Domain sockets

Summary of UNIX IPC so far

- Shared memory
    - Anonymous mmap between related process (i.e., parent and child)
    - File-backed mmap or XSI shared memory between unrelated processes
- Synchronization
    1. Multiple threads in a single process
        - pthread mutex, rw lock, condition variable
    2. Multiple threads & processes sharing a memory region
        - Unnamed POSIX semaphore
        - pthread mutex, rw lock, condition variable with PTHREAD_PROCESS_SHARED attribute
    3. Multiple processes with no shared memory
        - Named POSIX semaphore
- Pipe
    - only between related processes
    - half duplex (i.e., one way communication)
- FIFO (aka named pipe)
    - represented as a file, thus can be used between unrelated processes
    - still half duplex
- TCP socket
    - connects any two processes including remote processes
    - full duplex
    - high protocol overhead
    - reliable stream socket – reliable byte stream, but message boundaries are not preserved
- UDP sockets
    - lower protocol overhead than TCP
    - unreliable datagram socket – message boundaries are preserved, but deliveries are unreliable

UNIX domain sockets

- A cross between pipes and sockets

  1. can be used as a full duplex pipe using `socketpair()`
  2. can be used as a local-only socket using sockets API
  3. **reliable** when used in datagram mode
  4. can transport special things like open file descriptor

- Full duplex pipe using `socketpair()`

  ```
  int socketpair(int domain, int type, int protocol, int sv[2]);

          Returns 0 if OK, -1 on error
  ```

  - same picture as the one for `pipe()` but arrows going both ways

  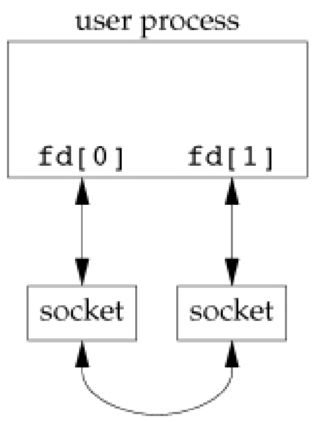

  Figure 17.1, APUE

- Example of exchanging datagram using domain socket

  recv-unix.c:

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

static void die(const char *m) { perror(m); exit(1); }

int main(int argc, char **argv)
{
    if (argc != 3) {
        fprintf(stderr, "usage: %s <domain-sock-name> <num-chars>\n", argv[0]);
        exit(1);
    }

    const char *name = argv[1];
    int num_to_recv = atoi(argv[2]);

    struct sockaddr_un  un;

    if (strlen(name) >= sizeof(un.sun_path)) {
        errno = ENAMETOOLONG;
        die("name too long");
    }

    int fd, len;

    // create a UNIX domain datagram socket
    if ((fd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
        die("socket failed");

    // remove the socket file if exists already
    unlink(name);

    // fill in the socket address structure
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);

    // bind the name to the descriptor
    if (bind(fd, (struct sockaddr *)&un, len) < 0)
        die("bind failed");

    char buf[num_to_recv + 1];

    for (;;) {
        memset(buf, 0, sizeof(buf));
        int n = recv(fd, buf, num_to_recv, 0);
        if (n < 0)
            die("recv failed");
        else
```

```
            printf("%d bytes received: \"%s\"\n", n, buf);
    }

    close(fd);
    unlink(name);
    return 0;
}
```

send-unix.c:

```
#include "apue.h"
#include <sys/socket.h>
#include <sys/un.h>
#include <errno.h>

static void die(const char *m) { perror(m); exit(1); }

int main(int argc, char **argv)
{
    if (argc != 4) {
        fprintf(stderr, "usage: %s <domain-sock-name> <msg> <num-repeat>\n", arg
        exit(1);
    }

    const char *name = argv[1];
    const char *msg = argv[2];
    int num_repeat = atoi(argv[3]);

    struct sockaddr_un  un;

    if (strlen(name) >= sizeof(un.sun_path)) {
        errno = ENAMETOOLONG;
        die("name too long");
    }

    int fd, len, i;

    // create a UNIX domain datagram socket
    if ((fd = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0)
        die("socket failed");

    // fill in the socket address structure with server's address
    memset(&un, 0, sizeof(un));
    un.sun_family = AF_UNIX;
    strcpy(un.sun_path, name);
    len = offsetof(struct sockaddr_un, sun_path) + strlen(name);

    for (i = 0; i < num_repeat; i++) {
        int n = sendto(fd, msg, strlen(msg), 0, (struct sockaddr *)&un, len);
        if (n < 0)
            die("send failed");
        else
            printf("sent %d bytes\n", n);
    }

    close(fd);
    return 0;
}
```

- Passing file descriptors

- duplicate a file descriptor across running processes

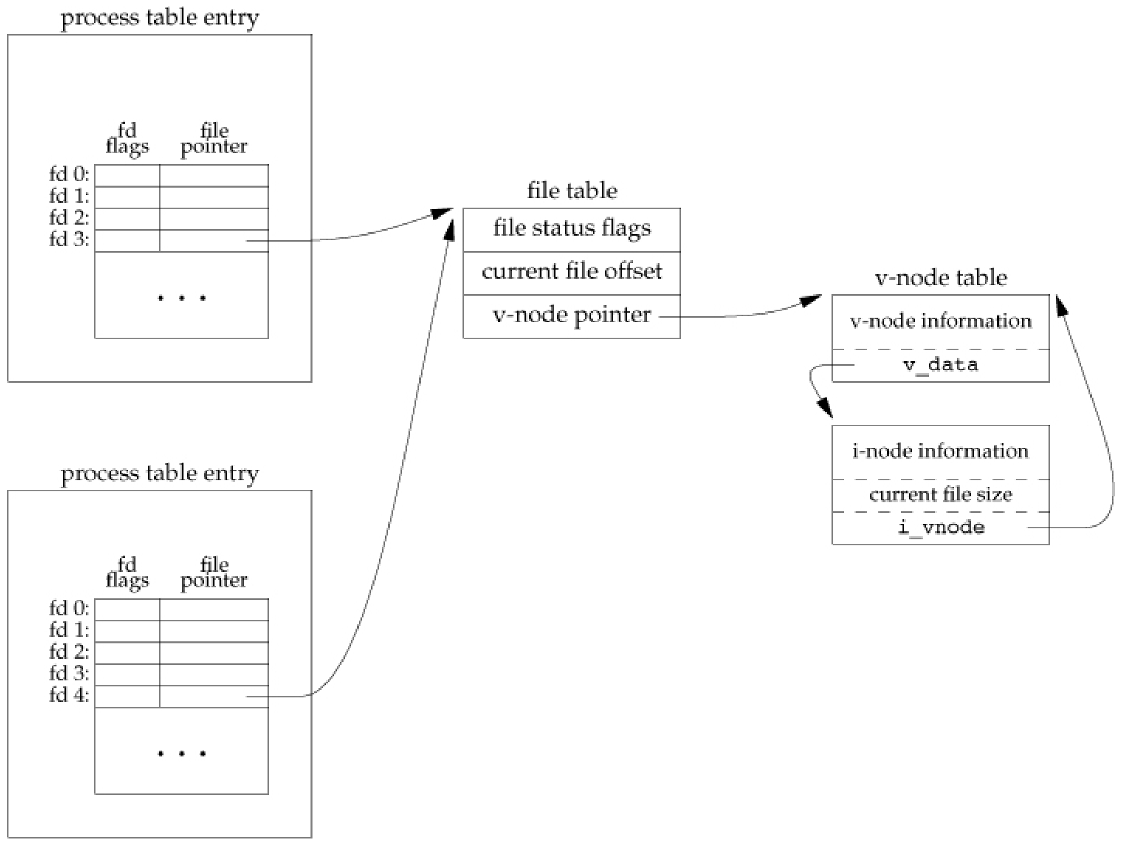

Figure 17.11, APUE

Last updated: 2015–02–26