



系統程式

郭大維教授/施吉昇教授
臺灣大學資訊工程系

Contents

1. Preface/Introduction
2. Standardization and Implementation
3. File I/O
4. Standard I/O Library
5. Files and Directories
6. System Data Files and Information
7. Environment of a Unix Process
8. Process Control
9. Signals
- 10. Inter-process Communication

Interprocess Communication

- Except for communicating via the file system, processes can communicate via various IPCs.

IPC type	POSIX.1	XPG3	V7	SVR2	SVR3.2	SVR4	4.3BSD	4.3+BSD
pipes (half duplex)	●	●	●	●	●	●	●	●
FIFOs	●	●		●	●	●		
Stream pipes (full duplex)					●	●	●	●
named stream pipes					●	●	●	●
message queues		●		●	●	●		
semaphores		●		●	●	●		
shared memory		●		●	●	●		
sockets						●	●	●
streams					●	●		

Pipes

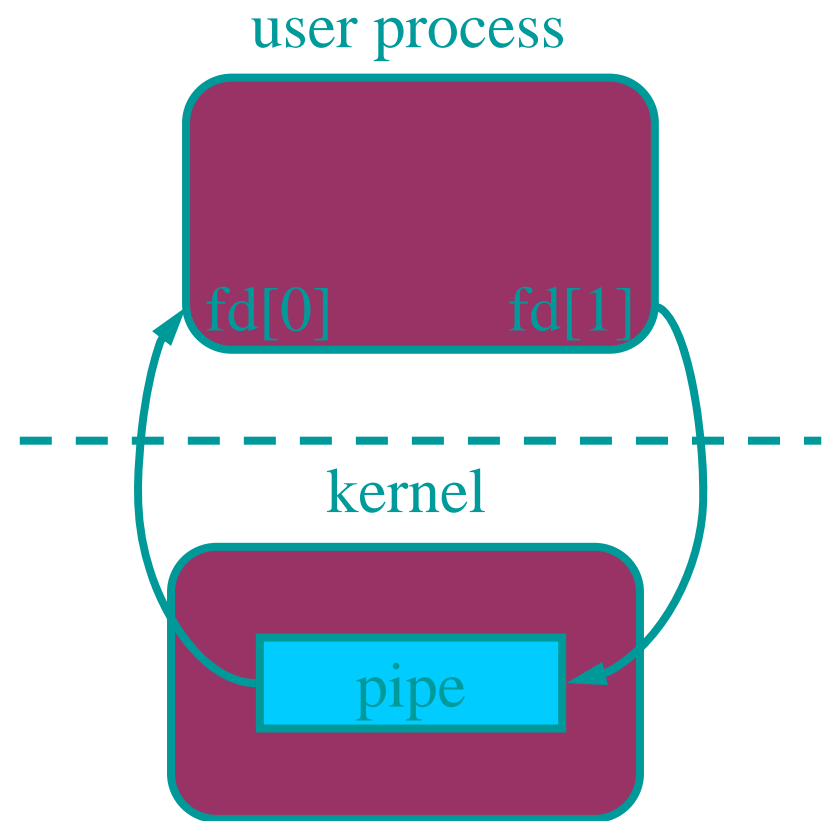
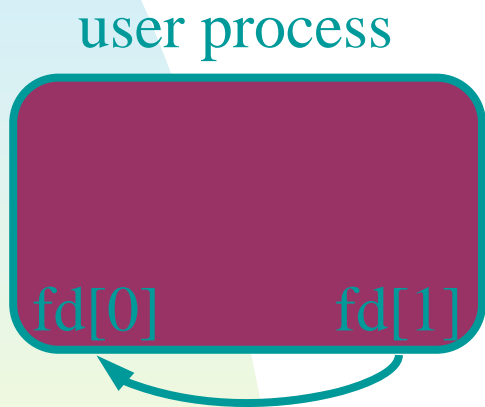
- Pipes have two limitations:
 - They are half-duplex.
 - They can be used between processes that have a common ancestor.
- Stream pipes are duplex and named stream pipes can be used between process not having the same ancestor.

- `pipe` function

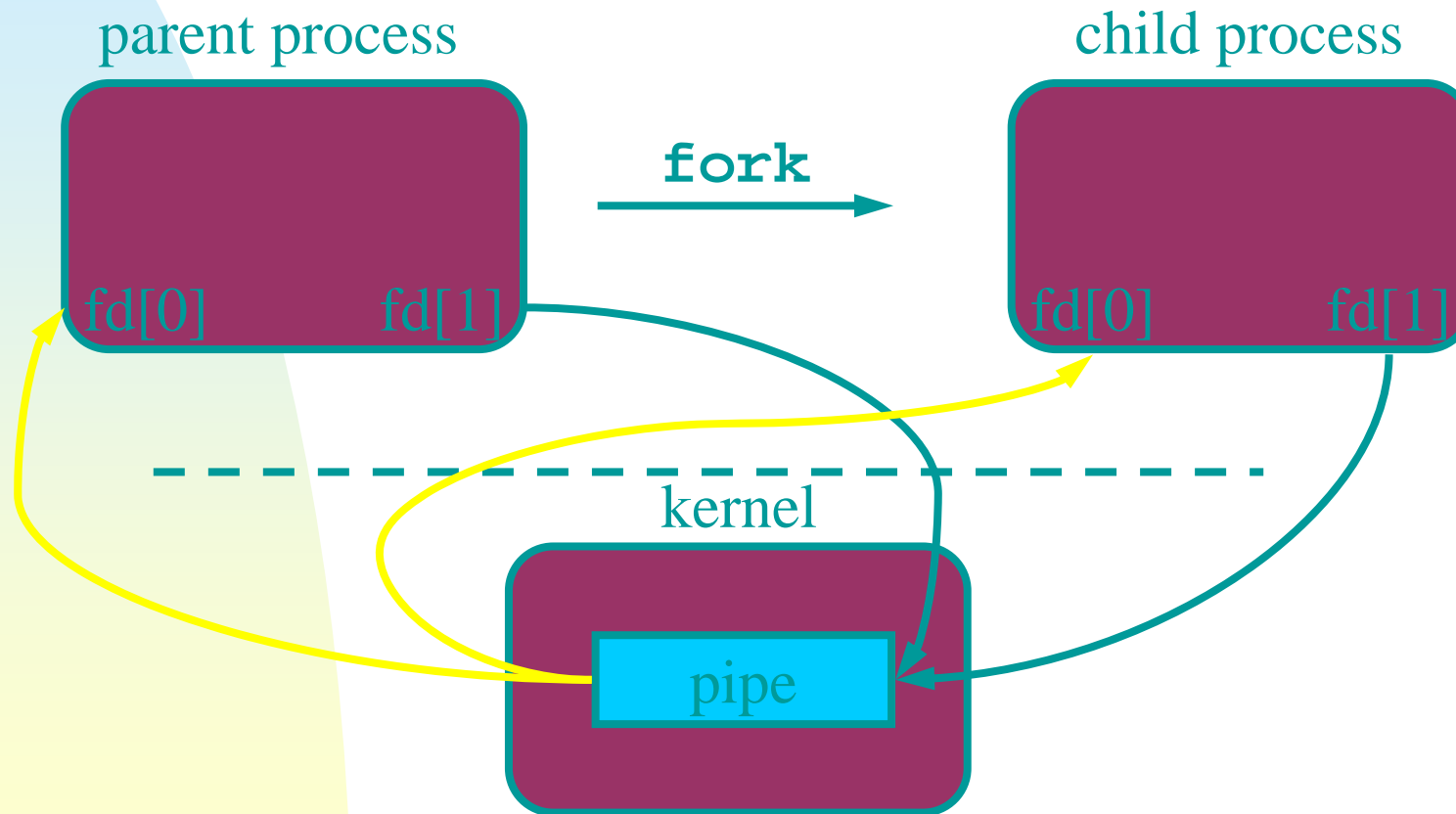
```
#include <unistd.h>
```

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

Unix pipes

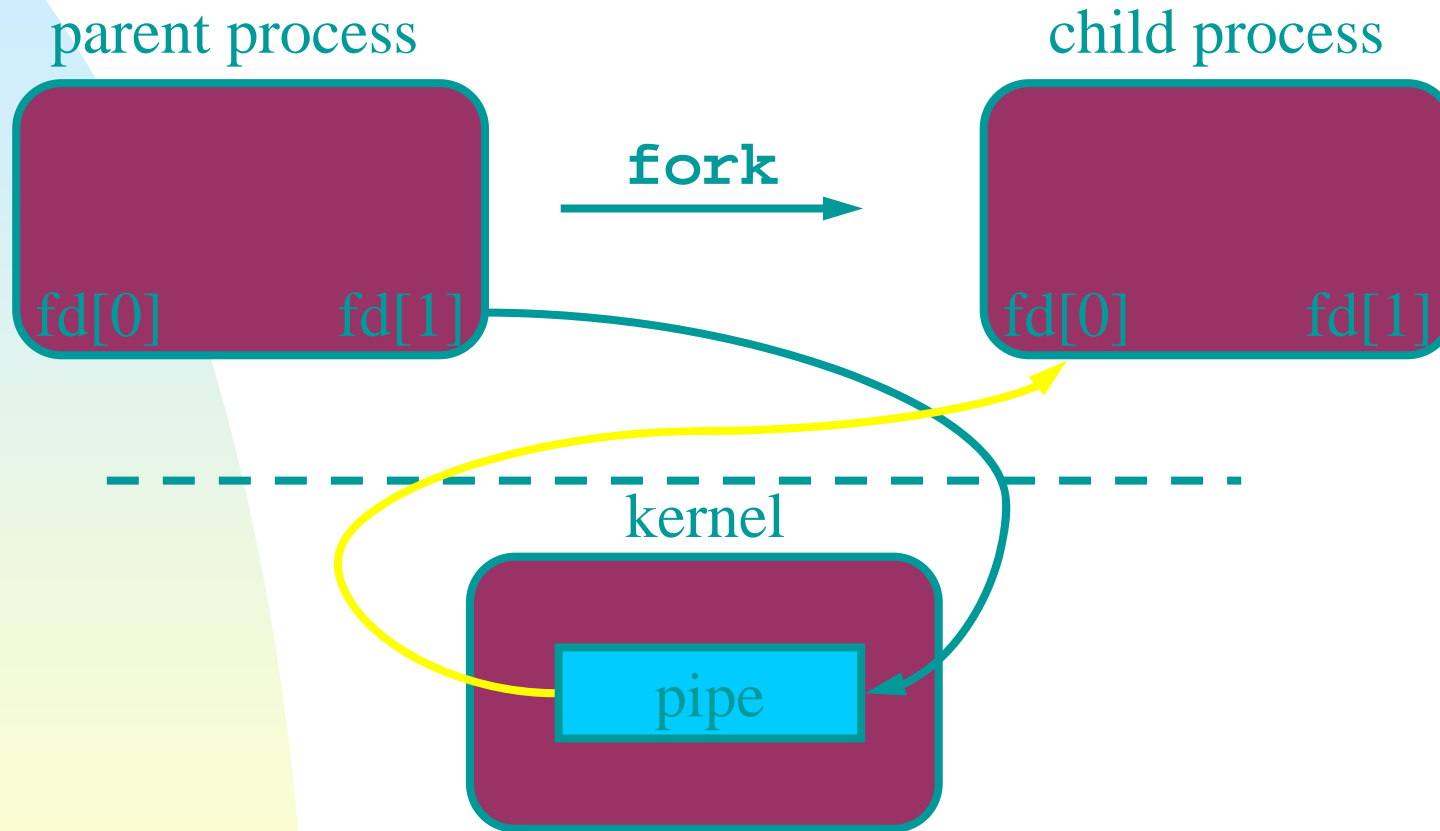


Pipes between processes



Half-duplex pipe after a **fork**

Useful pipe

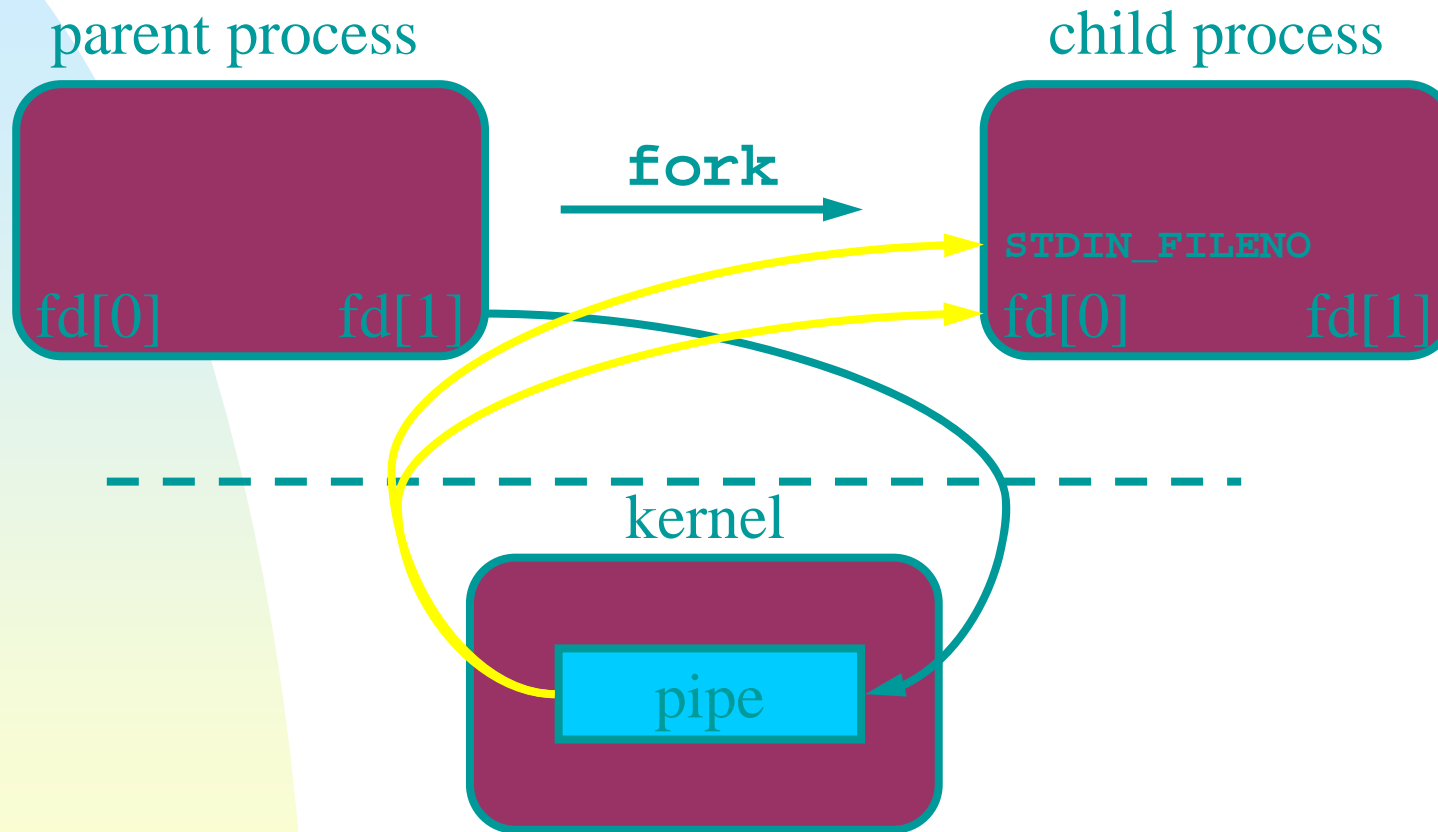


Pipe from parent process to child process

Read from/Write to pipes

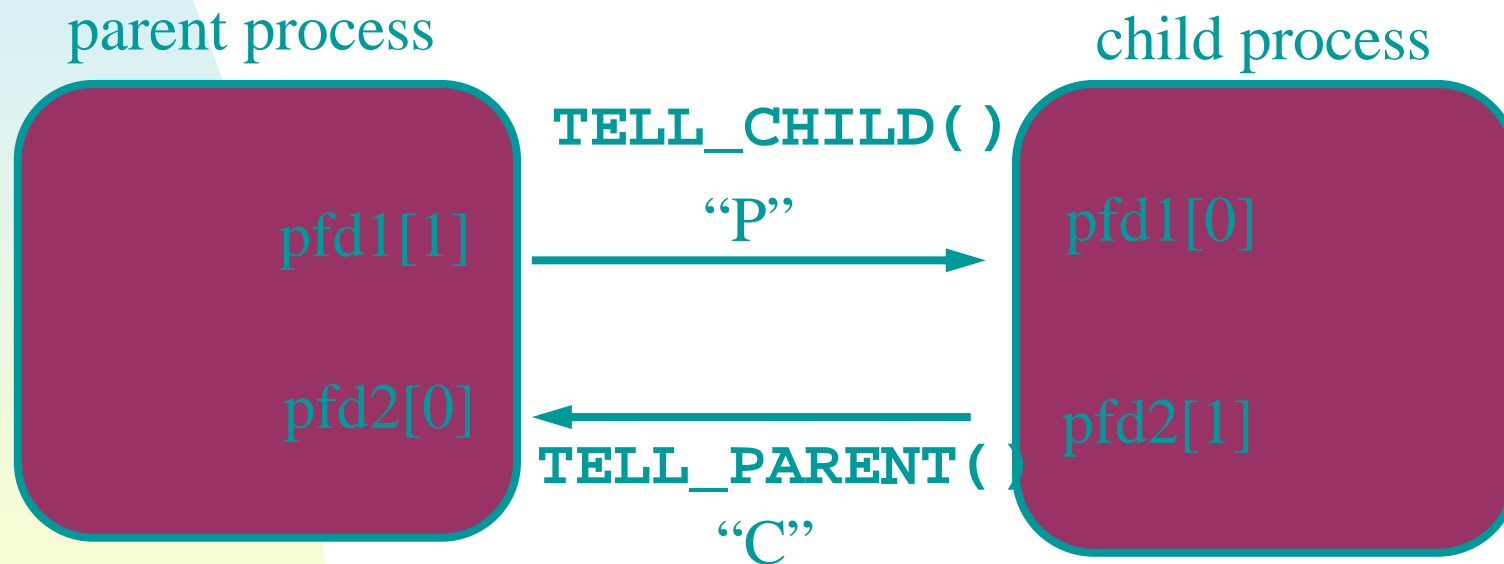
- When the write end is closed, reading from a pipe returns 0
- When the read end of a pipe is closed, writing to a pipe triggers `SIGPIPE` signal.
- Pipe buffer size:
 - Constant `PIPE_BUF` specifies the pipe buffer size.
 - Writing `PIPE_BUF` or less will not be interleaved.
- Program 14.1: creating a pipe from the parent to the child and send data down the pipe.

Connecting pipes to standard input/output



Pipe from parent process to child process

Using pipes for parent-child synchronization



`popen` and `pclose` function

- Two functions handling all the dirty works:
 - the creation of a pipe,
 - the `fork` of a child,
 - closing the unused ends of the pipe,
 - `executing` a shell to execute the command, and
 - `waiting` for the command to terminate.
- Functions:

```
# include <stdio.h>
FILE *popen(const char *cmdstring, const char *type);
int pclose(FILE *fp);
```
- Rewrite 14.2 using `popen` and `pclose`

fp = popen(*command*, "r")

parent process

cmdstring(child)



fp = popen(*command*, "w")

parent process

cmdstring(child)

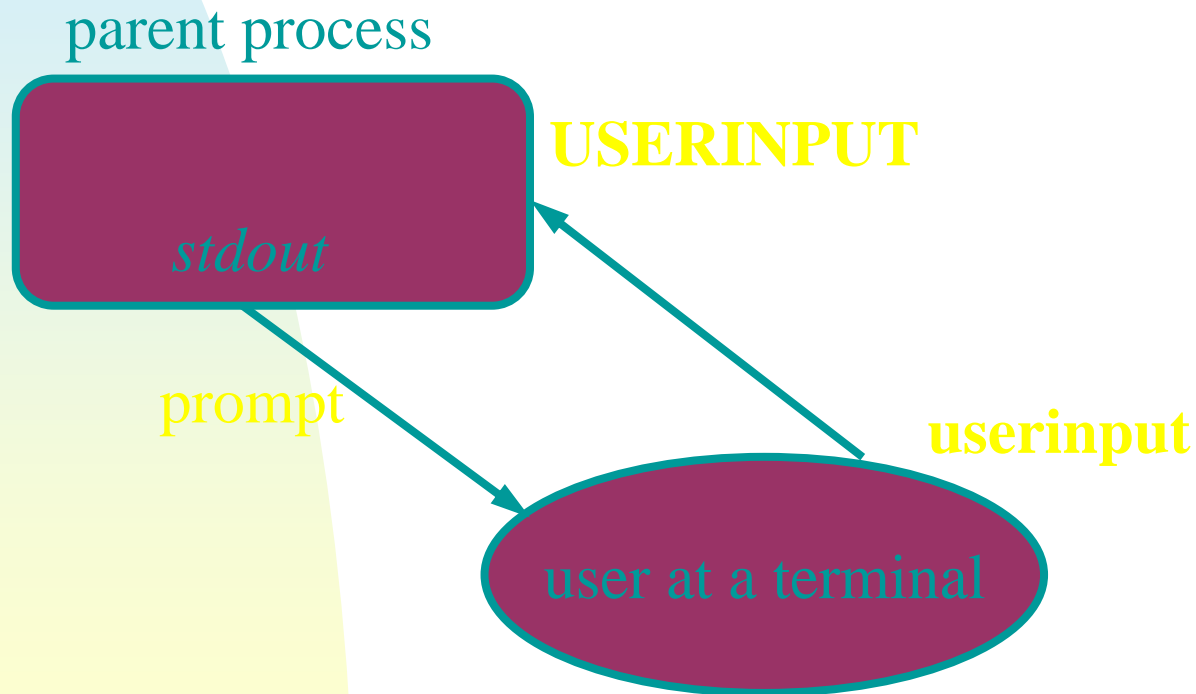


Implementing `popen` and `pclose`

- We have to keep the PIDs of child processes to synchronize.
 - `popen` and `pclose` could be called several times in a process.
 - How to keep the PIDs?
- We have to close the files opened by the child processes whose PIDs may not be available.

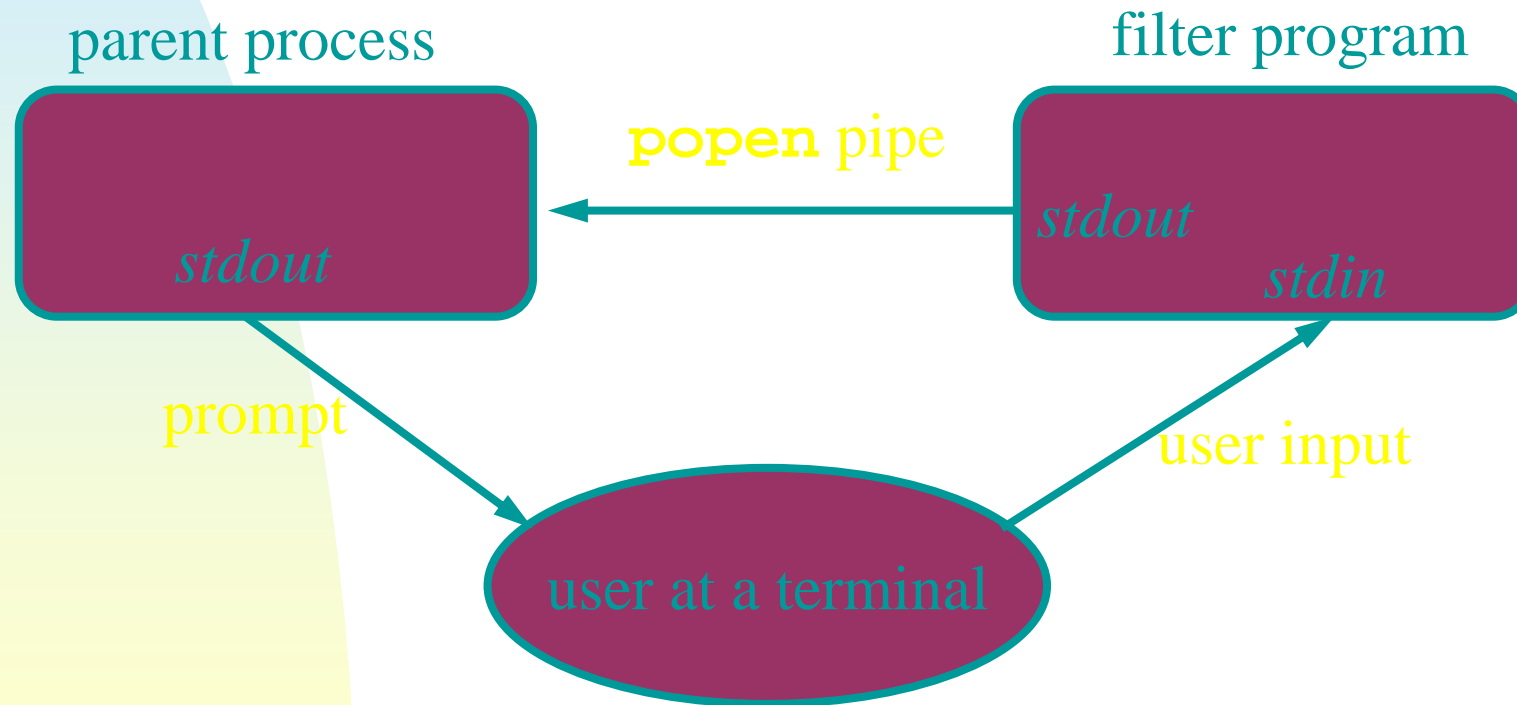
Another example

- An application to write a prompt to standard output and read a line from standard input.



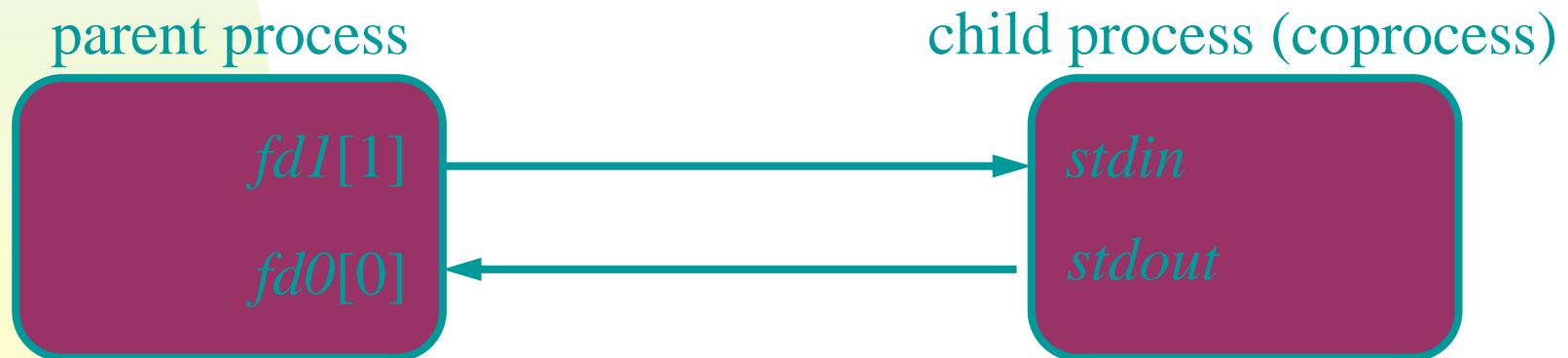
Another example

- An application to write a prompt to standard output and read a line from standard input.



Coprocesses

- Coprocesses: a process runs in the background from a shell and its standard input and output are connected to another program.
- Who are providing coprocesses?
 - KornShell provides coprocess
 - Bourne shell and C shell do not.

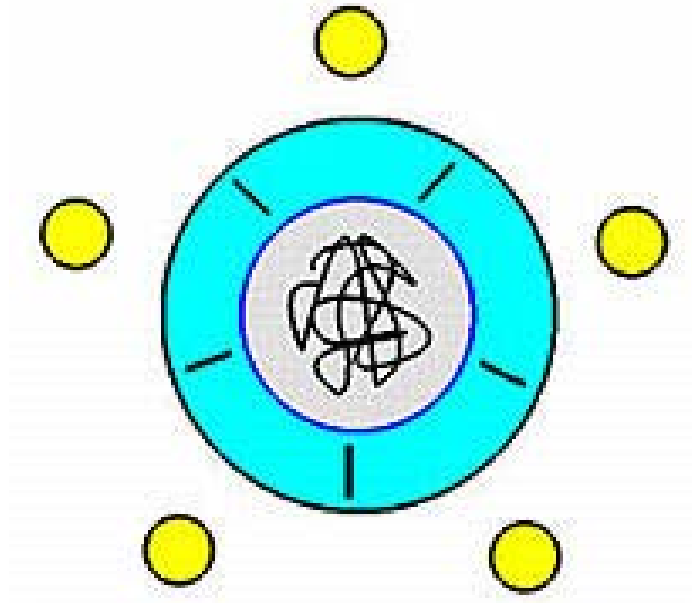


An example of Co-Process

- Program 14.8 is a simple co-process that reads two numbers from its stdin and writes the sum to its stdout.
- Program 14.9 invokes the co-process, after reading two numbers from its stdin.
- Program 14.10 re-writes the co-process using standard I/O.
- But, the two programs could cause a **deadlock**.

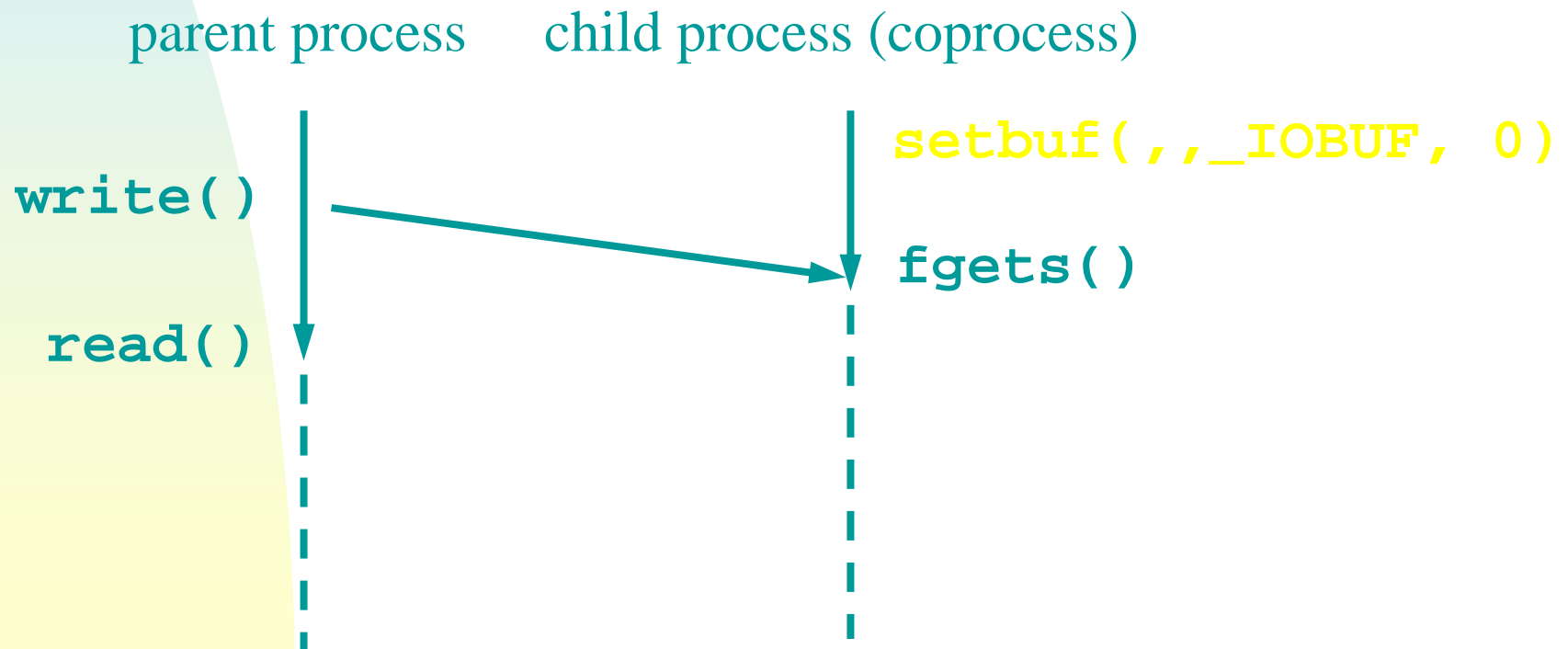
Dining Philosophers

- Dining Philosophers problem is invented by E. W. Dijkstra in 1965.
- Philosophers only think and eat, and may starve.



Deadlock in Program 14.10

- The parent process and co-process are blocked by read.



FIFOs

- FIFOs – named pipes for (un-)related processes to exchange data.
 - A file type – `st_mode` (`S_ISFIFO` macro)
 - Data in a FIFO removed when the last referencing process terminates.

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

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

```
int mkfifo(const char *pathname, mode_t  
mode);
```

- mode and usr/grp ownership = those of a file
- Op's: open, close, read, write, unlink, etc.

FIFOs

- `O_NONBLOCK`
 - `NO` → an open for read-only blocks until some other process opens the FIFO for writing (write-only as well).
 - `Yes` → an open for read-only always returns, while that for write-only returns with an error (`errno=ENXIO`) if there is no reader.
- Write for a no-reader FIFO → `SIGPIPE`
 - Closing of the last writer → `EOF`
- Atomic writes
 - `PIPE_BUF`

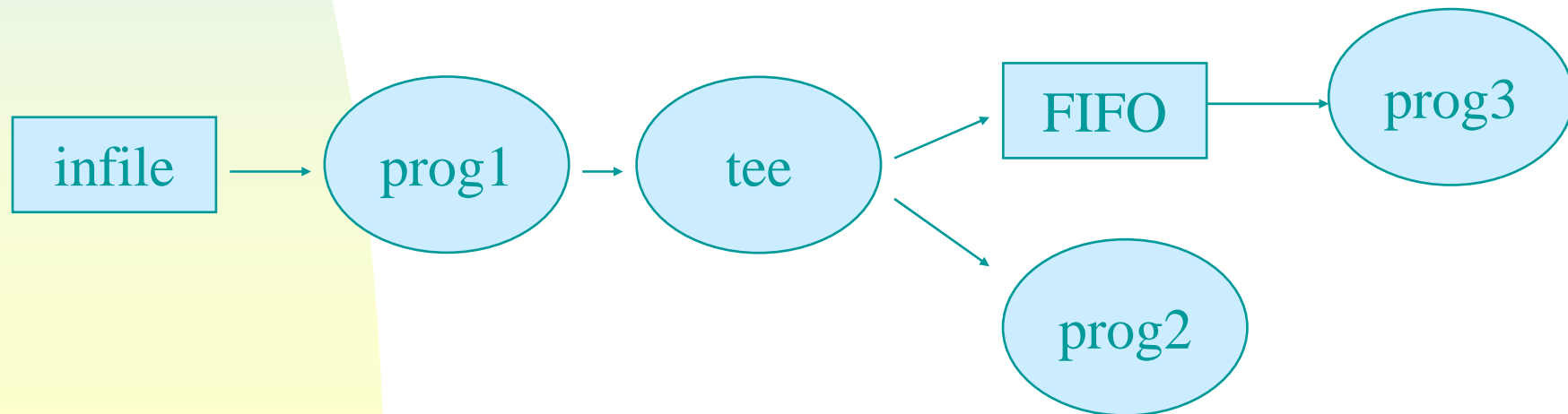
FIFOs

- Example – Stream Duplication

```
mkfifo fifo1
```

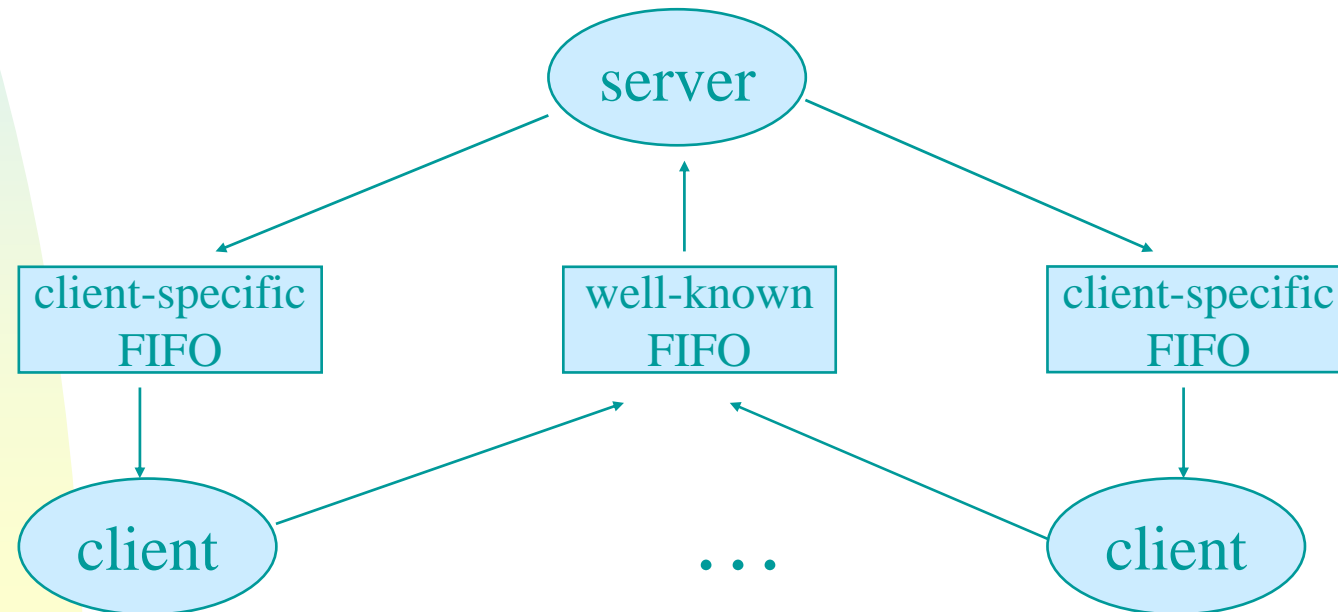
```
prog3 < fifo1 &
```

```
prog1 < infile | tee fifo1 | prog2
```



FIFOs

- Example – Client-Server Communication
 - One server & multiple clients
 - Issues: replies, SIGPIPE, removing of all clients



System V IPC

- System V IPC
 - Message queue, semaphore, shared memory segment
 - An ipc_perm structure for each IPC structure
 - An IPC identifier – slot usage sequence number [0...max#] (MSGMNI)
 - Key – msgget, semget, shmget
 - The Kernel converts it into an identifier.
 - IPC_PRIVATE
 - Key stored in a header.
 - `key_t ftok(const char *path, int id);`

```
struct ipc_perm {
    uid_t uid; /* owner euid */
    gid_t gid; /* owner egid */
    uid_t cuid; /* creator euid */
    gid_t cgid; /* creator egid */
    mode_t mode; /* access */
    ulong seg; /* slot usage seq # */
    key_t key;
}
```

System V IPC

- Creation of a new IPC structure
 - Key = IPC_PRIVATE
 - Key is not associated with any other ICP structure & IPC_CREAT bit is on in *flag*.
 - IPC_CREAT + IPC_EXCL
 - errno = EEXIST
- Referencing of a IPC structure
 - Key!= IPC_PRIVATE → Use the key used in the creation of the structure & IPC_CREAT bit is not set.
 - Key== IPC_PRIVATE → Use the identifier.

System V IPC

- Figure 14.14 – access rights
- Modifications to an ipc_perm structure
 - uid, gid, mode – msgctl, semctl, shmctl
- Problems
 - No reference count!
 - No name in the file system.
 - Needs of brand new functions
 - No multiplexing I/O
- Advantages – Figure 14.15 (Page 452)
 - Reliable, flow control, records, msg types or priorities, etc.

```
struct ipc_perm {  
    uid_t uid; /* owner euid */  
    gid_t gid; /* owner egid */  
    uid_t cuid; /* creator euid */  
    gid_t cgid; /* creator egid */  
    mode_t mode; /* access */  
    ulong seg; /* slot usage seq # */  
    key_t key; }
```

Message Queues

- Message Queue or Queue
 - A linked list of messages stored within the kernel and with a queue ID

```
struct msqid_ds {
    struct ipc_perm msg_perm;
    struct msg *msg_first; /* ptr to the first msg */
    struct msg *msg_last; /* ptr to the last msg */
    ulong msg_cbytes; /* current # of bytes */
    ulong msg_qnum; /* # of msgs */
    ulong msg_qbytes; /* max # of bytes */
    pid_t msg_lspid; /* pid of the last msgsnd() */
    pid_t msg_lrpid; /* pid of the last msgrcv() */
    time_t msg_stime; /* the last msgsnd() time */
    time_t msg_rtime; /* the last msgrcv() time */
    time_t msg_ctime; /* last change time */
}
```

- Figure 14.16 – Page 454
- msg_first and msg_last are useless!
- Msgs could be fetched based on their type field.

Message Queues

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

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

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

```
int msgget(key_t key, int flag);
```

- Key → identifier, flag → mode (Figure 14.14)
- msg_qnum=msg_lspid=msg_lrpid=msg_stime=msg_rtime=0
- msg_ctime=the current time
- msg_qbytes=MSGMNB
- Return a non-negative queue ID!

Message Queues

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

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

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

```
int msgctl(int msqid, int cmd, struct  
msqid_ds *buf);
```

- Cmd = IPC_STAT, IPC_SET, IPC_RMID
 - msg_perm.uid, msg_perm.gid, msg_perm.mode, msg_qbytes
 - euid = msg_perm.uid or msg_perm.cuid, or superuser
 - msg_qbytes set only by the superuser!

Message Queues

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

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

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

```
int msgsnd(int msqid, const void *ptr, size_t  
nbytes, int flag);
```

- ptr → struct mymsg { long mtype; char mtext[XXX];}
- IPC_NOWAIT bit in flag – non-blocking flag
 - errno = EAGAIN if it is specified, and the queue is full.
 - Otherwise, it is blocked until (a) there is room left (b) the queue is removed (EIDRM) (c) a signal is caught (EINTR)

Message Queues

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

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

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

```
int msgrcv(int msqid, void *ptr, size_t nbytes,  
long type, int flag);
```

- *nbytes* > msg length
 - MSG_NOERROR bit in flag → truncate the msg!
 - Otherwise → errno = E2BIG, and the msg is left in the queue.
- *type* == 0 → the first msg
 - *type* > 0 → the first msg with the type
 - *type* < 0 → the first msg with the smallest type value ≤ *type*
- IPC_NOWAIT bit in the *flag*