SYS2(OS) Lab 6: Inter-process Communication

Dr Poonam Yadav and Sven Signer

Inter-process Communication

The previous lab session looked at implementing multiple threads inside a process using the POSIX pthreads library.

In real-world applications, processes often need to be able to communicate data between processes.

This lab session looks at two different methods for inter-process communication: shared memory and pipes.

Shared Memory

New processes created with fork() get a copy of the parent process's memory, but any later changes are not shared between processes.

Shared memory objects allow processes to designate a shared region of memory to communicate between processes.

Accessing shared memory is in practice typically a two stage process: creating/opening a shared memory object, then mapping it into to process's address space.

Beyond very simple examples, careful synchronisation of access to this memory may be required to prevent race conditions.

Shared Memory Objects

```
#include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t
mode);
```

- Opens /creates a shared memory object with the given name, access flags and permissions.
- Requires linker option `-lrt` when compiling.
- Returns a file descriptor to the shared memory block.
- Full details are available in the man pages:
  `man 3 shm_open`

Memory Mapping

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot,
int flags, int fd, off_t offset);
```

- Maps an object specified by the file descriptor fd into the process's address space.
- Returns a void pointer (i.e. a pointer to memory with no specific type) to the mapped area.
- Full details in the man pages:
  `man 2 mmap`

Shared Memory Cleanup

```
#include <sys/mman.h>
int munmap(void *addr, size_t length);
int shm_unlink(const char *name);
```

- When they are no longer needed, shared memory can be unmapped from memory and the associated objects deleted.
- Full details in the man pages:
  `man 2 munmap`
  `man 3 shm_unlink`

Shared Memory Example

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

int main() {
    int shm_fd = shm_open("Shared", O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, 4096);
    char *ptr = mmap(0, 4096, PROT_WRITE | PROT_READ, MAP_SHARED, shm_fd, 0);
    …
```

Three steps to create a shared memory region:

1) A shared memory object is opened/created with `shm_open`, returning a file descriptor.
2) The size of the shared memory object is set using `ftruncate`.
3) The shared memory object is mapped into the process's address space with `mmap`.

Pipes

Pipes allow for a simple way of sending data from one process to another.

Pipes are (typically) unidirectional data channels between a read-end and a write-end.

Using pipes avoids some of the synchronisation issues required when communicating over shared memory.

This lab looks at two basic types of pipes: "ordinary" pipes and "named" pipes. In POSIX terminology, these are respectively referred to as pipes and FIFOs.

An overview of the Linux implementation of pipes and FIFOs can be found in section 7 of the man pages: `man 7 pipe`

Pipes

Description

- Pipes are data channels used for interprocess communication.
- Each pipe has a read end and a write end. Data written to the write end is accessible from the read end on a first-in first-out basis.
- Pipes need to be setup by a parent process so that the child inherits the file descriptor. Therefore, pipes are typically used for communication between parent and child processes, or between child processes.

Call Signature

```
include <unistd.h>

int pipe(int pipefd[2]);
```

- Returns 0 on success.
- File descriptors for the read and write end of the pipe are stored into `pipefd[0]` and `pipefd[1]` respectively.
- See manual page for further details: `man 2 pipe`

Pipe Example

```
#include <unistd.h>

int main() {
    int fd[2];
    pipe(fd);

    pid_t cpid = fork();
    if (cpid == 0) {
        close(fd[0]);
        ...
    }
    else {
        close(fd[1]);
        ...
    }
```

The pipe is created, storing the read-end file descriptor in fd[0], and the write-end file descriptor into fd[1].

Fork a child process, which inherits a copy of the open file descriptors.

Parent and child process each close opposite unused ends of the pipe, such that one keeps the read-end open and the other keeps the write-end other. The process with the write end can then send message to other process.

FIFOs

Description

- Named pipes allow pipes to be setup between unrelated processes.
- The pipe is identified by a path on the filesystem (but the pipe contents is not stored on disk).
- Read and write end of the pipe have the same path, the flags given to the open command determine which end is accessed. This allows a process to swap between read and write ends by reopening the pipe.
- Other than the way they are created/opened, they are equivalent to ordinary pipes.

Call Signature

```
include <sys/stat.h>
int mkfifo(const char *pathname,
           mode_t mode);
```

- Creates a named pipe with the given pathname and access permissions.
- Returns 0 on success. A file descriptor can then be opened using the standard open system call.
- See manual pages for further details:
  `man 3 mkfifo`
  `man 2 open`

FIFO Example

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

int main() {
    mkfifo("/tmp/mypipe", 0666);
    pid_t cpid = fork();
    if (cpid == 0) {
        int fd = open("/tmp/mypipe", O_RDONLY);
        …
    }
    else {
        int fd = open("/tmp/mypipe", O_WRONLY);
        …
    }
```

Create a FIFO with the given pathname.

Fork a child process.

Two processes open different ends of the FIFO by specifying different access modes in the open flags.

Assignment

This lab session looks at different forms of inter-process communication.

Example programs for each of the covered communication primitives are provided on the VLE to get you started.