Cooperating Processes

- Independent processes cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process
  - Advantages
    - Information sharing
    - Computational speed-up
  - Disadvantages of process cooperation
    - Data corruption, deadlocks
    - Requires processes to synchronize their processing

- Need a mechanism for facilitating communications and synchronization between processes
  - Involve the OS for communication

Message Passing vs. Shared Memory

- Shared Memory
  - Multiple processes map a region of memory into their address spaces
  - Changes made by one show up instantly in the address space of all others sharing the same region
    - This is the idea behind threading
  - Synchronization of access to shared structures must be handled
    - E.g. Using Semaphore

- Message Passing
  - Think of multiple processes operating by passing small chunks of information (message) between them

Message Passing: Unix Pipes

- Pipe sets up communication channel between two (related) processes

  ![Diagram of processes communicating through a pipe](image)

- One process writes to the pipe, the other reads from the pipe
- Looks exactly the same as reading from/to a file
  - Use systems calls read and write
- System call:
  ```c
  int fd[2] ;
  pipe(fd) ; //creates the pipe
  fd[0] now holds descriptor to read from pipe
  fd[1] now holds descriptor to write into pipe
  ```
Example

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
char *message = "This is a message!!!
int main(){
    char buf[1024] ;
    int fd[2] ;
    pipe(fd);    /*create pipe*/
    if (fork() != 0) { /* I am the parent */
        write(fd[1], message, strlen (message) + 1) ;
        printf("Parent %d sent the message\n", getpid());
    }
    else { /*Child code */
        read(fd[0], buf, 1024) ;
        printf("Got this from Parent %d: %s\n", getppid(), buf) ;
    }
    return 0;
}
```

Pipe for pipeline

- Can set up pipeline
- E.g. `who | sort | lpr`

Steps involved in pipeline

- Shell creates a pipe
- Use fork() to make three children
- Children inherit the descriptors
- Close the pipe links which are not needed
- Duplicates file descriptor using dup2(...) making them aliases and then deleting the old file descriptor
- E.g. output of "who" is which is to written to stdout is now written to the pipe
  ```c
  dup2( fd[1], 1); //connect the write end of a pipe to stdout
  ```
- Replace children by programs using exec()

Multiple Readers and Writes

- Perfectly possible to have multiple readers and writers attached to a pipe
- Can cause confusion or problems
  ```c
  Solution: process closes the link it does not need
  ```
Pipe size, full or empty

- Size of a pipe is finite
  - Only a certain amount of bytes can remain in the pipe without being read

- read() & write() are blocking by default
  - If pipe to read is empty
  - If pipe to write is full or there not enough room in the pipe
    ➢ Since write() can block in middle of its output, then output from multiple writers may be mixed up

- Can make them non-blocking
  - May want to return an error instead
  - Want to poll several pipes in turn until one has data
  - Using fcntl()

Limitations of Pipes

- Processing using a pipe must come from a common ancestor
  - E.g. parent and child

- Pipes are not permanent
  - Disappear when process terminates

- Communication is one way (half duplex)

- Readers and Writers do not know each other

Message Queue

- Provides standard interface for creating/opening message queue to sending and receive messages

- Similar to pipes in that they are opened and closed and have readers and writers

- Different compared to pipes in that messages have structure (attributes), and message have associate priorities

  - Attributes: maximum size, size of each message, number of messages currently in queue etc.

POSIX Message Queue Implementation

- Header: mqueue.h

  - The header defines the \texttt{mqd_t} (message queue descriptor type), a long integer to identify the queue

  - \texttt{mqd_t} is used by other subroutines to refer to that message queue
## Creating a queue

- **mq_open**
  ```c
  mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);
  ```
  - Creates a message queue or opens an existing one
  - **name** of the queue
    - processes can operate on the same queue by passing the same name to `mq_open`
  - **oflag** is operation flags
    - controls whether the queue is created or merely accessed
    - Defined constants to specify operation by a process
  - **mode** determines the mode
    - Used to set rwx permissions
    - Defined constants described in `<sys/stat.h>`
  - **attr**
    - struct that contains size of message, queue length etc.
    - If argument is NULL, then default attributes are selected
  - Returns `mqd_t`

## mq_open Example

### Option 1
```c
mqd_t mq;

mq = mq_open(Q_NAME, O_RDONLY|O_CREAT|O_EXCL, S_IRWXU, NULL);
```
- Oflag constants: Opened in read only exclusive mode
- `S_IRWXU`: read, write, execute by user/owner
- `NULL` for `mq_attr` (implies use default attributes)

### Option 2
```c
mq = mq_open(Q_NAME, O_WRONLY, S_IRWXU, NULL);
```
- Message queue by name `Q_NAME` opened in write only mode

## Functions for send and receive

- **mq_send**
  ```c
  mqd_t mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio);
  ```
  - adds the message referred by `msg_ptr` to the message queue specified by `mqdes`
  - `msg_len` is the length of the message in bytes referred by `msg_ptr`
  - `msg_prio` is a non-negative integer that specifies the priority of this message
  - Messages are placed on the queue in decreasing order of priority

- **mq_receive**
  ```c
  size_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);
  ```
  - receive the oldest of the highest priority message(s) from the message queue
  - message is removed from the queue and copied to the buffer referred by the `msg_ptr` argument
  - returns the length of the selected message in bytes and the message is removed from the queue

- **Queue full or empty**
  - The sending/receiving process is blocked

## More mqueue functions

### Cleanup
- **mq_close**
  ```c
  int mq_close(mqd_t mqdes);
  ```
  - removes the association between the message queue descriptor, `mqdes`, and its message queue

- **mq_unlink**
  ```c
  int mq_unlink(const char *name);
  ```
  - Removes the message queue referred by `name`
  - Is blocking if another process(es) has the mqueue open
  - After a successful call to `mq_unlink()` with `name`, a call to `mq_open()` with `name` fails if the flag `O_CREAT` is not set in flags

### Getting Attributes
- **mq_getattr**
  ```c
  int mq_getattr(mqd_t mqdes, struct mq_attr *mqstat)
  ```
  - Similarly there is `mqattr` to set attributes
Synchronous vs. Asynchronous Communication

Synchronous
- One makes a connection to another, sends a request and waits for a reply
  - E.g. a user sends a request for a web page and then waits for a reply

Asynchronous
- Idea is that an application may need to notify another that an event has occurred, but does not need to wait for a response
- We will use this model for the next assignment
  - Sender and receiver of the message do not need to interact with the message queue at the same time
  - Messages are placed onto the queue and are stored until the recipient retrieves them

mqueue files
- Since mqueue are stored as files you can see the message queues in the filesystem
- But you have to mount the mqueue filesystem
  - `mkdir /dev/mqueue`
  - `mount -t mqueue none /dev/mqueue`
- Queues can be cleaned up by simply deleting the appropriate file in that directory
  - Note that you can only delete queues that you have created
  - Also the OS only lets you create a very limited number of queues
    - So make sure you close and unlink all queues before you quit the application