What Are POSIX® Processes and Virtual Memory?

POSIX® Processes

In POSIX®, an executing instance of a program is called a process. To be conformant with the POSIX® standard, processes must be kept separate through the use of memory protection. An operating system (OS) that supports multiple processes is referred to as a "multiprocessing" OS.

It is important to understand that not all OSes that claim to support some of the POSIX® APIs actually support the POSIX® process model and its separation of processes into their own memory address spaces. Many OSes run proprietary APIs with a wrapper layer over them so that they can claim support for a subset of POSIX® APIs. However, if your OS doesn't have proper address space separation for processes, a bad pointer or invalid memory access can corrupt data in the applications. For instance, an invalid pointer access can corrupt an entirely different application, corrupt the kernel, crash the system, and can lead to bugs that are extremely difficult to understand and fix.

If you are looking at an operating system for FACE™ support, you must be aware that only the Operating System Segment (OSS) Safety-Extended Profile and General Purpose Profile require supporting the `fork()` system call. FACE™ OSes that only support the Safety-Base Profile and lesser profiles are not required to support the `fork()` system call. If you are considering a FACE™ OS that does not support both the `fork()` system call and the POSIX® process model with its separated address spaces, you could be setting yourself up for future debugging headaches.

POSIX® assumes that each process in a system resides in a different address space. In order to do this, LynxOS-178® uses the Memory Management Unit (MMU) of the processor to physically isolate the processes from each other so that they cannot access each other's memory; this makes systems more robust by preventing one process from reaching over and accidentally (or illicitly) accessing the address space of another process.

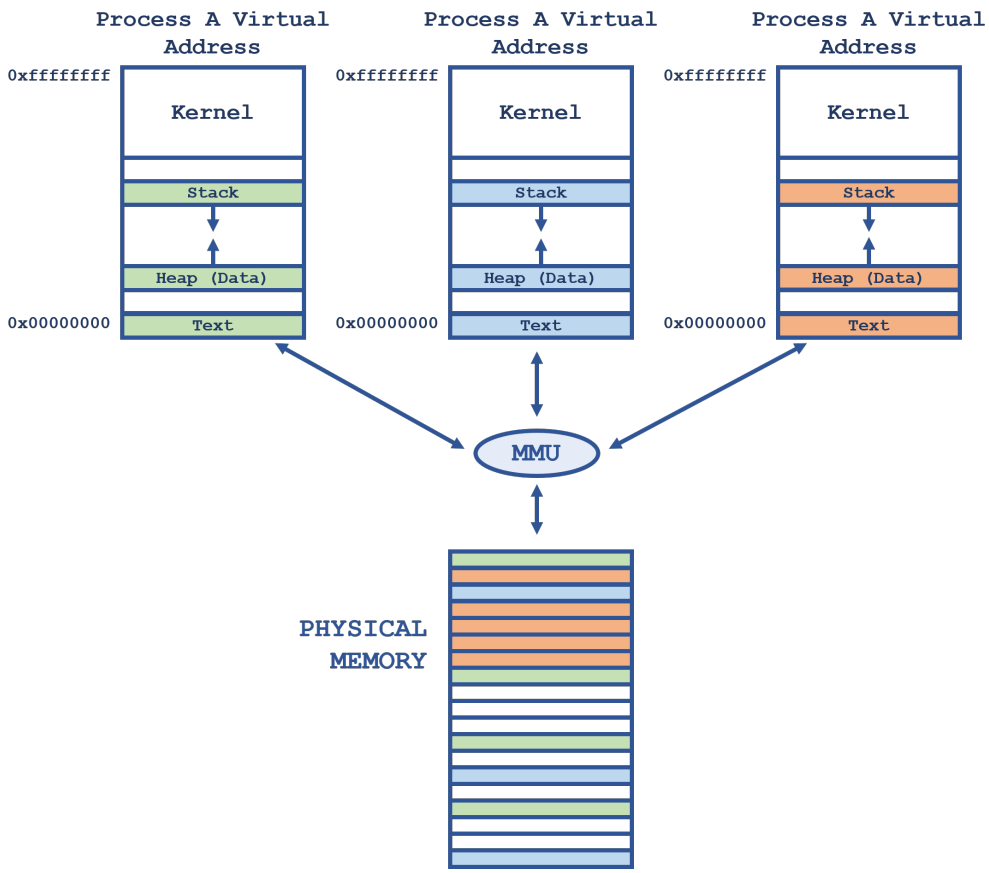
POSIX® Virtual Memory

Each process is assigned its own private memory in RAM. This memory may be located anywhere in physical RAM. The addresses used within a process are virtual addresses which are translated at run-time to their physical memory locations by the MMU. These virtual addresses compose the process' virtual address space.

A process' virtual-to-physical address space translations are private to that process. The memory addresses owned by the process can only be read or written by that process and cannot be accessed by other processes. This prevents processes from corrupting — or snooping on — each other's memory.

As shown in *Figure 1* (following page), a process' virtual address space is composed of segments:

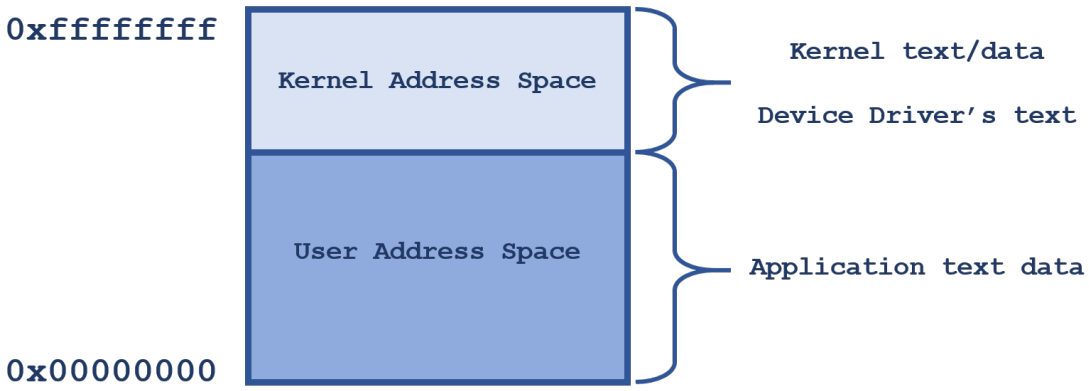
- The text segment contains the program's instructions
- The data segment contains global data and dynamically allocated data
- The stack segment contains the stacks (one stack is owned by each thread)
- Additional segments can also be defined



If a thread in a process attempts to address a physical page that is not mapped to it by the MMU, an exception is generated. The exception sends a signal to the offending thread, which has a default action of terminating the process. Alternatively, the thread may catch the signal for user-defined actions, if desired.

Processes Cannot Corrupt Kernel Memory

On LynxOS-178® and other POSIX® operating systems, a user application can only address data in an address range known as "user space". The remaining portion of the address space is called "kernel space". If you look at each of the processes in *Figure 1* (above), you'll notice that the kernel (and its variables) is mapped in the upper address range. Thus, each process has a view of memory like that shown in *Figure 2* (below):

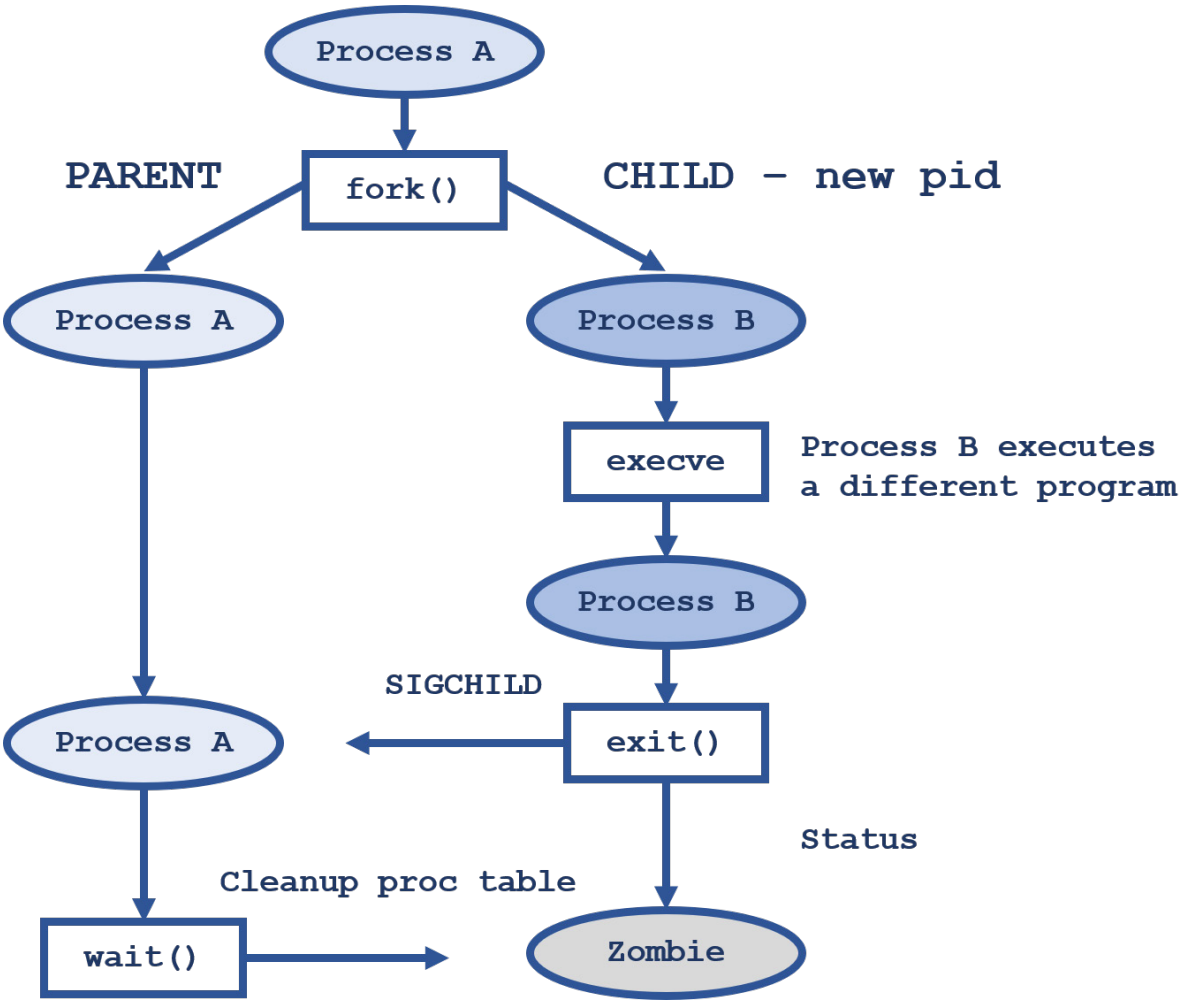


An application cannot access an address in kernel space. LynxOS-178® uses the MMU to protect the kernel and its data against an application accessing, and possibly corrupting, the kernel's address space.

> Sidebar: **Why is protecting kernel memory important?** Ignoring the obvious security concerns of allowing an application unfettered access to the kernel's memory, if an application were to have a wayward pointer that corrupted the kernel's memory, it could potentially crash the OS — or perhaps even worse, not immediately crash the OS. I've worked on OSes that ran everything in a flat memory address space. Debugging pointer errors that caused crashes in that environment was difficult and frustrating. The system itself was extremely unstable because of the way memory was left unprotected. Debugging on POSIX® operating systems that separate and protect processes' address spaces is much easier — and they are much more stable and robust when faced with the inevitable application error.

Processes and Parent-Child Relationships

The POSIX® `fork()` system call creates a new process called a child process. The original process is called the parent process. See *Figure 3* (below) for a visual representation of how a child process is created and the resulting parent-child relationship:



The child is a near-exact copy of the parent. Its virtual address space is a copy of the parent's, and it is given copies of the parent's open file descriptors, semaphores, message queues, mutex states, and scheduling priority and policy. However, the child has only one thread of execution – the thread that returns from the `fork()` system call – and some resources, such as file locks, asynchronous I/O operations, pending signals, and timers are not inherited. The child also has its own process ID.

Recovering the Exit Status of a Process

A parent process can recover the exit status of a child using the `wait()` or `waitpid()` POSIX® functions. The `wait()` function blocks until any child process terminates. The `waitpid()` function waits for a specific child process to terminate, based upon the child's process ID.

POSIX® defines pairs of macros for decoding a child's exit code from `wait()` and `waitpid()`. Here is a short summary of the POSIX® status code macros available in LynxOS-178®:

`WIFEXITED(exitcode)` — True if child exited normally (i.e. it called exit)
`WEXITSTATUS(exitcode)` — Returns exit code of normally exited child
`WIFSIGNALED(exitcode)` — True if child was killed by an uncaught signal
`WTERMSIG(exitcode)` — Returns signal that killed child
`WIFSTOPPED(exitcode)` — True if child is stopped
`WSTOPSIG(exitcode)` — Returns signal that stopped child

It is important to note that the order of checking is important with these POSIX® macros. You must check for a normal exit status before checking for signal status information.

Discussed POSIX® Functions and Macros —

We have briefly described the following POSIX® functions and macros:

POSIX® Functions:

```
int execve(const char *path, char *const argv[], char *const envp[]);
pid_t fork(void);
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

POSIX® Macros:

```
WIFEXITED(exitcode)
WEXITSTATUS(exitcode)
WIFSIGNALED(exitcode)
WTERMSIG(exitcode)
WIFSTOPPED(exitcode)
WSTOPSIG(exitcode)
```

Leveraging POSIX® for Your Next Project

Lynx has over 30 years' experience in helping customers across avionics, automotive, and industrial markets to realize the benefits of POSIX® for their complex safety- and security-critical embedded software systems. To learn more about how to leverage POSIX® for your project, please visit www.lynx.com or direct your inquiries to **inside@lynx.com** and a representative will reach out to you within 1-2 business days.