Implementation of fork() system call in haiku operating system

Fork System Call in Operating System

In Haiku OS, the fork() system call, like in other Unix-like operating systems, is used to create a new process that is a copy of the current process. It creates a child process that is a duplicate of the parent process, with the notable exception that the return value of fork() differs for the parent and child processes. In the child, fork() returns 0, while in the parent it returns the PID (Process ID) of the child. A failed fork call returns -1.

How fork() works in Haiku:

1. Memory Copy:

The core of fork() is the creation of a copy of the parent process's address space. This includes all the memory, file descriptors, and other resources.

2. Copy-on-Write:

The memory copy is typically implemented using copy-on-write. This means that initially, both the parent and child processes share the same physical memory pages. If either process attempts to modify a page, a copy of that page is made, and the respective process is assigned ownership of the new copy.

3. Process Creation:

A new process control block (PCB) is created for the child, which contains all the necessary information to manage the process. This includes its PID, parent PID, user ID, group ID, memory map, and more.

4. Return Values:

The key difference lies in the return value of the system call:

In the child process: fork() returns 0.

In the parent process: fork() returns the PID of the child.

If an error occurs: fork() returns -1.

5. Execution:

After fork() returns, both the parent and child processes continue executing from the po int immediately following the call. The parent, knowing the child's PID, can then use oth er system calls like waitpid() to monitor the child's execution.

```
// src/system/kernel/syscalls.cpp

#include <kernel/Thread.h>

#include <kernel/Team.h>

#include <vm/vm.h>

#include <vm/VMAddressSpace.h>

status_t sys_fork(team_id* _childTeamID) {

Thread* parentThread = thread_get_current_thread();

Team* parentTeam = parentThread->team;
```

Step 1: Create child team

```
Team* childTeam = team_create("forked child", TEAM_FLAG_DEFAULT);

if (childTeam == NULL)

   return B_NO_MEMORY;
```

Step 2: Duplicate address space

```
AddressSpace* parentAddressSpace = parentTeam->address_space;
```

```
AddressSpace* childAddressSpace = vm_clone_address_space(parentAddressSpace);

if (childAddressSpace == NULL) {

   team_delete(childTeam);

   return B_NO_MEMORY;

}

childTeam->address_space = childAddressSpace;
```

Step 3: Clone file descriptors (simplified)

```
fd_table_clone(parentTeam->fd_table, &childTeam->fd_table);
```

Step 4: Create thread in child

```
   thread_id childThreadID = thread_create_team_thread(childTeam, parentThread->entry
_point,

                                parentThread->args, THREAD_FLAG_DEFAULT);

   if (childThreadID < 0) {

      team_delete(childTeam);

      return childThreadID;

   }
```

Step 5: Schedule the thread

```
   thread_resume(childThreadID);
```

```
    if (_childTeamID)

        *_childTeamID = childTeam->id;

    return B_OK;

}
```