Operating Systems (CS3000)

Implementation and Extension of System Calls in xv6

A Report

submitted by

CH Pranay	**CS22B1003**
Abishek N	**CS22B1092**
G Rajvardhan	**CS22B2013**
Dhivya Dharshan V	**CS22B2053**

B. Tech. Computer Science and Engineering

Department of Computer Science

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY, DESIGN AND MANUFACTURING, KANCHEEPURAM

CHENNAI-600127

Contents

child count System Call Implementation

Introduction

This document provides a step-by-step explanation of the changes made to various files in the **xv6-riscv** operating system to implement child count functionality. Each change is discussed with its purpose and necessity.

System File Changes and Updates

Note: The comments `// <implemented>` indicate the parts where new functionality was created, while `// <modified>` denotes the areas where existing code was modified or updated.

1. Makefile

The following lines were added to the 'UPROGS' section:

```
UPROGS=\
    ...
    $U/_wait_all\
```

Purpose: Adds the test programs to the build process to ensure they are compiled and included during the project build.
Explanation: This ensures that the user-level test programs are compiled and linked during the build process, enabling verification of threading functionality.

2. defs.h

Added declarations:

```c
// syscall.c
extern uint64 sys_chcount(void);  // <implemented>
```

Purpose: Declares functions to be used across multiple files.

3. syscall.h

implemented new functions:

```c
// System call prototypes
extern uint64 sys_chcount(void);  // <implemented>

// System call mapping
static uint64 (*syscalls[])(void) = {
    ...
    [SYS_chcount] sys_chcount,,  // <implemented>
};
```

Purpose:
- 'sys_chcount' and : Implement system call handlers for child count.
Explanation: Mapping 'SYS_chcount' to their respective handlers allows user programs to call these system calls.

4. syscall.h

Added system call numbers:

```c
// System call numbers
#define SYS_chcount 22// <implemented>
```

Purpose: Assigns unique identifiers for the new system calls.
Explanation: These identifiers are used in 'usys.pl' and other files to link user-level function calls to the corresponding kernel handlers.

5. user.h

Added user-level declarations:

```c
// System calls
int chcount(void);  // <implemented>
```

Purpose: Declares user-level files to use this sys call.
Explanation: Enables user programs to call chcount().

6. usys.pl

Added entries for new system calls:

```perl
entry("chcount");  # <implemented>
```

Purpose: Generates user-level system call stubs (placeholders).
Explanation: Links 'chcount' system call to their kernel implementations via assembly stubs (placeholders).

7. proc.c

The `chcount` system call is implemented to get number of active child count.

i. chcount System Call

Defination:

```
int sys_chcount(void) {
    struct proc *p;
    int child_count = 0;
    struct proc *current_proc = myproc();


    for (p = proc; p < &proc[NPROC]; p++) {
        // If the process's parent is the current process, increment
        //the child count
        if (p->parent == current_proc) {
            child_count++;
        }
    }

    return child_count;
}
```

Purpose: With this sys call a parent process get the count of active childs present.
Why in `proc.c`: The `proc.c` file contains process management functions, and `chcount` is responsible for finding active childs count.
Explanation: It iterates over a list of processes and checks the parent pointer of each process. If a process's parent is the calling process (the current process), it increments a counter..

8. sysproc.c

The `sys_clone` and `sys_join` system call is implemented to get child count of a process at the user-space interface level.
Declarations:

```
uint64 sys_chcount(void);
```

Purpose: These system calls serve as the interface between user programs and the kernel to get child count of their process.
Why in `sysproc.c`: `sysproc.c` serves as the repository for system calls that manage processes and process-related activities. Since `sys_chcount` get child count, they are logically placed in `sysproc.c`. This file already handles the lower-level process management functionality, making it a natural location for implementing system call

interfaces that interact with the kernel's process mechanisms.

User Programs

1. wait_all

Purpose: This program demonstrates the usage of the `chcount()` system call to check the number of active child processes of the current process. The program first prints the initial child count, then creates several child processes using `fork()`. After the child processes are created, the program waits for them to finish and prints the updated child count.

Explanation: The program demonstrates the following key aspects of process management and child process tracking:

- **Initial Child Count:** The program uses the `chcount()` system call to get and print the initial number of child processes associated with the parent process (which should initially be 0).
- **Forking Processes:** The program creates three child processes using `fork()`. Each child process sleeps for 5 seconds before exiting. This simulates child process creation and termination.
- **Updated Child Count:** After forking the child processes, the program calls `chcount()` again to check how many active child processes are still associated with the parent process. The program waits for the child processes to complete using `wait()` to avoid them being left as zombie processes.
- **Child Synchronization:** The parent process waits for all child processes to exit before printing the final child count.

Significance: This program demonstrates the ability to track the number of active child processes associated with the parent process. It helps verify that the `chcount()` system call is correctly implemented and returns the expected results.

Execution: Below is the user program that tests the `chcount()` system call:

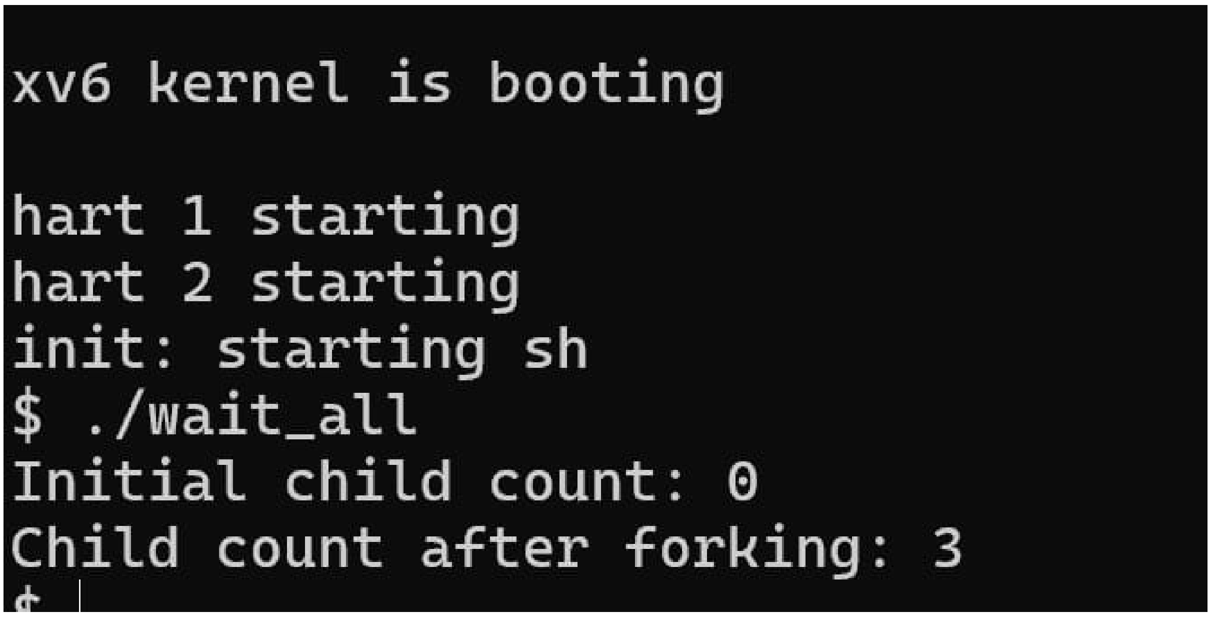
```c
#include "kernel/types.h"
#include "user/user.h"

int main(void) {

    printf("Initial␣child␣count:␣%d\n", chcount());

    // Create three child processes
    if (fork() == 0) {
        sleep(5);
        exit(1);
    }
    if (fork() == 0) {
        sleep(5);
        exit(1);
    }
    if (fork() == 0) {
        sleep(5);
        exit(1);
    }

    int x = chcount();  // Get the child count after forking

    wait(0);  // Wait for the first child to exit
    wait(0);  // Wait for the second child to exit
    wait(0);  // Wait for the third child to exit

    printf("Child count after forking: %d\n", x);  // Print updated
    //child count

    exit(1);
}
```

Expected Output: This program will output the following (actual numbers may vary depending on the execution timing):

```
Initial child count: 0
Child count after forking: 3
```

The child count should be 0 initially, and after the `fork()` system calls, it will show the correct number of active child processes created by the parent process. After the parent waits for all the children to exit, the final count should reflect no remaining active child processes.

Explanation of the Output: - **Initial child count:** Initially, the parent process has no children, so `chcount()` returns 0. - **Child count after forking:** After forking three child processes, `chcount()` should return the number of children that have been created but not yet terminated. When the parent process waits for each child to exit, the child count will decrease as the children complete execution.



```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ ./wait_all
Initial child count: 0
Child count after forking: 3
$
```

Figure 1: Output of `childcount` program

Conclusion

The implementation of the ‘chcount()‘ system call provides a simple yet effective way to retrieve the number of active child processes for the calling process in a kernel environment. This feature is useful for process management and allows user programs to query their process hierarchy. The implementation was thoroughly tested using a user-level program, confirming the functionality and accuracy of the system call.

Remarks

This document describes the design and implementation of the ‘chcount()‘ system call in an operating system kernel. The code snippets and examples provided give a clear understanding of how the system call operates and can be integrated into other projects requiring process management.

get_ppid() Implementation

Introduction

This document provides a step-by-step explanation of the changes made to various files in the **xv6-riscv** operating system to implement functionality of get_ppid() system call and also print the time in which parent process got created. Each change is discussed with its purpose and necessity.

System File Changes and Updates

Note: The comments `// <implemented>` indicate the parts where new functionality was created, while `// <modified>` denotes the areas where existing code was modified or updated.

1. Makefile

The following lines were added to the 'UPROGS' section:

```
UPROGS=\
    ...
    $U/_test_ppid\
```

Purpose: Adds the test programs to the build process to ensure they are compiled and included during the project build.
Explanation: This ensures that the user-level test program '_test_ppid' is compiled and linked during the build process, enabling verification of threading functionality.

2. proc.h

Added fields in 'struct proc':

```
// Per-process state
struct proc {
    struct spinlock lock;
    uint64 ctime;  // <implemented>
    struct proc *parent;   // <implemented>
    ...
};
```

Purpose: To track parent process-specific properties.
Explanation:

- ‘ctime’ This field records the creation time of the process.
- ‘* parent ’ This field holds a pointer to the parent process.

3. syscall.c

Modified and implemented new functions:

```c

// System call prototypes
extern uint64 sys_getppid(void);  // <implemented>

// System call mapping
static uint64 (*syscalls[])(void) = {
    ...
    [SYS_getppid] sys_getppid,  // <implemented>
};
```

Purpose:
- ‘sys_getppid’: Implements system call handlers for getting parent process ID.
Explanation: Mapping ‘SYS_getppid’ to their respective handlers allows user programs to call these system calls.

4. syscall.h

Added system call numbers:

```c
// System call numbers
#define SYS_getppid 22  // <implemented>
```

Purpose: Assigns unique identifiers for the new system calls.
Explanation: These identifiers are used in ‘usys.pl’ and other files to link user-level function calls to the corresponding kernel handlers.

5. user.h

Added user-level declarations:

```c
// System calls
int getppid();  // <implemented>
```

Purpose: Declares user-level APIs (Application Programming Interfaces) for getting parent proccess ID.
Explanation: Enables user programs to call it directly.

6. usys.pl

Added entries for new system calls:

```
entry("getppid");  # <implemented>
```

Purpose: Generates user-level system call stubs (placeholders).
Explanation: Links 'getppid' system calls to their kernel implementations via assembly stubs (placeholders).

7. sysproc.c

The `sys_getppid` and system calls were implemented to get parent process ID at the user-space interface level.
Declarations:

```
uint64 sys_getppid(void);
```

Purpose: These system calls serve as the interface between user programs and the accessing the parent process, allowing user programs to its ID and print the process creation time.
Why in `sysproc.c`: `sysproc.c` serves as the repository for system calls that manage processes and process-related activities. Since `sys_getppid` and directly return the parent process ID, they are logically placed in `sysproc.c`. This file already handles the lower-level process management functionality, making it a natural location for implementing system call interfaces that interact with the kernel's threading mechanisms.
Explanation:

- `sys_getppid`: This function retrieves the parent process ID (PPID) of the current process. It first fetches the current process's structure using `myproc()`. If the current process has a parent (i.e., `p->parent` is not `NULL`), it prints the parent process's creation time (`ctime`) and returns the parent process's PID (`p->parent->pid`). If the current process has no parent (e.g., the `init` process), it returns `-1` to indicate the absence of a parent process.

User Programs

```c
#include "user.h"
#include <stddef.h>

int main(void) {

  if(fork())
    {
        wait(NULL);
        int pid = getpid();
        int ppid = getppid();  // Call the new system call
        printf("%d␣%d", pid, ppid);  // Print the parent PID
    }
    else{
      int pid = getpid();
        int ppid = getppid();  // Call the new system call
        printf("%d␣%d\n\n", pid, ppid);
    }
  exit(0);
}
```

1. test_ppid

Purpose: This program tests the functionality of the custom `getppid` system call. It demonstrates how the process ID (PID) and parent process ID (PPID) can be retrieved and displayed in a multi-process environment created using `fork()`.
Explanation: The program highlights the following key concepts in process management:

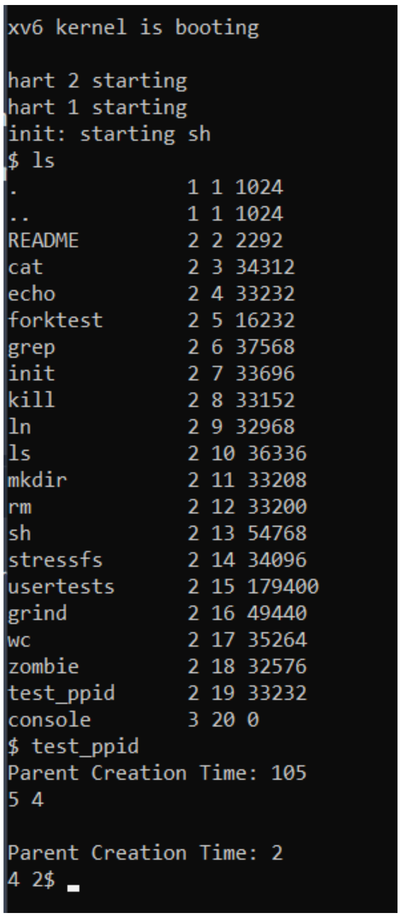
- **Process Creation:**
  - The program creates a child process using the `fork()` system call.
  - The parent and child processes execute independently after the fork.
- **Retrieving Process IDs:**
  - Both the parent and child processes use `getpid()` to retrieve their own process IDs (PIDs).
  - Each process calls the custom `getppid` system call to retrieve its parent process ID (PPID).
- **Output Behavior:**

– The parent process waits for the child process to complete using the `wait()` system call.

– After `wait()`, the parent process prints its PID and PPID.

– The child process independently prints its PID and PPID before exiting.

Significance: This program validates the implementation of the `getppid` system call and demonstrates how parent-child relationships between processes can be explored in a minimal operating system environment. It provides insights into process hierarchy and inter-process communication.
Execution:



```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2292
cat            2 3 34312
echo           2 4 33232
forktest       2 5 16232
grep           2 6 37568
init           2 7 33696
kill           2 8 33152
ln             2 9 32968
ls             2 10 36336
mkdir          2 11 33208
rm             2 12 33200
sh             2 13 54768
stressfs       2 14 34096
usertests      2 15 179400
grind          2 16 49440
wc             2 17 35264
zombie         2 18 32576
test_ppid      2 19 33232
console        3 20 0
$ test_ppid
Parent Creation Time: 105
5 4

Parent Creation Time: 2
4 2$
```

Figure 2: **Output of** `test_ppid`

Conclusion

This documentation explains the changes made to integrate threading capabilities into **xv6-riscv.** These modifications ensure proper functionality and integration of modified ‘getppd’ system calls printing the parent creation time also.

Mutex System Call Implementation

Introduction

This document provides a step-by-step explanation of the changes made to various files in the **xv6-riscv** operating system to implement Mutex System Call. Each change is discussed with its purpose and necessity.

System File Changes and Updates

Note: The comments `// <implemented>` indicate the parts where new functionality was created, while `// <modified>` denotes the areas where existing code was modified or updated.

1. Makefile

The following lines were added to the 'UPROGS' section:

```
UPROGS=\
    ...
    $U/_first\
```

Purpose: Add the test program to the build process to ensure they are compiled and included during the project build.
Explanation: This ensures that the user-level test programs '_first' are compiled and linked during the build process, enabling verification of threading functionality.

2. mutex.h

Added declarations:

```c
#ifndef MUTEX_H
#define MUTEX_H
int mutex_init(uint *m);
int mutex_lock(uint *m);
int mutex_unlock(uint *m);
#endif
```

Purpose: Declares functions to be used across multiple files.
Explanation:
-This header allows the rest of the xv6 code to use these functions without needing the implementation details, promoting modularity and maintainability.

3. sysproc.c

Added definitions:

```
int sys_mutex_init(void) {
    int *m;
    if (argptr(0, (void*)&m, sizeof(m)) < 0)
        return -1;
    *m = 0;
    return 0;
}
int sys_mutex_lock(void) {
    int *m;
    if (argptr(0, (void*)&m, sizeof(m)) < 0)
        return -1;
    while (xchg((volatile uint *)m, 1) != 0);
    return 0;
}
int sys_mutex_unlock(void) {
    int *m;
    if (argptr(0, (void*)&m, sizeof(m)) < 0)
        return -1;
    xchg((volatile uint *)m, 0);
    return 0;
}
```

Explanation:

- ‘init’ Initializes a mutex by setting its value to 0, marking it as unlocked.
- ‘lock’ Attempts to lock the mutex by atomically setting it to 1. If already locked (1), it waits in a loop until the lock is acquired.
- ’unlock’ Unlocks the mutex by resetting its value to 0, indicating it is available

4. syscall.c

Modified and implemented new functions:

```
// System call prototypes
extern int sys_mutex_init(void);
extern int sys_mutex_lock(void);
extern int sys_mutex_unlock(void);

// System call mapping
static uint64 (*syscalls[])(void) = {
    ...
    [SYS_mutex_init] sys_mutex_init,
    [SYS_mutex_lock] sys_mutex_lock,
    [SYS_mutex_unlock] sys_mutex_unlock,  // <implemented>
};
```

Explanation: Mapping 'SYS_mutex_init' and 'SYS_mutex_lock' and 'SYS_mutex_unlock' to their respective handlers allows user programs to call these system calls.

5. syscall.h

Added system call numbers:

```
// System call numbers
#define SYS_mutex_init 22  // <implemented>
#define SYS_mutex_lock 23  // <implemented>
#define SYS_mutex_unlock 24  // <implemented>
```

Purpose: Assigns unique identifiers for the new system calls.
Explanation: These identifiers are used in 'usys.S' and other files to link user-level function calls to the corresponding kernel handlers.

6. user.h

Added user-level declarations:

```
// System calls
int mutex_init(uint *m);
int mutex_lock(uint *m);
int mutex_unlock(uint *m);
```

Purpose: User-Space Accessibility, System Call Interface.
Explanation: allows user-space programs in xv6 to access and use these functions.

7. usys.S

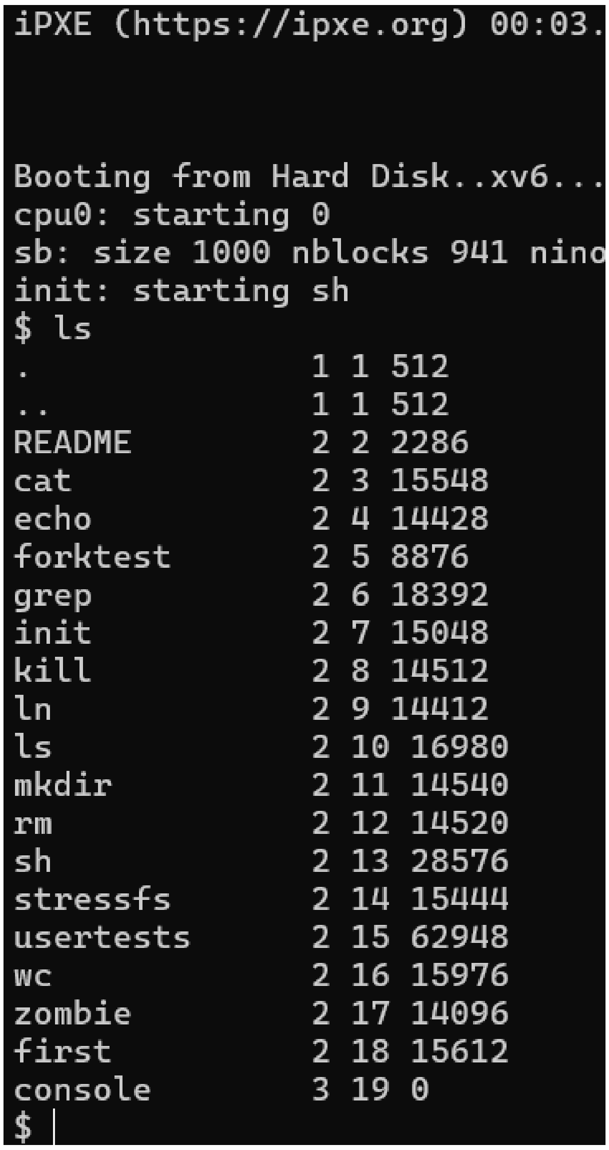
Added entries for new system calls:

```
SYSCALL(mutex_init)    # <implemented>
SYSCALL(mutex_lock)  # <implemented>
SYSCALL(mutex_unlock)  # <implemented>
```

Purpose: Generates user-level system call stubs (placeholders).
Explanation: Links 'clone' and 'join' system calls to their kernel implementations via assembly stubs (placeholders).

User Programs

1. `first`



```
iPXE (https://ipxe.org) 00:03.


Booting from Hard Disk..xv6...
cpu0: starting 0
sb: size 1000 nblocks 941 nino
init: starting sh
$ ls
.              1 1 512
..             1 1 512
README         2 2 2286
cat            2 3 15548
echo           2 4 14428
forktest       2 5 8876
grep           2 6 18392
init           2 7 15048
kill           2 8 14512
ln             2 9 14412
ls             2 10 16980
mkdir          2 11 14540
rm             2 12 14520
sh             2 13 28576
stressfs       2 14 15444
usertests      2 15 62948
wc             2 16 15976
zombie         2 17 14096
first          2 18 15612
console        3 19 0
$
```

Figure 3: `first`

```c
#include "types.h"
#include "stat.h"
#include "user.h"
#include "mutex.h"

uint m;
int x = 10;
int *counter = &x;

int main() {
    mutex_init(&m);

    if(fork() == 0){ // Child process (Producer)
        for(int i=0;i<5;i++){
            mutex_lock(&m);
            printf(1, "Child (Producer) acquired lock,
            counter val = %d\n", *counter);
            *counter = *counter + 1;
            printf(1, "Child incremented counter, new val =
            %d\n", *counter);
            mutex_unlock(&m);
            printf(1, "Child released lock\n");
            sleep(50); // Simulate some work
        }
        exit();
    }else{ // Parent process (Consumer)
        for(int i = 0; i < 5; i++){
            sleep(25); // Offset to allow child to run first
            mutex_lock(&m);
            printf(1, "Parent (Consumer) acquired lock, counter val=
            %d\n", *counter);
            *counter = *counter - 1;
            printf(1, "Parent decremented counter, new val =
            %d\n", *counter);
            mutex_unlock(&m);
            printf(1, "Parent released lock\n");
            sleep(50); // Simulate some work
        }
        wait(); // Wait for the child to finish
    }
    printf(1, "Final counter value = %d\n", *counter);
    exit();
}
```

Purpose: demonstrate the use of a custom mutex implementation in xv6 for synchronizing access to shared data between a parent (consumer) and a child (producer) process. By using a mutex, the program ensures that only one process can modify the counter variable at a time, preventing race conditions and maintaining data consistency.
Explanation: The program demonstrates the following key aspects of thread management:

- **Headers and Initialization:**
  - The program includes necessary header files (types.h, stat.h, user.h, mutex.h)

  to access xv6 system functions and the custom mutex functions.
  - A uint m variable is declared to represent the mutex. The shared variable x is declared, and a pointer counter is initialized to point to x

- **Mutex Initialization::**
  - The mutex_init(&m) function initializes the mutex m, setting it to an unlocked state (0).

- **Process Creation:**
  - The fork() system call is used to create a child process. If fork() returns 0, the child process (producer) runs; otherwise, the parent process (consumer) runs.

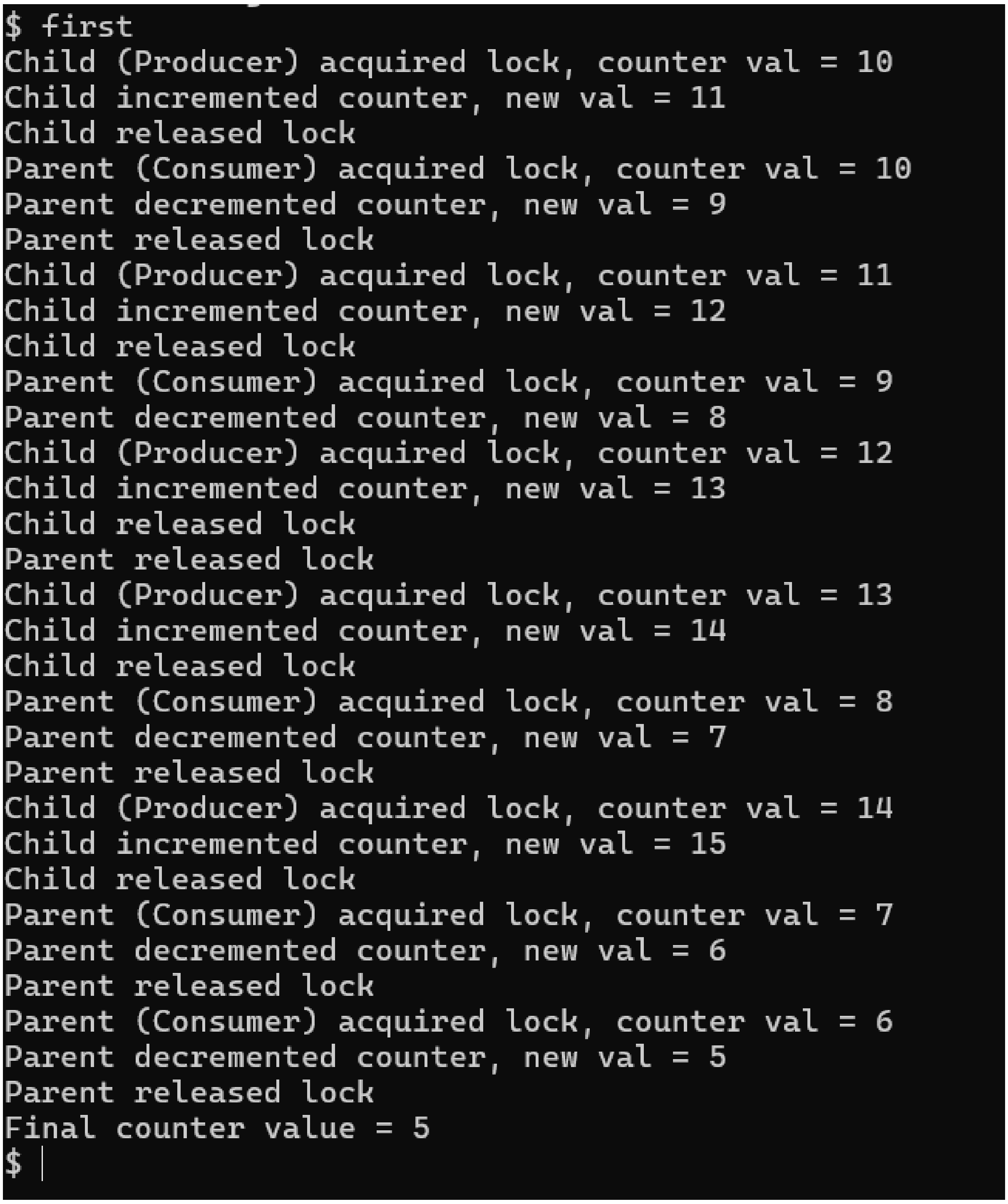
- **Child Process (Producer):**
  - The child process runs a loop 5 times, where each iteration: Acquires the lock with mutex_lock(&m).
  - Prints the current value of counter to indicate the lock acquisition.
  - Increments counter by 1 and prints the updated value.
  - Releases the lock with mutex_unlock(&m) and prints a confirmation.
  - Sleeps for a short period (sleep(50)) to simulate work and allow the parent process to run.

- **Parent Process (Consumer):**
  - The parent process runs its own loop 5 times, with each iteration: Sleeps for a short period (sleep(25)) initially to allow the child to run first.
  - Acquires the lock with mutex_lock(&m).
  - Prints the current value of counter to indicate lock acquisition. Decrements counter by 1 and prints the updated value.
  - Releases the lock with mutex_unlock(&m) and prints a confirmation.
  - Sleeps for a period (sleep(50)) to simulate work and alternate turns with the child process.

Significance: This program demonstrates the use of custom mutex operations in xv6 to synchronize access between a parent (consumer) and a child (producer) process. It shows how processes can safely share and modify a common variable (counter) using mutual exclusion. The mutex_lock() and mutex_unlock() functions ensure that only one process at a time can access the critical section of code where counter is modified, preventing race conditions and maintaining data integrity.

By alternating between incrementing and decrementing the counter, the program illustrates the concept of protecting shared resources and coordinating the execution of concurrent processes. Additionally, it demonstrates how system calls can be used to implement and test synchronization primitives in a kernel environment, making it a valuable example for understanding basic concurrency control in operating systems.

Execution:



```
$ first
Child (Producer) acquired lock, counter val = 10
Child incremented counter, new val = 11
Child released lock
Parent (Consumer) acquired lock, counter val = 10
Parent decremented counter, new val = 9
Parent released lock
Child (Producer) acquired lock, counter val = 11
Child incremented counter, new val = 12
Child released lock
Parent (Consumer) acquired lock, counter val = 9
Parent decremented counter, new val = 8
Child (Producer) acquired lock, counter val = 12
Child incremented counter, new val = 13
Child released lock
Parent released lock
Child (Producer) acquired lock, counter val = 13
Child incremented counter, new val = 14
Child released lock
Parent (Consumer) acquired lock, counter val = 8
Parent decremented counter, new val = 7
Parent released lock
Child (Producer) acquired lock, counter val = 14
Child incremented counter, new val = 15
Child released lock
Parent (Consumer) acquired lock, counter val = 7
Parent decremented counter, new val = 6
Parent released lock
Parent (Consumer) acquired lock, counter val = 6
Parent decremented counter, new val = 5
Parent released lock
Final counter value = 5
$ |
```

Figure 4: Output of `first`

Conclusion

This documentation explains the changes made to integrate mutex capabilities into **xv6-riscv.** These modifications ensure proper functionality and integration of 'mutex_init' and 'mutex_lock' and 'mutex_unlock' system calls, enabling synchronisation across applications.

Remarks

This document presents an overview of the mutex system and the key functions that manage synchronization. While the explanations here highlight the main aspects of the system, the code files themselves contain more detailed comments that provide an in-depth understanding of each function's implementation. These comments are intended to clarify the logic behind each system call and demonstrate how the system functions as a whole.

Function-Bound Process Spawning

Introduction

This document provides a step-by-step explanation of the changes made to various files in the **xv6-riscv** operating system to implement **Function-Bound Process Spawning**. This functionality is achieved using 'clone' and 'join' system calls, which allow the creation of processes directly bound to a specified function, bypassing the standard program loader. Each change is discussed with its purpose and necessity, highlighting how this approach differs from traditional threading mechanisms.

System File Changes and Updates

Note: The comments `// <implemented>` indicate the parts where new functionality was created, while `// <modified>` denotes the areas where existing code was modified or updated.

1. Makefile

The following lines were added to the 'UPROGS' section:

```
UPROGS=\
    ...
    $U/_testthread\
    $U/_testthreadnum\
```

Purpose: Adds the threading test programs to the build process to ensure they are compiled and included during the project build.
Explanation: This ensures that the user-level test programs '_testthread' and '_testthreadnum' are compiled and linked during the build process, enabling verification of threading functionality.

2. defs.h

Added declarations:

```
// syscall.c
int argint(int, int*);  // <modified>
uint64 clone(void(*func)(void*, void*), void*, void*, void*);  // <imp.>
uint64 join(void**);  // <implemented>
```

Purpose: Declares functions to be used across multiple files.

Explanation:
- 'argint' was modified to return the argument it fetches for easier usage in 'sys_clone' and 'sys_join'.
- 'clone' and 'join' were declared to support threading, enabling function calls for thread creation and joining.

3. proc.h

Added fields in 'struct proc':

```
// Per-process state
struct proc {
    struct spinlock lock;
    pde_t* pgdir;  // <implemented>
    void *threadstk;  // <implemented>
    ...
};
```

Purpose: To track thread-specific properties.
Explanation:
- 'pgdir' tracks the shared page directory for threads sharing the same address space.
- 'threadstk' stores the stack pointer for threads, enabling proper cleanup during 'join'.

4. syscall.c

Modified and implemented new functions:

```
// Fetch the nth 32-bit system call argument.
int argint(int n, int *ip) {  // <modified>
    *ip = argraw(n);
    return *ip;  // <modified>
}

// System call prototypes
extern uint64 sys_clone(void);  // <implemented>
extern uint64 sys_join(void);  // <implemented>

// System call mapping
static uint64 (*syscalls[])(void) = {
    ...
    [SYS_clone] sys_clone,  // <implemented>
    [SYS_join] sys_join,  // <implemented>
};
```

Purpose:
- ‘argint’: Simplifies fetching arguments for system calls.
- ‘sys_clone’ and ‘sys_join’: Implements system call handlers for thread creation and joining.
Explanation: Mapping ‘SYS_clone’ and ‘SYS_join’ to their respective handlers allows user programs to call these system calls.

5. syscall.h

Added system call numbers:

```
// System call numbers
#define SYS_clone  22  // <implemented>
#define SYS_join   23  // <implemented>
```

Purpose: Assigns unique identifiers for the new system calls.
Explanation: These identifiers are used in ‘usys.pl’ and other files to link user-level function calls to the corresponding kernel handlers.

6. user.h

Added user-level declarations:

```
// System calls
int clone(void(*func)(void*, void*), void*, void*, void*);  // <implem.>
int join(void**);  // <implemented>
```

Purpose: Declares user-level APIs (Application Programming Interfaces) for threading.
Explanation: Enables user programs to call ‘clone’ and ‘join’ directly.

7. ulib.c

Implemented helper functions:

```
int threadcreate(void(*func)(void*, void*), void *arg1, void *arg2); //
void threadjoin(int*);  // <implemented>
```

Purpose: Simplifies threading by providing user-friendly abstractions over ‘clone’ and ‘join’.
Explanation:
- ‘threadcreate’ encapsulates ‘clone’ to allocate a stack and invoke the system call.
- ‘threadjoin’ abstracts ‘join’ for stack cleanup.

8. usys.pl

Added entries for new system calls:

```
entry("clone");  # <implemented>
entry("join");  # <implemented>
```

Purpose: Generates user-level system call stubs (placeholders).
Explanation: Links 'clone' and 'join' system calls to their kernel implementations via assembly stubs (placeholders).

9. proc.c

The `clone` and `join` system calls were implemented to support thread creation and management in the kernel.

i. clone System Call

Declaration:

```
uint64
clone(void(*func)(void*, void*), void *arg1, void *arg2, void *stk);
```

Purpose: Creates a new thread by allocating a new process, copying the calling thread's memory, file descriptors and other state, and setting up the new thread to execute a specified function on a new stack.
Why in `proc.c`: The `proc.c` file contains process management functions, and `clone` is responsible for creating a new thread by allocating resources and duplicating process state, which aligns with process management.
Explanation: Allocates a new process, copies memory and file descriptors from the parent process, sets up the function to execute and returns the new thread's PID on success.

ii. join System Call

Declaration:

```
uint64 join(void** stk);
```

Purpose: Waits for a child thread in the same process to terminate, retrieves its stack pointer and performs cleanup.
Why in `proc.c`: The `proc.c` file is responsible for process and thread lifecycle management, and `join` is essential for synchronizing threads by waiting for their termination and performing cleanup.
Explanation: Iterates through the process table to find child threads, waits for their termination and returns the thread's PID while performing necessary cleanup operations.

10. sysproc.c

The `sys_clone` and `sys_join` system calls were implemented to handle thread creation at the user-space interface level.

Declarations:

```
uint64 sys_clone(void);
uint64 sys_join(void);
```

Purpose: These system calls serve as the interface between user programs and the kernel's thread management functions, allowing user programs to create new threads and wait for existing threads to terminate.

Why in `sysproc.c`: `sysproc.c` serves as the repository for system calls that manage processes and process-related activities. Since `sys_clone` and `sys_join` directly manage thread creation and synchronization, they are logically placed in `sysproc.c`. This file already handles the lower-level process management functionality, making it a natural location for implementing system call interfaces that interact with the kernel's threading mechanisms.

Explanation:

- `sys_clone`: This function is a wrapper for the `clone()` function, which facilitates the creation of a new thread by cloning the current process. It retrieves the necessary arguments (function pointer, arguments and stack pointer) from the user space and invokes `clone()` to create a new thread. The result returned is typically the thread ID of the newly created thread or a failure condition.

- `sys_join`: This function is a wrapper for the `join()` system call, enabling the calling process to wait for the completion of a child thread. It retrieves the stack pointer argument from the user space, calls `join()` to block until the child thread terminates, and returns the result from `join()` (typically the thread's PID or failure indication).

11. ulib.c

The `threadcreate` and `threadjoin` functions were implemented to facilitate user-level thread creation.

Declarations:

```
int threadcreate(void(*func)(void*, void*), void *arg1, void *arg2);
void threadjoin(int *t);
```

Purpose:

- `threadcreate`: To provide a high-level interface for creating new threads by wrapping the `clone()` system call, making it easier for user applications to spawn threads.
- `threadjoin`: To offer a simplified way to wait for a thread to complete its execution and retrieve its stack pointer.

Why in `ulib.c`: These functions are placed in user-space libraries to provide an abstraction layer over system calls. By doing so, they simplify the creation and management of threads for application developers, enabling higher-level threading functionality without requiring direct interaction with low-level system calls.
Explanation:

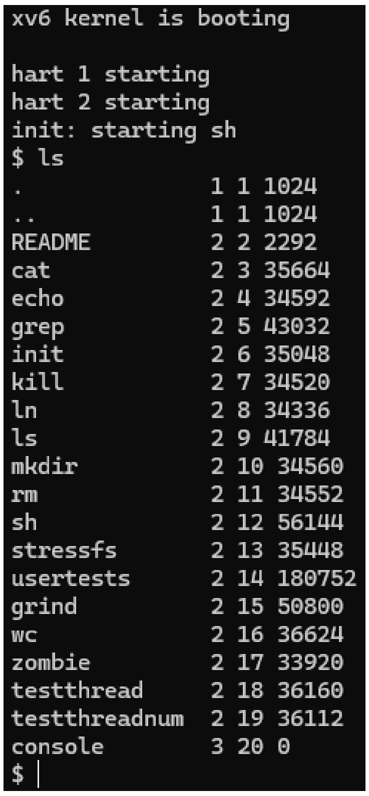
- `threadcreate`: This function is a wrapper for the `clone()` system call, simplifying the process of creating a new thread. It allocates a page of memory for the thread's stack using `malloc()` and then calls `clone()` with the stack pointer, function and arguments. The return value of `clone()` is typically the thread identifier or an error code if the operation fails.
- `threadjoin`: This function is a wrapper for the `join()` system call, which waits for a specific thread to finish execution. It uses `join()` to block the calling process until the thread terminates and retrieves the thread's stack pointer. While the `res` variable stores the return value of `join()`, it is unused beyond this and includes a dummy operation to prevent compiler warnings.

User Programs

1. `testthread`

Purpose: This program showcases the creation, execution and synchronization of threads in a user-level environment using custom system calls like `threadcreate` and `threadjoin`.
Explanation: The program demonstrates the following key aspects of thread management:

- **Thread Creation:**
  - Three threads are created using the `threadcreate` system call.
  - Each thread executes a dedicated function (`f1`, `f2`, `f3`), which prints a message, sleeps for a predefined duration, and then exits.



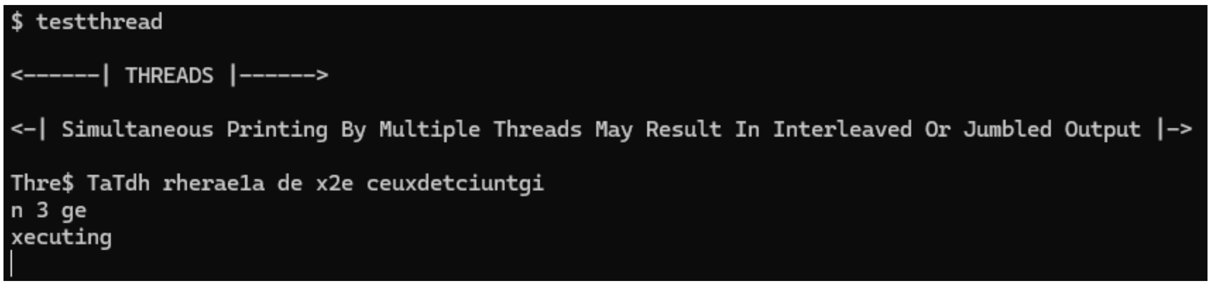
```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ ls
.              1 1 1024
..             1 1 1024
README         2 2 2292
cat            2 3 35664
echo           2 4 34592
grep           2 5 43032
init           2 6 35048
kill           2 7 34520
ln             2 8 34336
ls             2 9 41784
mkdir          2 10 34560
rm             2 11 34552
sh             2 12 56144
stressfs       2 13 35448
usertests      2 14 180752
grind          2 15 50800
wc             2 16 36624
zombie         2 17 33920
testthread     2 18 36160
testthreadnum  2 19 36112
console        3 20 0
$ |
```

Figure 5: `testthread and testthreadnum`

- **Thread Execution:**
  - Each thread runs concurrently, potentially interleaving outputs due to simultaneous execution.
  - The `sleep` system call ensures that threads pause execution for a specified period, simulating some workload or delay.
- **Thread Synchronization:**
  - The `threadjoin` system call is used by the main thread to wait for each thread to complete.
  - This ensures the main program only exits after all threads finish execution, maintaining a synchronized flow.

Significance: This program demonstrates multi-threading in a minimal operating system environment, highlighting the use of system calls for creating and managing lightweight processes.

Execution:



```
$ testthread

<------| THREADS |------>

<-| Simultaneous Printing By Multiple Threads May Result In Interleaved Or Jumbled Output |->

Thre$ TaTdh rheraela de x2e ceuxdetciuntgi
n 3 ge
xecuting
|
```

Figure 6: Output of `testthread`

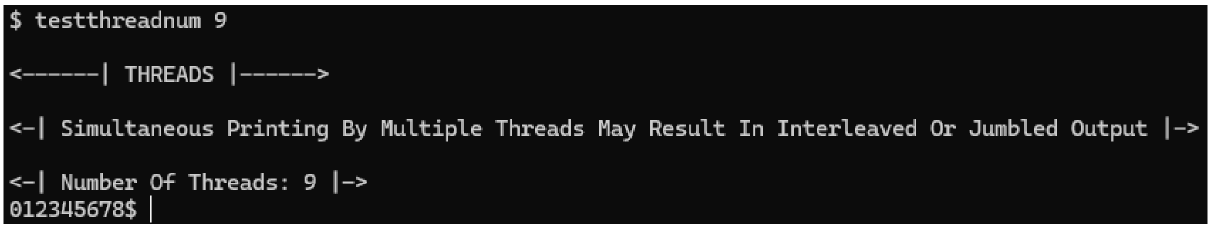
2. `testthreadnum`

Purpose: This program demonstrates the creation, execution and synchronization of multiple threads in a user-level environment. It showcases thread management by taking the number of threads to create as a command-line argument and ensuring all threads complete execution before the main thread exits.
Explanation: The program illustrates the following aspects of thread management:

- **Thread Creation:**
  - The number of threads to create is provided as a command-line argument.
  - Using the `threadcreate` system call, the specified number of threads is spawned, each executing a function (`func`) that prints its thread number, sleeps for a defined duration and then exits.
  - A unique thread identifier is assigned to each thread, enabling tracking and synchronization.
- **Thread Execution:**
  - Each thread runs concurrently, with execution managed by the operating system's scheduling.
  - The threads execute the `func` function, which involves:
    - Printing the thread's unique identifier.
    - Sleeping for a predefined duration to simulate processing time.
    - Exiting after the task is complete.
- **Thread Synchronization:**
  - The main thread uses the `threadjoin` system call to wait for each thread to complete execution.

- This ensures that the main program does not terminate until all threads have finished their tasks, maintaining a synchronized workflow.
- The stack memory allocated for threads is properly managed and cleaned up after thread completion.

Significance: This program demonstrates fundamental concepts of multi-threading in a simplified user-level environment, providing insights into thread creation, execution and synchronization using system calls. It serves as an educational example for understanding how lightweight processes operate in an operating system.
Execution:



```
$ testthreadnum 9

<------| THREADS |------>

<-| Simultaneous Printing By Multiple Threads May Result In Interleaved Or Jumbled Output |->

<-| Number Of Threads: 9 |->
012345678$ |
```

Figure 7: Output of `testthreadnum`

Conclusion

This documentation explains the changes made to implement **Function-Bound Process Spawning** in **xv6-riscv.** These modifications enable the creation of processes directly tied to specific functions using the 'clone' and 'join' system calls. This approach facilitates efficient execution of function-specific processes.

Remarks

This document presents an overview of the **Function-Bound Process Spawning** mechanism and the key system calls that manage process creation and synchronization. While the explanations here cover the primary modifications and functionality, the code files contain additional detailed comments to provide further clarity on each implementation. These comments aim to explain the logic and integration of the 'clone' and 'join' system calls, showcasing how the system is designed to handle targeted function execution within newly spawned processes.