

# Processes and Inter-Process Communication

by Mohamad H. Danesh

# What is a Process?

- Program
  - An executable file
- Process
  - An instance of a program that is being executed by the OS.
  - Every process has a unique ID (**PID**).

# Process Statuses

- Running
  - The process is either running or it is ready to run.
- Waiting
  - The process is waiting for an event or for a resource.
- Stopped
  - The process has been stopped, usually by receiving a signal.
- Zombie
  - The process that has completed execution but still has an entry in the process table.
- Orphan
  - The process that its parent is terminated.

# Type of Processes

- Interactive Process
  - Initiated from (and controlled by) a shell
- Daemon Process
  - Run in the background until required

# Related Commands

- top
  - Display top CPU processes.
- ps
  - Report process status.
- pstree
  - Display a tree of processes.
- kill
  - Send signal to a process.

# Related Commands (Cont.)

- jobs
  - List the active jobs.
- bg
  - Place a job in background (similar to &);
- fg
  - Place a job in foreground.
- Ctrl+z
  - Stopped a process.
- Ctrl+c
  - Terminate a process.

# Process Control

- fork and vfork
- exit
- wait and waitpid
- exec
- signal
- kill

# fork

**PID1**

**PID2**

**fork**

**Parent**



**Child**



# fork (cont.)

- `int fork();`
- The **only way** a new process is created by the Unix kernel.
  - The new process created is called the child process.
- The child is a **copy** of the parent.
  - The child gets a copy of the parent's data space, heap and stack.
  - The parent and child don't share these portions of memory.

# fork (cont.)

- This function is called **once**, but return **twice**.
  - returns the process ID of the child process to the parent process.
    - A process can have more than one child.
  - returns a value of 0 to the child process.

# fork Sample

```
main()
{
    int pid;
    pid = fork();
    if (pid < 0)
        // error
    else if (pid == 0)
        //child
    else
        //parent
}
```

# fork (cont.)

- We never know if the child starts executing before the parent or vice versa.
  - This depends on the **scheduling algorithm** used by the kernel.
    - first come, first serve
    - round robin
    - earliest deadline first

# vfork

- `int = vfork();`
- It has the same calling sequence and same return values as `fork`.
- The child **doesn't copy** the parent data space.
  - The child runs in the address space of the parent.
- With `vfork`, child runs first, then parent runs.

# exit

- Normal termination
  - Executing a **return** from the main function.
  - Calling the **exit** function.
  - Calling the **\_exit** function.
- Abnormal termination
  - Calling **abort**.
  - Receives certain signals.

# exit (cont.)

- `int exit (int state);`
- Sometimes we want the terminating process to be able to notify its parent how it terminated.
- For the `exit` and `_exit` function this is done by passing an `exit status` as the argument to these two functions.
- The parent of the process can obtain the `termination status` from either the `wait` or `waitpid` function.

# Exit Reserved Codes

- 0 - Successful
- 1 - Catchall for general errors
- 2 - Misuse of shell builtins (according to Bash documentation)
- 126 - Command invoked cannot execute
- 127 - “command not found”
- 128 - Invalid argument to exit
- 128+n - Fatal error signal “n”
- 130 - Script terminated by Control-C
- 255\\* - Exit status out of range



# Termination Conditions

- Parent terminate before the child
  - The **init** process becomes the parent process of any process whose parent terminated.
  - The child process is called **orphan**

# Termination Conditions

- Child terminate before the parent
  - The child is completely disappeared, but the parent wouldn't be able to fetch its termination status.
  - The kernel has to keep a certain amount of information for every terminating process.
  - The process that has terminated, but whose parent has not waited for it, is called **zombie**.

# wait

- The parent of the process can obtain the termination status from either the **wait** or **waitpid** function.

# wait (cont.)

- The process that calls wait or waitpid can:
  - Block (if all of its children are still running)
  - Return immediately with the termination status of a child (if a child has terminated)
  - Return immediately with an error (if it doesn't have any child process)

# wait and waitpid

- The difference between these two function:
  - wait can block, while waitpid has an option that prevents it from blocking.
  - waitpid doesn't wait for the first child to terminate (it can control which process it waits for)

# exec

- Replaces the program in the current process with a brand new program.
- When a process calls one of the exec functions, that process is completely **replaced** by the new program.
- The process ID doesn't change across an exec.

# exec functions

- `int execl(char *path, char *arg0, ... /*(char *) 0 */);`
- `int execl_e(char *path, char *arg0, ... /*(char *) 0, char *envp[] */);`
- `int execlp(char *filename, char *arg0, ... /*(char *) 0 */);`
- `int execv(char *pathname, char *argv0[]);`
- `int execve(char *pathname, char *argv0[], char *envp[]);`
- `int execvp(char *filename, char *envp[]);`

# signal

- Signals are software interrupts.
- The name of signals all begin with the three character SIG  
: **SIG**ABRT



# signal (cont.)

- `void (*signal (int signo, void (*func) (int))) (int);`
- Kernel do when a signal occurs:
  - Ignore the signal.
  - Catch the signal.
    - Call a function of ours whenever the signal occurs.
  - Let the default action apply.
    - Every signal has a default action.

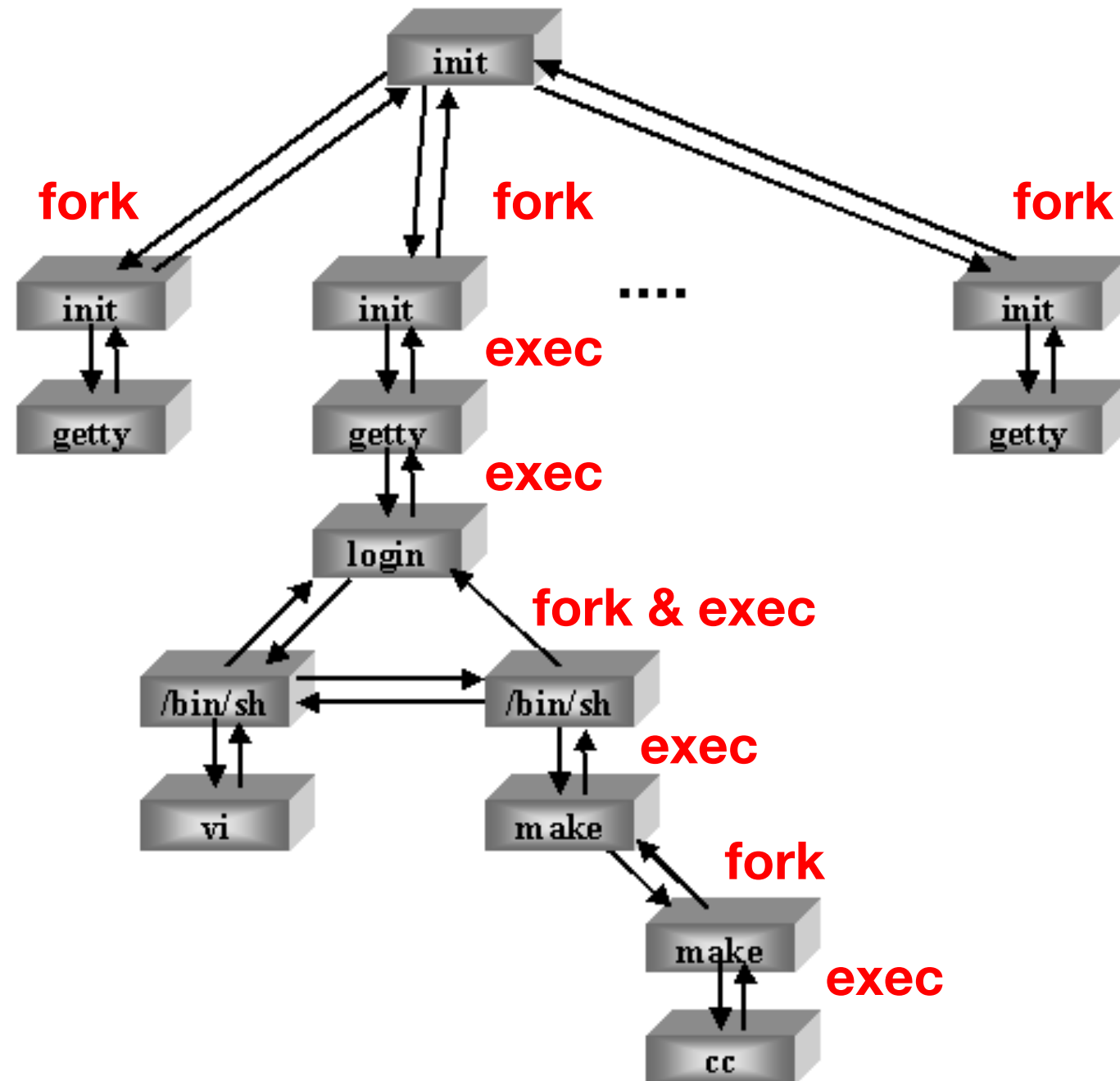
# signal Sample

```
main()
{
    signal (SIGUSER, sig_user);
    while (1);
}
//-----
void sig_user (int signo)
{
    if (signo == SIGUSER)
        printf ("receive signal\n");
    return;
}
```

# kill

- `int kill (int pid, int signo);`
- Send a signal to a process or group of the processes.

# Process Relationship

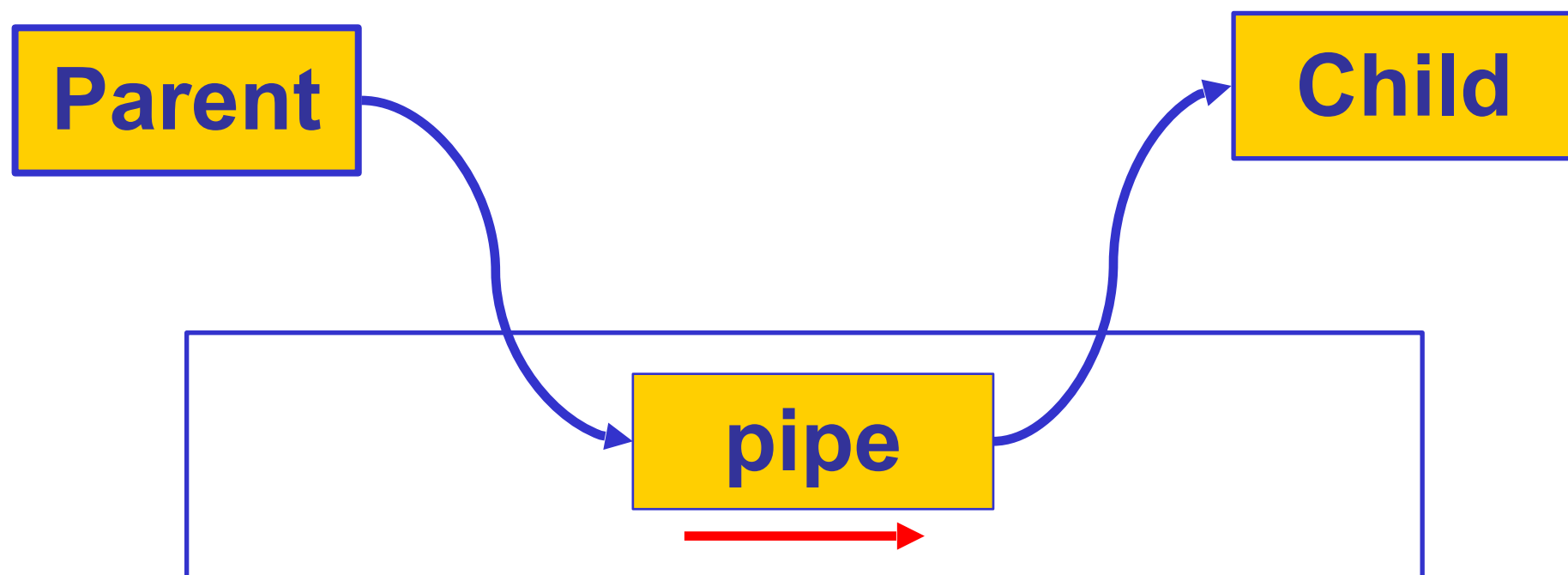


# IPC

- Pipe
- FIFO
- Message queue
- Shared memory

# pipe

- It provides a one-way flow of data.
- It is in the kernel
- It can only be used between processes that have **a parent** process in common.



# pipe (cont.)

- `int pipe (int *filedes);`
- `filedes[0]` : open for reading
- `filedes[1]` : open for writing
- pipe command :
  - `cat test.txt | grep testString`

# FIFO

- It is similar to a pipe.
- Unlike pipes, a FIFO has a name associated with it (**named pipe**).
- It uses a file as a communication way.



# Message Queue

- Message queues are a **linked list** of messages stored within the kernel.
- We don't have to fetch messages in a first-in, first-out order.
  - We can fetch messages based on their type field.
- A process wants to impose some structure on the data being transferred.

# Message Queue (cont.)

- `int msgget (key_t key, int msgflag);`
- A new queue is created, or an existing queue is open by `msgget`.

# Message Queue (cont.)

- `int msgsnd(int msgid, void *ptr, size_t len, int flag);`
- Data is placed onto a message queue by calling `msgsnd`;

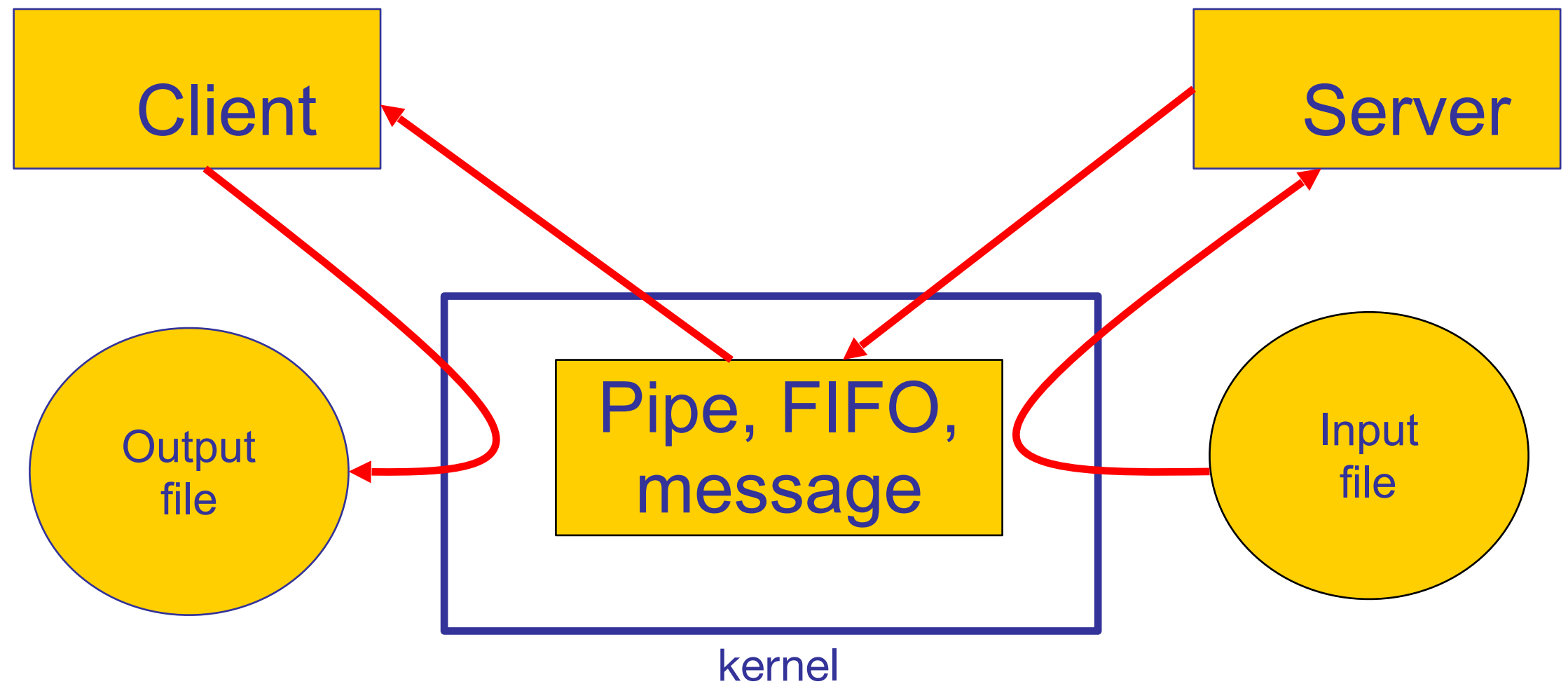
# Message Queue (cont.)

- `int msgrcv (int msgid, void *ptr, sizet len, long mtype, int flag);`
- The type argument lets us specify which message we want:
  - `mtype == 0`, the first message on the queue
  - `mtype > 0`, the first message on the queue whose type equals `mtype`.
  - `mtype < 0`, the first message on the queue whose type is the lowest value less or equal to the absolute value of `mtype`.

# Message Queue (cont.)

- `int msgctl (int msgid, int cmd, struct msgid_ds *buf);`
- The `msgctl` function performs various operations in a queue.

# Shared memory



# Shared memory (cont.)

- `int shmget (key_t key, int size, int flag);`
- A shared memory segment is created.

# Shared memory (cont.)

- `Char *shmat (int shmid, char *shmaddr, int shmflg);`
- The `shmget` does not provide access to the segment for the calling process.
- We must attach the shared memory segment by calling the `shmat` system call.



# Shared memory (cont.)

- `int shmdt (char *shmaddr);`
- When a process is finished with a shared memory segment, it detaches the segment by calling the `shmdt` system call.
- This call does not delete the shared memory segment.

# Shared memory (cont.)

- `int shmctl (int shmid, int cmd, struct shmid_ds *buf);`
- The `msgctl` function performs various operations in a shared memory segment.

# Semaphore

- Semaphores are a synchronization primitive.
- To obtain a shared resource:
  - Test the semaphore that controls the resource.
  - If the value is positive the process can use the resource. The process decrements the value by 1.
  - If the value is 0, the process goes to sleep until the value is greater than 0.

# Semaphore (cont.)

- `int semget (key_t key, int nsems, int flag);`
- This function get a semaphore ID.
- `int semctl (int semid, int semnum, int cmd, union semun arg);`
- The `semctl` function performs various operations in a semaphore.

# To Do

- Open two terminals and use *write* to send a message from one to the other. Describe how it works.
- Write a shell script to check if a given command operation was successful.
- Deliverables:
  - What does `kill -9 -1` do?
  - Name 3 processes that couldn't have had `init` as an initial parent.
  - Write about `exec` different functions.