Winfried Schuffert
Dustin Lee
Felix Kosmalla

Assignment 2

Design Document

Introduction

During this assignment we want to extend OS161 in a way so that it is possible to run userland programs. For this, we need to implement several system calls, for starting and managing processes. Furthermore we will implement system calls for file manipulation.

First we will introduce the several parts of this assignment in a higher level, followed by in-depth analysis of the single parts. These should serve as roadmap for the implementation.

Overview

Identifying processes

Processes are identified by *process ids* (*PID*s). These obviously have to be unique, otherwise we could not unambiguously identify the process which is e.g. needed for waitpid. Furthermore, process IDs have to be reusable. This is necessary if an OS runs for a long time with many starts and stops of processes. Each process is assigned an ID, intuitively the ID would increment for each process. This eventually leads to the limit of the integer representation. Thus, IDs must be reusable but it is essential that IDs still remain unique (e.g. only ‘old’ IDs of processes which terminated should be allowed to be used.) Implementation details can be found in the following sections.

File Descriptors

Since processes should not be allowed to access the filesystem directly the OS must provide an abstraction layer. This is done by file descriptors. A file descriptor is an integer, held by a process, which points to a file resource provided by the operating system. A file descriptor is acquired by calling *open().* It can then be used to *read()* or *write()* a file and eventually to *close()* it. This file resource should be organized in a custom struct which holds the actual vnode reference, the mode (reading or writing), and the current offset.

Extension of process structure

For the functions that we have to implement for this assignment we need further fields in the Process structure.

- pid *// for each process unique*
- child_process_list *// list of forked child processes*
- child_process_list_lock *// lock for child_process_list*
- parent *// pointer to parent process structure*
- returnvalue *// to store the return value of the process (in _exit(int ret))*
- file_descriptor_table *// hash table with currently opened files*

Fork

fork() is the mechanism for other user processes to create new processes. It makes an exact copy of the invoking process. It creates a thread (each user process has exact one thread) and copies the address space, program code, a list of file_descriptors from files the parent process is using, working directory of the parent process and makes sure that the parent and child processes each observe the correct return value (that is, 0 for the child and the newly created pid for the parent). Furthermore the parent and the child process get suited with a link to each other such that they can somehow communicate to afford the possible for a parent process to wait for it's child processes (waitpid()) until they exits.

execv()

execv() is a syscall used to begin executing a user program. Unlike thread_fork, it just replaces the current state of execution with a new, different one. That means that it needs its own address space to load its own code, stack, resources, etc. while the old stuff is freed. Since there can only be one active address space at a time, program arguments must be loaded into the kernel from the original userspace before being moved into the new userspace as arguments. Additionally, because the old execution environment is replaced, execv() does not return to the caller, and should close all of the file descriptors except in, out, and err. However, it keeps the same process id as it is still the same process. The thread array also remains unchanged since the execution continues from the start of the new program.

Waitpid/Exit

waitpid() is a syscall that allows a parent process (process that forked child processes before) to wait for one of its child processes to exit. For this feature we can use the already implemented thread mechanism here. That means, in case the parent calls the waitpid() syscall, the OS changes to kernel mode to serve the syscall. Since we created the child

process thread joinable it remains until it was joined. When the child process already exited, and the parent calls waitpid() it moves on without waiting. If the parent process waits for one of its child processes that does not have exit yet, it waits in the already implemented semaphore (with 0 initialized) of the child process thread (tries to decrease).
The implementation of _exit() is intimately connected to the implementation of waitpid(). They are essentially two halves of the same mechanism. _exit() is used when a process exits. In case the process has children (parent process) it notifies its children not to remain if they exit. And in case the process is a child, it notifies its parent by increasing its semaphore value. Either that wakes the waiting parent process or it causes that the parent does not have to wait when calling waitpid() on it.

kill_curthread()

When the processor's error handler catches an exception, it calls kill_curthread() to stop the thread's process. Should clean up any resources the process struct itself has, like file descriptors and pid, but doesn't attempt to do anything in userspace and just throws away the address space since it may be corrupted.

In Depth Analysis

ID Allocation

As already mentioned in the overview, a running process is identified by a unique ID, the PID. The ID has to be unique and must be designed to be reusable. This leads to the following design of the ID allocation process.

Whenever a new process is started it obtains it's new ID from a superior allocation system. In general the interface of this system should be as simple as a single function returning a 'new' unique process ID: *int get_new_process_id().*

The system must stand above the CPU layer since PIDs must be unique across multiple CPUs (in case the parent process is running on CPU A and the child process running on CPU B). A counter which is used to hold the current PID number is incremented every time a new process number is demanded. Eventually the counter will reach its limit and needs to start again at 0. Then it needs to find an 'old' free PID. This sets the requirement that after a process exits, the ID allocation system has to be notified that a PID is again available. An interface could look like this: *void free_process_id(int pid);* Besides the counter, the allocation systems needs to keep track which PIDs are busy and which ones are free. This leads to the following constraints.

Limit of process IDs

In a perfect system where memory is unlimited it would be possible that the number of running processes equals the number of available PIDs. In this case either the new process could not be started or it would have to wait until another process terminates and releases it's PID.

Since our systems has limited memory it should be safe to define a fixed number of possible PIDs. When reaching this number the counter would start from the beginning.

Synchronization

Since our OS can be run using multiple CPUs it would be possible that two process are spawned at the same time, requiring a PID also at the same time. This means that the allocation system must utilize proper locking of the counter and the list of free PIDs.

Implementation Details

kern/include/kern/limits.h already sets the lower and upper limit of PID numbers (2 and 32767).

The following pseudo code should outline the general concept of the allocation system.

```
int counter     = __PID_MIN
list freepids   = list()
lock pid_lock

int get_new_process_id(){
     lock.acquire();

     if(!freepids.empty()){
          return freepids.pop();
     }

     if(counter==__PID_MAX){
          return -1;
     }



     int return_pid = ++counter;

     lock.release();
```

```
        return return_pid;
    }

    void free_process_id(int pid){
        lock.acquire();
        freepids.append(pid);
        lock.release();
    }
```

When there are no more Process IDs left fork should return ENPROC;

getpid()

returns curthread->proc->pid

Testing

- test the upper limit of the tests, get_new_pid should return an error
- test if the queue works
- test general case
- test corner cases if queue is empty but counter not max
- test corner cases if queue is empty and counter is max -> error
- test corner case if queue is empty but _had_ items in it
- test lower limit (should be the same as defined in limits.h)

File Descriptors

A file descriptor struct could look like the following

```
    struct file_descriptor{
        struct vnode *file;
        int mode;
        int offset;
        lock lock;
    }
```

These file descriptors should be saved in a hashmap for fast access. Additionally to the hashmap a list of file_descriptors (or their ids) must be maintained to copy the file descriptors when forking the process.

open(const char *filename, int flags, mode_t mode)

On open of a new file, the system has to check if there is a non-occupied spot left in the file descriptor table. This could be done similar to finding a new PID. If there is no spot left, the system should respond with an appropriate error message (EMFILE). Otherwise it should call vfs_open() and create the file_descriptor struct. This can then be used by future system calls to actually read, write, or close the file. vfs_open itself will respond with an error code in case of non existing files / wrong flags. This will be forwarded to the user.

read(int fd, userptr_t buf, size_t buflen)

Since we don't want to give the user access to the actual memory location in which the file is saved, we have to copy the part of the file which the userland program wants to read into userland memory, i.e. write the data into *buf.* Error codes will be forwarded to the user.

For this, the following steps have to be performed.

- create a *struct iovec*
  this holds the buffer *buf* and the length *buflen*
- create a *struct uio* and link the *iovec*
- set uio_offset according to the file_descriptor offset
- set uio_resid to something?
- set uio_segflg to UIO_USERSPACE
- set uio_space to the address space of the calling process
- set uio_rw to UIO_READ
- Call VOP_READ on the uio and vnode
- update the file_descriptor.offset according to uio.uio_offset

write(int *fd*, const void **buf*, size_t *nbytes*)

Write follows the same concept of *read*

For this, the following steps have to be performed.

- create a *struct iovec*
  this holds the buffer *buf* and the length *buflen*
- create a *struct uio* and link the *iovec*
- set uio_offset according to the file_descriptor offset
- set uio_resid to something?

- set uio_segflg to UIO_USERSPACE
- set uio_space to the address space of the calling process
- set uio_rw to UIO_WRITE
- Call VOP_WRITE on the uio and vnode
- update the file_descriptor.offset according to uio.uio_offset

close(int fd)

Call vfs_close() on the vnode and strip down the file_descriptor.

synchronization problems

Multiple processes should be allowed to have a file handler on the same file. However, it must not be possible for two or more processes to write to the same file at the same time, but it would be convenient if multiple processes could read from the file at the same time. Our file_descriptor struct points to a vnode which is eventually used to write the contents of a file. Since multiple file_desriptor structs can point to the same vnode, the locking mechanism has to be placed in the vnode, respectively in the read and write functions.

Since multiple processes should be able to read one particular file at the same time but only one process should be able to write, we have to implement a mechanism which can handle this. For this, we will orient ourselves towards the ‘Readers & Writers’ lecture.

Since during a process fork a references of the file descriptor will be copied into a new process, it is possible that the same file_descriptor points at the same vnode. This has to be taken care of when close() is called on a file descriptor. We have to add a reference count field to the file_descriptor struct which will be incremented on open and decremented on close. When the count hits there the file_descriptor can be destroyed and the vnode really closed.

Testing

- test sequential reading / writing with a sequence of
    - open
    - write
    - close
    - open
    - read
    - close
- Do this for every mode (e.g. create if not existent, append, etc.)
- Test simultaneous reading (x threads reading a file with a predefined content)

- Test simultaneous writing (write A, B, C, with the help of three threads and see if either ABC, ACB, CAB, CBA, BAC, BCA comes out)
- Test file creation modes
- Test error codes
- test faulty user inputs

Process Management

execv(const char *program, char **args)

Remember: execv replaces the current process with a new, different one while fork duplicates the current process. execv() itself is not machine dependent and so can be in the syscalls directory since it uses enter_new_process, which is machine dependent.

- make sure args is null terminated
- vfs_open the program's file
- copyin() up to ARG_MAX (64KB) of user provided arg strings into kernel heap while still in old userspace using pointers from args
- create, switch to, and then activate a new address space
- load_elf() the program into the new userspace, save the entrypoint
- vfs_close the program's file
- define stack with as_define_stack()
- copyout() args from kernel into the new address space
- put those addresses into a new userspace argv
- close non-standard file descriptors
- call enter new process with the new argv, its count, null for the environment, the new stack pointer, and the entrypoint

If any of the syscalls it uses return an error, it should stop and return that status code. The main thing to test is that the arguments are passed correctly. You can do this with a script that just tests/prints the arguments it receives. Should try it with varying sizes of arguments, between 0 and 64KB. Another thing to test is concurrency; even if multiple threads call execv, only one should be copying args in and out at once to save memory.

pid_t fork(void)

- From the syscall() function we get the parent trap frame as argument.
- We try to do as much tests as possible at the start of fork() so that we can fail early if necessary.
- Check if there are already too many processes on the system
  if yes -> error: ENPROC & return -1
- Check if current user already has too many processes
  if yes -> error: EMPROC & return -1
- Allocate new pid (getpid()) and assign it to new child process

- Create trapframe for child and copy all values of the parret trapframe
- Create new address space and copy the content from parent address space to child address space (as_copy(parent_as, child_as)) (this should copy the memory content from the parent to the child space)
- Create a new process skeleton (proc_create_runprogram(name)), whereas name is given by the “child_” + child pid
- Create child process thread
- Add parent information to the child process thread (initialises the semaphore for the waitpid mechanism)
- Create file descriptor table for child process and copy content of parent file descriptor table into it.
- Create open file table for child and copy content of parent open file handle table into it. Remove files from the child open file table that are opened as writable in the parent list.
- Check if sufficient virtual memory for the new process was available (that means check if all create operations succeed)
  if not -> error: ENOMEM + return -1 + cleanup
- Add child process to child_process_list of parent process
- Set parent variable in child to current parent
- Modify return value of child trapframe to 0 (trapframe->v0 = 0)
- Modify return value of parent to the child pid (trapframe->v0 = child pid)
- return (if no error occured)

Testing

- test forking several processes
- test if one can fork more processes than he is allowed
- test fork the max number of processes + 1

pid_t waitpid(pid_t pid, int *status, int options)

- Check if pid argument named a nonexistent process
  if yes -> error: ESRCH & return -1
- Check if pid argument named a process that was not a child of the current process (check child_pid_list of parent)
  if yes -> error: ECHILD & return -1
- Check if status argument is an invalid pointer (status == NULL)
  if yes -> error: EFAULT & return -1
- Check if options argument requested invalid or unsupported options (options != 0)
  if yes -> error: EINVAL & return -1

- The parent wants to decrement the semaphore of the child process thread. The child process thread increments the semaphore if it exits. If the child has not finished (exit) yet, the parent will wait until the child exits. If the child already finished, parent don't has to wait.
- After a successfully wait (child process exits and parent process is up again and grabbed the return value of the child which is saved in the childs process struct)
- Clean up child process ressources (free allocated space and pid)
- Return child pid

Testing

- test function call with different arguments (also with invalid ones) to check if we reach all branches
- create a process that forks childs and wait on them
    - test different scenarios:
        - exit child regular
        - child exits with fatal error
        - child exits before parent starts waiting
        - child exits after parent starts waiting

void _exit(int exitcode)

- Does process has child processes? (!empty(child_process_list))
    - if yes:
        - destroy all semaphores from the childs process threads (we don't need them anymore)
        - set variable parent from all its childs structs to NULL
        - Clean up process ressources (free allocated space and pid) of all its child processes that are waiting that parent "joins". Here we can also use the thread join mechanism again. The system call triggers the kernel to change the state of the waiting child process threads such that the cpu can cleanup the ressources.
- Check if process has parent (parent != NULL)
    - if yes:
        - store exitcode in process struct (returnvalue = exitcode)
        - increment thread semaphore to maybe wake up the parent
        - process structure remains until parent "joins" or until it gets cleaned up
    - if no:
        - Clean up process ressources (free allocated space and pid)

Testing

- test exit with different scenarios:
    - exit (process is not child and has no childs)
    - exit (process is child)
    - exit (process is parent)

kill_curthread(vaddr_t epc, unsigned code, vaddr_t vaddr)

- cleanup struct process:
    - close vnodes in fd table
    - free fd table
    - erase address space of process
    - based on implementation of waitpid, wake any waiters, and give them the status code
    - free process struct

Risk Analysis

File Handlers

The highest risk of the implementation of file handlers are the arise of errors concerning concurrency and security. Errors in concurrency can be tested to a limited amount using unit tests. We are trying to avoid security errors by carefully reading the code concerning transferring data from userland to kernel and vice versa.

Time Estimation

open 2 hours
write 2 hours
read 2 hours
close 2 hours
synchronization structures 4

writing tests 2 hours

Total 14

Process Identification

The creation and management of process IDs should not hide high risks since it can be easily tested. This is also valid for get_process ID.

Time Estimation

get_new_process_id() 2 hours
testing 1 hour
getpid() 1 hours
testing 1 hour

Total 5

Process Management

execv()

The args parameter has to be checked for correctness: it must be null terminated, and each element must be a valid userspace address of a valid string, otherwise it might cause a memory fault or improper copy. When copying the arguments into and out of the kernel, you have to be certain which pointers are in kernel space, the old user space, and the new user space, and using user_ptr's correctly should prevent that. However, we aren't sure where exactly the arguments should be copied to in the new userspace. Only one thread should be copying args into and then out of the kernel at one time since it potentially takes up 64K memory. Should also make sure the program isn't overwritten while it's being read, but that's probably the responsibility of read()/write().

waitpid()/exit()/waitpid()/kill_curthread()

To avoid concurrency problems we want to secure the child_process_list with a lock.

Sometimes it can happen, depending on the user programs, that a user process crashes. In this case we can run into trouble with our wait and exit mechanism. To avoid this we have to extend the kill_curthread() by the exit functionality:

- When a child dies (e.g. fatal error) the parent process is notified by removing the child process from its childe_process_list, such that the parent is not being able to wait for this process if it tries to. If the parent process is already waiting for the child process, wake it up. Here we can again run into trouble if we do not check again if the child

process is still in our child_process_list. To avoid this problem we have to check the child_process_list, also each time we wake up.
- When a parent dies (e.g. fatal error) or just exits without waiting for any of its childs, the parent will also notify the children, so the children can exit without notifying the parent.

Furthermore, it can cause undefined problems if e.g. a parent and one of its childs die/exit simultaneously on different cpus. To avoid deadlocks here we want to acquire locks in the same order to avoid exclusion.

Time Estimation

struct extension 0.25 hours
fork() 4 hours
waitpid() 4 hours
exit() 2 hours
execv() 5 hours
kill_curthread() 2 hour
testing 10 hours

Total 27.25

Implementation Plan

execv() needs to close the non standard file descriptors, but otherwise doesn't depend on the other stuff.

kill_curthread() and exit() should be pretty similar, and they close file descriptors and possibly handle parent/child wake up stuff.

What	**Who**	**Deadline**
Struct extension	Winfried	10/31
Process Identification	Felix	11/2
execv()	Dustin	11/3
fork() + testing	Winfried	11/4

File Handlers (incl. read/write)	Felix	11/4
waitpid() + testing	Winfried	11/8
exit() + testing	Dustin	11/10
kill_curthread() + testing	Dustin	11/10

Code Reading Questions

1) loading and running user level programs

1. **What are the ELF magic numbers?**

   The ELF magic numbers are the first few bytes that get checked when loading an ELF to ensure it is a "32-bit ELF-version-1 executable for our processor type".

2. **What is the difference between UIO_USERISPACE and UIO_USERSPACE? When should one use UIO_SYSSPACE instead?**

   They are enums for the type of memory of a source/destination pointer for uio calls. UIO_USERISPACE is for user process code, so it's executable, while UIO_USERSPACE is for user process data. UIO_SYSSPACE is for reading/writing kernel-space.

3. **Why can the struct uio that is used to read in a segment be allocated on the stack in load_segment() (i.e., where does the memory read actually go)?**

   The struct uio just defines how the uio operation executes, the read actually puts the data into the process' virtual memory, not the stack.

4. **In runprogram(), why is it important to call vfs_close() before going into usermode?**

   Since runprogram() calls vfs_open(), it increments the reference counter for the program it's running to read it, and so it must decrement with vfs_close() when it's done, before it enters usermode and doesn’t have the reference.

5. **What function forces the processor to switch into usermode? Is this function machine dependent**

   runprogram() calls enter_new_process(), which calls mips_usermode(), which is machine dependent. enter_new_process also has to set up the new process' trapframe, which is also machine dependent

6. **In what file are copyin and copyout defined? memmove? why can't copyin and copyout be implemented as simply as memmove?**

   copyin() and copyout() are defined in kern/vm/copyinout.c. memmove() is defined in common/libc/string/memmove.c. copyin/out need to make sure user addresses aren't out of bounds and handle a copy fail gracefully so the kernel doesn't crash.

7. **What (briefly) is the purpose of userptr_t?**

   userptr_t is for developer(and sometimes compiler)-enforced type-checking. If the developer sees a userptr_t, they know that it needs to be checked for validity (not out of bounds, etc.).

2) kern/arch/mips: traps and syscalls

8. **What is the numerical value of the exception code for a MIPS system call?**

   #define EX_SYS 8 /* Syscall */ (trapframe.h:91)
   Therefore, the numerical value is 8.

9. **How many bytes is an instruction in MIPS? (Answer this by reading syscall() carefully, not by looking somewhere else.)**

   After the syscall returns the program counter stored in the trapframe must be increased by 4 bytes (one instruction) to avoid restarting the syscall over and over again.

10. **Why do you "probably want to change" the implementation of kill_curthread()?**

    When we reach the point to call this function, a fatal exception in the user process occurred and the current thread's userspace state can't be trusted anymore. To save the kernel and to avoid a possible kernel panic, we need to clean up the user process

by taking it off the processor in as judicious a manner as possible, but without returning execution to the user level.

11. **What would be required to implement a system call that took more than 4 arguments?**

    The first 4 arguments are passed through the argument registers a0 - a3 and further arguments must be fetched from the user-level stack, starting at sp+16.

3) MIPS & Syscall

1. **What is the purpose of the SYSCALL macro?**
   The SYSCALL macro creates a set of MIPS instructions to do an actual system call via the MIPS *syscall* instruction. For this, it just copies the syscall number into the *v0* register and then jumps to the __syscall label which handles the syscall.

2. **What is the MIPS instruction that actually triggers a system call? (Answer this by reading the source in this directory, not looking somewhere else.)**
   syscall

3. **After reading syscalls-mips.S and syscall.c, you should be prepared to answer the following question: OS/161 supports 64-bit values; lseek() takes and returns a 64-bit offset value. Thus, lseek() takes a 32-bit file handle (arg0), a 64-bit offset (arg1), a 32-bit whence (arg3), and needs to return a 64-bit offset value. In void syscall(struct trapframe *tf) where will you find each of the three arguments (in which registers) and how will you return the 64-bit offset?**
   Since we have arguments of mixed lengths, aligned pairs of registers have to be used. This results in the following register usage:

a0	arg0
a1	*unused*
a2	first half of arg1
a3	second half of arg2

   Since we only have 4 argument registers, the third argument *arg2 (whence)* is stored on the user-level stack starting at sp+16.

   The return value of lseek will be stored in the v0 and v1 registers since we have a

64-bit return value.