System and Device Programming

Asynchronous I/O

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

Synchronous I/O

- All standard I/O operations are synchronous
  - I/O is **blocking** and the task waits until the I/O operation completes
  - Select delivers a synchronous form of notification
- Unfortunately, synchronous operations are inherently **slow** compared to other processing
  - Delays may be caused by
    - Hardware device, e.g., track and sector seek time on random access, etc.
    - Relatively slow data transfer rate between a physical device and the system memory
    - Network transfer using file servers, storage area networks, etc.

Asynchronous I/O

- Threads can perform asynchronous I/O
  - A task can continue **without** waiting for an I/O operation to complete
- There are two techniques
  - Multithread I/O
  - POSIX asynchronous I/O

Asynchronous I/O

- Multithread I/O
  - We use multiple threads
  - Each thread within a process (or in different processes) may use a synchronous model
    - Each thread is responsible for a sequence of one or more synchronous, **blocking** I/O operations
    - Each thread should have its own file or pipe handle
  - **Other** threads can **continue** execution
    - The threads run asynchronous to each other
  - This is the most general technique

Asynchronous I/O

- Asynchronous I/O
  - We incur additional complexity when we use the POSIX asynchronous I/O interfaces
    - We have to worry about sources of errors for every asynchronous operation
    - The interfaces involve a lot of extra setup and processing rules compared to their conventional counterparts
    - Recovering from errors can be difficult
      - For example, if we submit multiple asynchronous writes and one fails, how should we proceed? If the writes are related, we might have to undo the ones that succeeded

The aiocb data structure

- POSIX gives us a consistent way to perform asynchronous I/O, regardless of the type of file
- The interfaces use AIO control blocks to describe I/O operations
  - The aiocb structure defines an AIO control block
  - It contains at least the fields shown in the following structure (implementations might include additional fields)
  - Note
    - Insert the <aio.h> library in the C file
    - Compile the C file with the realtime library (librt.a), i.e., -lrt

The aiocb data structure

```
#include <aio.h>

struct aiocb {
  int aio_fildes;
  off_t aio_offset;
  volatile void *aio_buf;
  size_t aio_nbytes;
  int aio_reqprio;
  struct sigevent aio_sigevent;
  int aio_lio_opcode;
};
```

- Parameters
  - The **aio_fildes** field is the file descriptor open for the file to be read or written

The aiocb data structure

- Read or writes start at the offset specified by **aio_offset**
  - For a read Data is copied to the buffer that begins at the address specified by aio_buf
  - For a write, data is copied from this buffer
- The **aio_nbytes** field contains the number of bytes to read or write

```
struct aiocb {
  int aio_fildes;
  off_t aio_offset;
  volatile void *aio_buf;
  size_t aio_nbytes;
  int aio_reqprio;
  struct sigevent aio_sigevent;
  int aio_lio_opcode;
};
```

The aiocb data structure

- The **aio_reqprio** field is a hint that gives applications a way to suggest an ordering for the asynchronous I/O requests
  - The system has only limited control over the exact ordering, however, so there is no guarantee that the hint will be honored

```
struct aiocb {
  int aio_fildes;
  off_t aio_offset;
  volatile void *aio_buf;
  size_t aio_nbytes;
  int aio_reqprio;
  struct sigevent aio_sigevent;
  int aio_lio_opcode;
};
```

The aiocb data structure

- The **aio_sigevent** field controls how the application is notified about the completion of the I/O event
  - It is described by a sigevent structure

```
struct sigevent {
  int sigev_notify;
  int sigev_signo;
  union sigval sigev_value;
  void (*sigev_notify_function)(union sigval);
  pthread_attr_t *sigev_notify_attributes;
};
```

The aiocb data structure

- The sigev_notify field controls the type of notification
  - SIGEV_NONE the process is not notified when the asynchronous I/O request completes
  - SIGEV_SIGNAL the signal specified by the sigev_signo field is generated when the asynchronous I/O request completes
  - SIGEV_THREAD the function specified by the sigev_notify_function field is called when the asynchronous I/O request completes
    address of a pthread attribute

```
struct sigevent {
  int sigev_notify;
  int sigev_signo;
  union sigval sigev_value;
  void (*sigev_notify_function)(union sigval);
  pthread_attr_t *sigev_notify_attributes;
};
```

The aiocb data structure

- The **aio_reqprio** field is a hint that gives applications a way to suggest an ordering for the asynchronous I/O requests
  - The system has only limited control over the exact ordering, however, so there is no guarantee that the hint will be honored

```
struct aiocb {
  int aio_fildes;
  off_t aio_offset;
  volatile void *aio_buf;
  size_t aio_nbytes;
  int aio_reqprio;
  struct sigevent aio_sigevent;
  int aio_lio_opcode;
};
```

Read and Write

```
#include <aio.h>

int aio_read(struct aiocb *aiocb);
int aio_write(struct aiocb *aiocb);
```

- To perform asynchronous I/O, we need to
  - Initialize an AIO control block
  - Call either **aio_read** or the **aio_write**
- When these functions return success, the asynchronous I/O request has been queued for processing by the operating system
  - The return value has no relation to the result of the actual I/O operation

Guidelines

- While the I/O operation is pending, we have to be careful to ensure that the AIO control block and data buffer remain stable
  - Their underlying memory must remain valid and we cannot reuse them until the I/O operation completes

```
int aio_read(struct aiocb *aiocb);
int aio_write(struct aiocb *aiocb);
```

Synchronization

```
#include <aio.h>

int aio_fsync (int op, struct aiocb *aiocb);
```

- We can use this function to force all pending asynchronous writes to persistent storage without waiting
- The aio_fsync operation returns when the synch is scheduled
  - The data will not be persistent until the asynchronous synch completes
  - The AIO control block controls how we are notified

Synchronization

- The aio_fildes field in the AIO control block indicates the file whose asynchronous writes are synched
- If the op argument is set to
  - O_DSYNC, then the operation behaves like a call to fdatasync
  - O_SYNC, the operation behaves like a call to fsync

```
int aio_fsync (int op, struct aiocb *aiocb);
```

Suspension

```
#include <aio.h>

int aio_suspend(const struct aiocb *const list[],
  int nent, const struct timespec *timeout);
```

- We use asynchronous I/O when we have other processing to do and we don't want to block while performing the I/O operation
- However, when we have completed the processing and find that we still have asynchronous operations outstanding, we can call the aio_suspend function to block until an operation completes

Cancel

```
#include <aio.h>

int aio_cancel (int fd, struct aiocb *aiocb);
```

- When we have pending asynchronous I/O operations that we no longer want to complete, we can attempt to cancel them with the aio_cancel function