Wprowadzenie do systemu xv6 na architekturze RISC-V

Szymon Groszkowski 193141
Kamil Śliwiński 193740

20 Grudnia 2024

Spis treści

1 Wprowadzenie

1.1 Cel skryptu

Celem skryptu jest ułatwienie rozpoczęcia pracy z systemem xv6, w szczególności jego wersją na architekturę RISC-V. xv6 to minimalistyczny system operacyjny, który jest doskonałym materiałem dydaktycznym dla osób chcących zgłębić podstawy działania systemów operacyjnych. Jego prostota oraz przejrzystość kodu źródłowego sprawiają, że idealnie nadaje się dla studentów.

Skrypt ten ma na celu oszczędzenie czasu i wysiłku osobom, które są zainteresowane zabawą i nauką xv6. Dzięki temu nie muszą one samodzielnie przeszukiwać różnych źródeł i rozwiązywać problemów związanych z konfiguracją. Wszystkie potrzebne informacje i instrukcje zostały zebrane w jednym miejscu, aby zapewnić płynne rozpoczęcie pracy z tym systemem.

1.2 Czym jest xv6?

xv6 to minimalistyczny system operacyjny stworzony przez profesorów MIT (Massachusetts Institute of Technology) jako materiał dydaktyczny dla kursu z systemów operacyjnych. Jest to współczesna reimplementacja klasycznego UNIX-a w wersji Sixth Edition (V6), który po raz pierwszy został wydany przez Bell Labs w 1975 roku. xv6 został zaprojektowany z myślą o prostocie i przejrzystości kodu, co czyni go idealnym narzędziem do nauki podstaw działania systemów operacyjnych.

1.3 Dlaczego warto poznać xv6?

xv6 jest narzędziem edukacyjnym, które zyskało uznanie dzięki swojemu minimalistycznemu podejściu. Kod źródłowy systemu jest kompaktowy (ok. 10tyś. lini kodu) i zawiera jedynie najbardziej podstawowe elementy systemu operacyjnego, co umożliwia jego dokładną analizę i zrozumienie.

Kolejną zaletą xv6 jest jego zgodność z klasycznym UNIX-em, na którym bazuje. Wiele mechanizmów i struktur zaimplementowanych w xv6 ma swoje odpowiedniki w nowoczesnych systemach operacyjnych. Dodatkowo, kod xv6 został zaprojektowany w sposób przejrzysty i dobrze udokumentowany w skrypcie (xv6 book).

xv6 jest także uniwersalnym narzędziem, które można uruchamiać zarówno na rzeczywistym sprzęcie, jak i w emulatorze, takim jak QEMU. Ta wszechstronność sprawia, że xv6 jest świetnym wyborem do eksperymentowania i nauki, umożliwiając łatwe testowanie i modyfikowanie systemu operacyjnego w bezpiecznym środowisku. xv6 jest także uważany za świetny start dla osób, które planują zagłębić się w kod kernela Linuxa.

1.4 Gdzie xv6 jest używany?

xv6 jest przede wszystkim wykorzystywany w środowisku akademickim. Stanowi podstawę wielu kursów na temat systemów operacyjnych prowadzonych na uczelniach na całym świecie, w tym w MIT, Stanford University i wielu innych prestiżowych instytucjach.

1.5 Architektura RISC-V

RISC-V to nowoczesna, otwarta i darmowa architektura procesorów oparta na zasadach RISC (Reduced Instruction Set Computer). Jest modułowa, co oznacza, że podstawowy zestaw instrukcji można rozszerzać o dodatkowe funkcjonalności w zależności od potrzeb. Znajduje zastosowanie w różnych dziedzinach np.: mikrokontrolery czy superkomputery. Dzięki rosnącej społeczności i wsparciu wielu firm, RISC-V staje się istotnym konkurentem dla zamkniętych architektur, takich jak ARM czy x86. Historię powstania oraz zastosowania architektury RISC-V świetnie przedstawiono w video: Explaining RISC-V: An x86 & ARM Alternative.

2 Konfiguracja środowiska

Aby uruchomić system `xv6-riscv`, który został zaprojektowany z myślą o architekturze RISC-V, musimy dysponować odpowiednimi narzędziami umożliwiającymi zarówno translację kodu źródłowego na instrukcje tej architektury, jak i jego emulację. Komputery, z których korzystamy na co dzień, opierają się głównie na architekturach takich jak x86-64 czy ARM, co sprawia, że bez dodatkowych narzędzi nie jesteśmy w stanie bezpośrednio uruchomić takiego systemu.

Do pracy z `xv6-riscv` potrzebny jest specjalny *toolchain* – zestaw narzędzi, który umożliwia kompilację kodu na docelową architekturę oraz jego emulację. Toolchain ten składa się m.in. z dedykowanego kompilatora dla RISC-V (`riscv-gnu-toolchain`), emulatora (np. `QEMU`) oraz dodatkowych bibliotek i narzędzi wspierających. Choć możliwe jest ręczne zainstalowanie i skonfigurowanie całego toolchaina, takie podejście bywa problematyczne i czasochłonne. Instalacja poszczególnych komponentów wymaga dokładnej znajomości zależności między nimi, a także umiejętności rozwiązywania potencjalnych błędów, które mogą wystąpić podczas instalacji.

Alternatywnym rozwiązaniem jest skorzystanie z gotowego systemu przygotowanego na platformie `Docker`. Docker to narzędzie, które pozwala na uruchamianie aplikacji w odizolowanych środowiskach – kontenerach, które zawierają wszystkie niezbędne narzędzia i biblioteki. Dzięki niemu nie musimy ręcznie instalować żadnych dodatkowych komponentów na naszym komputerze – wszystko, czego potrzebujemy, znajduje się w gotowym obrazie Dockera. Takie podejście eliminuje ryzyko błędów konfiguracyjnych, przyspiesza proces przygotowania środowiska i pozwala od razu skupić się na nauce systemu `xv6`.

W tym skrypcie skupimy się na drugim rozwiązaniu i w kolejnych krokach pokażemy, jak skonfigurować środowisko do uruchamiania `xv6-riscv` przy użyciu Dockera.

2.1 Konfiguracja środowiska Docker

Instrukcje instalacji Dockera są szeroko dostępne w Internecie i różnią się w zależności od systemu operacyjnego, dlatego w tej sekcji pominiemy dokładny opis instalacji. Można je znaleźć na oficjalnej stronie Dockera (`https://docs.docker.com/get-docker/`) lub w wielu innych zasobach online.

Załóżymy, że Docker został już poprawnie zainstalowany na komputerze i skupimy się tylko na konfiguracji obrazu Dockera oraz przygotowaniu kontenera dla `xv6-riscv`. Kontener

ten zawiera wszystkie niezbędne narzędzia, takie jak kompilator `riscv-gnu-toolchain` oraz emulator `QEMU`, co pozwoli na szybkie rozpoczęcie pracy z systemem `xv6-riscv`. W kolejnych krokach pokażemy, jak pobrać i skonfigurować taki kontener.

Krok 1: Pobranie obrazu Docker z Docker Hub

Najpierw należy pobrać gotowy obraz zawierający narzędzia do pracy z architekturą RISC-V. Wykorzystamy do tego obraz `szymongrrr/xv6-riscv-environment:latest`, dostępny w repozytorium Docker Hub, bazującym na pliku Dockerfile udostępnionym na platformie GitHub (xv6-riscv-docker).

```
sudo docker pull szymongrrr/xv6-riscv-environment:latest
```

Krok 2: Weryfikacja obrazu

Aby upewnić się, że obraz został poprawnie pobrany i działa, należy uruchomić tymczasowy kontener:

```
sudo docker run -it szymongrrr/xv6-riscv-environment:latest
```

Jeśli kontener uruchomił się poprawnie, w terminalu powinniśmy zobaczyć poniższą linię, wskazującą, że znajdujemy się w środowisku kontenera:

`root@<containerID>:/home/os-iitm#`

Po zakończeniu testu możemy zakończyć kontener, wciskając Ctrl+D oraz usunąć go, aby utrzymać porządek w systemie. W tym celu należy:

1. Wyświetlić listę wszystkich kontenerów (również tych zakończonych):

```
sudo docker ps -a
```

2. Znaleźć identyfikator kontenera (container ID) z listy, a następnie usunąć go za pomocą polecenia:

```
sudo docker rm <containerID>
```

podstawiając `<containerID>` identyfikatorem naszego kontenera.

2.2 Pierwsze uruchomienie xv6-riscv w kontenerze

Aby uruchomić xv6-riscv, należy wykonać poniższe kroki:

Klonowanie repozytorium xv6-riscv

Najpierw należy sklonować oficjalne repozytorium xv6-riscv do systemu hosta:

```
git clone https://github.com/mit-pdos/xv6-riscv
```

Uruchomienie xv6-riscv w kontenerze Docker

Aby uruchomić xv6-riscv w kontenerze, wykonaj poniższe kroki:

1. Uruchom kontener Docker, udostępniając sklonowany katalog `xv6-riscv` z systemu hosta do kontenera:

```
docker run -it -v <ścieżka do folderu z xv6-riscv w systemie
↪  hosta>:/home/os-iitm/xv6-riscv szymongrrr/xv6-riscv-environment:latest
```

   Zastąp `<ścieżka do folderu z xv6-riscv w systemie hosta>` ścieżką do katalogu xv6-riscv uzyskaną z polecenia `pwd`.

   Flaga `-v` umożliwia współdzielenie folderów między systemem hosta a kontenerem Docker. Repozytorium `xv6-riscv` będzie dzięki temu dostępne w kontenerze.

2. Po uruchomieniu kontenera znajdziesz się w środowisku kontenera (np. `root@<containerID>:/home/os-iitm#`).

3. W środowisku kontenera przejdź do katalogu `/xv6-riscv` i uruchom xv6-riscv za pomocą poleceń:

```
cd xv6-riscv
make qemu
```

Jeśli wszystkie operacje zostały wykonane poprawnie, w terminalu powinien pojawić się komunikat:

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$
```

W tym momencie system xv6 jest uruchomiony i gotowy do działania.

Aby zakończyć działanie systemu xv6 i emulatora QEMU, należy użyć kombinacji klawiszy: Ctrl+A, następnie X. Po zakończeniu pracy możesz opuścić środowisko kontenera, używając kombinacji klawiszy `Ctrl+D`, co spowoduje wylogowanie się i zatrzymanie kontenera.

2.3 Ponowne uruchomienie kontenera

Aby ponownie uruchomić wcześniej utworzony kontener i kontynuować pracę w jego środowisku, wykonaj poniższe kroki:

1. **Wyświetlenie listy kontenerów**
   Najpierw sprawdź dostępne kontenery (w tym te zatrzymane):

```
sudo docker ps -a
```

Znajdź identyfikator (`container ID`) kontenera, który chcesz uruchomić.

2. **Uruchomienie kontenera w trybie interaktywnym**
   Aby wznowić pracę w zatrzymanym kontenerze, użyj poniższego polecenia:

   ```
   sudo docker start -i <containerID>
   ```

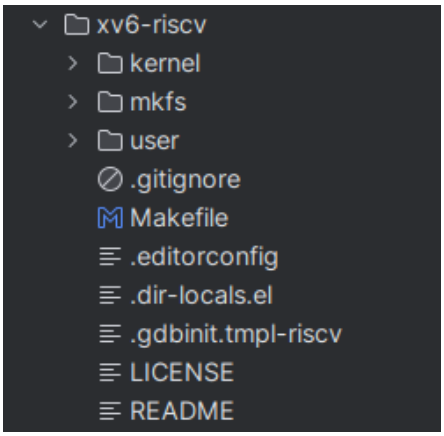
   Zastąp `<containerID>` identyfikatorem wybranego kontenera.

3. **Powrót do środowiska kontenera**
   Po wykonaniu powyższego polecenia znajdziesz się ponownie w środowisku kontenera (np. `root@<containerID>:/home/os-iitm#`), z dostępem do wszystkich wcześniej wykonanych operacji i zmian.

Dzięki tym krokom możesz wznawiać pracę w wcześniej skonfigurowanym środowisku Docker bez konieczności ponownego tworzenia lub uruchamiania nowego kontenera.

3 Wprowadzenie do struktury xv6

Struktura xv6 opiera się na podziale na dwie przestrzenie: przestrzeń jądra (*kernel space*) oraz przestrzeń użytkownika (*user space*), z których każda ma swoje unikalne funkcje i odpowiedzialności.



Struktura katalogów systemu xv6-riscv

3.1 Przestrzeń jądra

Przestrzeń jądra (*kernel space*) jest sercem systemu xv6. To tutaj implementowane są wszystkie kluczowe mechanizmy systemu operacyjnego, które umożliwiają zarządzanie zasobami sprzętowymi i procesami. Kod źródłowy przestrzeni jądra znajduje się w katalogu `kernel/`. Przykładowe pliki i ich funkcje to:

- `proc.c`: zarządzanie procesami, w tym implementacja funkcji schedulera.
- `syscall.c`: obsługa wywołań systemowych.
- `trap.c`: obsługa przerwań.

Przykładowy kod z `proc.c`, przedstawiający uproszczoną implementację schedulera:

```
void scheduler(void) {
  struct proc *p;
  struct cpu *c = mycpu();
  c->proc = 0;
  for (;;) {
    intr_on();
    for (p = proc; p < &proc[NPROC]; p++) {
      acquire(&p->lock);
      if (p->state == RUNNABLE) {
        p->state = RUNNING;
        c->proc = p;
        swtch(&c->context, &p->context);
        c->proc = 0;
      }
      release(&p->lock);
    }
  }
}
```

Funkcja `scheduler` iteruje przez tablicę procesów i przydziela czas procesora procesom w stanie `RUNNABLE`.

3.2 Przestrzeń użytkownika

Przestrzeń użytkownika (*user space*) to warstwa, w której działają programy użytkownika korzystające z funkcji udostępnianych przez jądro. Programy te realizują różne zadania, takie jak przeglądanie katalogów, edytowanie plików, czy komunikacja między procesami.

Kod źródłowy programów użytkownika znajduje się w katalogu `user/`. Przykładowe programy to:

- `ls.c`: wyświetlanie zawartości katalogu.
- `cat.c`: wyświetlanie zawartości plików.
- `sh.c`: implementacja powłoki.

Przykładowy kod z programu `ls.c`:

```
int main(int argc, char *argv[]) {
  int fd;
  struct dirent de;
  if ((fd = open(".", O_RDONLY)) < 0) {
    printf("ls: cannot open .\n");
    exit(1);
```

```
  }
  while (read(fd, &de, sizeof(de)) == sizeof(de)) {
    if (de.inum == 0)
      continue;
    printf("%s\n", de.name);
  }
  close(fd);
  exit(0);
}
```

Kod odczytuje zawartość katalogu `.` i wyświetla nazwy plików.

3.3 Powłoka

Powłoka `sh.c` to jeden z kluczowych programów użytkownika w `xv6`. Jest prostym interpreterem poleceń, który umożliwia użytkownikowi interakcję z systemem. Powłoka oczekuje na polecenia użytkownika, które następnie analizuje, interpretuje i wykonuje. Może uruchamiać programy użytkownika, zarządzać procesami oraz przekazywać dane między nimi za pomocą potoków.

Główne elementy powłoki:

- `Pętla główna`: Powłoka działa w nieskończonej pętli, w której wyświetla prompt, odczytuje polecenia od użytkownika i je przetwarza
- `Analiza poleceń`: W funkcji parsecmd() powłoka dzieli wprowadzone przez użytkownika polecenia na argumenty i identyfikuje operatory, takie jak przekierowania czy potoki.
- `Tworzenie procesów`: W funkcji runcmd() powłoka wykorzystuje `fork`, aby tworzyć nowe procesy dla uruchamianych programów. Proces potomny wykonuje polecenie za pomocą `exec`.

Przykładowy kod demonstruje pętlę główną powłoki:

```
int main(void) {
  static char buf[100];
  while (getcmd(buf, sizeof(buf)) >= 0) {
    if (fork() == 0)
      runcmd(parsecmd(buf));
    wait(0);
  }
  exit(0);
}
```

W powyższym kodzie:

- `getcmd`: odczytuje polecenie użytkownika.
- `parsecmd`: analizuje wprowadzone polecenie.
- `runcmd`: wykonuje polecenie w procesie potomnym.

4 Modyfikacja systemu

4.1 Dodanie nowego wywołania systemowego

Dodanie nowego wywołania systemowego do jądra xv6 wymaga kilku kroków. Oto jak możesz dodać wywołanie systemowe o nazwie kmemfree(), które zwraca ilość wolnej pamięci jądra. Instrukcję dodania nowego wywołania systemowego oprzemy na materiale: Getting Started with xv6 - User Progs and System Calls.

1. **Zdefiniuj wywołanie systemowe**
   Najpierw musisz zdefiniować wywołanie systemowe. Otwórz `kernel/syscall.h` plik i dodaj nowy numer wywołania systemowego na końcu listy. Na przykład:

   ```
   #define SYS_kmemfree 22
   ```

2. **Implementacja wywołania systemowego**
   Następnie musisz zaimplementować wywołanie systemowe. Utwórz nową funkcję w `kernel/sysproc.c`:

   ```
   uint64
   sys_kmemfree(void)
   {
     return (uint64) kmemfree();
   }
   ```

   Teraz musisz zaimplementować tę funkcję w `kernel/kalloc.c`:

   ```
   uint64
   kmemfree(void)
   {
     struct run *r;
     uint64 free = 0;
     for(r = kmem.freelist; r; r=r->next)
       free += PGSIZE;
     return free;
   }
   ```

3. **Dodaj prototyp do `kernel/defs.h`**

   ```
   // kalloc.c
   void*           kalloc(void);
   void            kfree(void *);
   void            kinit(void);
   uint64          kmemfree(void);
   ```

4. **Dodaj wywołanie systemowe do tabeli wywołań systemowych**
   Następnie musisz dodać wywołanie systemowe do tabeli wywołań systemowych. Otwórz `kernel/syscall.c` i dodaj:

```
extern uint64 sys_kmemfree(void);
```

Następnie dodaj kolejne wejście w tablicy wywołań systemowych:

```
[SYS_kmemfree] sys_kmemfree,
```

5. **Dodaj `kmemfree()` do `user/usys.pl`**

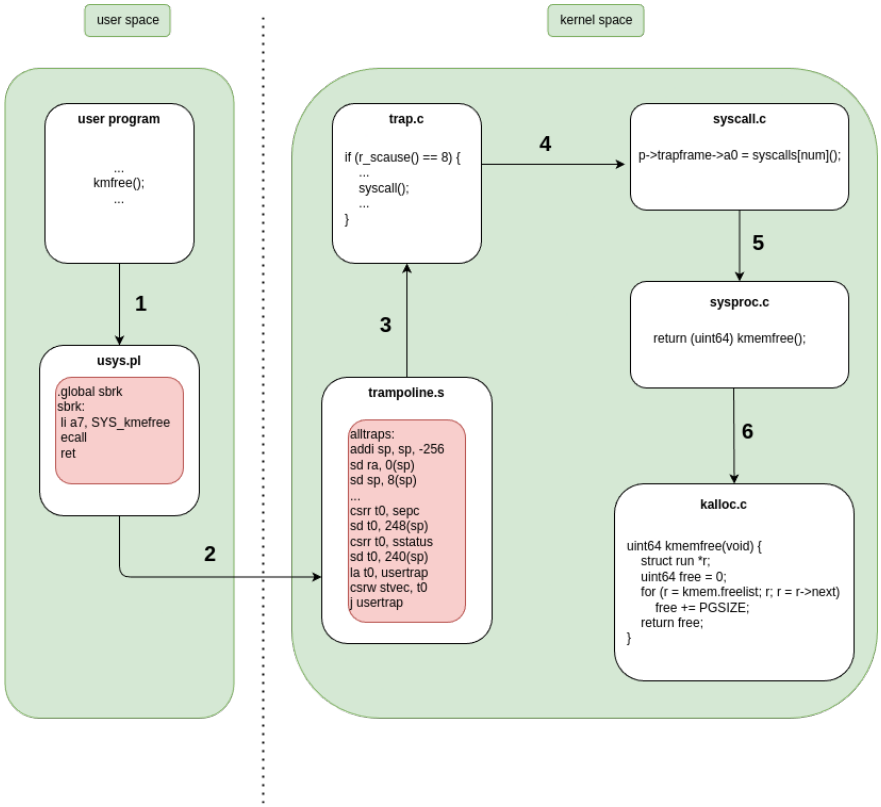
```
entry("kmemfree");
```

6. **Dodaj `kmemfree()` do `user/usys.h`**

```
int kmemfree(void);
```

4.2 Workflow wywołania systemowego na przykładzie `kmemfree()`

Po dodaniu wywołania systemowego prześledzmy jego działanie. Poniżej znajduje się diagram pokazujący, jak żądanie z poziomu programu użytkownika (user space) przechodzi przez różne warstwy systemu, zanim dotrze do funkcji jądra, a następnie wraca z wynikiem.



Przepływ wywołania systemowego na przykładzie `kmemfree()`

1. **Program użytkownika:** Wywołanie rozpoczyna się w programie użytkownika, który wykonuje funkcję `kmemfree()`. Program ten działa w przestrzeni użytkownika i wysyła żądanie do jądra systemu operacyjnego.

```
uint64 free_memory = kmemfree();
```

2. **Wrapper w `usys.pl`:** Wywołanie systemowe jest mapowane na odpowiednią funkcję w pliku `usys.pl`. Funkcja ta ustawia numer wywołania systemowego w rejestrze `a7` i wykonuje instrukcję `ecall`, która generuje przerwanie i przekazuje kontrolę do jądra.

```
.global kmemfree
kmemfree:
    li a7, SYS_kmemfree
    ecall
    ret
```

3. **Trampoline w `trampoline.s`:**

Plik `trampoline.s` pełni kluczową rolę w obsłudze przejść między przestrzenią użytkownika a przestrzenią jądra w xv6. Kod w tym pliku jest mapowany do tej samej przestrzeni wirtualnej zarówno w przestrzeni użytkownika, jak i w przestrzeni jądra, co pozwala na sprawne przełączanie kontekstu procesów.

Funkcja `alltraps` w `trampoline.s` odpowiada za zapisanie aktualnego stanu rejestrów procesora na stosie procesu, a następnie przekierowuje kontrolę do funkcji `usertrap()` w jądrze. Jest to pierwszy etap obsługi wywołania systemowego lub przerwania z przestrzeni użytkownika.

Każdy proces w xv6 posiada swoją własną instancję tego kodu w przestrzeni jądra, co umożliwia zachowanie bezpieczeństwa i izolacji między procesami. Poniżej przedstawiono kluczowy fragment kodu z `trampoline.s`:

```
alltraps:
    // Zapisz rejestry na stosie procesu
    addi sp, sp, -256
    sd ra, 0(sp)
    sd sp, 8(sp)
    ...
    csrr t0, sepc  // Pobierz licznik programu użytkownika
    sd t0, 248(sp)
    csrr t0, sstatus
    sd t0, 240(sp)

    // Przekieruj kontrolę do handlera w jądrze
    la t0, usertrap
    csrw stvec, t0
    j usertrap
```

4. **Handler w `trap.c`:** Funkcja `usertrap()` w `kernel/trap.c` obsługuje przerwania i wyjątki generowane przez procesy użytkownika. W przypadku wywołania systemowego (określonego przez numer r_scause() == 8), przekierowuje kontrolę do funkcji `syscall()`.

```
// Zapisz licznik programu użytkownika
p->trapframe->epc = r_sepc();

if (r_scause() == 8) {  // Wywołanie systemowe
    if (killed(p))
        exit(-1);

    // Przesuń licznik programu, aby pominąć instrukcję ecall
    p->trapframe->epc += 4;

    // Włącz przerwania po obsłudze rejestrów
    intr_on();

    // Obsłuż wywołanie systemowe
    syscall();
}
   }
```

5. **Dispatcher w `syscall.c`:** Plik syscall.c działa jako dyspozytor wywołań systemowych. To tam zdefiniowana jest tablica `syscalls[]`, która kieruje numery syscalli do odpowiednich funkcji. Implementacje tych funkcji są jednak umieszczane w plikach odpowiadających ich obszarowi odpowiedzialności, takich jak sysproc.c (procesy), czy sysfile.c (operacje na plikach). Funkcja `syscall()` odczytuje numer wywołania systemowego z rejestru i wywołuje odpowiednią funkcję w globalnej tablicy wywołań systemowych (`syscalls[]`).

```
   extern uint64 sys_kmemfree(void);
   static uint64 (*syscalls[])(void) = {
       ...
       [SYS_kmemfree] sys_kmemfree,
   };
```

```
void
syscall(void)
{
  int num; // Numer wywołania systemowego
  struct proc *p = myproc(); // Pobierz aktualnie działający proces

  // Odczytaj numer wywołania systemowego z rejestru a7
  num = p->trapframe->a7;

  // Sprawdź, czy numer wywołania jest poprawny (mieści się w zakresie tablicy)
  if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
    // Wywołaj odpowiednią funkcję wywołania systemowego z tablicy syscalls[]
    // Wynik funkcji zapisz w rejestrze a0, aby mógł być dostępny dla programu
    ↪  użytkownika
    p->trapframe->a0 = syscalls[num]();
  } else {
    // Jeśli numer wywołania jest niepoprawny, wypisz ostrzeżenie i zwróć -1
    printf("%d %s: unknown sys call %d\n", p->pid, p->name, num);
```

```
    p->trapframe->a0 = -1;
  }
}
```

6. **Funkcja w `sysproc.c`:** Funkcja `sys_kmemfree()` realizuje logikę wywołania systemowego, wywołując funkcję `kmemfree()` zaimplementowaną w `kalloc.c`, która oblicza ilość dostępnej pamięci.

```
uint64 sys_kmemfree(void) {
    return (uint64) kmemfree();
}
```

7. **Rzeczywista realizacja funkcji kmfree() w `kalloc.c`:** Funkcja `kmemfree()` przeszukuje listę wolnych stron pamięci (`freelist`) i oblicza ich łączny rozmiar.

```
uint64 kmemfree(void) {
    struct run *r;
    uint64 free = 0;
    for (r = kmem.freelist; r; r = r->next)
        free += PGSIZE;
    return free;
}
```

8. **Powrót do programu użytkownika:**

Po zakończeniu wywołania systemowego system operacyjny musi zwrócić wynik tej operacji do programu użytkownika i umożliwić mu kontynuację działania.

Pierwszym krokiem jest zapisanie wyniku wywołania systemowego w rejestrze `a0`. Rejestr ten pełni funkcję standardowego miejsca przechowywania wyników wywołań systemowych w architekturze RISC-V. W przypadku wywołania systemowego `kmemfree()`, rejestr `a0` będzie zawierał liczbę bajtów dostępnych w wolnej pamięci jądra.

Następnie system musi przywrócić pełen kontekst procesu użytkownika, który wywołał syscall `kmfree()`, aby mógł on kontynuować swoje działanie dokładnie od miejsca, w którym przerwał. Kontekst obejmuje m.in. stan rejestrów , licznik programu (*program counter*), wskaźnik stosu oraz inne dane dotyczące procesu. Kluczowym elementem jest tutaj ustawienie licznika programu (`PC`) na adres następnej instrukcji do wykonania, co w xv6 oznacza instrukcję znajdującą się tuż za `ecall`.

Kolejnym krokiem jest przełączenie systemu z trybu jądra (*kernel mode*) do trybu użytkownika (*user mode*). W tym momencie proces użytkownika zostaje odizolowany od bezpośredniego dostępu do zasobów systemowych, co zapewnia bezpieczeństwo działania zarówno samego programu, jak i całego systemu operacyjnego. Tryb użytkownika umożliwia dalsze wykonywanie programu, ale jednocześnie ogranicza jego zdolność do ingerowania w działanie jądra.

Proces ten kończy się wznowieniem działania programu użytkownika. Licznik programu (`PC`) wskazuje na instrukcję, która miała być wykonana po zakończeniu wywołania systemowego, a program może odczytać wynik operacji z rejestru `a0`. Przykładowo, jeśli

użytkownik wywołał funkcję `kmemfree()`, to wynik (liczba bajtów wolnej pamięci) będzie dostępny bezpośrednio w tym rejestrze.

W xv6 proces ten jest realizowany przez funkcję `usertrapret()`, która odpowiada za przywrócenie kontekstu programu użytkownika, przełączenie trybu z jądra na użytkownika oraz ustawienie odpowiednich wartości rejestrów. Oto fragment kodu odpowiedzialny za przesunięcie licznika programu (`PC`) oraz przywrócenie działania procesu w pliku `/kernel/trap.c` w funkcji `usertrap()`.

```
p->trapframe->epc += 4; // Przeskok za instrukcję ecall
usertrapret(); // Przywrócenie kontekstu użytkownika
```

Dzięki tym mechanizmom xv6 zapewnia bezpieczne i płynne przechodzenie między przestrzenią użytkownika a przestrzenią jądra, co realizuje podstawowe założenia nowoczesnych systemów operacyjnych opartych na architekturze UNIX.

4.3 Dodanie nowego programu użytkownika

Możemy teraz utworzyć nowy program użytkownika, który wywołuje dodane wywołanie systemowe.

1. **Utwórz nowy plik `user/kmemfree.c`:**

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

int
main(int argc, char *argv[])
{
  printf("Free kernel memory: %d\n", kmemfree());
  exit(0);
}
```

2. **Dodaj program użytkownika do pliku Makefile**
   Otwórz `Makefile` i dodaj *freekmem* do listy `UPROGS`:

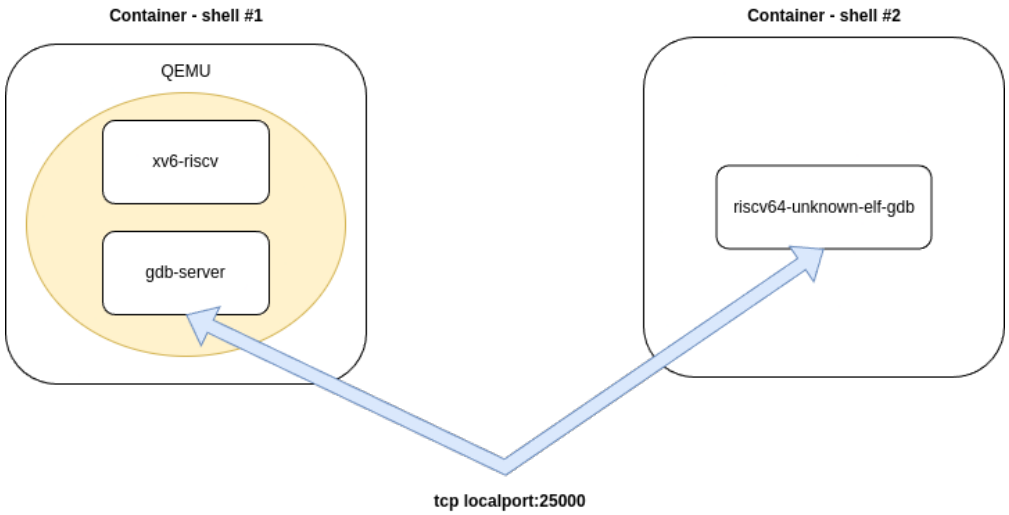
```
UPROGS=\
    _cat\
    ...
    _kmemfree\
    ...
```

5 Debugowanie xv6

Debugowanie xv6 wymaga użycia debuggera GDB, który łączy się z emulatorem QEMU przez zdalne połączenie na localhost. GDB zapewnia możliwość analizowania rejestrów procesora, pamięci oraz przepływu programu, co jest kluczowe dla zrozumienia działania systemu. Dzięki

temu można zatrzymywać wykonanie programu, przechodzić krok po kroku przez kod oraz rozwiązywać ewentualne problemy.

QEMU oferuje serwer debugowania, który nasłuchuje na porcie lokalnym, np. `localhost:25000`. Klient GDB uruchomiony w innej powłoce tego samego kontenera łączy się z tym serwerem, umożliwiając wgląd w stan symulowanego środowiska xv6 oraz wysyłanie poleceń debuggowania. Taka konfiguracja pozwala na zachowanie izolacji między emulatorem a debuggerem, co upraszcza proces debugowania.



Schemat debuggowania xv6 przy użyciu zdalnego połączenia GDB

W kolejncyh krokach pokażemy, jak należy uruchomić emulator QEMU z xv6 oraz serwerem GDB nasłuchującym na porcie localhost.

5.1 Konfiguracja GDB

1. Uruchomienie QEMU z serwerem GDB

Aby ułatwić debugowanie, należy dodać flagę `-O0`, która wyłącza optymalizacje kompilatora, umożliwiając dokładniejsze odwzorowanie kodu źródłowego w debugerze.

W pliku `Makefile` można zastosować warunek, który automatycznie doda flagę `-O0` tylko w przypadku, gdy celem kompilacji jest `qemu-gdb`. Przykład konfiguracji:

```
# Wyłączenie optymalizacji kompilatora dla trybu qemu-gdb
ifeq ($(MAKECMDGOALS),qemu-gdb)
CFLAGS += -O0
endif
```

Następnie, w kontenerze uruchom emulator QEMU w trybie debugowania:

```
make qemu-gdb
```

Emulator zatrzyma wykonywanie xv6 na pierwszej instrukcji, oczekując na połączenie klienta GDB.

W terminalu pojawi się podobny komunikat:

```
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3
↪  -nographic -global virtio-mmio.force-legacy=false -drive
↪  file=fs.img,if=none,format=raw,id=x0 -device
↪  virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0 -S -gdb tcp::25000
```

Linia `-gdb tcp::25000` wskazuje, że serwer GDB nasłuchuje na porcie TCP 25000, co pozwala na połączenie klienta GDB z instancją QEMU poprzez GDB-server.

2. Uruchomienie klienta GDB w nowym shellu

Na systemie hosta otwórz nową powłokę i sprawdź działające kontenery:

```
sudo docker ps
```

Znajdź identyfikator kontenera (container ID) z listy, następnie uruchom w nim kolejną powłokę, a w niej klienta GDB:

```
1 sudo docker exec -it <container_id> bash
2 riscv64-unknown-elf-gdb
```

3. Połączenie GDB z QEMU

W kliencie GDB połącz się z serwerem debugowania na port podany w kroku nr 1. W naszym przypadku jest to port TCP 25000.

```
target remote localhost:25000
```

4. Debugowanie

Po nawiązaniu połączenia z serwerem GDB można rozpocząć proces debugowania kodu systemu. W zależności od tego, czy chcemy debugować kod jądra (`kernel/`), czy programy użytkownika (`user/`), konieczne jest załadowanie odpowiedniego pliku binarnego.

Kod jądra (`kernel/`) jest zawarty w jednym pliku binarnym `kernel/kernel` i zawiera cały skompilowany kod w przestrzeni jądra, natomiast programy użytkownika (`user/`) mają swoje oddzielne pliki binarne, odpowiadające poszczególnym programom.

Debugowanie kodu jądra (`kernel space`)

Aby debugować pliki w katalogu `kernel/`, należy:

1. W GDB załadować plik binarny jądra:

```
file xv6-riscv/kernel/kernel
```

2. Ustawić punkt przerwania w wybranej funkcji, np. `syscall()` w pliku `kernel/syscall.c`:

```
b syscall
```

3. Kontynuować wykonanie programu:

```
c
```

GDB zatrzyma wykonanie programu za każdym razem, gdy program wejdzie do funkcji `syscall`.

Debugowanie kodu użytkownika (`user space`)

Jeśli chcesz debugować pliki w katalogu `user/`, należy:

1. W GDB załadować odpowiedni plik binarny programu użytkownika. Na przykład, aby debugować program `ls` w pliku `user/ls.c`:

```
file xv6-riscv/user/_ls
```

Plik `user/_ls` odpowiada programowi `ls`.

2. Ustawić punkt przerwania w funkcji, np. `ls`:

```
b ls
```

3. Kontynuować wykonanie programu:

```
c
```

W ten sposób GDB zatrzyma wykonanie programu przy każdym wejściu do funkcji `ls`, pozwalając na analizę działania programu użytkownika.

Podsumowując, w przypadku debugowania xv6 należy zawsze wybrać odpowiedni plik binarny:

- Jeśli debugujesz kod jądra, załaduj `kernel/kernel`.
- Jeśli debugujesz program użytkownika, załaduj plik binarny odpowiadający temu programowi (np. `user/_ls` dla programu `ls`).

Wybór właściwego pliku binarnego i funkcji do debugowania jest kluczowy, aby GDB mógł prawidłowo interpretować kod i zatrzymywać wykonywanie programu w odpowiednich miejscach.

6. Zakończenie debugowania

Aby zakończyć debugowanie:

- (shell #2) Wyjdź z GDB, wpisując `quit`.
- (shell #1) Zatrzymaj QEMU za pomocą kombinacji klawiszy `Ctrl+A`, a następnie `x`.

5.2 Najbardziej użyteczne komendy GDB

1. **Uruchamianie programu i sterowanie jego wykonaniem:**
   - `run (r)` - Uruchamia program od początku.
   - `continue (c)` - Kontynuuje wykonanie programu do momentu napotkania breakpointu.
   - `step (s)` - Wchodzi do kolejnej instrukcji, włączając funkcje.
   - `next (n)` - Przechodzi do następnej instrukcji, ale omija szczegóły funkcji.
   - `finish` - Kończy aktualnie wykonywaną funkcję i wraca do miejsca, w którym została wywołana.
   - `quit (q)` - Zamyka GDB.
2. **Praca z punktami przerwania (breakpoints):**
   - `break (b) <location>` - Ustawia punkt przerwania w określonym miejscu, np. w funkcji lub linii kodu (np. `b main`, `b kernel.c:42`).
   - `info breakpoints` - Wyświetla listę wszystkich ustawionych punktów przerwania.
   - `delete <breakpoint-number>` - Usuwa punkt przerwania o określonym numerze.
   - `disable <breakpoint-number>` - Tymczasowo wyłącza punkt przerwania.
   - `enable <breakpoint-number>` - Ponownie włącza wyłączony punkt przerwania.
3. **Inspekcja zmiennych:**
   - `print (p) <variable>` - Wyświetla wartość zmiennej (np. `p x`).
   - `print <expression>` - Wyświetla wynik wyrażenia (np. `p x + y`).
   - `info locals` - Wyświetla wszystkie zmienne lokalne w aktualnym zakresie funkcji.
   - `set <variable> = <value>` - Zmienia wartość zmiennej w czasie debugowania (np. `set x = 5`).
4. **Praca z pamięcią:**
   - `x/<format> <address>` - Wyświetla zawartość pamięci pod wskazanym adresem. Format określa sposób wyświetlania danych, np. `x/4xw 0x1234` (4 słowa w formacie szesnastkowym).

- `info registers` - Wyświetla wartości wszystkich rejestrów procesora.
- `info address <symbol>` - Wyświetla adres w pamięci dla wskazanego symbolu (np. zmiennej lub funkcji).

5. **Śledzenie źródła kodu:**
   - `list (l)` - Wyświetla fragment kodu źródłowego w miejscu zatrzymania programu.
   - `list <function>` - Wyświetla kod źródłowy dla określonej funkcji.
   - `info line <line-number>` - Wyświetla informacje o linii kodu, np. adres w pamięci.
6. **Dodatkowe komendy:**
   - `help <command>` - Wyświetla szczegółowe informacje o danej komendzie.
   - `info threads` - Wyświetla listę aktywnych wątków w programie.
   - `thread <thread-number>` - Przełącza kontekst debugowania na wybrany wątek.

6 Materiały edukacyjne z MIT i inne zasoby

Najlepszym sposobem na naukę budowy i działania xv6 jest korzystanie z materiałów edukacyjnych przygotowanych przez Massachusetts Institute of Technology (MIT). Te zasoby, stworzone przez autorów xv6, zostały przystosowane do potrzeb studentów, co czyni je doskonałym punktem wyjścia. Oprócz materiałów MIT, w sieci dostępne są wysokiej jakości zasoby przygotowane przez profesorów i pasjonatów systemów operacyjnych, które uzupełniają wiedzę oraz pozwalają lepiej zrozumieć xv6 i architekturę RISC-V.

1. **Materiały z MIT**

   - **Oficjalne repozytorium xv6:**
     github.com/mit-pdos/xv6-riscv
   - **Dokumentacja xv6:**
     xv6/book-riscv.pdf
     Szczegółowa książka opisująca system xv6, podzielona na rozdziały. Czytanie wymaga pewnego zaangażowania, ale dostarcza cennych informacji, które można wykorzystać przy nauce innych systemów operacyjnych.
   - **Laboratoria MIT:**
     pdos.csail.mit.edu/6.828/2024/
     W zakładce *Labs* znajdują się zadania dotyczące różnych aspektów xv6. Każde zadanie jest oznaczone poziomem trudności i zawiera wskazówki dotyczące rozdziałów dokumentacji, które należy przeczytać. Laboratoria pozwalają na praktyczne zrozumienie systemu i wymagają dogłębnego zapoznania się z tematem.

2. **Inne źródła**

- **Teoretyczne wyjaśnienia mechanizmów xv6:**
  Playlista filmów skupiających się na wyjaśnieniu, jak działają konkretne mechanizmy w xv6:
  `xv6 Kernel-1: Intro and Overview`
- **Analiza kodu źródłowego xv6:**
  Playlista filmów omawiających xv6 na poziomie kodu źródłowego:
  `How does an OS boot? // Source Dive // 001`

Naszym zdaniem najlepszym sposobem na naukę xv6 jest połączenie kilku podejść: wykonywanie laboratoriów, czytanie książki *xv6* oraz oglądanie filmów edukacyjnych. Dzięki takiemu podejściu mamy pewność, że oprócz wiedzy teoretycznej zdobędziemy również praktyczne umiejętności. Laboratoria pozwalają na zastosowanie teorii w praktyce, co pomaga lepiej zrozumieć działanie systemu. Książka *xv6* dostarcza solidnych podstaw teoretycznych, wyjaśniając kluczowe koncepcje i szczegóły implementacji xv6 wraz z opsiem mechanizmów wykorzystywanych w nowoczesnych systemach operacyjnych. Filmy edukacyjne natomiast upraszczają trudniejsze zagadnienia i pokazują je w przystępny sposób. Takie połączenie pozwala na pełniejsze zrozumienie xv6 oraz szeroko pojętej tematyki systemów operacyjnych i architektury RISC-V.