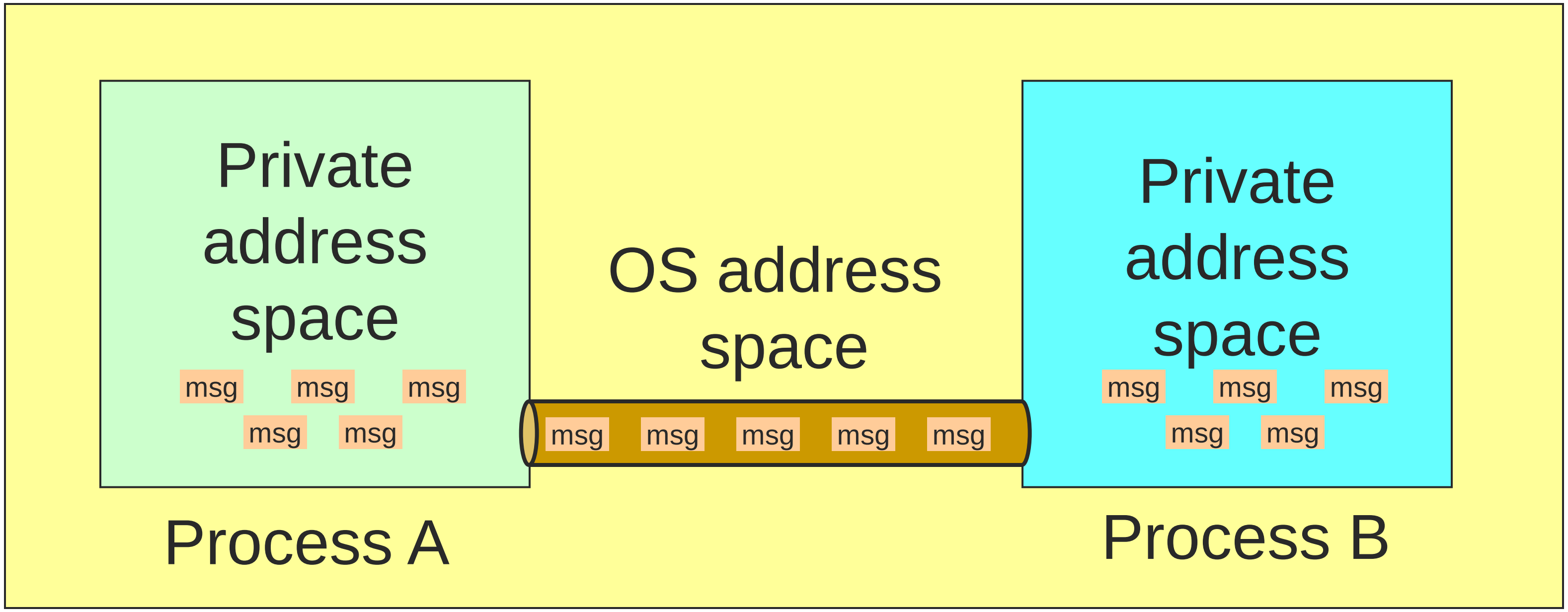
Interprocess Communication

IPC Solutions

- Two options
  - Support some form of shared address space
    - Shared memory, memory mapped files
  - Use OS mechanisms to transport data from one address space to another
    - Pipes, FIFOs
    - Messages, signals

Communication Over a Pipe



UNIX Pipes

```
#include <unistd.h>
int pipe(int fildes[2]);
```

- Create a message pipe
  - Anything can be written to the pipe, and read from the other end
  - Data is received in the order it was sent
  - OS enforces mutual exclusion: only one process at a time
  - Accessed by a file descriptor, like an ordinary file
  - Processes sharing the pipe must have same parent in common
- Returns a pair of file descriptors
  - `fildes[0]` is connected to the read end of the pipe
  - `fildes[1]` is connected to the write end of the pipe

UNIX Pipe Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>

int main(void) {
    int pfds[2];
    char buf[30];

    pipe(pfds);

    if (!fork()) {
        printf(" CHILD: writing to pipe\n");
        write(pfds[1], "test", 5);
        printf(" CHILD: exiting\n");
        exit(0);
    } else {
        printf("PARENT: reading from pipe\n");
        read(pfds[0], buf, 5);
        printf("PARENT: read \"%s\"\n", buf);
        wait(NULL);
    }

    return 0;
}
```

fildes[0] = read end of the pipe
fildes[1] = write end of the pipe

Really using a pipe

- Command-line pipe
  - **`ls | wc -l`**
- How do we implement this with **`pipe()`**?
  - Need to attach the **`stdout`** of **`ls`** to the **`stdin`** of **`wc`**

Duplicating a file descriptor

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

```
int dup(int oldfd);
```

- Create a copy of an open file descriptor
- Returns:
  - Return value ≥ 0 : Success - New file descriptor on success
  - Return value = -1: Error, check value of `errno`
- Parameters:
  - `oldfd`: the open file descriptor to be duplicated

Duplicating a file descriptor

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

- Create a copy of an open file descriptor: put new copy in a specific location!
  - Closes `newfd` if it was open

- Returns:
  - Return value ≥ 0 : Success - New file descriptor on success
  - Return value = -1: Error, check value of `errno`
- Parameters:
  - `oldfd`: the open file descriptor to be duplicated

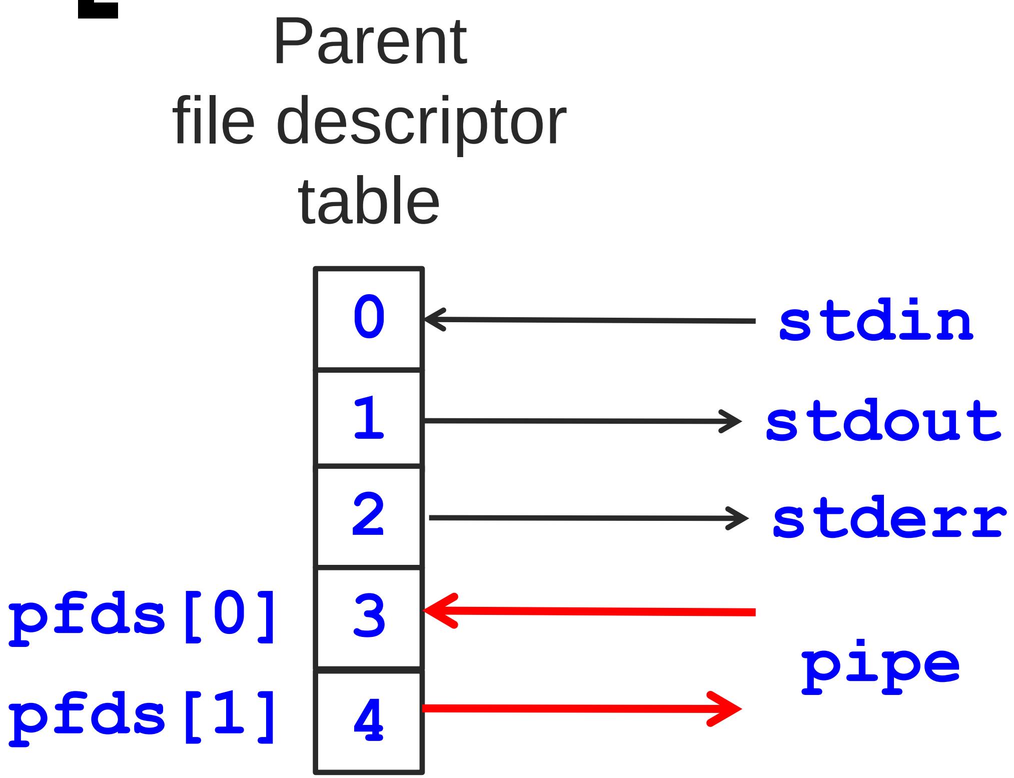
UNIX Pipe Example: `ls | wc -l`

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

int main(void) {
    int pfds[2];

    pipe(pfds);
    if (!fork()) {
        close(1);          /* close stdout */
        dup(pfds[1]);      /* make stdout pfds[1] */
        close(pfds[0]);    /* don't need this */
        execlp("ls", "ls", NULL);
    } else {
        close(0);          /* close stdin */
        dup(pfds[0]);      /* make stdin pfds[0] */
        close(pfds[1]);    /* don't need this */
        execlp("wc", "wc", "-l", NULL);
    }
    return 0;
}
```

`fildes[0]` = read end of the pipe
`fildes[1]` = write end of the pipe

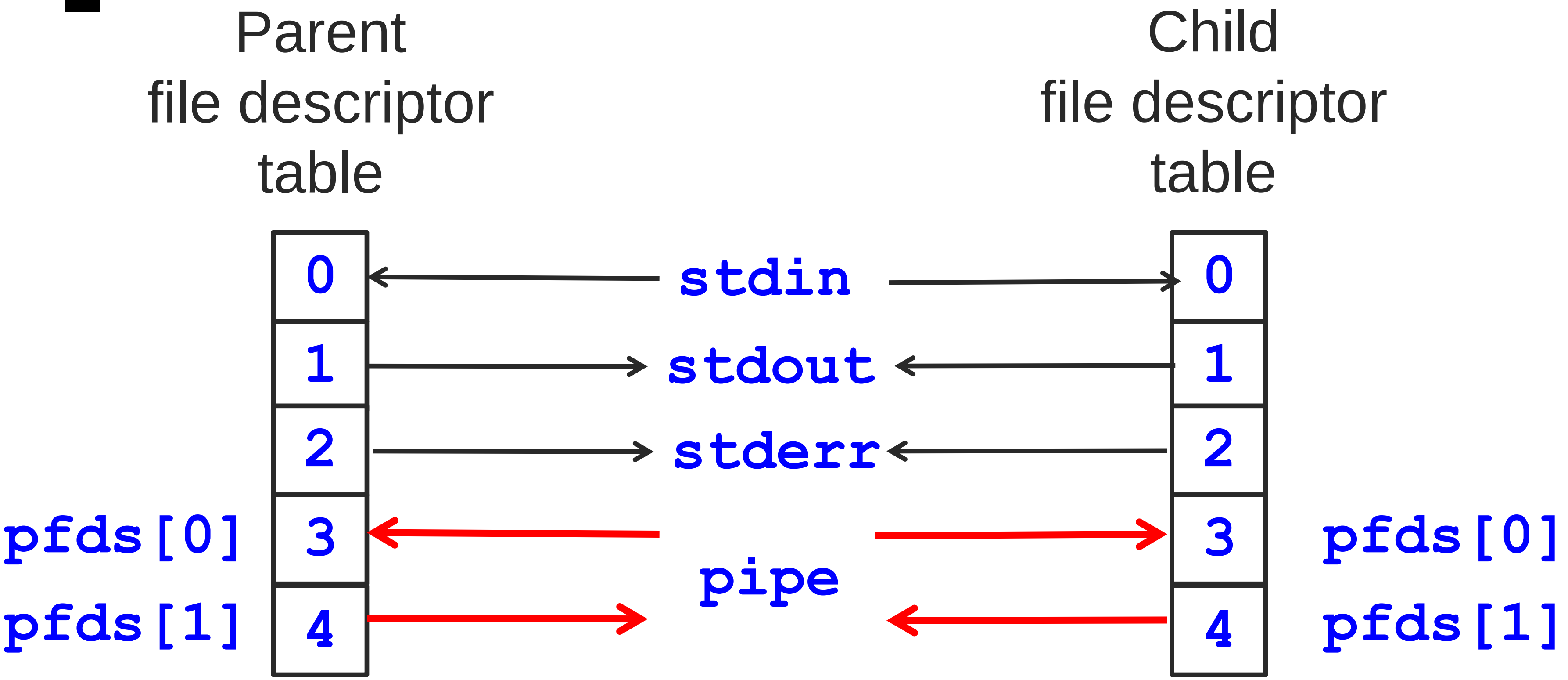
Run demo

UNIX Pipe Example: `ls | wc -l`



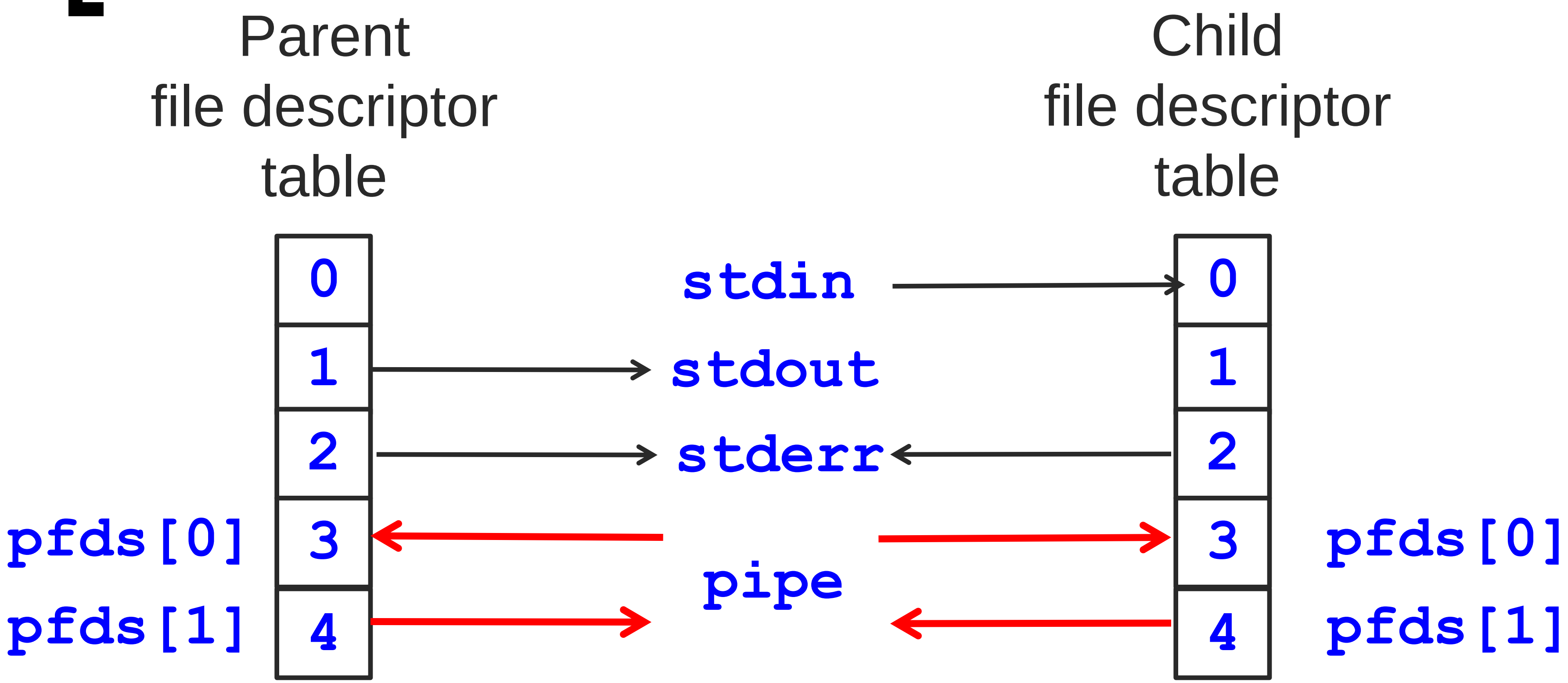

```
pipe(pfds);
```

UNIX Pipe Example: `ls | wc -l`




```
pipe(pfds);
fork()
```

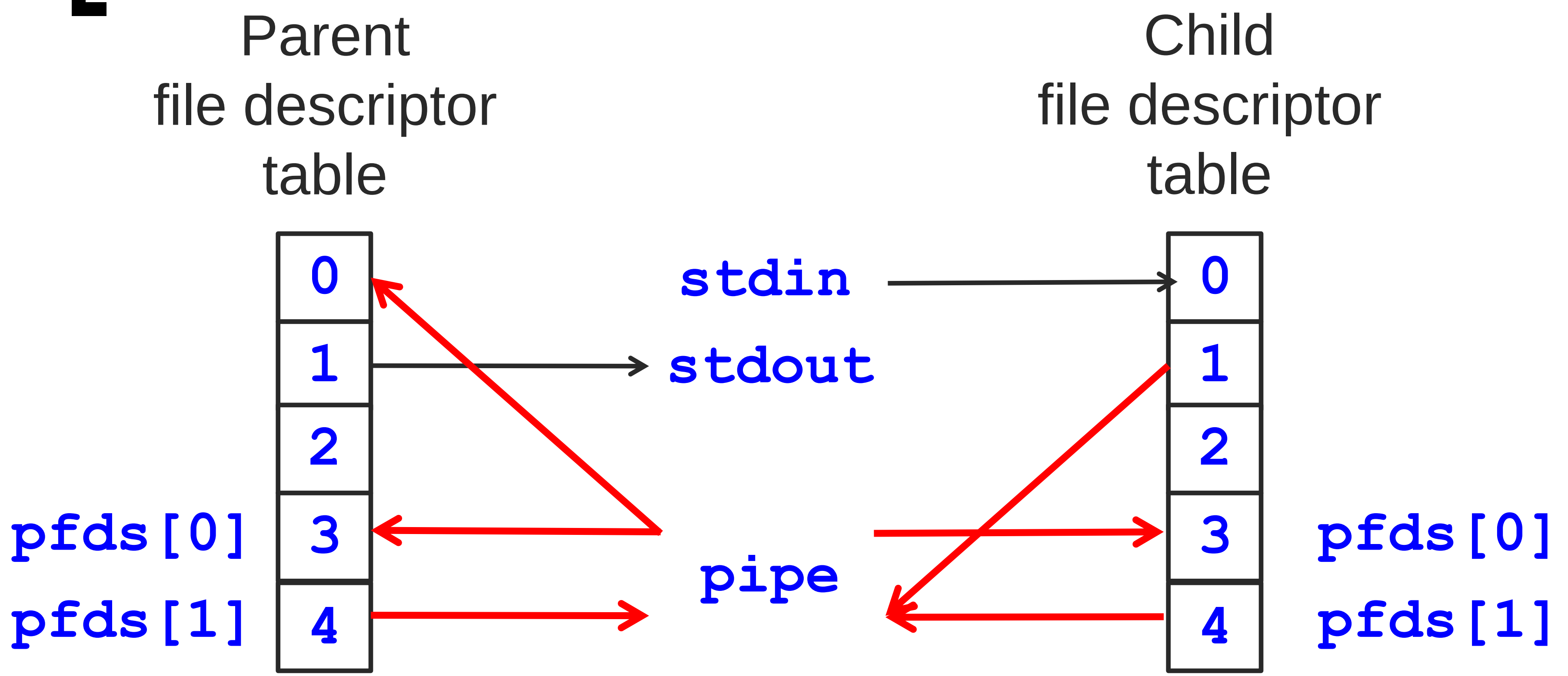
UNIX Pipe Example: `ls | wc -l`




```
pipe(pfds);
fork()
close(0)
```

```
close(1)
```

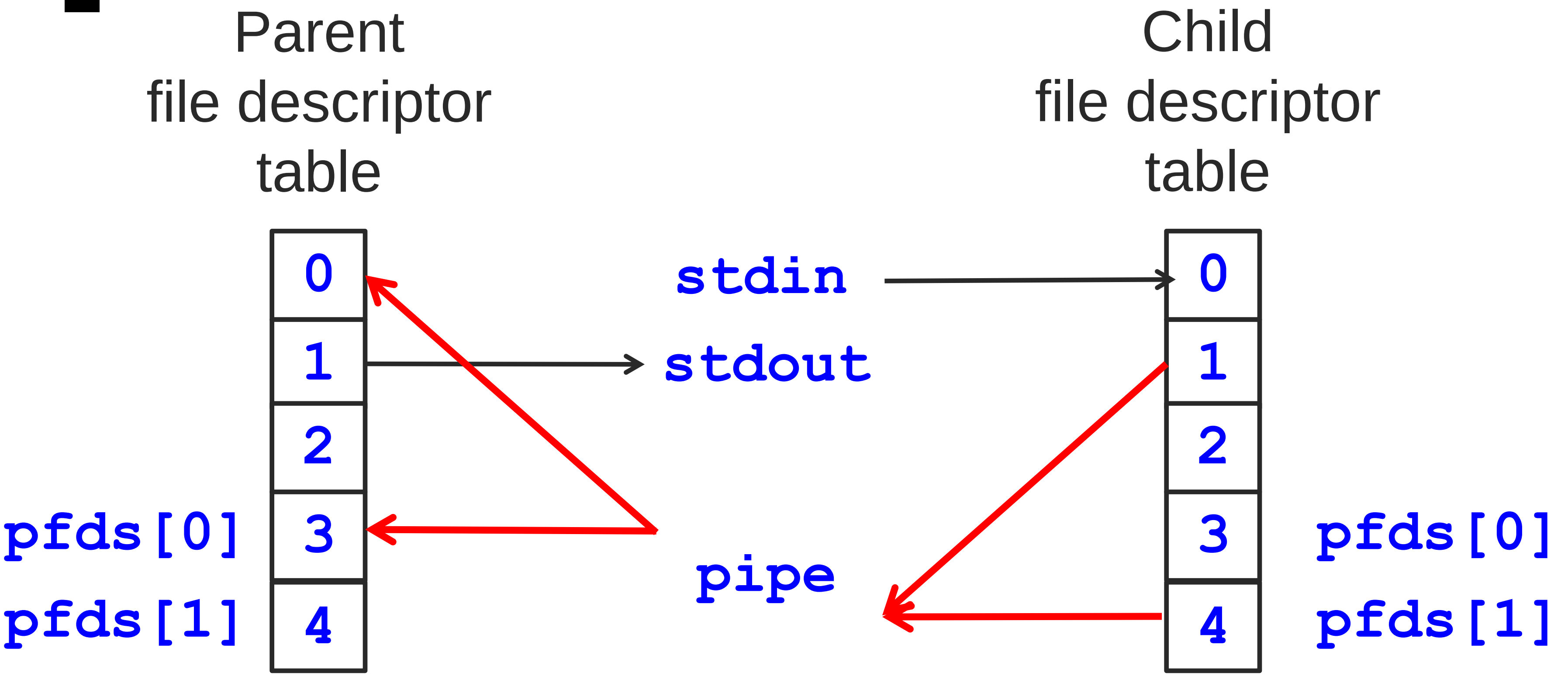
UNIX Pipe Example: `ls | wc -l`




```
pipe(pfds);
fork()
close(0)
dup(pfds[0]);
```

```
close(1)
dup(pfds[1]);
```

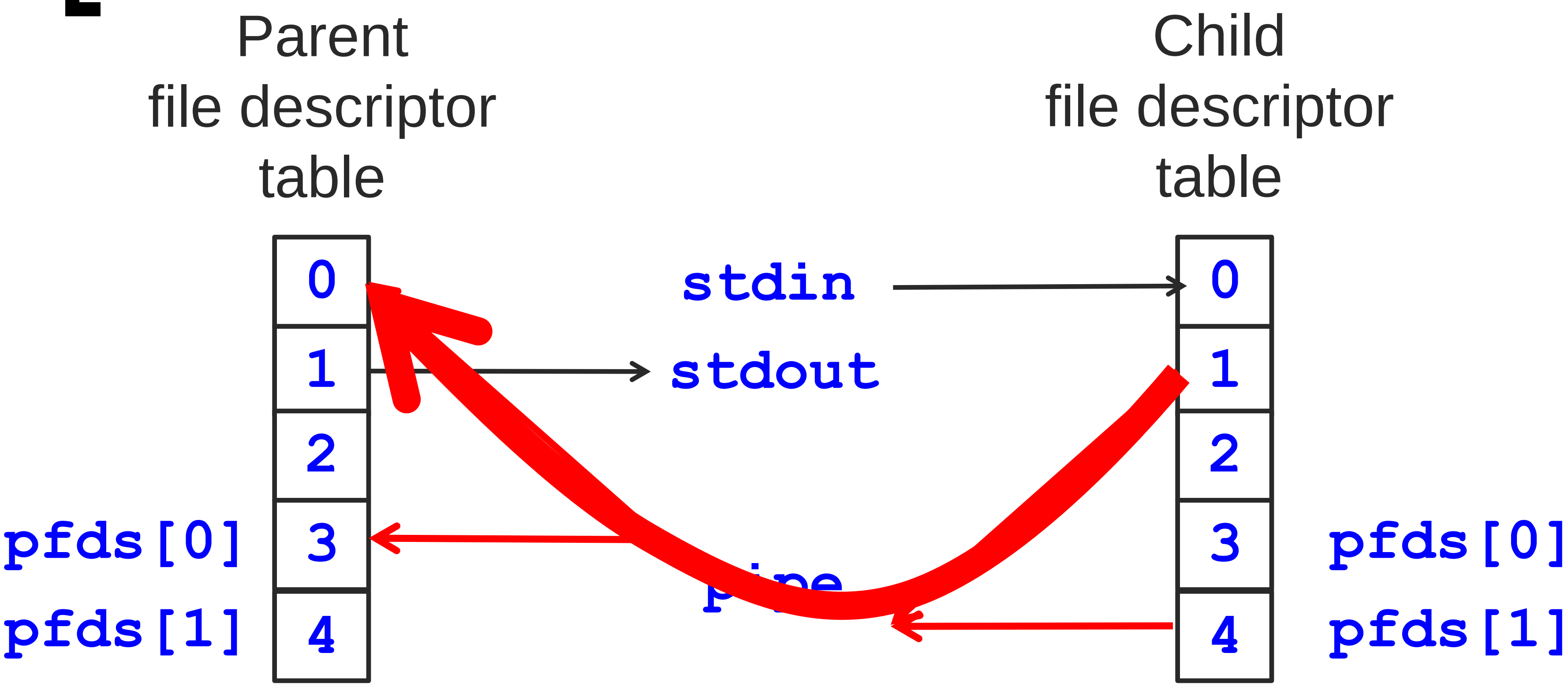
UNIX Pipe Example: `ls | wc -l`



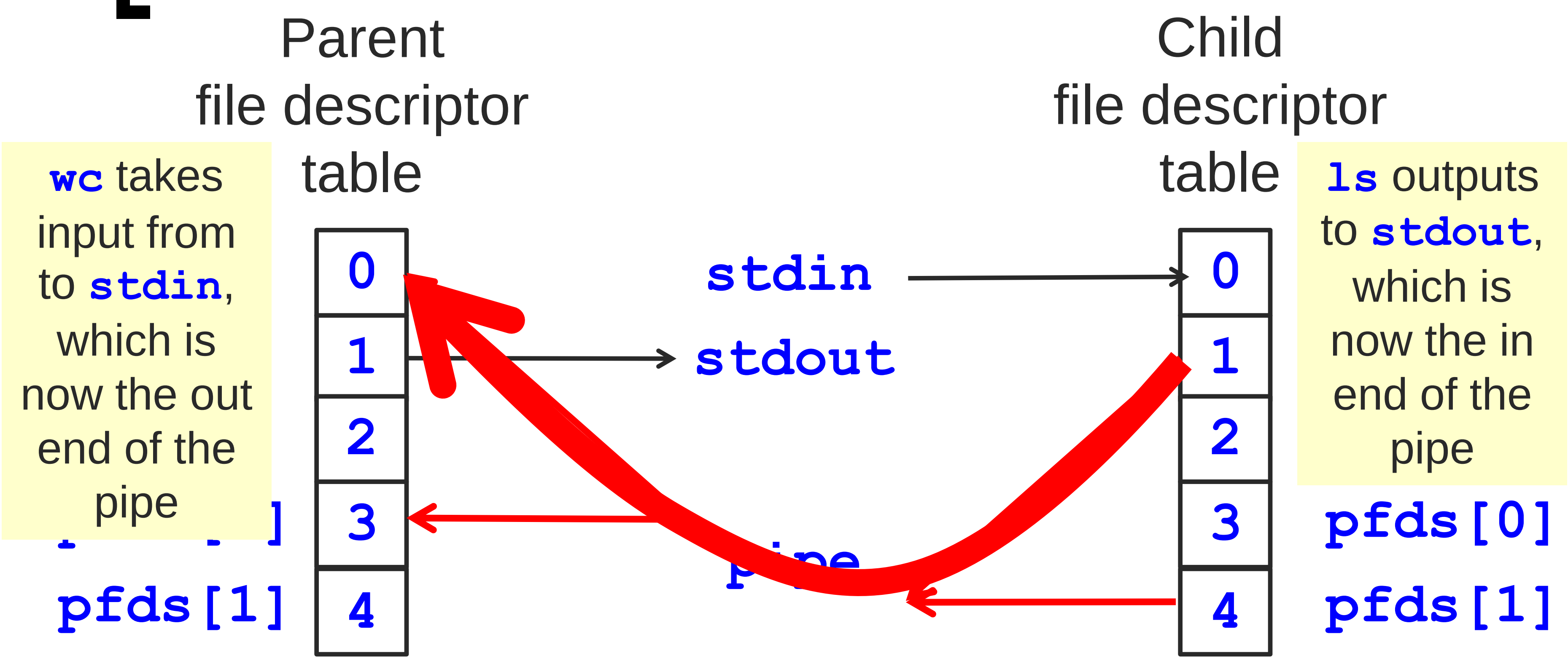

```
pipe(pfds);
fork()
close(0)
dup(pfds[0]);
close(pfds[1]);
```

```
close(1)
dup(pfds[1]);
close(pfds[0]);
```

UNIX Pipe Example: `ls | wc -l`



UNIX Pipe Example: `ls | wc -l`




```
execlp("wc", "wc", "-l", NULL);
```

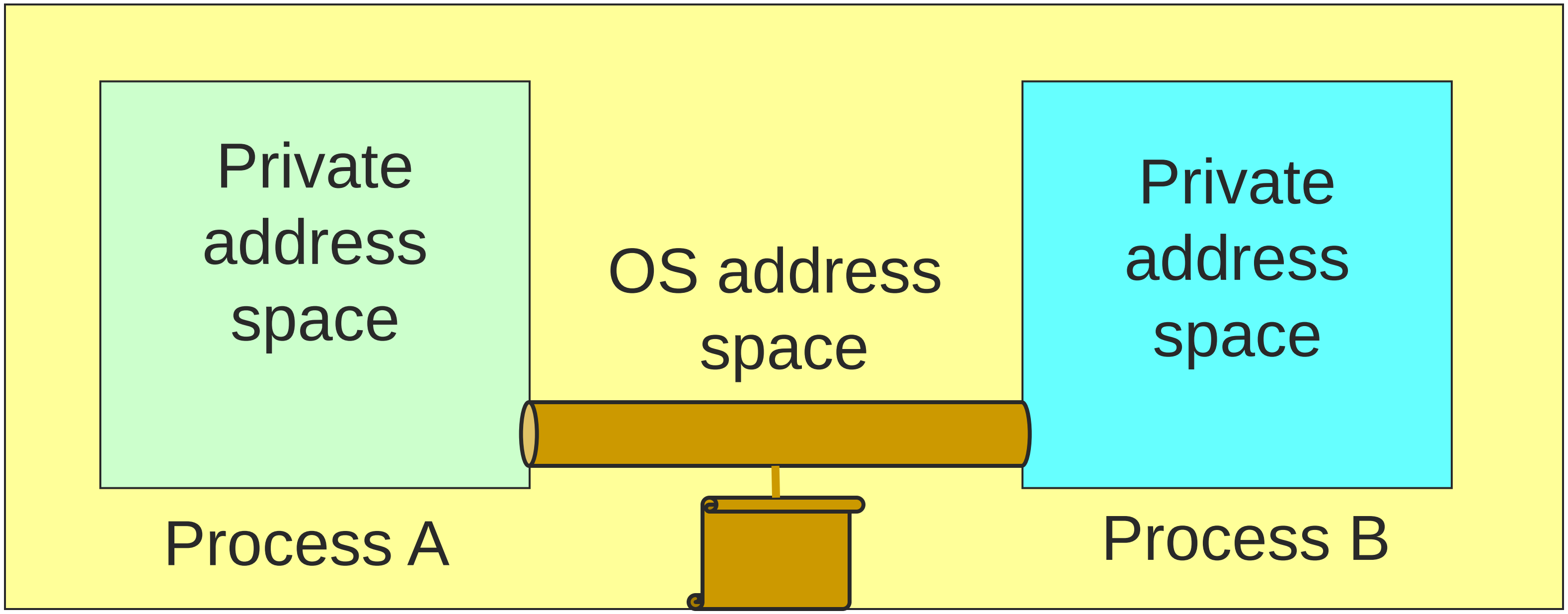
```
execlp("ls", "ls", NULL);
```

FIFOs

- A pipe disappears when no process has it open
- FIFOs = named pipes
  - Special pipes that persist even after all the processes have closed them
  - Actually implemented as a file

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

int status;
...
status = mkfifo("/home/cnd/mod_done", /* mode=0644 */
              S_IWUSR | S_IRUSR | S_IRGRP | S_IROTH);
```

Communication Over a FIFO



- First **`open`** blocks until second process opens the FIFO
- Can use **`O_NONBLOCK`** flag to make operations non-blocking
- FIFO is persistent : can be used multiple times
- Like pipes, OS ensures atomicity of writes and reads

FIFO Example: Producer-Consumer

- Producer
  - Writes to fifo
- Consumer
  - Reads from fifo
  - Outputs data to file
- Fifo
  - Ensures atomicity of write

FIFO Example

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/stat.h>
#include "restart.h"

int main (int argc, char *argv[]) {
  int requestfd;

  if (argc != 2) { /* name of consumer fifo on the command line */
    fprintf(stderr, "Usage: %s fifoname > logfile\n", argv[0]);
    return 1;
  }
```

FIFO Example

```
/* create a named pipe to handle incoming requests */
if ((mkfifo(argv[1], S_IRWXU | S_IWGRP| S_IWOTH) == -1)
    && (errno != EEXIST))
{
  perror("Server failed to create a FIFO");
  return 1;
}
/* open a read/write communication endpoint to the pipe */
if ((requestfd = open(argv[1], O_RDWR)) == -1) {
  perror("Server failed to open its FIFO");
  return 1;
}
/* Write to pipe like you would to a file */
...
}
```

What if there are multiple producers?

What if there are multiple producers?

- Examples
  - Multiple children to compute in parallel; wait for output from any
  - Network server connected to many clients; take action as soon as any one of them sends data
- Use **`read`**, **`write`**, **`scanf`**, etc.
  - Problem
    - Blocks waiting for that one file, even if another has data ready & waiting!
  - Solution
    - Need a way to wait for any one of a set of events to happen
    - Something similar to **`wait()`** to wait for any child to finish, but for events on file descriptors

Select and Poll

- Checking for input with select/poll
  - Similar functions
    - Parameters
      - Set of file descriptors
      - Set of events for each descriptor
      - Timeout length
    - Return value
      - Set of file descriptors
      - Events for each descriptor
- Notes
  - Select is somewhat simpler
  - Poll supports more events

Select and Poll: Prototypes

- Select
  - Wait for readable/writable file descriptors

```
#include <sys/time.h>
int select (int num_fds, fd_set* read_set, fd_set*
    write_set, fd_set* except_set, struct timeval*
    timeout);
```

- Poll
  - Poll file descriptors for events

```
#include <poll.h>
int poll (struct pollfd* pfds, nfds_t nfds, int
    timeout);
```

Select

```
int select (int num_fds, fd_set* read_set, fd_set*
    write_set, fd_set* except_set, struct timeval*
    timeout);
```

- Wait for readable/writable file descriptors.
- Return:
  - Number of descriptors ready
  - -1 on error, sets **`errno`**
- Parameters:
  - **`num_fds`**:
    - number of file descriptors to check, numbered from 0
  - **`read_set`**, **`write_set`**, **`except_set`**:
    - Sets (bit vectors) of file descriptors to check for the specific condition
  - **`timeout`**:
    - Time to wait for a descriptor to become ready

File Descriptor Sets

- Bit vectors
  - Often 1024 bits, only first **num_fds** checked
  - Macros to create and check sets

```
fds_set myset;
void FD_ZERO (&myset);      /* clear all bits */
void FD_SET (n, &myset);    /* set bits n to 1 */
void FD_CLEAR (n, &myset); /* clear bit n */
int FD_ISSET (n, &myset);  /* is bit n set? */
```

File Descriptor Sets

- Three conditions to check for
  - Readable
    - Data available for reading
  - Writable
    - Buffer space available for writing
  - Exception
    - Out-of-band data available (TCP)

Select: Example

```
fd_set my_read;
FD_ZERO(&my_read);
FD_SET(0, &my_read);

if (select(1, &my_read, NULL, NULL) == 1) {
   ASSERT(FD_ISSET(0, &my_read);
   /* data ready on stdin */
```

Poll

```
#include <poll.h>
int poll (struct pollfd* pfds, nfds_t nfds, int
    timeout);
```

- Poll file descriptors for events.
- Return:
  - Number of descriptors with events
  - -1 on error, sets **`errno`**
- Parameters:
  - **`pfds`**:
    - An array of descriptor structures. File descriptors, desired events and returned events
  - **`nfds`**:
    - Length of the **`pfds`** array
  - **`timeout`**:
    - Timeout value in milliseconds

Descriptors

- Structure

```
struct pollfd {
    int fd;                   /* file descriptor */
    short events;             /* queried event bit mask */
    short revents;            /* returned event mask */
```

- Note:
  - Any structure with **`fd`** < 0 is skipped

Event Flags

- **`POLLIN`**:
  - data available for reading
- **`POLLOUT`**:
  - Buffer space available for writing
- **`POLLERR`**:
  - Descriptor has error to report
- **`POLLHUP`**:
  - Descriptor hung up (connection closed)
- **`POLLVAL`**:
  - Descriptor invalid

Poll: Example

```
struct pollfd my_pfds[1];

my_pfds[0].fd = 0;
my_pfds[0].events = POLLIN;

if (poll(&my_pfds, 1, INFTIM) == 1) {
   ASSERT (my_pfds[0].revents & POLLIN);
   /* data ready on stdin */
```

IPC Solutions

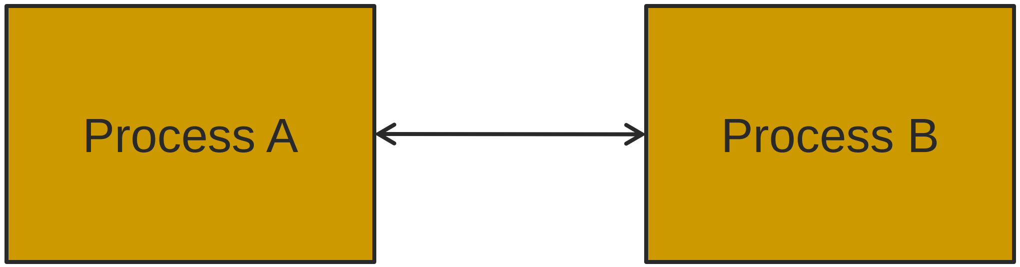
- Two options
  - Support some form of shared address space
    - Shared memory, memory mapped files
  - Use OS mechanisms to transport data from one address space to another
    - Pipes, FIFOs
    - Messages, signals

Message-based IPC

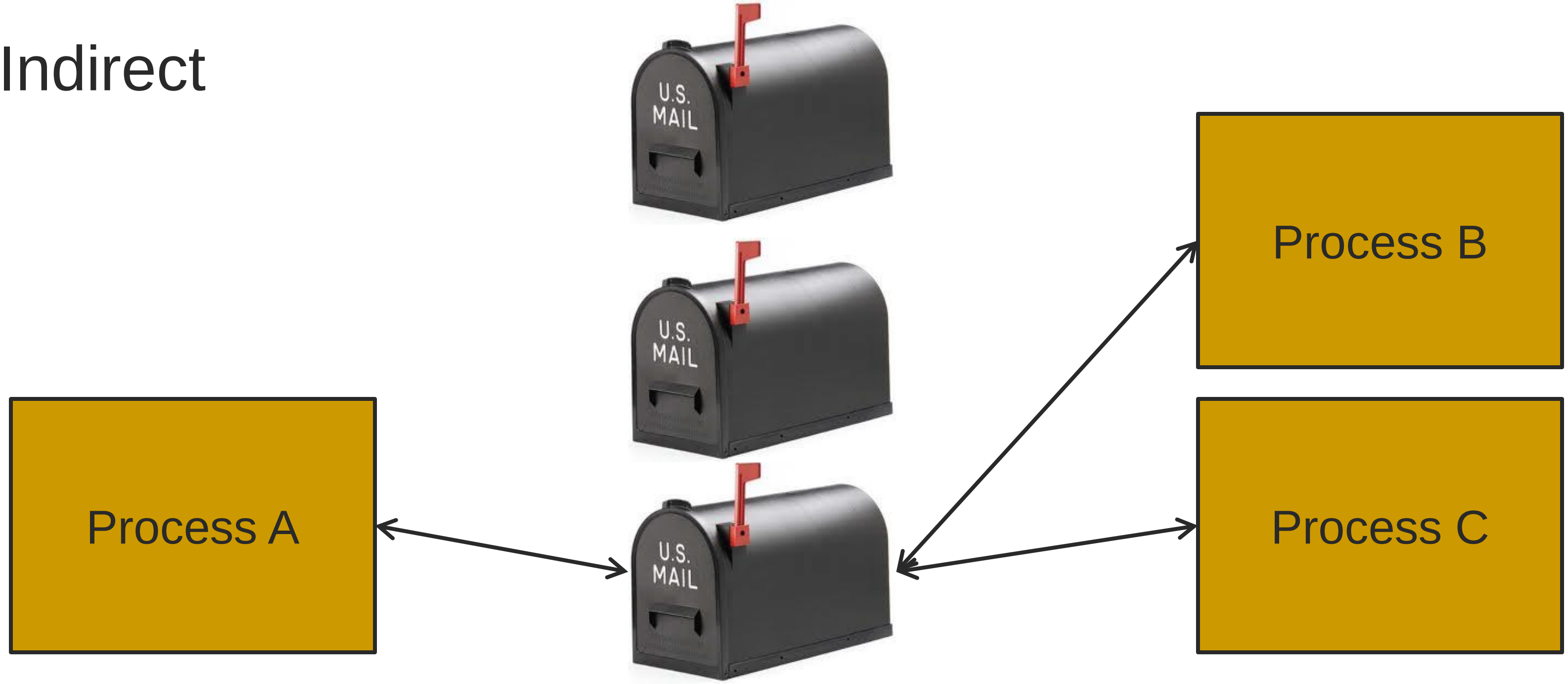
- Message system
  - Enables communication without resorting to shared variables
- To communicate, processes P and Q must
  - Establish a communication link between them
  - Exchange messages
- Two operations
  - send(message)
  - receive(message)

Message Passing

Direct



Indirect



Direct Message Passing

- Processes must name each other explicitly
  - `send (P, message)`
    - Send a message to process **`P`**
  - `receive(Q, message)`
    - Receive a message from process **`Q`**
  - `receive(&id, message)`
    - Receive a message from any process
- Link properties
  - Established automatically
  - Associated with **exactly** one pair of processes
  - There exists **exactly** one link between each pair
- Limitation
  - Must know the name or ID of the process(es)

Indirect Message Passing

- Process names a mailbox (or port)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Link properties
  - Established only if processes share a common mailbox
  - May be associated with **many** processes
  - Each pair of processes may share **multiple** links
  - Link may be unidirectional or bi-directional

Mailbox Ownership

- Process
  - Only the owner receives messages through mailbox
  - Other processes only send.
  - When process terminates, any “owned” mailboxes are destroyed
- System
  - Process that creates mailbox owns it (and so may receive through it) but may transfer ownership to another process.

Indirect Message Passing

- Mailboxes are a resource
  - Create and Destroy
- Primitives
  - **`send(A, message)`**
    - Send a message to mailbox **`A`**
  - **`receive(A, message)`**
    - Receive a message from mailbox **`A`**

Indirect Message Passing

- Mailbox sharing
  - `P1`, `P2`, and `P3` share mailbox `A`
  - `P1`, sends; `P2` and `P3` receive
  - Who gets the message?
- Options
  - 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 arbitrarily select the receiver and notify the sender

IPC and Synchronization

- Blocking == synchronous
  - Blocking send
    - Sender blocks until the message is received
  - Blocking receive
    - Receiver blocks until a message is available
- Non-blocking == asynchronous
  - Non-blocking send
    - Sender sends the message and continues
  - Non-blocking receive
    - Receiver receives a valid message or null

Buffering

- IPC message queues
  1. Zero capacity
     - No messages may be queued
     - Sender must wait for receiver
  2. Bounded capacity
     - Finite buffer of n messages
     - Sender blocks if link is full
  3. Unbounded capacity
     - Infinite buffer space
     - Sender never blocks

Buffering

- Is a buffer needed?

```
P1:   send(P2, x)         P2: receive(P1, x)
      receive(P2, y)          send(P1, y)
```

- Is a buffer needed?

```
P1:   send(P2, x)         P2: send(P1, x)
      receive(P2, y)          receive(P1, y)
```

Example: Message Passing

```
void Producer() {
    while (TRUE) {
        /* produce item */
        build_message(&m, item);
        send(consumer, &m);
        receive(consumer, &m); /* wait for ack */
    }
}

void Consumer {
    while(TRUE) {
        receive(producer, &m);
        extract_item(&m, &item);
        send(producer, &m); /* ack */
        /* consume item */
    }
}
```

Signals == Messages

- Signals are a simple form of message passing
  - Non-blocking
  - No buffering