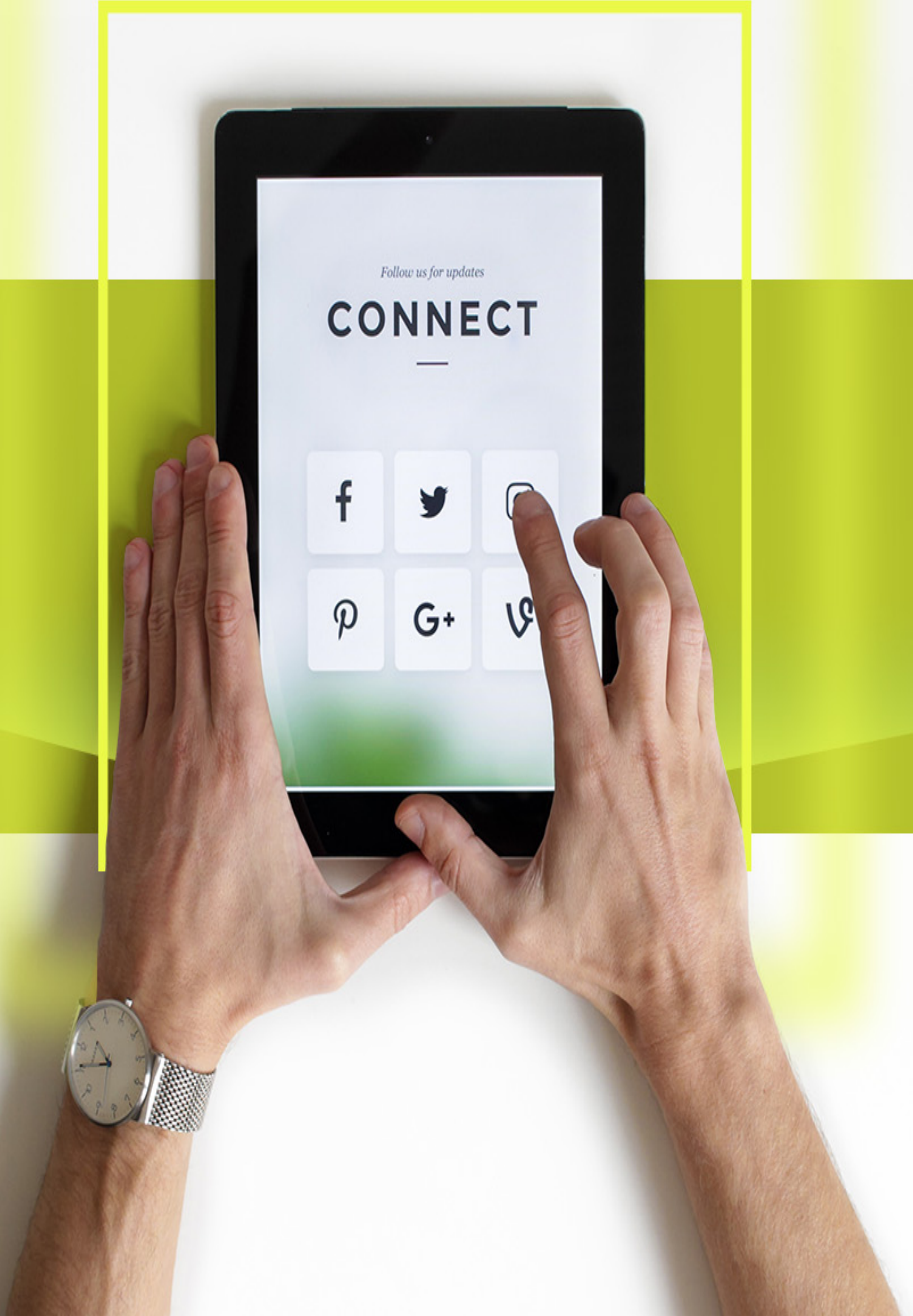
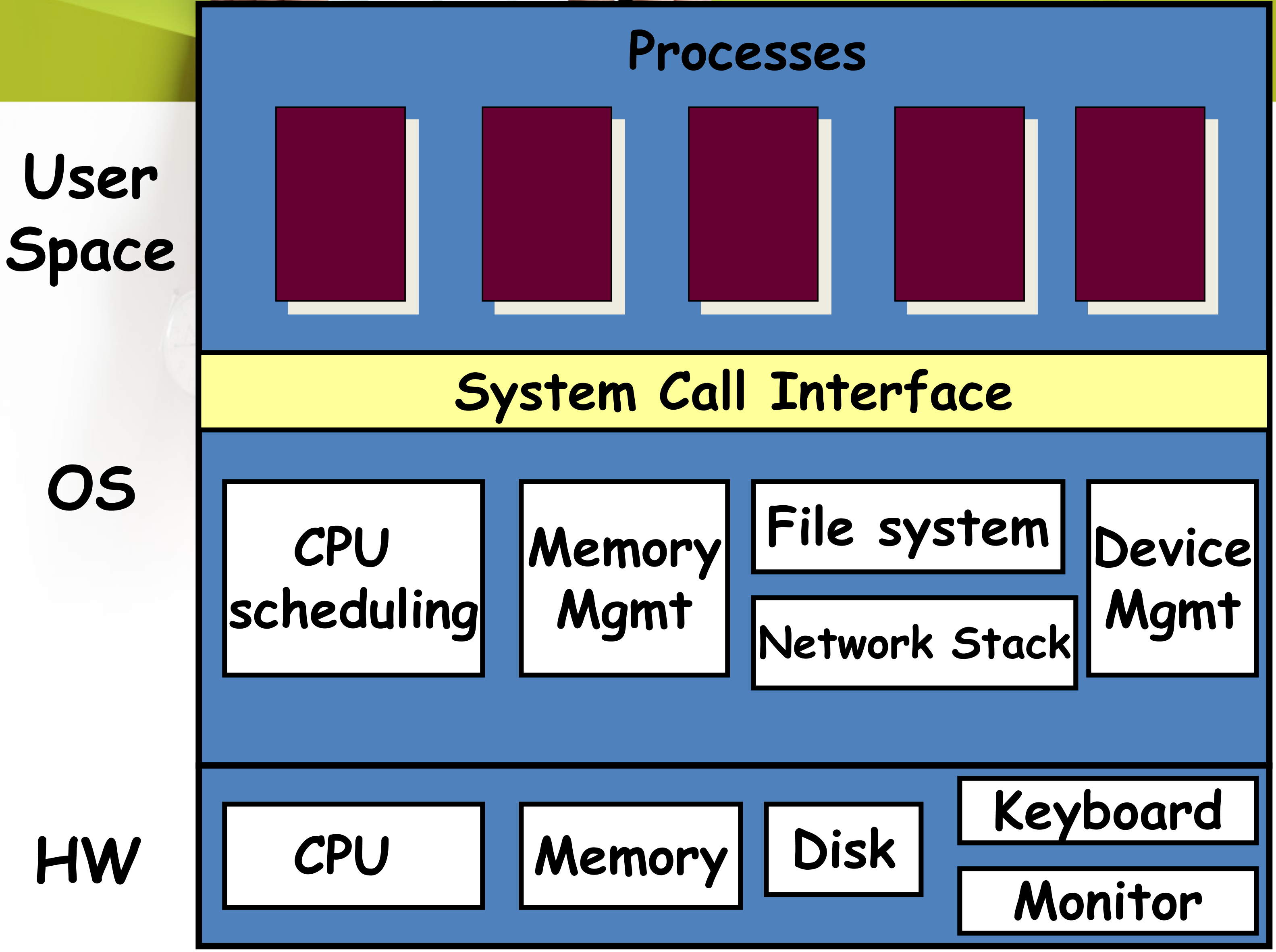


# Process

# Topics

- Introducing process: the basic mechanism for concurrent programming
  - Process management related system calls
    - Process creation
    - Process termination
    - Running another program in a process
    - Synchronization between Parent/child processes

# Computer systems Overview



User Space

Processes

System Call Interface

OS

CPU scheduling

Memory Mgmt

File system

Network Stack

Device Mgmt

HW

CPU

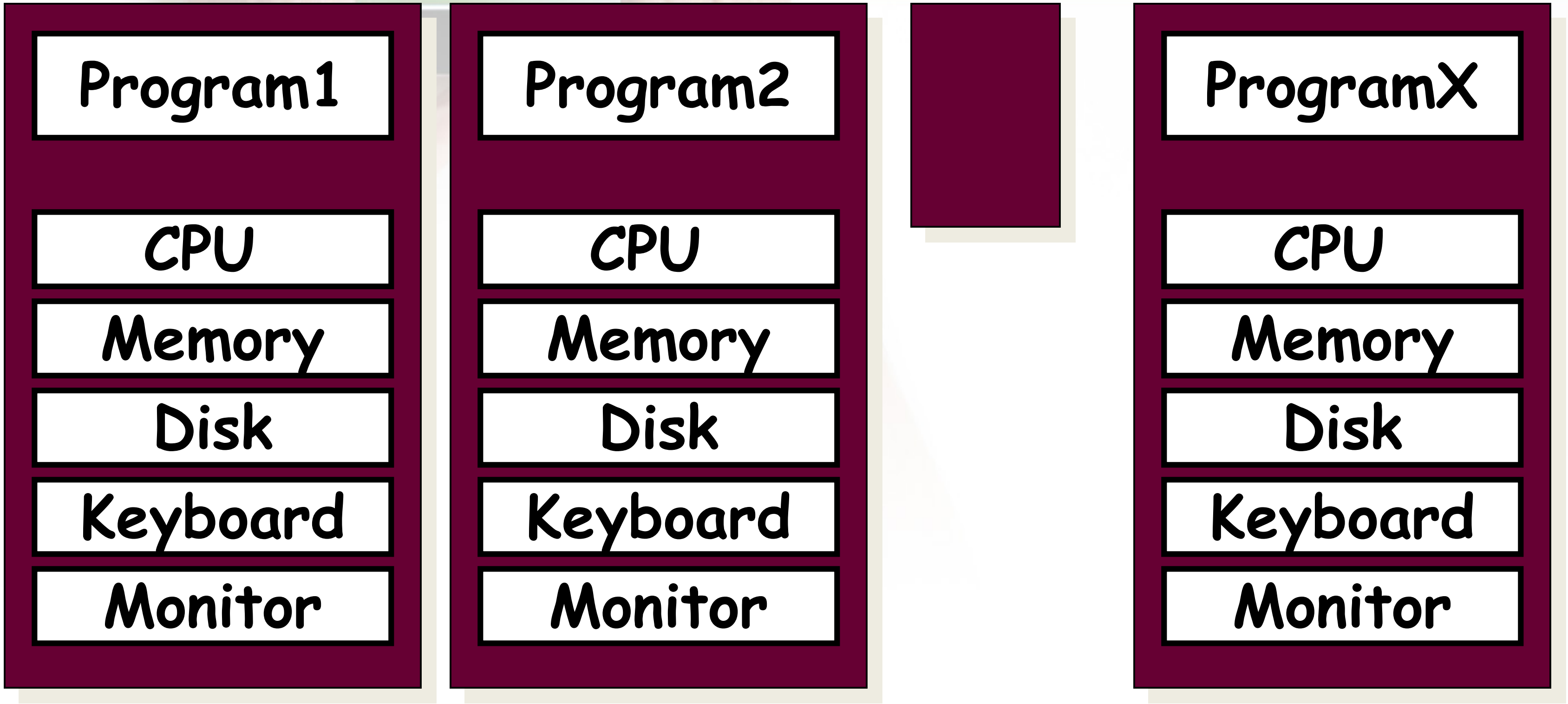
Memory

Disk

Keyboard

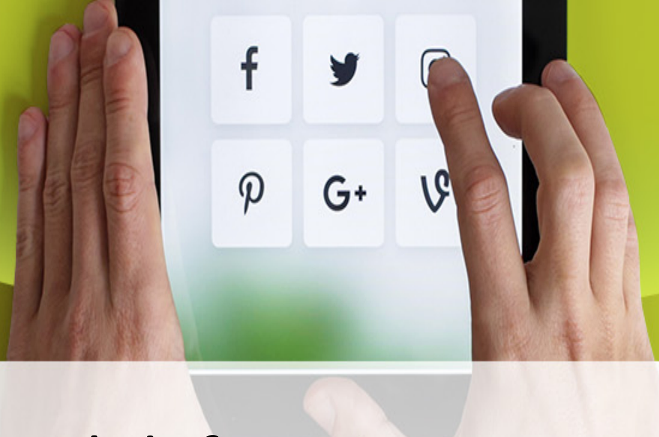
Monitor

# Computer systems user's view



| Program1 | Program2 | | ProgramX |
|---|---|---|---|
| CPU | CPU | | CPU |
| Memory | Memory | | Memory |
| Disk | Disk | | Disk |
| Keyboard | Keyboard | | Keyboard |
| Monitor | Monitor | | Monitor |

Each program owns its own (virtual) computer.
The execution of a program does not affect one another.



# Process

- Informal definition:

  A process is a program in execution.

- Process is not the same as a program.
  – Program is a passive entity stored in disk
  – Program (code) is just one part of the process.

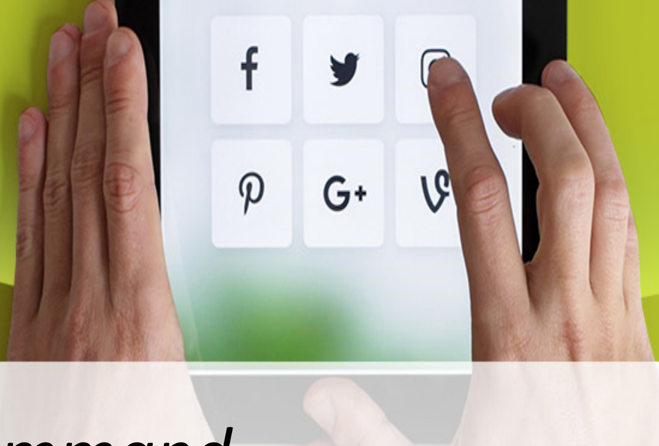
# What else in a process?

- Process context – everything needed to run resume execution of a program:
  - Memory space (static, dynamic)
  - Procedure call stack
  - Open files, connections
  - Registers and counters :
    - Program counter, Stack pointer, General purpose registers
  - ……

# Why process?

- Multiple processes (users) share the system resources.
- Multiple processes run independently

- Which of the following is more important?
  - Process isolation (the illusion that each process is the only one on the machine).
  - Process interaction (synchronization, inter-process communication).

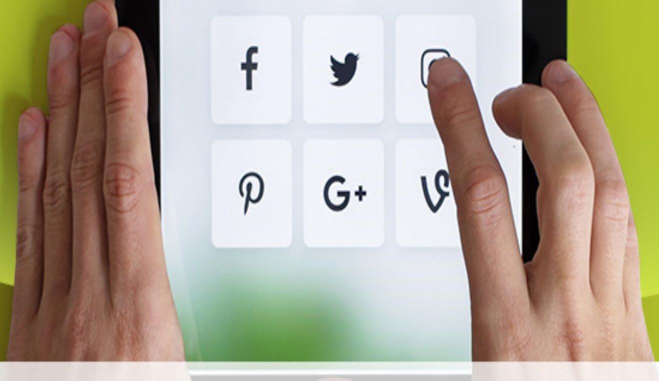


# Examining Processes in Unix

- *ps command*
  - Standard process attributes
- /proc  directory -> cpuinfo
  - More interesting information.
  - Try “man proc”
- *Top, vmstat command*
  - *Examining CPU and memory usage statistics.*

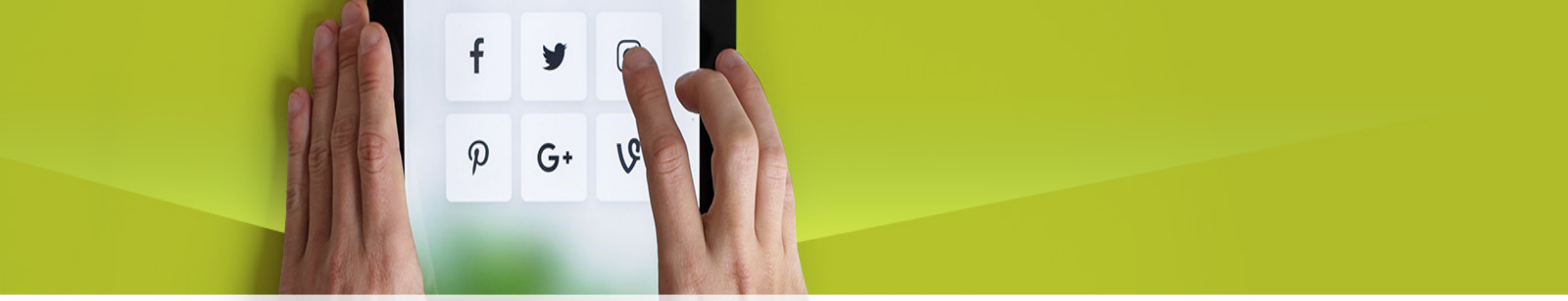
# Creating a New Process - fork()

```
pid = fork();

if (pid == -1) {
    fprintf(stderr, "fork failed\n");
    exit(1);
}
if (pid == 0) {
    printf(“This is the child\n");
    exit(0);
}
if (pid > 0) {
    printf(“This is parent. The child is %d\n", pid);
    exit(0);
}
```



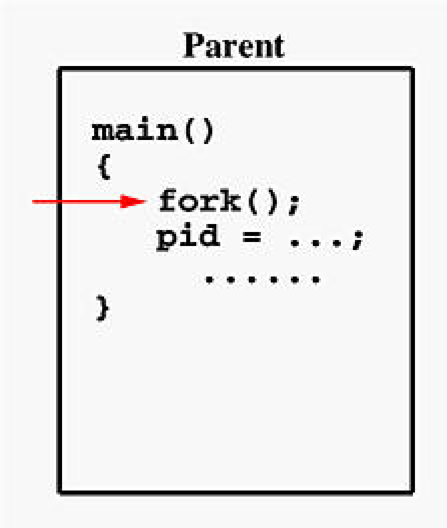
# Points to Note

- fork() is called once …
- … but it returns twice!!
  - Once in the parent and
  - Once in the child
- Fork() basically duplicates the parent process image
  - Both processes are exactly the same after the fork() call.
    - Are there any dependence between the two processes?
  - Provide a way to distinguish the parent and the child.



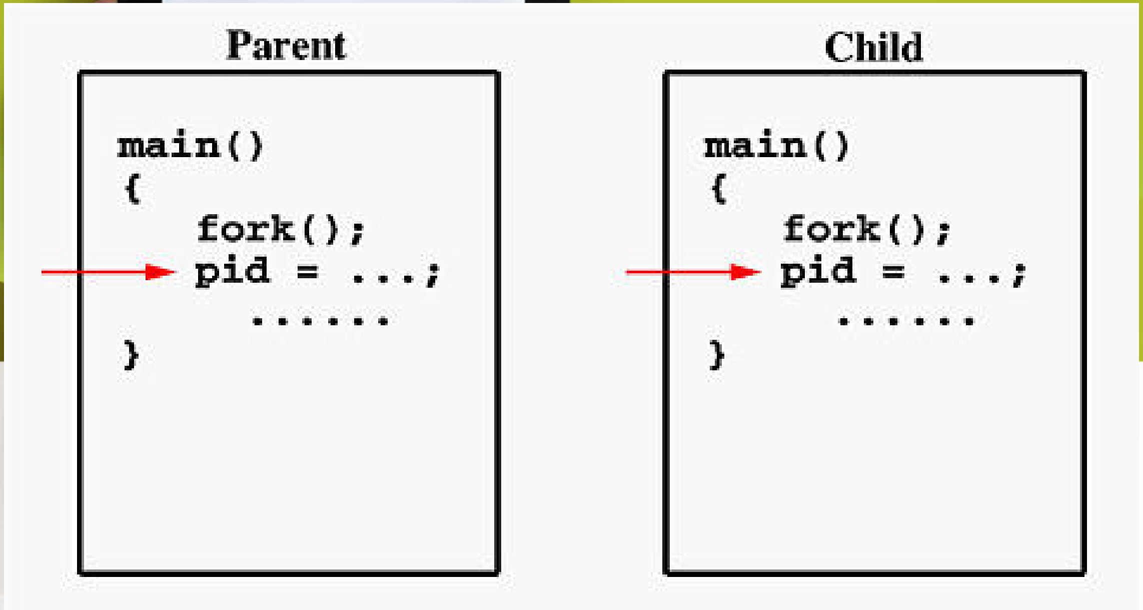
- When the main program executes **fork()**, an identical copy of its address space, including the program and **all** data, is **created**.
- System call **fork()** returns the child **process** ID to the parent and returns 0 to the child **process**.

```
/* ----------------------------------------------------------------- */
/* PROGRAM  fork-01.c                                                */
/*    This program illustrates the use of fork() and getpid() system */
/* calls.  Note that write() is used instead of printf() since the   */
/* latter is buffered while the former is not.                       */
/* ----------------------------------------------------------------- */
#include  <stdio.h>
#include  <string.h>
#include  <sys/types.h>

#define   MAX_COUNT  200
#define   BUF_SIZE   100

void  main(void)
{
    pid_t  pid;
    int    i;
    char   buf[BUF_SIZE];

    fork();
    pid = getpid();
    for (i = 1; i <= MAX_COUNT; i++) {
        sprintf(buf, "This line is from pid %d, value = %d\n", pid, i);
        write(1, buf, strlen(buf));
    }
}
```



Parent

```
main()
{
   fork();
   pid = ...;
     ......
}
```



```
main()
{
    fork();
    pid = ...;
       ......
}
```

```
main()
{
    fork();
    pid = ...;
       ......
}
```

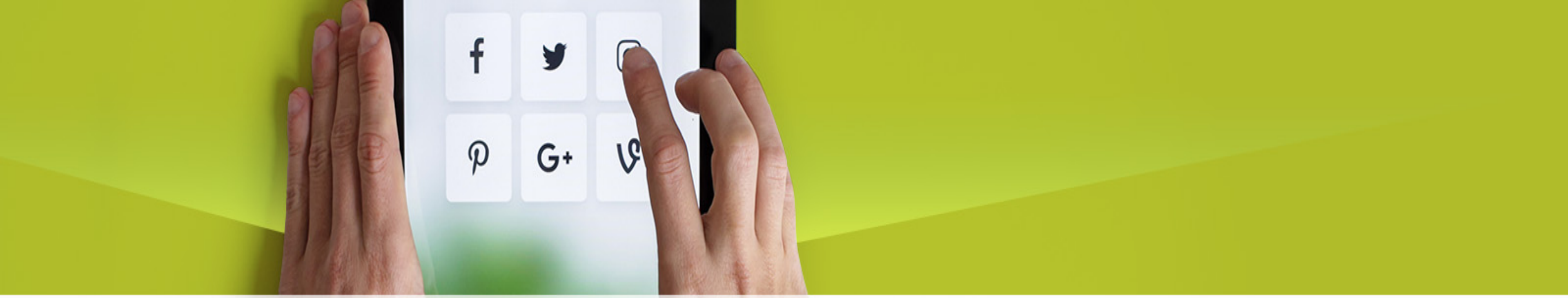
- If the call to **fork()** is executed successfully, Unix will make two identical copies of address spaces, one for the parent and the other for the child.
- Both processes will start their execution at the next statement following the **fork()** call. In this case, both processes will start their execution at the assignment statement.

- Both processes start their execution right after the system call **fork()**.

- Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process's address space.

- Other address spaces created by **fork()** calls will not be affected even though they have identical variable names.

- **printf()** is "buffered," meaning **printf()** will group the output of a process together.
- While buffering the output for the parent process, the child may also use **printf** to print out some information, which will also be buffered. As a result, since the output will not be send to screen immediately, you may not get the right order of the expected result.
- The output from the two processes may be mixed in strange ways. To overcome this problem, you may consider to use the "unbuffered" **write**.



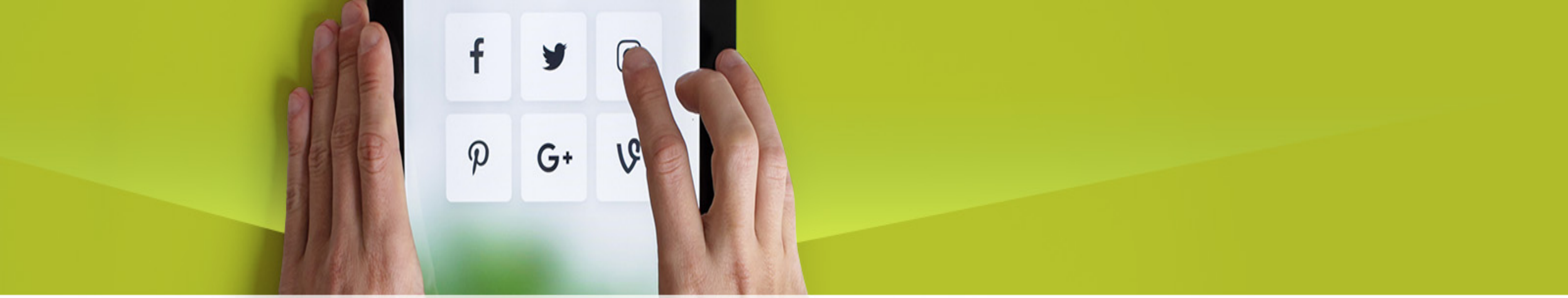
If you run this program, you might see the following on the screen:

```
     ................
This line is from pid 3456, value 13
This line is from pid 3456, value 14
     ................
This line is from pid 3456, value 20
This line is from pid 4617, value 100
This line is from pid 4617, value 101
     ................
This line is from pid 3456, value 21
This line is from pid 3456, value 22
     ................
```

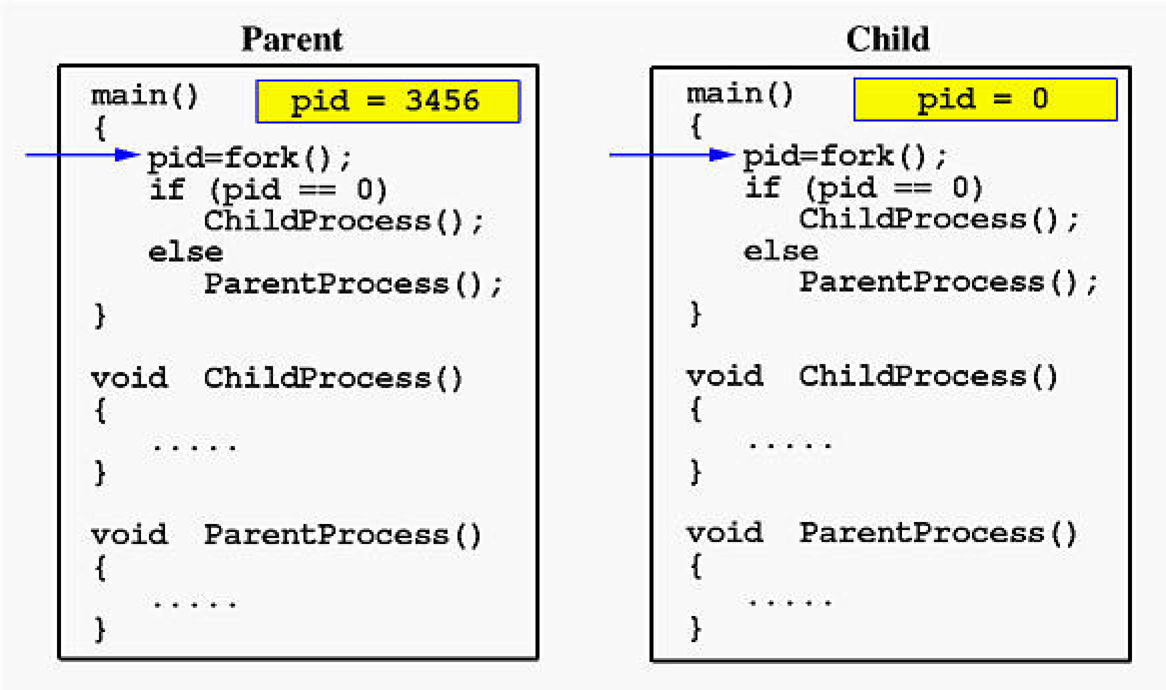
Process ID 3456 may be the one assigned to the parent or the child. Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way. Moreover, the order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result.

```
/* ----------------------------------------------------------------- */
/* PROGRAM fork-02.c                                    */
/*   This program runs two processes, a parent and a child.
Both of */
/* them run the same loop printing some messages.  Note
that printf()*/
/* is used in this program.                             */
/* ----------------------------------------------------------------- */

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

#define   MAX_COUNT  200

void  ChildProcess(void);              /* child process prototype
*/
void  ParentProcess(void);             /* parent process
prototype */

void  main(void)
{
    pid_t  pid;

    pid = fork();
    if (pid == 0)
        ChildProcess();
    else
        ParentProcess();
}
```

```
void  ChildProcess(void)
{
    int   i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("   This line is from child, value = %d\n",
i);
    printf("   *** Child process is done ***\n");
}

void  ParentProcess(void)
{
    int   i;

    for (i = 1; i <= MAX_COUNT; i++)
        printf("This line is from parent, value = %d\n",
i);
    printf("*** Parent is done ***\n");
}
```



- When the main program executes **fork()**, an identical copy of its address space, including the program and all data, is created.
- System call **fork()** returns the child process ID to the parent and returns 0 to the child process.
- The following figure shows that in both address spaces there is a variable **pid**. The one in the parent receives the child's process ID 3456 and the one in the child receives 0.

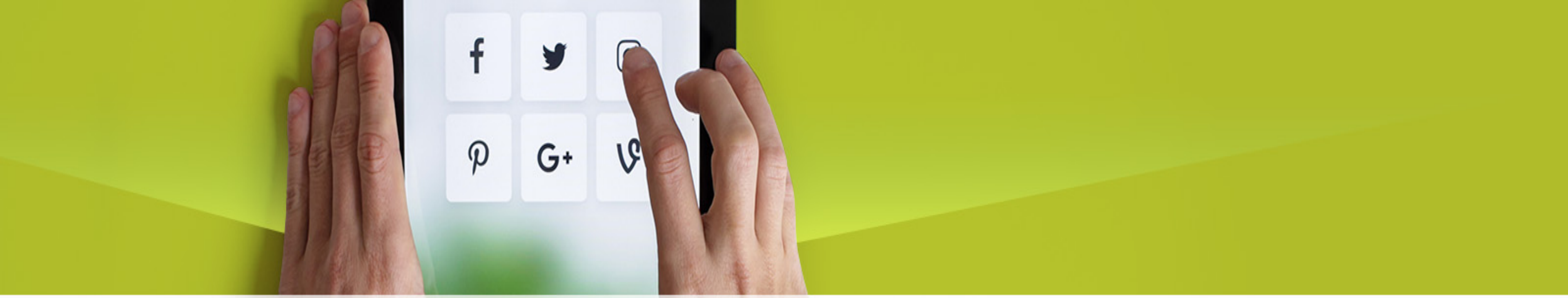


Parent

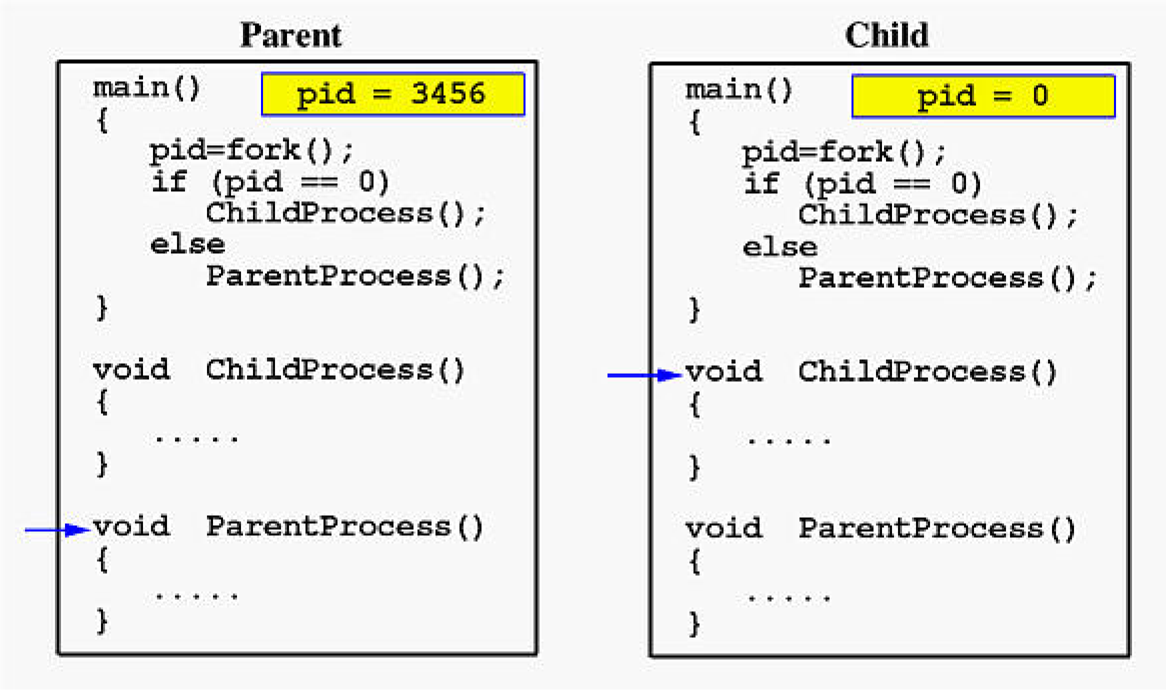
```
main()             pid = 3456
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

void  ChildProcess()
{
   .....
}

void  ParentProcess()
{
   .....
}
```

Child

```
main()             pid = 0
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

void  ChildProcess()
{
   .....
}

void  ParentProcess()
{
   .....
}
```



- In the parent, since **pid** is non-zero, it calls function **ParentProcess()**. On the other hand, the child has a zero **pid** and calls **ChildProcess()** as shown below:

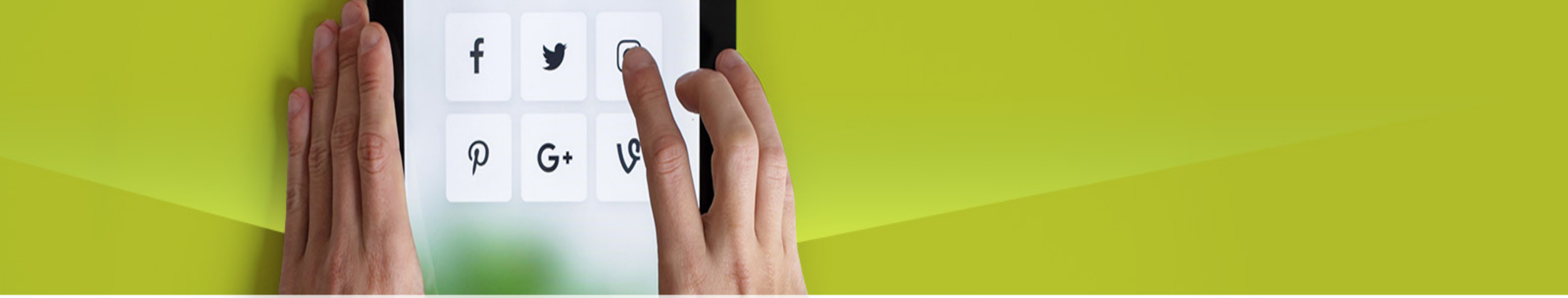


Parent

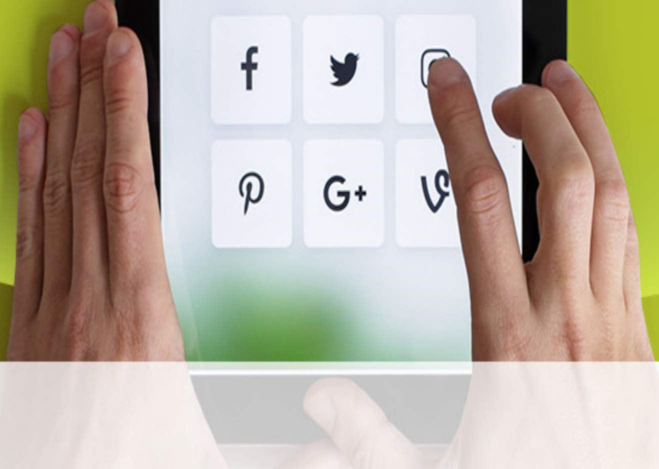
```
main()            pid = 3456
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

void  ChildProcess()
{
   .....
}

void  ParentProcess()
{
   .....
}
```

Child

```
main()            pid = 0
{
   pid=fork();
   if (pid == 0)
      ChildProcess();
   else
      ParentProcess();
}

void  ChildProcess()
{
   .....
}

void  ParentProcess()
{
   .....
}
```

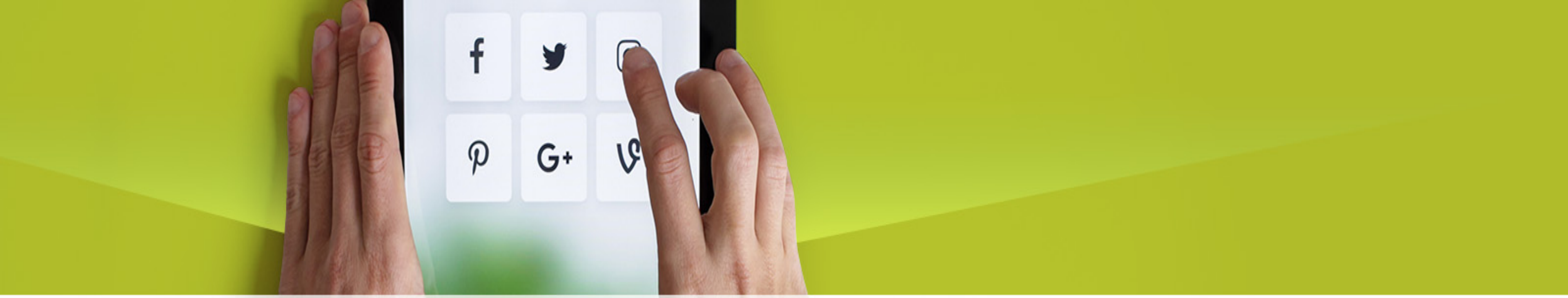


- Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process.
- Therefore, the value of **MAX_COUNT** should be large enough so that both processes will run for at least two or more time quanta.
- If the value of **MAX_COUNT** is so small that a process can finish in one time quantum, you will see two groups of lines, each of which contains all lines printed by the same process.



# Points to Note

- How to distinguish parent and child??
  - Return value in child = 0
  - Return value in parent = process id of child
- What about the data in the program?
- Return value of -1 indicates error in all UNIX system calls – another UNIX convention
- Is it true: All processes are created by fork() in UNIX?



- Unix has evolved: fork followed by exec is no longer the only way to run a program.
- vfork was created to be a more efficient fork for the case where the new process intends to do an exec right after the fork.
  After doing a vfork, the parent and child processes share the same data space, and the parent process is suspended until the child process either execs a program or exits.
- posix_spawn creates a new process and executes a file in a single system call. It takes a bunch of parameters that let you selectively share the caller's open files and copy its signal disposition and other attributes to the new process.

# Running an existing command in a program – exec()

- `int execl(char * pathname, char * arg0, … , (char *)0);`
- `int execv(char * pathname, char * argv[]);`
- `int execle(char * pathname, char * arg0, … , (char *)0, char envp[]);`
- `int execve(char * pathname, char * argv[], char envp[]);`
- `int execlp(char * filename, char * arg0, … , (char *)0);`
- `int execvp(char * filename, char * argv[]);`

# execv

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

Example: to run "/bin/ls –l –a /"

pathname: file path for the executable

char *argv[]: must be exactly the same as the C/C++ commanc line argument.

# Execute a Program: the execvp() System Call

The created child process does not have to run the same program as the parent process does.

The **exec** type system calls allow a process to run any program files, which include a binary executable or a shell script

The **execvp()** system call requires two arguments:

1.The first argument is a character string that contains the name of a file to be executed.

2.The second argument is a pointer to an array of character strings. More precisely, its type is **char ****, which is exactly identical to the **argv** array used in the main program:int main(int argc, char **argv)

When **execvp()** is executed, the program file given by the first argument will be loaded into the caller's address space and **over-write** the program there.

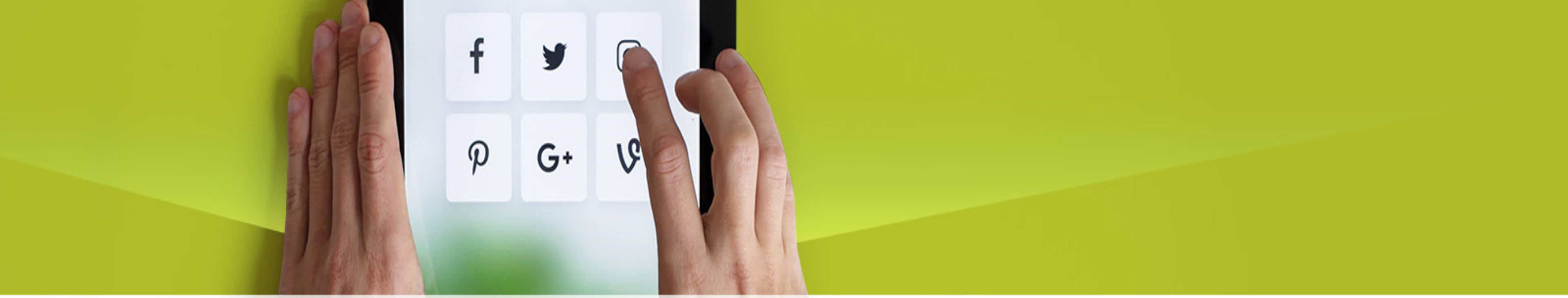
Then, the second argument will be provided to the program and starts the execution.

As a result, once the specified program file starts its execution, the original program in the caller's address space is gone and is replaced by the new program.

**execvp()** returns a negative value if the execution fails

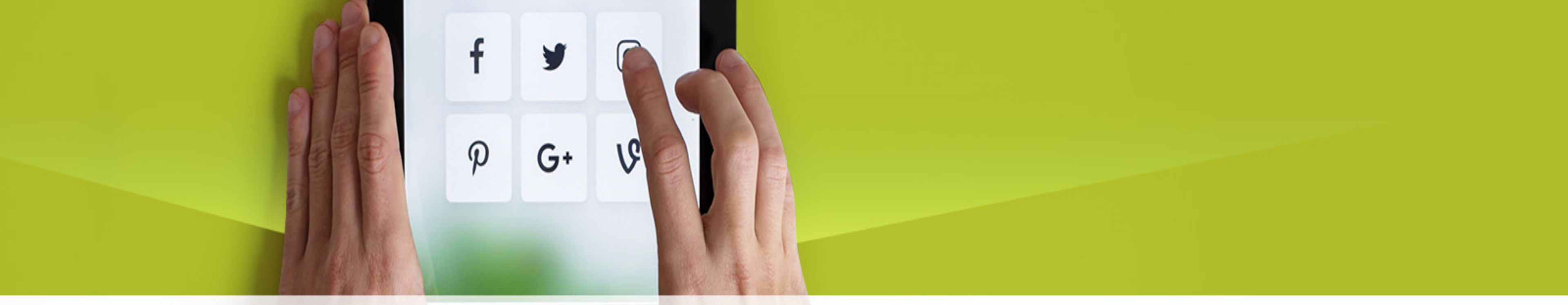
# Properties of exec ( )

- Replaces current process image with new program image.
  - E.g. parent image replaced by the new program image.
  - If successful, everything after the exec() call will NOT be executed.
    - Will execv() return anything other than -1?



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

void  parse(char *line, char **argv)
{
     while (*line != '\0') {       /* if not the end of line ....... */
          while (*line == ' ' || *line == '\t' || *line == '\n')
               *line++ = '\0';     /* replace white spaces with 0    */
          *argv++ = line;          /* save the argument position     */
          while (*line != '\0' && *line != ' ' &&
                 *line != '\t' && *line != '\n')
               line++;             /* skip the argument until ...    */
     }
     *argv = '\0';                 /* mark the end of argument list  */
}
```

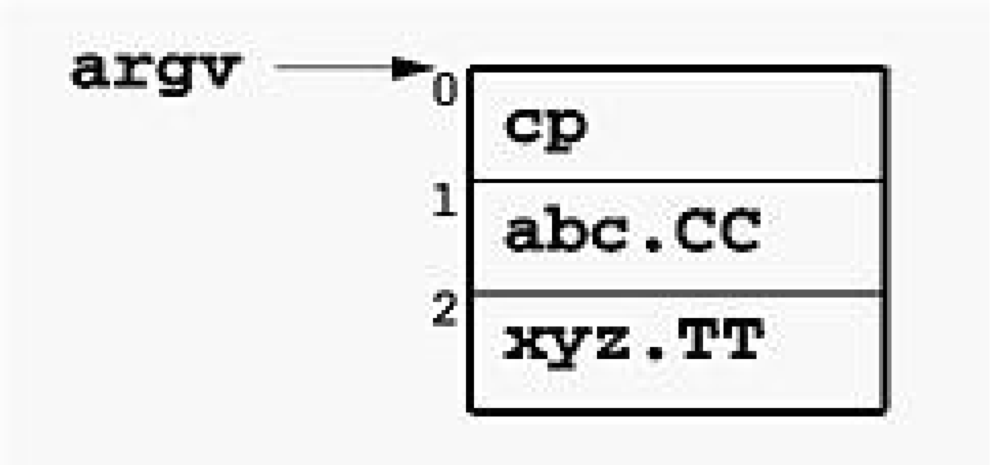
Function **parse()** takes an input line and returns a zero-terminated array of **char** pointers, each of which points to a zero-terminated character string. This function loops until a binary zero is found, which means the end of the input line **line** is reached. If the current character of **line** is not a binary zero, **parse()** skips all white spaces and replaces them with binary zeros so that a string is effectively terminated.
Once **parse()** finds a non-white space, the address of that location is saved to the current position of **argv** and the index is advanced. Then, **parse()** skips all non-whitespace characters.
This process repeats until the end of string **line** is reached and at that moment **argv** is terminated with a zero.

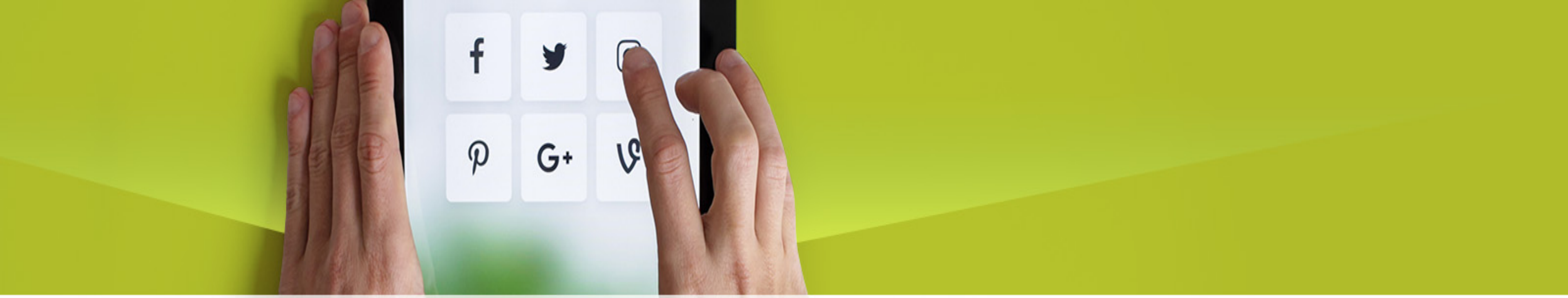


For example, if the input line is a string as follows:

"cp abc.CC xyz.TT"

Function **parse()** will return array **argv[]** with the following content:



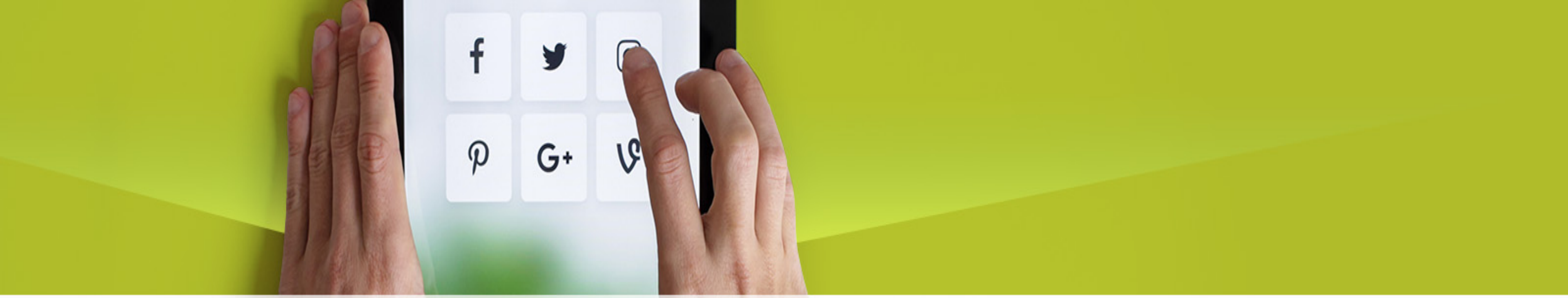


```
void  execute(char **argv)
{
    pid_t  pid;
    int    status;

    if ((pid = fork()) < 0) {     /* fork a child process           */
        printf("*** ERROR: forking child process failed\n");
        exit(1);
    }
    else if (pid == 0) {          /* for the child process:         */
        if (execvp(*argv, argv) < 0) {     /* execute the command  */
            printf("*** ERROR: exec failed\n");
            exit(1);
        }
    }
    else {                                  /* for the parent:      */
        while (wait(&status) != pid)       /* wait for completion  */
            ;
    }
}
```

Function **execute()** takes array **argv[]**, treats it as a command line arguments with the program name in **argv[0]**, forks a child process, and executes the indicated program in that child process.

While the child process is executing the command, the parent executes a **wait()**, waiting for the completion of the child.

In this special case, the parent knows the child's process ID and therefore is able to wait a specific child to complete.



```
void  main(void)
{
    char  line[1024];            /* the input line                */
    char  *argv[64];             /* the command line argument
*/

    while (1) {                /* repeat until done ....        */
        printf("Shell -> ");     /*   display a prompt             */
        gets(line);            /*   read in the command line     */
        printf("\n");
        parse(line, argv);       /*   parse the line               */
        if (strcmp(argv[0], "exit") == 0)  /* is it an "exit"?     */
            exit(0);           /*   exit if it is                */
        execute(argv);          /* otherwise, execute the
command */
    }
}
```

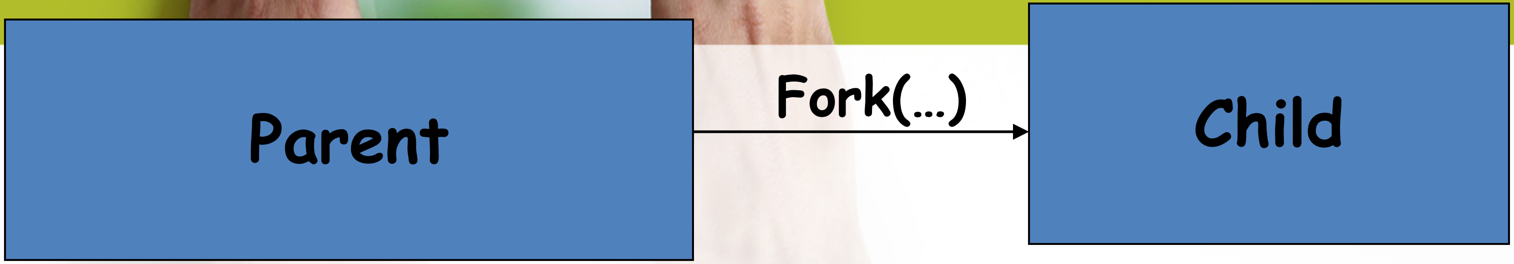
It prints out a command prompt, reads in a line, parses it using function **parse()**, and determines if the name is **"exit"**. If it is **"exit"**, use **exit()** to terminate the execution of this program; otherwise, the main uses **execute()** to execute the command.

# Running a command without killing the process

# Running a command without killing the process

Parent

# Running a command without killing the process?

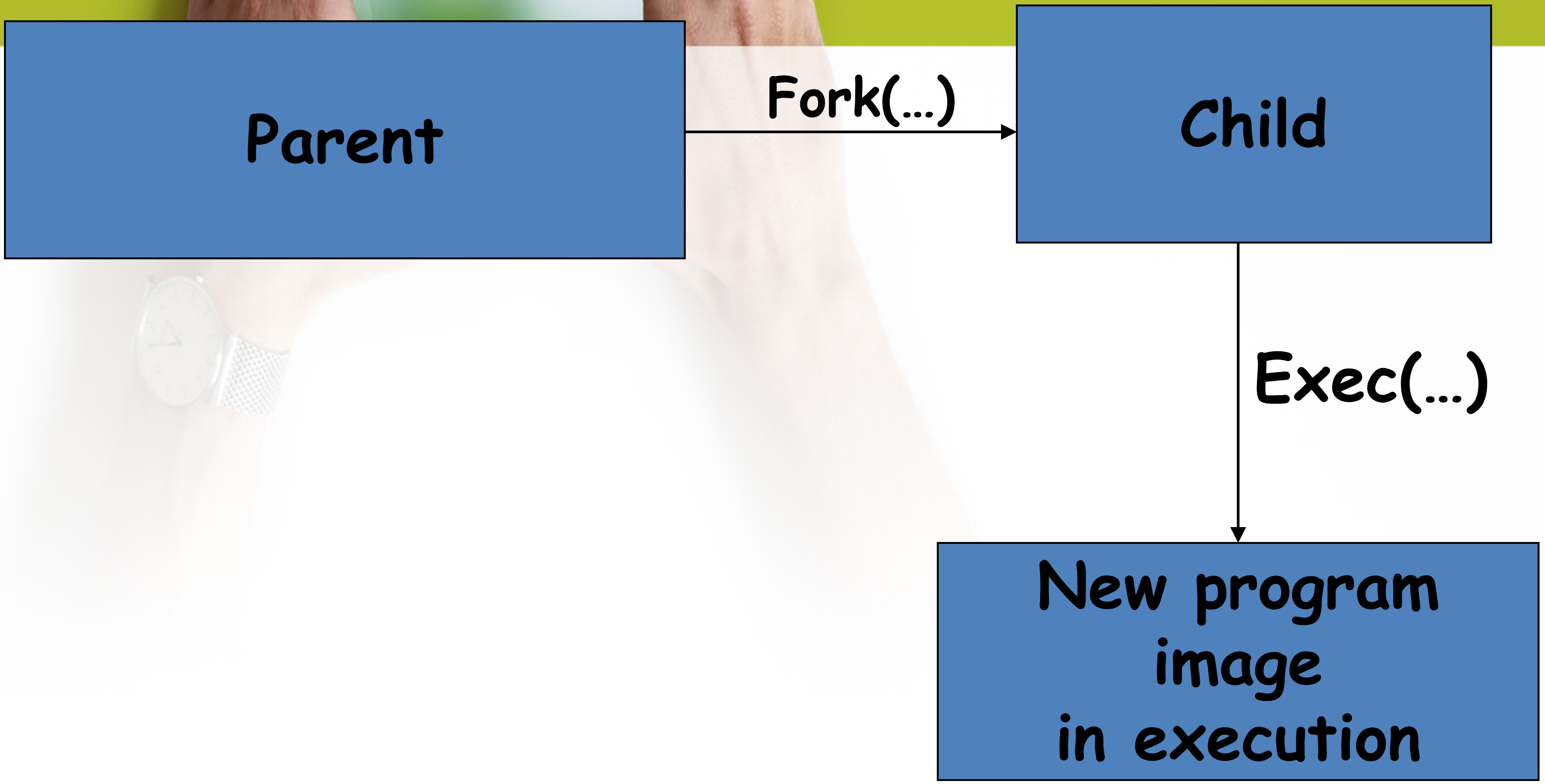


Parent

Fork(…)

Child

# Running a command without killing the process,



Parent

Fork(…)

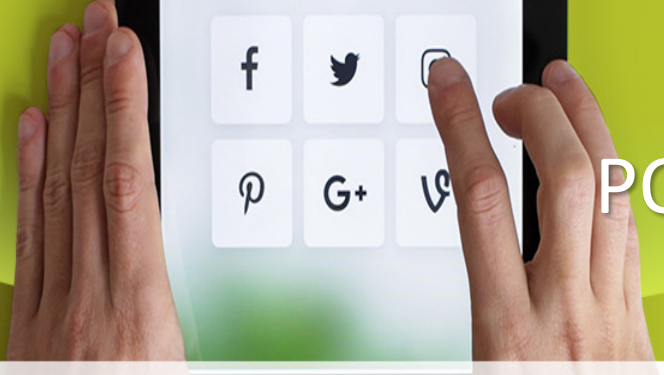
Child

Exec(…)

New program image in execution

# Terminating a process

- exit (int status)
  - Clean up the process (e.g close all files)
  - Tell its parent processes that he is dying (SIGCHLD)
  - Tell child processes that he is dying (SIGHUP)
  - Exit status can be accessed by the parent process.
- When a process exits – not all resources associated with the process are freed yet!!
  - ps can still see the process (<defunct>),



# POSIX-Defined Signals (1)

* SIGALRM: Alarm timer time-out. Generated by ***alarm( )*** API.
* SIGABRT: Abort process execution. Generated by ***abort( )*** API.
* SIGFPE: Illegal mathematical operation.
* SIGHUP: Controlling terminal hang-up.
* SIGILL: Execution of an illegal machine instruction.
* SIGINT: Process interruption.  Can be generated by *<Delete> or <ctrl_C>* keys.
* SIGKILL: Sure kill a process. Can be generated by *"**kill** -9 <process_id>"* command.
* SIGPIPE: Illegal write to a pipe.
* SIGQUIT: Process quit. Generated by *<crtl_\>* keys.
* SIGSEGV: Segmentation fault. generated by de-referencing a NULL pointer.

# POSIX-Defined Signals (2)

* SIGTERM: process termination. Can be generated by *“**kill** <process_id>”* command.
* SIGUSR1: Reserved to be defined by user.
* SIGUSR2: Reserved to be defined by user.
* SIGCHLD: Sent to a parent process when its child process has terminated.
* SIGCONT: Resume execution of a stopped process.
* SIGSTOP: Stop a process execution.
* SIGTTIN: Stop a background process when it tries to read from from its controlling terminal.
* SIGTSTP: Stop a process execution by the control_Z keys.
* SIGTTOUT: Stop a background process when it tries to write to its controlling terminal.

# Signaling Processes

* Signal

A signal is a notification to a process that an event has occurred.  Signals are sometimes called “software interrupts”.

* Features of Signal

- Signal usually occur asynchronously.
- The process does not know ahead of time exactly when a signal will occur.
- Signal can be sent by one process to another process (or to itself) or by the kernel to a process.

# Sources for Generating Signals

* Hardware

- A process attempts to access addresses outside its own address space.

- Divides by zero.

* Kernel

- Notifying the process that an I/O device for which it has been waiting is available.

* Other Processes

- A child process notifying its parent process that it has terminated.

* User

- Pressing keyboard sequences that generate a quit, interrupt or stop signal.

# Three Courses of Action

Process that receives a signal can take one of three action:

* Perform the system-specified default for the signal

  - notify the parent process that it is terminating;

  - generate a core file;

    (a file containing the current memory image of the process)

  - terminate.

* Ignore the signal

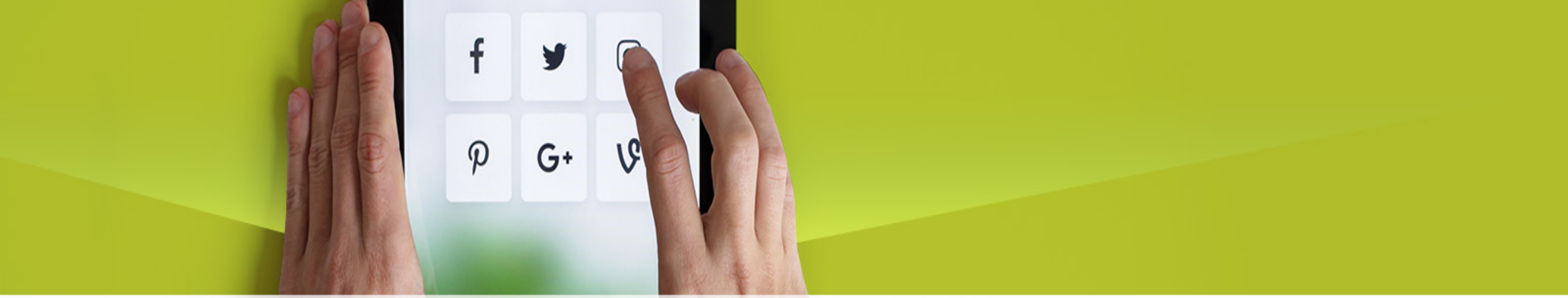
  A process can do ignoring with all signal but two special signals: SIGSTOP and SIGKILL.

* Catch the Signal

  When a process catches a signal, except SIGSTOP and SIGKILL, it invokes a special signal handing routine.

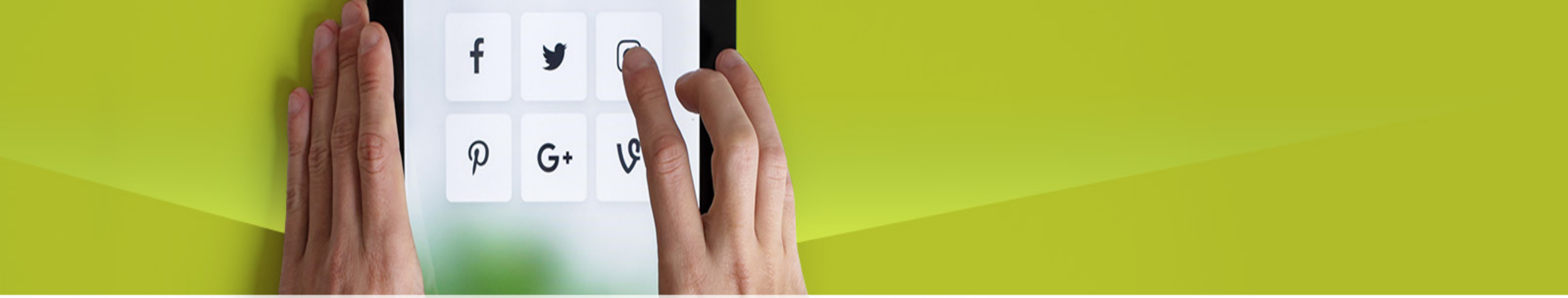
# Parent/child synchronization

- Parent created the child, he has the responsibility to see it through:
  - check if the child is done.
    - wait, waitpid
      - This will clean up all trace of the child process from the system.
  - check if the exit status of the child
    - pid_t wait(int *stat_loc),
  - Some others such as whether the child was killed by an signal. etc
- A child has no responsibility for the parent



- Processes are identified by a process id (pid)
  - getpid(): find your own pid
  - getppid(): find the pid of the parent

- A question: How to implement the *system* routine?



- In higher-level computer languages, many commonly-needed routines are prepackaged as function , which are routines with specified programming interfaces.
- Some functions can be compiled in line with other code. Other functions are compiled in as stub that make dynamic calls for system services during program execution.
- Functions are sometimes called *library routines* . The compiler and a set of library routines usually come as part of a related software development package.

# STUB

- A stub is a small program routine that substitutes for a longer program, possibly to be loaded later or that is located remotely. For example, a program that uses Remote Procedure Calls ( RPC ) is compiled with stubs that substitute for the program that provides a requested procedure. The stub accepts the request and then forwards it (through another program) to the remote procedure. When that procedure has completed its service, it returns the results or other status to the stub which passes it back to the program that made the request.

# Review

- Why processes?
- What is process context?
- How to check processes in UNIX?
- What does fork() do? What is its return value?
- Does fork() create all processes in a UNIX system?
- What does execv() do? What is its return value?
- How to run a command in a program without getting killed?
- Does exit completely clean up a process?
- Can a parent process tell if its child process terminate normally?
- Can a child process tell if its parent process terminate normally?