SOP1

Asynchronous Input/Output

Introduction to AIO

- Parallel I/O operations:
  - speed up processing
    - process does not wait for the operation to complete
    - several I/O operations can be conducted in parallel
    - allow a process or thread to overlap some parts of computation with I/O processing.
  - limits:
    - works on real file descriptors (low level I/O, not pipes nor sockets on Linux)
    - not all programs can benefit form aio (algorithm, small < 10MB read/write buffers)
  - require Real Time library (`-lrt`)

Introduction to AIO

- Although POSIX defines interface for AIO, support is not mandatory in compliant systems
- AIO is available if `_POSIX_ASYNCHRONOUS_IO` is defined in `<unistd.h>`
- AIO is not 100% portable between POSIX systems, if not available program must supply its own implementation or change its logic
- AIO is implemented on most (if not all) POSIX compliant systems and we assume it to be portable enough for laboratory purposes

Introduction to AIO

- ```
  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;
      ...
  };
  ```
  ```

Introduction to AIO

- ```
  union sigval {
        int     sival_int;
        void   *sival_ptr;  };
  ```

- ```
  struct sigevent {
      int      sigev_notify;
      /*SIGEV_NONE,SIGEV_SIGNAL,SIGEV_THREAD*/
      int      sigev_signo;
      union sigval sigev_value;
      void (*sigev_notify_function) (union sigval);
      void    *sigev_notify_attributes;
      pid_t    sigev_notify_thread_id;   };
  ```

Introduction to AIO

- All parallel operations are controlled with use of the following AIO Control Block structure (`struct aiocb`)
  - `aio_fildes` – opened file descriptor
  - `aio_offset` – position in the file to start the operation
    - is not valid for writing if O_APPEND flag is set on opened descriptor
    - offset cannot exceed maximum file offset, but process can write after the end of file (like f. `write`)
  - `aio_buf` – the operation buffer (read/write)
    - buffer should not be accessed or deallocated during operation

Introduction to AIO

- **`aiocb`:** (cont.)
  - `aio_nbytes` – the size of the buffer/requested number of bytes to read or write
  - `aio_reqprio` – i/o operation priority
  - `aio_sigevent` – how to notify caller about the I/O operation termination, structure:
    - `sigev_notify` (requires aio_fsync to work)
      - *SIGEV_NONE* – no notification
      - *SSIGEV_SIGNAL* – signal is sent (`sigev_signo`)
      - *SIGEV_THREAD* – specified function is stared as separate **detached** thread: `sigev_notify_function`,`sigev_notify_attributes`

Introduction to AIO

- **`aiocb`:** (cont.)
  - `aio_lio_opcode` – for f. `lio_listio` only:
    - *LIO_READ,LIO_WRITE*
    - *LIO_NOP* – to store unused blocks on the list
  - it is advisable to clear the `aiocb` structure before use (f. `memset`)
  - the structure must remain unchanged during aio operation, and after until caller checks the operation status and possible errors
  - one aiocb structure cannot be used to schedule two or more aio operations!!!

Introduction to AIO

- AIO can be implemented in kernel or on threads (linux), both approaches are POSIX compliant
- AIO operations cannot be interrupted by signal handling routine. This is not obvious only for thread implementation. The thread controlling IO operation is not handling any signals, thus EINTR is impossible to observe

Asynchronous Read/Write

- **`int aio_read(struct aiocb *aiocbp);`**
  - schedules single I/O read operation and returns immediately
  - function is equivalent to synchronous:

```
read(p->aio_fildes,p->aio_buf,p->aio_nbytes);
```

  - reading starts at absolute position `aiocbp->aio_offset` regardless of current file position
  - multiple read operations on the same file part are possible

Asynchronous Read/Write

- **`int aio_write(struct aiocb *aiocbp);`**
  - start single I/O write operation and returns immediately
  - function is equivalent to synchronous:

```
write(p->aio_fildes,p->aio_buf,p->aio_nbytes);
```

  - O_APPEND
    - if bit is not set writing starts at absolute position `aiocbp->aio_offset` regardless of current file position
    - if is set, data will be appended, if multiple append calls are scheduled the sequence of appending will reflect the calls order
  - multiple write operations on the same file part will produce undefined results

Asynchronous Read/Write

- **`aio_read aio_write`** functions:
  - `aiocbp->aio_lio_opcode` is ignored
  - can report some of the I/O errors synchronously
  - process will be informed about termination of the operation by polling, by signal or by thread start depending on `sigev_notify` value
  - file position after aio operation is unspecified according to POSIX

AIO - Examples

- How to schedule aio read (by J.Bartuszek) :

```
if ((buffers[index] = (char *)calloc(size + 1,
  sizeof(char))) == NULL)
        error("Cannot allocate memory");

aiocbs->aio_buf = (void *)buffers[index];
aiocbs->aio_offset = offset;
aiocbs->aio_nbytes = size;

if (aio_read(aiocbs) == -1)
        error("Cannot read");
```

Asynchronous Read/Write

- **`int lio_listio(int mode, struct aiocb *const aiocb_list[], int nitems, struct sigevent *sevp);`**
  - schedules multipe I/O read/write operations as if multiple `aio_read` and `aio_write` calls were issued
  - function requires an array of pointers to `aiocb` structures with `aio_lio_opcode` set, NULLs are ignored
  - order of operations is undefined
  - in case of failure or successes the caller **must examine all `aiocb` structures for errors !!!**

Asynchronous Read/Write

- **`lio_listio`** function (cont.):
  - mode:
    - *LIO_NOWAIT* function returns immediately after all operations are scheduled for execution, the termination will be notified according to `sig` parameter
      - NULL - no notification after termination of all operations
      - signal or a new thread (set like in `aiocb` structure)
      - **process still can receive signals after individual operations**
    - *LIO_WAIT* waits for termination of all scheduled operations
      - in case of error `aio_error` helps to determine which operation failed
      - if all operations are successful zero is returned
      - function may be interrupted [EINTR]

Status of AIO

- **`int aio_error(const struct aiocb *aiocbp);`**
  - determines the status of operation, can be called several times, be careful not to implement busy waiting.
    - *EINPROGRESS* - operation successfully queued but still unfinished
    - zero – operation is finished without errors
    - value that would be stored in `errno` for synchronous `read/write fsync/fdatasync`
      - `perror` will not work properly
  - always call `aio_error` before `aio_return` and not the other way round

Status of AIO

- **`ssize_t aio_return(struct aiocb *aiocbp);`**
  - returns what corresponding `read/write fsync` would return
  - do not expect operation to be interrupted by signal (no EINTR)
  - **can be called only once after aiocb stucture was used in `aio_read/aio_write` or `lio_listio` !!!**
  - **cannot be called as long as `aio_error` returns *EINPROGRESS*!!!**

AIO - Examples

- How to do **busy waiting** (from HP-UX man page):

```
/* wait for completion */
while ((retval = aio_error(&myaiocb)) ==
                                        EINPROGRESS);
/* free the aiocb */
retval = aio_return(&myaiocb);
```

Synchronization of AIO

- **`int aio_fsync(int op, struct aiocb *aiocbp);`**
  - schedules device synchronization **after operation completes**, just like calling `fsync` after `read/write`, does not wait for completion of operation
  - required in multiprocess programs that operate on the same files
  - errors are reported:
    - synchronously (return value + errno)
    - asynchronously by `aio_error`
  - op :
    - *O_SYNC* – `fsync` call
    - *O_DSYNC* - `fdatasync` call

Synchronization of AIO

- Main code can synchronise with AIO operations:
  - Synchronously(easiest buy may waste some CPU time)
    - **`aio_suspend`** function
  - Asynchronously (apply to some algorithms only)
    - Signals (efficient but notification handling may be complicated, what aio block finished?)
    - Threads (efficient and not complicated)

Synchronization of AIO

- **`int aio_suspend(const struct aiocb *const list[], int nent,const struct timespec *timeout);`**
  - suspends execution until **at least one** from the list of submitted `aiocb` points to finished operation
  - if any of operations on the list is finished by the time of call the function returns immediately – **be aware of busy waiting**
  - can be terminated before any operation successes by
    - signal handling routine (signal can be generated by termination of one of operations) - *EINTR*
    - by timeout (if not *NULL*) - *EAGAIN*

Synchronization of AIO

- **`aio_suspend`** function(cont.):
  - usually if `aio_suspend` is used signal notification is not needed and can be disturbing
  - ignores NULL in the array of pointers to `aiocb` structures , the same list as for `lio_listio` can be used
  - returns zero on successes (**not the number of terminated operations!**), list of operations should be checked with `aio_error`
  - manual (POSIX) does not specify if `aiocb` structures with `aio_lio_opcode` set to *LIO_NOP* are ignored

AIO - Examples

- How to synchronize aio (by J.Bartuszek) :

```
void suspend(struct aiocb *aiocbs) {
  struct aiocb *aiolist[1];
  aiolist[0] = aiocbs;
  while (aio_suspend((const struct aiocb *const *)
     aiolist, 1, NULL) == -1) {
        if (!work) return; /*signal received*/
        else if (errno == EINTR) continue;
        error("Error suspending");
  }
  if (aio_error(aiocbs) != 0)
        error("Async. I/O error");
  if ((status = aio_return(aiocbs)) == -1)
        error("Async. I/O return error");
}
```

AIO - Examples

- How to start thread with aio (from IBM website, code flaws in red) :

```
bzero( (char *)&my_aiocb, sizeof(struct aiocb) );
my_aiocb.aio_fildes = fd;
my_aiocb.aio_buf = malloc(BUF_SIZE+1);
my_aiocb.aio_nbytes = BUF_SIZE;
my_aiocb.aio_offset = next_offset;
my_aiocb.aio_sigevent.sigev_notify = SIGEV_THREAD;
my_aiocb.aio_sigevent.notify_function =
aio_completion_handler;
my_aiocb.aio_sigevent.notify_attributes = NULL;
my_aiocb.aio_sigevent.sigev_value.sival_ptr =
&my_aiocb;
ret = aio_read( &my_aiocb );
```

AIO - Examples

- How to start thread with aio (from IBM website, cont.) :

```
void aio_completion_handler( sigval_t sigval ){
  struct aiocb *req;
  req = (struct aiocb *)sigval.sival_ptr;
  if (aio_error( req ) == 0) {
    ret = aio_return( req );
  }
...
  return;
}
```

AIO - Examples

- How to send signal to synchronize with aio (from IBM website, code flaws in red) :

```
   sig_act.sa_flags = SA_SIGINFO;
   sig_act.sa_sigaction = aio_completion_handler;
   ret = sigaction( SIGIO, &sig_act, NULL );
...
bzero( (char *)&m_aiocb, sizeof(struct aiocb) );
m_aiocb.aio_fildes = fd;
m_aiocb.aio_buf = malloc(BUF_SIZE+1);
m_aiocb.aio_nbytes = BUF_SIZE;
m_aiocb.aio_offset = next_offset;
m_aiocb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
m_aiocb.aio_sigevent.sigev_signo = SIGIO;
m_aiocb.aio_sigevent.sigev_value.sival_ptr=&my_aiocb;
ret = aio_read( &my_aiocb );
```

AIO - Examples

- How to send signal to synchronize with aio (from IBM website, cont.) :

```
void aio_completion_handler( int signo, siginfo_t
  *info, void *context )
{
  struct aiocb *req;
  if (info->si_signo == SIGIO) {
    req = (struct aiocb *)info->si_value.sival_ptr;
    if (aio_error( req ) == 0) {
      ret = aio_return( req );
    }
  }
  return;
}
```

Synchronization of AIO

- **`int aio_cancel(int fildes, struct aiocb *aiocbp);`**
  - ask for operation cancellation
  - can be ignored by implementation, just a hint
  - if operation cannot be cancelled it must not be interrupted or altered in any way
  - cancelled operation aio_error status id (`ECANCELED`)
  - can cancel single operation or all operations on given descriptor

Synchronization of AIO

- **`aio_cancel`** function (cont.):
  - function returns:
    - AIO_CANCELED if all requests were successfully cancelled
    - AIO_NOTCANCELED if at least one operation was not cancelled (use aio_error to check which operations were cancelled)
    - AIO_ALLDONE if all operations were finished by the time of this call