UXP - Projekt wstępny (W13, W23)

Data przekazania projektu wstępnego - 25.04.20
Data przekazanie dokumentacji końcowej - 10.06.20

Skład zespołu:

- Rafal Babinski
- Mateusz Kordowski
- Radosław Mierzwa
- Roman Moskalenko

Treść: Napisać wieloprocesowy system realizujący prosty system plików (w przestrzeni użytkownika, nie jądra). System powinien udostępniać następujące funkcje (zachowanie analogiczne do systemowych odpowiedników)

```
int simplefs_open(char *name, int mode);
int simplefs_creat(char *name, int mode);
int simplefs_read(int fd, char *buf, int len);
int simplefs_write(int fd, char *buf, int len);
int simplefs_lseek(int fd, int whence, int offset);
int simplefs_unlink(char *name);
int simplefs_mkdir(char *name);
int simplefs_rmdir(char *name);
```

System powinien być zrealizowany w obszarze pamięci współdzielonej zachowującym się jak uproszczony dysk logiczny, tj. powinien on zawierać: tablicę i-node-ów, katalog główny, mapę wolnych bloków, obszar na pliki. Należy zaimplementować tylko pliki zwykłe i katalogi.
Implementacja powinna pozwalać na jednoczesny dostęp wielu procesów, tak więc należy zrealizować wykluczanie i wielodostęp, nie na poziomie całego mini-systemu plików, ale możliwie „drobno-ziarniście”, tj. na poziomie poszczególnych obiektów (katalogów, tablicy inode-ów). Wykluczanie zrealizować przy pomocy semaforów sem_*(<semaphore.h>).

Dodatkowe założenia i wymagania:

- Kontrolę dostępu na poziomie U/G/O należy pominąć (całość działa w kontekście procesów i „plików” tego samego użytkownika), wskazane jest aby zróżnicować tryby otwarcia R/W, można pominąć prawo X dla katalogów

- można pominąć flagi: APPEND, TRUNC, NODELAY, itp.
- można przyjąć statyczny rozmiar: katalogu głównego, pod katalogów i tablicy i-nodeów (bez relokacji)
- metryczka pliku może być maksymalnie uproszczona (nie musi odzwierciedlać struktury stat)
- można założyć ciągłą alokację z opcjonalnym prostym mechanizmem defragmentacji

Interpretacja treści, uściślenia

W ramach zadania realizujemy bibliotekę do zarządzania systemem plików w pamięci współdzielonej, która udostępnia API (funkcje, które mamy zaimplementować) programom użytkownika.

Przyjmujemy następujące założenia:

- Zakładamy, że system plików będzie miał stały rozmiar (256MiB) oraz zostanie zaalokowany jednorazowo w jednym ciągłym obszarze przy tworzeniu systemu plików.

- Maksymalna liczba inode'ów w systemie wynosi 65536.

- Foldery w systemie posiadają strukturę drzewiastą.

- Zakładamy blokową alokację pamięci.

- Zakładamy, że procesy, korzystające z systemu, kończą się poprawnie. Także zakładamy, że programista korzystający z API dokonuje zamknięcia otwartych plików (funkcja simplefs_close).

Opis funkcjonalny, API

W ramach API udostępniamy następujące funkcje:

int **simplefs_open**(char *name, int mode);

Opis:

- Funkcja pozwala na powiązanie pliku z deskryptorem pliku oraz jego otwarcie w trybie zgodnym z podanym w argumencie *mode.* Przypisany przez tę funkcję deskryptor pliku będzie służył innym funkcjom I/O do wskazywania tego pliku.
- Funkcja *simplefs_open*() zwraca deskryptor pliku dla nazwanego pliku, będący najmniejszym nie używanym w chwili otwarcia deskryptorem dla danego procesu.
- Położenie kursora I/O z pliku w momencie otwarcia zawsze wskazuje na początek pliku.
- Zmienne statusu pliku oraz trybu dostępu do niego są ustawiane zgodnie z podanym trybem otwarcia.

Argumenty:

- *name* to ścieżka do pliku, który ma zostać otwarty.
- *mode* to wybór trybu otwarcia pliku.

Dostępne tryby otwarcia pliku to dokładnie jedno spośród:

- O_RDONLY – otwarcie pliku tylko do odczytu
- O_WRONLY – otwarcie pliku tylko do zapisu
- O_RDWR – otwarcie pliku do zapisu i odczytu

Wartość zwracana:

W przypadku poprawnego wykonania funkcja otwiera plik i zwraca nieujemną liczbę całkowitą będącą najmniejszym nie używanym do tej pory deskryptorem. W przeciwnym wypadku funkcja zwraca ujemną wartość powiązaną z odpowiednim kodem błędu.

Błędy:

- **EINVAL** – wartość -2

Wartość podana jako tryb otwarcia jest niepoprawna.

- **ENOTDIR** - wartość -3

Element prefiksu ścieżki nie jest katalogiem.

- **EMFILE** – wartość -16

Osiągnięta już została maksymalna liczba {OPEN_MAX} deskryptorów plików w danym procesie.

- **ENAMETOOLONG** - wartość -4

Długość ścieżki przekroczyła dozwoloną wartość {PATH_MAX} lub nazwa elementu ścieżki przekroczyła długość {NAME_MAX}.

- **ENFILE** – wartość -5

Została już osiągnięta maksymalna liczba otwartych plików w systemie.

- **ENOENT** – wartość -6

Wskazany przez ścieżkę plik nie istnieje bądź ścieżka jest łańcuchem pustym.

int **simplefs_creat**(char *name, int mode);

Opis:

Tworzy nowy plik o podanej w argumencie *name* ścieżce, jeżeli nie istnieje i zwraca deskryptor do istniejącego już pliku i otwiera go zgodnie z podanym w zmiennej *mode* trybem. Tryby są identyczna jak dla funkcji *simplefs_open()*.

Wartość zwracana:

patrz *simplefs_open()*.

Błędy:

patrz *simplefs_open()*.

int **simplefs_read**(int fd, char *buf, int len);

Opis:

- Funkcja *simplefs_read()* dokona próby odczytu *len* bajtów z pliku powiązanego z deskryptorem pliku *fd* do bufora wskazanego przez *buf*.
- Odczyt pliku rozpocznie się w miejscu wskazywanym w danym momencie przez kursor I/O danego pliku. Pozycja kursora jest przesuwana wraz z odczytem kolejnych bajtów pliku.
- Nie dochodzi do przesyłu danych w przypadku, gdy kursor znajduje się za położeniem końca pliku. Jeżeli kursor znajduje się na końcu pliku lub za nim zostanie zwrócone 0.
- W przypadku poprawnego wykonania funkcji, gdy *len* jest większe od 0, simplefs_*read*() zwróci liczbę odczytanych bajtów, nie większą niż *len*. Wartość zwrócona może być mniejsza od *len* jeżeli w pliku zostało mniej bajtów do odczytania niż *len.*

Argumenty:

- *fd* – deskryptor pliku, z którego ma nastąpić odczytem
- *buf* – bufor, do którego zostaną zapisane odczytane dane
- len – ile danych ma zostać odczytanych

Wartość zwracana:

W przypadku sukcesu zwraca nieujemną liczbę całkowitą reprezentującą liczbę przeczytanych bajtów. W przeciwnym przypadku zostanie zwrócona wartość całkowita ujemna powiązana z określonym błędem.

Błędy:

- **EBADF** – wartość -7

Argument *fd* nie jest poprawnym deskryptorem pliku wskazującym na plik otwarty do odczytu.

- **EIO** - wartość - 8

Błąd I/O na poziomie fizycznym.

- **EINVAL** – wartość -2

Położenie kursora jest niewłaściwe, przed początkiem pliku.

- **EOVERFLOW** – wartość -9

Została dokonana próba odczytu poza obszarem, w którym jest dopuszczone jego dokonanie dla danego pliku.

int **simplefs_write**(int fd, char *buf, int len);

Opis:

- Funkcja *simplefs_write()* dokona próby zapisu *len* bajtów z bufora wskazanego przez *buf* do pliku powiązanego z deskryptorem *fd*.
- Zapis do pliku odbywa się od pozycji wskazywanej przez kursor I/O powiązany z deskryptorem pliku. Wraz z zapisem przesuwa się pozycja kursora.
- Zapisanych zostanie tyle bajtów na ile pozwala maksymalny rozmiar pliku w systemie lub wielkość fizycznego nośnika.

Argumenty:

- *fd* – deskryptor pliku, do którego ma nastąpić zapis
- *buf* – bufor, z którego zostaną pobrane dane do zapisu
- len – ile danych ma zostać zapisanych

Wartość zwracana:

W przypadku sukcesu zwraca nieujemną liczbę całkowitą reprezentującą liczbę zapisanych bajtów, nie większa od *len*. W przeciwnym przypadku zostanie zwrócona wartość całkowita ujemna powiązana z określonym błędem.

Błędy:

- **EBADF** – wartość -7

Argument *fd* nie jest poprawnym deskryptorem pliku wskazującym na plik otwarty do zapisu.

- **EFBIG** – wartość -10

Próba zapisu, w której liczba zapisywanych bajtów przekracza maksymalny rozmiar pliku. Nie było miejsca do dokonania zapisu.

- **ENOSPC** – wartość -11

Brak wolnej pamięci na urządzeniu zawierającym plik.

- **EINVAL** – wartość -2
  Kursor jest w nieprawidłowym położeniu - offset jest ujemny.

int **simplefs_lseek**(int fd, int whence, int offset);

Opis:

- Funkcja *simplefs_lseek*() ustawia położenie kursora (offset pliku) dla otwartego pliku powiązanego z deskryptorem *fd*.

Argumenty:

- *fd* – deskryptor pliku, do którego ma nastąpić zapis
- *whence* – sposób ustawienia offsetu
- *offset* – wartość offsetu w bajtach

Wartości *whence*:

- SEEK_SET - wartość offsetu ma być ustawiona na *offset.*
- SEEK_CUR – wartość offsetu ma być ustawiona na obecny offset plus *offset*.

Wartość zwracana:

W przypadku powodzenia funkcja zwraca uzyskany offset liczony w bajtach od początku pliku. W przeciwnym przypadku wartość zostanie zwrócona wartość ujemna odpowiadająca określonemu kodowi błędu.

Błędy:

- **EBADF** – wartość -7

Argument *fd* nie jest poprawnym deskryptorem pliku wskazującym na otwarty plik.

- **EINVAL** - wartość -2

Argument *whence* jest nieprawidłowy, bądź wynikowy offset miałby wartość ujemną.

int **simplefs_unlink**(char *name);

Opis:

- Ponieważ w naszym systemie plików nie są przewidziane dowiązania sztywne i symboliczne dana funkcja w przypadku gdy wskazany plik nie jest otwarty przez któryś z procesów usuwa go. Zajmowany przez niego obszar pamięci jest zwalniany i jest on już więcej niedostępny. W przypadku, gdy jest usuwany plik jest otwarty przez któryś z procesów zostanie zwrócony błąd EBUSY.

Argumenty:

- *name* to ścieżka do pliku, który ma zostać otwarty.

Wartość zwracana:

0 w przypadku sukcesu. W przeciwnym przypadku ujemny kod błędu.

Błędy:

- **EBUSY** – wartość -12

Nie można usunąć pliku, gdyż jest on używany przez system bądź inne procesy.

- **ENAMETOOLONG** - wartość -4

Długość ścieżki przekroczyła dozwoloną wartość {PATH_MAX} lub nazwa elementu ścieżki przekroczyła długość {NAME_MAX}.

- **ENOTDIR** - wartość -3

Element prefiksu ścieżki nie jest katalogiem.

- **ENOENT** – wartość -6

Wskazany przez ścieżkę plik nie istnieje bądź ścieżka jest łańcuchem pustym.

int **simplefs_mkdir**(char *name);

Opis:

- Funkcja *simplefs_mkdir()* tworzy nowy pusty katalog o ścieżce *name*.

Argumenty:

- *name* to ścieżka do katalogu, który ma zostać stworzony.

Wartość zwracana:

0 w przypadku sukcesu. W przeciwnym przypadku ujemny kod błędu.

Błędy:

- **EEXIST** – wartość -13

Istnieje już katalog lub plik o tej nazwie.

- **ENAMETOOLONG** - wartość -4

Długość ścieżki przekroczyła dozwoloną wartość {PATH_MAX} lub nazwa elementu ścieżki przekroczyła długość {NAME_MAX}.

- **ENOTDIR** - wartość -3

Element prefiksu ścieżki nie jest katalogiem.

- **ENOENT** – wartość -6

Wskazany przez ścieżkę folder nie istnieje bądź ścieżka jest łańcuchem pustym.

- **ENOSPC** – wartość -11

Brak wolnej pamięci w systemie plików.

int **simplefs_rmdir**(char *name);

Opis:

- Funkcja *simplefs_rmdir()* usuwa katalog wskazany przez ścieżkę *name.* Jest on usuwany wyłącznie, gdy jest pusty.

Argumenty:

- *name* to ścieżka do katalogu, który ma zostać usunięty.

Wartość zwracana:

0 w przypadku sukcesu. W przeciwnym przypadku ujemny kod błędu.

Błędy:

- **ENAMETOOLONG** - wartość -4

Długość ścieżki przekroczyła dozwoloną wartość {PATH_MAX} lub nazwa elementu ścieżki przekroczyła długość {NAME_MAX}.

- **ENOTDIR** - wartość -3

Element prefiksu ścieżki nie jest katalogiem.

- **ENOENT** – wartość -6

Wskazany przez ścieżkę plik nie istnieje bądź ścieżka jest łańcuchem pustym.

- **ENOTEMPTY –** wartość -15

Usuwany katalog nie jest pusty.

int **simplefs_close**(int fd);

Opis:

- Funkcja kończy powiązanie deskryptora z plikiem, tak że można ponownie wykorzystać dany deskryptor.

Wartość zwracana:

0 - w przypadku sukcesu, ujemny kod błędu w przeciwnym przypadku.

Błędy:

- **EBADF** – wartość -7

Argument *fd* nie jest poprawnym deskryptorem pliku wskazującym na otwarty plik.

Z biblioteki mogą korzystać na raz wiele procesów. Zapewniamy spójność systemu, synchronizacja realizowana za pomocą semaforów.
- Synchronizacji poddane są wszystkie operacje na systemie plików. Synchronizacja jest wszyta w bibliotekę, z punktu widzenia użytkownika jest niewidoczna.

- Dodatkowo oferujemy funkcje do monitorowania stanu systemu:
  - Wypisanie liczby plików w systemie, liczby folderów, zajętych inode'ów.
  - Wyliczenie sumarycznego wolnego miejsca w systemie oraz wyświetlenie zajętości pamięci.
  - Wyświetlenie w formie tekstowej drzewa plików reprezentującym hierarchię folderów i plików.
  - Wypisanie nazw otwartych plików, ich liczby oraz trybów otwarcia.

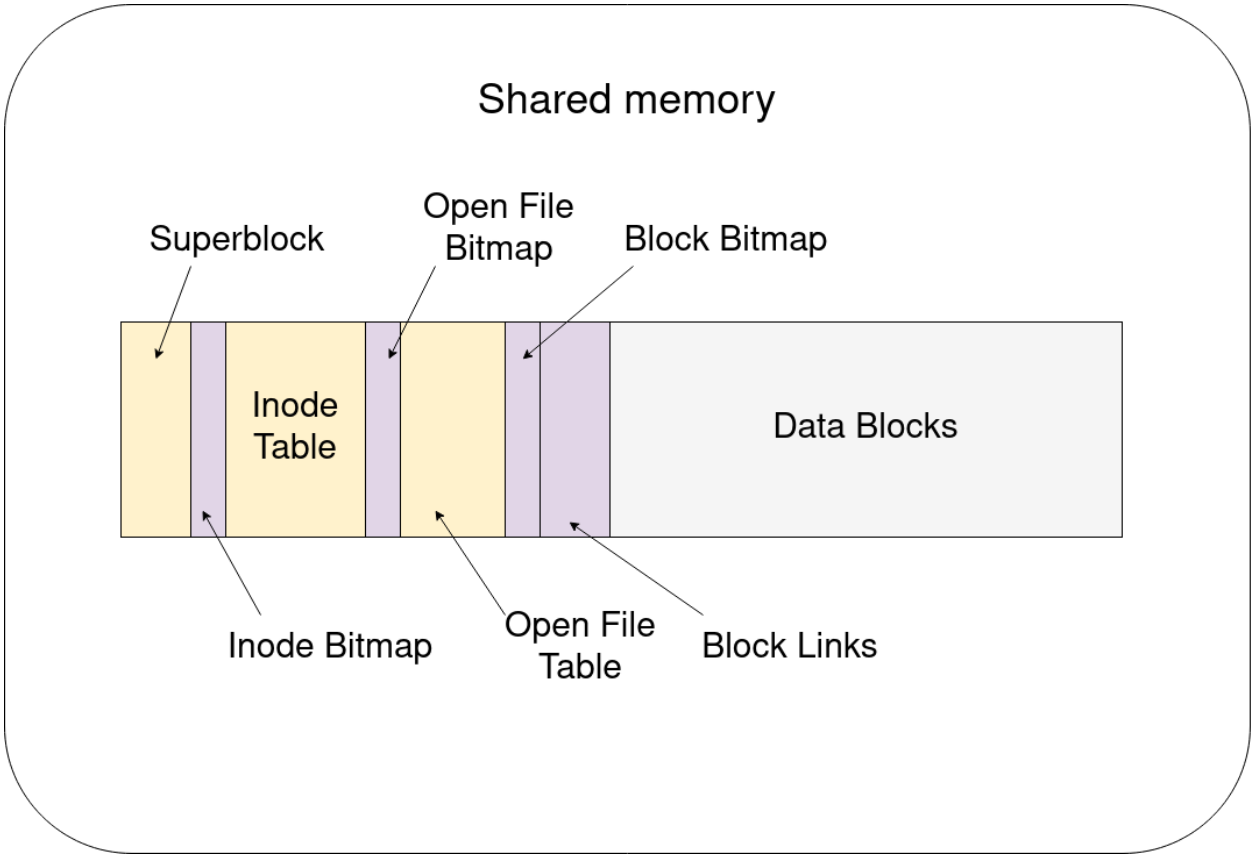
Opis architektury systemu plików

Struktura systemu plików w pamięci współdzielonej

Założenia:

- Alokacja blokowa.
- Każdy blok ma rozmiar 1024 bajtów.
- W tablicy inodów zerowy inode będzie pusty,
- Pierwszy inode będzie reprezentował katalog główny.
- Każdy plik bądź folder zajmuje swój inode.
- Liczba folderów jest ograniczona jedynie dostępnością inodów oraz pamięci.
- Maksymalna liczba inode'ów 2^16 = 65536.
- Katalog główny (/) ma już zaalokowany plik, gdzie jego pierwszy blok jest pod indeksem 0.

Schemat poniżej opisuje segmenty systemu plików o stałej długości



```
struct superblock { // sum 352 b, 44 B
        uint16 max_number_of_files; // 16
        uint16 filesystem_checks; // 16
        uint16 data_block_size; // 16
        uint32 fs_size; // 32
        uint32 open_file_table_pointer; // 32
        uint32 open_file_bitmap_pointer; // 32
        uint32 block_links_pointer // 32
        uint32 block_bitmap_pointer; // 32
        uint32 inode_bitmap_pointer; // 32
        uint32 inode_table_pointer; // 32
}
```

superblock - posiada wszystkie dane dotyczące stanu systemu plików. Blok zawiera:

- inode_used - liczba zajętych inode'ów.
- data_block_size - rozmiar jednego bloku danych.
- fs_size - rozmiar całego systemu plików
- used_data_blocks - liczba użytych bloków na dane.
- mem_used - śledzi, ile pamięci zostało użyte w data_block.
- max_number_of_files - maksymalna liczba plików (inodów)
- filesystem_checks - flagi bitowe przechowujące jakieś pomocnicze informacje na temat systemu plików.
- open_file_table_pointer - wskazanie na pierwszą strukturę open_file.
- inode_table_pointer - wskazanie na pierwszy inode.
- block_links_pointer - wskazanie na pierwszy blok na dane.
- inode_bitmap_pointer - wskazanie na lokalizację bloku inode_bitmap.
- block_bitmap_pointer - wskazanie na lokalizację bloku block_bitmap.
- Open_file_bitmap_pointer - wskazanie na lokalizację bloku open_file_bitmap.

```
struct inode_stat {
        bool inode_bitmap[NUMBER_OF_INODES];  // typ danych to 1 bit // 128B
        uint16 inode_used; // 16 // liczba wszystkich inodów w użyciu.
}
```

inode_bitmap - blok pamięci o określonym rozmiarze, mówiący o tym, które inody są wolne. Domyślny typ danych ma 8 bitów, jednak można je przeglądać jako 64, 32 lub 16 bitowe zmienne. Jeżeli maksymalna ilość inodów nie będzie wielokrotnością 8 bitów, to i tak zostanie zużyte dodatkowe 8 bitów na zapisanie stanu pozostałych inodów. Traktujemy to, jako mapę bitową, więc bit o wartości 1 na określonej pozycji będzie

mówił o tym, że inode na odpowiadającej mu pozycji jest wolny. Dzięki temu poprzez porównanie z 0 będzie można łatwo sprawdzić, które segmenty bloku inodów mają co najmniej jeden wolny inode.

```
struct inode_table {
        inode itable[NUMBER_OF_INODES];
}
```

inode_table - blok w pamięci o określonym rozmiarze, w którym przechowywane są struktury inode. Dostęp do tych struktur odbywa się za pomocą indeksów (od 0 do n), które wskazują na określone inody. Inody o numerach 0 oraz 1 są specjalne i nie można ich usunąć.

```
struct inode { // sum 96b, 12 B
        uint32 block_index; // 32
        uint16 file_size; // 16
        char mode; // 8
        uint8 ref_count; // 8
        uint8 readers; // 8
        uint8 padding[3] // 8
}
```

inode - przechowuje dane odnośnie konkretnego pliku lub folderu.
Struktura zawiera:

- mode - prawa dostępu R/W. Pozwala również na zawarcie informacji o tym, czy dany inode jest dla katalogu czy dla zwykłego pliku. Jeżeli ma ustawioną odpowiedni bit, to oznacza, że inode nie jest zajęty
- file_size - rozmiar pliku. Służy do wyliczenia adresu końca pliku. Rozmiar w bajtach.
- block_index - wskaźnik na początek danych w pliku. Dla folderu wskazuje na miejsce pliku mieszczącym katalog o strukturze dir_file. Jeżeli ma wartość 0, to oznacza, że inode nie jest zajęty, ponieważ blok 0 jest już udostępniony dla katalogu głównego.

```
struct block_links {
        uint32 block_num[BLOCK_NUMBER]; // zależy od tego, ile
                                                    // pamięci przydzielimy
}
```

struct **block_stat** {
    bool block_bitmap[BLOCK_NUMBER]; // typ danych to 1 bit // 128B
    uint32 used_data_blocks // 32
}

BLOCK_NUMBER - liczba bloków do użytku w segmencie bloku na dane.

block_bitmap - blok pamięci o określonym rozmiarze, który na pojedynczych bitach wskazuje, które bloki pamięci są zajęte, a które puste.

block_links - tablica o rozmiarze wynikającym z maksymalnej możliwej liczby bloków na dane w systemie plików. Indeksy odpowiadają określonemu blokowi, a wartość w danym indeksie może zawierać następujące informacje (są to wartości liczbowe):

- next_block_id - numer następnego bloku w łańcuchu. Ostatni blok ma wartość EMPTY_BLOCK.

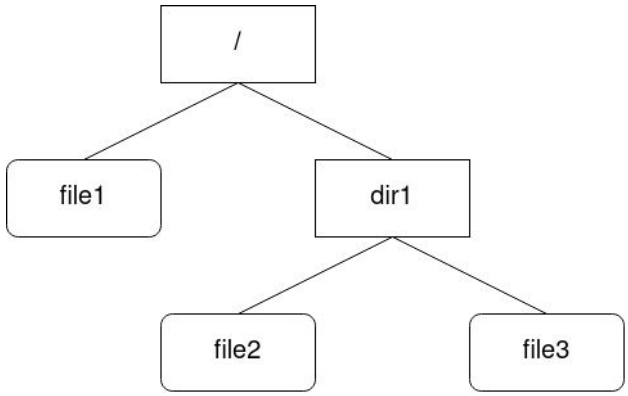
Struct **open_file_stat** {
    bool open_file_bitmap[1024]; // typ danych to 1 bit // 128B
    uint16 opened_files; //16
}

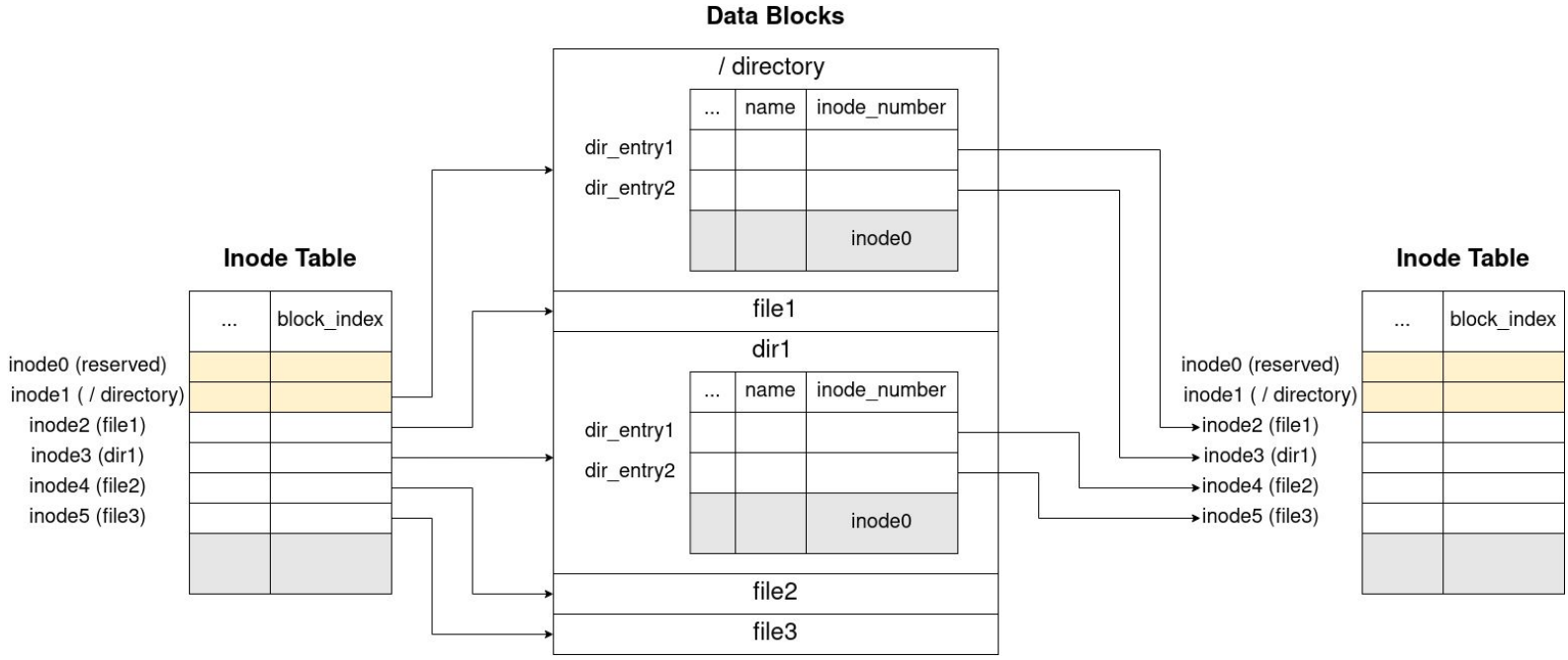
open_file_bitmap - blok pamięci o określonym rozmiarze, który na pojedynczych bitach wskazuje, które struktury otwartych plików są zajęte, a które puste.

open_file_table - tablica otwartych plików, która przechowuje struktury open_file. Może pomieścić z góry określoną liczbę danej struktury.

data_block - jest to niezajęta część pamięci, w której zostaną zapisane właściwe dane plików.

Na rysunkach poniżej pokazano przykładową strukturę plików w systemie plików, oraz odpowiadającą jej strukturę inode'ów i bloków pamięci.






```
struct dir_entry { // sum 512 b, 64 B
        char name[61]; // 488
        uint8 name_len; // 8
        uint16 inode_number; // 16
}
```

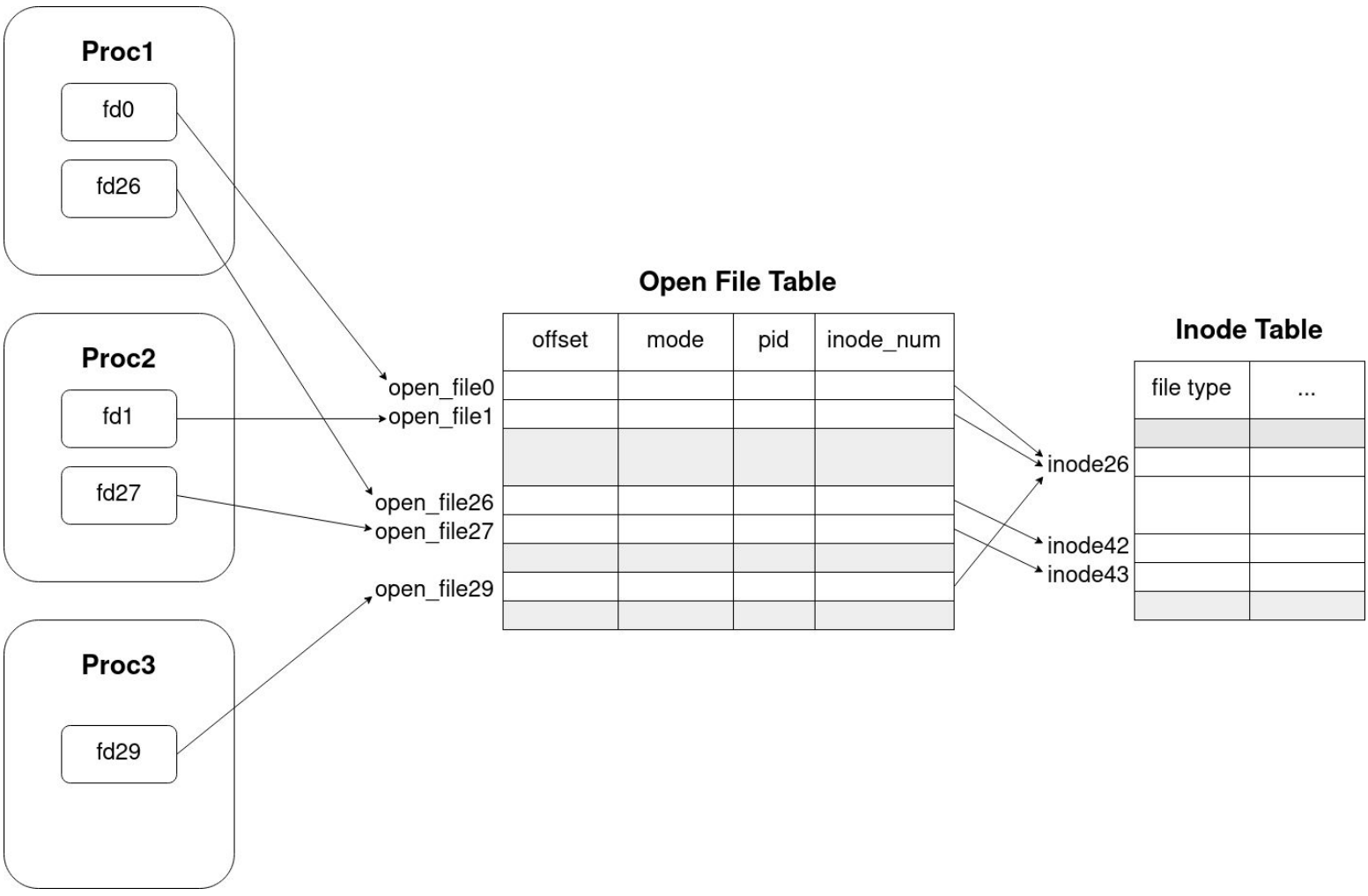
dir_entry - służy do połączenia nazwy z plikiem lub katalogiem. Jest ona zawarta w pliku określającym powiązania między folderem, a jego plikiem lub innym folderem. Zawiera:

- inode_number - numer inoda pliku. Nieużywane będzie miało wartość 0.
- name - nazwa pliku, znaki po 8 bitów, maksymalnie 61 znaków.
- name_len - długość nazwy w bajtach.

```
struct dir_file { // sum 1024 B
        struct dir_entry entry[16]; // 1024 B
}
```

dir_file - jest to plik zawierający wiele struktur dir_entry. Jest on tworzony w bloku data_block, gdzie może przyjmować z góry określone rozmiary w wielokrotności wielkości pojedynczego bloku danych.

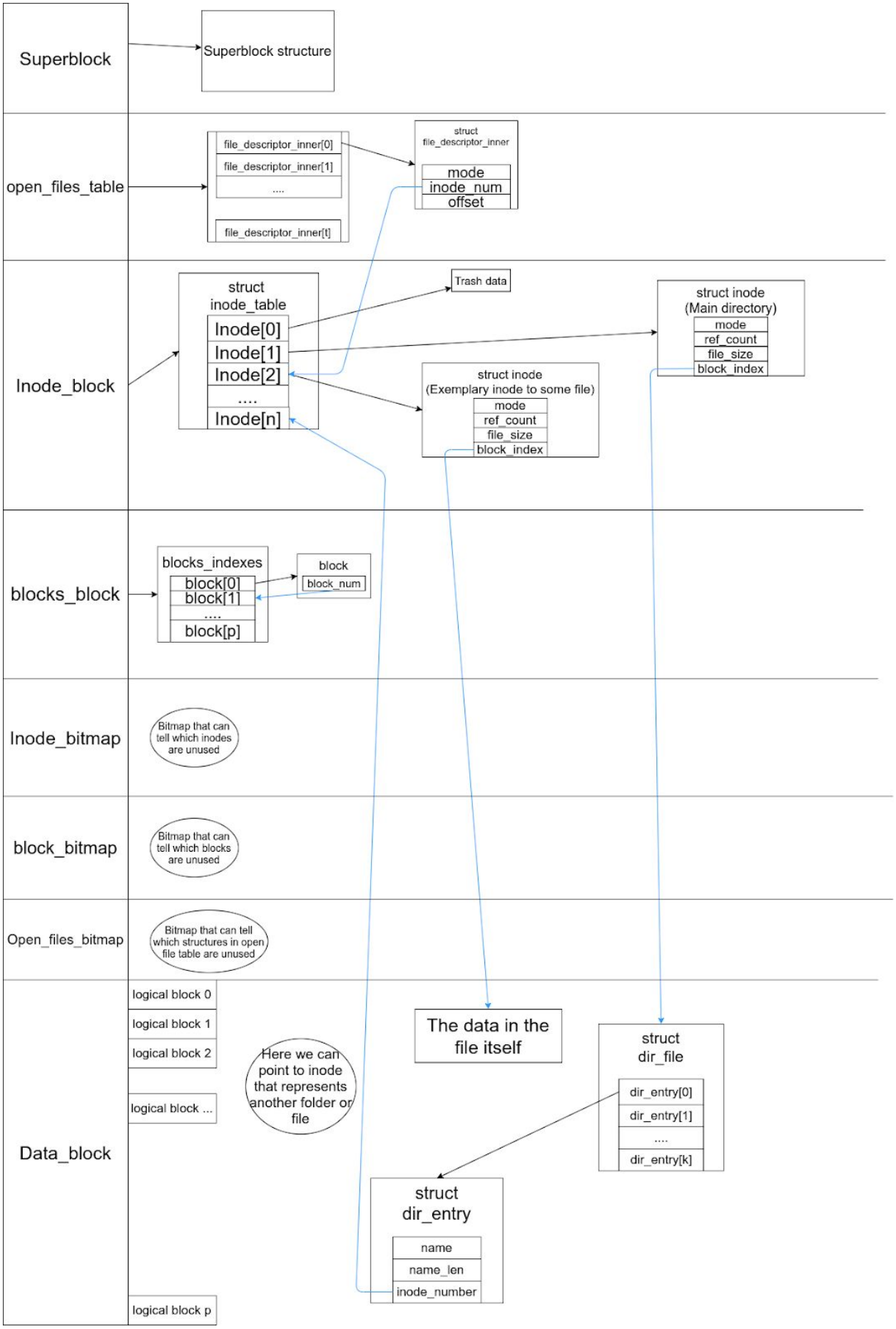
Na rysunku poniżej pokazano przykładowe procesy i ich otwarte pliki.



```
struct open_file { sum 96 b, 12 B
        uint16 mode; // 16
        uint16 inode_num; // 16
        uint32 offset; // 32
}
```

open_file - struktura zawierająca się w tablicy open_file_table. Odpowiada ona za tworzenie powiązań pomiędzy procesami, a plikami poprzez uchwyt, który jest wartością int, identyfikującą dany open_file. Nie jest możliwe, aby dwa procesy miały ten sam open_file, ponieważ nie udostępniono takiej funkcji. Z tego względu open_file jest wskazywany tylko przez jeden proces. Procesy mogą mieć wiele uchwytów. Uchwyt do danego open_file jest pozycją, na której ta struktura znajduje się w tablicy open_file_table. Struktura zawiera:

- mode - uprawnienia deskryptora do R / W. Nie oznacza to, że inode będzie posiadał takie same flagi. Jeżeli inode nie jest zajęty, wtedy zostanie odpowiednio zmodyfikowany. W przeciwnym wypadku proces musi poczekać.
- offset - wskaźnik w pliku.
- inode_num - numer inoda, do którego dana struktura się odwołuje.
- pid - pid procesu, który korzysta z tej struktury.


Superblock
Superblock structure
open_files_table
file_descriptor_inner[0]
file_descriptor_inner[1]
....
file_descriptor_inner[t]
struct
file_descriptor_inner
mode
inode_num
offset
Inode_block
struct
inode_table
Inode[0]
Inode[1]
Inode[2]
....
Inode[n]
Trash data
struct inode
(Main directory)
mode
ref_count
file_size
block_index
struct inode
(Exemplary inode to some file)
mode
ref_count
file_size
block_index
blocks_block
blocks_indexes
block[0]
block[1]
....
block[p]
block
block_num
Inode_bitmap
Bitmap that can tell which inodes are unused
block_bitmap
Bitmap that can tell which blocks are unused
Open_files_bitmap
Bitmap that can tell which structures in open file table are unused
Data_block
logical block 0
logical block 1
logical block 2
logical block ...
logical block p
Here we can point to inode that represents another folder or file
The data in the file itself
struct
dir_file
dir_entry[0]
dir_entry[1]
....
dir_entry[k]
struct
dir_entry
name
name_len
inode_number

Synchronizacja

Synchronizacja będzie zapewniona przy pomocy semaforów nazwanych (semaphore.h) używając funkcji *sem_open*.

Semafory:

/inode_stat - semafor na bitmapę inode'ów i inode_table

/block_stat - semafor na bitmapę bloków i block_links.

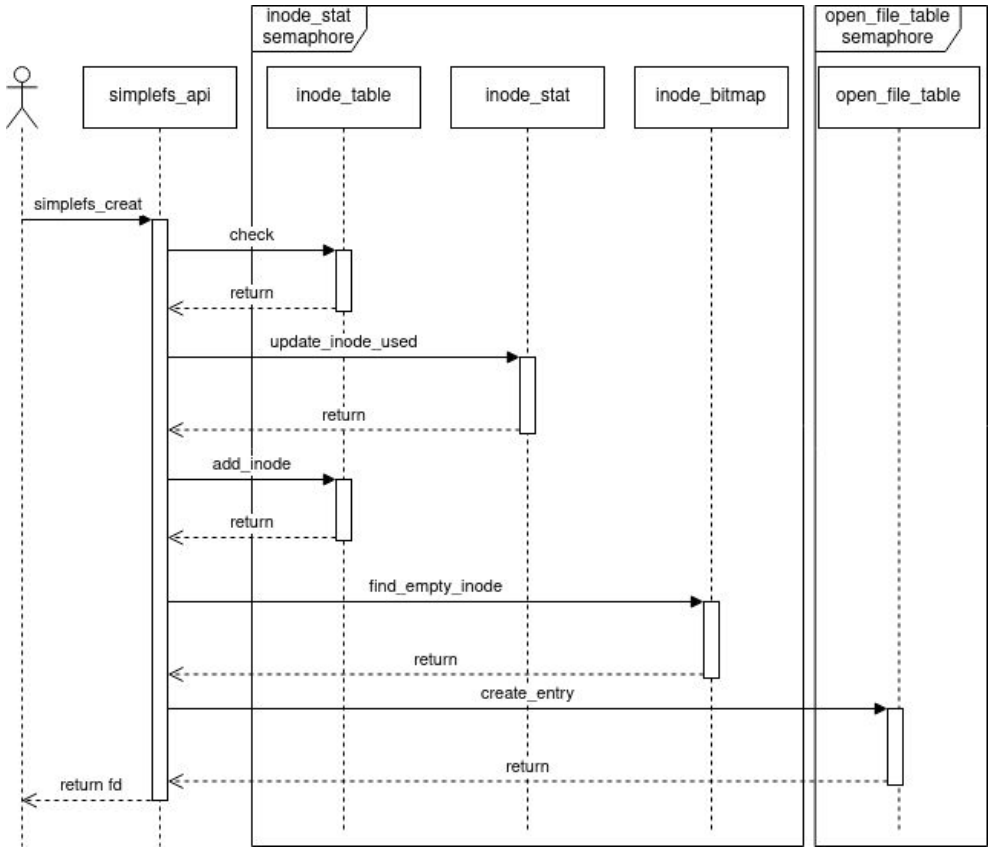
/open_file_stat - semafor dla tablicy otwartych plików i bitmapy.

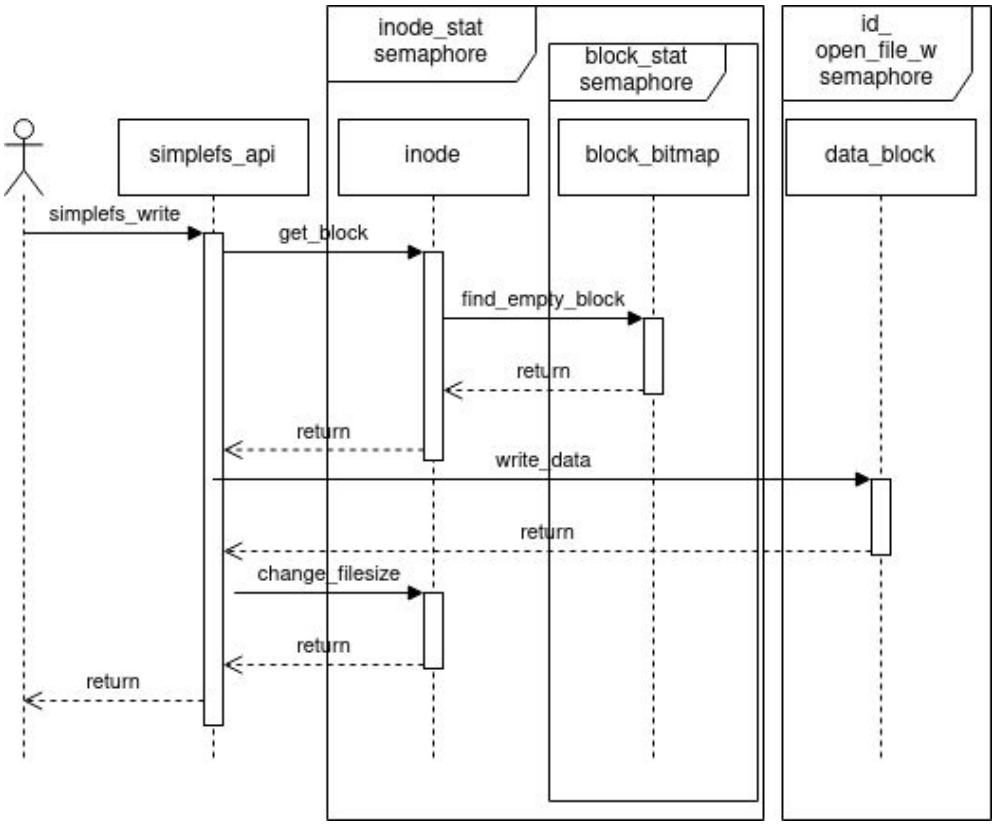
/id_open_file_r - semafor na plik do czytania. Id w nazwie mówi o id konkretnej struktury w tablicy otwartych plików.

/id_open_file_w - semafor na plik do pisania. Id w nazwie mówi o id konkretnej struktury w tablicy otwartych plików.

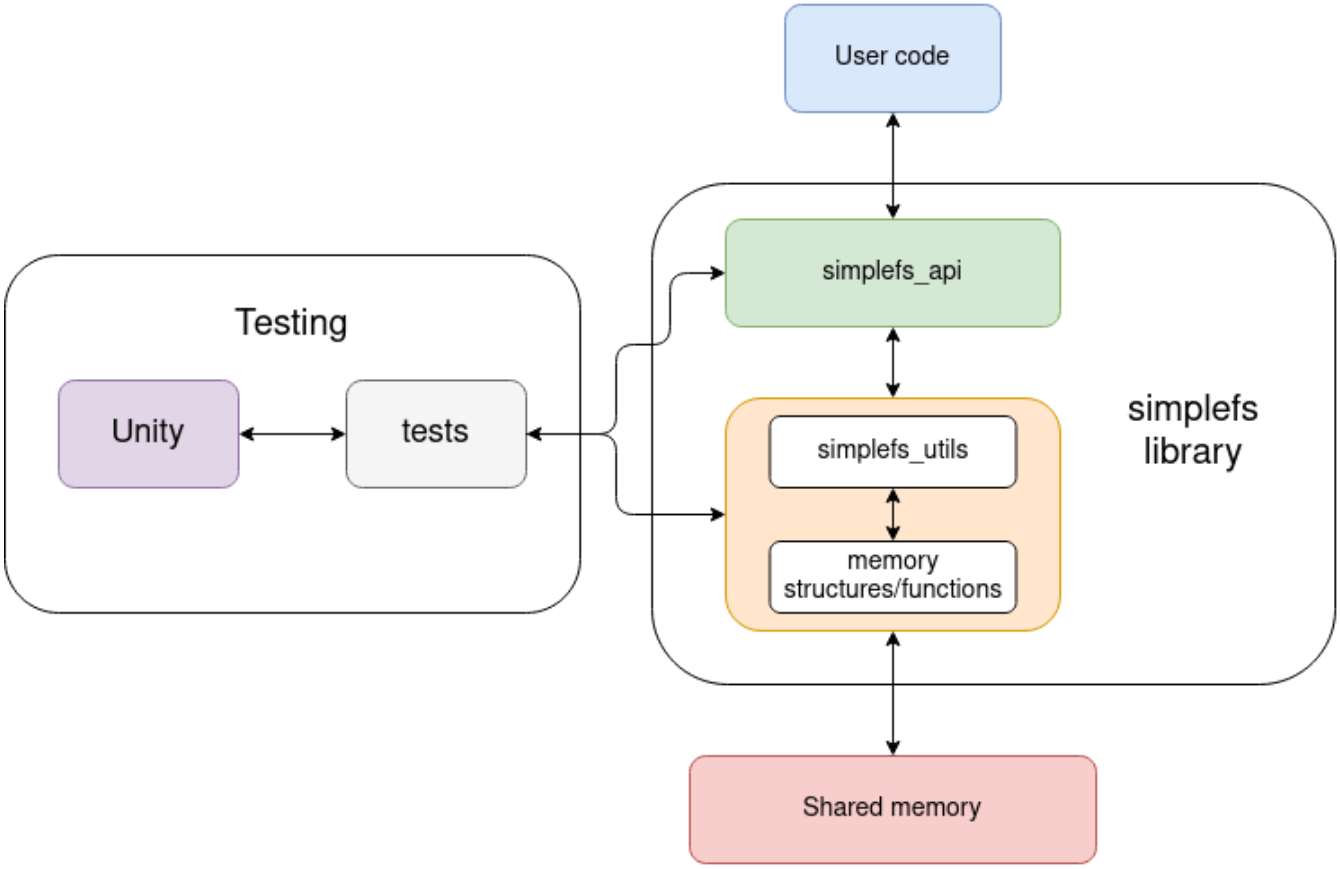
Semafory istnieją cały czas za wyjątkiem semaforów dotyczących inode'ów, które to są usuwane, kiedy ich licznik referencji spadnie do 0. Ma to zmniejszyć zapotrzebowanie na zasoby systemu hostującego.
Semafory dotyczące określonych inode'ów w problemie czytelników i pisarzy, będą służyły temu, aby uśpić inne procesy, które będą chciały wykonać określoną operację na danym inodzie.

Na następnej stronie umieściliśmy diagramy sekwencji, które opisują jaki w tym momencie przewidujemy sposób synchronizacji.


inode_stat
semaphore
open_file_table
semaphore
simplefs_api
inode_table
inode_stat
inode_bitmap
open_file_table
simplefs_creat
check
return
update_inode_used
return
add_inode
return
find_empty_inode
return
create_entry
return
return fd


inode_stat
semaphore
block_stat
semaphore
id_
open_file_w
semaphore
simplefs_api
inode
block_bitmap
data_block
simplefs_write
get_block
find_empty_block
return
return
write_data
return
change_filesize
return
return

Podział na moduły:



Na rysunku zilustrowano poglądowy podział projektu na mniejsze części:

- "User code" - pliki źródłowe użytkownika, który chce skorzystać z api udostępnianego przez naszą bibliotekę. Przy kompilacji do nich zostanie dołączona biblioteka.

- "simplefs library" - realizowana biblioteka do korzystania z systemu plików oraz zarządzania nim w obszarze pamięci współdzielonej.
Dzielimy ją logicznie na:
  - "simplefs_api" - zawiera funkcje, które udostępniamy użytkownikowi.
  - "simplefs_utils" - funkcje pomocnicze, mające na celu uprościć korzystanie ze struktur i funkcji bezpośrednio związanych z pamięcią współdzieloną.
  - “memory structures/functions” - część biblioteki, w której znajdują się operacje wykonywane bezpośrednio na końcowych strukturach danych systemu plików.

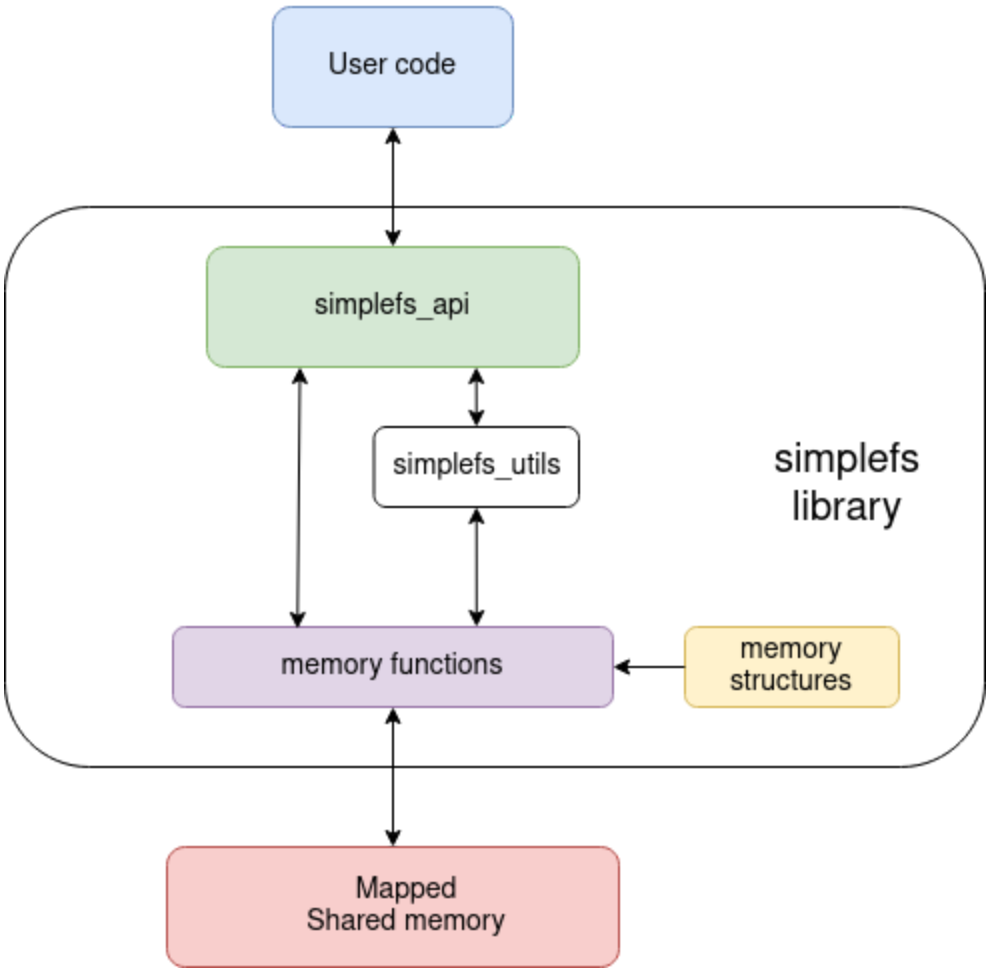
- "Testing" - testy jednostkowe do przetestowania poprawności działania biblioteki.
- "tests" - kod źródłowy testów
- “Unity” - lekki i prosty framework do testów jednostkowych w C.
- "Shared memory" - reprezentuje pamięć współdzieloną. Sposób jej wykorzystania oraz podział systemu plików na części został opisany w rozdziale dot. systemu plików.

Realizacja projektu

Zmiany projektu wstępnego

W strukturze projektu dokonaliśmy zmianę:
Wcześniejszy moduł “simplefs_internals” został zastąpiony dwoma podmodułami “simplefs_utils” oraz “memory structures/functions”.
W funkcjach zostały usunięte zbędne wartości zwracane przy błędnym zachowaniu.

Struktura komunikacji między modułami



Obrazek powyżej pomoże opisać komunikację między modułami.
Dla uproszczenia został pominięty moduł do testowania.

“Memory structures” - odpowiada za zdefiniowane w projekcie struktury danych takie jak inode_bitmap, open_file_table etc. Dla każdej struktury został napisany szereg funkcji do zarządzania daną strukturą: odczytywanie i zapisywanie wartości, usuwanie i tworzenie struktur itd. Logicznie struktury i związane z nimi funkcje zostały podzielone i zapisane w oddzielnych plikach w folderze “memory” (opis plików i folderów projektu znajduje się w rozdziale niżej).

“Memory functions” - funkcje wspomniane w poprzednim podpunkcie. Większość tych funkcji wykonuje proste jednostkowe operacje na strukturach typu zapis/odczyt. Na tych funkcjach oparte implementacje funkcji api.

“Simplefs_utils” - aby uprościć i usprawnić korzystanie z “memory functions” wprowadzono wyższy poziom funkcji, które agregują/opakowują funkcje z najniższego poziomu.

Najważniejsze rozwiązania funkcjonalne

- Dostęp do systemu plików

Na początku każdej funkcji z api musimy dostać wskaźnik na zmapowany system plików aby umożliwić dalsze operacje na nim. Żeby to zrobić stworzyliśmy specjalną funkcję **get_ptr_to_fs()**, zadaniem której jest zwrócić wskaźnik na superblock systemu plików. Ta funkcja najpierw sprawdza czy system plików już istnieje: za to odpowiada zmienna globalna, która przechowuje albo NULL, albo wskaźnik na superblock. Jeśli system plików nie został wcześniej stworzony funkcja **get_ptr_to_fs()** tworzy go. Aby uniemożliwić sytuację, w której dwa procesy jednocześnie próbują stworzyć nowy system plików zabezpieczyliśmy to za pomocą semaforów.

- Utworzenie systemu plików

System plików jest tworzony za jednym razem w całości (256 MiB). Potem jest mapowany w całości do przestrzeni adresowej procesu.

- Przekazywanie wskaźnika na system plików

Otrzymany wskaźnik na system plików musimy przekazać go każdej funkcji, która wykonuje operacje na systemie plików.

- Wprowadzone ograniczenia

System ma wpisane na sztywno ograniczenia co do rozmiaru plików (64 KiB) i długości przetwarzanych ścieżek (255 znaków przy czym każdy element ścieżki do 61 znaków).

- Superblock

Superblock zawiera takie informacje opisujące system plików, takie jak maksymalna liczba inode'ów, otwartych plików, bloków danych. Poza tym przechowuje wskaźniki na ważne struktury naszego systemu takie jak bitmapy, tablicy, obszar bloków danych.

- Parsowanie ścieżek

Aby uprościć operacje na ścieżkach skorzystaliśmy z funkcji systemowych **basename**, **dirname.**

Synchronizacja

Aby wyznaczyć gdzie występują sekcje krytyczne i jak najlepiej zabezpieczyć je przygotowaliśmy tabelę, która ilustruje jak poszczególne funkcje api wpływają na system plików i na poszczególne struktury danych.

Target inode - odpowiada za cel końcowy (plik lub folder) wywołania danej funkcji.

Wykorzystanie struktur systemu przez udostępniane funkcje

r - odczyt
w - zapis

	Inode stat	Inode Table	Target Inode +sem	Block stat	Block links	Data blocks	Open File stat	Open File Table
open		**r**	**rw**				**rw**	**w**
creat	**rw**	**rw**	**w**				**rw**	**w**
unlink	**w**	**rw**	**r**	**w**		**rw**		
mkdir	**rw**	**rw**	**w**	**rw**	**w**	**rw**		
rmdir	**w**	**r**	**r**	**w**		**rw**		
read			**r**		**r**	**r**		**rw**
write			**rw**	**rw**	**rw**	**w**		**rw**
lseek								**rw**
close			**rw**				**w**	**rw**

W ramach synchronizacji rozróżniamy 3 rodzaje plików:

- Dostępne dla użytkownika, po **2 semafory** dla otwartego pliku, faworyzacja pisarzy.
- Plik ukryty, zawiera dir_file. Nie zawierają semaforów.
- Plik specjalny, **folder /**. Synchronizowany dwoma semaforami, faworyzacja pisarzy.

A więc dla każdego otwartego pliku (ma zapis w Open File Table) stworzymy po **2 semafory**. Tylko zwykłe pliki mogą być otwarte (nie foldery).

Także dla synchronizacji oddzielnych struktur potrzebujemy po **jednym semaforze** na:

- Inode stat + inode table
- Block stat + block links + data blocks
- Open file stat + open file table

Poniższa tabela opisuje co dokładnie synchronizujemy w przypadku każdej funkcji api.

Oznaczenia:

+ użyta synchronizacja

* semafor może zostać stworzony i nie będzie zajęty

** semafor może zostać usunięty

Tworzenie i usuwanie tych semaforów(* i **) tylko wewnątrz sekcji krytycznej inode stat

/ folder oznacza wszystkie foldery w fs.

	/ folder read	**/ Folder write**	**Target Inode read**	**Target Inode write**	**Inode stat**	**Block stat**	**Open File stat**
open	+		*	+*			+
creat		+		+*	+	+	+
unlink		+	+**	**	+	+	
mkdir		+			+	+	
rmdir		+			+	+	
read			+				
write				+		+	
lseek							
close			**	+**			+

Aby nie nastąpiło zagłodzenie procesów chcących pisać do plików zaimplementowaliśmy synchronizację na plikach otwartych z faworyzacją pisarzy.

Jak widać z tabeli operacje **read**, **write**, **lseek** oraz **close** na różnych plikach mogą odbywać się niezależnie i równolegle (z wyjątkiem sytuacji, kiedy podczas zapisu do pliku musi zaalokować dodatkowe bloki). Pozostałe funkcje **open**, **creat**, **unlink**, **mkdir**, **rmdir** wymagają dostępu do zasobów współdzielonych, a więc na raz tylko jeden proces będzie miał do nich dostęp.

Interfejs użytkownika

Oprócz kodu źródłowego podajemy też **makefile**, który w sposób wygodny umożliwia budowanie projektu.

Podany makefile zawiera następujące polecenia:

- **all** - opcja domyślna, kompiluje bibliotekę projektu, a także kompiluje pliki użytkownika podane w folderze usr_src, i dołącza do każdego bibliotekę.
- **lib** - kompilowana jest tylko biblioteka z udostępnianymi funkcjami.
- **clean** - usunięcie wszystkich plików wygenerowanych w czasie kompilacji
- **distclean** - przywrócenie stanu początkowego projektu (dotyczy to wygenerowanych plików, pobieranych bibliotek itd.)
- **test** - najpierw kompilowana jest biblioteka, potem kompiluje pliki źródłowe, które zawierają testy. Do tego jest potrzebny framework Unity z którego korzystamy, a więc przy pierwszym uruchomieniu on zostanie pobrany z repozytorium github.
- **build_tests** - to samo co test, tylko testy nie zostają uruchomione.
- **run_tests** - uruchamia wszystkie pliki wykonywalne w folderze test/build/bin, które mają następujący wzorzec test_*.out
- **valgrind** - działa jak flaga, należy używać w połączeniu z test, lub run_tests

Także w przypadku testowania makefile tworzy plik tekstowy test_log.txt, gdzie zapisywane są wyniki każdego testowania.

Sposób działania funkcji api został opisany wcześniej. Natomiast nie opisaliśmy jak poprawnie dołączyć bibliotekę aby móc z niej skorzystać. Należy umieścić plik źródłowy, w którym wykorzystane zostały funkcje api, w folderze **usr_src**, oraz załączyć nagłówek biblioteki: #include “simplefs_api.h”.

Potem za pomocą polecenia make zostanie wygenerowany plik binarny z dołączoną biblioteką. Możliwe jest też ręczne dołączenie biblioteki bez użycia makefile.

Opis struktury projektu

W folderze projektu znajdują się foldery **simplefs**, **test** i **usr_src** oraz plik **makefile**.

- **test** - służy do przechowywania kodu testów zawartego w plikach test/src/test_*.c. Także w tym folderze umieszczane są wyniki kompilacji związane z testami. Dodatkowo przy pierwszym odpaleniu opcji test w makefile zostanie pobrany framework do testowania Unity.

- **usr_src** - w tym folderze należy umieszczać pliki źródłowe, do których dołączamy bibliotekę naszego systemu plików. W celu demonstracji umieściliśmy w nim kilka przykładowych przypadków użycia naszego systemu.

- **simplefs** - odpowiada za działanie systemu
    - memory - w tym folderze w plikach o odpowiedniej nazwie zdefiniowano struktury systemu plików oraz funkcje do nich.
    - simplefs_api.c/h - zawiera funkcje, które udostępniamy w ramach naszego api
    - simplefs_utils.c/h - zawiera funkcje, ułatwiające korzystanie ze struktur danych oraz funkcji najniższego poziomu. Poziom funkcji w simplefs_utils jest między poziomem funkcji api, a funkcji działających bezpośrednio na pamięci współdzielonej zmapowanej do procesu.
    - simplefs_synchronization.c/h - zawiera zagregowane funkcje do podnoszenia i opuszczania semaforów.

Wykorzystane narzędzia

Projekt został napisany w całości w języku C z użyciem systemowych funkcji Linuksa do użycia pamięci współdzielonej i semaforów POSIX. Do budowania użyliśmy kompilatora gcc oraz programu make. Do debugowanie wykorzystaliśmy gdb oraz valgrind. Do testowania wykorzystaliśmy framework Unity.

Testowanie

Testy znajdują się w test/src/. Staraliśmy przetestować każdą funkcję api pod względem sytuacji błędnych opisanych dla każdej funkcji, i też w sytuacjach kiedy powinno działać dobrze. Dodatkowo napisaliśmy szereg testów do funkcji niższym poziomów, które operują na kilku bądź na pojedynczych strukturach.

Do testowania używamy framework Unity. Podajemy też link do strony oficjalnej tego framework'u.

Aby uruchomić testy należy wykonać polecenie: make test. Ściągnie to (jeśli nie zrobiło tego wcześniej) framework do testowania, skompiluje bibliotekę, skompiluje testy i dołączy do nich bibliotekę, oraz uruchomi wszystkie testy.