Process Creation

Eric Missimer

Process ID

- Every process has a process ID, a unique non-negative integer
- Process 0
  - Traditionally was the swapper task (used for swapping memory to disk)
  - On Linux it is the idle task
  - In either case this is not a user level program, it is part of the kernel
- Process 1
  - The init process (usually)
  - This is the process that the kernel first runs when it is done initializing
- `pid_t getpid(void)` – returns process ID of calling process
- `pid_t getppid(void)` – returns parent process ID of calling process

Creating a Process – `fork`

- A new process is created by creating a duplicate of the parent process via the `fork` system call
- `pid_t fork(void)`
- `fork.c` code example

Creating a Process – `fork`

- A new process is created by creating a duplicate of the parent process via the `fork` system call
- `pid_t fork(void)`
- `fork.c` code example
- Interfaces change over time

Creating a Process – fork

- A new process is created by creating a duplicate of the parent process via the `fork` system call
- `pid_t fork(void)`
- `fork.c` code example
- Interfaces change over time
- `fork` in glibc actually makes the `clone` system call

clone

- `fork` is still a system call and the way the glibc `fork` function calls the system call `clone` has the same behavior as the `fork` system call
-
```
#define ARCH_FORK()                                   \
INLINE_SYSCALL (clone, 4,                             \
                CLONE_CHILD_SETTID |                  \
                CLONE_CHILD_CLEARTID |                \
                SIGCHLD, 0,                           \
                NULL, &THREAD_SELF->tid)
```

To Recap

fork	library	Is a wrapper for the clone system call
fork	system call	Still exists just isn't called when we call the fork library call duplicates a process
clone	library	Wraps the clone system call, architecture independent function prototype
clone	system call	Function prototype definition can be architecture dependent duplicates a process with a varying amount of shared resources

Back to `fork`

- `fork` is called once and returned twice (assuming no error)
- If there is an error -1 is returned
- On success:
  - In the parent: the child process id is returned
  - In the child: 0 is returned

Sidenote: File Descriptor Table

- Why are file descriptors integers?

Sidenote: File Descriptor Table

- Why are file descriptors integers?
- They are used to index into the file descriptor table

Sidenote: File Descriptor Table

- Why are file descriptors integers?
- They are used to index into the file descriptor table
- The file descriptor table is a per process data structure stored in the kernel
- When we use a file descriptor in a syscall the kernel uses that to index into the descriptor table which contains things such as the file status flags, the current file offset and the v-node pointer.
- the v-node pointers points to an entry in the v-node table. Each entry contains things such as the information about the v-node, and the corresponding i-node.

Sharing Files

- When a process calls `fork` its userspace memory isn't the only thing duplicated
- The kernel copies the file descriptor table too
- Therefore all the previous file descriptors in the child correspond to the same files
- We saw this in the previous code example; the child printed to the same terminal as the parent process

exec

- So `fork` gives us a copy of the parent process
- That's great but how do we run a new program

exec

- So `fork` gives us a copy of the parent process
- That's great but how do we run a new program
- The answer is the `exec` system call

`exec`

- So `fork` gives us a copy of the parent process
- That's great but how do we run a new program
- The answer is the `exec` system call
- `exec` replaces the current process image with a new process image
- Where do the get the new process image?

- So `fork` gives us a copy of the parent process
- That's great but how do we run a new program
- The answer is the `exec` system call
- `exec` replaces the current process image with a new process image
- Where do the get the new process image?
- Answer: from a file

exec variants

- `int execl(const char *path, const char *arg, ...)`
- `int execlp(const char *file, const char *arg, ...)`
- `int execle(const char *path, const char *arg, ..., char * const envp[])`
- `int execv(const char *path, char *const argv[])`
- `int execvp(const char *file, char *const argv[])`
- `int execvpe(const char *file, char *const argv[], char *const envp[])`
- Decoding the exec variants
  - `l` - the arguments to the program are passed to exec as an argument list
  - `v` - the arguments to the program are passed to exec as a null-terminated array
  - `p` - if `file` does not contain a slash look in the PATH environment variable for the specified file
  - `e` - the last argument is a null-terminating array of strings

vfork

- `vfork` exists as a lightweight `fork`
- It has the calling sequence and return values as `fork`
- `vfork` creates a new process just like `fork`
- `vfork` differs from `fork` in that it does not copy the address space
- `vfork.c` code example

`vfork`

- Why is `vfork` useful?

`vfork`

- Why is `vfork` useful?
- If we are about to call `exec` we don't have to copy the address space

`vfork`

- Why is `vfork` useful?
- If we are about to call `exec` we don't have to copy the address space
- Sidenote: `vfork` has become a little obsolete do to copy-on-write
- `vfork` guarantees that the child runs first, until the child calls exec or exit. When the child calls either of these functions, the parent resumes

`wait`/`waitpid`

- Why does `exit` take an integer value?

`wait`/`waitpid`

- Why does `exit` take an integer value?
- It is returned to the parent process

wait/waitpid

- Why does `exit` take an integer value?
- It is returned to the parent process
- The parent process can access the value via `wait/waitpid`
- `pid_t wait(int *status)`
- `pid_t waitpid(pid_t pid, int *status, int options)`
- The `wait` function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
- The `waitpid` function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for