff;

if ((rc=posix_memalign(&Buff, PageSize, length)) < 0)
   {handle error}
```

logically equivalent to

```
#include <stdlib.h>
void *Buff;

if (!(Buff=memalign(sysconf(_SC_PAGESIZE), length)))
   {handle error}
```

Alignment Within A Page

```
#include <stdlib.h>
#include <unistd.h>

static size_t PageSize  = sysconf(_SC_PAGESIZE);
       size_t Alignment = PageSize;
       void  *Buff;

if (length < Alignment)
   {do {Alignment = Alignment >> 1;}
       while(length < Alignment);
    Alignment = Alignment << 1; length = Alignment;
   }

if (posix_memalign((void **)&Buff, Alignment, length))
   {handle error}
```

Conclusions

- Basic I/O API's work with any device
  - These are the workhorses you will usually use
- Few optimizations available
  - Using readv/writev where appropriate
  - Page aligning frequently used buffers
    - May be be required in some cases
      - E.g. O_DIRECT open option