Processes
Threads
Interprocess Communication
Processes – Defined

- The process is the OS’s abstraction for execution
  - the unit of execution
  - a unit of scheduling
  - the dynamic execution context

- Process is often called a **job**, **task**, or **sequential process**
Processes – Contents

- A process in Unix or Windows comprises (at least):
  - an *address space* – usually protected and virtual – mapped into memory
  - the *code* for the running program
  - the *data* for the running program
  - an *execution stack* and *stack pointer* (SP)
  - the *program counter* (PC)
  - a set of processor *registers* – general purpose and status
  - a set of system *resources*
    - files, network connections, privileges, …
Processes – Address Space

Virtual address space

0xFFFFFFFF

0x00000000

stack (dynamically allocated)

heap (dynamically allocated)

static data

program code (text)

PC

SP
Processes in the OS – Representation

- A process is identified by its *Process ID* (PID)
- In the OS, processes are represented by entries in a *Process Table* (PT)
  - PID “points to” a PT entry
  - PT entry = *Process Control Block* (PCB)
- PCB is a large data structure that contains or points to all info about the process
Typical PCB contains:

- execution state
- PC, SP & processor registers – stored when process is made inactive
- memory management info
- Privileges and owner info
- scheduling priority
- resource info
- accounting info
Process – starting and ending

- Processes are created …
  - When the system boots
  - By another process
  - By user
  - By batch manager

- Processes terminate when …
  - Normally – exit
  - Voluntarily on an error
  - Involuntarily on an error
  - Terminated (killed) by the actions a user or a process
Processes – Switching

- When a process is running, its *hardware state* is in the CPU – PC, SP, processor registers
- When the OS suspends running a process, it saves the hardware state in the PCB
- *Context switch* is the act of switching the CPU from one process to another
  - timesharing systems may do 100s or 1000s of switches/sec
  - takes 1-100 microseconds on today’s hardware
States

- **Process has an execution state**
  - *ready*: waiting to be assigned to CPU
  - *running*: executing on the CPU
  - *waiting*: waiting for an event, e.g. I/O

![Process State Diagram]

- New
- Ready
- Dispatch
- Interrupt
- I/O Complete
- Waiting
- I/O Wait
- Running
- Exit
State Queues

- The OS maintains a collection of process state queues
  - typically one queue for each state – e.g., ready, waiting, ...
  - each PCB is put onto a state queue according to its current state
  - as a process changes state, its PCB is unlinked from one queue, and linked to another
- Process state and the queues change in response to events – interrupts, traps
Process Creation

- Unix/Linux
  - Create a new (child) process – *fork*();
    - Allocates new PCB
    - Clones the calling process (almost)
      - Copy of parent process address space
      - Copies resources in kernel (e.g. files)
    - Places PCB on Ready queue
    - Return from *fork*() call
      - 0 for child
      - child PID for parent
Processes – Address Space

Virtual address space

Kernel Space

User Space

0xFFFFF

Kernel Code and Data
- stack (dynamically allocated)
- heap (dynamically allocated)
- static data
- code (text)

0x00000000

SP

PC
Example of fork()

```c
int main(int argc, char **argv)
{
    char *name = argv[0];
    int child_pid = fork();
    if (child_pid == 0) {
        printf("Child of %s is %d\n", name, child_pid);
        return 0;
    } else {
        printf("My child is %d\n", child_pid);
        return 0;
    }
}

% ./forktest
Child of forktest is 0
My child is 486
```
Another example

#include <stdio.h>

main(int argc, char *argv[])
/* argc -- number of arguments */
/* argv -- an array of strings */
{
    int pid;
    int i;

    /* print out the arguments */
    printf("There are %d arguments:\n", argc);
    for (i = 0; i < argc; i++)
        printf("%s\n", argv[i]);

    if ((pid = fork()) < 0) {
        fprintf(stderr, "Fork error\n");
        exit(1);
    }
    else if (pid == 0) { /* child process */
        for (i = 0; i < 5; i++)
            printf("child (%d) : %s\n", getpid(), argv[2]);
        exit(0);
    }
    else {
        /* parent */
        for (i = 0; i < 5; i++)
            printf("parent (%d): %s\n", getpid(), argv[1]);
        exit(0);
    }
}
> gcc -o ptest ptest.c
> ./ptest x y

output:
There are 3 arguments:
ptest
x
y
parent (690): x
parent (690): x
child (7686) : y
parent (690): x
child (7686) : y
parent (690): x
child (7686) : y
parent (690): x
child (7686) : y
child (7686) : y
New Programs

- Starting another program
  - Unix – **int exec (char *prog, char **argv)**
    - Check privileges and file type
    - Loads program “prog” into address space
    - Initializes context – e.g. passes arguments (*argv)
    - Place PCB on **ready** queue
  - Windows/NT – combines *fork & exec*
    - **CreateProcess**(10 arguments)
    - Not a parent child relationship
    - Note – privileges required to create a new process
execve

- `execve(name, argv, envp)`:
- **name**
  - -- name of the file to execute.
- **argv**
  - NULL-terminated array of pointers to NULL-terminated character strings.
- **envp**
  - NULL-terminated array of pointers to NULL-terminated strings. Used to pass environment information to the new process.
process execution

- a process first starts up
  - started via `exec`

- After startup the C library:
  - makes the arguments passed to `exec` available as arguments to the main procedure in the new process.
  - places a copy of `envp` in the global variable `environ`. 
Process Creation

- #include <sys/types.h>
- #include <unistd.h>

pid_t fork(void);

int execve (const char *filename,
            char *const argv[],
            char *const envp[]);
int main(int argc, char **argv)
{
    char *argvNew[5];
    int pid;
    if ((pid = fork()) < 0) {
        printf( "Fork error
");
        exit(1);
    } else if (pid == 0) { /* child process */
        argvNew[0] = "/bin/ls";
        argvNew[1] = "-l";
        if (execve(argvNew[0], argvNew, environ) < 0) {
            printf( "Execve error
");
            exit(1);
        }
    } else { /* parent */
        wait(pid); /* wait for the child to finish */
    }
}

Returns only in error condition
utility functions

- `execl(name, arg0, arg1, arg2, ..., 0)`
  - used when the arguments are known in advance.
  - 0 terminates the argument list.

- `execv(name, argv)`
  - argv is the same for `execve`.

- `execvp(name, argv)`
  - argv is the same as for `execve`.
  - executable file is searched for in the path

- eventually `execve` will be called
  - global variable `environ` in place of the `envp` argument

- child processes `inherits` the parent's environment.
basic shell like application

while (1) {
    type_prompt(); /* show prompt */
    read_command(command,parameters); /* get input */
    if (fork != 0) {
        /* Parent code */
        waitpid(-1,&status,0);
    } else {
        /* child code */
        execve(command,parameters,0);
    }
}
Synchronization
Interprocess Communication (IPC)
Interprocess Communication

- Mechanism for processes to communicate and to synchronize their actions.
Interprocess Communication

- **Types**
  - Pipes & streams
  - Sockets & Messages
  - Remote Procedure Call
  - Shared memory

- **OS dependent**

- Depends on whether the communicating processes share all or part of an address space
Common IPC mechanisms

- *shared memory* – read/write to shared region
  - E.g., `shmget()`, `shmctl()` in Unix
  - Memory mapped files in WinNT/2000
  - Need critical section management

- *semaphores*
  - `post_s()` notifies *waiting* process
  - Shared memory or not

- *software interrupts* - process notified asynchronously
  - `signal()`

- *pipes* - unidirectional stream communication

- *message passing* - processes send and receive messages
  - Across address spaces
Software Interrupts

- Similar to hardware interrupt.
  - Processes interrupt each other
  - Non-process activities interrupt processes
- Asynchronous
- Stops execution then restarts
  - Keyboard driven – e.g. cntl-C
  - An alarm scheduled by the process expires
    - Unix: SIGALRM from `alarm()` or `settimer()`
  - resource limit exceeded (disk quota, CPU time...)
  - programming errors: invalid data, divide by zero
Software Interrupts (continued)

- **SendInterrupt**(pid, num)
  - Send signal type num to process pid,
  - `kill()` in Unix
  - (NT doesn’t allow signals to processes)

- **HandleInterrupt**(num, handler)
  - type num, use function handler
  - `signal()` in Unix
  - Use exception handler in WinNT/2000

- **Typical handlers:**
  - ignore
  - terminate (maybe w/core dump)
  - user-defined
A pipe is a unidirectional stream connection between 2 processes

- Unix/Linux
  - 2 file descriptors
  - Byte stream

- Win/NT
  - 1 handle
  - Byte stream and structured (messages)
(Named) Pipes

- Classic IPC method under UNIX:
  
  ```bash
  > ls -l | more
  ```

  - shell runs two processes `ls` and `more` which are linked via a pipe
  - the first process (`ls`) writes data (e.g., using `write`) to the pipe and
    the second (`more`) reads data (e.g., using `read`) from the pipe

- the system call `pipe( fd[2] )`
  creates one file descriptor for reading (`fd[0]`) and one for writing (`fd[1]`)
  - allocates memory page to hold data
Pipe Example

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

char *msg = "Hello Pipe!";

main()
{
    char inbuf[MSGSIZE];
    int p[2];
    pid_t pid;

    /* open pipe */
    if (pipe(p) == -1) { perror("pipe call error"); exit(1); }

    switch( pid = fork() ) {
        case -1: perror("error: fork call"); exit(2); 
        case 0: close(p[0]); /* close the read end of the pipe */
               write(p[1], msg, MSGSIZE);
               printf("Child: %s\n", msg);
               break;
        default: close(p[1]); /* close the write end of the pipe */
                 read(p[0], inbuf, MSGSIZE);
                 printf("Parent: %s\n", inbuf);
                 wait(0);
    }
    exit(0);
}
```
creating file descriptors

- `$ exec 3< file1`
  - creates a file descriptor called 3
  - 3 is descriptor for file called file1

- **Standard descriptors**
  - 0 – read
  - 1 – write
  - 2 – error

- `$ read <&3 var1`
  - reads from file with descriptor 3
  - result is places in var1
  - echo $var1
```bash
#!/bin/sh
#process a file line by line

if [ $# != 1 ]; then
    echo "Usage: $0 input-file"
echo " "
else
    processfile=$1
fi

# assign file descriptor 3 to file
exec 3< $processfile

#read from file through descriptor

until [ $done ]; do
    read <&3 out
    if [ $? != 0 ]; then
        done=1
        continue
    fi

    # process file
    echo $out
done
echo " That is al folks!"
```
file1:  
date  
users  
pwd  
ls  

uskudarli@uskudarli:~/code/shell$
  ./file_descriptor_read_write file1
date
users
pwd
ls
That is all folks!!!
IPC – Message Passing

- Communicate information from one process to another via primitives:
  
  ```
  send(dest, &message)
  receive(source, &message)
  ```

- Receiver can specify **ANY**

- Receiver can choose to **block** or not
Message Passing

```c
void Producer() {
    while (TRUE) {
        /* produce */
        build_message(&m, item);
        /* send message */
        send(consumer, &m);
        /* wait for ack */
        receive(consumer, &m);
    }
}
```

```c
void Consumer {
    while (TRUE) {
        receive(producer, &m);
        /* receive message */
        extract_item(&m, &item);
        /* send ack */
        send(producer, &m);
        /* consume item */
    }
}
```
send

send ( ) operation

- Synchronous
  - Returns after data is sent
  - Blocks if buffer is full

- Asynchronous
  - Returns as soon as I/O started
  - Done?
    - Explicit check
    - Signal
  - Blocks if buffer is full
receive

receive () operation

- **Syncronous**
  - Returns if there is a message
  - Blocks if not

- **Asyncronous**
  - Returns if there is a message
  - Returns indication if no message
Mailbox

- Indirect Communication – mailboxes
  - Messages are sent to a named area – *mailbox*
  - Processes read messages from the mailbox
  - Mailbox must be created and managed
  - Sender blocks if mailbox is full
  - Enables many-to-many communication
Message Passing issues

- Scrambled messages (checksum)
- Lost messages (acknowledgements)
- Lost acknowledgements (sequence no.)
- Process unreachable (down, terminates)
- Naming
- Authentication
- Performance (copying, message building)
Buffering/Queuing

- Queue of messages attached to the link
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous).
  2. Bounded capacity – finite length of $n$ messages
     Sender must wait if link full.
  3. Unbounded capacity – infinite length
     Sender never waits.
Message Passing (2)

- Message passing may be:
  - **Blocking**
    - synchronous.
  - **Non-blocking**
    - asynchronous.
- **send** and **receive** primitives may be either blocking or non-blocking.
Synchronization in message passing (1)

- For the sender
  - convenient to not to be blocked after send
  - send several messages to multiple destinations.
  - usually expects acknowledgment of message receipt

- For the receiver
  - it is more natural to be blocked after issuing receive:
    - the receiver usually needs the info before proceeding.
    - but could be blocked indefinitely if sender process fails before send.
Synchronization in message passing (2)

- Other alternatives
  - blocking send and blocking receive:
    - both are blocked until the message is received.
    - when the communication link is unbuffered (no message queue).
  - tight synchronization (*rendezvous*).
Synchronization in message passing (3)

- 3 meaningful combinations:
  1. Blocking send, Blocking receive
  2. Nonblocking send, Nonblocking receive
  3. Nonblocking send, Blocking receive

3rd is most popular
Messages and Pipes Compared

**Message queue**

P1 → **Send** → Message queue → **Receive** → P2

Fixed-size messages

**Pipe buffer**

P1 → **Write** → Pipe buffer → **Read** → P2

Read and write in any sizes
### Some Signals

<table>
<thead>
<tr>
<th>Signal</th>
<th>Number</th>
<th>Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>1</td>
<td>Exit</td>
<td>Hangup</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Exit</td>
<td>Interrupt</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>3</td>
<td>Core</td>
<td>Quit</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Core</td>
<td>Illegal Instruction</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>5</td>
<td>Core</td>
<td>Trace/Breakpoint Trap</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Core</td>
<td>Abort</td>
</tr>
<tr>
<td>SIGEMT</td>
<td>7</td>
<td>Core</td>
<td>Emulation Trap</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Core</td>
<td>Arithmetic Exception</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Exit</td>
<td>Killed</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>10</td>
<td>Core</td>
<td>Bus Error</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>Core</td>
<td>Segmentation Fault</td>
</tr>
<tr>
<td>SIGSYS</td>
<td>12</td>
<td>Core</td>
<td>Bad System Call</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>13</td>
<td>Exit</td>
<td>Broken Pipe</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>14</td>
<td>Exit</td>
<td>Alarm Clock</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>15</td>
<td>Exit</td>
<td>Terminated</td>
</tr>
</tbody>
</table>
```c
#include <stdio.h>
#include <signal.h>

void sigproc() {
    signal(SIGINT, sigproc);
    printf("you have pressed ctrl-c - disabled \n");
}

void quitproc() {
    printf("ctrl-\ pressed to quit\n");
    exit(0); /* normal exit status */
}

main() {
    signal(SIGINT, sigproc);
    /* DEFAULT ACTION: term */
    signal(SIGQUIT, quitproc);
    /* DEFAULT ACTION: term */
    printf("ctrl-c disabled use ctrl-\ to quit\n");
    for(;;);
}
```
#include <stdio.h>
#include <signal.h>

void sighup();
void sigint();
void sigquit();

main()
{
    int pid;

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

    if (pid == 0) /* child */
    {
        signal(SIGHUP, sighup);
        signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);
        for(;;);
    }
    else /* parent */
    {
        printf("\nPARENT: sending SIGHUP\n\n");
        kill(pid,SIGHUP);
        sleep(3);
        printf("\nPARENT: sending SIGINT\n\n");
        kill(pid,SIGINT);
        sleep(3);
        printf("\nPARENT: sending SIGQUIT\n\n");
        kill(pid,SIGQUIT);
        sleep(3);
    }
}
void sighup()
{
    signal(SIGHUP,sighup);
    /* reset signal */
    printf("CHILD: received SIGHUP\n");
}

void sigint()
{
    signal(SIGINT,sigint);
    /* reset signal */
    printf("CHILD: received SIGINT\n");
}

void sigquit()
{
    printf("Parent Killed me!!!\n");
    exit(0);
}
Summary

- Many ways to perform send messages or perform IPC on a machine
  - mailboxes - FIFO, messages has types
  - pipes – FIFO, no type
  - shared memory – shared memory mapped into virtual space
  - signals – send a signal which can invoke a special handler
Critical Sections

- Critical section of code involve shared access in a concurrent situation
- More than one process/thread must not enter
- Synchronization mechanisms must be used
Blocking

- Blocking is synchronous
- Non-blocking is asynchronous
Threads
Processes are very heavyweight

- Lots of data in process context
- Processor caches a lot of information
  - Memory Management information
- Costly context switches and traps
  - 100’s of microseconds
Processes are Heavyweight

- Separate processes have separate address spaces
  - Shared memory is limited or nonexistent
  - Applications with internal concurrency are difficult

- Isolation between independent processes vs. cooperating activities
  - Fundamentally different goals
Example

- **Web Server – How to support multiple concurrent requests**

  - **One solution:**
    - create several processes that execute in parallel
    - Use shared memory (`shmget()` ) to map to the *same* address space in the processes
    - have the OS schedule them in parallel

  - **Not efficient**
    - space: PCB, page tables, etc.
    - time: creating OS structures (`fork()` ) and context switch
Example 2

- Transaction processing systems
  - E.g, airline reservations or bank ATM transactions
- 1000’s of transactions *per second*
  - Very small computation per transaction
- Separate processes per transaction are too costly
Solution:– *Threads*

- A *thread* is the execution of a program or procedure within the context of a Unix or Windows process
  - I.e., a specialization of the concept of *process*
- A thread has its own
  - Program counter, registers
  - Stack
- A thread shares
  - Address space, heap, static data
  - All other resources

with other threads in the same process
Threads

Virtual address space

0xFFFFFFFF

SP

PC

0x00000000

thread 1 stack

thread 2 stack

thread 3 stack

heap

static data

code (text)
Thread Interface

- From POSIX pthreads API:
  - int `pthread_create(pthread_t *thread, const pthread_attr_t *attr, void (*)(*start_routine) (void), void *arg)`;
    - creates a new thread of control
    - new thread begins executing at start_routine
  - `pthread_exit(void *value_ptr)`
    - terminates the calling thread
  - `pthread_join(pthread_t thread, void **value_ptr)`;
    - blocks the calling thread until the thread specified terminates
  - `pthread_t pthread_self()`
    - Returns the calling thread's identifier
Threads

- Linux, Windows, and various versions of Unix have their own thread interfaces
  - Similar, not standardized

Some issues
- E.g., `ERRNO` in Unix — a static variable set by system calls
Threads – Management

- Who/what creates and manages threads?
  - *Kernel* level – new system calls and new entity to manage
    - Linux: *lightweight process*
    - Win/NT & XP: *threads*
  - *User* level
    - done with function library (POSIX)
    - Runtime system – similar to process management except in user space
    - Win/NT – *fibers*: x user-level thread mechanism
Threads – User Space

- **Thread Scheduler**
  - Queues to keep track of threads’ state
  - Scheduler – non-preemptive
    - Assume threads are **well-behaved**
    - Thread gives up CPU by calling yield() – does context switch to another thread
  - Scheduler – preemptive
    - Assumes threads may not be well-behaved
    - Scheduler sets timer to create a *signal* that invokes scheduler
    - Scheduler can force thread context switch
    - Increased overhead

- Application must handle *all* concurrency itself!
Threads inside the OS kernel

- Kernels have evolved into large, multi-threaded programs.
- Lots of concurrent activity
  - Multiple devices operating at one time
  - Multiple application activities at one time
Threads – Summary

- Processes are very heavyweight in Unix, Linux, Windows, etc.
  - Need for isolation between processes conflicts the need for concurrency within processes

- Threads provide an efficient alternative
  Thread implementation and management strategies depend upon expected usage
  - Kernel support or not
  - Processor support or not
Processes

- Processes are created in a hierarchical structure
- depth is limited by the virtual memory available to the virtual machine
- A process may control the execution of any of its descendants
  - suspend
  - resume
  - change relative priority
  - terminate
- Termination of a process causes termination of all its descendants
- Termination of the root process terminates the session
- Linux assigns a process ID (PID) to the process
Processes

- **Foreground**
  - a process runs in terminal
  - invoked from prompt
  - when process terminates it returns to prompt

- **Background**
  - process runs in the background
  - invoked with “&” at the end of the command line,
  - the prompt immediately returns
  - terminal is free to execute other commands
Processes

- **Daemons**
  - Background processes for system administration are referred to as “daemons”
  - These processes are usually started during the boot process
  - The processes are not assigned any terminals

<table>
<thead>
<tr>
<th>UID</th>
<th>PID</th>
<th>PPID</th>
<th>C</th>
<th>STIME</th>
<th>TTY</th>
<th>TIME</th>
<th>CMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1999</td>
<td>?</td>
<td>00:00:14</td>
<td>[kswapd]</td>
</tr>
<tr>
<td>bin</td>
<td>254</td>
<td>1</td>
<td>0</td>
<td>1999</td>
<td>?</td>
<td>00:00:00</td>
<td>[portmap]</td>
</tr>
<tr>
<td>root</td>
<td>307</td>
<td>1</td>
<td>0</td>
<td>1999</td>
<td>?</td>
<td>00:00:23</td>
<td>syslogd -m 0</td>
</tr>
<tr>
<td>root</td>
<td>350</td>
<td>1</td>
<td>0</td>
<td>1999</td>
<td>?</td>
<td>00:00:34</td>
<td>httpd</td>
</tr>
</tbody>
</table>
Processes

```
[root@penguinvm log]# sleep 10h &
[1] 6718
[root@penguinvm log]# ps -ef
UID        PID  PPID  C STIME TTY          TIME CMD
root      6718  6692  0 14:49 ttyp0    00:00:00 sleep 10h
```

- `&` causes the process to be run in "background"
- Job Number
- Process ID (ID)
- Parent Process ID
Processes - UID & GID

- Real UID
  - At process creation, the real UID identifies the user who has created the process

- Real GID
  - At process creation, the real GID identifies the current connect group of the user for which the process was created
Processes - UID & GID

- Effective UID
  - The effective UID is used to determine owner access privileges of a process.
  - Normally the same as the real UID. It is possible for a program to have a special flag set that, when this program is executed, changes the effective UID of the process to the UID of the owner of the program.
  - A program with this special flag set is said to be a set-user-ID program (SUID). This feature provides additional permissions to users while the SUID program is being executed.
Processes - UID & GID

- Effective GID
  - Each process also has an effective group
  - The effective GID is used to determine group access privileges of a process
  - Normally the same as the real GID. A program can have a special flag set that, when this program is executed, changes the effective GID of the process to the GID of the owner of this program
  - A program with this special flag set is said to be a set-group-ID program (SGID). Like the SUID feature, this provides additional permission to users while the set-group-ID program is being executed
Processes - Process Groups

- Each process belongs to a process group
- A *process group* is a collection of one or more processes
- Each process group has a unique process group ID
- It is possible to send a signal to every process in the group just by sending the signal to the process group leader
- Each time the shell creates a process to run an application, the process is placed into a new process group
- When an application spawns new processes, these are members of the same process group as the parent
Processes - PID

- PID
  - A process ID is a unique identifier assigned to a process while it runs
  - Each time you run a process, it has a different PID (it takes a long time for a PID to be reused by the system)
  - You can use the PID to track the status of a process with the `ps` command or the `jobs` command, or to end a process with the `kill` command
Processes - PGID

- **PGID**
  - Each process in a process group shares a process group ID (PGID), which is the same as the PID of the first process in the process group.
  - This ID is used for signaling-related processes.
  - If a command starts just one process, its PID and PGID are the same.
Processes - PPID

- **PPID**
  - A process that creates a new process is called a *parent process*; the new process is called a *child process*.
  - The parent process (PPID) becomes associated with the new child process when it is created.
  - The PPID is not used for job control.