IPC

Zilogic Systems

1. Pipes

- A pipe is a in-memory kernel buffer (FIFO), that can be written to and read from using two file descriptors.
- Pipes are uni-directional. Data can be transferred through the pipe only in one direction.
- When a process writes to the pipe, the data is stored in the FIFO.
- When a process reads from the pipe, the data is removed from the FIFO.
- When a process writes to the pipe, and no space is available in the FIFO, the process is blocked till space becomes available. That is, the other process reads from the FIFO.
- When a process reads from the pipe, and no data is available in the FIFO, the process is blocked till data becomes available. That is, the other process writes to the FIFO.
- A pipe can be created using `pipe()` system call.

```
ret = pipe(fd)

int fd[2];
int ret;
```

- The pipe is created and the read fd is available in `fd[0]`, the write fd is available in `fd[1]`.

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

#include <stdio.h>
#include <error.h>
#include <errno.h>

int main()
{
        int fd[2];
        pid_t pid;
        int ret;

        ret = pipe(fd);
        if (ret == -1)
                error(1, errno, "error creating pipe");

        pid = fork();
        if (pid == -1)
                error(1, errno, "error forking");

        if (pid == 0) {
                printf("In child\n");

                close(fd[1]);
                dup2(fd[0], STDIN_FILENO);
                execlp("sort", "sort", 0);
                error(1, errno, "error in execing");
        } else {
                printf("In parent\n");

//              close(fd[0]);
                dup2(fd[1], STDOUT_FILENO);

                printf("embedded\n");
                printf("training\n");
                printf("linux\n");
        }
}
```

2. Message Queue

- A message queue can be modeled as linked list of messages. Messages can be added to and removed from the queue.
- Message is a chunk of bytes. The maximum size of the message, and the no. of outstanding messages in a queue can be defined during the creation of the message queue.
- Message queues are uni-directional. Messages can be transferred through the message queue only in one direction.
- Messages can be added to the queue by opening the message queue for writing.
- Messages can be removed from the queue by opening the message queue for reading.
- Just as with pipes, processes are blocked, when the queue is full or empty.
- Messages queues also allows the user to specify priority to messages. A higher priority message is read before any of the lower priority message in the queue.
- A message queue is created using `mq_open()`.

```
fd = mq_open(name, flags);

mqd_t fd;
char * name;
int flags;
```

- Each message queue is uniquely identified by a name. The name should start with a `/`. The name of the message queue to be opened should be passed as argument to `mq_open()`.
- The `flags` is similar to that of the `open()` syscall. `O_RDONLY`, `O_WRONLY` can be used to indicate the direction of message transfer.
- `mq_open` returns the message queue descriptor on success and `-1` on failure. In Linux, the message queue descriptor is nothing but a file descriptor.
- Additional `O_CREAT` can be used to create the message queue, if it is not already present.
- When `O_CREAT` is used `mq_open()` accepts two other arguments - `mode` and `attr`.

```
fd = mq_open(name, flags | O_CREAT, mode, attr);

char * name;
int flags;
mqd_t fd;
mode_t mode;
```

```
struct mq_attr * attr;
```

- Just as with files, the `mode` specifies the read/write permission for user, group and others. Note that execute permission is not allowed on message queues.
- The `attr` specifies the message queue attributes, like the maximum size of a message, the maximum no. of messages in a queue, etc.

```
struct mq_attr {
       long mq_flags;       /* Flags: 0 or O_NONBLOCK */
       long mq_maxmsg;      /* Max. # of messages on queue */
       long mq_msgsize;     /* Max. message size (bytes) */
       long mq_curmsgs;     /* # of messages currently in queue */
};
```

- The user can explicitly specify during open whether the accesses to message queue shoud block using the `O_NONBLOCK` flag in the `flag` argument.
- The message queue can be made non-blocking by setting `O_NONBLOCK` is `mq_flags`. When such a message queue is openend, the message will be non-blocking irrespective of whether `O_NONBLOCK` is specified in the `open()` syscall.
- `mq_send()` and `mq_receive` is used to send data to and receive data from the message queue.

```
ret = mq_send(fd, msg, len, prio);
sret = mq_receive(fd, msg, len, &prio);

mqd_t fd;
char * msg;
size_t len;
unsinged prio;
int ret;
ssize_t sret;
```

- The message queue can be closed using `mq_close()` syscall. The message queue can be deleted using `mq_unlink()` syscall.
- Message queue wall, source code.

```
#include <poll.h>
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>
#include <stdio.h>
#include <errno.h>
#include <error.h>
#include <string.h>

#define ROWS 20
#define COLS 64

void wall_init(struct pollfd * pollfds, char buf[ROWS][COLS])
{
        int i;

        struct mq_attr attr = { 0, 10, COLS };
        for (i = 0; i < ROWS; i++) {
                char name[32];
                sprintf(name, "/user%u", i);

                pollfds[i].fd = mq_open(name, O_RDWR | O_CREAT, 0666, &attr);
                if (pollfds[i].fd == -1)
                        error(1, errno, "error opening message queue");

                pollfds[i].events = POLLIN;
        }

        system("clear");
}

void wall_cleanup(struct pollfd * pollfds)
{
        int i;

        for (i = 0; i < ROWS; i++)
                mq_close(pollfds[i].fd);
}

void trim(char * s)
{
        char * p = s;
        int l = strlen(p);

        while(isspace(p[l - 1])) p[--l] = 0;
        while(* p && isspace(* p)) ++p, --l;

        memmove(s, p, l + 1);
}

void wall_update(struct pollfd * pollfds, char buf[ROWS][COLS])
{
        int i;
        int ret;

        system("clear");

        for (i = 0; i < ROWS; i++) {
                if (pollfds[i].revents & POLLIN) {
                        ret = mq_receive(pollfds[i].fd, buf[i], sizeof(buf[i]), NULL);
                        if (ret == -1) {
                                if (errno == ETIMEDOUT)
                                        break;

                                error(1, errno, "error receving message");
                        }
                }

                trim(buf[i]);
                puts(buf[i]);
        }
}

void main(int argc, char * argv[])
{
        int ret;
        int i;
        char * retp;
        struct pollfd pollfds[ROWS];
        char buf[ROWS][COLS] = {};

        wall_init(pollfds, buf);

        while (1) {
                ret = poll(pollfds, ROWS, -1);
                if (ret == -1)
                        error(1, errno, "error polling fds");

                wall_update(pollfds, buf);
        }

        wall_cleanup(pollfds);
}
```

- Code to send messages to the wall.

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>
#include <stdio.h>
#include <errno.h>
#include <error.h>
#include <string.h>

int main(int argc, char *argv[])
{
        int fd;
        char buf[64];
        char * retp;
        int ret;

        if (argc != 2)
                error(1, 0, "incorrect no. of arguments");

        fd = mq_open(argv[1], O_WRONLY);
        if (fd == -1)
                error(1, errno, "error opening message queue");

        while (1) {
                printf("Enter message: ");
                retp = fgets(buf, sizeof(buf), stdin);
                if (retp == NULL)
```

```
                        break;

                ret = mq_send(fd, buf, strlen(buf) + 1, 0);
                if (ret == -1)
                        error(1, errno, "error sending message");
        }

        return 0;
}
```

3. Shared Memory

- When two processes perform memory mapped I/O on the same file (with `MAP_SHARED`), we have shared memory.
- A file can be memory mapped using `mmap()` syscall.

```
vaddr = mmap(addr, len, prot, flags, fd, offset);

void * vaddr;
void * addr;
size_t len;
int prot;
int flags;
int fd;
off_t offset;
```

- `fd` specifies the file to be mapped.
- `len` specifies the no. of bytes to mapped.
- `offset` specifies the starting position of the file to be mapped.
- `addr` specifies the virtual address to use for the mapping. When specified as `NULL` the kernel chooses an available virtual memory address for the mapping.
- The `prot` specifies, whether the mapping should be executable `PROT_EXEC`, readable `PROT_READ` and writable `PROT_WRITE`.
- The `flags` specifies, whether the mapping is shared `MAP_SHARED` or private `MAP_PRIVATE`. `MAP_SHARED` is used implement shared memory.
- `mmap()` returns the alloted virtual address on success and `MAP_FAILED` on failure.
- The mapping can be released using the `munmap()` syscall.

```
ret = munmap(addr, len)

void * addr;
int len;
int ret;
```

- Implementation of `cat` using memory mapping.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
```

```

#include <stdio.h>
#include <error.h>
#include <errno.h>
#include <string.h>

int main(int argc, char *argv[])
{
        int i;
        int ret;
        int sfd;
        char * smem;
        struct stat stat;
        off_t len;

        if (argc != 2)
                error(1, errno, "insufficient arguments");

        sfd = open(argv[1], O_RDONLY);
        if (sfd == -1)
                error(1, errno, "error opening src.");

        ret = fstat(sfd, &stat);
        if (ret == -1)
                error(1, errno, "error stating src.");

        len = stat.st_size;

        smem = mmap(NULL, len, PROT_READ, MAP_SHARED, sfd, 0);
        if (smem == MAP_FAILED)
                error(1, errno, "error mapping src.");

        for (i = 0; i < len; i++)
                putchar(smem[i]);

        ret = munmap(smem, len);
        if (ret == -1)
                error(1, errno, "error unmapping src.");

        close(sfd);
}
```

- Memory mapped wall.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

#include <stdio.h>
#include <error.h>
#include <errno.h>
```

```
#include <string.h>

enum {
        ROWS = 20,
        COLS = 20,
};

struct shared {
        char buf[ROWS][COLS];
};

int main(int argc, char *argv[])
{
        int i, j;
        int ret;
        int sfd;
        struct stat stat;
        off_t len;
        struct shared * shared;

        sfd = open("/tmp/gfx", O_RDWR | O_CREAT, 0666);
        if (sfd == -1)
                error(1, errno, "error opening src.");

        ret = ftruncate(sfd, sizeof(*shared));
        if (ret == -1)
                error(1, errno, "error truncating src.");

        shared = mmap(NULL, sizeof(*shared),
                      PROT_READ, MAP_SHARED, sfd, 0);

        if (shared == MAP_FAILED)
                error(1, errno, "error mapping src.");

        while (1) {
                for (i = 0; i < ROWS; i++) {
                        for (j = 0; j < COLS; j++) {
                                putchar(shared->buf[i][j]);
                        }
                        putchar('\n');
                }

                sleep(1);
                system("clear");
        }

        ret = munmap(shared, sizeof(*shared));
        if (ret == -1)
                error(1, errno, "error unmapping src.");

        close(sfd);
}
```

- Send messages to the wall.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>

#include <stdio.h>
#include <error.h>
#include <errno.h>
#include <string.h>

enum {
        ROWS = 20,
        COLS = 20,
};

struct shared {
        char buf[ROWS][COLS];
};

int main(int argc, char *argv[])
{
        int i, j;
        int ret;
        int sfd;
        struct shared * shared;
        struct stat stat;
        off_t len;

        sfd = open("/tmp/gfx", O_RDWR | O_CREAT, 0666);
        if (sfd == -1)
                error(1, errno, "error opening src.");

        shared = mmap(NULL, sizeof(*shared),
                      PROT_WRITE, MAP_SHARED, sfd, 0);

        if (shared == MAP_FAILED)
                error(1, errno, "error mapping src.");

        while (1) {
                memcpy(shared->buf[2], "hello", 5);
                sleep(1);
                memcpy(shared->buf[2], "     ", 5);
                sleep(1);
        }

        ret = munmap(shared, sizeof(*shared));
        if (ret == -1)
                error(1, errno, "error unmapping src.");

        close(sfd);
}
```

4. Semaphores

- Semaphores can be used for mutual exclusion or signalling.
- Semaphore is an integer variable, that is incremented or decremented using syscalls.
- Incrementing a semaphore is called a post operation, and decrementing a semaphore is called a wait operation.
- When the semaphore is 0, the wait operation will block, until someother process increments the semaphore using a post operation.

4.1. Mutex

- Code regions where a shared memory is accessed is called critical region.
- When two processes enter the critical region simulatenously we get a race condition.
- Race condition can be avoided by allowing sequential access to the shared memory, using a binary semaphore.
- Before accessing the shared memory the semaphore is decremented and after the process is done with the shared memory, the semaphore is incremented.

4.2. Signalling

- One other problem with memory mapped I/O is that the receiver, has no indication that data is available for reading.
- If such signalling is required, semaphores can be used.
- The semaphore is initialized to 0. The receiving process decrements the semaphore, and gets blocked.
- The sender, puts up the data in the shared memory, and signals the receiver by incrementing the semaphore.

4.3. API

- The semaphore is represented using the `sem_t` datatype.
- The semaphore should be accessible to multiple processes, and is put up in a shared memory. The shared memory can be initialized using `sem_init()`.

```
ret = sem_init(sem, shared, value)

sem_t * sem;
int shared;
unsigned value;
int ret;
```

- `sem` is the pointer to the semaphore to be initialized.
- `shared` specifies whether the semaphore is shared between threads of the same process (`0`), or between multiple processes (`1`).
- `value` is the initial value of the semaphore.
- `sem_init()` returns `0` on success and `-1` on failure.
- The semaphore can be incremented using `sem_post()` and can be decremented using `sem_wait()`.

```
ret = sem_post(sem);
ret = sem_wait(sem);

sem_t sem;
int ret;
```

- Memory mapped wall with semaphore signalling.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <semaphore.h>

#include <stdio.h>
#include <error.h>
#include <errno.h>
#include <string.h>

enum {
        ROWS = 20,
        COLS = 20,
};

struct shared {
        char buf[ROWS][COLS];
        sem_t sem;
};

int main(int argc, char *argv[])
{
        int i, j;
        int ret;
        int sfd;
        struct shared * shared;
        struct stat stat;
        off_t len;
        sem_t * sem;

        sfd = open("/tmp/gfx", O_RDWR | O_CREAT, 0666);
        if (sfd == -1)
                error(1, errno, "error opening src.");

        ret = ftruncate(sfd, sizeof(*shared));
        if (ret == -1)
                error(1, errno, "error truncating src.");

        shared = mmap(NULL, sizeof(*shared),
                      PROT_READ | PROT_WRITE, MAP_SHARED, sfd, 0);

        if (shared == MAP_FAILED)
                error(1, errno, "error mapping src.");
        ret = sem_init(&shared->sem, 1, 0);
        if (ret == -1)
                error(1, errno, "error init. semaphore");

        while (1) {
                system("clear");
                for (i = 0; i < ROWS; i++) {
                        for (j = 0; j < COLS; j++) {
                                putchar(shared->buf[i][j]);
                        }
                        putchar('\n');
                }

                sem_wait(&shared->sem);
        }

        sem_destroy(&shared->sem);

        ret = munmap(shared, len);
        if (ret == -1)
                error(1, errno, "error unmapping src.");

        close(sfd);
}
```

- Sending messages to the wall.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <semaphore.h>

#include <stdio.h>
#include <error.h>
#include <errno.h>
#include <string.h>

enum {
        ROWS = 20,
        COLS = 20,
};

struct shared {
        char buf[ROWS][COLS];
        sem_t sem;
};

int main(int argc, char *argv[])
{
        int ret;
        int sfd;
        struct shared * shared;
        struct stat stat;
        off_t len;

        sfd = open("/tmp/gfx", O_RDWR | O_CREAT, 0666);
        if (sfd == -1)
                error(1, errno, "error opening src.");

        shared = mmap(NULL, sizeof(*shared),
                      PROT_READ | PROT_WRITE, MAP_SHARED, sfd, 0);

        if (shared == MAP_FAILED)
                error(1, errno, "error mapping src.");

        while (1) {
                memcpy(shared->buf[2], "hello", 5);
                sem_post(&shared->sem);
                sleep(1);

                memcpy(shared->buf[2], "     ", 5);
                sem_post(&shared->sem);
                sleep(1);
        }

        ret = munmap(shared, sizeof(*shared));
        if (ret == -1)
                error(1, errno, "error unmapping src.");

        close(sfd);
}
```