DCS216 Operating Systems

Lecture 08
Inter-process Communication (2)

Mar 20th, 2024

Instructor: Xiaoxi Zhang

Sun Yat-sen University

Content

Content

Content

- Overview
- Shared-Memory Systems
- Message-Passing Systems
  - Message-Passing Design Principles
  - System V Message Queues
  - POSIX Message Queues
- Pipes
  - Unnamed Pipes
  - Named Pipes
- Communication in Client-Server Systems
  - Sockets
  - Remote Procedure Calls (RPCs)

Taxonomy of UNIX IPC

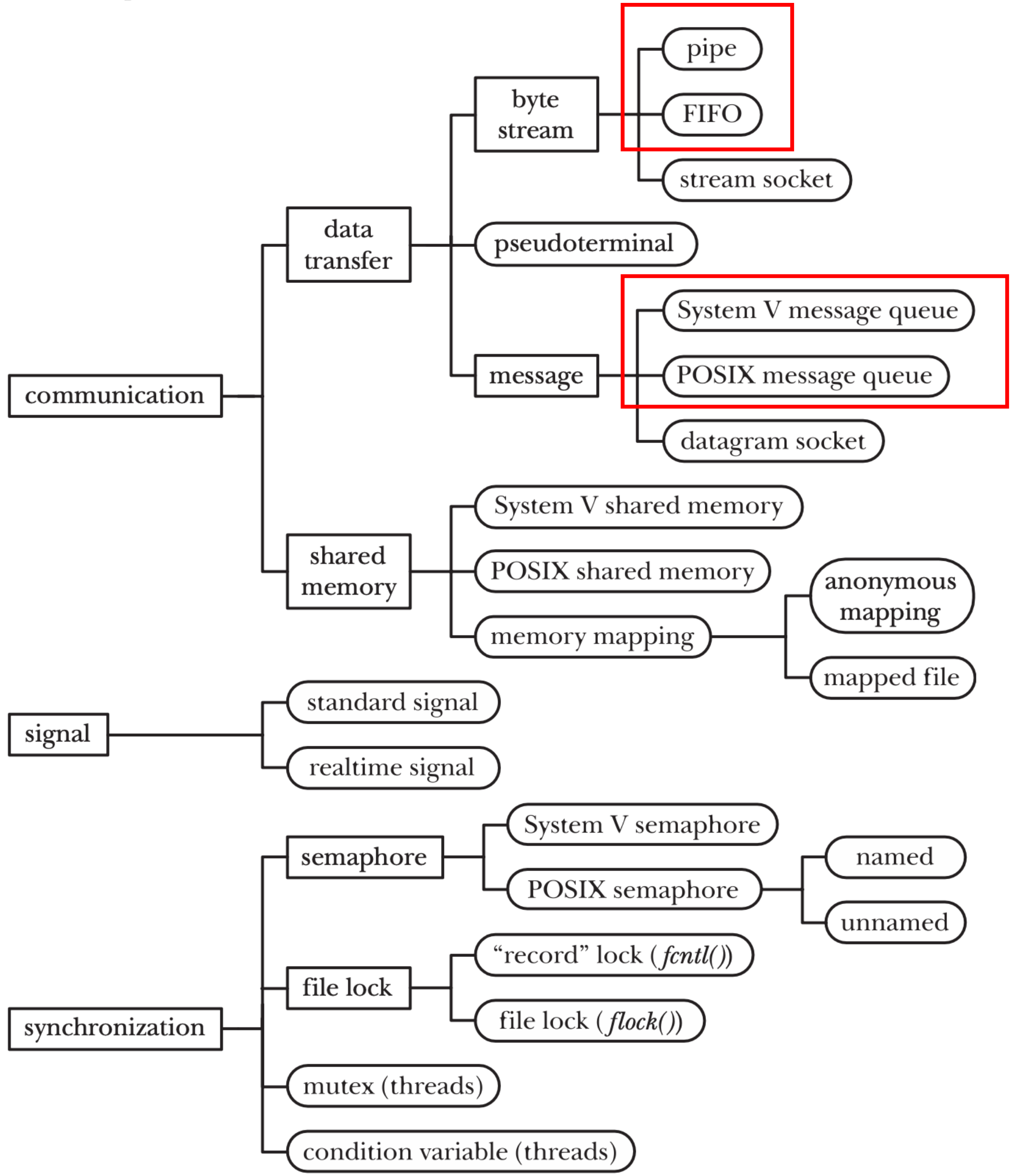


Figure 43-1: A taxonomy of UNIX IPC facilities

Message-Passing Systems

- Message passing provides a mechanism to allow processes to communicate and synchronize their actions without sharing the same address space, i.e. shared variables. (无需依靠共享变量来同步)

Message-Passing Systems

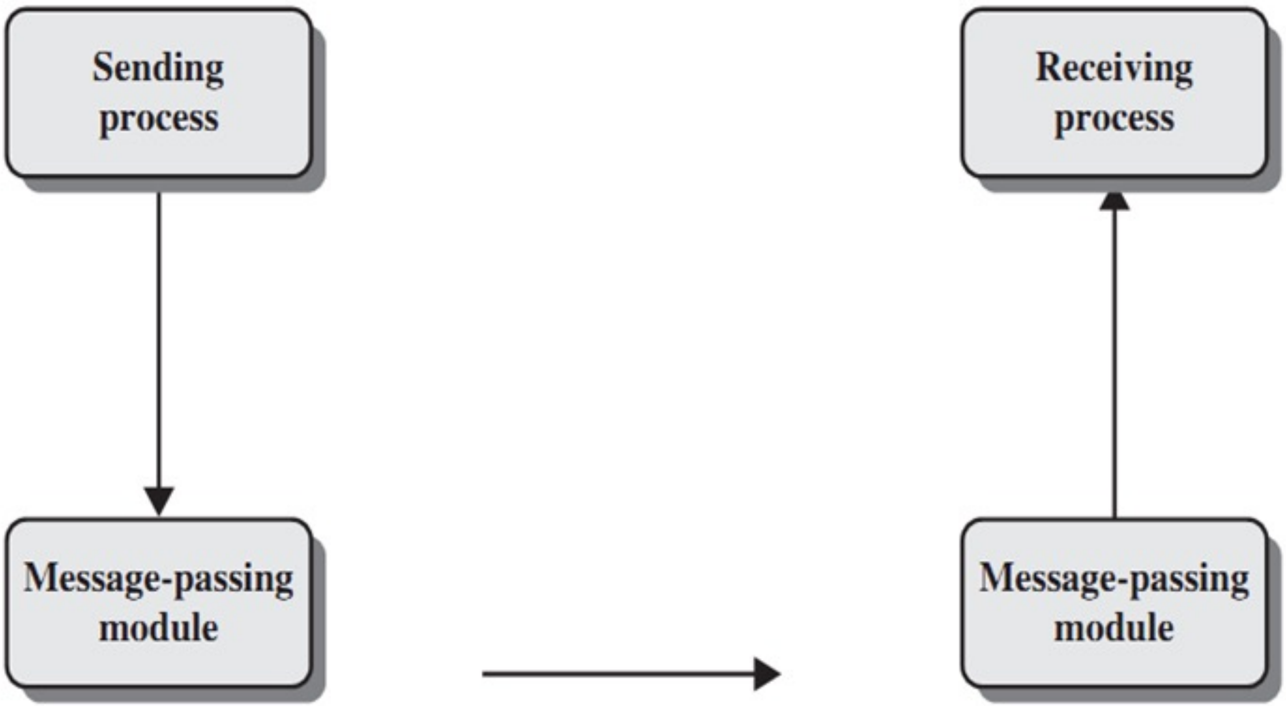
- Message passing provides a mechanism to allow processes to communicate and synchronize their actions without sharing the same address space, i.e. shared variables. (无需依靠共享变量来同步)
  - Communication takes place by means of messages exchanged between cooperating processes
  - Useful for exchanging small amounts of data
  - Typically implemented using system calls, and thus require more time-consuming task of kernel intervention
  - **Easier** to implement in a distributed system **than** shared memory
  - Message queues reside in kernel space with **race condition or resource confliction handled by the kernel.**

Message-Passing Systems

- Message size can be either **fixed** or **variable**
  - **Variable-sized** messages require a more **complex** system-level implementation
    - but the programming task is much **simpler**
  - **Fixed-sized** messages is **easier** to implement on the system-level
    - but it makes the task of programming more **difficult**
  - Common kind of tradeoff seen throughout operating system design
    - Unlike **algorithm design**, which typically focuses on finding an efficient way to solve a specific problem (mathematical)
    - **System design** involves creating an architecture for software systems, which often require balancing various goals and constraints (e.g., scalability, reliability, maintainability, cost and performance)

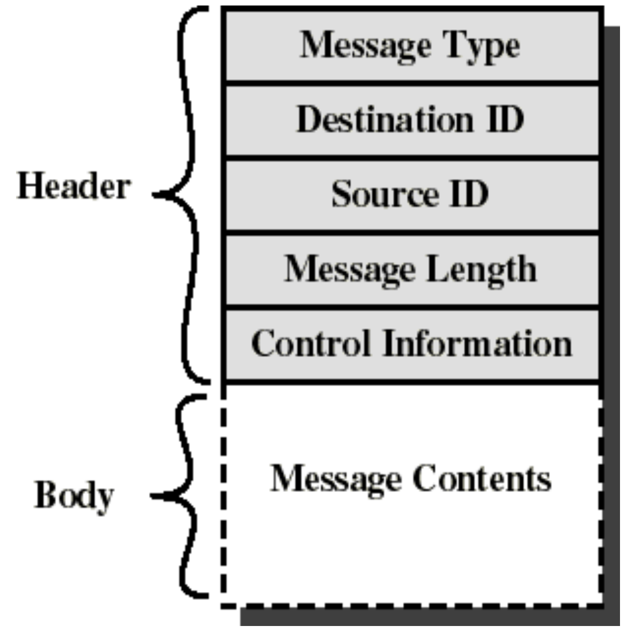
Message-Passing Systems

- Message passing provides (at least) two basic operations:
  - `send(message)` or `send(dest, message)`
  - `receive(message)` or `receive(src, message)`



Message Format

- Header
  - Message Type
  - Message Length
  - Destination ID & Source ID
  - Control Information
    - What to do if run out of buffer space?
    - Sequence numbers
    - Priority
- Body
  - Message Content
- Queueing discipline: usually FIFO, but can also include priorities
- Example: a message format containing only message type and message content:




```c
struct message {
    long mtype;       // message type
    char mtext[80];   // message text
};
```

Message-Passing Systems

- If processes P and Q wish to communicate, they need to:
  - Establish a **communication link** between them
  - Exchange messages via **send/receive**
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communication processes?
  - What is the capacity of a link?
  - Is the message size fixed or variable?
  - Is a link unidirectional or bidirectional?

- **Message-Passing Systems**
  - Logical Implementation of Communication Link
    - **Direct** or **Indirect** Communication
    - **Synchronous** or **Asynchronous**
    - **Automatic** or **Explicit** buffering

Message-Passing Systems

Direct Communication

- Processes must name each other explicitly:
  - **send**(P, message) – send a message to process P
  - **receive**(Q, message) – receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bidirectional

Message-Passing Systems

- **Indirect Communication**
  - Messages are directed and received from mailboxes (ports)
    - Each mailbox has a unique ID
    - Processes can communicate only if they share a mailbox (port)
  - Properties of communication link
    - Link established only if processes share a common mailbox
    - A link may be associated with many processes
    - Each pair of processes may share several communication links
    - Link may be unidirectional or bidirectional

Message-Passing Systems

- **Indirect Communication**
  - Messages are directed and received from mailboxes (ports)
    - Each mailbox has a unique ID
    - Processes can communicate only if they share a mailbox (port)
  - Properties of communication link
    - Link established only if processes share a common mailbox
    - A link may be associated with many processes
    - Each pair of processes may share several communication links
    - Link may be unidirectional or bidirectional
  - Operations
    - Create a new mailbox (port)
    - Send and receive messages through mailbox
    - Delete a mailbox
  - Primitives:
    - **send**(A, message) – send a mailbox to mailbox A
    - **receive**(A, message) – receive a message from mailbox A

- **Indirect Communication**
  - Primitives:
    - **send**(A, message) – send a mailbox to mailbox A
    - **receive**(A, message) – receive a message from mailbox A
  - Mailbox sharing
    - P1, P2, and P3 share mailbox A
    - P1 sends;
    - P2 and P3 receive
    - Who gets the message?
  - Solutions
    - Allow a link to be associated with at most two processes
    - Allow only one process at a time to execute a receive operation
    - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking
  - **Blocking** (阻塞)is considered synchronous (同步)
    - Blocking send – the sender is blocked until the message is received
    - Blocking receive – the receiver is blocked until message is available
  - **Non-blocking** (非阻塞)is considered asynchronous (异步)
    - Non-blocking send – the sender sends the message and continue
    - Non-blocking receive – the receiver receives:
      - A valid message, or
      - Null message
  - Different combinations possible
    - If both send and receive are blocking, we have a rendezvous(汇聚点).

Message-Passing Systems

- **Buffering**
  - Queue of messages attached to the link.
  - Implemented in one of three ways:
    - **Zero Capacity** – no messages are queued on a link
      - Sender must **wait** for receiver (rendezvous)
    - **Bounded Capacity** – finite length of N messages
      - Sender must **wait** if link is full
    - **Unbounded Capacity** – infinite length
      - Sender **never waits**

IPC Models

Shared Memory

- Direct Sharing (**System V Standard**) System Calls:
  - `shmget()`, `shmat()`, `shmdt()`, `shmctl()`
- Indirect Sharing (**POSIX Standard**) Library Calls:
  - `shm_open()`, `shm_unlink()`, `ftruncate()`, `mmap()`

Message Passing

- **Pipes**
  - Unnamed Pipe: `pipe()`
  - Named Pipe: `mkfifo()`
- **Message Queues**
  - System V: `msgget()`, `msgsnd()`, `msgrcv()`, `msgctl()`
  - POSIX: `mq_open()`, `mq_close()`, `mq_send()`, `mq_receive()`
- **Sockets**: `socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `send()`, `recv()`
- **Signals**: `signal()`, `sigaction()`

Inter-process Communication

IPC Models

- Shared Memory
  - Direct Sharing (**System V Standard**) System Calls:
    - `shmget()`, `shmat()`, `shmdt()`, `shmctl()`
  - Indirect Sharing (**POSIX Standard**) Library Calls:
    - `shm_open()`, `shm_unlink()`, `ftruncate()`, `mmap()`
- Message Passing
  - **Pipes**
    - Unnamed Pipe: `pipe()`
    - Named Pipe: `mkfifo()`
  - **Message Queues**
    - System V: `msgget()`, `msgsnd()`, `msgrcv()`, `msgctl()`
    - POSIX: `mq_open()`, `mq_close()`, `mq_send()`, `mq_receive()`
  - **Sockets**: `socket()`, `bind()`, `listen()`, `accept()`, `connect()`, `send()`, `recv()`
  - **Signals**: `signal()`, `sigaction()`

System V Message Queues API

- **Get** a System V message queue identifier
  - `int msgget(key_t key, int msgflg);`
- **Send** data into a message queue
  - ```
    int msgsnd(int msqid, const void *msgp, size_t msgsz,
               int msgflg);
    ```
- **Receive** data from a message queue
  - ```
    ssize_t msgrcv(int msqid, void *msgp, size_t msgsz,
                   long msgtyp, int msgflg);
    ```
- **Perform** various operations on a message queue
  - `int msgctl(int msqid, int cmd, struct msqid_ds *buf);`
    - E.g., `msgctl(msgid, IPC_RMID, NULL)` **removes** the message queue `msgid`
- **CLI Utility**: Show Message Queue Info
  - `ipcs -q`

System V Message Queues Example

```c
/* sender.c */
#include "message.h"
int main() {
  int msgid;
  struct message msg;
  // Create a message queue
  msgid = msgget(KEY, 0666 | IPC_CREAT);
  // Prepare the message
  msg.mtype = 1; // Set the message type
  strcpy(msg.mtext, "Hello Message...");
  // Send the message
  msgsnd(msgid, &msg, sizeof(msg.mtext), 0);
  printf("Sent message: %s\n", msg.mtext);
  return 0;
}
```

```c
/* receiver.c */
#include "message.h"
int main() {
  int msgid;
  struct message msg;
  // Access the message queue
  msgid = msgget(KEY, 0666);
  // Receive the message
  msgrcv(msgid, &msg, sizeof(msg.mtext), 0,
         0);
  printf("Recv message: %s\n", msg.mtext);
  // Optionally, remove the message queue
  msgctl(msgid, IPC_RMID, NULL);
  return 0;
}
```

```c
/* message.h */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define KEY 0x42
struct message {
    long mtype;      // message type
    char mtext[80];  // message text
};
```

System V Message Queues Example

```c
/* sender.c */
#include "message.h"
int main() {
  int msgid;
  struct message msg;
  // Create a message queue
  msgid = msgget(KEY, 0666 | IPC_CREAT);
  // Prepare the message
  msg.mtype = 1; // Set the message type
  strcpy(msg.mtext, "Hello Message...");
  // Send the message
  msgsnd(msgid, &msg, sizeof(msg.mtext), 0);
  printf("Sent message: %s\n", msg.mtext);
  return 0;
}
```

```c
/* receiver.c */
#include "message.h"
int main() {
  int msgid;
  struct message msg;
  // Access the message queue
  msgid = msgget(KEY, 0666);
  // Receive the message
  msgrcv(msgid, &msg, sizeof(msg.mtext), 0,
         0);
  printf("Recv message: %s\n", msg.mtext);
  // Optionally, remove the message queue
  msgctl(msgid, IPC_RMID, NULL);
  return 0;
}
```

```c
/* message.h */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define KEY 0x42
struct message {
    long mtype;      // message type
    char mtext[80];  // message text
};
```

```
$ ./sender
Sent message: Hello Message...
$ ipcs -q
------ Message Queues --------
key         msqid    owner    perms    used-bytes    messages
0x00000042  1        zxx      666      80            1
$ ./receiver
Recv message: Hello Message...
$ ipcs -q
------ Message Queues --------
key         msqid    owner    perms    used-bytes    messages
```

System V Message Queues Example

```c
/* sender.c */
#include "message.h"
int main() {
  int msgid;
  struct message msg;
  // Create a message queue
  msgid = msgget(KEY, 0666 | IPC_CREAT);
  // Prepare the message
  msg.mtype = 1; // Set the message type
  strcpy(msg.mtext, "Hello Message...");
  // Send the message
  msgsnd(msgid, &msg, sizeof(msg.mtext), 0);
  printf("Sent message: %s\n", msg.mtext);
  return 0;
}
```

```c
/* receiver.c */
#include "message.h"
int main() {
  int msgid;
  struct message msg;
  // Access the message queue
  msgid = msgget(KEY, 0666);
  // Receive the message
  msgrcv(msgid, &msg, sizeof(msg.mtext), 0,
         0);
  printf("Recv message: %s\n", msg.mtext);
  // Optionally, remove the message queue
  msgctl(msgid, IPC_RMID, NULL);
  return 0;
}
```

```c
/* message.h */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define KEY 0x42
struct message {
    long mtype;       // message type
    char mtext[80];   // message text
};
```

```
$ ./sender
Sent message: Hello Message...
$ ipcs -q
------ Message Queues --------
key         msqid    owner    perms    used-bytes    messages
0x00000042  1        zxx      666      80            1
$ ./receiver
Recv message: Hello Message...
$ ipcs -q
------ Message Queues --------
key         msqid    owner    perms    used-bytes    messages
```

System V Message Queues Example

```c
/* sender.c */
#include "message.h"
int main() {
  int msgid;
  struct message msg;
  // Create a message queue
  msgid = msgget(KEY, 0666 | IPC_CREAT);
  // Prepare the message
  msg.mtype = 1; // Set the message type
  strcpy(msg.mtext, "Hello Message...");
  // Send the message
  msgsnd(msgid, &msg, sizeof(msg.mtext), 0);
  printf("Sent message: %s\n", msg.mtext);
  return 0;
}
```

```c
/* receiver.c */
#include "message.h"
int main() {
  int msgid;
  struct message msg;
  // Access the message queue
  msgid = msgget(KEY, 0666);
  // Receive the message
  msgrcv(msgid, &msg, sizeof(msg.mtext), 0,
         0);
  printf("Recv message: %s\n", msg.mtext);
  // Optionally, remove the message queue
  msgctl(msgid, IPC_RMID, NULL);
  return 0;
}
```

```c
/* message.h */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define KEY 0x42
struct message {
    long mtype;       // message type
    char mtext[80];   // message text
};
```

```
$ ./sender
Sent message: Hello Message...
$ ipcs -q
------ Message Queues --------
key          msqid    owner    perms    used-bytes    messages
0x00000042   1        zxx      666      80            1
$ ./receiver
Recv message: Hello Message...
$ ipcs -q
------ Message Queues --------
key          msqid    owner    perms    used-bytes    messages
```

Message-Passing Systems

System V Message Queues Example

```c
/* sender.c */
#include "message.h"
int main() {
  int msgid;
  struct message msg;
  // Create a message queue
  msgid = msgget(KEY, 0666 | IPC_CREAT);
  // Prepare the message
  msg.mtype = 1; // Set the message type
  strcpy(msg.mtext, "Hello Message...");
  // Send the message
  msgsnd(msgid, &msg, sizeof(msg.mtext), 0);
  printf("Sent message: %s\n", msg.mtext);
  return 0;
}
```

```c
/* receiver.c */
#include "message.h"
int main() {
  int msgid;
  struct message msg;
  // Access the message queue
  msgid = msgget(KEY, 0666);
  // Receive the message
  msgrcv(msgid, &msg, sizeof(msg.mtext), 0,
         0);
  printf("Recv message: %s\n", msg.mtext);
  // Optionally, remove the message queue
  msgctl(msgid, IPC_RMID, NULL);
  return 0;
}
```

```c
/* message.h */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define KEY 0x42
struct message {
    long mtype;       // message type
    char mtext[80];   // message text
};
```

```
$ ./sender
Sent message: Hello Message...
$ ipcs -q
------ Message Queues --------
key          msqid    owner    perms    used-bytes    messages
0x00000042   1        zxx      666      80            1
$ ./receiver
Recv message: Hello Message...
$ ipcs -q
------ Message Queues --------
key          msqid    owner    perms    used-bytes    messages
```

System V Message Queues Example

```c
/* sender.c */
#include "message.h"
int main() {
  int msgid;
  struct message msg;
  // Create a message queue
  msgid = msgget(KEY, 0666 | IPC_CREAT);
  // Prepare the message
  msg.mtype = 1; // Set the message type
  strcpy(msg.mtext, "Hello Message...");
  // Send the message
  msgsnd(msgid, &msg, sizeof(msg.mtext), 0);
  printf("Sent message: %s\n", msg.mtext);
  return 0;
}
```

```c
/* receiver.c */
#include "message.h"
int main() {
  int msgid;
  struct message msg;
  // Access the message queue
  msgid = msgget(KEY, 0666);
  // Receive the message
  msgrcv(msgid, &msg, sizeof(msg.mtext), 0,
         0);
  printf("Recv message: %s\n", msg.mtext);
  // Optionally, remove the message queue
  msgctl(msgid, IPC_RMID, NULL);
  return 0;
}
```

```c
/* message.h */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define KEY 0x42
struct message {
    long mtype;      // message type
    char mtext[80];  // message text
};
```

Sender and receiver don't need to run concurrently. Msg stays in the kernel buffer...

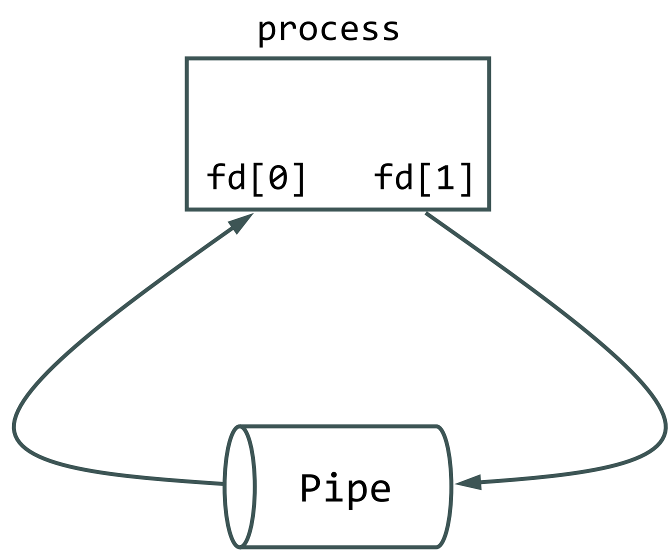
```
$ ./sender
Sent message: Hello Message...
$ ipcs -q
------ Message Queues --------
key          msqid    owner    perms    used-bytes    messages
0x00000042   1        zxx      666      80            1
$ ./receiver
Recv message: Hello Message...
$ ipcs -q
------ Message Queues --------
key          msqid    owner    perms    used-bytes    messages
```

POSIX Message Queues API

- Similar to File I/O API:
  - `open()`, `close()`, `read()`, `write()`
- Open a message queue descriptor
  - `mqd_t mq_open(const char *name, int oflag);`
- Close a message queue descriptor
  - `int mq_close(mqd_t mqdes);`
- Send a message to a message queue
  - ```
    int mq_send(mqd_t mqdes, const char *msg_ptr,
                size_t msg_len, unsigned int msg_prio);
    ```
- Receive a message from a message queue
  - ```
    ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t
                       msg_len, unsigned int *msg_prio);
    ```
- See `man mq_overview`

Pipes

- Pipes are the **oldest** form of IPC and provided by **all** UNIX systems.
  - Historically, pipes have been half duplex (i.e., data flows in only one direction). Some systems provide full-duplex pipes, but not recommended.
  - Pipes can be used only between parent and child processes.
  - Despite these limitations, half-duplex pipes are still the most commonly used form of IPC
- A pipe is one form of Message Passing IPC.



Pipes

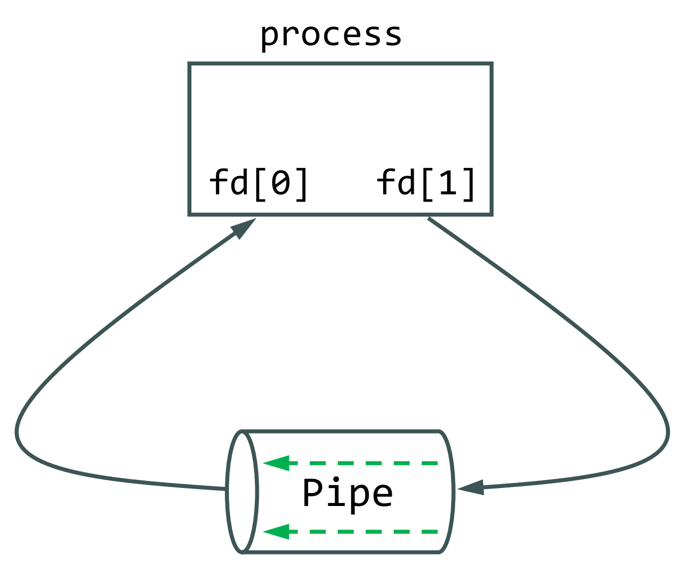
- A pipe is created by calling the `pipe()` function.
  - `int pipe(int fd[2]);`
- Two file descriptors are returned through the `fd` argument:
  - `fd[0]` is open for reading
  - `fd[1]` is open for writing

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#define MAXLINE 80

int main() {
    int len;
    int fd[2];
    char buf[MAXLINE];

    pipe(fd);
    write(fd[1], "Hello, world!", 14);
    len = read(fd[0], buf, MAXLINE);
    printf("buf(%d): %s\n", len, buf);
    return 0;
}
```

```
$ gcc -o pipe1 pipe1.c
$ ./pipe1
buf(14): Hello, world!
```



Pipes

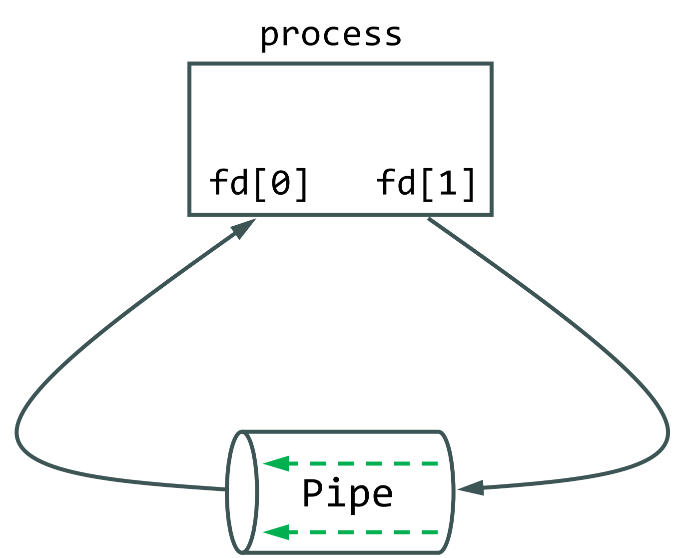
- Two file descriptors are returned through the `fd` argument:
  - `fd[0]` is open for reading
  - `fd[1]` is open for writing
- A pipe **in a single process** is next to useless.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#define MAXLINE 80

int main() {
    int len;
    int fd[2];
    char buf[MAXLINE];

    pipe(fd);
    write(fd[1], "Hello, world!", 14);
    len = read(fd[0], buf, MAXLINE);
    printf("buf(%d): %s\n", len, buf);
    return 0;
}
```

```
$ gcc -o pipe1 pipe1.c
$ ./pipe1
buf(14): Hello, world!
```

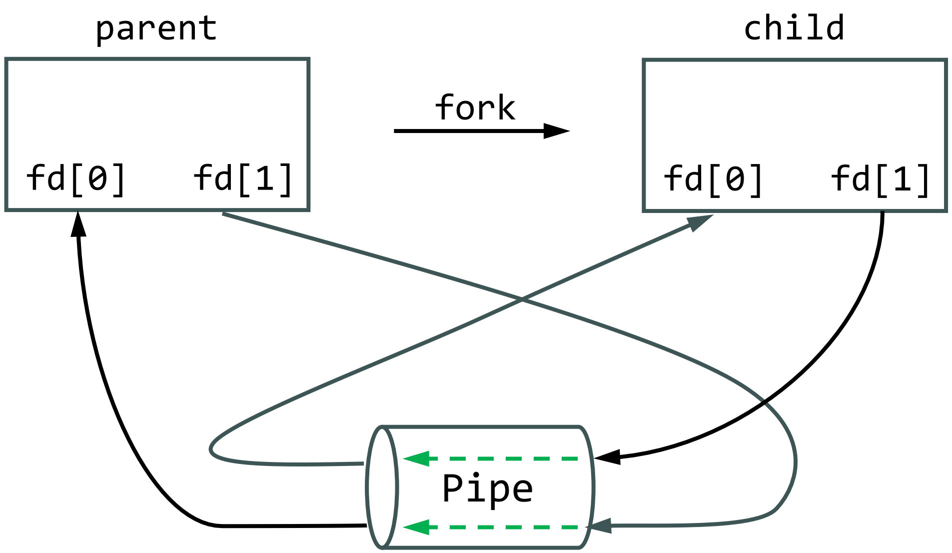


Pipes

- Two file descriptors are returned through the `fd` argument:
  - `fd[0]` is open for reading
  - `fd[1]` is open for writing
- A pipe **in a single process** is next to useless.
- followed by `fork()` to create IPC channel between parent & child

```c
int main() {
    int len;
    int fd[2];
    char buf[MAXLINE];

    pipe(fd);
    pid_t rc = fork();
    if (rc == 0) {      /* Child  */
        close(fd[0]);
        write(fd[1], "Hello, world!", 14);
    } else {            /* Parent */
        close(fd[1]);
        len = read(fd[0], buf, MAXLINE);
        printf("buf(%d): %s\n", len, buf);
    }
    return 0;
}
```



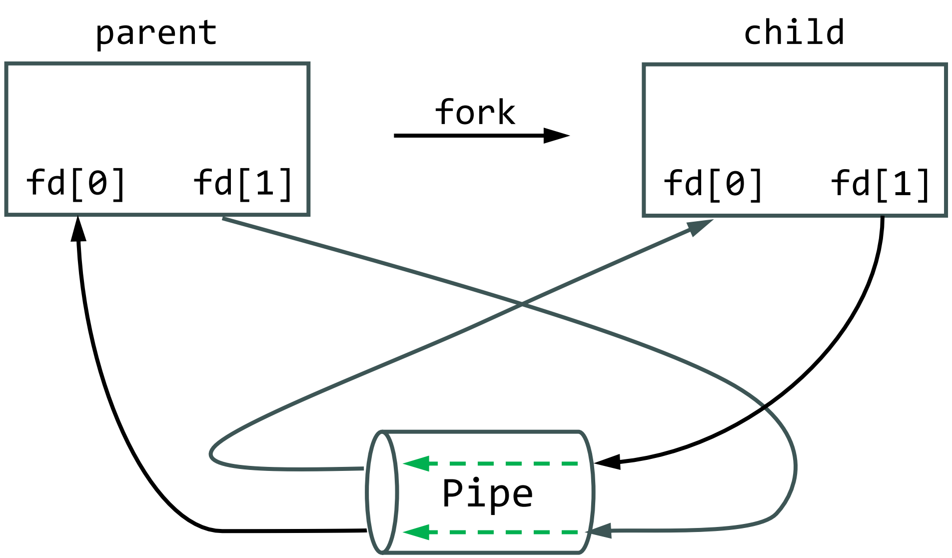
Pipes

- Two file descriptors are returned through the `fd` argument:
  - `fd[0]` is open for reading
  - `fd[1]` is open for writing
- A pipe **in a single process** is next to useless.
- followed by `fork()` to create IPC channel between parent & child

```c
int main() {
    int len;
    int fd[2];
    char buf[MAXLINE];

    pipe(fd);
    pid_t rc = fork();
    if (rc == 0) {        /* Child  */
        close(fd[0]);
        write(fd[1], "Hello, world!", 14);
    } else {              /* Parent */
        close(fd[1]);
        len = read(fd[0], buf, MAXLINE);
        printf("buf(%d): %s\n", len, buf);
    }
    return 0;
}
```

```
$ gcc -o pipe2 pipe2.c
$ ./pipe2
buf(14): Hello, world!
```



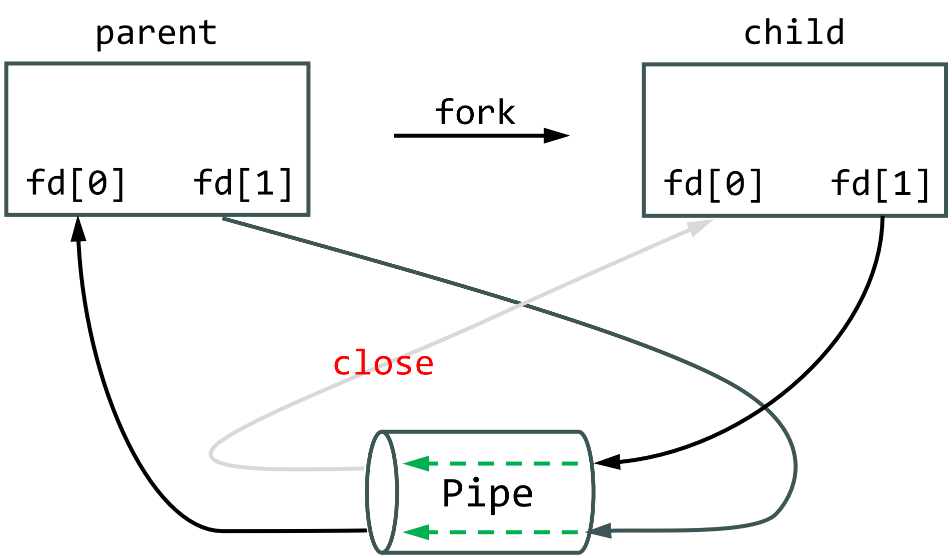
Pipes

- Two file descriptors are returned through the `fd` argument:
  - `fd[0]` is open for reading
  - `fd[1]` is open for writing
- A pipe **in a single process** is next to useless.
- followed by `fork()` to create IPC channel between parent & child

```c
int main() {
    int len;
    int fd[2];
    char buf[MAXLINE];

    pipe(fd);
    pid_t rc = fork();
    if (rc == 0) {          /* Child  */
        close(fd[0]);
        write(fd[1], "Hello, world!", 14);
    } else {                /* Parent */
        close(fd[1]);
        len = read(fd[0], buf, MAXLINE);
        printf("buf(%d): %s\n", len, buf);
    }
    return 0;
}
```

```
$ gcc -o pipe2 pipe2.c
$ ./pipe2
buf(14): Hello, world!
```



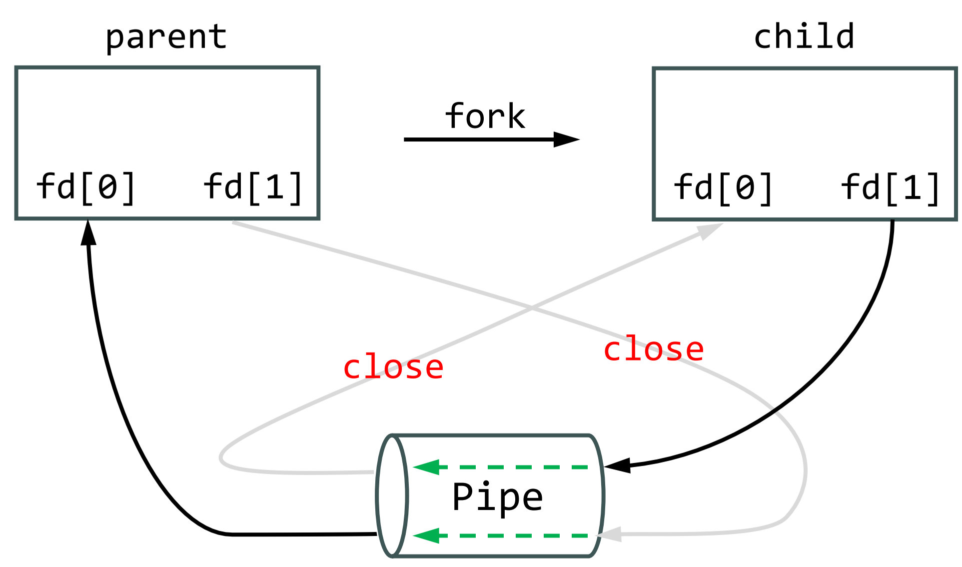
Pipes

- Two file descriptors are returned through the `fd` argument:
  - `fd[0]` is open for reading
  - `fd[1]` is open for writing
- A pipe **in a single process** is next to useless.
- followed by `fork()` to create IPC channel between parent & child

```c
int main() {
    int len;
    int fd[2];
    char buf[MAXLINE];

    pipe(fd);
    pid_t rc = fork();
    if (rc == 0) {          /* Child  */
        close(fd[0]);
        write(fd[1], "Hello, world!", 14);
    } else {                /* Parent */
        close(fd[1]);
        len = read(fd[0], buf, MAXLINE);
        printf("buf(%d): %s\n", len, buf);
    }
    return 0;
}
```

```
$ gcc -o pipe2 pipe2.c
$ ./pipe2
buf(14): Hello, world!
```



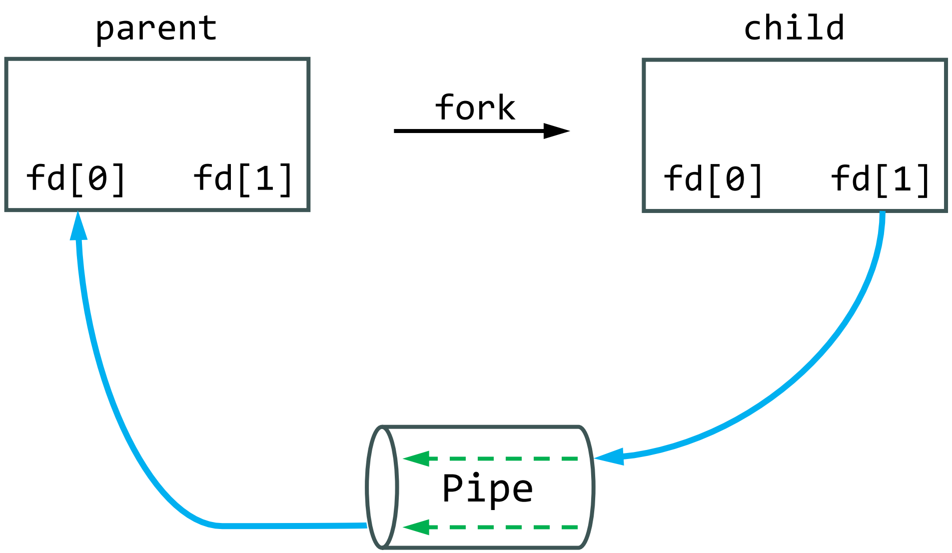
Pipes

- Two file descriptors are returned through the `fd` argument:
  - `fd[0]` is open for reading
  - `fd[1]` is open for writing
- A pipe **in a single process** is next to useless.
- followed by `fork()` to create IPC channel between parent & child

```
int main() {
    int len;
    int fd[2];
    char buf[MAXLINE];

    pipe(fd);
    pid_t rc = fork();
    if (rc == 0) {        /* Child  */
        close(fd[0]);
        write(fd[1], "Hello, world!", 14);
    } else {              /* Parent */
        close(fd[1]);
        len = read(fd[0], buf, MAXLINE);
        printf("buf(%d): %s\n", len, buf);
    }
    return 0;
}
```

```
$ gcc -o pipe2 pipe2.c
$ ./pipe2
buf(14): Hello, world!
```



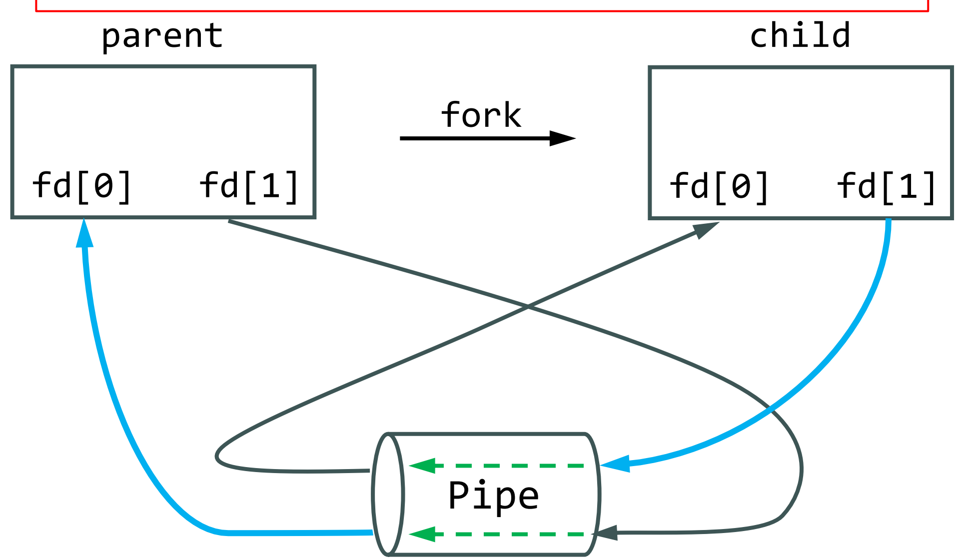
Pipes

- Two file descriptors are returned through the `fd` argument:
  - `fd[0]` is open for reading
  - `fd[1]` is open for writing
- A pipe **in a single process** is next to useless.
- followed by `fork()` to create IPC channel between parent & child

```
int main() {
    int len;
    int fd[2];
    char buf[MAXLINE];

    pipe(fd);
    pid_t rc = fork();
    if (rc == 0) {          /* Child  */
        // close(fd[0]);
        write(fd[1], "Hello, world!", 14);
    } else {                /* Parent */
        // close(fd[1]);
        len = read(fd[0], buf, MAXLINE);
        printf("buf(%d): %s\n", len, buf);
    }
    return 0;
}
```

It might seem OK to leave `fd[0]` in **child** and `fd[1]` in **parent** alone without closing them. But it's **recommended** to **close unused file descriptors** to ensure proper pipe termination.



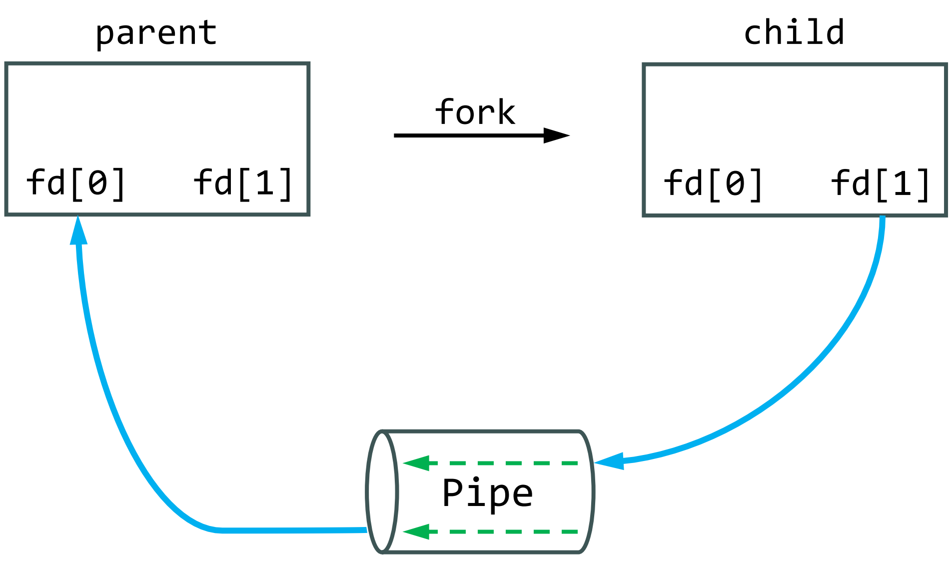
Pipes

- A pipe is created by calling the `pipe()` function.
- followed by `fork()` to create IPC channel between parent & child
- also called **Unnamed Pipes/Ordinary Pipes:** cannot be accessed from outside the process that created it.

```c
int main() {
    int len;
    int fd[2];
    char buf[MAXLINE];

    pipe(fd);
    pid_t rc = fork();
    if (rc == 0) {          /* Child  */
        close(fd[0]);
        write(fd[1], "Hello, world!", 14);
    } else {                /* Parent */
        close(fd[1]);
        len = read(fd[0], buf, MAXLINE);
        printf("buf(%d): %s\n", len, buf);
    }
    return 0;
}
```

```
$ gcc -o pipe2 pipe2.c
$ ./pipe2
buf(14): Hello, world!
```



Pipes

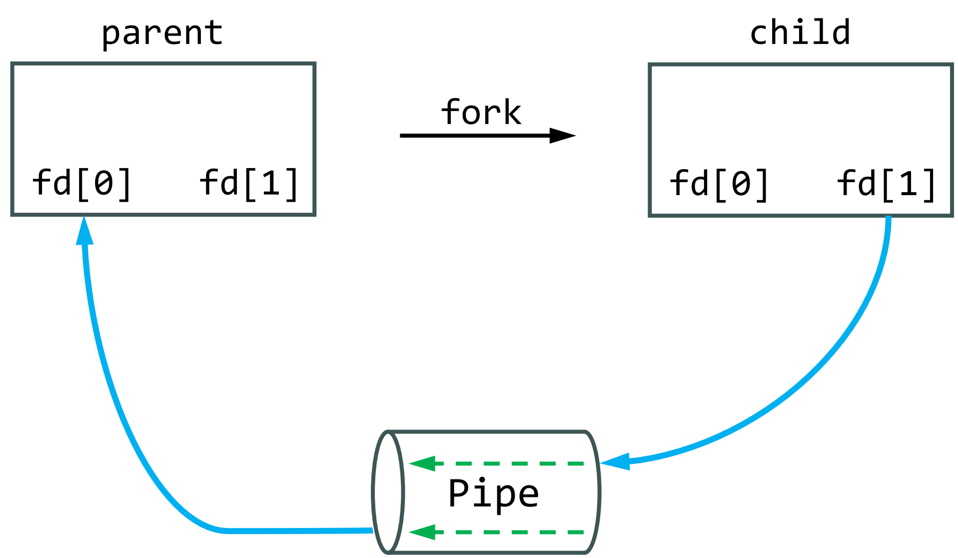
- A pipe is created by calling the `pipe()` function.
- followed by `fork()` to create IPC channel between parent & child
- also called **Unnamed Pipes/Ordinary Pipes:** cannot be accessed from outside the process that created it.

```c
int main() {
    int len;
    int fd[2];
    char buf[MAXLINE];

    pipe(fd);
    pid_t rc = fork();
    if (rc == 0) {         /* Child  */
        close(fd[0]);
        write(fd[1], "Hello, world!", 14);
    } else {               /* Parent */
        close(fd[1]);
        len = read(fd[0], buf, MAXLINE);
        printf("buf(%d): %s\n", len, buf);
    }
    return 0;
}
```

Both `write()` and `read()` are **blocking** function calls, but eventually they will have a **rendezvous**(汇聚点). Hence no need to use `sleep()` or `wait()`

```
$ gcc -o pipe2 pipe2.c
$ ./pipe2
buf(14): Hello, world!
```



Pipes

- Consider the following program. How to make sure child prints first?

```c
int main(int argc, char **argv) {
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed.\n");
        exit(1);
    } else if (rc == 0) {
        printf("hello\n");
    } else {

        printf("goodbye\n");
    }
    return 0;
}
```

Pipes

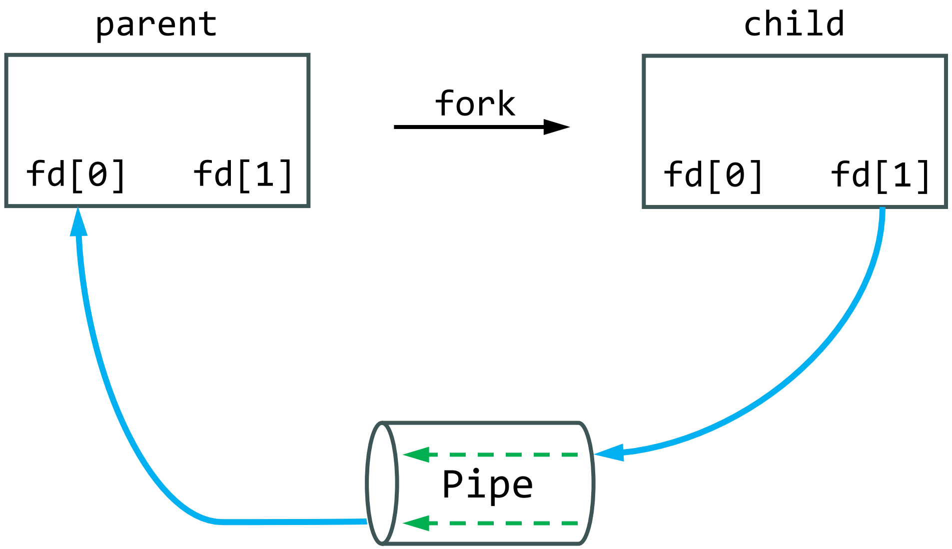
- Consider the following program. How to make sure child prints first?
  - Can you do this ***without*** calling `wait()` in parent?

```
int main(int argc, char **argv) {
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed.\n");
        exit(1);
    } else if (rc == 0) {
        printf("hello\n");
    } else {
        wait(NULL);
        printf("goodbye\n");
    }
    return 0;
}
```

Pipes

- Consider the following program. How to make sure child prints first?
  - Use `pipe()`. Make use of `read()` and `write()` rendezvous!

```c
int main(int argc, char **argv) {
    int fd[2];
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed.\n");
        exit(1);
    } else if (rc == 0) {
        close(fd[0]);
        printf("hello\n");
        write(fd[1], "k", 1);
        close(fd[1]);
    } else {
        close(fd[1]);
        char buf[2];
        read(fd[0], buf, 1);
        printf("goodbye\n");
        close(fd[0]);
    }
    return 0;
}
```

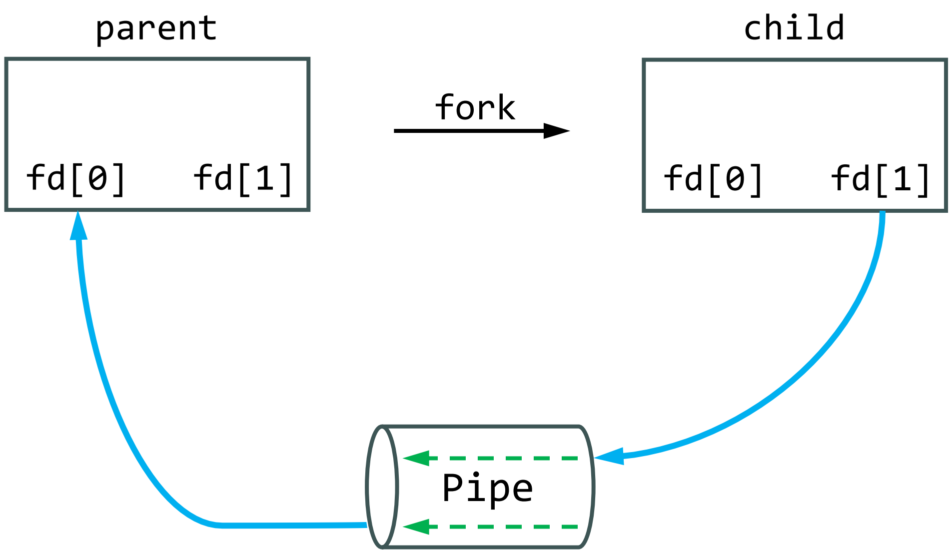


Pipes

- Consider the following program. How to make sure child prints first?
  - Use `pipe()`. Make use of `read()` and `write()` rendezvous!

```
int main(int argc, char **argv) {
    int fd[2];
    int rc = fork();
    if (rc < 0) {
        fprintf(stderr, "fork failed.\n");
        exit(1);
    } else if (rc == 0) {
        close(fd[0]);
        printf("hello\n");
        write(fd[1], "k", 1);
        close(fd[1]);
    } else {
        close(fd[1]);
        char buf[2];
        read(fd[0], buf, 1);
        printf("goodbye\n");
        close(fd[0]);
    }
    return 0;
}
```

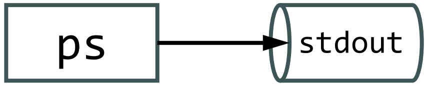
Blocking calls



Pipes

- Example (Shell Commands): **List** processes of current user and **sort** them by their PID using pipes (`cmd1 | cmd2`)

```
$ ps
    PID TTY          TIME CMD
1464792 pts/245  00:00:00 ps
1464794 pts/245  00:00:00 sort
 693845 pts/245  00:00:00 bash
1464793 pts/245  00:00:00 tee
1464791 pts/245  00:00:00 wc
```

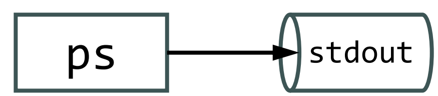


Pipes

Pipes

- Example (Shell Commands): **List** processes of current user and **sort** them by their PID using pipes (`` `cmd1 | cmd2` ``)

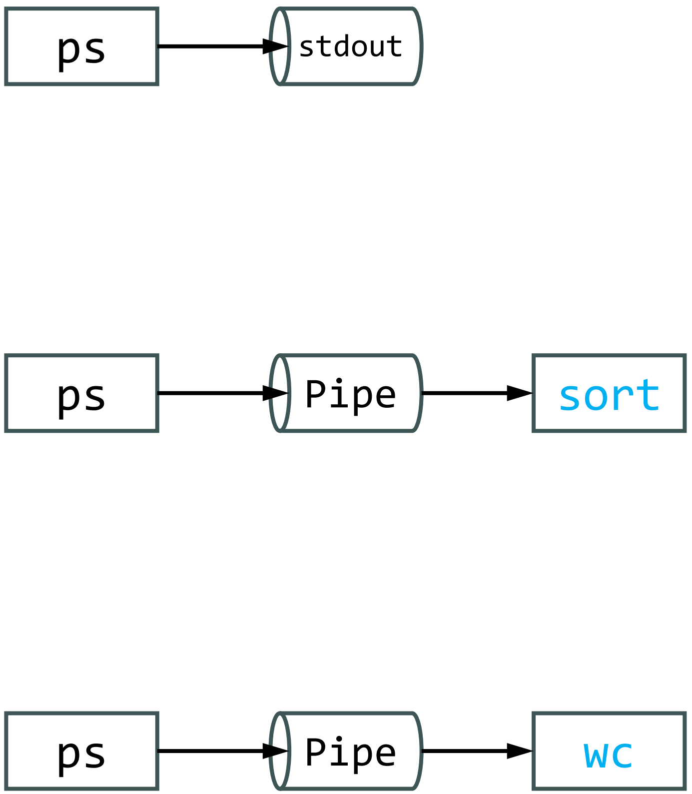
```
$ ps
    PID TTY          TIME CMD
1464792 pts/245  00:00:00 ps
1464794 pts/245  00:00:00 sort
 693845 pts/245  00:00:00 bash
1464793 pts/245  00:00:00 tee
1464791 pts/245  00:00:00 wc
$ ps | sort -k1n
 PID TTY          TIME CMD
 693845 pts/245  00:00:00 bash
1464791 pts/245  00:00:00 wc
1464792 pts/245  00:00:00 ps
1464793 pts/245  00:00:00 tee
1464794 pts/245  00:00:00 sort
```



Pipes

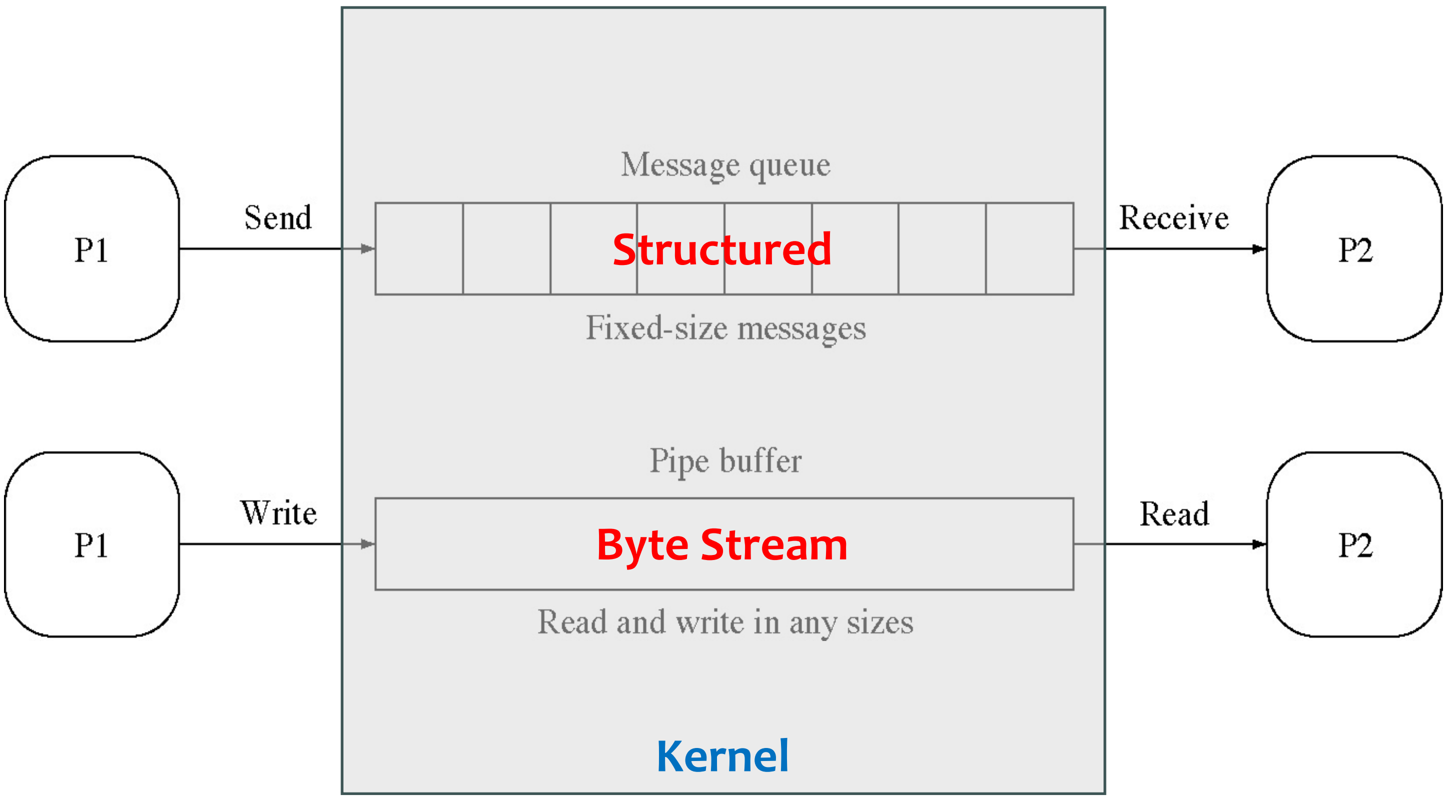
- Example (Shell Commands): **List** processes of current user and **sort** them by their PID using pipes (`` `cmd1 | cmd2` ``)

```
$ ps
    PID TTY          TIME CMD
1464792 pts/245  00:00:00 ps
1464794 pts/245  00:00:00 sort
 693845 pts/245  00:00:00 bash
1464793 pts/245  00:00:00 tee
1464791 pts/245  00:00:00 wc
$ ps | sort -k1n
 PID TTY          TIME CMD
 693845 pts/245  00:00:00 bash
1464791 pts/245  00:00:00 wc
1464792 pts/245  00:00:00 ps
1464793 pts/245  00:00:00 tee
1464794 pts/245  00:00:00 sort
$ ps | wc -l
6
```



Pipes

- Message Queues vs Pipes



- **Pipes**
  - Acts as a conduit (管道) allowing two processes to communicate
  - **Unnamed Pipes/Ordinary Pipes**: cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
    - Windows calls these *Anonymous Pipes*.
  - Limitations:
    - Ordinary pipes exist only while processes are communicating.
      - If either process terminates, the pipe no longer exists.
    - Ordinary pipes are unidirectional and half-duplex.
    - Ordinary pipes require parent-child relationship.
    - Synchronous and **blocking** operations: reads and writes to pipes are generally blocking, e.g., a process that reads from an empty pipe will be blocked until there is data to read.

Pipes

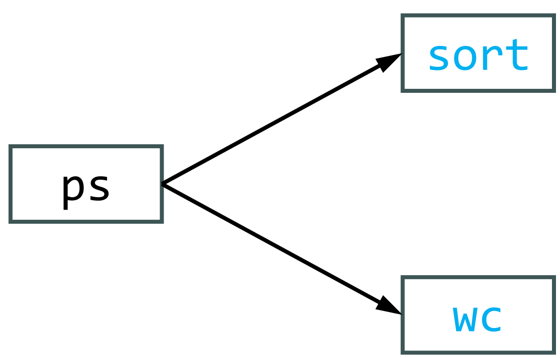
- **Pipes**
  - Acts as a conduit (管道) allowing two processes to communicate
  - **Unnamed Pipes/Ordinary Pipes:** cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
    - Windows calls these *Anonymous Pipes*.
  - **Named Pipes(命名管道):** provides a much more **powerful** communication tool compared with Ordinary Pipes. Named Pipes are also called **FIFOs** in UNIX systems.
  - Named pipes extend the concept of Ordinary Pipes by providing a named, persistent endpoint in the file system that can be accessed by multiple, potentially unrelated processes.

Named Pipes

- Named pipes are referred to as **FIFOs** in UNIX systems.
- Once created, they appear as typical **files** in the file system.
- A FIFO is created with the `mkfifo()` system call
  - and manipulated with ordinary `open()`, `read()`, `write()`, `close()` system calls
- A FIFO continues to exist until it is explicitly deleted from the file system.
  - Named pipes are **persistent** (cont. to exist after processes terminate).
  - Named pipes are **bidirectional**
    - although bidirectional, only **half-duplex** transmission is permitted
    - ..to achieve **full-duplex**, two **FIFOs** are typically used.
  - Named pipes **don't require parent-child relationship**
    - allows several readers and writers
  - Named pipes can be configured as **non-blocking** (defaults to blocking)
  - Named pipes work for processes on the same machine
    - ..for communication over networks, **sockets** must be used

Named Pipes

- Example (Shell Commands): Suppose we want to process the output of `ps` **twice** using `sort` and `wc` without using a temp file.
  - Since `ps` output changes all the time, if we execute `ps | sort -k1n` and `ps | wc -l` separately, we are not dealing with the same `ps` output.



- **Named Pipes**
  - Example (Shell Commands): Suppose we want to process the output of `ps` **twice** using `sort` and `wc` without using a temp file.

```
$ mkfifo fifo1
```

FIFO

Named Pipes

- Example (Shell Commands): Suppose we want to process the output of `ps` **twice** using `sort` and `wc` without using a temp file.

```
$ mkfifo fifo1
$ ls -l fifo1
prw-rw-r-- 1 zxx zxx 0 Mar 19 08:39 fifo1
$ file fifo1
fifo1: fifo (named pipe)
```

'p' in the file permission bits indicates Pipe.

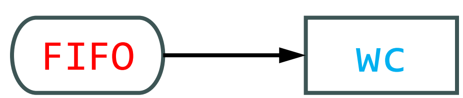
FIFO

Named Pipes

- Example (Shell Commands): Suppose we want to process the output of `` `ps` `` **twice** using `` `sort` `` and `` `wc` `` without using a temp file.

```
$ mkfifo fifo1
$ wc -l < fifo1 &
[1] 1464791
```

< **redirects** the output of `fifo1` to `` `wc` `` (as its stdin)
& **indicates** that `` `wc` `` runs in the background, so that we can run another program `` `sort` `` in the foreground



Pipes

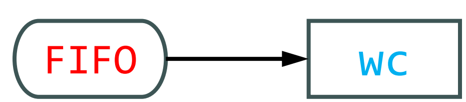
Named Pipes

- Example (Shell Commands): Suppose we want to process the output of `ps` **twice** using `sort` and `wc` without using a temp file.

```
$ mkfifo fifo1
$ wc -l < fifo1 &
[1] 1464791
```

`wc` runs in the background with job id [1]. We can use `jobs` to list background processes.

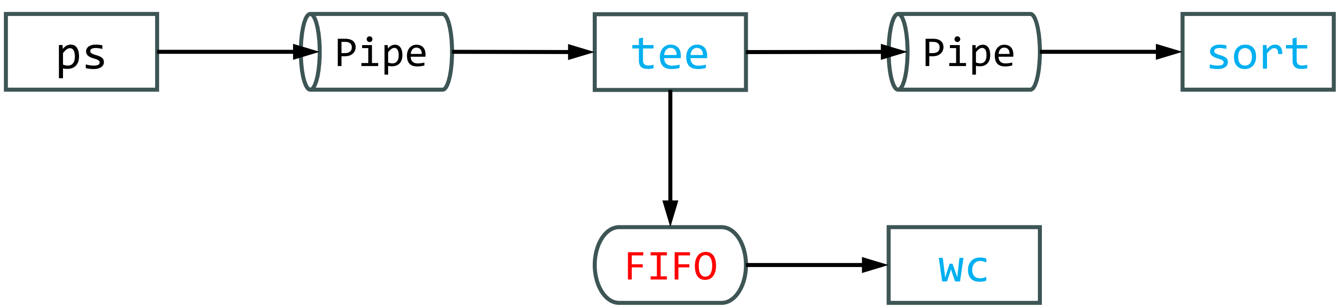
< **redirects** the output of `fifo1` to `wc` (as its stdin)
& **indicates** that `wc` runs in the background, so that we can run another program `sort` in the foreground



Named Pipes

- Example (Shell Commands): Suppose we want to process the output of `ps` **twice** using `sort` and `wc` without using a temp file.

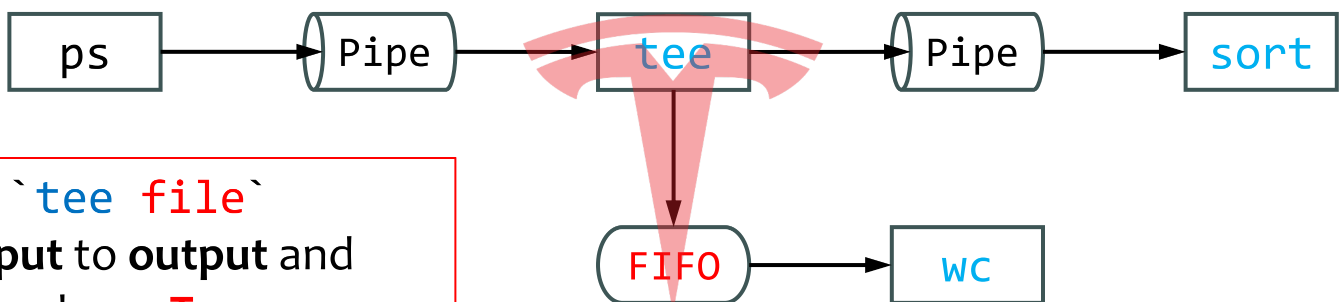
```
$ mkfifo fifo1
$ wc -l < fifo1 &
[1] 1464791
$ ps | tee fifo1 | sort -k1n
    PID TTY          TIME CMD
 693845 pts/245  00:00:00 bash
1464791 pts/245  00:00:00 wc
1464792 pts/245  00:00:00 ps
1464793 pts/245  00:00:00 tee
1464794 pts/245  00:00:00 sort
      6      24     180
```



Named Pipes

- Example (Shell Commands): Suppose we want to process the output of `ps` **twice** using `sort` and `wc` without using a temp file.

```
$ mkfifo fifo1
$ wc -l < fifo1 &
[1] 1464791
$ ps | tee fifo1 | sort -k1n
    PID TTY          TIME CMD
 693845 pts/245  00:00:00 bash
1464791 pts/245  00:00:00 wc
1464792 pts/245  00:00:00 ps
1464793 pts/245  00:00:00 tee
1464794 pts/245  00:00:00 sort
      6      24     180
```



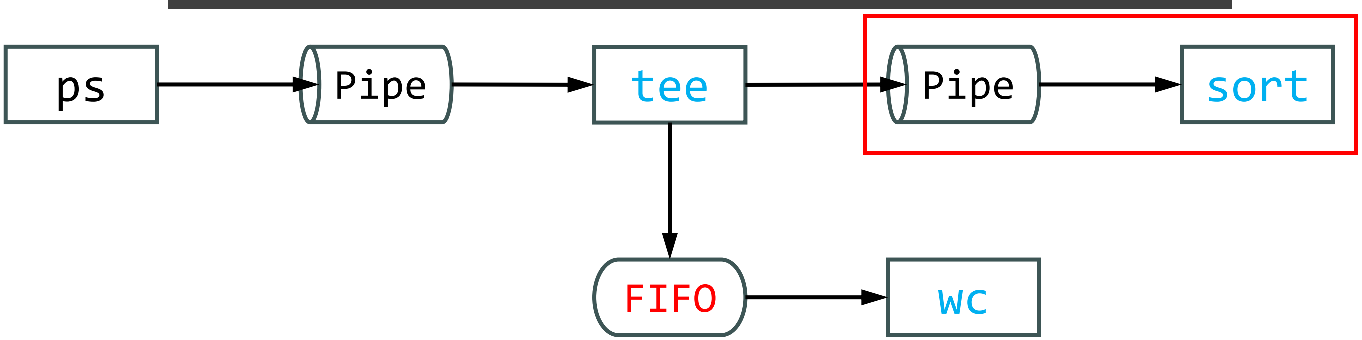
The program `tee file` duplicates **input** to **output** and **file**, hence the shape **T**

Named Pipes

- Example (Shell Commands): Suppose we want to process the output of `ps` **twice** using `sort` and `wc` without using a temp file.

```
$ mkfifo fifo1
$ wc -l < fifo1 &
[1] 1464791
$ ps | tee fifo1 | sort -k1n
    PID TTY          TIME CMD
 693845 pts/245  00:00:00 bash
1464791 pts/245  00:00:00 wc
1464792 pts/245  00:00:00 ps
1464793 pts/245  00:00:00 tee
1464794 pts/245  00:00:00 sort
      6      24     180
```

sorted

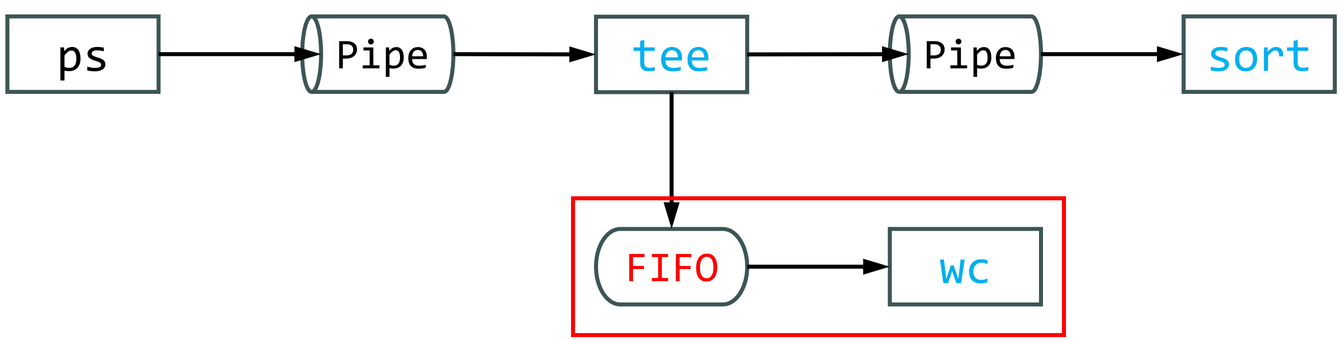


Named Pipes

- Example (Shell Commands): Suppose we want to process the output of `ps` **twice** using `sort` and `wc` without using a temp file.

```
$ mkfifo fifo1
$ wc -l < fifo1 &
[1] 1464791
$ ps | tee fifo1 | sort -k1n
    PID TTY          TIME CMD
 693845 pts/245  00:00:00 bash
1464791 pts/245  00:00:00 wc
1464792 pts/245  00:00:00 ps
1464793 pts/245  00:00:00 tee
1464794 pts/245  00:00:00 sort
      6       24      180
```

`wc -l` output



Thank you!