CS 3502
Operating Systems

Inter Process Communication

Kun Suo

Computer Science, Kennesaw State University

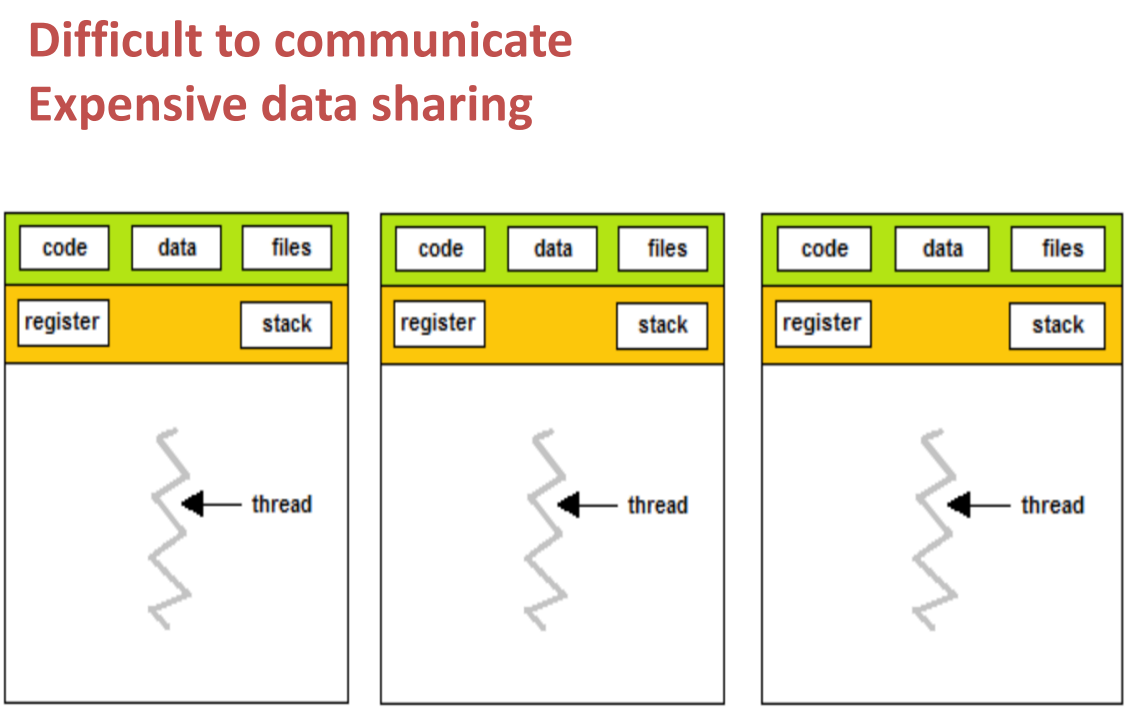
https://kevinsuo.github.io/

Outline

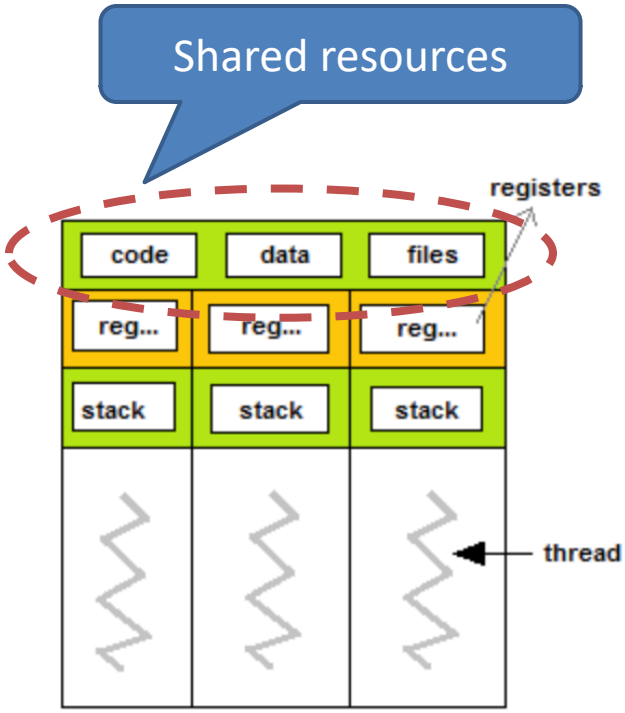
- Interprocess communication
- Main types of IPC
  - Pipe
  - Shared memory
  - Signal
  - Semaphore
  - Message queue
  - Socket

Why not multiple processes ?

Occupy more memory,
complex switching
low CPU utilization
Difficult to communicate
Expensive data sharing



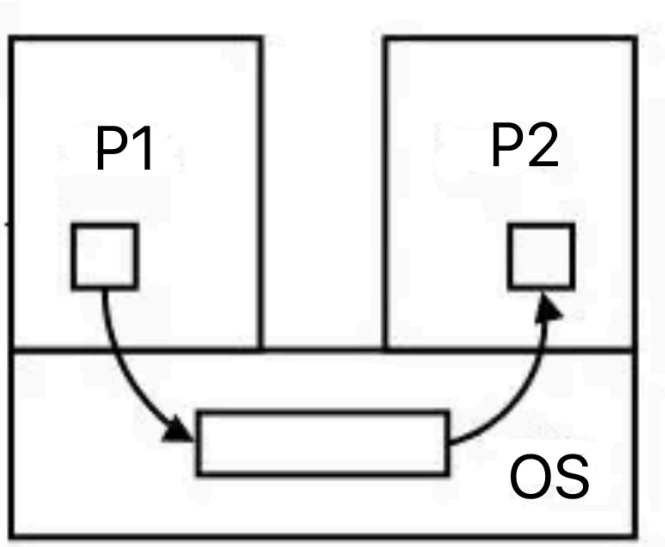
Three processes



Three threads

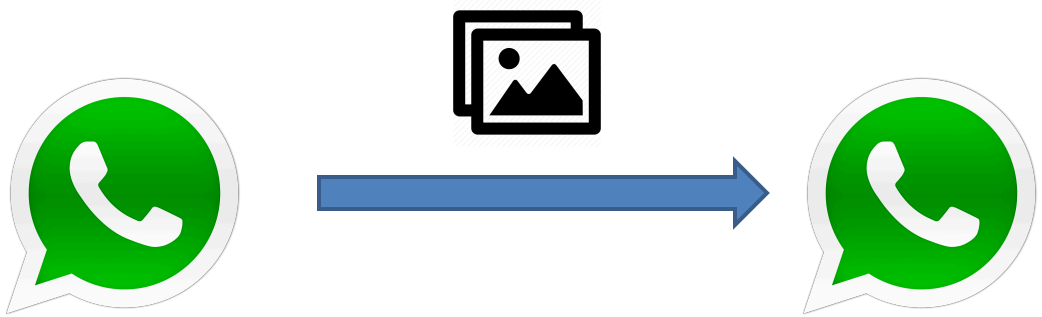
What is inter process communication?

- Different processes have their **own address spaces**
- Variables and data in one process **cannot be accessed** by another process
- Inter process communication: the mechanisms an **OS provides** to allow the processes to communicate and share data



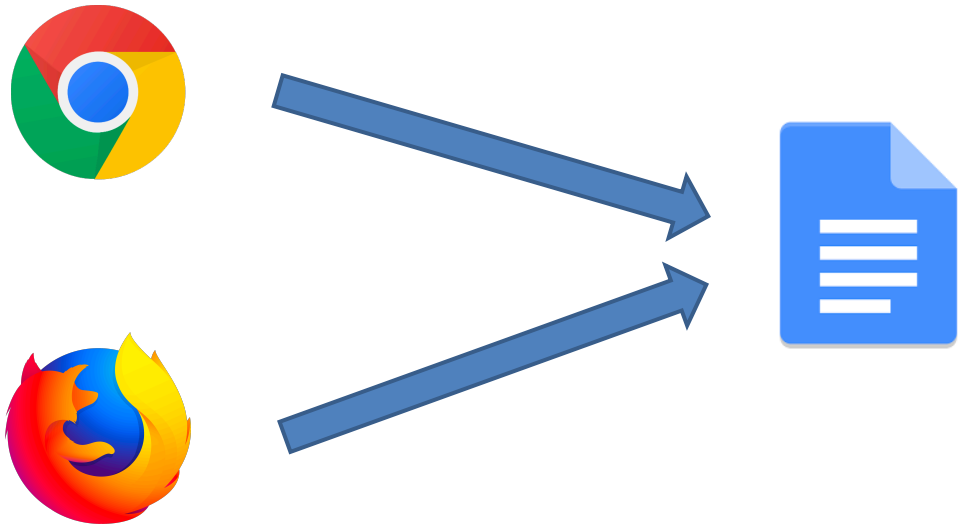
Why we need inter process communication?

- Data transmission
  - A process needs to send its data to another process
- Example:



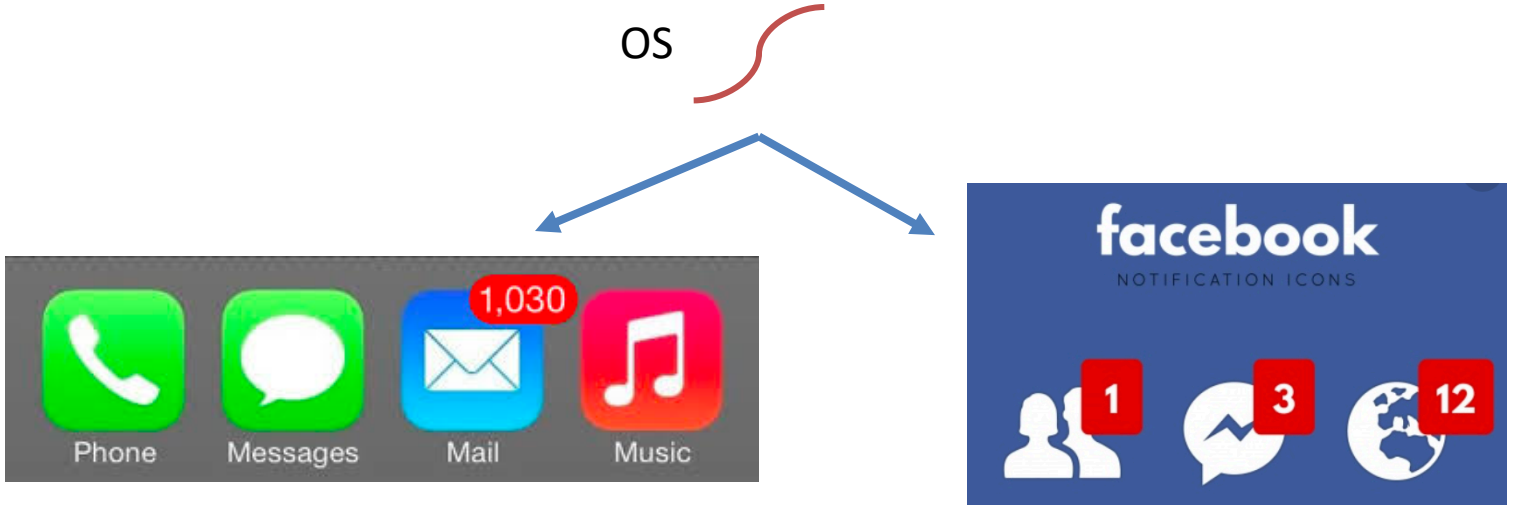
Why we need inter process communication?

- Data sharing
  - Multiple processes manipulate the shared data
- Example



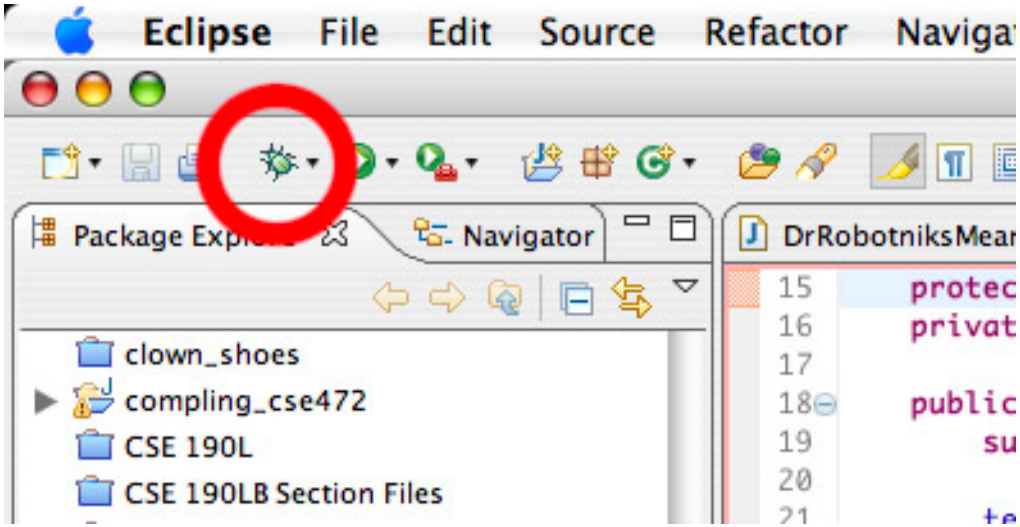
Why we need inter process communication?

- Notification
  - A process sends a message to another or a group of processes to notify it (they) that something has happened
- Example

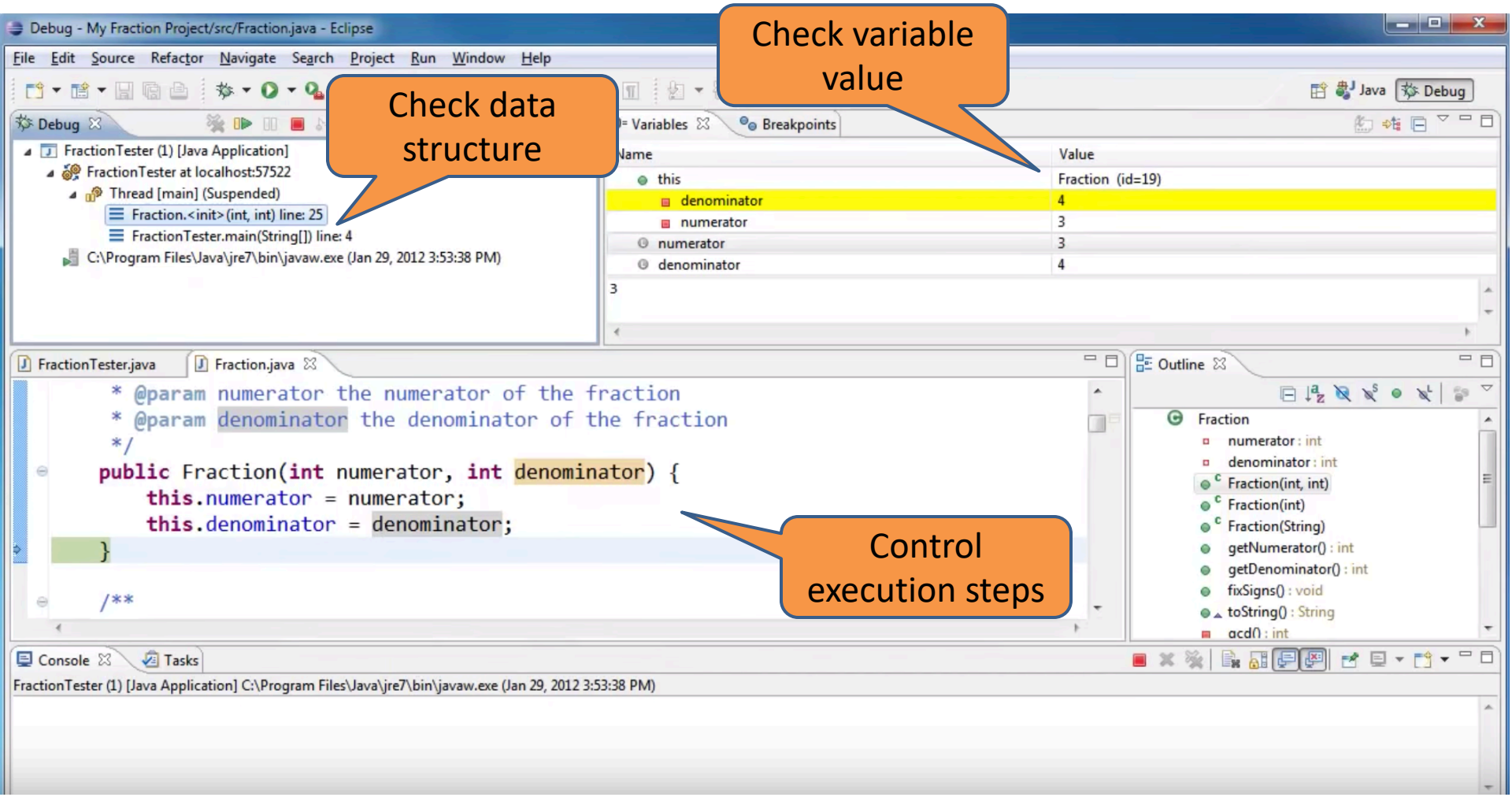


Why we need inter process communication?

- Process control
  - Some processes need full control over the execution of another process (e.g., Debug process)
- Example



Process control in Eclipse debug

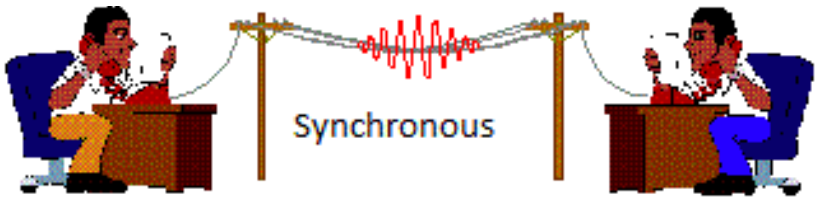


Why we need inter process communication?

- Data transmission
  - A process needs to send its data to another process
- Data sharing
  - Multiple processes manipulate the shared data
- Notification
  - A process sends a message to another or a group of processes to notify it (they) that something has happened
- Process control
  - Some processes need full control over the execution of another process (e.g., Debug process)

Synchronous / Asynchronous communication

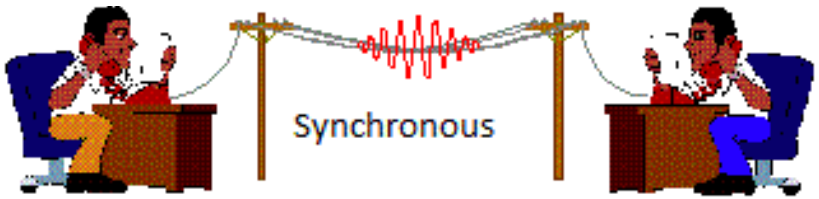
- Synchronous communication (Blocked)



Blocking send: sender process will wait after sending message until the receiver process received

Synchronous / Asynchronous communication

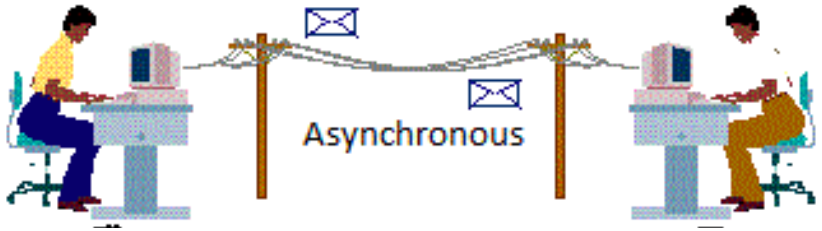
- Synchronous communication (Blocked)



Blocking receive: receiver process will wait after requesting a message until it successfully received a message

Synchronous / Asynchronous communication

- Asynchronous communication (non-Blocked)



Non-Blocking send: sender process **can do other operations** after sending messages

Non-Blocking receive: receiver process **can do other operations** after requesting a message

Synchronous / Asynchronous communication

- Synchronous communication (Blocked)


...

- Asynchronous communication (non-Blocked)

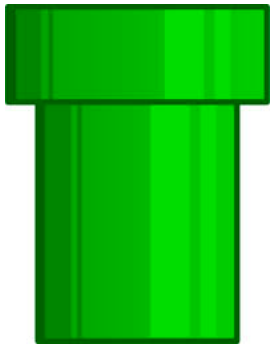


...

Outline

- Interprocess communication
- Main types of IPC
  - Pipe
  - Shared memory
  - Signal
  - Semaphore
  - Message queue
  - Socket

Pipe

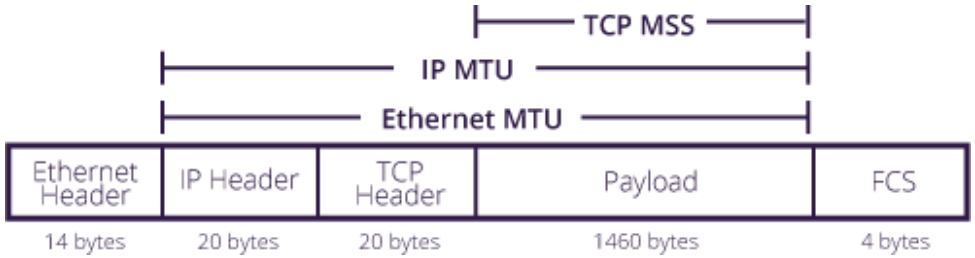


- Pipe: send the output of one command/program/process to another command/program/process
- Write to one end, read from another
- Limitation:
  - Pipes are single directional (one way)
  - Can only be used between processes that are related (e.g., parent and child)
  - Size is limited in pipe

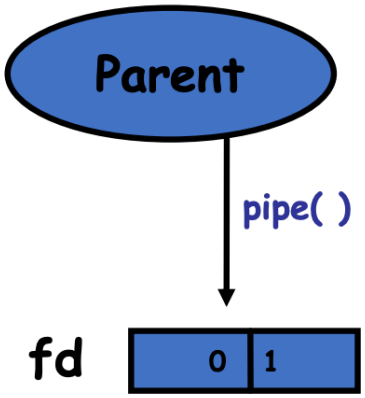
Pipe provides a "byte-stream" abstraction

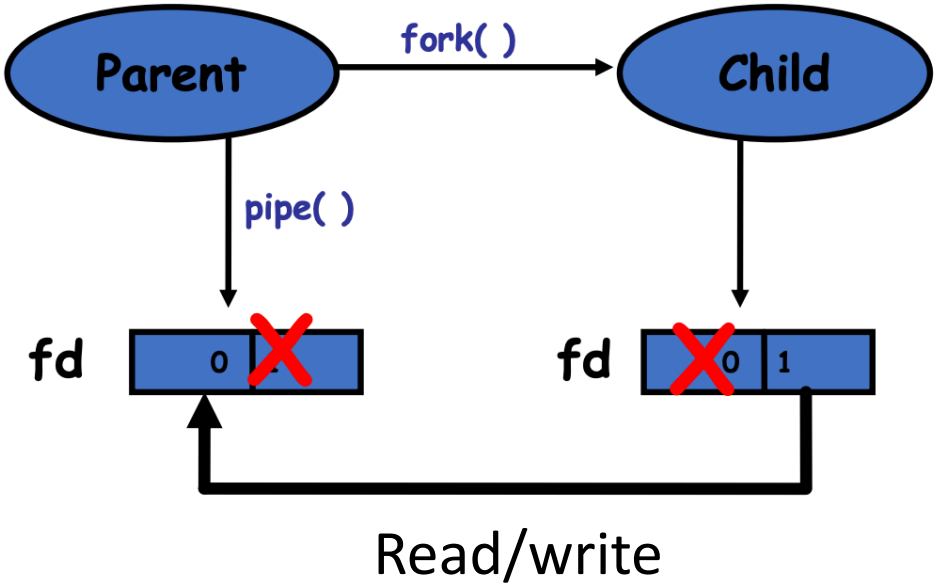
- You can read and write at arbitrary byte boundaries.
  - E.g. Byte lengths sequence written
    - 10, 10, 10, 10
  - byte lengths sequence read
    - 5, 15, 15, 5

- As opposed to message abstraction, which provides explicit message boundaries.
  - E.g. network packets



Pipe example





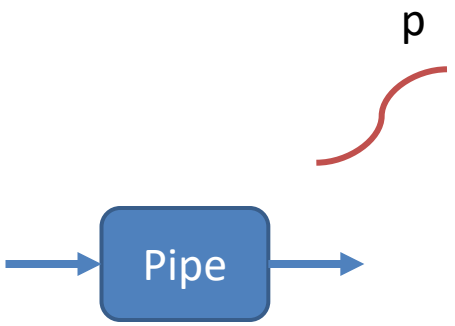
Read from fd[0]

Write to fd[1]

Pipe example

https://github.com/kevinsuo/CS3502/blob/master/pipe.c

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

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

        /* create a pipe */
        if (pipe(pfds) == -1) {
                perror("pipe");
                exit(1);
        }

        /* fork a child */
        if ( (pid = fork()) < 0) {
                perror("fork");
                exit(1);
        }
```



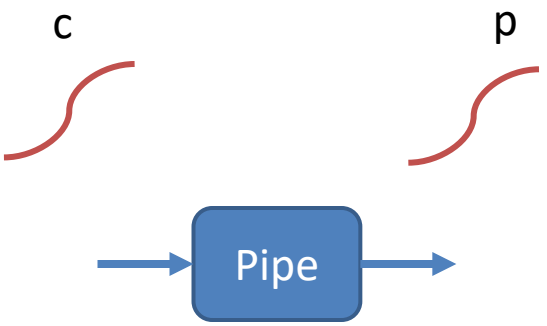
System call No.22 pipe

https://filippo.io/linux-syscall-table/

Pipe example

https://github.com/kevinsuo/CS3502/blob/master/pipe.c

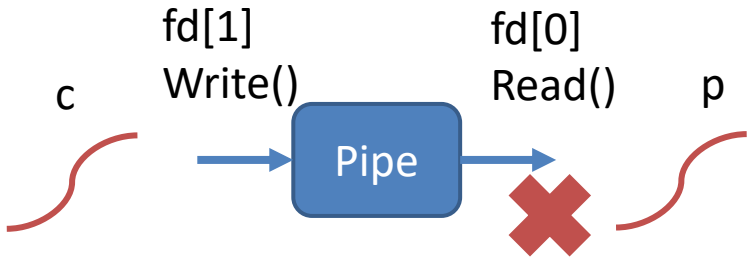
```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

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

        /* create a pipe */
        if (pipe(pfds) == -1) {
                perror("pipe");
                exit(1);
        }

        /* fork a child */
        if ( (pid = fork()) < 0) {
                perror("fork");
                exit(1);
        }
```



Pipe example

https://github.com/kevinsuo/CS3502/blob/master/pipe.c

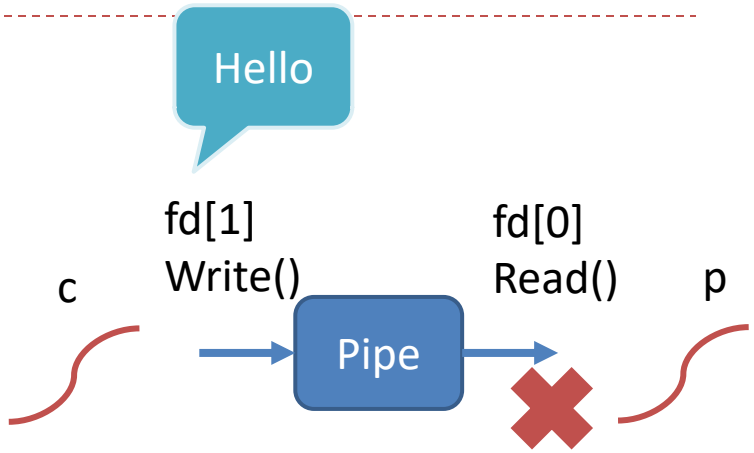
```
    if ( pid == 0 ) {
            printf("Child: writing to the pipe\n");

            /* close the read end */
            close(pfds[0]);

            /* write message to parent  through the write end */
            if(write(pfds[1], "Hello", 6) <= 0) {
                    perror("Child");
                    exit(1);
            }

            printf("Child: exiting\n");

            exit(0);
    } else {
            sleep(1);

            printf("Parent: reading from pipe\n");
            /* close the write end */
            close(pfds[1]);

            /* read message from child through the read end */
            if( read(pfds[0], buf, 6) <= 0 ) {
                    perror("Parent");
                    exit(1);
            }

            printf("Parent: read \"%s\"\n", buf);

            /* wait for child to complete */
            wait(NULL);
    }
}
```



Pipe example

https://github.com/kevinsuo/CS3502/blob/master/pipe.c

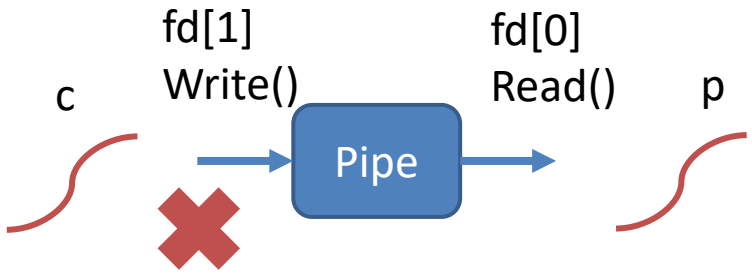
```
    if ( pid == 0 ) {
            printf("Child: writing to the pipe\n");

            /* close the read end */
            close(pfds[0]);

            /* write message to parent  through the write end */
            if(write(pfds[1], "Hello", 6) <= 0) {
                    perror("Child");
                    exit(1);
            }

            printf("Child: exiting\n");

            exit(0);
    } else {
            sleep(1);

            printf("Parent: reading from pipe\n");
            /* close the write end */
            close(pfds[1]);

            /* read message from child through the read end */
            if( read(pfds[0], buf, 6) <= 0 ) {
                    perror("Parent");
                    exit(1);
            }

            printf("Parent: read \"%s\"\n", buf);

            /* wait for child to complete */
            wait(NULL);
    }
}
```



Pipe example

https://github.com/kevinsuo/CS3502/blob/master/pipe.c

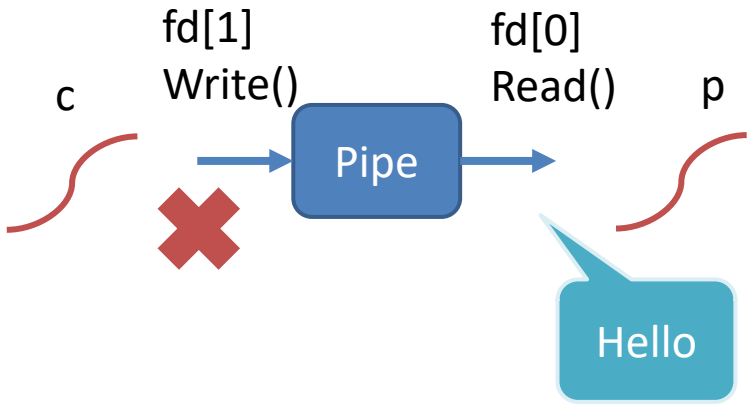
```
    if ( pid == 0 ) {
            printf("Child: writing to the pipe\n");

            /* close the read end */
            close(pfds[0]);

            /* write message to parent  through the write end */
            if(write(pfds[1], "Hello", 6) <= 0) {
                    perror("Child");
                    exit(1);
            }

            printf("Child: exiting\n");

            exit(0);
    } else {
            sleep(1);

            printf("Parent: reading from pipe\n");
            /* close the write end */
            close(pfds[1]);

            /* read message from child through the read end */
            if( read(pfds[0], buf, 6) <= 0 ) {
                    perror("Parent");
                    exit(1);
            }

            printf("Parent: read \"%s\"\n", buf);

            /* wait for child to complete */
            wait(NULL);
    }
}
```



Pipe example

https://github.com/kevinsuo/CS3502/blob/master/pipe.c

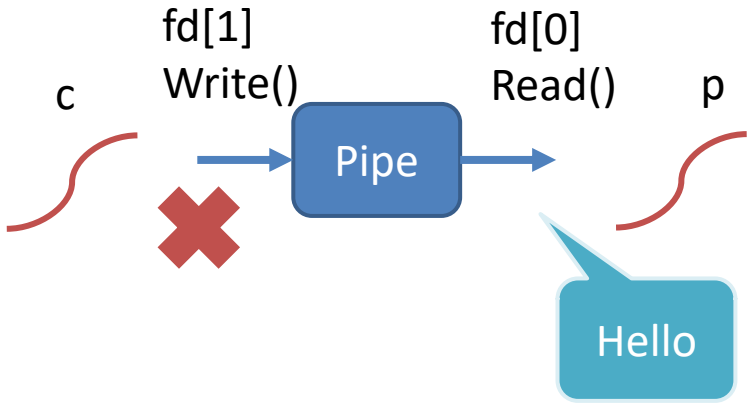
```
    if ( pid == 0 ) {
            printf("Child: writing to the pipe\n");

            /* close the read end */
            close(pfds[0]);

            /* write message to parent  through the write end */
            if(write(pfds[1], "Hello", 6) <= 0) {
                    perror("Child");
                    exit(1);
            }

            printf("Child: exiting\n");

            exit(0);
    } else {
            sleep(1);

            printf("Parent: reading from pipe\n");
            /* close the write end */
            close(pfds[1]);

            /* read message from child through the read end */
            if( read(pfds[0], buf, 6) <= 0 ) {
                    perror("Parent");
                    exit(1);
            }

            printf("Parent: read \"%s\"\n", buf);

            /* wait for child to complete */
            wait(NULL);
    }
}
```



Pipe example

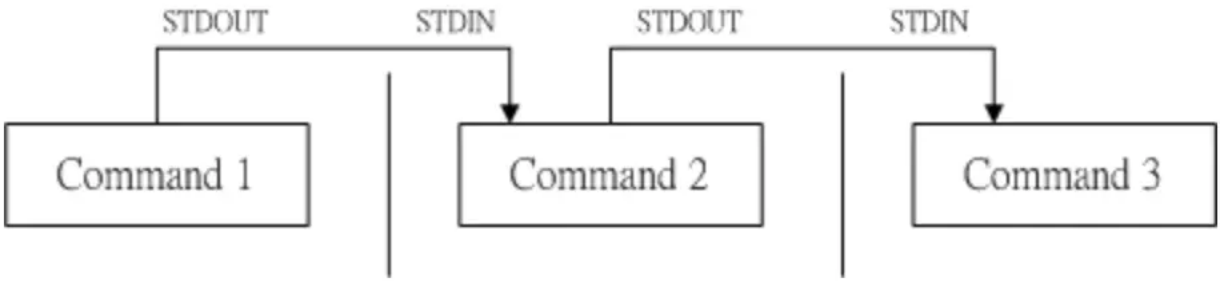
https://github.com/kevinsuo/CS3502/blob/master/pipe.c

```
        if ( pid == 0 ) {
                printf("Child: writing to the pipe\n");

                /* close the read end */
                close(pfds[0]);

                /* write message to parent  through the write end */
                if(write(pfds[1], "Hello", 6) <= 0) {
                        perror("Child");
                        exit(1);
                }

                printf("Child: exiting\n");

                exit(0);
        } else {
                sleep(1);

                printf("Parent: reading from pipe\n");
                /* close the write end */
                close(pfds[1]);

                /* read message from child through the read end */
                if( read(pfds[0], buf, 6) <= 0 ) {
                        perror("Parent");
                        exit(1);
                }

                printf("Parent: read \"%s\"\n", buf);

                /* wait for child to complete */
                wait(NULL);
        }
}
```




```
pi@raspberrypi ~/Downloads> ./pipe.o
Child: writing to the pipe
Child: exiting
Parent: reading from pipe
Parent: read "Hello"
```

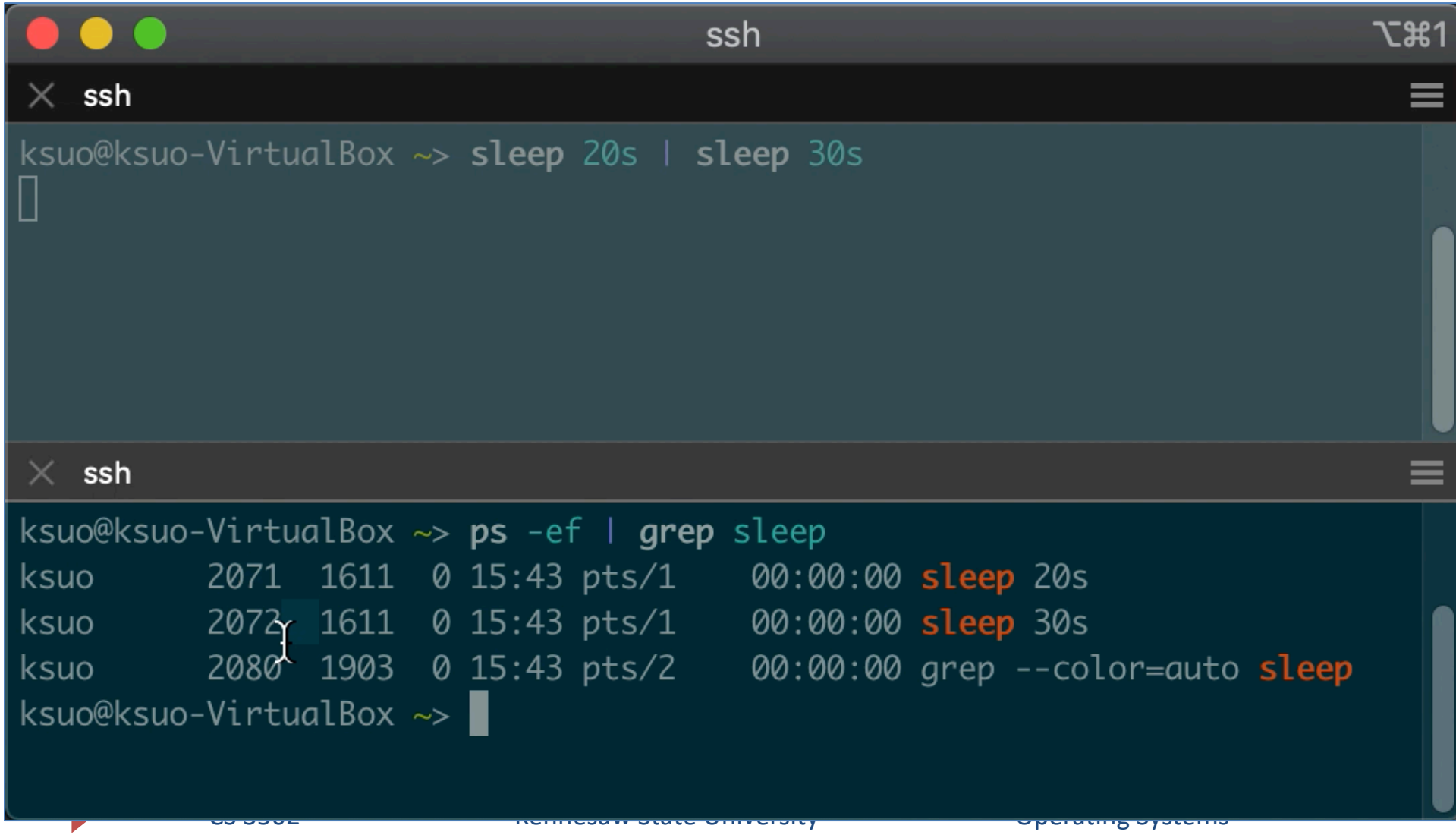
Pipe in Shell



- $ command1 | command2 | command3;

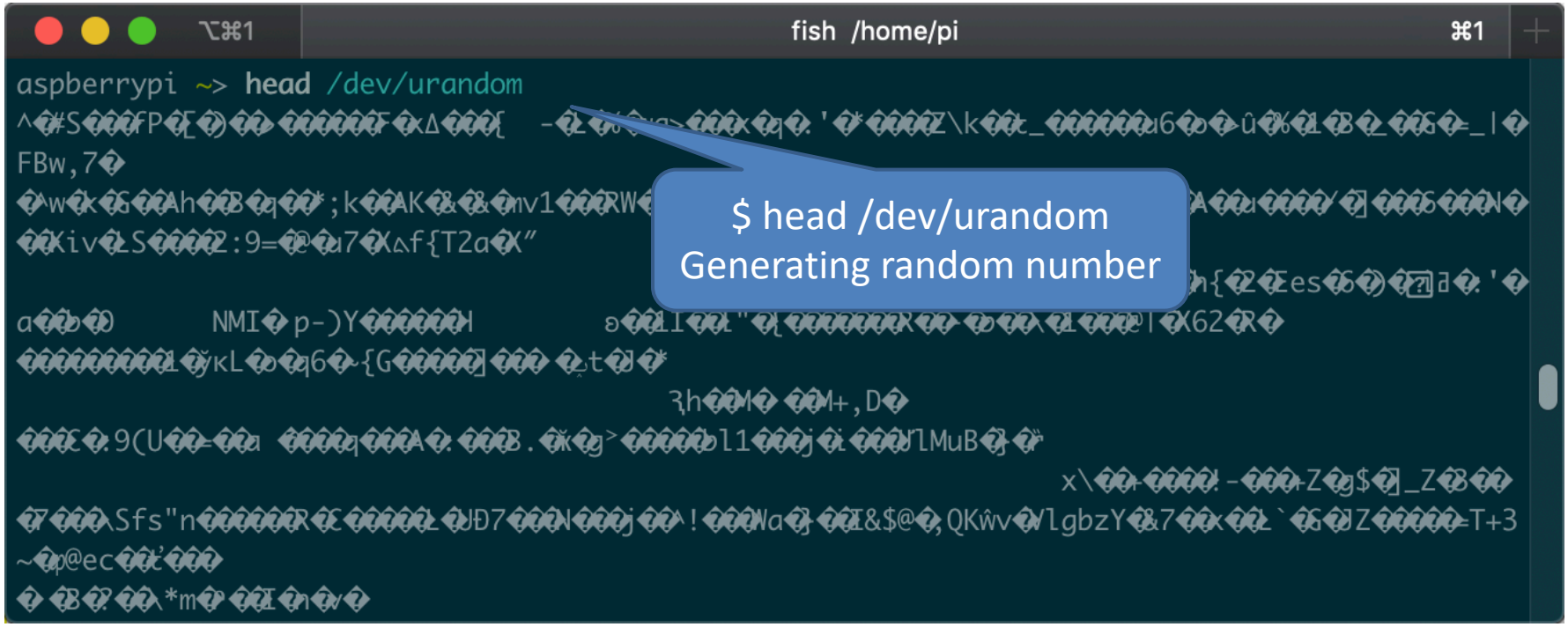
- The output of command 1 is as the input of command 2
- The output of command 2 is as the input of command 3
- The output of command 3 is printed on the screen

Pipe in Shell



Pipe in Shell example

- Generating a length of 20 password containing all letters or numbers
  - $ head /dev/urandom | tr -dc A-Za-z0-9 | head -c 20



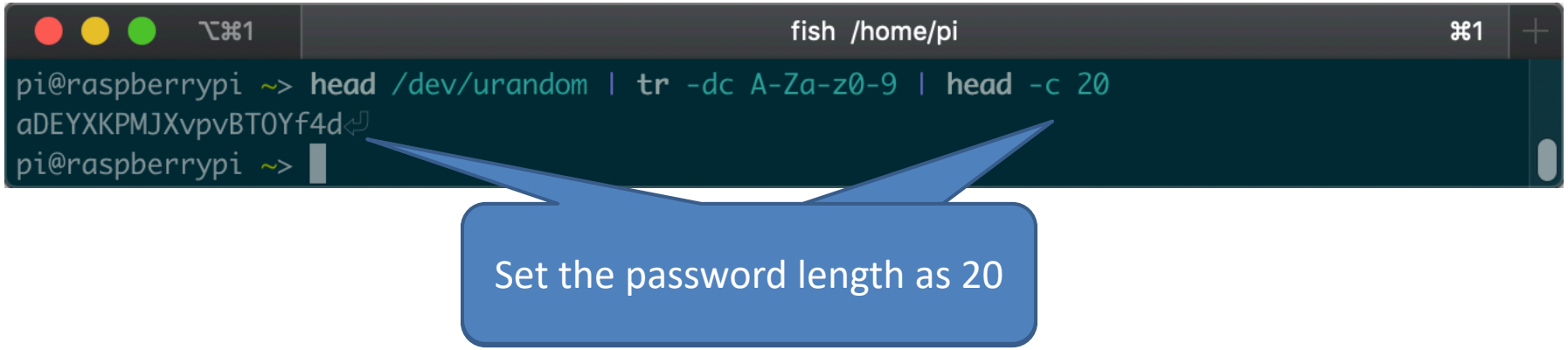
Pipe in Shell example

- Generating a length of 20 password containing all letters or numbers
  - $ head /dev/urandom | tr -dc A-Za-z0-9 | head -c 20

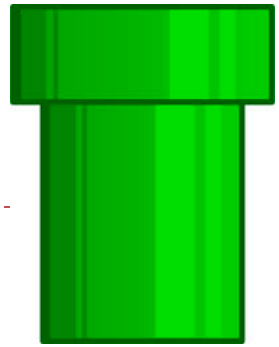
Make the password containing all letters or numbers

Pipe in Shell example

- Generating a length of 20 password containing all letters or numbers
  - $ head /dev/urandom | tr -dc A-Za-z0-9 | head -c 20



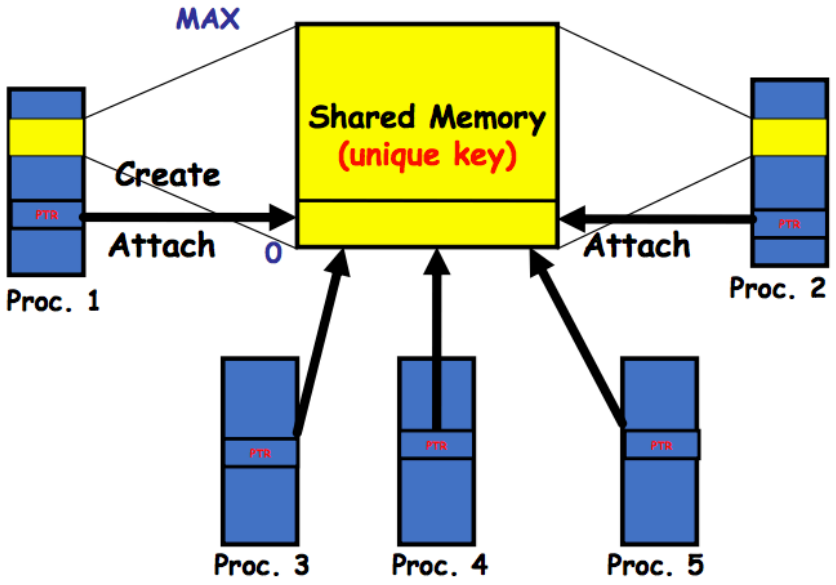
Pipe summary



- Pipe: simplest IPC, send the output of one command/program/process to another command/program/process
- Write to one end, read from another
- Limitation:
  - Pipes are single directional (one way)
  - Can only be used between processes that are related (e.g., parent and child)
  - Size is limited in pipe

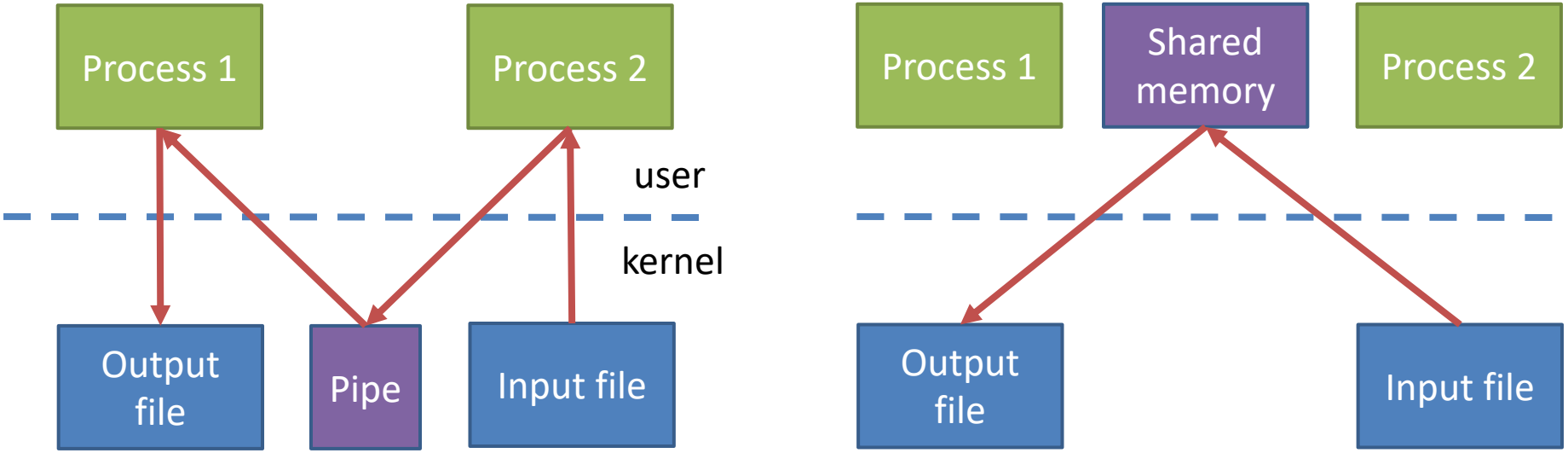
Shared memory

- Shared memory maps a piece of memory that can be accessed by other processes
- Shared memory is created by one process, but multiple processes can access it.
- Shared memory is one of the fastest ways of IPC



Shared memory vs. Pipe

- Pipe needs **four** data copies between kernel space and user space
- Shared memory needs only **two** data copies between kernel and user space



Shared memory

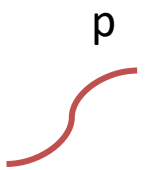
- Man pages : https://filippo.io/linux-syscall-table/
  - Shmget: create a shared memory 29
  - Shmat: attach a shared memory 30
  - Shmdt: detach a shared memory 67
  - Shmctl: delete a shared memory 31

```
Example:
    key_t key;
    int shmid;
    char *data;
    key = ftok("<somefile>", 'A');
    shmid = shmget(key, 1024, 0644);
    data = shmat(shmid, (void *)0, 0);
    // read or write something to data here.
    shmdt(data);
```

Shared memory example

https://github.com/kevinsuo/CS3502/blob/master/shared_memory.c

```c
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
        int protection = PROT_READ | PROT_WRITE;
        int visibility = MAP_ANONYMOUS | MAP_SHARED;
        return mmap(NULL, size, protection, visibility, -1, 0);
}

int main() {
        char* parent_message = "1111";  // parent process will write this message
        char* child_message = "2222";    // child process will then write this one

        void* shmem = create_shared_memory(128);
        memcpy(shmem, parent_message, sizeof(parent_message));  //copy parent to shared memory

        int pid = fork();

        if (pid == 0) {   //child process
                printf("Child read: %s\n", shmem);
                memcpy(shmem, child_message, sizeof(child_message));   //copy child to shared memory
                printf("Child wrote: %s\n", shmem);

        } else {    //parent process
                printf("Parent read: %s\n", shmem);
                sleep(5);
                printf("After 5s, parent read: %s\n", shmem);
        }
}
```

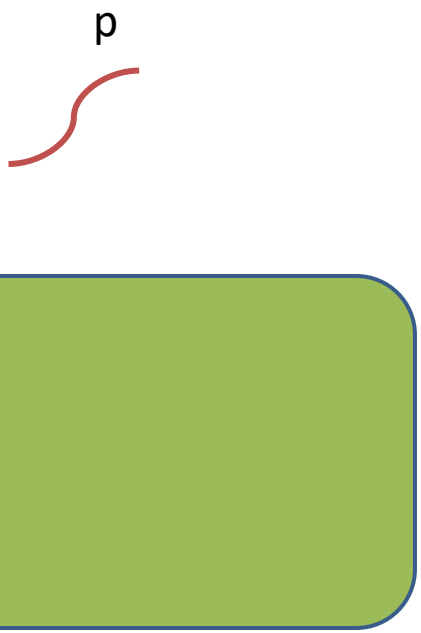


1111 2222

Shared memory example

https://github.com/kevinsuo/CS3502/blob/master/shared_memory.c

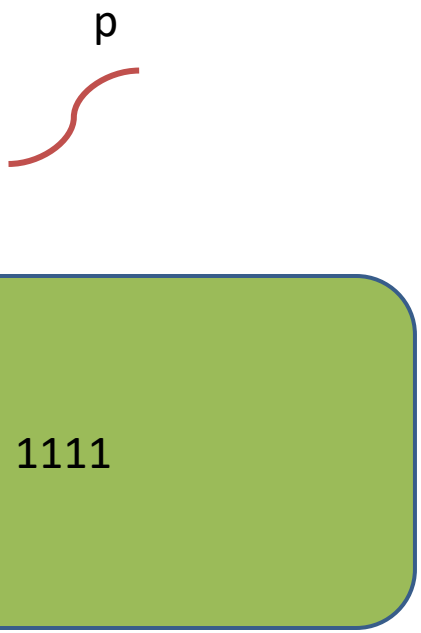
```c
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
        int protection = PROT_READ | PROT_WRITE;
        int visibility = MAP_ANONYMOUS | MAP_SHARED;
        return mmap(NULL, size, protection, visibility, -1, 0);
}

int main() {
        char* parent_message = "1111";  // parent process will write this message
        char* child_message = "2222";    // child process will then write this one

        void* shmem = create_shared_memory(128);
        memcpy(shmem, parent_message, sizeof(parent_message));  //copy parent to shared memory

        int pid = fork();

        if (pid == 0) {   //child process
                printf("Child read: %s\n", shmem);
                memcpy(shmem, child_message, sizeof(child_message));   //copy child to shared memory
                printf("Child wrote: %s\n", shmem);

        } else {    //parent process
                printf("Parent read: %s\n", shmem);
                sleep(5);
                printf("After 5s, parent read: %s\n", shmem);
        }
}
```



Shared memory example

https://github.com/kevinsuo/CS3502/blob/master/shared_memory.c

```c
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
        int protection = PROT_READ | PROT_WRITE;
        int visibility = MAP_ANONYMOUS | MAP_SHARED;
        return mmap(NULL, size, protection, visibility, -1, 0);
}

int main() {
        char* parent_message = "1111";  // parent process will write this message
        char* child_message = "2222";    // child process will then write this one

        void* shmem = create_shared_memory(128);
        memcpy(shmem, parent_message, sizeof(parent_message));  //copy parent to shared memory

        int pid = fork();

        if (pid == 0) {   //child process
                printf("Child read: %s\n", shmem);
                memcpy(shmem, child_message, sizeof(child_message));   //copy child to shared memory
                printf("Child wrote: %s\n", shmem);

        } else {    //parent process
                printf("Parent read: %s\n", shmem);
                sleep(5);
                printf("After 5s, parent read: %s\n", shmem);
        }
}
```

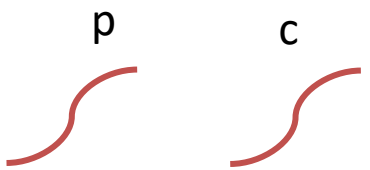


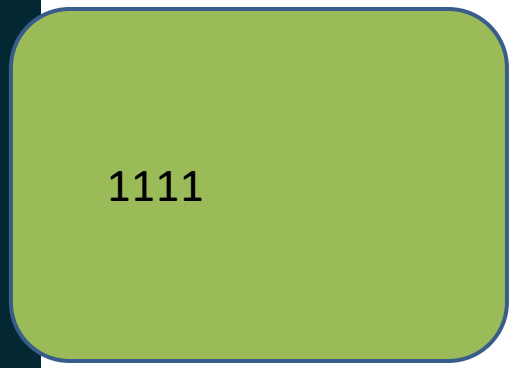

```
pi@raspberrypi ~/Downloads> ./shared_memory.o
Parent read: 1111
Child read: 1111
Child wrote: 2222
After 5s, parent read: 2222
```

Shared memory example

https://github.com/kevinsuo/CS3502/blob/master/shared_memory.c

```c
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
        int protection = PROT_READ | PROT_WRITE;
        int visibility = MAP_ANONYMOUS | MAP_SHARED;
        return mmap(NULL, size, protection, visibility, -1, 0);
}

int main() {
        char* parent_message = "1111";  // parent process will write this message
        char* child_message = "2222";    // child process will then write this one

        void* shmem = create_shared_memory(128);
        memcpy(shmem, parent_message, sizeof(parent_message));  //copy parent to shared memory

        int pid = fork();

        if (pid == 0) {   //child process
                printf("Child read: %s\n", shmem);
                memcpy(shmem, child_message, sizeof(child_message));   //copy child to shared memory
                printf("Child wrote: %s\n", shmem);

        } else {    //parent process
                printf("Parent read: %s\n", shmem);
                sleep(5);
                printf("After 5s, parent read: %s\n", shmem);
        }
}
```



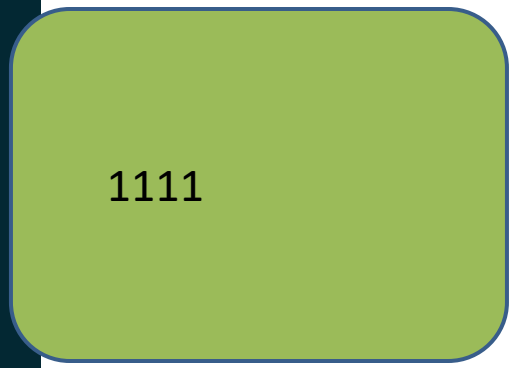


1111 2222

Shared memory example

https://github.com/kevinsuo/CS3502/blob/master/shared_memory.c

```c
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
        int protection = PROT_READ | PROT_WRITE;
        int visibility = MAP_ANONYMOUS | MAP_SHARED;
        return mmap(NULL, size, protection, visibility, -1, 0);
}

int main() {
        char* parent_message = "1111";  // parent process will write this message
        char* child_message = "2222";    // child process will then write this one

        void* shmem = create_shared_memory(128);
        memcpy(shmem, parent_message, sizeof(parent_message));  //copy parent to shared memory

        int pid = fork();

        if (pid == 0) {   //child process
                printf("Child read: %s\n", shmem);
                memcpy(shmem, child_message, sizeof(child_message));   //copy child to shared memory
                printf("Child wrote: %s\n", shmem);

        } else {    //parent process
                printf("Parent read: %s\n", shmem);
                sleep(5);
                printf("After 5s, parent read: %s\n", shmem);
        }
}
```



1111 2222

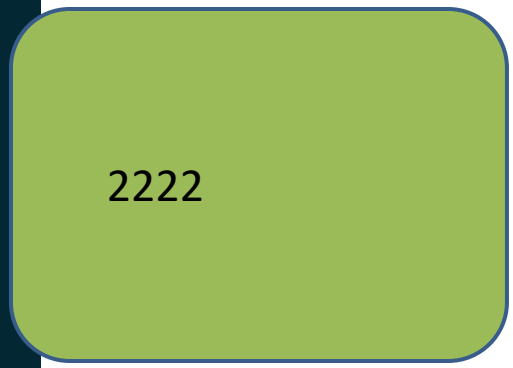
```
pi@raspberrypi ~/Downloads> ./shared_memory.o
Parent read: 1111
Child read: 1111
Child wrote: 2222
After 5s, parent read: 2222
```

Shared memory example

https://github.com/kevinsuo/CS3502/blob/master/shared_memory.c

```c
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
        int protection = PROT_READ | PROT_WRITE;
        int visibility = MAP_ANONYMOUS | MAP_SHARED;
        return mmap(NULL, size, protection, visibility, -1, 0);
}

int main() {
        char* parent_message = "1111";  // parent process will write this message
        char* child_message = "2222";    // child process will then write this one

        void* shmem = create_shared_memory(128);
        memcpy(shmem, parent_message, sizeof(parent_message));  //copy parent to shared memory

        int pid = fork();

        if (pid == 0) {   //child process
                printf("Child read: %s\n", shmem);
                memcpy(shmem, child_message, sizeof(child_message));
                printf("Child wrote: %s\n", shmem);

        } else {    //parent process
                printf("Parent read: %s\n", shmem);
                sleep(5);
                printf("After 5s, parent read: %s\n", shmem);
        }
}
```

child to shared memory



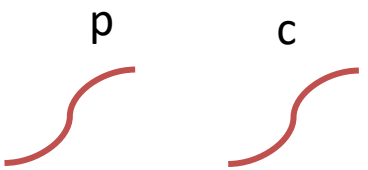
1111 2222

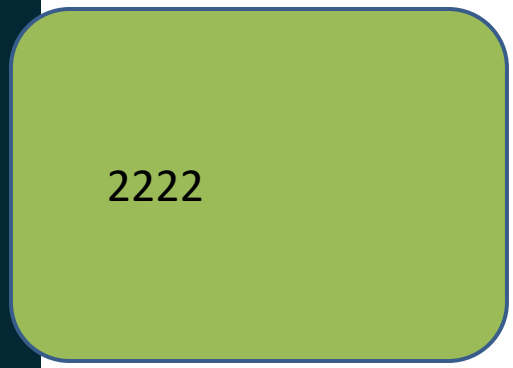
```
pi@raspberrypi ~/Downloads> ./shared_memory.o
Parent read: 1111
Child read: 1111
Child wrote: 2222
After 5s, parent read: 2222
```

Shared memory example

https://github.com/kevinsuo/CS3502/blob/master/shared_memory.c

```c
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
        int protection = PROT_READ | PROT_WRITE;
        int visibility = MAP_ANONYMOUS | MAP_SHARED;
        return mmap(NULL, size, protection, visibility, -1, 0);
}

int main() {
        char* parent_message = "1111";  // parent process will write this message
        char* child_message = "2222";    // child process will then write this one

        void* shmem = create_shared_memory(128);
        memcpy(shmem, parent_message, sizeof(parent_message));  //copy parent to shared memory

        int pid = fork();

        if (pid == 0) {   //child process
                printf("Child read: %s\n", shmem);
                memcpy(shmem, child_message, sizeof(child_message));   //copy child to shared memory
                printf("Child wrote: %s\n", shmem);

        } else {    //parent process
                printf("Parent read: %s\n", shmem);
                sleep(5);
                printf("After 5s, parent read: %s\n", shmem);
        }
}
```





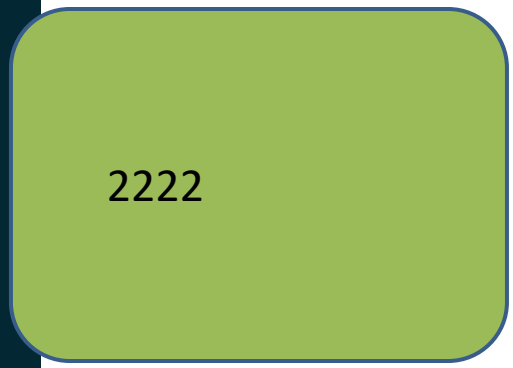
1111 2222

```
pi@raspberrypi ~/Downloads> ./shared_memory.o
Parent read: 1111
Child read: 1111
Child wrote: 2222
After 5s, parent read: 2222
```

Shared memory example

https://github.com/kevinsuo/CS3502/blob/master/shared_memory.c

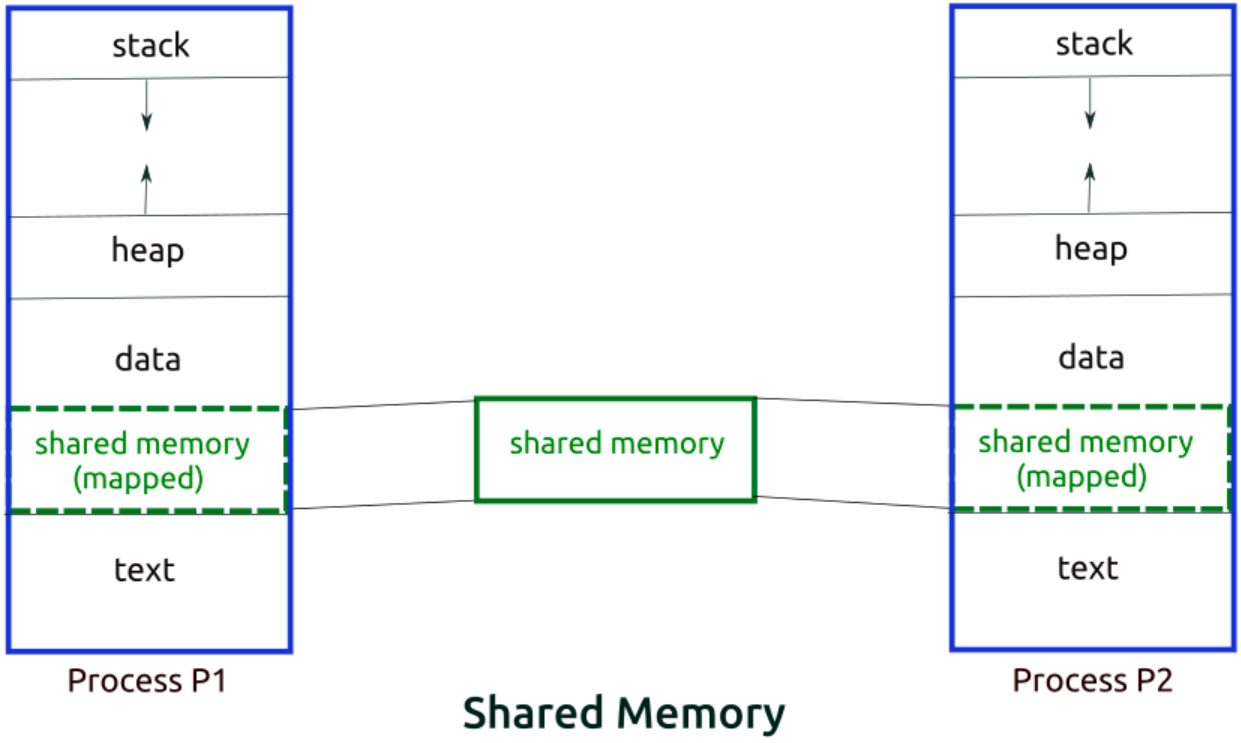
```c
#include <string.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>

void* create_shared_memory(size_t size) {
        int protection = PROT_READ | PROT_WRITE;
        int visibility = MAP_ANONYMOUS | MAP_SHARED;
        return mmap(NULL, size, protection, visibility, -1, 0);
}

int main() {
        char* parent_message = "1111";  // parent process will write this message
        char* child_message = "2222";    // child process will then write this one

        void* shmem = create_shared_memory(128);
        memcpy(shmem, parent_message, sizeof(parent_message));  //copy parent to shared memory

        int pid = fork();

        if (pid == 0) {   //child process
                printf("Child read: %s\n", shmem);
                memcpy(shmem, child_message, sizeof(child_message));   //copy child to shared memory
                printf("Child wrote: %s\n", shmem);

        } else {    //parent process
                printf("Parent read: %s\n", shmem);
                sleep(5);
                printf("After 5s, parent read: %s\n", shmem);
        }
}
```



1111 2222

```
pi@raspberrypi ~/Downloads> ./shared_memory.o
Parent read: 1111
Child read: 1111
Child wrote: 2222
After 5s, parent read: 2222
```

Shared memory



Advantages	Efficient, no need to make any copies Simple functions and APIs Not limited between specific processes like pipe
Possible problems	Linux does not provide a synchronization mechanism. If more than one process writes to the memory at the same time, it will be messed up. Other techniques are needed to match the shared memory mechanism.

Signals

- A signal is an asynchronous notification sent to a process/thread within the same process in order to notify it of an event that occurred

Signal	dispositions
Term	Default action is to **terminate** the process
Ign	Default action is to **ignore** the signal
Core	Default action is to **terminate** the process and **dump core** (used for debug)
Stop	Default action is to **stop** the process
Cont	Default action is to **continue** the process if it is currently stopped.

Feature of Signals

- Signals can be sent to a process **at any time** without knowing the state of the process.

- If the process is not currently running, the signal is **saved** by the kernel. When the process executes on the CPU again, the kernel will **pass** the signals to the process.

- If a signal is blocked by the process, the signal passing will be **delayed** until the signal is allowed again

Standard signals in Linux

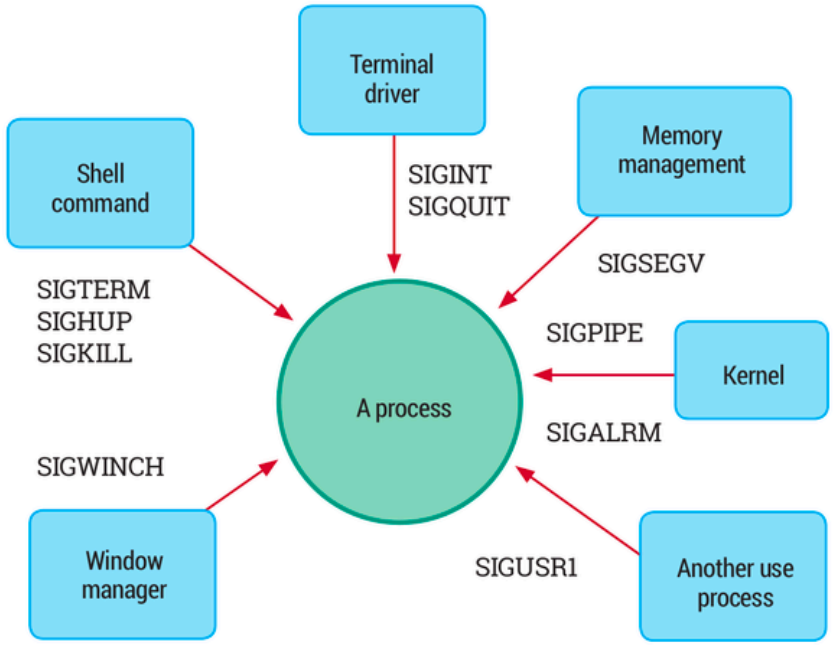
Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from abort(3)
SIGALRM	P1990	Term	Timer signal from alarm(2)
SIGBUS	P2001	Core	Bus error (bad memory access)
SIGCHLD	P1990	Ign	Child stopped or terminated
SIGCLD	-	Ign	A synonym for SIGCHLD
SIGCONT	P1990	Cont	Continue if stopped
SIGEMT	-	Term	Emulator trap
SIGFPE	P1990	Core	Floating-point exception
SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	P1990	Core	Illegal Instruction
SIGINFO	-		A synonym for SIGPWR
SIGINT	P1990	Term	Interrupt from keyboard
SIGIO	-	Term	I/O now possible (4.2BSD)
SIGIOT	-	Core	IOT trap. A synonym for SIGABRT
SIGKILL	P1990	Term	Kill signal
SIGLOST	-	Term	File lock lost (unused)
SIGPIPE	P1990	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGPOLL	P2001	Term	Pollable event (Sys V).

Standard signals in Linux (cont.)

Signal	Standard	Action	Comment
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	-	Term	Power failure (System V)
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTKFLT	-	Term	Stack fault on coprocessor (unused)
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call (SVr4);
SIGTERM	P1990	Term	Termination signal
SIGTRAP	P2001	Core	Trace/breakpoint trap
SIGTTIN	P1990	Stop	Terminal input for background process
SIGTTOU	P1990	Stop	Terminal output for background process
SIGUNUSED	-	Core	Synonymous with SIGSYS
SIGURG	P2001	Ign	Urgent condition on socket (4.2BSD)
SIGUSR1	P1990	Term	User-defined signal 1
SIGUSR2	P1990	Term	User-defined signal 2
SIGVTALRM	P2001	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	P2001	Core	CPU time limit exceeded (4.2BSD);
SIGXFSZ	P2001	Core	File size limit exceeded (4.2BSD);
SIGWINCH	-	Ign	Window resize signal (4.3BSD, Sun)

Signal sources

- Normally, the signals come from two sources:
- Hardware:
  - User press Ctrl+C on keyboard
  - Hardware exception like storage is not enough
- Software:
  - Signals for process activities like stop, kill, etc.
  - Software exception like visit unsafe address, zero division operation, data overflow, etc.



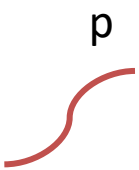
Handling Signals

- Signals can be caught – i.e. an action can be associated with them
  - SIGKILL and SIGSTOP cannot be caught.
- Actions to signals can be customized using
  - sigaction(…), which associates a signal handler with the signal.
- Default action for most signals is to terminate the process
  - Except SIGCHLD and SIGURG are ignored by default.
- Unwanted signals can be ignored
  - Except SIGKILL or SIGSTOP

Signal example

https://github.com/kevinsuo/CS3502/blob/master/signal.c

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

void int_handler(int signum)
{
        printf("\nSIGINT signal handler.\n");
        printf("exit.\n");
        exit(-1);
}

int main()
{
        signal(SIGINT, int_handler);
        printf("int_handler set for SIGINT\n");

        while(1)
        {
                printf("go to sleep.\n");
                sleep(60);
        }

        return 0;
}
```



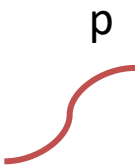
SIGINT: Interrupt from keyboard

```
pi@raspberrypi ~/Downloads> ./signal.o
int_handler set for SIGINT
go to sleep.
^C
SIGINT signal handler.
exit.
```

Signal example

https://github.com/kevinsuo/CS3502/blob/master/signal.c

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

void int_handler(int signum)
{
        printf("\nSIGINT signal handler.\n");
        printf("exit.\n");
        exit(-1);
}

int main()
{
        signal(SIGINT, int_handler);
        printf("int_handler set for SIGINT\n");

        while(1)
        {
                printf("go to sleep.\n");
                sleep(60);
        }

        return 0;
}
```



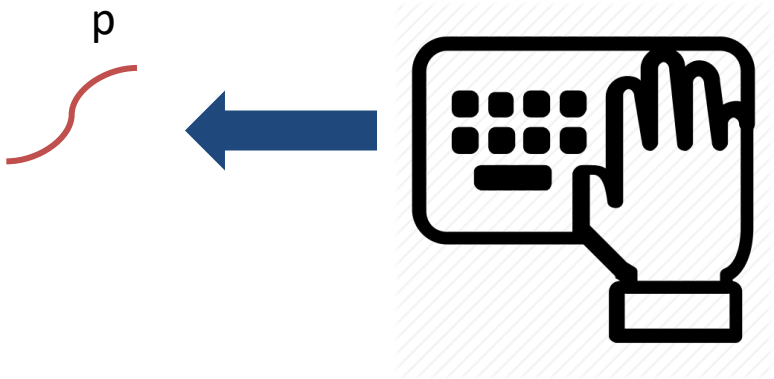
SIGINT: Interrupt from keyboard

```
pi@raspberrypi ~/Downloads> ./signal.o
int_handler set for SIGINT
go to sleep.
^C
SIGINT signal handler.
exit.
```

Signal example

https://github.com/kevinsuo/CS3502/blob/master/signal.c

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

void int_handler(int signum)
{
        printf("\nSIGINT signal handler.\n");
        printf("exit.\n");
        exit(-1);
}

int main()
{
        signal(SIGINT, int_handler);
        printf("int_handler set for SIGINT\n");

        while(1)
        {
                printf("go to sleep.\n");
                sleep(60);
        }

        return 0;
}
```



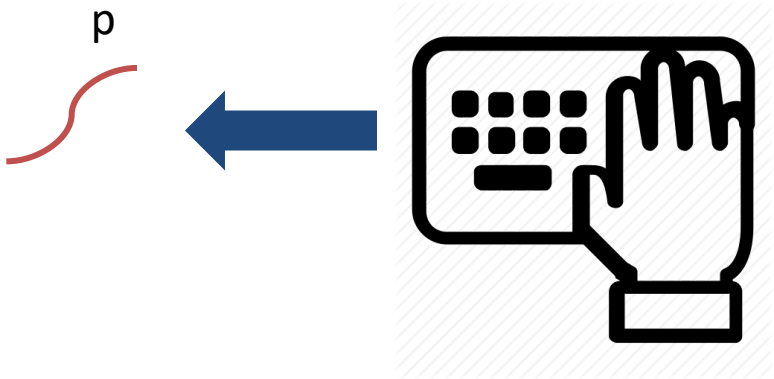
SIGINT: Interrupt from keyboard

```
pi@raspberrypi ~/Downloads> ./signal.o
int_handler set for SIGINT
go to sleep.
^C
SIGINT signal handler.
exit.
```

Signal example

https://github.com/kevinsuo/CS3502/blob/master/signal.c

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

void int_handler(int signum)
{
        printf("\nSIGINT signal handler.\n");
        printf("exit.\n");
        exit(-1);
}

int main()
{
        signal(SIGINT, int_handler);
        printf("int_handler set for SIGINT\n");

        while(1)
        {
                printf("go to sleep.\n");
                sleep(60);
        }

        return 0;
}
```



SIGINT: Interrupt from keyboard

```
pi@raspberrypi ~/Downloads> ./signal.o
int_handler set for SIGINT
go to sleep.
^C
SIGINT signal handler.
exit.
```

Signals summary

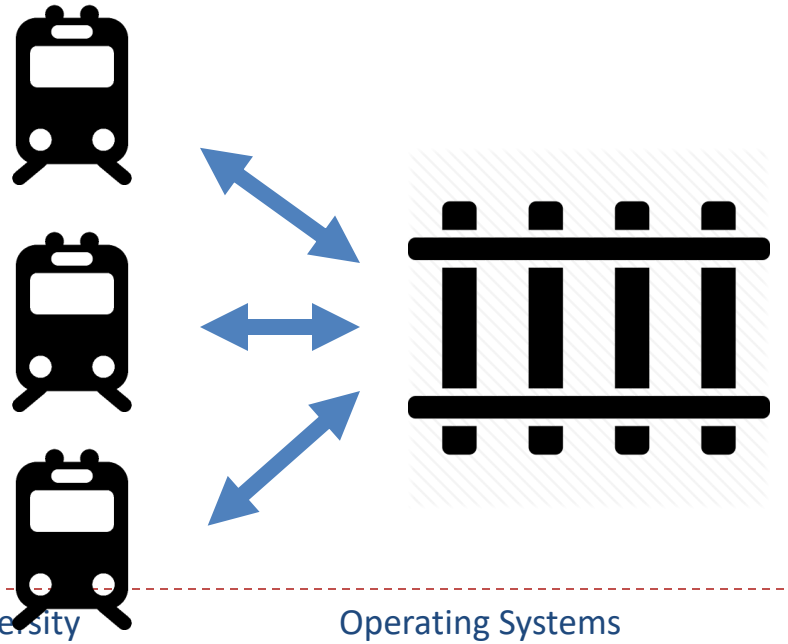
- A signal is an asynchronous notification sent to a process/thread within the same process in order to notify it of an event that occurred
- Signals can be sent to a process at any time, can be saved, can be delayed
- Signals come from two sources: hardware and software
- When signals are caught, an action associated with them will be executed

Semaphore

- A **counter** to control access to shared resources by multiple processes/threads
- A **lock mechanism** to prevent a process from accessing a shared resource
- A means of **synchronization** between processes and between different threads within the same process

Semaphore

- A semaphore “sem” is a special integer on which only two operations can be performed.
  - DOWN(sem)
  - UP(sem)



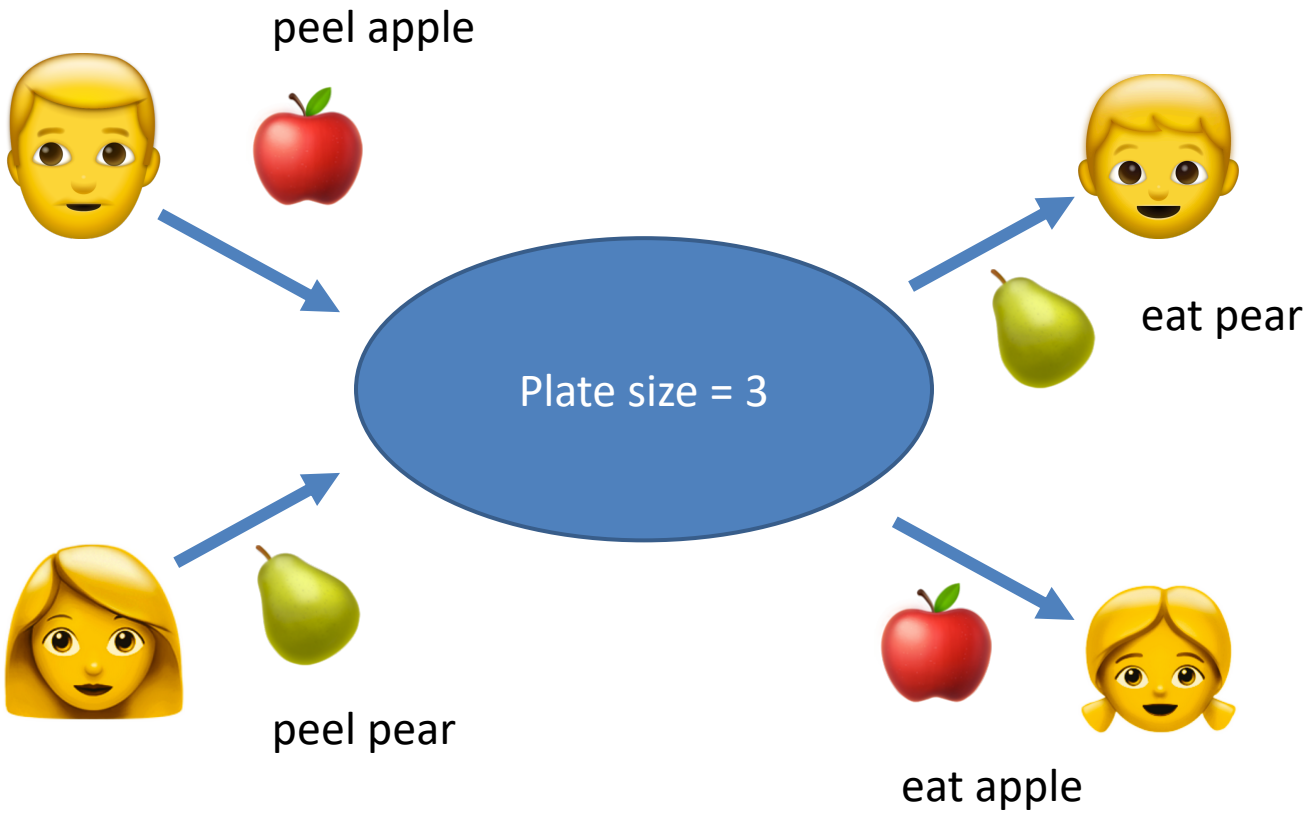
Semaphore

- Down operation (P; request):
  - Checks if a semaphore is > 0, sem--
    - Request one unit resource and one process enters

- Up operation (V; release)
  - sem++
    - Release one unit resource and one process leaves

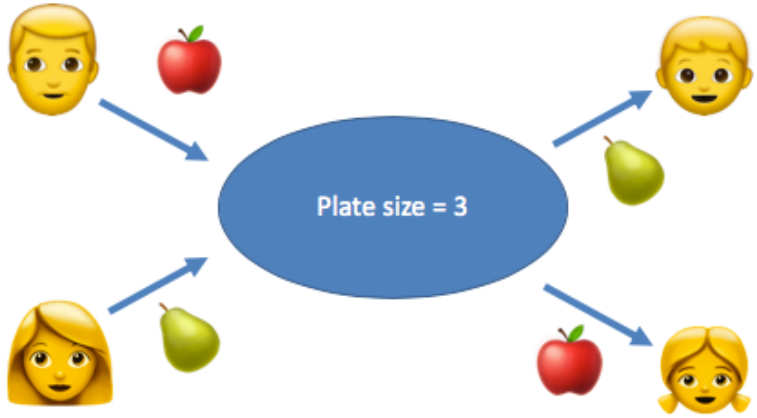
Semaphore example

https://github.com/kevinsuo/CS3502/blob/master/semaphore.c



Semaphore example

https://github.com/kevinsuo/CS3502/blob/master/semaphore.c



- Semaphore:
  - Son: whether there is pear, s1
  - Daughter: whether there is apple, s2
  - Father/Mother: whether there is space, s3

Father thread:
peel apple
P(s3)
put apple
V(s2)

Mother thread:
peel pear
P(s3)
put apple
V(s1)

Son thread:
P(s1)
get pear
V(s3)
eat pear

Daughter thread:
P(s2)
get apple
V(s3)
eat apple

Semaphore summary

- A **counter** to control access to shared resources by multiple processes/threads
- A **lock mechanism** to prevent a process from accessing a shared resource
- A means of **synchronization** between processes and between different threads within the same process
- Syscall 64/65/66: https://filippo.io/linux-syscall-table/

Message queue

- A linked list of messages, stored in the kernel and identified by the message queue identifier

- Message Queuing overcomes the disadvantages of
  - less information (signal),
  - only passing unformatted byte streams (pipe), and limited buffer size

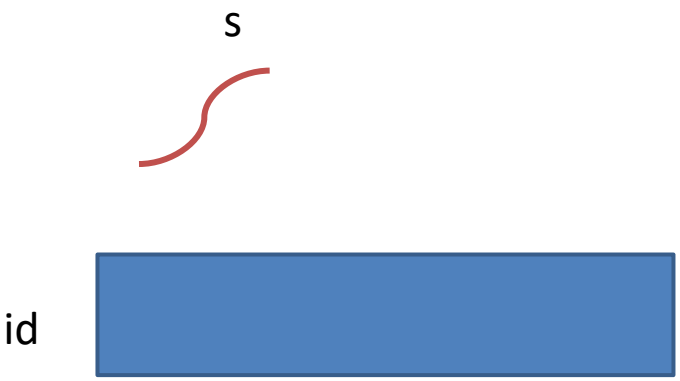
Message queue example

https://github.com/kevinsuo/CS3502/blob/master/msg_queue_sender.c

https://github.com/kevinsuo/CS3502/blob/master/msg_queue_receiver.c

Sender

```c
int main()
{
        int running = 1;
        struct msg_st data;
        char buffer[BUFSIZ];
        int msgid = -1;
        int len;

        /* create a message queue */
        msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
        if(msgid == -1)
        {
                fprintf(stderr, "msgget failed with error: %d\n", errno);
                exit(EXIT_FAILURE);
        }

        while(running)
        {

                printf("Send data : ");
                fgets(buffer, BUFSIZ, stdin);
                data.msg_type = 5;
                strcpy(data.text, buffer);
                len = strlen(data.text);

                /* Send message to msg queue */
                if(msgsnd(msgid, (void*)&data, len-1, 0) == -1)
                {
                        fprintf(stderr, "msgsnd failed\n");
                        exit(EXIT_FAILURE);
                }

                if(strncmp(buffer, "end", 3) == 0)
                        running = 0;
                usleep(100000);
        }
        exit(EXIT_SUCCESS);
}
```



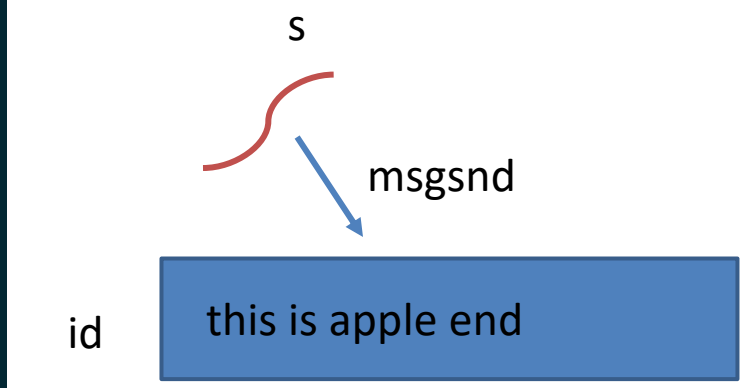
https://github.com/kevinsuo/CS3502/blob/master/msg_queue_sender.c

https://github.com/kevinsuo/CS3502/blob/master/msg_queue_receiver.c

Message queue example

Sender

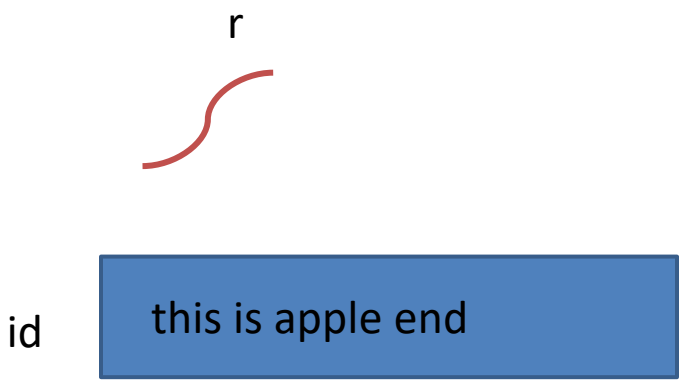
```c
int main()
{
        int running = 1;
        struct msg_st data;
        char buffer[BUFSIZ];
        int msgid = -1;
        int len;

        /* create a message queue */
        msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
        if(msgid == -1)
        {
                fprintf(stderr, "msgget failed with error: %d\n", errno);
                exit(EXIT_FAILURE);
        }

        while(running)
        {

                printf("Send data : ");
                fgets(buffer, BUFSIZ, stdin);
                data.msg_type = 5;
                strcpy(data.text, buffer);
                len = strlen(data.text);

                /* Send message to msg queue */
                if(msgsnd(msgid, (void*)&data, len-1, 0) == -1)
                {
                        fprintf(stderr, "msgsnd failed\n");
                        exit(EXIT_FAILURE);
                }

                if(strncmp(buffer, "end", 3) == 0)
                        running = 0;
                usleep(100000);
        }
        exit(EXIT_SUCCESS);
}
```



Message queue example

https://github.com/kevinsuo/CS3502/blob/master/msg_queue_sender.c

https://github.com/kevinsuo/CS3502/blob/master/msg_queue_receiver.c



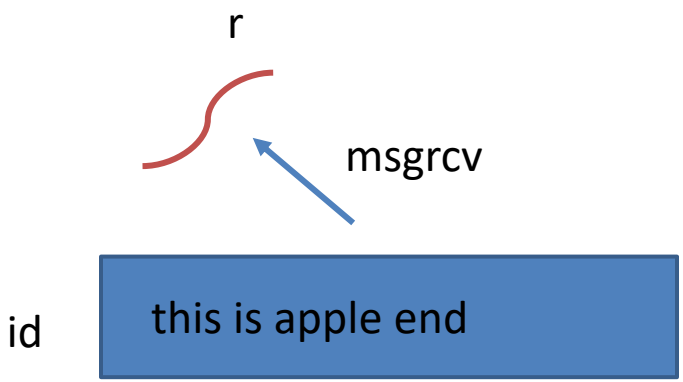
Receiver

```c
int main()
{
        int running = 1;
        int msgid = -1;
        int len = 0;
        long int msgtype = 5;
        struct msg_st data;

        /* create a message queue */
        msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
        if(-1 == msgid )
        {
                fprintf(stderr, "msgget failed with error: %d\n", errno);
                exit(EXIT_FAILURE);
        }

        while(running)
        {
                memset(&data.text, 0, 128);
                /* receive message from msg queue */
                len = msgrcv(msgid, (void*)&data, 128, msgtype, 0);
                if(-1 == len)
                {
                        fprintf(stderr, "msgrcv failed with errno: %d\n", errno);
                        exit(EXIT_FAILURE);
                }
                printf("Receive: %s\n",data.text);

                if(0 == strncmp(data.text, "end", 3))
                        running = 0;
        }

        /* remove message queue */
        if(-1 == msgctl(msgid, IPC_RMID, 0))
        {
                fprintf(stderr, "msgctl(IPC_RMID) failed\n");
                exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
}
```

https://github.com/kevinsuo/CS3502/blob/master/msg_queue_sender.c

https://github.com/kevinsuo/CS3502/blob/master/msg_queue_receiver.c

Message queue example



Receiver

```c
int main()
{
        int running = 1;
        int msgid = -1;
        int len = 0;
        long int msgtype = 5;
        struct msg_st data;

        /* create a message queue */
        msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
        if(-1 == msgid )
        {
                fprintf(stderr, "msgget failed with error: %d\n", errno);
                exit(EXIT_FAILURE);
        }

        while(running)
        {
                memset(&data.text, 0, 128);
                /* receive message from msg queue */
                len = msgrcv(msgid, (void*)&data, 128, msgtype, 0);
                if(-1 == len)
                {
                        fprintf(stderr, "msgrcv failed with errno: %d\n", errno);
                        exit(EXIT_FAILURE);
                }
                printf("Receive: %s\n",data.text);

                if(0 == strncmp(data.text, "end", 3))
                        running = 0;
        }

        /* remove message queue */
        if(-1 == msgctl(msgid, IPC_RMID, 0))
        {
                fprintf(stderr, "msgctl(IPC_RMID) failed\n");
                exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
}
```

Message queue example

https://github.com/kevinsuo/CS3502/blob/master/msg_queue_sender.c

https://github.com/kevinsuo/CS3502/blob/master/msg_queue_receiver.c

id

this is apple end

this is apple end

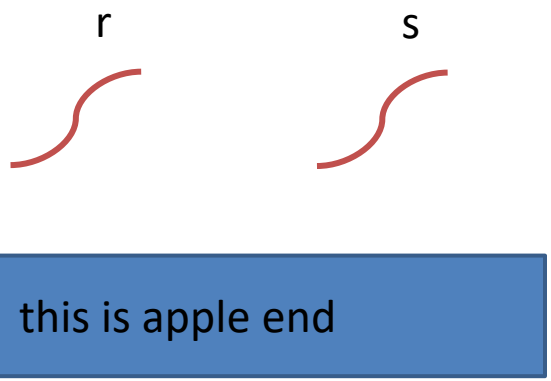
Receiver

```c
int main()
{
        int running = 1;
        int msgid = -1;
        int len = 0;
        long int msgtype = 5;
        struct msg_st data;

        /* create a message queue */
        msgid = msgget((key_t)1234, 0666 | IPC_CREAT);
        if(-1 == msgid )
        {
                fprintf(stderr, "msgget failed with error: %d\n", errno);
                exit(EXIT_FAILURE);
        }

        while(running)
        {
                memset(&data.text, 0, 128);
                /* receive message from msg queue */
                len = msgrcv(msgid, (void*)&data, 128, msgtype, 0);
                if(-1 == len)
                {
                        fprintf(stderr, "msgrcv failed with errno: %d\n", errno);
                        exit(EXIT_FAILURE);
                }
                printf("Receive: %s\n",data.text);

                if(0 == strncmp(data.text, "end", 3))
                        running = 0;
        }

        /* remove message queue */
        if(-1 == msgctl(msgid, IPC_RMID, 0))
        {
                fprintf(stderr, "msgctl(IPC_RMID) failed\n");
                exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
}
```

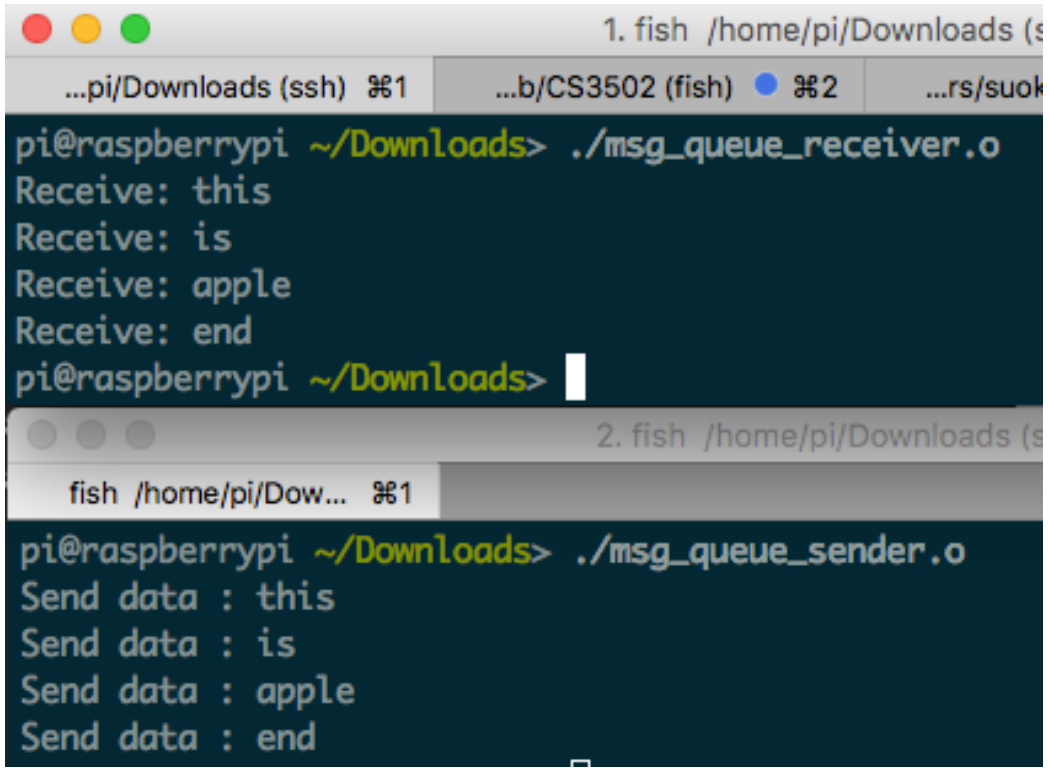
Message queue example

https://github.com/kevinsuo/CS3502/blob/master/msg_queue_sender.c

https://github.com/kevinsuo/CS3502/blob/master/msg_queue_receiver.c



this is apple end



Message queue vs. Pipe

- Message Queuing can be used between **different processes**, regardless of whether these processes are related
- Message: not only data, but also **rich context** including priority, message arrival notifications, etc.
- Message queue is **independent** from sender/receiver and does not open/close in pipe

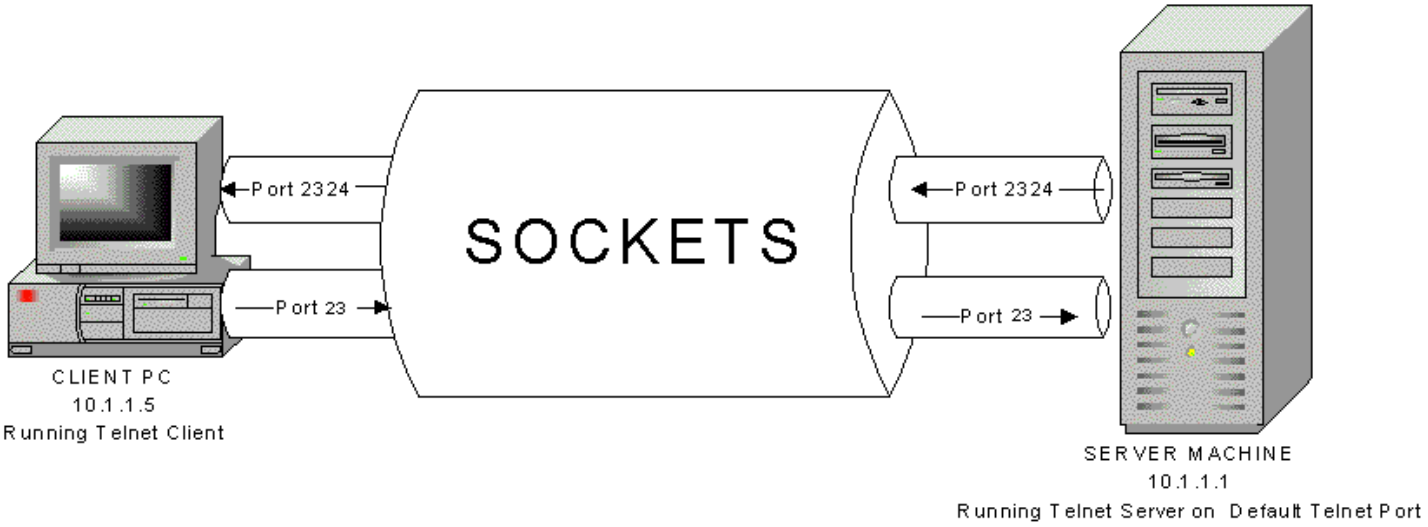
Message queue summary

- A linked list of messages, stored in the kernel and identified by the message queue identifier
- Message Queuing can be used between different processes, containing rich context, and is independent from sender/receiver
- Message queue system call 68-71: https://filippo.io/linux-syscall-table/

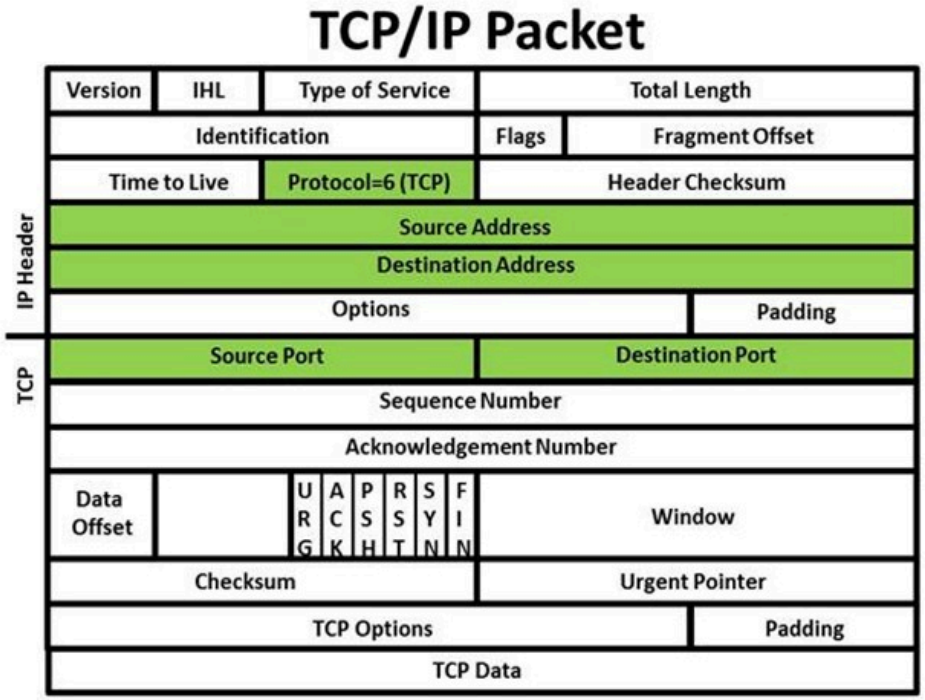
68	msgget	sys_msgget	ipc/msg.c
69	msgsnd	sys_msgsnd	ipc/msg.c
70	msgrcv	sys_msgrcv	ipc/msg.c
71	msgctl	sys_msgctl	ipc/msg.c

Socket

- An inter process communication mechanism. Unlike other IPC mechanisms, it can be used for process communication **between different machines**



Socket

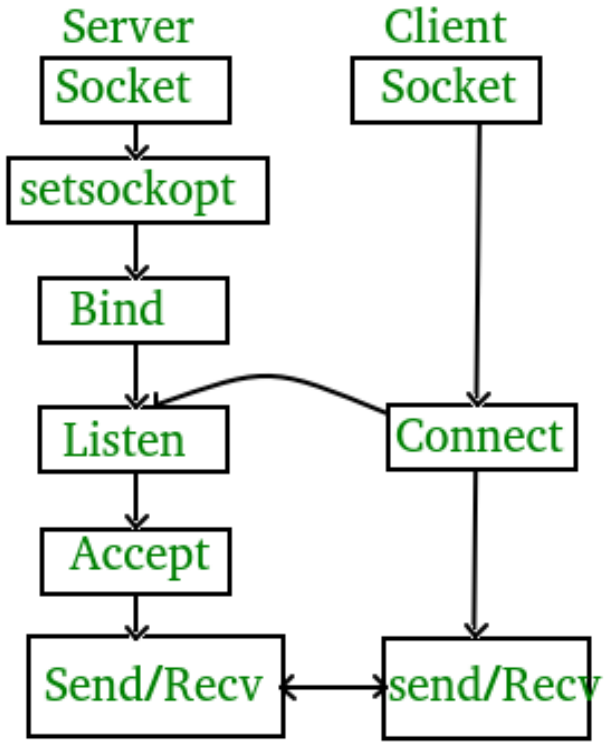


- A Socket is an end point of communication between two systems on a network. To be precise, a socket is a combination of ***IP address*** and ***port*** on one system.

- Each connection between two processes running at different systems can be uniquely identified through their ***4-tuple***.

State diagram for server and client communication

1. **Socket**: Socket creation
2. **Setsockopt**: manipulating options for the socket referred by the file descriptor sockfd
3. **Bind**: binds the socket to the address and port number specified in addr
4. **Listen**: waits for the client to approach the server to make a connection
5. **Accept:** connection is established between client and server, and they are ready to transfer data
6. **Send/Recv**: data transferring



Socket example

https://github.com/kevinsuo/CS3502/blob/master/client.c

https://github.com/kevinsuo/CS3502/blob/master/server.c

```c
#include <unistd.h>
#include <stdio.h>
#include <sys/socket.h>
#include <stdlib.h>
#include <netinet/in.h>
#include <string.h>
#define PORT 8080


int main(int argc, char const *argv[])
{
    int server_fd, new_socket, valread;
    struct sockaddr_in address;
    int opt = 1;
    int addrlen = sizeof(address);
    char buffer[1024] = {0};
    char *hello = "Hello from server";

    // Creating socket file descriptor
    if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
    {
        perror("socket failed");
        exit(EXIT_FAILURE);
    }

    // Forcefully attaching socket to the port 8080
    if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT,
                                                  &opt, sizeof(opt)))
    {
        perror("setsockopt");
        exit(EXIT_FAILURE);
    }
    address.sin_family = AF_INET;
    address.sin_addr.s_addr = INADDR_ANY;
    address.sin_port = htons( PORT );

    // Forcefully attaching socket to the port 8080
    if (bind(server_fd, (struct sockaddr *)&address,
                                 sizeof(address))<0)
    {
        perror("bind failed");
        exit(EXIT_FAILURE);
    }
    if (listen(server_fd, 3) < 0)
    {
        perror("listen");
        exit(EXIT_FAILURE);
    }
    if ((new_socket = accept(server_fd, (struct sockaddr *)&address,
                        (socklen_t*)&addrlen))<0)
    {
        perror("accept");
        exit(EXIT_FAILURE);
    }
    valread = read( new_socket , buffer, 1024);
    printf("%s\n",buffer );
    send(new_socket , hello , strlen(hello) , 0 );
    printf("Hello message sent\n");
    return 0;
}
```

```c
#include <stdio.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <string.h>
#define PORT 8080

int main(int argc, char const *argv[])
{
    int sock = 0, valread;
    struct sockaddr_in serv_addr;
    char *hello = "Hello from client";
    char buffer[1024] = {0};
    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        printf("\n Socket creation error \n");
        return -1;
    }

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(PORT);

    // Convert IPv4 and IPv6 addresses from text to binary form
    if(inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr)<=0)
    {
        printf("\nInvalid address/ Address not supported \n");
        return -1;
    }

    if (connect(sock, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        printf("\nConnection Failed \n");
        return -1;
    }
    send(sock , hello , strlen(hello) , 0 );
    printf("Hello message sent\n");
    valread = read( sock , buffer, 1024);
    printf("%s\n",buffer );
    return 0;
}
```

```
ksuo@ksuo-VirtualBox ~> ./server.o
Hello from client
Hello message sent
```

```
fish  /home/ksuo
ksuo@ksuo-VirtualBox ~> ./client.o
Hello message sent
Hello from server
```

Interprocess communication comparison

IPC	Mechanism	Normal Use Case
Pipe	Write to one end, read from another. Unidirectional. Between related processes.	Simple data sharing, such as producers and consumers
Signal	Asynchronous notification to notify it of an event that occurred	Unable to transfer data. Signal is mainly used for process management
Semaphore	A counter to control access to shared resources by multiple processes/threads	Used for lock and synchronization
Shared memory	Shared memory maps a piece of memory that can be accessed by other processes	Large amount of data communication and transferring
Message Queue	A linked list of messages, stored in the kernel and identified by the message queue identifier	Communication and data sharing
Socket	General inter process communication mechanism	Networking, across hosts communication

Process vs thread communication

- Process communication: Data exchange
  - Pipe
  - Semaphore
  - Signal
  - Shared memory
  - Message queue
  - Socket
- Thread communication: synchronization
  - Lock
  - Semaphore
  - Signal

Talk about these in future classes

Conclusion

- Interprocess communication
- Main types of IPC
  - Pipe
  - Shared memory
  - Signal
  - Semaphore
  - Message queue
  - Socket