# ZendX_Console_Process_Unix - Ben Scholzen

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

## Zend Framework: Zend_Console_Process Component Proposal

| | |
|---|---|
| **Proposed Component Name** | Zend_Console_Process |
| **Developer Notes** | http://framework.zend.com/wiki/display/ZFDEV/Zend_Console_Process |
| **Proposers** | Ben Scholzen<br>Zend Liaison Ralph Schindler |
| **Revision** | 1.1 - 26 April 2008: Initial proposal (wiki revision: 12) |

## Table of Contents

1. Overview
2. References
3. Component Requirements, Constraints, and Acceptance Criteria
4. Dependencies on Other Framework Components
5. Theory of Operation
6. Milestones / Tasks
7. Class Index
8. Use Cases
9. Class Skeletons

## 1. Overview

Zend_Console_Process is a Processing component which simplifies the use of multiple pseudo-threads. This component can be used to accomplish multiple tasks at once. However, this component is limited to Unix bases systems, since Windows does not support the pcntl_* functions.

## 2. References

- Threading Wikipedia Entry

## 3. Component Requirements, Constraints, and Acceptance Criteria

- This component **will** allow running as many pseudo-threads parallely as possible.
- This component **will** allow communication between parent and child at any time.
- This component **will** keep an eye on zombie-processes.
- This component **will only** work on unix-like (Linux, BSD, Mac/OSx) systems with PHP CLI

## 4. Dependencies on Other Framework Components

- Zend_Exception
- pcntl_ functions
- shmop_ functions

## 5. Theory of Operation

The component must first be extended by another class, which implements the abstract method _run. After intantiation, the start() method can be called, which starts the new thread. While the thread is running, the parent can check via the isRunning() method, if the thread is still alive. To kill the process, the stop() method can be called.

## 6. Milestones / Tasks

- Milestone 1: [done] Working prototype checked into http://zend.svn.dasprids.de/
- Milestone 1: Proposal appprovement.
- Milestone 3: Unit tests exist and work.
- Milestone 4: Prototype and unit tests are checked into Incubator.
- Milestone 5: Initial documentation exists.

## 7. Class Index

- Zend_Console_Process_Abstract
- Zend_Console_Process_Exception

## 8. Use Cases

**UC-01**

```
class ThreadingTest extends Zend_Console_Process_Abstract
{
    protected function _run()
    {
        for ($i = 0; $i < 10; $i++) {
            sleep(1);
        }
    }
}

// This part should last about 10 seconds, not 20.
$thread1 = new ThreadingTest();
$thread1->start();

$thread2 = new ThreadingTest();
$thread2->start();

while ($thread1->isRunning() && $thread2->isRunning()) {
    sleep(1);
}

echo 'All threads completed';
```

## 9. Class Skeletons

```
class Zend_Console_Process_Exception extends Zend_Exception {}

abstract class Zend_Console_Process_Abstract
{
    /**
     * Constructor method
     *
     * Allocates a new pseudo-thread object. Optionally, set a PUID, a GUID and
     * a UMASK for the child process. This also initialize Shared Memory
     * Segments for process communications.
     *
     * @param int $puid
     * @param int $guid
     * @param int $umask
     */
    public function __construct($puid = null, $guid = null, $umask = null);

    /**
     * Test if the pseudo-thread is already started.
     *
     * @return bool
     */
    public function isRunning();

    /**
     * Set a variable into the shared memory segment, so that it can accessed
     * both from the parent and from the child process. Variable names with
     * underlines are only permitted to interal functions
     *
     * @param string $name
     * @param mixed $value
     */
    public function setVariable($name, $value);

    /**
```

```
     * Get a variable from the shared memory segment. Returns NULL if the
     * variable doesn't exist.
     *
     * @param string $name
     * @return mixed
     */
    public function getVariable($name);

    /**
     * Read the time elapsed since the last child setAlive() call.
     *
     * This method is useful because often we have a pseudo-thread pool and we
     * need to know each pseudo-thread status. If the child executes the
     * setAlive() method, the parent with getLastAlive() can know that child is
     * alive.
     *
     * @return int
     */
    public function getLastAlive();

    /**
     * Returns the PID of the current pseudo-thread.
     *
     * @return int
     */
    public function getPid();

    /**
     * Acutally Write a variable to the shared memory segment
     *
     * @param string $name
     * @param mixed $value
     */
    protected function _writeVariable($name, $value);

    /**
     * Set a pseudo-thread property that can be read from parent process
     * in order to know the child activity.
     *
     * Practical usage requires that child process calls this method at regular
     * time intervals; parent will use the getLastAlive() method to know
     * the elapsed time since the last pseudo-thread life signals...
     */
    protected function _setAlive();

    /**
     * This is called from within the parent method; all the communication stuff
     * is done here.
     *
     * @param string $methodName
     * @param array $argList
     */
    protected function _registerCallbackMethod($methodName, array $arglist);

    /**
     * This method actually implements the pseudo-thread logic.
     */
    abstract protected function _run();

    /**
     * Causes this pseudo-thread to begin parallel execution.
     *
     * This method first checks of all the Shared Memory Segment. If okay, it
     * forks the child process, attaches signal handler and returns immediatly.
     * The status is set to running, and a PID is assigned. The result is that
     * two pseudo-threads are running concurrently: the current thread (which
     * returns from the call to the start() method) and the other thread (which
     * executes its run() method).
     */
    public function start();
```

```
/**
 * Causes the current thread to die.
 *
 * The relative process is killed and disappears immediately from the
 * processes list.
 *
 * @return bool
 */
public function stop();

/**
 * Destroy thread context and free relative resources.
 */
protected function _cleanThreadContext();

/**
 * This is the signal handler that makes the communications between client
 * and server possible.
 *
 * @param int $signo
 */
protected function _sigHandler($signo);

/**
 * Sends signal to the child process
 */
protected function _sendSigUsr1();

/**
 * Wait for IPC Semaphore
 */
protected function _waitForIpcSemaphore();

/**
 * Read data from IPC segment
 */
protected function _readFromIpcSegment();

/**
 * Write data to IPC segment
 */
protected function _writeToIpcSegment();

/**
 * Create an IPC segment
 *
 * @return bool
 */
protected function _createIpcSegment();

/**
 * Create IPC semaphore
 *
 * @return bool
```

```
     */
    protected function _createIpcSemaphore();
}
```

]]></ac:plain-text-body></ac:macro>
]]></ac:plain-text-body></ac:macro>