Andrew Hanushevsky: Asynchronous I/O

Goals

- Explain the usefulness of asynchronous I/O
  – Indicate where is should and should not b used
- Explain AIO API's
  – Provide common examples
  – Explain how to do I/O to multiple devices
- Provide reasonable AIO alternatives

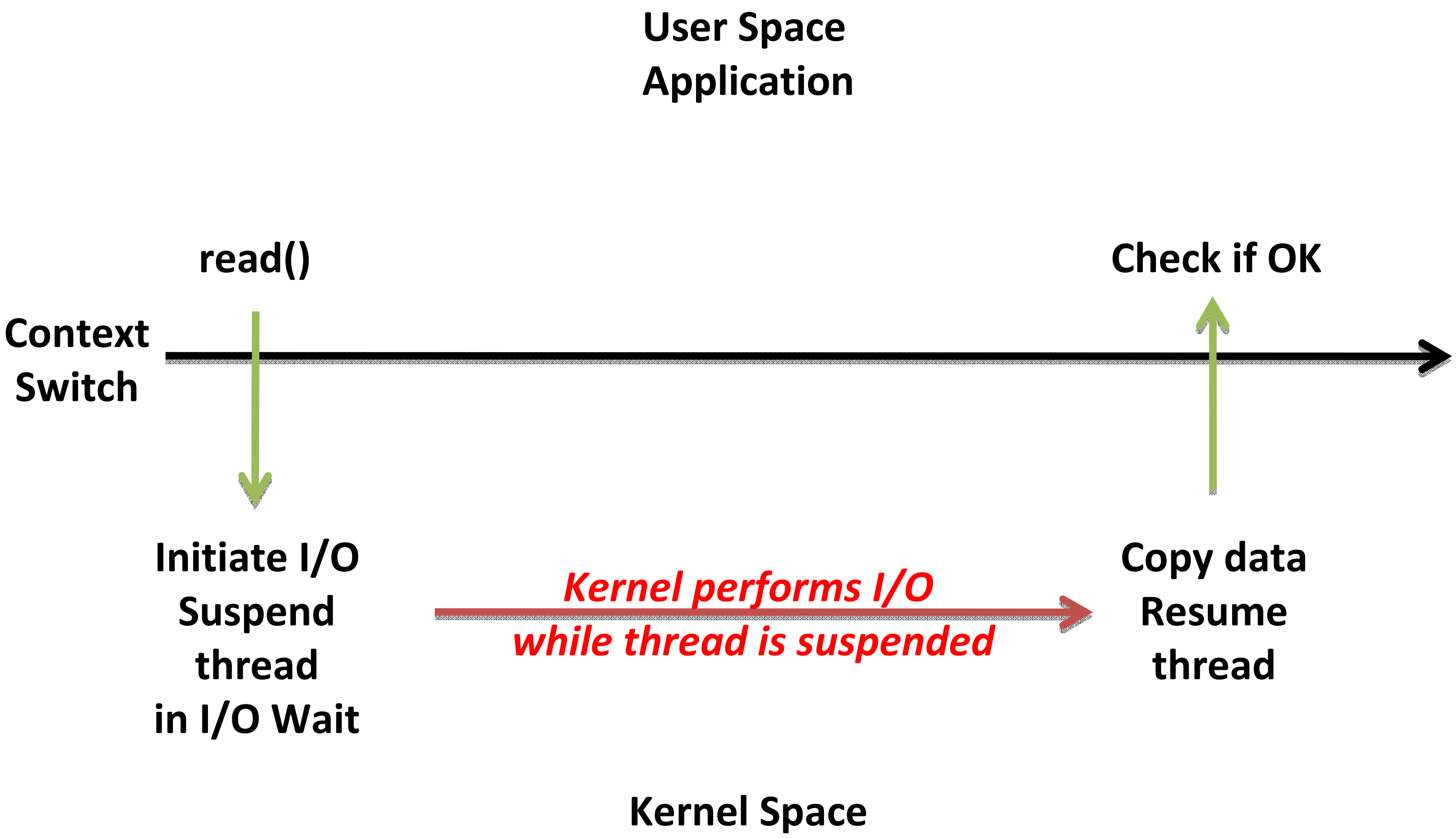
Asynchronous I/O

- I/O that occurs in “parallel” with the requestor
  - Set of OS interfaces similar to synchronous I/O
    - Read, write, sync
  - Set of OS interfaces to manage I/O & test completion
    - No synchronous counterparts
  - Not all OS’s implement interface
    - Linux 2.6 does
      - Conforms to POSIX.1-2001
    - GNU C defines the interface
      - Conforms to ISO/IEC 9945-1:1996

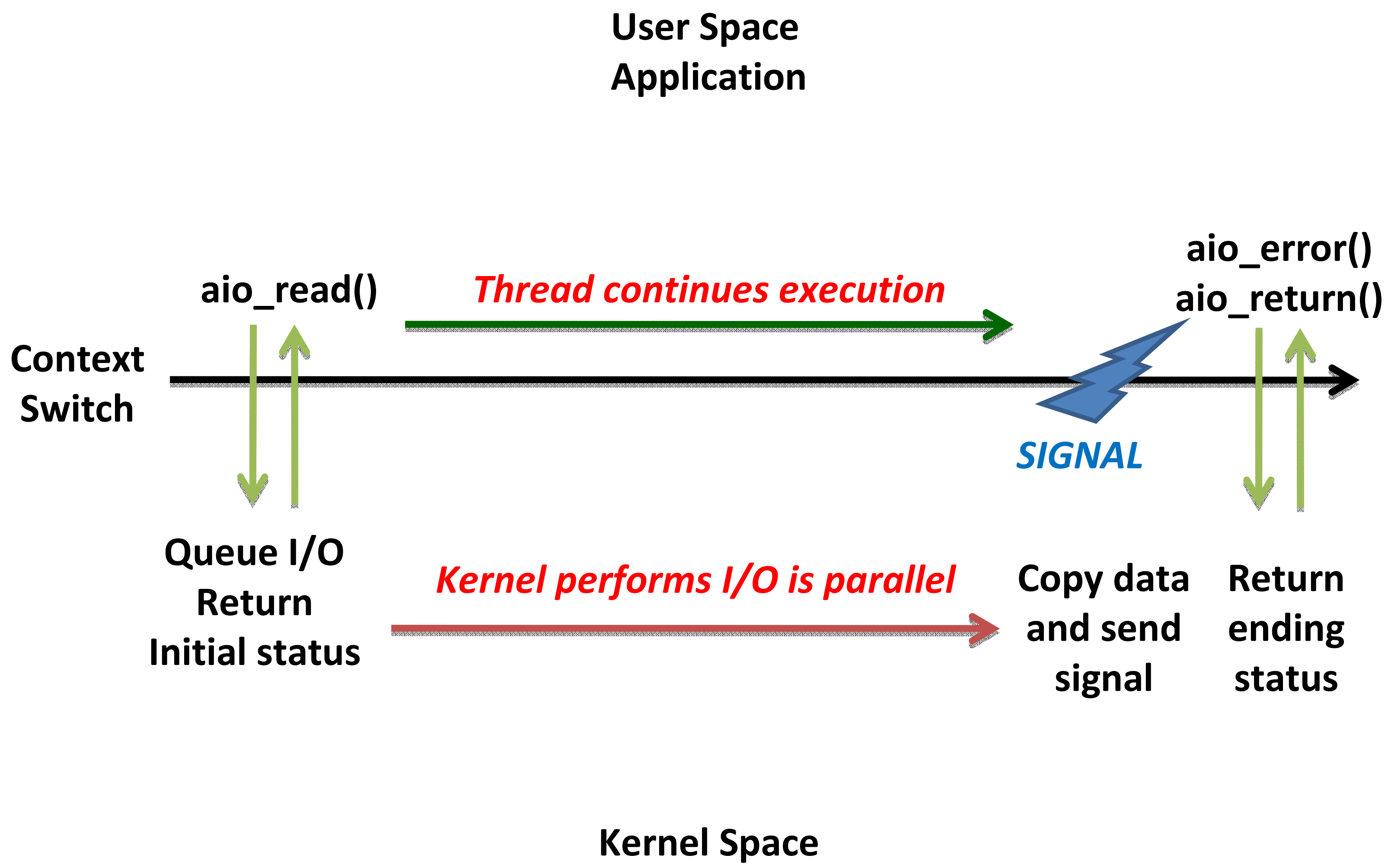
Synchronous vs Asynchronous

- Synchronous I/O is deterministic
  - Thread is suspended from the time an I/O request is issued to the time it completes.
- Asynchronous I/O is non-deterministic
  - Thread continues to run after the I/O request
    - Kernel does the I/O in parallel to process execution
  - Thread is responsible for checking completion
    - Can ask the kernel for a signal when I/O completes

Synchronous I/O



Asynchronous I/O



Implications

- Only sync I/O for blocking devices is sensible
  - Can use non-blocking option to prevent stalls
    - Parallelism is a manual programming process
  - But no I/O can occur until device is unblocked
- Sync or Async I/O to non-blocking devices OK
  - Using non-blocking options makes no sense
    - Though you are allowed to do so
- We will only discuss async non-blocking I/O

The AIO Interface

- **aio_read()**, **aio_write()**, **aio_fsync()**
  - Start a read, write, or fsync() operation
- **aio_cancel()**
  - Cancel a previous read, write, or fsync
- **aio_error()**
  - Check request's progress
- **aio_return()**
  - Get final status of completed request (use only once!)
- **aio_suspend()**
  - Wait for request completion

The Common AIO Element

- The **aiocb** structure correlates all requests
  - Defined in **aio.h**

```
struct aiocb
{
  int aio_fildes;                /* File descriptor.  */
  int aio_lio_opcode;            /* lio Operation. */
  int aio_reqprio;               /* Request priority offset.  */
  volatile void *aio_buf;        /* Location of buffer.  */
  size_t aio_nbytes;             /* Length of transfer.  */
  struct sigevent aio_sigevent; /* Signal number and value.  */
  off_t aio_offset;              /* File offset.  */
  .
  .                              /* Additional fields */
  .
};
```

Simplistic AIO Read Example

```
#include <aio.h>
•••
  int fd, rc, ret;
  struct aiocb my_aiocb;

  if ((fd = open( "my_file", O_RDONLY )) < 0) {handle error}

  memset((char *)&my_aiocb, 0, sizeof(my_aiocb)); //  Always zero it out!

  /* Allocate a data buffer for the aiocb request */
  if (!(my_aiocb.aio_buf = malloc(BUFSIZE))) {handle error}

  /* Initialize the necessary fields in the aiocb */
  my_aiocb.aio_fildes = fd;
  my_aiocb.aio_nbytes = BUFSIZE;
  my_aiocb.aio_offset = 0;

  if ((rc = aio_read( &my_aiocb )) < 0) {handle error}

  while((rc = aio_error( &my_aiocb )) == EINPROGRESS ) {};

  if ((ret = aio_return( &my_aiocb )) >= 0) {
    /* got ret bytes on the read */
  } else {
    /* read failed, rc is the errno value but errno is now set as well */
  }
```

Some Warnings!

- After **aio_read()**, do *not* change. . .
  - Any byte of the **aiocb** structure
  - The buffer passed via the **aiocb**
    - It must remain valid as well (i.e., no **free** or **munmap**)
- Failure to do so yields unpredictable results
- You may change memory *after* **aio_return()**
  - Which may one be called once!

Simplistic Approach is Bad!

- Example is essentially sync/non-blocking
  - A CPU eater and to always be avoided
- We can convert it to async/blocking
  - Much better but not particularly useful

```
struct aiocb *cblist[] = {&my_aiocb, 0, . .};

if ((rc = aio_read( &my_aiocb )) < 0) {handle error}

If ((rc = aio_suspend(cblist, 1, NULL))) {handle error}

while((rc = aio_error( &my_aiocb )) == EINPROGRESS ) {};

if ((ret = aio_return( &my_aiocb )) >= 0) {
    /* got ret bytes on the read */
   } else {
    /* read failed, rc has errno value and errno is now set too */
   }
```

Other Issues With aio_suspend()

- **aio_suspend()** can wait on *n* aiocb's
- Successful completion indicated by 0 return
  - Means *one or more* of the **aiocb**'s completed
  - You must now poll each one to find out which
    - Use **aio_error()**
  - This simply delays context switches
- Waiting on more than one is problematic

Cancelling AIO Requests

- **`aio_cancel(int fd, struct aiocb *aiocbp)`**
  - To cancel a particular request supply **fd** & **aiocbp**
  - To cancel all requests for an **fd** set **aiocbp** to zero
- Returns
  - **AIO_CANCELED** if all were cancelled
  - **AIO_NOTCANCELED** if at least one was not
  - **AIO_ALLDONE** if all completed already
  - -1 with **errno** for **aio_cancel()** call errors

Handling Multiple Requests

- **lio_listio()** can initiate a number of requests
  - `int lio_listio(int mode, struct aiocb *list[], int nent, struct sigevent *sig);`
  - mode
    - **LIO_WAIT** – wait until everything completes
    - **LIO_NOWAIT** – return once **aiocb**’s queued
  - nent
    - The number of **aiocb**’s in the **list[]**
  - sig
    - Defines signal notification for **LIO_NOWAIT**

lio_listio Details

```
struct aiocb aiocb1, aiocb2;
struct aiocb *list[2] = {&aiocb1, &aiocb2};
●●●
/* Prepare the first aiocb */
aiocb1.aio_fildes = fd;
aiocb1.aio_buf = malloc(BUFSIZE);
aiocb1.aio_nbytes = BUFSIZE;
aiocb1.aio_offset = next_offset;
aiocb1.aio_lio_opcode = LIO_READ; // Can be LIO_READ, LIO_WRITE, and LIO_NOP
●●●
ret = lio_listio(LIO_WAIT, list, 2, NULL);
```

```
struct aiocb aiocb1, aiocb2;
struct aiocb *list[2] = {&aiocb1, &aiocb2};
Struct aiocl {int num; struct aiocb *list;} aioList = {2, list};
struct sigevent aio_sigevent;
●●●
/* Prepare the first aiocb */
●●●
aio_sigevent.sigev_notify = SIGEV_SIGNAL;
aio_sigevent.sigev_signo = innocuous_signum;
aio_sigevent.sigev_value.sival_ptr = &aioList;;
●●●
ret = lio_listio(LIO_NOWAIT, list, 2, &aio_sigevent);
```

The Good Part of lio_listio()

- **lio_listio()** allows you to do a number of things
  - Start I/O on a number of different files
  - Start I/O on a number of different offsets
- All this is done in one system call
- If you need multi-faceted I/O this is it
  - Even with **LIO_WAIT** it's very effective
- But waiting for single aio requests is bad
  - Defeats the whole purpose of async I/O
    - Unfortunately, most aio requests are singletons

The Right AIO Approach

- To make AIO truly async we must use signals
  - They notify us when a request is completed
    - And, optionally, which **aiocb** completed
  - Means setting up a signal handler
  - Means setting up a request queue manager
    - Will handle completed requests out of signal handler
  - Multi-threading is the best model for this

AIO With Signals

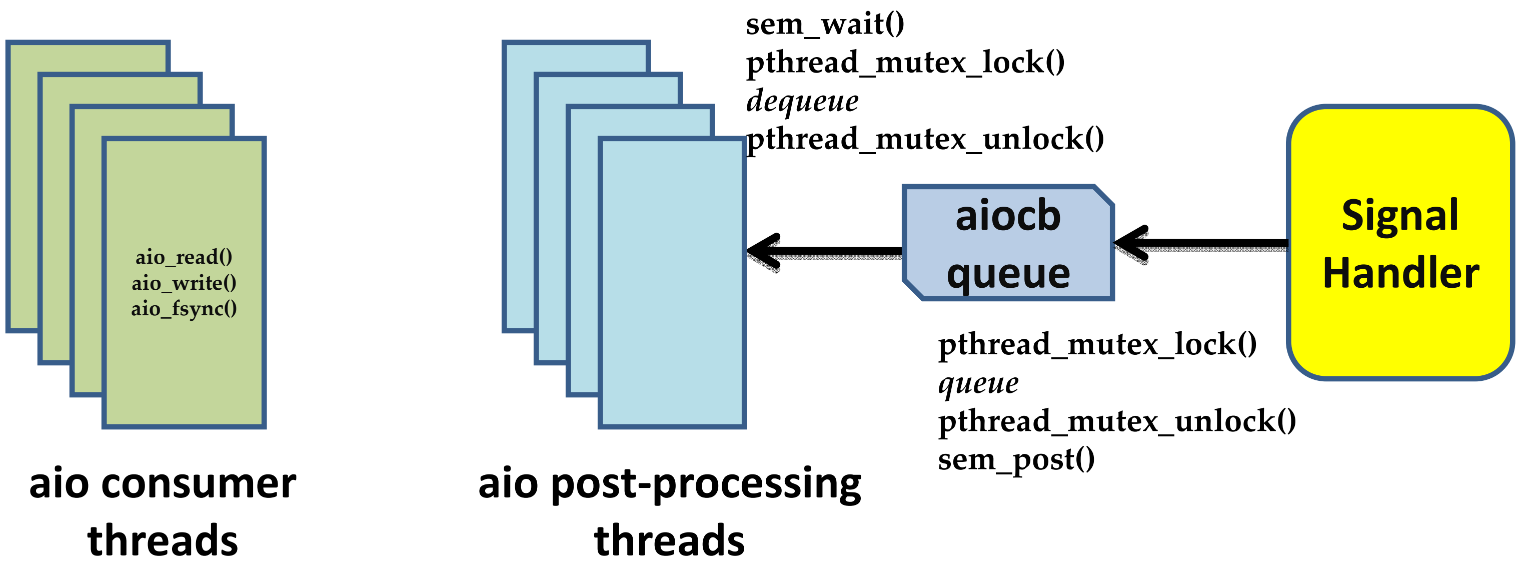
```
•••
struct sigaction sa;

sa.sa_sigaction = mySigHadler;
sa.sa_flags = SA_SIGINFO;
sigemptyset(&sa.sa_mask);
if (sigaction(innocuous_signum, &sa, NULL) < 0) {handle error}
•••
my_aiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
my_aiocb.aio_sigevent.sigev_signo = innocuous_signum;
my_aiocb.aio_sigevent.sigev_value.sival_ptr = &my_aiocb;

if ((rc = aio_read( &my_aiocb )) < 0) {handle error}

Off to do other things while I/O occurs and notification sent!

void mySigHandler(int signo, siginfo_t *info, void *context)
{  struct aiocb *req;

 if (info->si_signo == innocuous_signum && info->si_code == SI_ASYNCIO)
    {req = (struct aiocb *)info->si_value.sival_ptr;
/*
While we could do aio_error() and aio_return() here; a workable solution
requires that we queue this aiocb on a completion queue so that some other
thread can handle the post-processing which is usually too complex to be
done inside a signal handler (e.g., like more I/O).
*/
     }
}
```

What You Will Find In Practice

- You will need to embed aiocb in an object
  - The object can be used to coordinate requests
    - E.g., queuing and callbacks
    - The callback can do the **aio_error()** and **aio_return()**
      - It can also reflect the completion to the requestor

```
class aioRequest
{public:
 aioRequest   *Next;       // For queuing purposes
 struct aiocb  theAiocb;   // The actual request
•
•
•
 void          CallBack(); //Invoked when aiocb completes
•
•
•
};
```

A Workable Picture



Warning: **Not all platforms implement sem_xxx functions.**

Devil In The Details I

- What innocuous signal number to choose?
  - Real time signals are preferred
    - One between **SIGRTMIN** and **SIGRTMAX**
      - Defined if real time signals are supported
  - Otherwise, choose **SIGUSR1** or **SIGUSR2**
- AIO is not supported on all platforms
  - **_POSIX_ASYNCHRONOUS_IO** defined by **gcc** if present
- **sigwaitinfo()** is not present on all platforms
  - Though that is getting less so
    - It can be emulated but that is not straightforward

Devil In The Details II

- There are limits to the number of active AIO's
  - Linux supports a system limit
    - /proc/sys/fs/aio-max-nr (max number usually 64K)
    - /proc/sys/fs/aio-nr (number currently active)
  - Other platforms impose per process limits
    - Refer to the platform's **getrlimit()** and **sysconf()**
- AIO requests can fail if the limit is exceeded
  - Be prepared to revert to synchronous processing
    - Usually will get **EAGAIN** error on an aio request

Avoiding Signals

- You can automatically start a thread
  - sigev_notify = **SIGEV_THREAD**
  - sigev_notify_function = void (*func)(union sigval)
- Practical problems. . .
  - Not all platforms support this notification
  - Ill-defined actions when thread limit exceeded
  - Relatively heavy-duty for a simple notification
    - Though it makes programming easier
- Generally, I do not recommend using this

What's The Alternative?

- A multi-threaded I/O architecture can work
  - aio defined before threading became pervasive
- Implemented as a consumer/producer model
  - One or two dozen producer threads are sufficient
    - Can be dynamically created as needed
  - Producers use well established sync interfaces
  - Consumers see an asynchronous interface
  - The kernel works just as hard
- Better yet, use parallelizable algorithms

Conclusions

- AIO is a powerful performance technique
  - But historically geared to non-threaded event loops
  - Difficult to use and is error prone
  - Of these **lio_listio()** has the greatest potential
    - Consider using this for multiple disparate I/O requests
- Better alternative is to use multi-threading
  - Must use algorithms amenable to parallelism
  - Using synchronous interfaces only suspends thread
    - Computation still continues in other threads