Advanced I/O

- Overview
- Non-Blocking I/O
- Non-Blocking I/O on a FIFO
- Uses of Non-Blocking I/O on a FIFO
- Non-Blocking I/O Example
- Non-Blocking I/O Example Continued
- Non-Blocking I/O Example Continued
- Non-Blocking I/O Example Execution Log
- Synchronized I/O
- Integrity Measures
- Buffer Flushing System Calls
- `open()` Synchronization Flags
- Checking for I/O Ready Using `select()`
- `select()` Time Specifications
- Operating on File Descriptor Sets
- Select Example
- Select Example Continued
- Select Example Continued
- Controlling an Open File
- File Locking Using `fcntl()`
- Lock Inheritance
- Advisory versus Mandatory Locking

- Excluding Multiple Processes
- Excluding Multiple Processes Continued
- Excluding Multiple Processes Continued
- Excluding Multiple Processes Continued
- Excluding Multiple Processes Log
- Memory Mapped I/O
- Mapping Types
- Memory Mapped Files
- Memory Mapped I/O Continued
- Unmapping Memory
- Memory Mapped File Copy
- Memory Mapped File Copy Continued
- Shared Memory IPC
- `mprotect()` Call
- Auto Stack Growing
- Execution Log
- `main()` Program
- `main()` Program Continued
- Stack Initialization
- Stack Initialization Continued
- Stack Deallocation
- Stack Overflow Handler
- Postfix Interpreter
- Postfix Interpreter Continued

- References

Overview

- Non-blocking I/O.
- Synchronous I/O.
- `select()` call.
- File locking.
- Memory-mapped I/O.

Non-Blocking I/O

- Call to `open()` can specify `O_NONBLOCK` flag.
- Use `fcntl()` on a already open descriptor with `O_NONBLOCK` to make I/O non-blocking (must be used for pipes and sockets whose descriptors are not obtained using `open()`).
- If a call would have blocked, then it returns with an error with `errno` set to `EAGAIN`.
- Affects I/O to terminals, pseudo-terminals, pipes, FIFOs, sockets. Can also affect I/O for regular files when mandatory locking enabled.

Non-Blocking I/O on a FIFO

- `open()` of FIFO with `O_NONBLOCK` for reading succeeds immediately irrespective of whether or not write end has been opened (it makes sense to read no data from a FIFO which is not open for writing).

- `open()` of FIFO with `O_NONBLOCK` for writing succeeds immediately if read end has been opened; fails with error `ENXIO` if read end is not open (since writing to it would generate a `SIGPIPE` signal).

- Read on a non-blocking FIFO with no data fails with error `EAGAIN`.

- Write on a non-blocking FIFO which is full (cannot accomodate data without violating `PIPE_BUF` constraint) fails with error `EAGAIN`.

Uses of Non-Blocking I/O on a FIFO

- Allows single process to open both ends of a FIFO by reading using `O_NONBLOCK`.

- Avoids deadlocks between processes opening two FIFOs: P1 opens FIFO A for reading and then FIFO B for writing; P2 opens FIFO B for reading and FIFO A for writing; deadlock without `O_NONBLOCK`.

Non-Blocking I/O Example

```
int
main(int argc, char *argv[])
{
  int fifoFD;
  int fd[2];
  if (argc != 2) {
    fprintf(stderr, "usage: nonblock FIFO\n");
    exit(1);
  }
  if (fcntl(STDIN_FILENO, F_SETFL,
            O_NONBLOCK) < 0) {
    fprintf(stderr,
            "could not set STDIN non-block: %s\n",
            strerror(errno));
    exit(1);
  }
  if ((fifoFD = open(argv[1],
                    O_RDONLY|O_NONBLOCK)) < 0) {
    fprintf(stderr, "could not open %s: %s\n",
            argv[1], strerror(errno));
    exit(1);
  }
```

Non-Blocking I/O Example Continued

```
fd[0] = STDIN_FILENO; fd[1] = fifoFD;
while (1) {
  int i;
  for (i = 0; i < 2; i++) {
    char *src =
      (i == 0) ? "STDIN" : "FIFO";
    enum { MAX_BUF = 128 };
    char buf[MAX_BUF];
    int n;
    if ((n = read(fd[i], buf, MAX_BUF))
        < 0) {
      if (errno != EAGAIN) {
        fprintf(stderr, "read error on "
                "%s: %s\n",
                src, strerror(errno));
        exit(1);
      }
    }
```

Non-Blocking I/O Example Continued

```
      else if (n == 0) {
        return 0;
      }
      else {
        printf("%s: %.*s", src, n, buf);
      }
      sleep(10);
    } /* for (i = 0; i < 2; i++) */
  } /* while (1) */
}
```

Non-Blocking I/O Example Execution Log

```
$ mkfifo fifo
$ ls -l fifo
prw-rw-r--   1 umrigar  umrigar         0 Mar 13 13:03 fifo
$ ./nonblock fifo
Hello from stdin
STDIN: Hello from stdin
FIFO: Hello from fifo
FIFO: Hello again from fifo
Hello again from stdin
STDIN: Hello again from stdin
$
```

Concurrently, in another shell:

```
$ echo "Hello from fifo" >fifo
$ echo "Hello again from fifo" >fifo
$
```

Synchronized I/O

Synchronized

Data is actually written to I/O device.

Synchronous

Call returns only after data is read/written (usually, to buffer cache).

Normal Unix I/O is *synchronous* but *unsynchronized*. Asynchronous I/O uses a different group of system calls (`aio_write()`, etc).

Integrity Measures

Need to distinguish between *data* (file contents) and *metadata* (information about the file like size and timestamps).

Data Integrity

Data has been written out so that it can be retrieved. May require file size update but not necessarily file timestamps.

File Integrity

Superset of data integrity: all data and meta-data has been written out.

Buffer Flushing System Calls

```
void sync(void);
int fsync(int fd);
int fdatasync(int fd);
```

- `sync()` will flush the buffer cache at some future time (returns immediately). Used for shut down or unmounting removable device.

- `fsync()` does not return until **all buffers** associated with `fd` have been flushed to the I/O device. Ensures file integrity.

- `fdatasync()` does not return until **all data buffers** associated with `fd` have been flushed to the I/O device. Ensures data integrity.

- Since these are options, we need to check. If not supported, then `sync()` and `fsync()` may be NOP’s!!

`open()` Synchronization Flags

`O_SYNC`

Implicit `fsync()` after every write.

`O_DSYNC`

Implicit `fdatasync()` after every write.

`O_RSYNC`

Sync’s reads. Must be used with `O_SYNCH` or `O_DSYNC`. Causes inode access time to be flushed to disk. May disable read-ahead.

Using these flags can affect performance drastically.

Checking for I/O Ready Using `select()`

```
int select(int maxfd1, fd_set *readfds,
           fd_set *writefds,
           fd_set *exceptfds,
           struct timeval *tptr);
```

- `readfds` specifies a set of file descriptors on which to wait, until one or more is ready for reading.

- `writefds` specifies a set of file descriptors on which to wait, until one or more is ready for writing.

- `exceptfds` specifies a set of file descriptors on which to wait, until one or more gets an exceptional condition (arrival of out-of-band data on a network condition or certain conditions on a terminal in *packet mode*).

`select()` Time Specifications

- If `tptr` is NULL, then block forever, until some descriptor becomes ready.
- If elements of `*tptr` are 0, then return immediately after testing descriptors.
- If `tptr` is non-NULL and elements are non-zero, then return after specified time. If `readfds`, `writefds` and `exceptfds`, this gives a sleep with finer granularity than `sleep()`.

Operating on File Descriptor Sets

A `fd_set` is a set of file descriptors. Can only be operated on using the following macros:

`FD_ZERO(fd_set *fdset)`
    Clear entire `fdset`.

`FD_SET(int fd, fd_set *fdset)`
    Set `fd` in set `fdset`.

`FD_CLR(int fd, fd_set *fdset)`
    Clear `fd` in set `fdset`.

`FD_ISSET(int fd, fd_set *fdset)`
    Return non-zero if descriptor `fd` is set in `fdset`.

Select Example

```
int
main(int argc, char *argv[])
{
  int fifoFD;
  if (argc != 2) {
    fprintf(stderr, "usage: nonblock FIFO\n");
    exit(1);
  }
  if ((fifoFD = open(argv[1], O_RDONLY)) < 0) {
    fprintf(stderr, "could not open %s: %s\n",
            argv[1], strerror(errno));
    exit(1);
  }
```

Select Example Continued

```
while (1) {
  fd_set readfds;
  FD_ZERO(&readfds);
  FD_SET(STDIN_FILENO, &readfds);
  FD_SET(fifoFD, &readfds);
  if (select(fifoFD + 1, &readfds,
      NULL, NULL, NULL) < 0) {
    fprintf(stderr, "select error: %s\n",
            strerror(errno));
    exit(1);
  }
  else {
    int fd = (FD_ISSET(fifoFD, &readfds))
            ? fifoFD
            : STDIN_FILENO;
    char *src = (fd == fifoFD)
               ? "FIFO"
               : "STDIN";
```

Select Example Continued

```
      enum { MAX_BUF = 128 };
      char buf[MAX_BUF];
      int n;
      if ((n = read(fd, buf, MAX_BUF))
          < 0) {
        fprintf(stderr,
                "read error on %s: %s\n",
                src, strerror(errno));
        exit(1);
      }
      else if (n == 0) {
        return 0;
      }
      else {
        printf("%s: %.*s", src, n, buf);
      }
    }
  }
}
```

Controlling an Open File

File locking can be done using `fcntl()` or `flock()` (not always available).

Recall:

```
int fcntl(int fd, int op, ...);
```

- `fd` is descriptor of already opened file.
- `op` describes operation which may use the optional 3rd argument: Covered `F_DUPFD`, `F_GETFD`, etc. Will look at `F_GETLK`, `F_SETLK`, `F_SETLKW`.

File Locking Using `fcntl()`

When using `F_GETLK`, `F_SETLK`, `F_SETLKW`, 3rd argument specified as:

```
struct flock {
  short l_type;  /* lock type: one of F_RDLCK, F_WRLCK, F_UNLCK. */
  short l_whence;/* interpretation for l_start: as for lseek() */
  off_t l_start; /* start of locked section */
  off_t l_len;   /* length of locked section: 0 means to EOF from l_start */
  pid_t l_pid;   /* F_GETLK returns PID of process holding lock. */
};
```

Typically many processes can concurrently *read* a resource, while only a single process can *write* the resource. Hence *read-locks* are also referred to as *shared locks* while *write-locks* are referred to as *exclusive locks*.

`F_SETLK` returns with `errno` set to EAGAIN or EACCES if lock cannot be obtained, wherea `F_SETLKW` will block.

Lock Inheritance

Locks are associated with a process and a file.

- When process terminates all its locks are released.
- When a descriptor is closed, all locks on file associated with descriptor are released. This is true even if the file is open under another descriptor in the same process.
- Locks are not inherited by a child across a `fork()`.
- Locks are inherited across an `exec()`. However, if the close-on-exec flag is set for a file descriptor, then all locks on the underlying file are released when the descriptor is closed as part of the `exec()`.

Advisory versus Mandatory Locking

- With *advisory locking* locking will work only if processes cooperate.
- On some systems, *mandatory locking* can be obtained if the `setgid` bit is set and group-execute bit is off.
- On Linux, filesystem has to be mounted with `-o mand` option.
- If a mandatory exclusive lock is hit, blocking read/write will block, non-blocking read/write get an error (`EAGAIN`).
- If a `open()` tries to truncate (`O_TRUNC`) an existing file with outstanding mandatory locks, then error `EAGAIN` (solaris gives error even for `O_CREAT`).
- Mandatory locking works at system call level. Can have surprising effects at application level in that it is possible to *edit* a file with pending

mandatory locks!!

Excluding Multiple Processes

Following program:

```
static int
lockPidFile()
{
  int fd = open(FILE_NAME, O_WRONLY|O_CREAT, 0644);
  if (fd < 0) {
    fprintf(stderr, "cannot open %s: %s\n",
            FILE_NAME, strerror(errno));
    exit(1);
  }
  else {
    struct flock flock;
    memset(&flock, 0, sizeof(flock));
    flock.l_type = F_WRLCK;
    flock.l_whence = SEEK_SET;
    flock.l_start = 0;
    flock.l_len = 0;
```

Excluding Multiple Processes Continued

```
if (fcntl(fd, F_SETLK, &flock) < 0) {
  if (errno == EAGAIN &&
      fcntl(fd, F_GETLK, &flock) >= 0) {
    fprintf(stderr, "process %ld has lock\n",
            (long)flock.l_pid);
  }
  else {
    fprintf(stderr, "lock error: %s\n",
            strerror(errno));
  }
  exit(1);
}
if (ftruncate(fd, 0) != 0) {
  fprintf(stderr, "cannot truncate %s: %s\n",
          FILE_NAME, strerror(errno));
  exit(1);
}
```

Excluding Multiple Processes Continued

```
    {
      FILE *f = fdopen(fd, "w");
      if (!f) {
        fprintf(stderr, "cannot fdopen %s(%d): %s\n",
                FILE_NAME, fd, strerror(errno));
        exit(1);
      }
      fprintf(f, "%ld\n", (long)getpid());
      fflush(f);
    }
  }
  return 0;
}
```

Excluding Multiple Processes Continued

```
int main() {
  printf("%ld trying lock\n", (long)getpid());
  lockPidFile();
  printf("%ld got lock\n", (long)getpid());
  sleep(60);
  return 0;
}
```

Excluding Multiple Processes Log

```
$ ./fcntl_lock &
[1] 30598
$ 30598 trying lock
30598 got lock

$ ./fcntl_lock &
30599 trying lock
[2] 30599
$ process 30598 has lock

[2]+  Exit 1                  ./fcntl_lock
$ ps
  PID TTY          TIME CMD
30473 pts/15   00:00:00 bash
30598 pts/15   00:00:00 fcntl_lock
30603 pts/15   00:00:00 ps
$
[1]+  Done                    ./fcntl_lock
$ ./fcntl_lock &
[1] 30626
30626 trying lock
30626 got lock
$
```

Memory Mapped I/O

- Allow accessing files using normal memory access operations instead of `read()`, `write()`, `seek()`, etc.
- Can also be used for IPC between arbitrary processes.
- Often used by many `malloc` implementations for allocating requests for large memory blocks. Also used by GNU’s `mmalloc` library.

Mapping Types

Private file mapping

Maps a region of a file into memory, but writes to the memory by one process are not visible to other processes (uses copy-on-write).

Shared file mapping

Maps a region of a file into memory and modifications to the memory are reflected in the file and visible to other processes (does not use copy-on-write).

Private anonymous mapping

No corresponding file. Uses copy-on-write so that other processes do not see changes. Used for allocating large blocks of zero-filled memory.

Shared anonymous mapping

No corresponding file. No copy-on-write. Can be used for sharing memory between multiple related processes.

Memory Mapped Files

```
void *mmap(void *addr, size_t len, int prot,
           int flags, int filedes, off_t off);
```

addr

Specifies address at which filedes should be mapped. Normally specified as 0, to allow system to choose address. If non-zero, should be a multiple of the system page size, especially if MAP_FIXED flag is specified.

len

Number of bytes to be mapped.

prot

Must be specified as consistent combinations (or) of PROT_READ, PROT_WRITE, PROT_EXEC or PROT_NONE. Specified value must be consistent with mode on filedes. Because of hardware limitations, actual permissions may be less restrictive than those requested.o

Memory Mapped I/O Continued

`flags`

Consistent combinations of

`MAP_FIXED`

Map at specified address `addr` only.

`MAP_ANON`

Create an anonymous mapping not connected to a file. `fd` and `off` are ignored. Mapped memory is initialized with zeros. On some systems, where `MAP_ANON` is not supported, the same result may be obtained by mapping `/dev/zero` (further details below).

`MAP_SHARED`

Any writes to the memory update the file.

`MAP_PRIVATE`

Any writes to the memory update a copy of the file.

One of `MAP_SHARED` or `MAP_PRIVATE` must be specified.

`filedes`

File descriptor being mapped.

`off`

Starting offset of region in `filedes` being mapped.

Unmapping Memory

```
int munmap(void *addr, size_t len);
```

- Closing `filedes` does not unmap.
- Unmapped when process terminates, or by calling `unmap()`.

Memory Mapped File Copy

Timing results inconclusive.

```
int
main(int argc, char *argv[])
{
  int fdin, fdout;
  char *src, *dst;
  struct stat statbuf;

  if (argc != 3) {
    fprintf(stderr,
            "usage: a.out <fromfile> <tofile>\n");
    exit(1);
  }
  if ( (fdin = open(argv[1], O_RDONLY)) < 0) {
    fprintf(stderr,
            "can’t open %s for reading: %s\n",
            argv[1], strerror(errno));
    exit(1);
  }
  if ( (fdout =
          open(argv[2], O_RDWR|O_CREAT|O_TRUNC,
               0644)) < 0) {
    fprintf(stderr,
            "can’t creat %s for writing: %s\n",
            argv[1], strerror(errno));
    exit(1);
  }
```

Memory Mapped File Copy Continued

```
  if (fstat(fdin, &statbuf) < 0) {
    /* need size of input file */
    perror("fstat error"); exit(1);
  }
  /* set size of output file */
  if (lseek(fdout, statbuf.st_size - 1, SEEK_SET)
      == -1) {
    perror("lseek error"); exit(1);
  }
  if (write(fdout, "", 1) != 1) {
    perror("write error"); exit(1);
  }

  if ( (src = mmap(0, statbuf.st_size, PROT_READ,
                   MAP_FILE | MAP_SHARED, fdin, 0))
        == (caddr_t) -1) {
    perror("mmap error for input"); exit(1);
  }
  if ((dst=
         mmap(0, statbuf.st_size, PROT_READ|PROT_WRITE,
              MAP_FILE | MAP_SHARED, fdout, 0))
      == (caddr_t) -1) {
    perror("mmap error for output"); exit(1);
  }

  /* do the file copy */
  memcpy(dst, src, statbuf.st_size);

  exit(0);
}
```

Shared Memory IPC

- The special device file ‘/dev/zero’ is an infinite source of 0 bytes when read.
- Ignores any writes to it.
- If `/dev/zero` is mapped in a parent, using `MAP_SHARED`, then writes to the corresponding mapped memory are seen by all descendent processes.
- Allows IPC between related processes.
- Still need some form of synchronization.

`mprotect()` Call

```
int mprotect(const void *addr,
             size_t len, int prot);
```

- Usually `addr` must be page-aligned `len` should be a multiple of the page-size (use `getpagesize()`) (otherwise it is rounded up to a page size boundary).

- `prot` is a bitwise-or of `PROT_NONE`, `PROT_READ`, `PROT_WRITE`, `PROT_EXEC`.

Auto Stack Growing

- Translate infix expressions to postfix using a recursive descent parser (details unimportant; covered previously).
- Evaluate postfix expressions using a stack with size doubled on overflow.
- Use `mprotect()` to turn off access above stack. Catch `SIGSEGV` signal and double stack size. Note that catching this signal and continuing may not work on all systems.
- Need to restart instruction which blew stack: use `sigsetjmp()`, `siglongjmp()`.

Execution Log

```
$ ./mmapplay
>> 1+2
postfix:  1 2+
interpreted result (no stk check): 3
interpreted result (with stk check): 3
>> 1+2*3
postfix:  1 2 3*+
interpreted result (no stk check): 7
stack grown to 4 ints
interpreted result (with stk check): 7
>> 1 + (2 + (3 + (4 + (5 + (6 + (7 + (8 + 9)))))))
postfix:  1 2 3 4 5 6 7 8 9++++++++
interpreted result (no stk check): 45
stack grown to 4 ints
stack grown to 8 ints
stack grown to 16 ints
interpreted result (with stk check): 45
>> ^D$
```

main() Program

```
int
main()
{
  enum {
    IN_BUF_LEN = 80,
    OUT_BUF_LEN = 80
  };
  char inBuf[IN_BUF_LEN];
  char outBuf[OUT_BUF_LEN];
  while (printf(">> ") && fflush(stdout) == 0 &&
         fgets(inBuf, IN_BUF_LEN, stdin)) {
    int parseStatus;
    if (inBuf[strlen(inBuf) - 1] != ’\n’) {
      fprintf(stderr, "input too long\n"); continue;
    }
    parseStatus = parse(inBuf, outBuf, OUT_BUF_LEN);
```

`main()` Program Continued

```
    if (parseStatus == -1) {
      fprintf(stderr, "parse error\n");
    }
    else if (parseStatus == -2) {
      fprintf(stderr, "output buffer overflow\n");
    }
    else {
      fprintf(stdout, "postfix: %s\n", outBuf);
      { int sp[16];
        int result = interp(outBuf, sp);
        fprintf(stdout,
               "interpreted result (no stk check): %d\n",
               result);
      }
      {
        int result;
        initStk(2);
        result = interp(outBuf, NULL);
        freeStk();
        fprintf(stdout,
                "interpreted result (with stk check): %d\n",
                result);
      }
    }

  }
  return 0;
}
```

Stack Initialization

```
static void *mapBase;
int mapLen;

int *sp;
static int *stkBase;

void
initStk(int size)
{
  size_t ps = getpagesize();
  int zero = open("/dev/zero", O_RDWR);
  mapLen = 2*ps;
  if (zero < 0) {
    perror("/dev/zero open"); exit(1);
  }
  mapBase = mmap(NULL, mapLen,
                 PROT_READ|PROT_WRITE,
                 MAP_PRIVATE, zero, 0);
  if (mapBase == (void *)-1) {
    perror("mmap"); exit(1);
  }
  assert(((long)mapBase) % ps == 0);
```

Stack Initialization Continued

```
  { char *protPage = (char *)mapBase + ps;
    if (mprotect(protPage, ps, PROT_NONE) < 0) {
      perror("mprotect"); exit(1);
    }
  }
  { int *protPage = (int *)((char *)mapBase + ps);
    sp = stkBase = protPage - size;
  }
  { struct sigaction sigact;
    sigset_t sigmask;
    sigemptyset(&sigmask);
    sigact.sa_handler = growStk;
    sigact.sa_mask = sigmask;
    sigact.sa_flags = 0;
    if (sigaction(SIGSEGV, &sigact, NULL) < 0) {
      perror("signal"); exit(1);
    }
  }
}
```

Stack Deallocation

```
void
freeStk(void)
{
  if (munmap(mapBase, mapLen) < 0) {
    perror("munmap"); exit(1);
  }
}
```

Stack Overflow Handler

```
sigjmp_buf stkJmp;

/* double size of stack */
void
growStk(int signo)
{
  int initStkSize = sp - stkBase;
  int *newBase = stkBase - initStkSize;
  int i;
  for (i = 0; i < initStkSize; i++) {
    newBase[i] = stkBase[i];
  }
  sp = stkBase; stkBase = newBase;
  fprintf(stderr, "stack grown to %d ints\n",
          2*initStkSize);
  siglongjmp(stkJmp, 1);
}
```

Postfix Interpreter

```
int
interp(const char *postfix, int *sp1)
{
  const char *p = postfix;
  if (sp1 != NULL) sp= sp1;
  while (*p != ’\0’) {
    switch (*p++) {
      case ’ ’: {
        char *pp;
        volatile int v = strtol(p, &pp, 10);
        p = pp;
        sigsetjmp(stkJmp, 1);
        *sp = v;
        sp++;
      }
      break;
      case ’+’: {
        int right = *--sp;
        int left = *--sp;
        *sp++ = left + right;
      }
      break;
```

Postfix Interpreter Continued

```
      case ’-’: {
        int right = *--sp;
        int left = *--sp;
        *sp++ = left - right;
      }
      break;
      /* Code for * / similar */
      case ’u’: {
        int rand = *--sp;
        *sp++ = -rand;
      }
      break;
    }
  }
  return *--sp;
}
```

References

Text, 5.9, 13, 44.9, 49, 55, 63

APUE: Ch. 14.