Programmation Systèmes
Cours 5 — Interprocess Communication and Pipes

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire PPS, Université Paris Diderot

2012–2013

Outline

Outline

Interprocess communication

In the UNIX world, the term InterProcess Communication (IPC) is used—in its broadest meaning—to refer to various forms of information exchange among UNIX processes.

UNIX has traditionally made easy for process to communicate, offering many ways to do so and making them cheap.

On the importance of making IPC easy

- the easier it is for processes to communicate
  → the more programmers will be willing to use IPC
- encouraging IPC → encouraging breaking down large applications into separate, well-defined programs
- one of the pillars of component reuse on UNIX

UNIX philosophy, abridged:

Write programs that do one thing and do it well.

IPC facilities

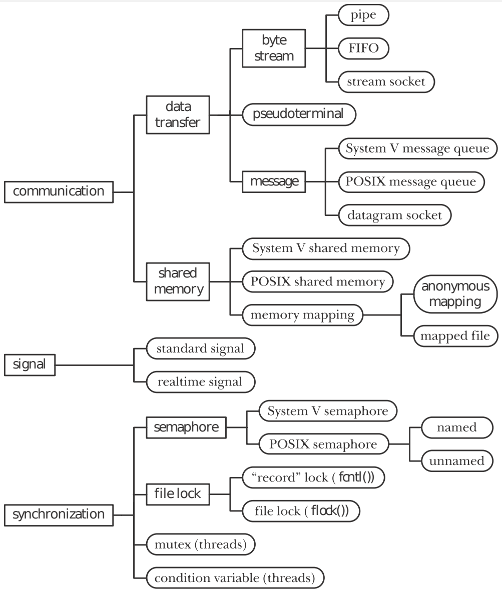
Many forms of IPC are available on UNIX systems.

All forms of IPC are either kernel-mediated (i.e. the kernel is involved in each usage of the facility) or require kernel intervention to be setup / torn-down, before / after use.

We can classify IPC facilities into the following categories:

communication	facilities concerned with exchanging data among processes
synchronization	facilities concerned with synchronizing actions among processes
signals	facilities concerned with notifying processes of events

A taxonomy of UNIX IPC facilities

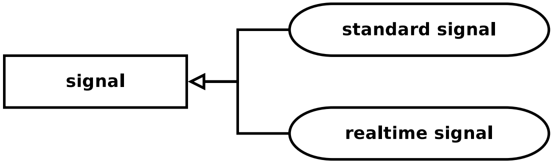


TLPI, Figure 43-1

Why so many IPC facilities?

- pedigree: different UNIX variants have grown different facilities, most of which ended up being merged throughout POSIX evolution
- new IPC facilities have been developed to overcome limitations of old IPC (e.g. POSIX IPC vs System V IPC)
- real differences in functionalities and/or communication paradigms

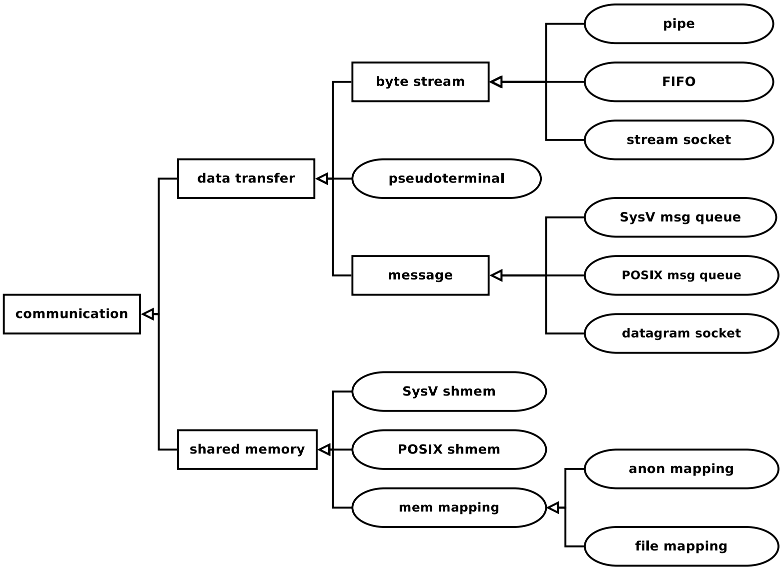
Signal facilities



We've already discussed UNIX signal handling at length.

Signals show that the categorization is indicative. While standard signals only permit event notification, real-time signals allow to exchange data via signal payloads.

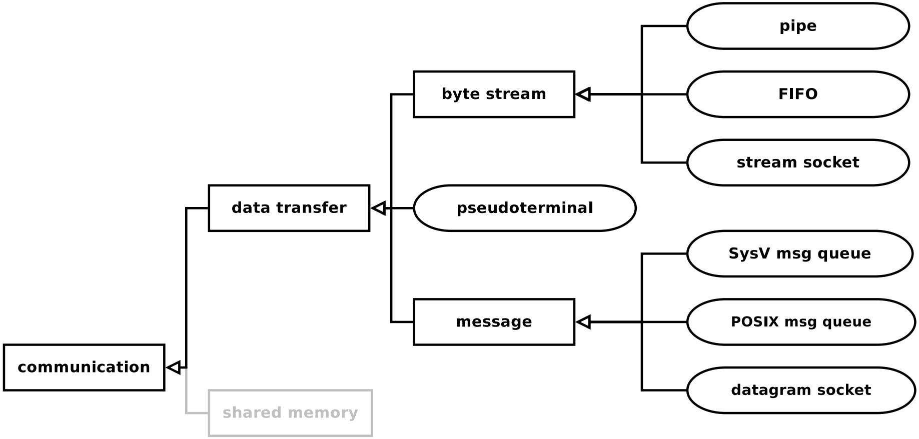
Communication facilities



Communication facilities — data transfer

Data transfer facilities allow communication between processes via explicit reads and writes on IPC objects

- communication is mediated by the kernel



Communication facilities — data transfer (cont.)

- byte stream facilities offer a file-like abstraction for IPC (i.e. undelimited streams of bytes)
- message facilities offer the abstraction of sending/receiving delimited messages
  - reads/writes happen at the message granularity
- pseudoterminal facilities permit to interact with processes that expect to be connected to a terminal, in the absence of it
  - e.g. remote logins

Communication facilities — shared memory

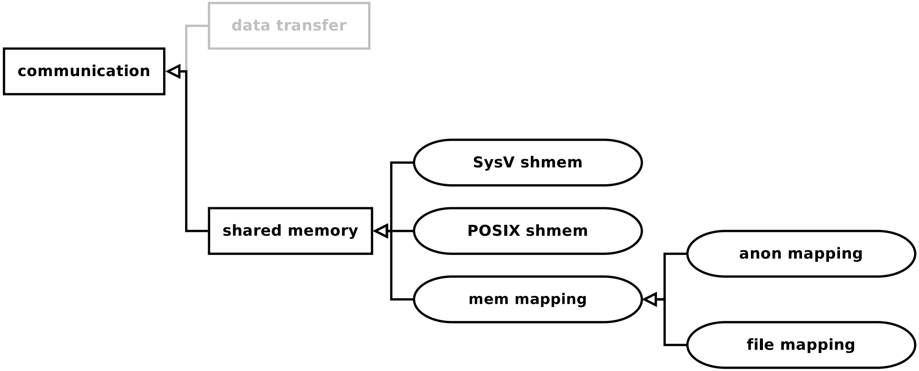
Shared memory IPC facilities allow different processes to map parts of their address spaces to the same memory frames.

After initial setup (by the kernel), communication is implicit. To "send" data to another process, we simply write data to shared memory (e.g. by assigning a value to a global variable located in shared memory); the other process will read from there.

Also: reading does not "consume" data, as it happens with data transfer.

- pro: no kernel mediation after initial setup → shared memory can be much faster than mediated IPC facilities
- cons: synchronization is needed to avoid memory corruption

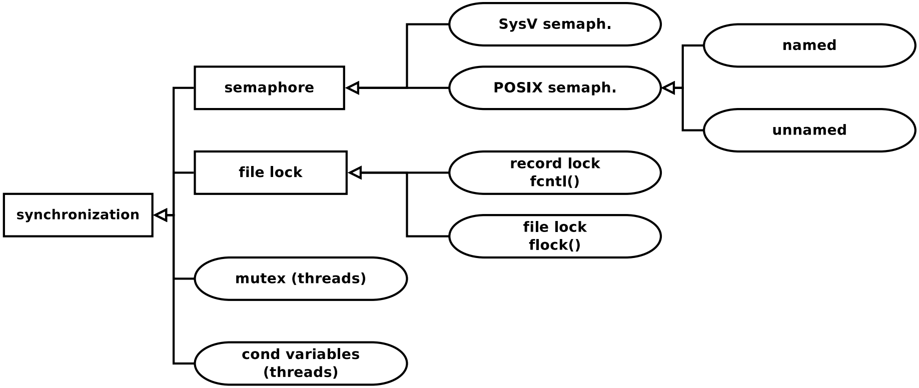
Communication facilities — shared memory (cont.)



Synchronization facilities

Synchronization is needed every time two (or more) processes want to coordinate their actions. Typical use cases come from race condition avoidance when dealing with shared resources

- such as, but not only as, shared memory...



Synchronization facilities (cont.)

Semaphores are kernel-maintained, global, non-negative integers. A process can request to decrement a semaphore (usually to reserve exclusive usage of a resource) or to increment it (to release exclusive usage, allowing others to go). Decrementing a 0-value semaphore blocks the caller; unblock is atomic with (future) decrement.

File locks are used to coordinate access to (regions of) a file. At any given time, multiple processes can hold read locks on (regions of) a file; but only one process can hold a write lock, which also excludes other read locks.

Mutexes and condition variables are higher-level synchronization facilities that can be used for fine-grained and event-driven coordination, which are normally used between threads.

IPC comparison — identifiers

How can you choose the IPC facility that best suite your needs?

A first discriminant are the identifiers used to rendez-vous on a IPC facility and the handles used to reference them once "opened".

Facility type	Name used to identify object	Handle used to refer to object in programs
Pipe	no name	file descriptor
FIFO	pathname	file descriptor
UNIX domain socket	pathname	file descriptor
Internet domain socket	IP address + port number	file descriptor
System V message queue	System V IPC key	System V IPC identifier
System V semaphore	System V IPC key	System V IPC identifier
System V shared memory	System V IPC key	System V IPC identifier
POSIX message queue	POSIX IPC pathname	mqd_t (message queue descriptor)
POSIX named semaphore	POSIX IPC pathname	sem_t * (semaphore pointer)
POSIX unnamed semaphore	no name	sem_t * (semaphore pointer)
POSIX shared memory	POSIX IPC pathname	file descriptor
Anonymous mapping	no name	none
Memory-mapped file	pathname	file descriptor
flock() lock	pathname	file descriptor
fcntl() lock	pathname	file descriptor

TLPI, Table 43-1

IPC comparison — functionalities

Data transfer vs shared memory

- data transfer
  - read/write + "consumable" messages (by the reader)
  - synchronization is implicit
- shared memory
  - allow sharing with many processes
  - "communication" is implicit
  - synchronization is, de facto, mandatory

IPC comparison — functionalities (cont.)

Which data transfer facility?

- byte stream vs message passing
  - the model might be forced by your application protocol
  - byte stream can be used to do message passing
- pipes, FIFOs, and sockets use file descriptors as handles
  - many advanced I/O functionalities expect such handles (e.g. `select`, `poll`)
- specific needs:
  - numeric priorities → message queues
  - message notification → POSIX(!) message queues
  - networking → UNIX domain sockets easily scale to internet socket
  - broadcast/multicast to multiple recipients → UDP sockets
  - file descriptor passing → UNIX domain sockets

IPC comparison — portability and System V IPC

Modern UNIX implementations support most of the UNIX IPC facilities we've discussed.

As an exception, POSIX IPC (message queues, sempahores, shared memory) are still catching up and are less widely available than their System V counterparts.

- e.g. POSIX IPC landed on Linux only from 2.6.x onward

System V IPC design issues

- System V IPC are connection-less → there is no way to know when to garbage collect them (for the kernel), or when it's safe to delete them (for an application)
- Weird namespace, inconsistent with the traditional "everything is a file" UNIX model

If you are looking at SysV-like IPC, either choose POSIX IPC or go for something completely different.

IPC comparison — accessibility & persistence

The two last axes for IPC comparison are:

accessibility i.e. which permission mechanism is used to control access to the IPC facility. Common cases are control by filesystem permission masks, virtual memory access control, free access, and access limited to related processes (for IPC facilities that are meant to be inherited upon `fork`).

persistence whether an IPC facility and its content persists as long as the (last) process who is using it, the kernel, or the filesystem

IPC comparison — accessibility & persistence (cont.)

Facility type	Accessibility	Persistence
Pipe	only by related processes	process
FIFO	permissions mask	process
UNIX domain socket	permissions mask	process
Internet domain socket	by any process	process
System V message queue	permissions mask	kernel
System V semaphore	permissions mask	kernel
System V shared memory	permissions mask	kernel
POSIX message queue	permissions mask	kernel
POSIX named semaphore	permissions mask	kernel
POSIX unnamed semaphore	permissions of underlying memory	depends
POSIX shared memory	permissions mask	kernel
Anonymous mapping	only by related processes	process
Memory-mapped file	permissions mask	file system
flock() file lock	open() of file	process
fcntl() file lock	open() of file	process

TLPI, Table 43-2

Outline

A brief history of UNIX pipes

Pipes are the oldest form of IPC on UNIX systems—pipes are one of the early *defining* features of UNIX-es, together with hierarchical file system and widespread regular expression usage.

late 50's McIlroy's seminal work on macros, as powerful constructs to *compose* commands

M. Douglas McIlroy
Macro Instruction Extensions of Compiler Languages
Communications of the ACM (3)4: 214–220. 1960.

1969 development of the first UNIX at Bell Labs

1973 first implementation of shell pipes in Bell Labs Unix by Ken Thompson

UNIX pipes in the shell — examples

UNIX pipes (i.e. the IPC mechanism) are the main building block of shell pipes (i.e. the "|" meta-character).

- `ps auxw | more`
  - no need to implement a pager in every program with long output
  - write once, use many (consistently)
  - can fix pager bugs in a central place
- `ps auxw | less`
  - enable users to choose a different pager
  - "less is more"
- `tr -c '[:alnum:]' '[\n*]' | sort -iu | grep -v '^[0-9]*$'`
  - enable to express complex tasks concisely, in terms of simple tools
- a pipe-based relational database (!)

  Evan Schaffer, Mike Wolf.
  The UNIX Shell As a Fourth Generation Language
  http://www.rdb.com/lib/4gl.pdf

Pipes — IPC characteristics

Let's review UNIX pipes against the general IPC characteristics we put forward:

- pipes are a data transfer, byte stream IPC facility that connect processes; the byte stream written to one end of the pipe can be read from the other
- no identifier is used to rendez-vous on pipes, they are requested directly to the kernel
- once created, pipes are referenced by file descriptor handles
- pipes are accessible only by related processes
- pipes are process-persistent; they disappear when related processes terminate
- pipes are highly portable: they are available on all known UNIX-es

pipe

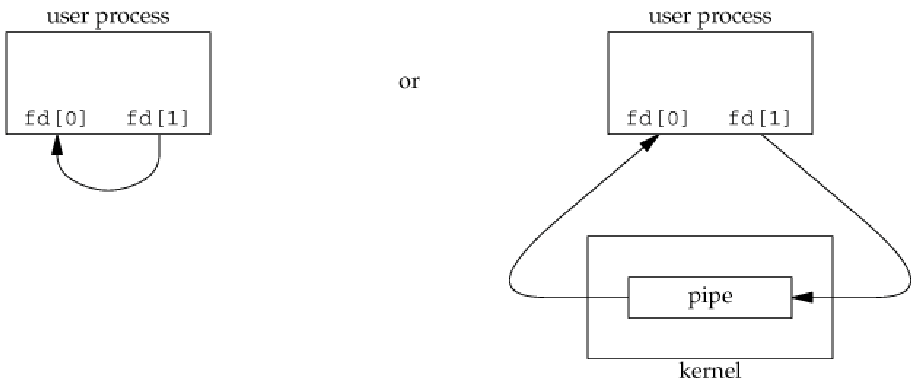
The creation of a pipe can be requested to the kernel using the `pipe` system call:

```
#include <unistd.h>

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

Returns: *0 if OK, 1 on error*

- `filedes` is an array of file descriptors; it should be allocated by the caller and will be filled-in by the kernel before returning
- `filedes[0]` is open for reading (*read-end*), `filedes[1]` is open for writing (*write-end*)
  - mnemonic: think of usual STDIN/STDOUT values
- the output of `filedes[1]` is the input of `filedes[0]`
  - pipes are half-duplex

Pipes — intuition



APUE, Figure 15.2

- on the left, the user process point of view
- on the right, the implementation point of view
  - every read from a pipe copy from kernel space to user space
  - every write to a pipe copy from user space to kernel space

Pipes — usage

As they are, pipes seem pretty useless: they only allow a process to write data to a file descriptor and read it back from another.

Pipes become most useful by exploiting the fact that file descriptors are inherited through fork.

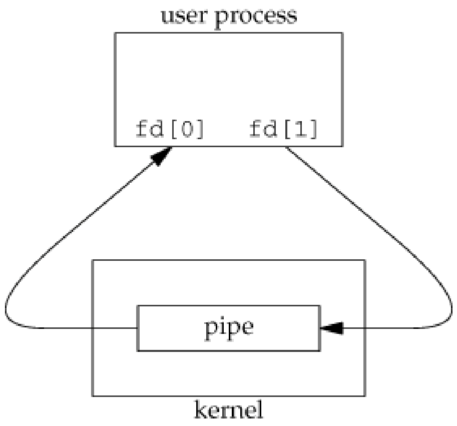
Half-duplex pipe recipe

1. `pipe(fds)`
2. `fork()`
3. parent: `close(fds[0])`
4. child: `close(fds[1])`
5. parent can transfer data to child with `write(fds[1], ...)`
   child can receive data from parent with `read(fds[0], ...)`

(exchange 0 and 1 for child to parent data transfer)

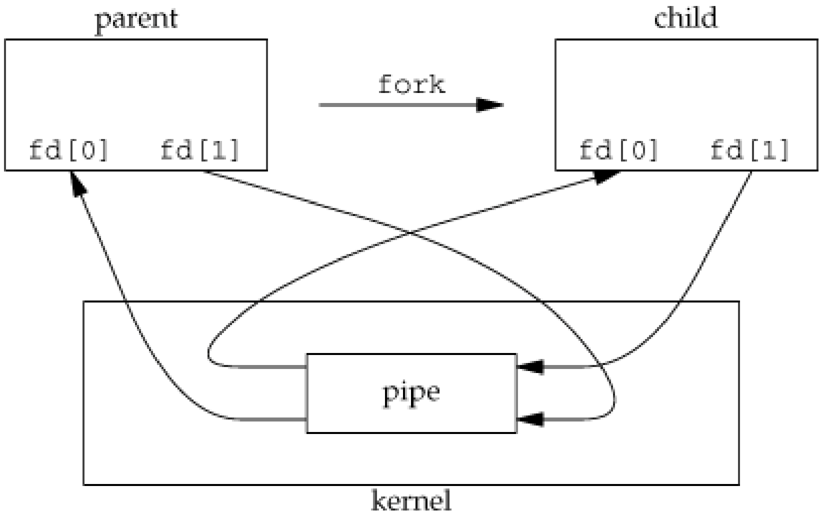
Pipes — usage (cont.)

after `pipe()...`



Pipes — usage (cont.)

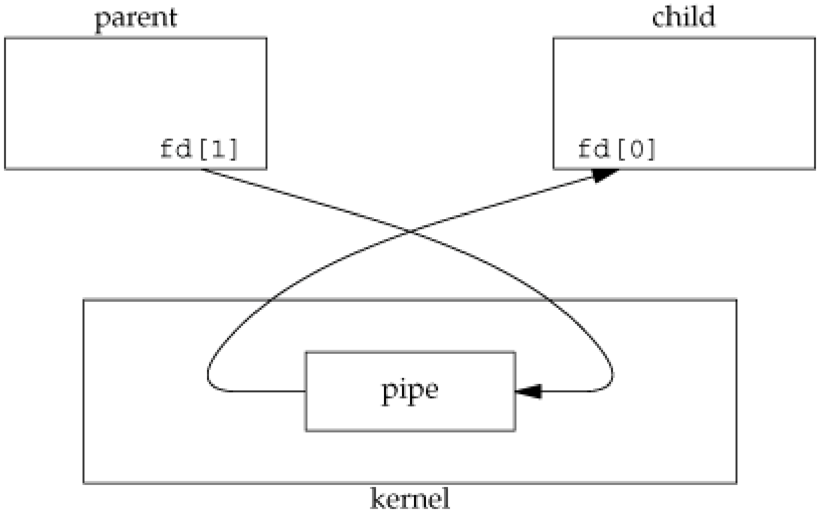
after fork()...



APUE, Figure 15.3

Pipes — usage (cont.)

after `close()`-ing unused ends...



APUE, Figure 15.4

Pipes — example

```
#include <unistd.h>
#include "helpers.h"

#define MAXLINE 1024
int main(void) {
        int n, fd[2];
        pid_t pid;
        char line[MAXLINE];

        if (pipe(fd) < 0)
                err_sys("pipe error");
        if ((pid = fork()) < 0) {
                err_sys("fork error");
        } else if (pid > 0) {    /* parent */
                close(fd[0]);
                write(fd[1], "Hello, World!\n", 14);
        } else {                 /* child */
                close(fd[1]);
                n = read(fd[0], line, MAXLINE);
                write(STDOUT_FILENO, line, n);
        }
        exit(EXIT_SUCCESS);
}
```

Pipes — example

Demo

Pipes — close behavior

The unused ends of a pipe are usually closed before starting to use a pipe. There are also legitimate reasons for closing the *used* ends, e.g. when one process wants to shutdown the communication.

Performing I/O on a pipe with closed end behaves as follows:

- `read` from a pipe whose write end is closed returns 0
  - intuition: indicate there is nothing else to read; 0 is the standard way of `read` to signal end-of-file
- `write` to a pipe whose read end is closed returns -1, with `errno` set to EPIPE; additionally, SIGPIPE is sent to the writing process
  - this is a new, pipe-specific condition
  - reminder: SIGPIPE default action is terminate

Outline

Filters

Pipes, as seen thus far, can be used to establish ad-hoc communication channels (half- or full-duplex) between processes.

Pipes become even more relevant in conjunction with UNIX filters.

Definition (UNIX filter)

In the UNIX jargon, a filter is a program that gets (most of) its *input from standard input* and writes (most of) its *output to standard output*.

Example

Many of the standard POSIX.1 command-line utilities are filters: awk, cat, cut, grep, head, sed, sort, strings, tail, tac, tr, uniq, wc, . . .

Pipes and filters

Consider a program of yours that wants to paginate its output. Ideally, you want to use the system pager (e.g. `more`) instead of writing your own.

How can you do that (with `pipe`)?

Pipes and filters

Consider a program of yours that wants to paginate its output. Ideally, you want to use the system pager (e.g. `more`) instead of writing your own.

1. `pipe`
2. `fork`
   - idea: parent will produce the content to be paginated,
   - child will execute external pager
3. child: duplicate the read end of the pipe on STDIN
   - when reading from STDIN, child will in fact read from the pipe
4. child: exec the pager
   - as the pager is a filter, it will read from STDIN by default
5. parent: write output to the write end of the pipe

Note: this is possible thanks to the `fork`/`exec` separation that allows to manipulate file descriptors in between.

Pipes and filters — example

```
#include <stdio.h>
#include <string.h>
#include <sys/wait.h>
#include <unistd.h>
#include "helpers.h"

#define DEF_PAGER       "/bin/more"
#define MAXLINE         1024
int main(int argc, char *argv[]) {
        int n, fd[2];
        pid_t pid;
        char *pager, *argv0;
        char line[MAXLINE];
        FILE *fp;

        if (argc != 2)
                err_quit("Usage: pager-pipe FILE");
        if ((fp = fopen(argv[1], "r")) == NULL)
                err_sys("fopen error");
        if (pipe(fd) < 0)
                err_sys("pipe error");
        if ((pid = fork()) < 0)
                err_sys("fork error");
```

Pipes and filters — example (cont.)

```
else if (pid > 0) {      /* parent */
        close(fd[0]);    /* close read end */
        /* parent copies from argv[1] to pipe */
        while (fgets(line, MAXLINE, fp) != NULL) {
                n = strlen(line);
                if (write(fd[1], line, n) != n)
                        err_sys("write error");
        }
        if (ferror(fp))
                err_sys("fgets error");
        close(fd[1]);    /* close write end of pipe for reader */
        if (waitpid(pid, NULL, 0) < 0)  err_sys("waitpid error");
}
```

Pipes and filters — example (cont.)

```
        else {  /* child */
                close(fd[1]);     /* close write end */
                if (fd[0] != STDIN_FILENO) {
                        if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
                                err_sys("dup2 error");
                        close(fd[0]);    /* no longer needed */
                }
                /* get arguments for execl() */
                if ((pager = getenv("PAGER")) == NULL)
                        pager = DEF_PAGER;
                if ((argv0 = strrchr(pager, '/')) != NULL)
                        argv0++;         /* step past rightmost slash */
                else
                        argv0 = pager;   /* no slash in pager */

                if (execl(pager, argv0, (char *)0) < 0)
                        err_sys("execl error");
        }
        exit(EXIT_SUCCESS);
}
/* pager-pipe.c */
/* based on APUE, Figure 15.6 */
```

Pipes and filters — example (cont.)

Demo

Notes:

- $PAGER is a UNIX convention to allow users to set their preferred pager, system-wide; we are good citizens and try to respect it
- dup2 does nothing if new and old file descriptors are the same. We are careful to avoid shutting down the pipe
  - Here it *should* never be the case: if the shell didn't setup STDIN, fd 0 would have been taken by fopen.
    We do it nonetheless as a defensive programming measure.

Implementing shell redirections

How can we implement `cmd > file` shell redirection?

Implementing shell redirections

How can we implement `cmd > file` shell redirection?
No pipes needed. File descriptor inheritance through `fork` and `fork`/`exec` separations are enough. Recipe:

1. `fork`
2. child: `open(file, ...) = fd`
3. child: duplicate `fd` onto STDOUT
4. child: `exec cmd`

Implementing shell redirections

How can we implement `cmd > file` shell redirection?
No pipes needed. File descriptor inheritance through `fork` and `fork/exec` separations are enough. Recipe:

1. `fork`
2. child: `open(file, ...) = fd`
3. child: duplicate `fd` onto STDOUT
4. child: `exec cmd`

How about `cmd < file`?

Implementing shell redirections

How can we implement `cmd > file` shell redirection?
No pipes needed. File descriptor inheritance through `fork` and `fork`/`exec` separations are enough. Recipe:

1. `fork`
2. child: `open(file, ...) = fd`
3. child: duplicate `fd` onto STDOUT
4. child: exec cmd

How about `cmd < file`?

1. `fork`
2. child: `open(file, ...) = fd`
3. child: duplicate `fd` onto STDIN
4. child: exec cmd

Implementing shell pipelines

How can we implement the `cmd1 | cmd2` pipeline construct?

Implementing shell pipelines

How can we implement the `cmd1 | cmd2` pipeline construct?
With a generalization of the mechanism we have seen:

1. pipe
2. fork, fork (once per command)
3. 1st child: duplicate write end of the pipe to STDOUT
4. 2nd child: duplicate read end of the pipe to STDIN
5. 1st child: exec cmd1
6. 2nd child: exec cmd2

Implementing shell pipelines

How can we implement the `cmd1 | cmd2` pipeline construct?
With a generalization of the mechanism we have seen:

1. pipe
2. fork, fork (once per command)
3. 1st child: duplicate write end of the pipe to STDOUT
4. 2nd child: duplicate read end of the pipe to STDIN
5. 1st child: exec cmd1
6. 2nd child: exec cmd2

Exercise (minimal shell)

Implement a minimal shell with support for n-ary pipes, file redirections, and command conditionals (e.g. ||, &&). The shell should properly handle CTRL-C, CTRL-\ and signals.

Full-duplex communication with pipes

Once more: pipes are half-duplex

- one pipe can be used to transfer data in one direction only, *either* from parent to child *or* from child to parent
- full-duplex pipes do exist, but are less portable and seldomly used (they are an optional feature of SUS)

To do portable full-duplex communication with pipes (i.e. transfer data in both directions), 2 `pipe` calls before `fork` are needed:

Full-duplex pipe recipe

1. `pipe(p2c); pipe(c2p)`
2. `fork()`
3. parent: `close(p2c[0]); close(c2p[1])`
4. child: `close(p2c[1]); close(c2p[0])`
5. parent → child: `write(p2c[1], ...)`
   child → parent: `write(c2p[1], ...)`

Pipe-based synchronization

Can we use pipes as IPC *synchronization* primitives?

Pipe-based synchronization

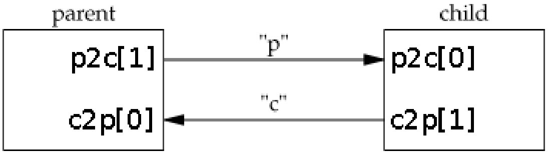
Pipes are data transfer IPC primitives. Nonetheless, we can exploit the fact that read is blocking by default to perform pipe-based synchronization between related processes.
To that end, we give a pipe-based implementation of the TELL/WAIT synchronization primitives.
Reminder:

```
int main(void) {
    pid_t pid;

    TELL_WAIT();

    if ((pid = fork()) < 0)  err_sys("fork error");
    else if (pid == 0) {
        WAIT_PARENT();    /* parent first */
        charatatime("output from child\n");
    } else {
        charatatime("output from parent\n");
        TELL_CHILD(pid);
    }
    exit(EXIT_SUCCESS);
}
```

Pipe-based synchronization — idea

- before fork, upon initialization, we set up two pipes for full-duplex communication between parent and child
- to wait for the child (resp. parent), we read from the pipe the control character "c" (resp. "p")
- to signal the child (parent), we write the control character "p" ("c") to the pipe

Note: *what* we actually write is not relevant, the fact we do is.



based on APUE, Figure 15.8

Pipe-based synchronization — implementation

```
static int p2c[2], c2p[2];

void TELL_WAIT(void) {  /* initialization */
        if (pipe(p2c) < 0 || pipe(c2p) < 0)
                err_sys("pipe error");
}

void TELL_PARENT(pid_t pid) {
        if (write(c2p[1], "c", 1) != 1)
                err_sys("write error");
}

void WAIT_PARENT(void) {
        char    c;
        if (read(p2c[0], &c, 1) != 1)
                err_sys("read error");
        if (c != 'p')
                err_quit("WAIT_PARENT: incorrect data");
}
```

Pipe-based synchronization — implementation (cont.)

```
void TELL_CHILD(pid_t pid) {
        if (write(p2c[1], "p", 1) != 1)
                err_sys("write error");
}

void WAIT_CHILD(void) {
        char    c;

        if (read(c2p[0], &c, 1) != 1)
                err_sys("read error");

        if (c != 'c')
                err_quit("WAIT_CHILD: incorrect data");
}
```

popen

The following use cases of pipes are recurrent patterns:

1. `pipe+fork+dup2+exec` to read from stdout of some command
2. `pipe+fork+dup2+exec` to write to stdin of some command

To reduce boilerplate, the standard C library offers popen:

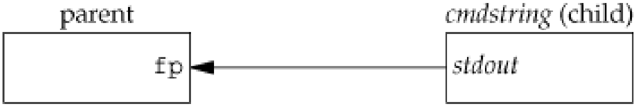
```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
```

Returns: *file pointer if OK, NULL on error*

- `cmdstring` is as per `system`, i.e. a shell command that will be interpreted by `/bin/sh -c`
- `type` discriminates among the two use cases: it's `'r'` for (1) and `'w'` for (2)
- the returned `FILE` handle is open for reading or writing, depending on the use case

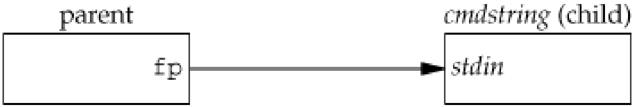
popen — process arrangements

```
fp = popen(cmdstring, "r")
```



APUE, Figure 15.9

```
fp = popen(cmdstring, "w")
```



APUE, Figure 15.10

pclose

To cleanup after using popen, more behind the scene work is needed than simply closing the FILE pointer—in particular, child process should be wait-ed for to avoid leaving zombies around.

The pclose syscall takes care of all the gory details and returns the termination status of the child process to the caller.

```
#include <stdio.h>

int pclose(FILE *fp);
```

Returns: *termination status of command if OK, 1 on error*

---

popen — example

```c
#include <sys/wait.h>
#include "helpers.h"

#define PAGER   "${PAGER:-more}" /* environment variable, or default */
#define MAXLINE 1024
int main(int argc, char *argv[]) {
        char line[MAXLINE];
        FILE *fpin, *fpout;

        if (argc != 2)  err_quit("usage: pager-popen FILE");
        if ((fpin = fopen(argv[1], "r")) == NULL)  err_sys("fopen error");
        if ((fpout = popen(PAGER, "w")) == NULL)   err_sys("popen error");

        /* copy argv[1] to pager */
        while (fgets(line, MAXLINE, fpin) != NULL) {
                if (fputs(line, fpout) == EOF)
                        err_sys("fputs error to pipe");
        }
        if (ferror(fpin))
                err_sys("fgets error");
        if (pclose(fpout) == -1)
                err_sys("pclose error");
        exit(EXIT_SUCCESS);
} /* end of pager-popen.c, based on APUE, Figure 15.11 */
```

popen — example (cont.)

Demo

Notes:

- code is much shorter now!
- we use shell special characters on the popen line

popen — implementation

Exercise

Provide an implementation of popen/`pclose` using the system calls we have seen thus far.

Watch out for the following details:

- keep track of all children that are currently executing popen "jobs" and maintain a mapping from FILE pointers to them
  - it's the only way to be able to `waitpid` for them when client code will invoke `pclose`
- ensure that signal handling in the popen caller does not interfere with popen jobs

popen and filters

We can use popen-like arrangements to interpose external processes between an application and its standard input/output.

Example

Consider an application that prompts the user and read line-based commands (AKA read-eval-print loop). We would like to delegate to a filter the task to normalize case to lowercase.

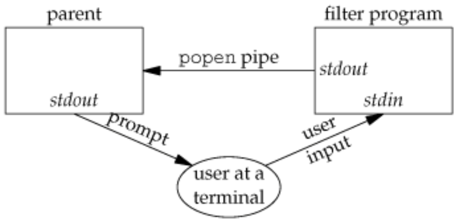
popen and filters

We can use popen-like arrangements to interpose external processes between an application and its standard input/output.

Example

Consider an application that prompts the user and read line-based commands (AKA read-eval-print loop). We would like to delegate to a filter the task to normalize case to lowercase.

We can do so with the following process arrangement:



APUE, Figure 15.13

- popen("r") affects STDOUT of the child process, but leaves untouched its STDIN
- STDIN is shared with the parent (as per fork), but the parent will (usually) only read it through popen's FILE pointer

popen and filters — example

```
#include <ctype.h>
#include <unistd.h>
#include "helpers.h"

int main(void) {
        int c;

        while ((c = getchar()) != EOF) {
                if (isupper(c))
                        c = tolower(c);
                if (putchar(c) == EOF)
                        err_sys("output error");
                if (c == '\n')
                        fflush(stdout);
        }
        exit(EXIT_SUCCESS);
} /* uc2lc.c, based on APUE Figure 15.14 */
```

popen and filters — example (cont.)

```
#include <sys/wait.h>
#include <unistd.h>
#include "helpers.h"

#define MAXLINE 1024

int main(void) {
        char line[MAXLINE];
        FILE    *fpin;
        if ((fpin = popen("./uc2lc", "r")) == NULL)
                err_sys("popen error");
        for ( ; ; ) {
                fputs("prompt> ", stdout);
                fflush(stdout);
                if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
                        break;
                if (fputs(line, stdout) == EOF)
                        err_sys("fputs error to pipe");
        }
        if (pclose(fpin) == -1)
                err_sys("pclose error");
        putchar('\n');
        exit(EXIT_SUCCESS);
} /* popen-filter.c, based on APUE, Figure 15.15 */
```

popen and filters — example (cont.)

Demo

Notes:

- we need `fflush` after prompt, because STDOUT is line-buffered by default and the prompt does not end with a newline

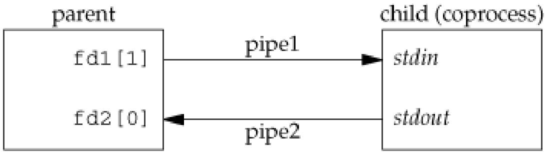
Coprocesses

Filters are usually connected linearly to form a pipeline.

Definition

A filter is *used as* a coprocess, when the process that drives the filer both (i) generates its input and (ii) read its output.

Coprocess architectures offer modularity in terms of separate programs that communicate as filters.
Process arrangement with coprocesses is the usual full-duplex pipe arrangement. The main difference is that the child process is a filter, which ignores that is being used as a coprocess.



APUE, Figure 15.16

Coprocesses — example

```c
#include <string.h>
#include <unistd.h>
#include "helpers.h"

#define MAXLINE 1024
int main(void) {
        int n, int1, int2;
        char line[MAXLINE];

        while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
                line[n] = 0;            /* null terminate */
                if (sscanf(line, "%d%d", &int1, &int2) == 2) {
                        sprintf(line, "%d\n", int1 + int2);
                        n = strlen(line);
                        if (write(STDOUT_FILENO, line, n) != n)
                                err_sys("write error");
                } else {
                        if (write(STDOUT_FILENO, "invalid args\n", 13) != 13
                                err_sys("write error");
                }
        }
        exit(EXIT_SUCCESS);
} /* add2.c, based on APUE Figure 15.17 */
```

Coprocesses — example (cont.)

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include "helpers.h"

static void sig_pipe(int signo) {
        printf("SIGPIPE caught\n");
        exit(EXIT_FAILURE);
}

#define MAXLINE 1024
int main(void) {
        int n, fd1[2], fd2[2];
        pid_t pid;
        char line[MAXLINE];

        if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
                err_sys("signal error");
        if (pipe(fd1) < 0 || pipe(fd2) < 0)
                err_sys("pipe error");
        if ((pid = fork()) < 0)
                err_sys("fork error");
```

Coprocesses — example (cont.)

```
else if (pid > 0) {     /* parent */
        close(fd1[0]);
        close(fd2[1]);
        while (fgets(line, MAXLINE, stdin) != NULL) {
                n = strlen(line);
                if (write(fd1[1], line, n) != n)
                        err_sys("write error to pipe");
                if ((n = read(fd2[0], line, MAXLINE)) < 0)
                        err_sys("read error from pipe");
                if (n == 0) {
                        fprintf(stderr, "child closed pipe");
                        break;
                }
                line[n] = 0;     /* null terminate */
                if (fputs(line, stdout) == EOF)
                        err_sys("fputs error");
        }
        if (ferror(stdin))  err_sys("fgets error on stdin");
        exit(EXIT_SUCCESS);
}
```

Coprocesses — example (cont.)

```
        else {              /* child */
                close(fd1[1]);
                close(fd2[0]);
                if (fd1[0] != STDIN_FILENO) {
                        if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
                                err_sys("dup2 error to stdin");
                        close(fd1[0]);
                }
                if (fd2[1] != STDOUT_FILENO) {
                        if (dup2(fd2[1], STDOUT_FILENO) != STDOUT_FILENO)
                                err_sys("dup2 error to stdout");
                        close(fd2[1]);
                }
                if (execl("./add2", "add2", (char *)0) < 0)
                        err_sys("execl error");
        }
        exit(EXIT_SUCCESS);
} /* coprocess.c */
```

Coprocesses — example (cont.)

Demo

Notes:

- the coprocess is resilient to failures, e.g. it does not quit upon (recoverable) error
- if we kill add2, parent process won't die immediately but will get a SIGPIPE at the next write
  - he can recover from that spawning the coprocess again!
  - in some sense, we can replace (e.g. for upgrade reasons) components of our "application" at runtime

Buffering issues

What would happen if we rewrite the add2 coprocess to use standard I/O instead of low-level syscall I/O as follows?

```
#include <stdio.h>
#include <unistd.h>
#include "helpers.h"

#define MAXLINE 1024
int main(void) {
        int int1, int2;
        char line[MAXLINE];
        while (fgets(line, MAXLINE, stdin) != NULL) {
                if (sscanf(line, "%d%d", &int1, &int2) == 2) {
                        if (printf("%d\n", int1 + int2) == EOF)
                                err_sys("printf error");
                } else {
                        if (printf("invalid args\n") == EOF)
                                err_sys("printf error");
                }
        }
        exit(EXIT_SUCCESS);
} /* add2-stdio-bad.c, based on APUE Figure 15.19 */
```

Why?

Buffering issues (cont.)

Demo

Notes:

- our coprocess-based architecture no longer works
- the (usual) culprit is standard I/O buffering
- standard I/O is line-buffered by default when connected to a terminal, but since the coprocess is connected to a pipe it becomes fully buffered
- to fix the problem, we have to set line buffering explicitly

Buffering issues — (cont.)

```c
#include <stdio.h>
#include <unistd.h>
#include "helpers.h"

#define MAXLINE 1024
int main(void) {
        int int1, int2;
        char line[MAXLINE];

        if (setvbuf(stdin, NULL, _IOLBF, 0) != 0
            || setvbuf(stdout, NULL, _IOLBF, 0) != 0)
                err_sys("setvbuf error");

        while (fgets(line, MAXLINE, stdin) != NULL) {
                if (sscanf(line, "%d%d", &int1, &int2) == 2) {
                        if (printf("%d\n", int1 + int2) == EOF)
                                err_sys("printf error");
                } else {
                        if (printf("invalid args\n") == EOF)
                                err_sys("printf error");
                }
        }
        exit(EXIT_SUCCESS);
} /* add2-stdio-good.c */
```

Preview: on the need of pseudoterminals

But we cheated!

One of the nice property of filters is that they speak a simple "protocol" (stdin/stout), as such they can be *used as* coprocess without modifications. On the other hand, to use the standard I/O implementation of the add2 filter as a coprocess we had to patch it (the filter).
We can't patch all existing filters. . . .

Example

We'd like to use the following awk script as coprocess

```
#!/usr/bin/awk -f
{ print $1 + $2 }
```

unfortunately, it won't work as a coprocess due to awk (legitimate!) buffer behavior. . .

Preview: on the need of pseudoterminals

But we cheated!

One of the nice property of filters is that they speak a simple "protocol" (stdin/stout), as such they can be *used as* coprocess without modifications. On the other hand, to use the standard I/O implementation of the add2 filter as a coprocess we had to patch it (the filter).
We can't patch all existing filters. . . .

Example

We'd like to use the following awk script as coprocess

```
#!/usr/bin/awk -f
{ print $1 + $2 }
```

unfortunately, it won't work as a coprocess due to awk (legitimate!) buffer behavior. . .

The solution is to make the coprocess believe that it is connected to a terminal, so that standard I/O becomes line buffered again.
Pseudoterminals will allow us to do precisely that. . .