Process Control functions (and more)

1. A summary of the main functions for process control

pcntl_fork - A PHP function that calls the fork system call that creates a new process by duplicating the calling process. The new process, referred to as the **child**, is an exact duplicate of the calling process (except for a few details, read more in the man page). **Both the newly created process and the parent process return from the call to fork**. On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure (e.g. if there is no more 'space' for new processes to be created), -1 is returned in the parent and no child process is created.
Refer to the Week 3 lecture slides to read the details for Copy On Write – the technique that the implementation of `fork()` uses to avoid unnecessary duplication of the calling process.

pcntl_wait - A class of PHP functions that call system calls that are used to wait for state changes in a child of the calling process and obtain information about the child whose state has changed. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal.

pcntl_exec – A PHP function that replaces the current process image with a new process image. The argument for this function must include the path to a binary executable or a script with a valid path pointing to an executable.

pcntl_signal – A PHP function that installs a new signal handler, or replaces the current signal handler for the signal indicated by signo (passed as an argument to the function).

2. Zombie processes

Unix terminology lends itself to some grim analogies (children who become zombies, kill, die all form part of Unix parlance). **A zombie process is essentially a child process that is dead (has terminated) but its death (its exit code) has not been acknowledged (collected) by its parent process.** As you would expect zombies to do, zombie processes still linger on after their death.

To understand the issue with zombie processes you need to know a bit about what happens with processes in Linux. When a process dies, it is not removed from memory immediately; it still has an entry in the process table (this only uses a small amount of memory). This happens so that the parent process can read the child's exit status. Once the exit status has been reaped (i.e. collected), the child's entry is completely removed from the process table. You should note that strictly speaking, a child process becomes a zombie in the very short period of time between its termination and the collection of the exit code by the parent via a wait(). But this happens for such a short period of time that is a non-issue.

Zombies become a problem when the parent has not issues a wait() to reap the child's exit status. If that is the case then the dead child processes retain their place in the process table until somehow their exit status is collected. So essentially the existence of zombie processes is due to program error (the parent does not have a wait() call to reap the

exit status of its children). Zombie processes will have a Status Z in the output of e.g. `ps` and `top` commands.

Why are zombies a problem? As we said zombie processes take up a small amount of memory in the process table, so although they do form a 'resource leak', they are not a significant one. The problem becomes when too many zombies linger on (zombie pandemic?), and it is not a memory problem. Process ids (PIds) in Unix-based systems have fixed upper limit (e.g. in 32-bit systems this is 32,767 – you can usually check by issuing a command such as: `cat /proc/sys/kernel/pid_max`). Zombies retain their pid as long as they are not reaped, so if you end up having too many around taking up many of the finite number of PIDs, you may find yourself in a position where the system can't create any more processes. However a few zombie processes left in a system are harmless, they just indicate a problem with the process that created them.

How to get rid of zombies? You can not kill zombie processes in the same way as you kill normal processes via the `kill` command (zombies are already dead after all, the analogies never end…). There are a few different ways of getting rid of zombies: You can explicitly send the SIGKILL signal to the parent process (e.g. `kill -s SIGKILL parent_pid`), but if the parent isn't programmed properly it can still refuse to take responsibility for its zombie children. The problem is solved by killing the parent (grim analogies continue…well ok, kill or simply close/end the parent process). What happens then is that the newly orphaned zombie children are adopted by the process `init` (who is the parent of all processes in Unix-based systems). init periodically executes the wait system call to reap zombies and so the problem will be solved. However it is important to fix a process that keeps creating zombie children (by ensuring that it properly issues a wait() to reap its zombie children).

3. Signals

A signal is a notification, a message sent by either the OS or some application to our process. Signals are a mechanism for one-way asynchronous notifications. A signal may be sent from the kernel to a process, from a process to another process, or from a process to itself. Signals typically alert a process to some event, such as a segmentation fault, or the user pressing Ctrl-C.

The Linux kernel implements about 30 signals. Each signal is identified by a number, from 1 to 31. Signals don't carry any argument and their names are mostly self-explanatory. For instance, **SIGKILL** or signal number **9**, tells the process that someone tries to kill it, and **SIGHUP** is used to signal that a terminal hangup has occurred, and it has a value of **1** on the i386 architecture.

With the exception of **SIGKILL** and **SIGSTOP** which always terminate the process, or stop the process, respectively, processes may control what happens when they receive a signal. They can:

1. Accept the default action, which may be to terminate the process, terminate and coredump the process, stop the process, or do nothing, depending on the signal.
2. Or, the processes can elect to explicitly ignore or to handle signals.
    1. **Ignored signals** are silently dropped.

2. **Handled signals** cause the execution of a user-supplied **signal handler** function. The program jumps to this function as soon as the signal is received and the control of the program resumes at the previously interrupted instruction.

The term "raise" is used to indicate the generation of a signal, and the term "catch" is used to indicate the receipt of a signal.

Signals are raised by error conditions, and they are generated by the shell and terminal handlers to cause interrupts and can also be sent from one process to another to pass information or to modify the behaviour.

In summary, signals can be:

- Raised
- Caught
- Acted upon
- Ignored

4. Example code

4.1 Standard use of pcntl_fork() – creating a process from within a process

```
<?php

// fork creates a copy of the current process
// the process id of the child process is returned
// the 'pid' variable is not assigned in the child variable
// the 'pid' is -1 if the fork fails
$pid = pcntl_fork();

// both processes continue to this point, but when pcntl_fork
// returns, we have to find out if we are in the parent process or
// in the child process, both run concurrently

if ($pid == -1) {
     die("Something went wrong, could not fork\n");
} else if ($pid) {
     // we are in the parent process, pcntl_fork returned pid>0, which is
     // the child's pid
     ...
} else {
     // we are in the child process, pcntl_fork returned 0, the child's PID
     // can be obtained by calling posix_getpid()
     ...
}
?>
```

4.2 pcntl_fork() with wait() – no zombies

```
<?php
$pid = pcntl_fork();
if ($pid == -1) {
     die("Something went wrong, could not fork\n");
} else if ($pid) {
     // we are in the parent process
     print("Parent: child process pid=: $pid \n") ;

     // wait for the child process to exit
```

```
      //  the 'status' variable is updated with the exit status
      pcntl_wait($status);

      // this function gets the exit code from the status
      $exitCode = pcntl_wexitstatus($status) ;
      print ("Child exit code is: $exitCode \n") ;
} else {
      // we are in the child process
      print("Child: my pid=") ;

      // this function gets the PID
      print posix_getpid() ;
      print "\n";
      exit (1) ;
}
?>
```

4.3 Using signals

```
<?php
// You need this for catching signals. Without this line your code
// will not be able to catch any signals. It instructs the PHP
// engine to check for signals on every 1 statement executed
declare(ticks=1);

// Install a signal handler for the SIGINT (CTRL-C) signal
// When the signal is caught, override the standard behavior and
// execute the code in function sig_handler
pcntl_signal(SIGINT, "sig_handler");

// The program just executes an infinite loop. Press CTRL-C to exit
// This will call the sig_handler function
while (1)
{
}

// Our signal handler
function sig_handler($signum) {
  echo("Phew, signal caught, killing process...\n");
  exit(1) ;
}
?>
```

5. Try to figure out what happens

Look at the following pieces of code and try to figure out what happens. They will help you understand how `fork()` works.

5.1 Process races

```
<?php

$pid = pcntl_fork();

if ($pid == -1) {
```

```
     die("Something went wrong, could not fork\n");
} else if ($pid) {
     // we are in the parent process
     print("Parent: child process pid= $pid \n") ;

     for ($i=0; $i<100; $i++) {
        echo "Parent, $i\n";
        //sleep(1);
} else {
     // we are in the child process
     for ($i=0; $i>-100; $i--) {
        echo "Child, $i\n";
        //sleep(1);
     }
}
?>
```

Can you guess the output? How will the behavior change if you add one of the sleep statements? If you add both?

5.2 Stack and fork()

```
<?php
$temp = 7;
$pid = pcntl_fork();
if ($pid==0)
   $temp = 6;
print("Temp and pid in here are: $temp, $pid \n");
?>
```

Can you figure out what happens and what the output will be? Can you understand why?

5.3 How many new processes?

How many new processes will be created here?

```
<?php
for ($i=0; $i<3; $i++)
   pcntl_fork();
?>
```