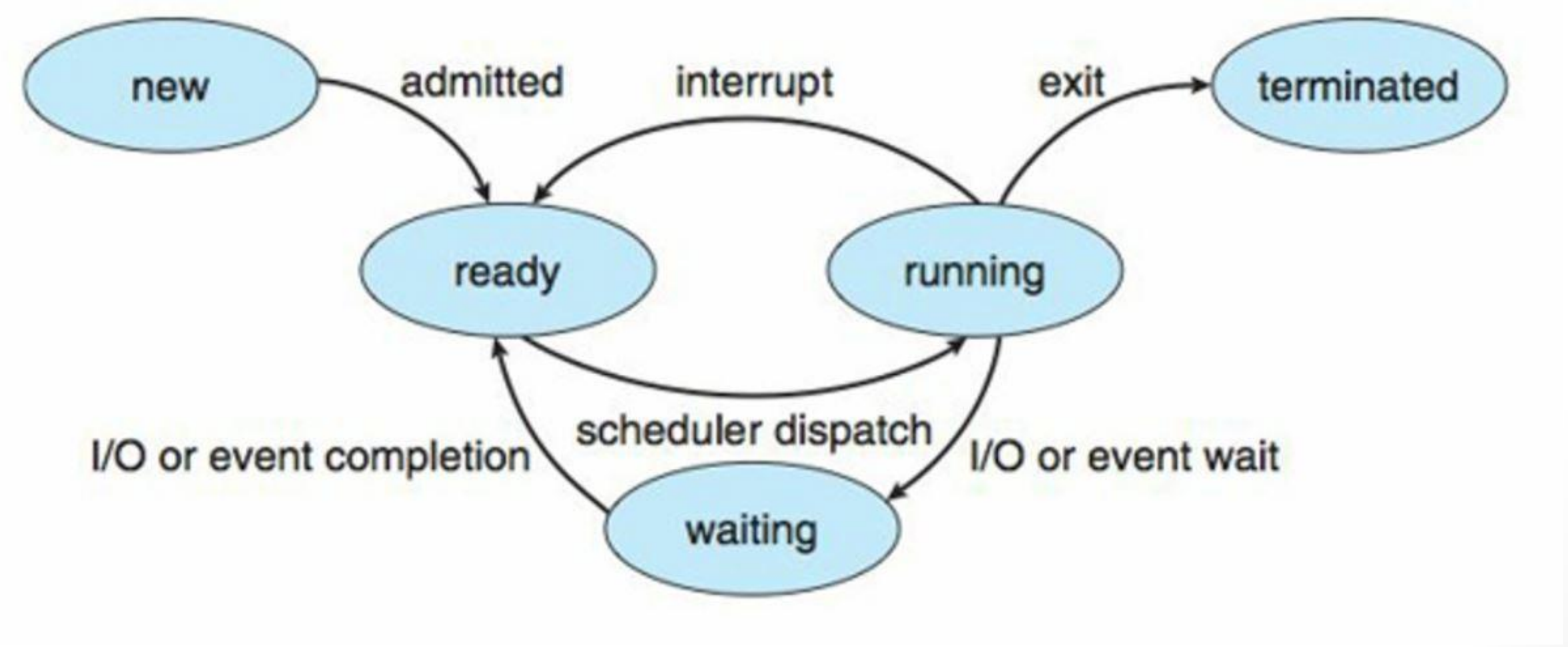
Operating System Design

Processes Operations
Inter Process Communication (IPC)

Neda Nasiriani
Fall 2018

Process

Process Lifecycle

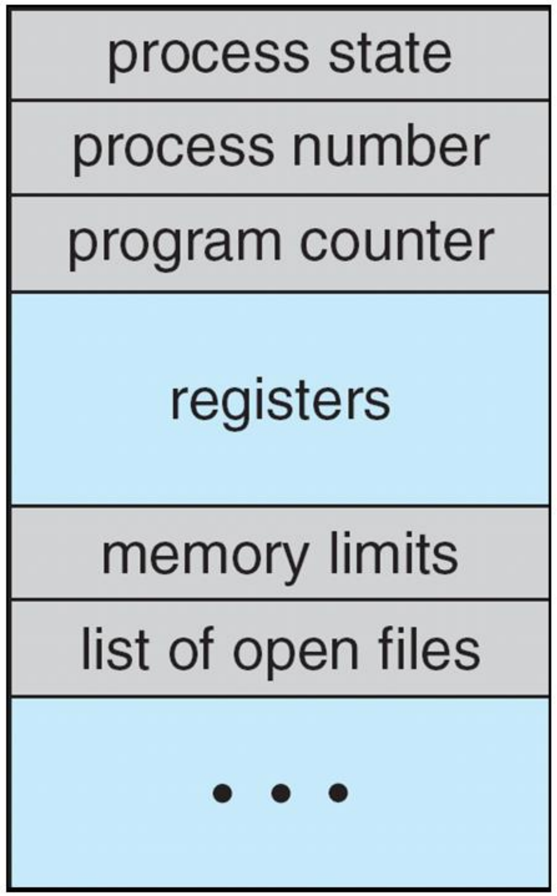


What information is needed?

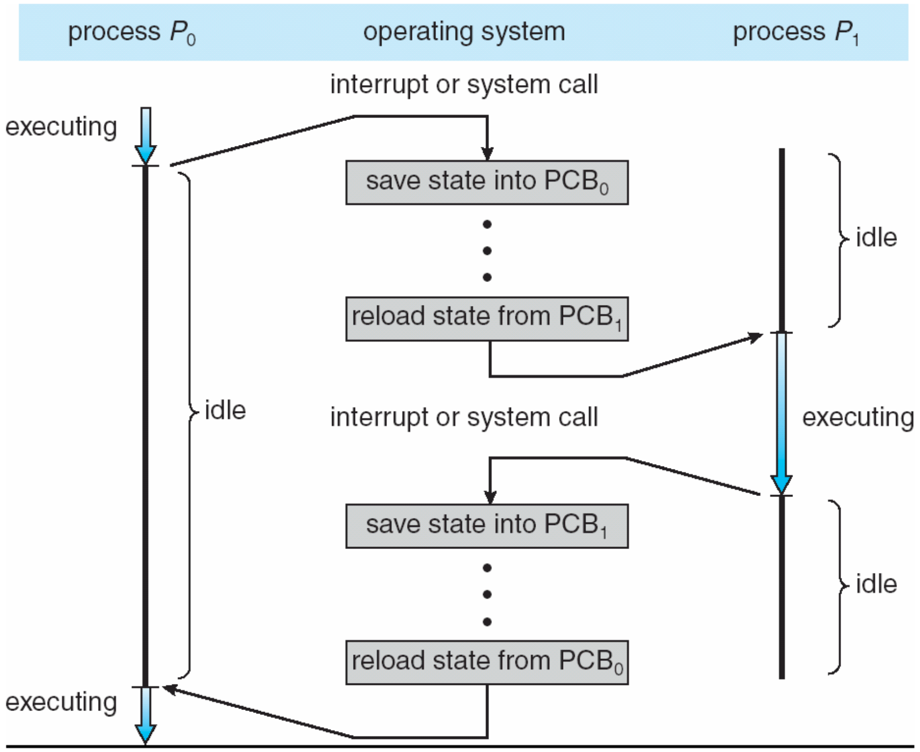
- If you want to design a scheduler to divide your time resource between a bunch of different processes, what info would you need in order to schedule effectively and fairly
  - Process state – running, waiting, etc
  - Program counter – location of instruction to next execute
  - CPU registers – contents of all process-centric registers
  - CPU scheduling information- priorities, scheduling queue pointers
  - Memory-management information – memory allocated to the process
  - Accounting information – CPU used, clock time elapsed since start, time limits
  - I/O status information – I/O devices allocated to process, list of open files

Where to keep that information?

- There is a data type called Process Control Block (PCB) that contains all this information about each process



CPU Switch between Processes



- Context Switch: When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**
- This time is pure overhead!

Process Creation: How?

- An existing process can create a new process by calling the fork() system call
- fork() runs once in the parent process but returns two times,
  1) In the child process, with returning value of 0
  2) In the parent process, with the value of the child process id
- Both the child and the parent process start executing with the instruction that follows the fork() system call
- The child process gets a copy of the parent's data space, heap and stack (It is a separate copy from the parent's)

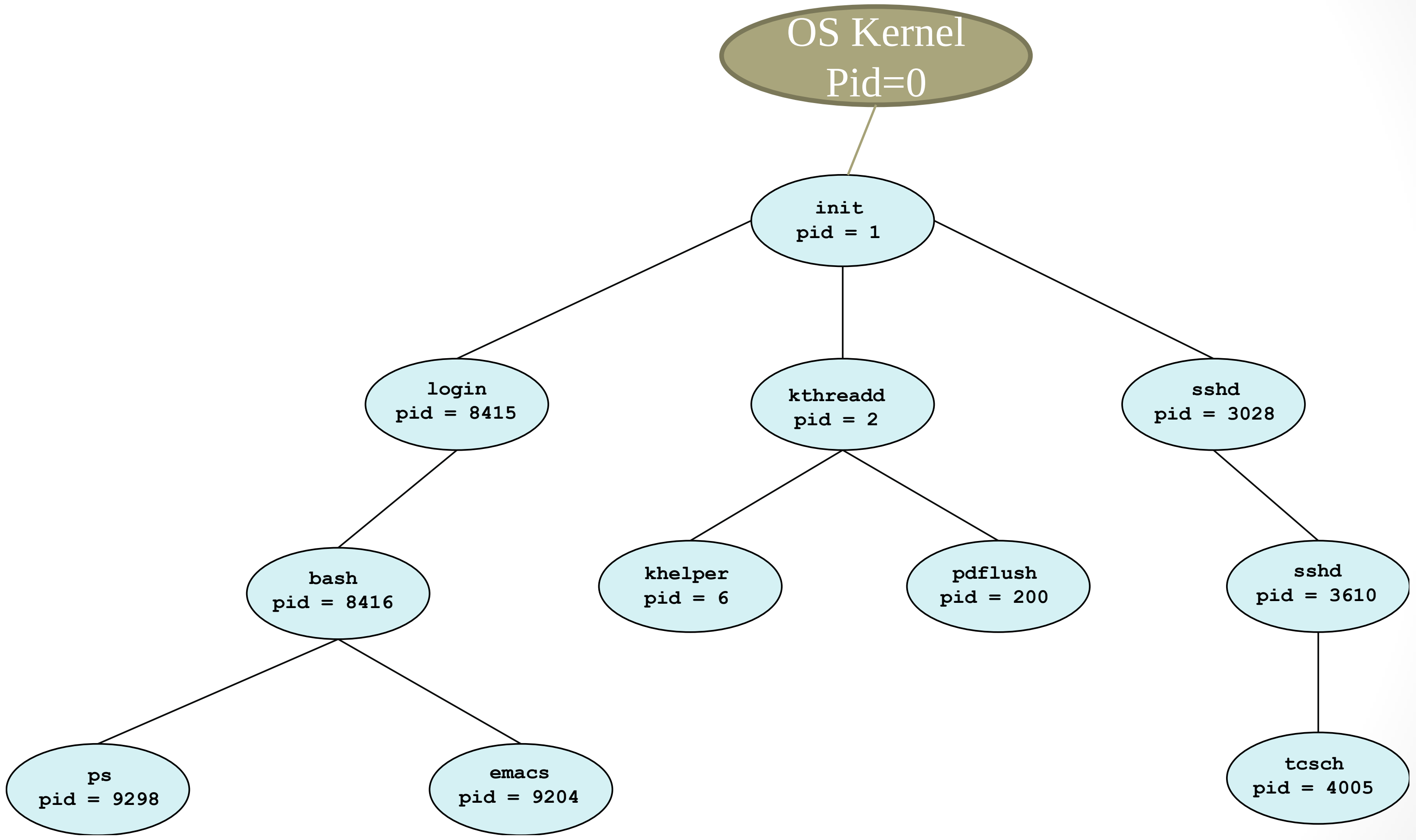
Quiz 2

You can use your notes.

Process Creation

- Processes may be created or deleted dynamically in the system
- Examples of creating a process within a process?
  - If you are designing a web server, you need to constantly listen to possible incoming requests
  - Also there could be 1000 of request every second, how can you address them all in a timely fashion?
  - You would like to run another program within your program
- Parent process creates children processes, which, in turn can create other processes, forming a tree of processes.

A Process Tree on Linux



Process Creation

- Resource sharing:
  - Parent and children share all resources,
  - Children share subset of parent's resources,
  - Parent and child share no resources.
- Execution:
  - Parent and children execute concurrently,
  - Parent may wait until children terminate.
- Address space:
  - Child has duplicate of parent's address space, or
  - Child can have a program loaded onto it.
- UNIX examples:
  - fork system call creates new process and returns with a pid (0 in child, > 0 in the parent),

FYR: Linux fork()

- The new process created by fork is called the child process. This function is called once but returns twice. The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children. The reason fork returns 0 to the child is that a process can have only a single parent, and the child can always call getppid to obtain the process ID of its parent. (Process ID 0 is reserved for use by the kernel, so it's not possible for 0 to be the process ID of a child.)

- Both the child and the parent continue executing with the instruction that follows the call to fork. The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack. Note that this is a copy for the child; the parent and the child do not share these portions of memory. The parent and the child do share the text segment, however (Section 7.6).

- Modern implementations don't perform a complete copy of the parent's data, stack, and heap, since a fork is often followed by an exec. Instead, a technique called copy-on-write (COW) is used. These regions are shared by the parent and the child and have their protection changed by the kernel to read-only. If either process tries to modify these regions, the kernel then makes a copy of that piece of memory only, typically a "page" in a virtual memory system. Section 9.2 of Bach [1986] and Sections 5.6 and 5.7 of McKusick et al. [1996] provide more detail on this feature.

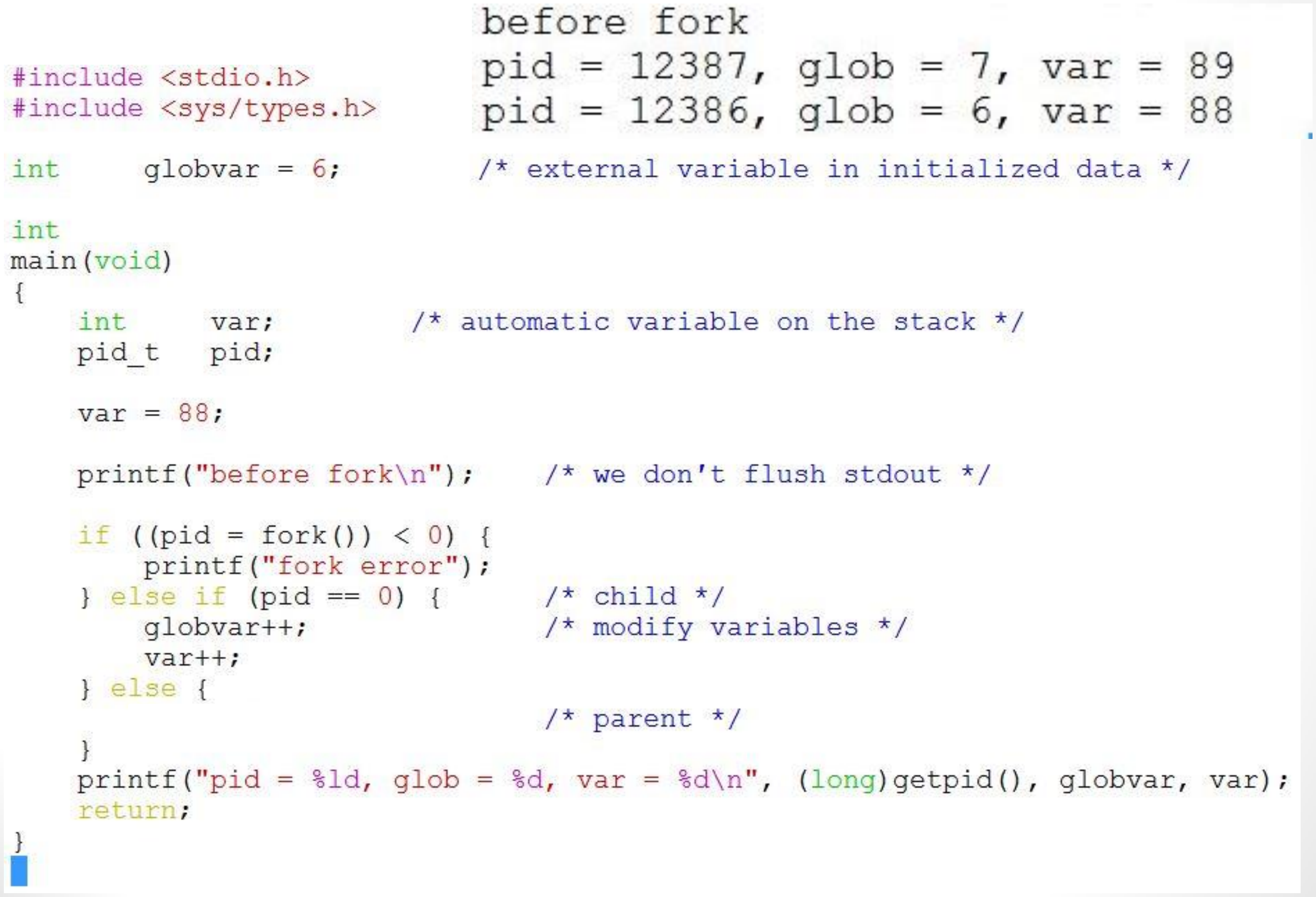
Process Creation: How?

- An existing process can create a new process by calling the fork() system call
- fork() runs once in the parent process but returns two times,
  1) In the child process, with returning value of 0
  2) In the parent process, with the value of the child process id
- Both the child and the parent process start executing with the instruction that follows the fork() system call
- The child process gets a copy of the parent’s data space, heap and stack (It is a separate copy from the parent’s)

Process Creation: sharing resources

- The child process gets **its own copy** of the data section, stack and heap of the parent process
- The child process get a duplicate of all open file descriptors in its parent process
  - The parent and the child share a file table entry for every open descriptor
  - The parent and the child share the same file offset
    - If a child process is writing to standard output when the parent process is executing it can append to the end of standard output
- Does changes in the child variables change the parent variables?
  - No

Process Creation: example

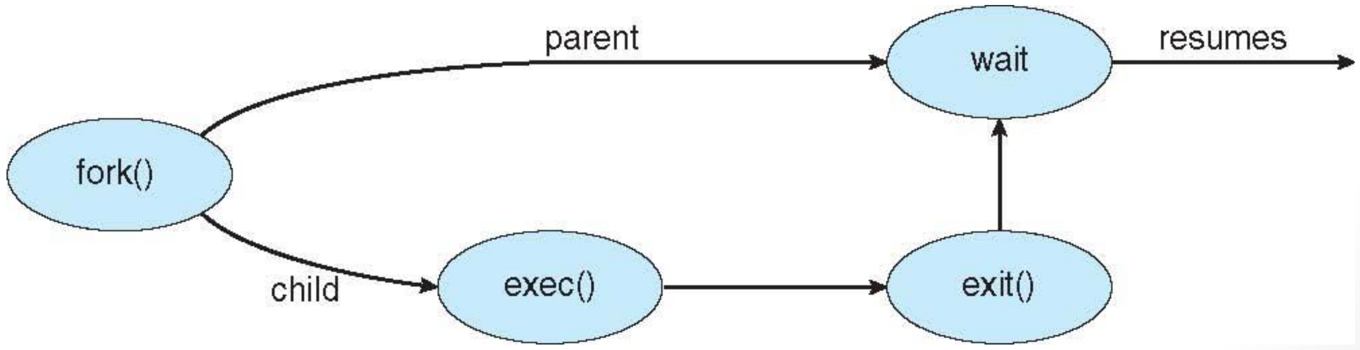


```
before fork
pid = 12387, glob = 7, var = 89
pid = 12386, glob = 6, var = 88
```

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

int     globvar = 6;        /* external variable in initialized data */

int
main(void)
{
    int     var;        /* automatic variable on the stack */
    pid_t   pid;

    var = 88;

    printf("before fork\n");    /* we don't flush stdout */

    if ((pid = fork()) < 0) {
        printf("fork error");
    } else if (pid == 0) {      /* child */
        globvar++;              /* modify variables */
        var++;
    } else {
                                /* parent */
    }
    printf("pid = %ld, glob = %d, var = %d\n", (long)getpid(), globvar, var);
    return;
}
```

Process Creation Diagram

- The parent can wait on the child process by system calls
  - pid_t wait (int * status);
  - pid_t waitpid (pid_t pid, int * status, int options);
  - Both these return process id on successful return or -1 in case of an error
- Otherwise the parent process might terminate before the child process terminates!!!



Process Execution

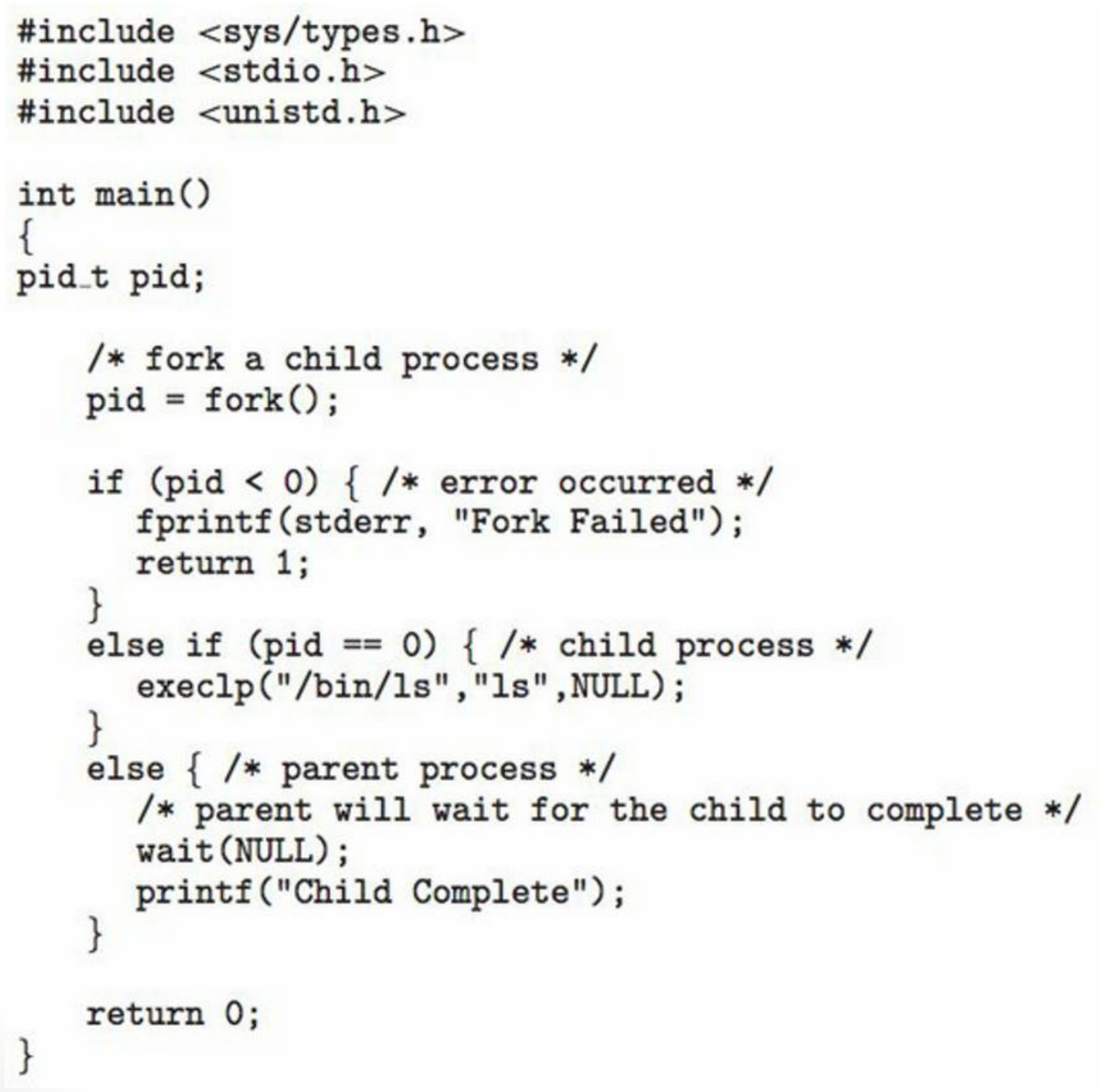
- **A child process can execute**

  1) A segment of its parent code (as we saw before)

  2) Another program that is loaded to its memory using exec() system call

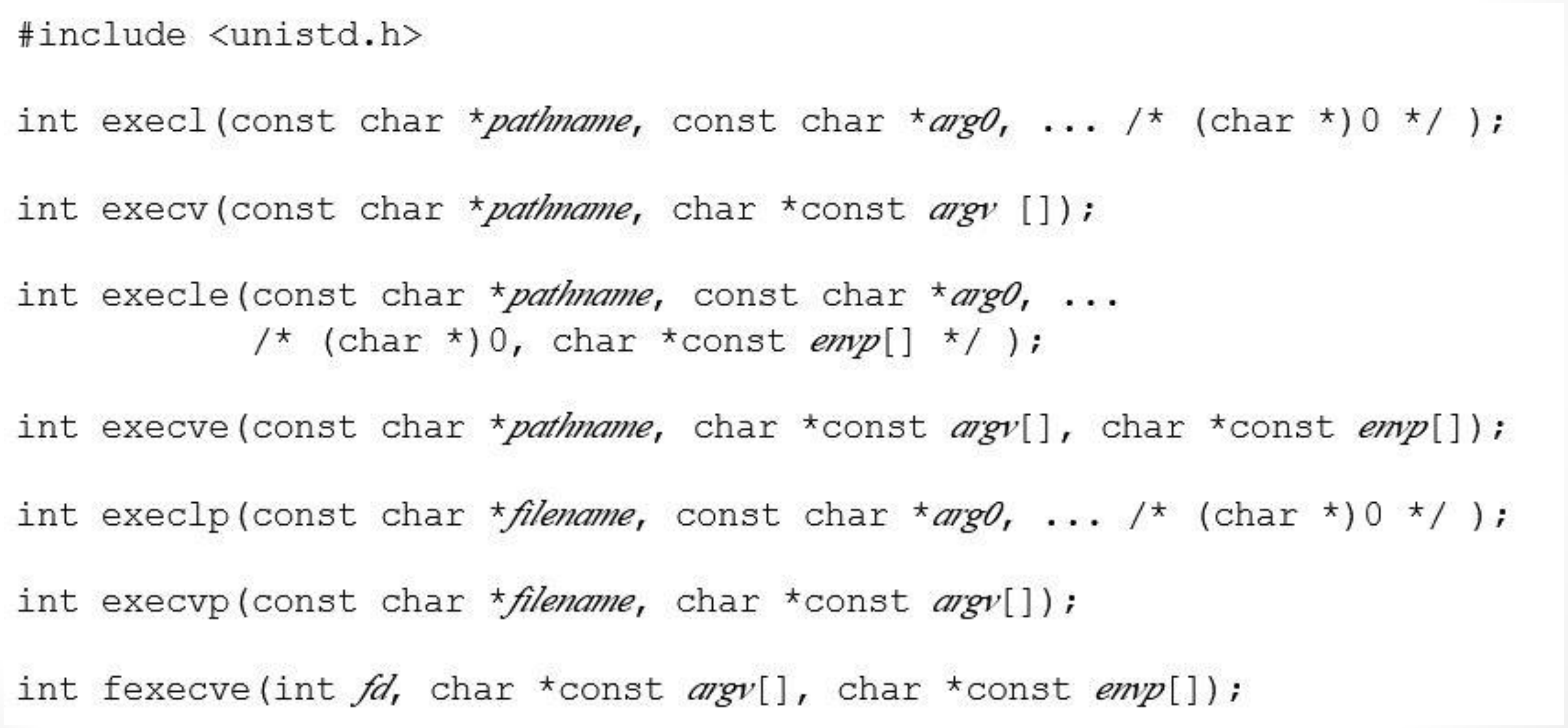
- When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function

Process Execution: example



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

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

FYR: exec system call



```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv []);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

Process Termination

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait(&status)`**)
  - Process' resources are deallocated by operating system
- The parent process may wait for termination of a child process by using the **`wait()`** system call. The call returns status information and the pid of the terminated process

```
      pid = wait(&status);
```

- If parent has not called **`wait()`** yet but the child is terminated, the info of child is still kept in the process table. This child process is a **zombie**
- If parent terminated without invoking **`wait`**, process is an **orphan**, which is adopted by **init** process

Demo!

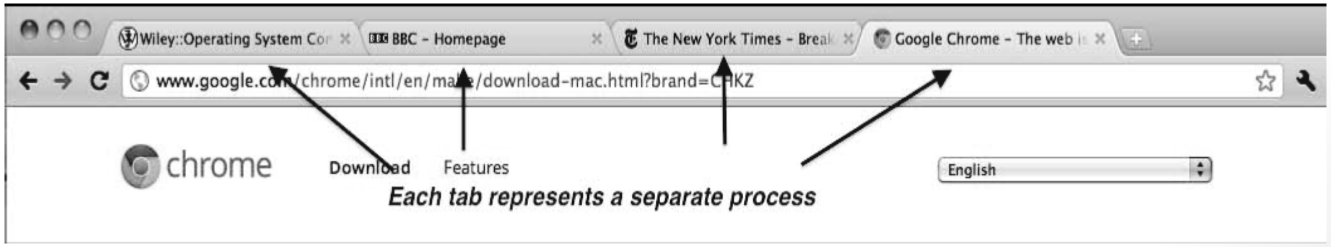
Interprocess Communication (IPC)

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Example: Chrome browser

Chrome Browser

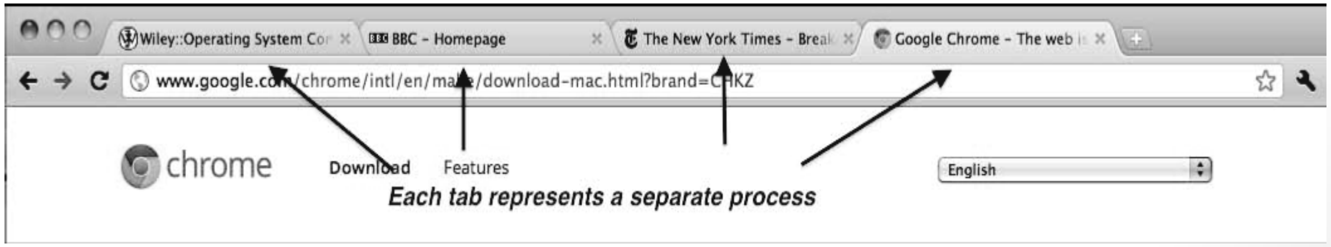
- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in


Wiley::Operating System Cor
BBC - Homepage
The New York Times - Break
Google Chrome - The web is
www.google.com/chrome/intl/en/make/download-mac.html?brand=CHKZ
chrome
Download
Features
English
Each tab represents a separate process

Chrome Browser

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome B types of processes:
  - **Browser**
  - **Renderer** Javascript
    - Runs in sa zing effect of security exploi
  - **Plug-in** process for each type of plug-in

Renderer needs to communicate with the browser
And plug-in process needs to talk to the Browser if a tab is using a plug-in

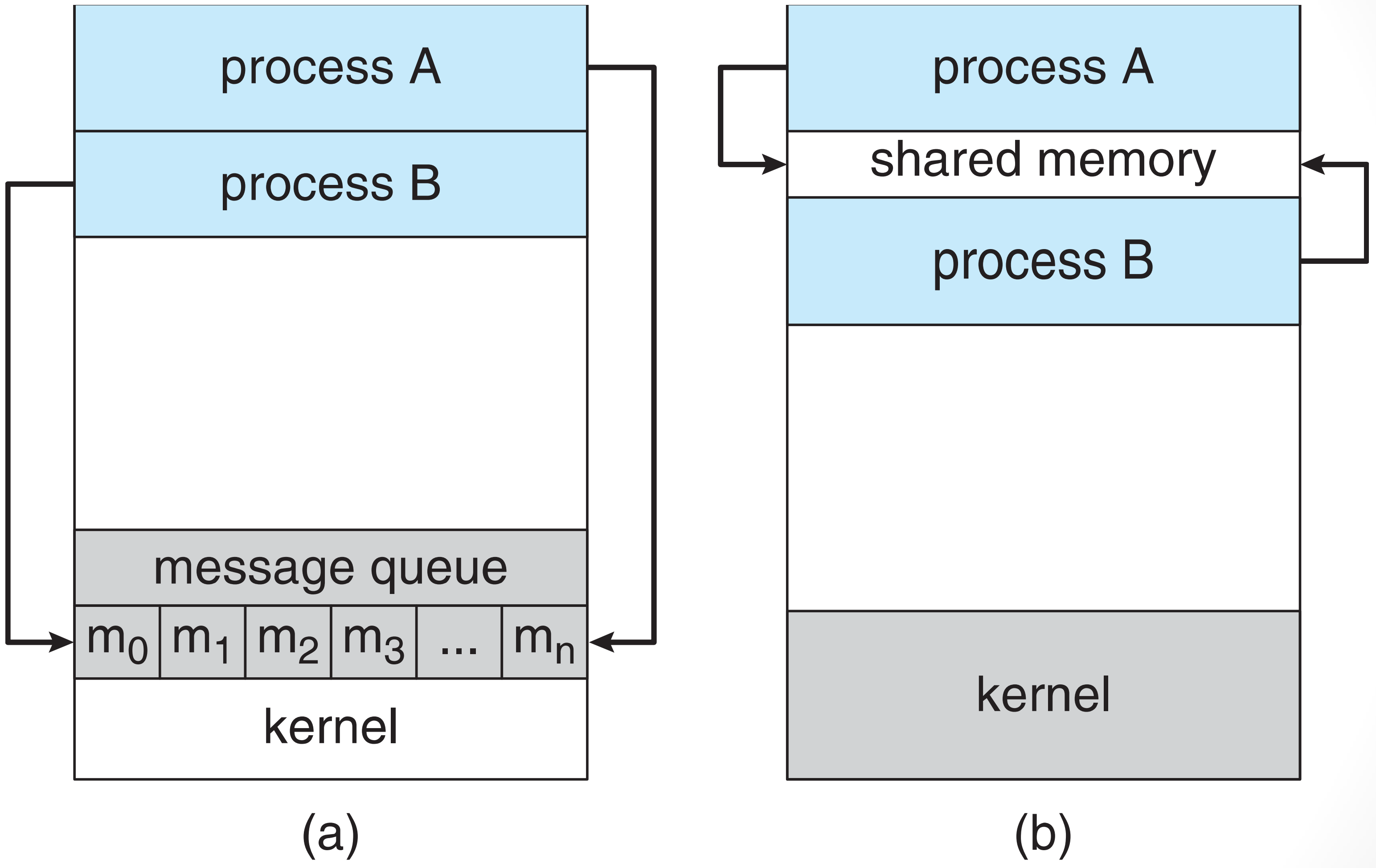

Wiley::Operating System Cor
BBC – Homepage
The New York Times – Breal
Google Chrome – The web i
www.google.com/chrome/intl/en/make/download-mac.html?brand=CHKZ
chrome
Download
Features
English

Each tab represents a separate process

Interprocess Communication

- Example: Chrome browser
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communication Models



(a) Message passing. (b) shared memory.

Shared Memory: Producer Consumer Example

- Assume process A is producing items and put them into a buffer of size N
- Process B is reading from the buffer
- Example
  - Compiler sends assembly code to the assembler

B: Consumer Code:

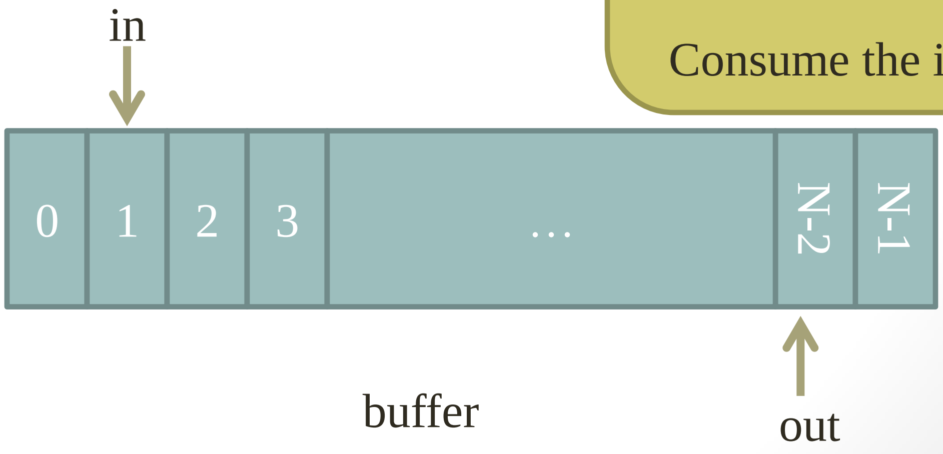
item = buffer [out]
out = (out + 1)%N

Consume the item

A:Producer Code:

Produce item

buffer [in] = item
in = (in + 1)%N



Producer Consumer Example

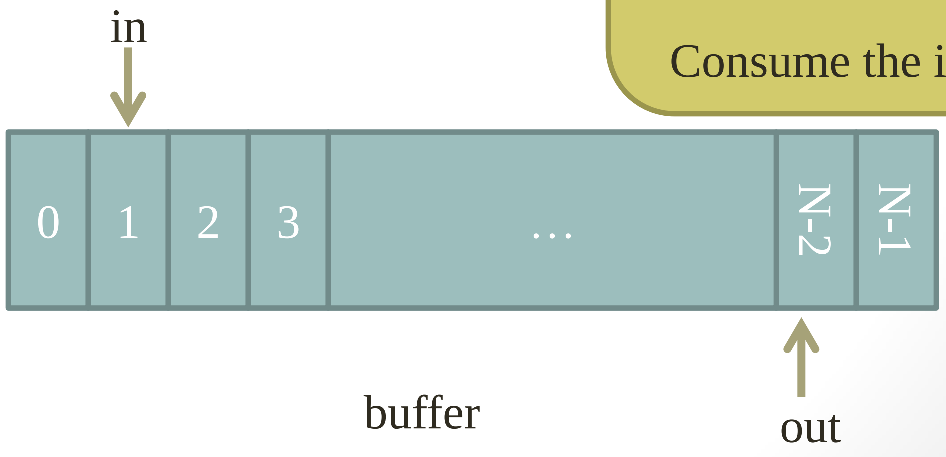
- What are the things that can go wrong here?
  - The buffer could be empty (nothing to be read by the consumer)
  - The buffer could be full (no items can be added by the producer)
  - What if they try to access the buffer at the same time?
- Initially in = out = 0

B: Consumer Code:

```
item = buffer [out]
out = (out + 1)%N

Consume the item
```

A:Producer Code:

```
Produce item

buffer [in] = item
in = (in + 1)%N
```



Producer-Consumer Example

```
item next_produced;
while (true) {
        /* produce an item in next produced */
        while (((in + 1) % N) == out)
                ; /* do nothing */
        buffer[in] = next_produced;
        in = (in + 1) % N;
}
```

```
item next_consumed;
while (true) {
        while (in == out)
                ; /* do nothing */
        next_consumed = buffer[out];
        out = (out + 1) % N;

        /* consume the item in next consumed */
}
```

Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Message Passing

- 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 communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Message Passing

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

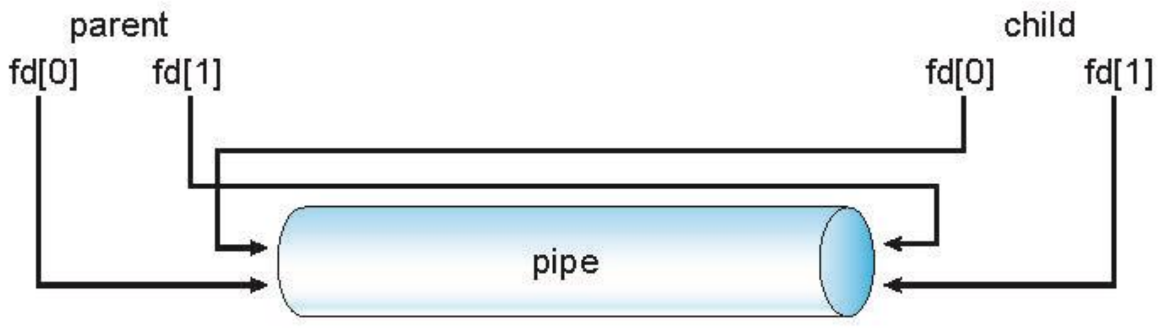
Pipes

Pipes

- Acts as a conduit allowing two processes to communicate
- Issues:
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e., ***parent-child***) between the communicating processes?
  - Can the pipes be used over a network?
- 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.
- Named pipes – can be accessed without a parent-child relationship.

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes



- Windows calls these **anonymous pipes**