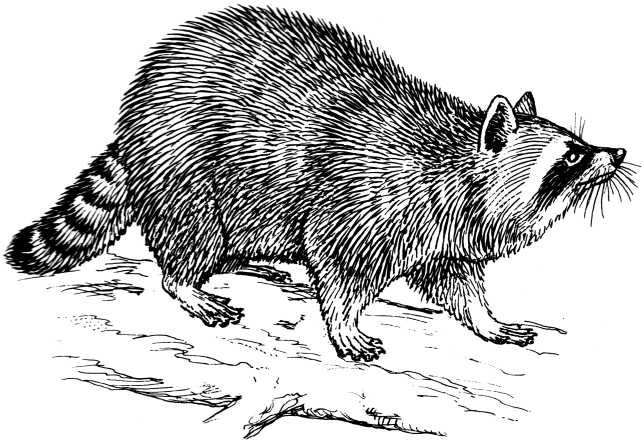

Systemy Operacyjne

Wybrane Zagadnienia

„The fact is, C++ chose bad identifiers to make reserved words."

Popełnione przez

Załatany Ponton
Dziurawy Ponton

Kraków
Anno Domini 2023

Spis treści

Licencja

Rozdział 1

Projekt – Minix

1.1 Wstęp

Mamy na celu stworzyć serwer na Minixie, który implementuje mechanizm barier.

Bariera to taki cosiek, na którym procesy mogą czekać w oczekiwaniu na inne procesy. Konkretniej – bariera ma ustaloną szerokość w. Jeśli na barierze czeka co najmniej w procesów to wszystkie zostają wznowione. W przeciwym razie procesy nadal czekają.

1.2 Podstawa dzialania

Sama implementacja syscalli jest generalnie prosta i bazuje na jednej, fundamentalnej i trywialnej obserwacji: proces czekający na odpowiedź na wysłaną przez siebie wiadomość za pomocą mechanizmu *sendrec* jest zablokowany do momentu otrzymania odpowiedzi. Jest to bardzo przydatne, bo to oznacza że nas serwer z barierami dostając syscalla mówiącego „ej zawieś mnie na takiej barierze" po prostu nie musi na nią odpowiadać żeby proces sobie wisiał. W momencie gdy uznamy, że czas „puścić" wszystkie procesy z danej bariery, po prostu posyłamy do każdego z nich jakąś wiadomość.

Można się zapytać, jak powinniśmy trzymać informacje o tym, które procesy są w jakiej barierze. Nasuwa się tu zwykła lista linkowana, ale po stronie systemu staramy się unikać wywołań funkcji *malloc*. Wynika to z faktu, że to generuje potencjał na deadlock, bo wymaga zadzwonienia do jakiegoś serwera i zablokowania się (a jednocześnie sami jesteśmy serwerem i szybko może stać się głupia sytuacja). Korzystamy więc z istnienia makra `NR_PROCS`, które jest ustalone na jakąś małą liczbę (presumably, 256). To oznacza, że możemy zaimplementować listę kursorowo, dowalając sobie ordynarnie tablicą na `NR_PROCS` elementów. Fajnie.

Do dokladniejszej implementacji przejdziemy w kolejnych sekcjach.

1.3 Strona użytkownika

Od strony użytkownika implementacja wygląda tak, jak to robiliśmy w przypadku poprzednich zadań.

- Dodajemy nagłówek `/usr/src/include/barriers.h`, w którym są definicje zadanych funkcji.
- Aktualizujemy `/usr/src/include/Makefile` o tenże nagłówek

- W pliku `/usr/src/lib/libc/sys-minix/barriers.c` piszemy implementację tych funkcji.

  Ponieważ nie znamy adresu naszego serwera to będziemy potrzebowali wywołać `minix_rs_lookup` z nagłówka `<minix/rs.h>`.

  Wykonujemy `_syscall`[1] i mapujemy errno z `EDEADSRCDST` na `EDEADEPT` żeby zgadzało się z opisem zadania.

- Aktualizujemy `/usr/src/lib/libc/sys-minix/Makefile.inc`
- opcjonalnie dodajemy sobie odpowiednie makra w `/usr/src/include/minix/com.h` – nie trzeba wtedy pamiętać liczb syscalli ani pól wiadomości, w których znajdują się informacje.

1.4 Serwer barier

Warto zacząć od poczytania implementacji serwera `ipc` – jest ona dość podobna do tego co my potrzebujemy zrobić. Generalnie to

1. Podpinamy się do SEF-a – `env_setargs(argc, argv); sef_startup();`
2. W pętli odbieramy wiadomość – `message m; sef_receive(ANY, m);`
3. Sprawdzamy `m.call_type` i na jego podstawie wywołujemy odpowiednią funkcję lub stwierdzamy, że komuś się coś pomieszało.
4. Implementujemy podstawowe mechanizmy
   - `do_barrier_init` po prostu inicjalizuje odpowiednią barierę i od razu odpowiada czy się udało czy nie
   - `do_barrier_destroy` podobnie po prostu deinicjalizuje barierę (o ile nikt na niej nie czeka) i od razu odpowiada
   - `do_barrier_wait` dodaje proces do kolejki bariery, ale nie odpowiada mu. Zamiast tego jeśli kolejka jest pełna to budzimy wszystkie procesy, które się w niej znajdują wysyłając do nich pustą wiadomość, której `m_type` wynosi `OK`.

Warto jeszcze może doprecyzować jak trzymać wszystkie te informacje sensownie. Ponieważ będziemy musieli potem rozmawiać z procfs-em to polecamy taką implementację:

W pliku `/usr/src/servers/barriers/structures.h` definiujemy `#BARRIERS 256` oraz dwie struktury :

Strukturę `barrier`, która trzyma:

- szerokość
- liczbę oczekujących procesów
- pierwszy element kolejki

Oraz strukturę `item`, która trzyma:

- endpoint procesu

[1] Dla fanów sportów ekstremalnych informujemy, że można po prostu użyć również gołego *sendrec*-a. Nie powiemy kto tak zrobił, ale jego imię zaczyna się na „D”, a kończy na „ominik”.

- id bariery na której on czeka
- następny proces
- poprzedni proces

Dane będziemy przechowywać w dwóch tablicach, których elementami będą powyższe struktury.

Tak jak powiedzieliśmy wcześniej – procesy trzymane są jako listy kursorowe tj. zamiast wskaźników trzymamy indeks w tablicy. Mając endpoint procesu, jego indeksem będzie slot w tabeli procesów, który możemy wyłuskać makrem `_ENDPOINT_P`.

Aby uprościć sobie smutną implementację listy wiązanej warto zaimplementować kolejkę jako listę cykliczną z wartownikiem.

1.5 Jak dogadać się z PMem

Wszystko fajnie, ale w wymaganiach jest napisane, że serwer barier w przypadku gdy ktoś kto czekał w barierze i oberwał sygnałem ma przestać na niej czekać (analogicznie do tego, jak oberwanie sygnałem przerywa reada). Jak to realizujemy? Istnieje sobie serwer *PM*, który zajmuje się właśnie sygnałami i takimi pięknymi rzeczami. W związku z powyższym, wystarczyłoby się tutaj „wstrzelić" gdzieś PMowi do odpowiedniej funkcji i kazać mu informować nasz serwer, że jakiś proces oberwał właśnie sygnałem. Pojawia się tu spoiler: odpowiednie miejsce do takich akcji znajduje się w pliku `signal.c` (kto by pomyślał?), a funkcja która się tym zajmuje to *unpause*. Można sobie spojrzeć na jej implementację – jest tam już mechanizm dzwonienia do *VFS*a w bardzo analogicznej sprawie.

Możemy wręcz skopiować funkcję `tell_vfs`, która wysyła wiadomość, tyle że do naszego serwera. Odbierając wiadomość potrzebujemy sprawdzić czy jest ona od PMa i wywalić przerwany/zabity proces oraz wysłać mu wiadomość z errno `EINTR`.

Jeśli proces spadł z rowerka również chcemy poinformować o tym barriersa – funkcja która się wywołuje, gdy proces oberwał sygnałem i umarł to *sig_proc_exit*. Można też skipnąć zbędne gadanie i pójść bezpośrednio do `proc_exit`, na jedno wyjdzie. Chyba.

Oczywiście pozostaje pytanie jak zrobić, żeby *PM* wiedział, na jaki endpoint zadzwonić jeśli chodzi o serwer barier. Są różne podejścia:

1. funkcja *minix_rs_lookup*, która dzwoni do *RS*a i prosi go o endpoint danego serwera: zły plan, prosto doprowadzić do deadlocku
2. dodanie do *barriers*a kodu, który każe mu dzwonić do *PM*a gdy *barriers* wstaje, aby podać mu swój endpoint (i żeby *PM* wiedział do kogo dzwonić). Wtedy też musimy jakoś informować *PM*a o tym, że *barriers* jest wyłączany, żeby nie próbował dzwonić do serwera który ktoś włączył, a potem wyłączył
3. wyszukiwanie *barriersa* na pałę w tablicy procesów *PM*a; wbrew pozorom nie jest to aż taki zły pomysł. While we're at it, przy iterowaniu przez tabelę procesów **zawsze sprawdzajcie flagę IN_USE**. Inaczej zaczniecie łapać dane o procesach które zostały ubite; w szczególności możecie po resecie barriersa dowiedzieć się, że *PM* źle adresuje wiadomości. Fun.

1.6 Procfs

Kiedy wywołujemy `cat /proc/barriers` to naszym oczom powinno okazać się eleganckie podsumowanie stanu aktywnych barier.

Idziemy zatem do `/usr/src/servers/procfs/root.c` i tam się bawimy:

1. Do tablicy `root_files` dopisujemy zgodnie ze wzorem odpowiednią linijkę
2. kopiujemy definicję dwóch tablic ze strukturami barier i procesów
3. W funkcji `root_barriers`
   (a) Szukamy serwera barier w tabeli `mproc`
   (b) Wywołujemy fajną funkcję `getsysinfo` dwa razy – raz na tablicę barier, drugi na tablicę procesów
   (c) Jeśli wszystko dobrze poszło to możemy wypisać te dwie struktury używając funkcji `buf_printf`

Dobra, ale `getsysinfo` samo nie zadziała (w końcu to tylko syscall, który wysyła wiadomość)[2], musimy jeszcze przekonać serwer barier, żeby na nie odpowiadał.

Wracamy więc do serwera barier i sprawdzamy przypadkiem czy nie dostaliśmy wiadomości o typie `COMMON_GETSYSINFO`. Jeśli tak to odpowiadamy na nie w dość prosty sposób. Aby wysłać jakąś tablicę `T` do gościa, który nas pyta wywołujemy po prostu:

```
sys_datacopy(SELF, (vir_bytes)T, m.m_source, (vir_bytes)m.SI_WHERE, m.SI_SIZE)
```

Sprawdzamy więc o którą tablicę jesteśmy pytani i odsyłamy mu jej zawartość. Tak jak wcześniej wpisywaliśmy makra w `com.h` tak tutaj warto w pliku `/usr/src/include/minix/sysinfo.h` dopisać sobie makra na informacje wyciągane od naszego serwera.

[2]Ponownie, możemy tu też zamiast bawić się w jakieś owijki zrobić własnego sendreca i być hepi

Rozdział 2

Rozwiązania egzaminów

2.1 Egzamin 2013/2014

2.1.1 Zadanie 1

Odpowiedź: konieczność wielokrotnego przełączania procesów spowodowana dużą liczbą wywołań systemowych,

2.1.2 Zadanie 2

Odpowiedź:

1. nie
2. nie
3. tak
4. tak

2.1.3 Zadanie 3

Odpowiedź: mniejszy koszt przełączania procesora pomiędzy wątkami niż pomiędzy procesami

2.1.4 Zadanie 4

Odpowiedź:

1. micro – mikrojądro potrzebuje serwerów
2. żaden – w mikrojądrze robi to tylko jądro, a w monolitycznym procesy mogą wykonywać jedynie kod jądra w trybie uprzywilejowanym
3. mono – nie ma serwerów, każdy sam przełącza się na bycie driverem przez chwilę
4. mono – gdy wykonuje się syscall proces jest tymczasowo przełączany w tryb uprzywilejowany aby wykonać potrzebny kod jądra

2.1.5 Zadanie 5

Odpowiedź:

1. tak
2. nie
3. nie
4. nie

2.1.6 Zadanie 6

Odpowiedź:

1. nie, co najwyżej ktoś przeczyta śmieci
2. tak – klasyczny przykład gdzie dwa procesy równolegle zawieszają się na semaforach AB oraz BA
3. nie – w minixie `notify` nie blokuje
4. tak – robiąc `sendrec` możemy stworzyć cykl

2.1.7 Zadanie 7

Odpowiedź: nie wiemy co to segmentacja

1. ?seg pag
2. ?seg pag
3. ?seg
4. ?seg pag
5. seg
6. ?seg
7. ?seg pag

2.1.8 Zadanie 8

Odpowiedź:

1. nie
2. tak
3. tak
4. tak
5.

2.1.9 Zadanie 9

Odpowiedź:

1. tak
2. tak

3. tak?
4. nie

2.2 Egzamin 2016/2017

2.2.1 Zadanie 1

Czytanie bajt po bajcie jest strasznie i koszmarnie nieefektywne (dużo syscalli się dzieje przez to, a to jest wolne). Pisanie również.

> *Of course, I'd also suggest that whoever was the genius who thought it was a good idea to read things ONE F*CKING BYTE AT A TIME with system calls for each byte should be retroactively aborted. Who the f*ck does idiotic things like that?*
>
> Linux Torvalds

2.2.2 Zadanie 2

Zauważmy tutaj bardzo sprytne posunięcie polegające na czekaniu na pierwsze dziecko. Jeśli plik jest odpowiednio duży, to ten *cat* zapcha całego pipe'a do którego miał pisać i się zablokuje. Nikt z kolei tego nie odczyta, bo jeszcze nie ma nikogo po drugiej stronie. Wykonanie całego programu się efektywnie zwiesi (bo rodzic czeka aż *cat* się skończy, a *cat* czeka aż ktoś coś przeczyta z pipe'a) i to będzie na tyle.

Natomiast gdy plik jest mały to po prostu, bardzo elegancko, pierwsze dziecko przepisze cały plik na wejście do pipe'a; drugie przepisze wszystko z tego pipe'a na stdout i program się zakończy.

2.2.3 Zadanie 3

2.3.3

2.2.4 Zadanie 4

To czy wyczerpał swój kwant czasu czy nie. Jeśli nie, to trafia na początek kolejki (ale z odpowiednio pomniejszonym kwantem czasu). Jeśli wyczerpał w całości, trafia na koniec kolejki.

2.2.5 Zadanie 5

- komunikacja z (fizycznym) dyskiem
- message-passing pomiędzy driverem a procesem

2.2.6 Zadanie 6

Możemy wykorzystać funkcję `select` albo `poll` aby czekać, aż któryś z pipe'ów będzie gotowy na wpisanie do niego paczki z danymi.

2.2.7 Zadanie 7

Dostajemy przerwanie sprzętowe od klawiatury i ktoś coś to obsługuje i hdd pojawia się w kolejce schedulera.

2.2.8 Zadanie 8

Obrywamy przerwaniem systemowym z zegara i budzi się scheduler i patrzy, czy komuś nie skończył się jego kwant czasu; jeśli tak, wywala go na koniec kolejki i w sumie to tyle.

2.2.9 Zadanie 9

Patrz: 2.3.11

2.2.10 Zadanie 10

Defragmentacja to tylko fancy nazwa na organizowanie dysku.

Jeśli dopisujemy coś do pliku w środku dysku, to nie mamy za bardzo innej opcji niż znalezienie kolejnego pustego fragmentu (który może być dość daleko) i wpisanie tam danych.

W wyniku takiego procesu po pewnym czasie działania mamy wiele porozrzucanych po całym dysku plików. Chyba nie trzeba tłumaczyć dlaczego to nie jest najlepszy scenariusz.

W każdym razie, defragmentacja polega na przepermutowaniu systemu plików w taki sposób, aby pliki ponownie tworzyły spójny obszar. Dzięki temu aby odczytać jakiś plik nie musimy aż tak skakać po dysku, czyli oszczędzamy trochę czasu.

2.3 Egzamin 2017/2018

2.3.1 Zadanie 1

Możliwe odpowiedzi:

1. `read` może nam zwrócić coś co nie ma (jeszcze sensu) bo system może to dowolnie pociąć
2. nie sprawdzamy co z sygnałami i `EINTR`
3. `exec` może się nie udać i dziecko się zapętli

Wczytujemy readem jakieś rzeczy do bufora i po (jednym!) readzie cokolwiek poleciało do bufora jest podawane jako argumenty programowi? I to wszystko robimy w pętli? Pomijając fakt, że jeśli sygnał nam przerwie reada to już też nic nie zrobimy? Co się dzieje, ratunku?

2.3.2 Zadanie 2

Po uważnym przeczytaniu tego kodu zauważamy o co tak naprawdę jest pytanie. Otóż chodzi o to, w jaki sposób programy wykonane przez `fork` oraz `exec` zachowują się względem obsługi sygnałów, którą ustawił rodzic.

Mamy tutaj trzy sposoby na obsługę sygnału:

1. ustawienie handlera
2. zablokowanie sygnału za pomocą `sigprocmask`
3. ustawienie sygnału `SIGINT` jako ignorowanego

Śmieszna rzecz polega na tym, że `SIGUSR1` jest tutaj „blokowane” poprzez ustawienie jakiegoś handlera (czyli tak naprawdę nie jest to blokada sygnału); to z kolei powoduje, że jak poleci exec to ten handler jest brutalnie i ordynarnie **wywalany**.

Natomiast w przypadku sygnałów `SIGUSR2` i `SIGINT` ustawiamy nie handler, a maskę sygnałów dla systemu. Maska sygnałów nie jest **wywalana** po execu, więc posłanie dziecku takich sygnałów generalnie go nie zakończy.

Ponieważ jednak w kodzie najpierw dowalamy dziecku `SIGUSR1`, to spada ono z rowerka po wypisaniu jednej kropki.

Dla pełności podajemy jeszcze inne możliwe scenariusze (bo ten jest nudny):

1. Najpierw strzelamy `SIGUSR2`, potem `SIGUSR1`, na końcu `SIGINT`.

   W takiej sytuacji wypisane są dwie kropki, bo tak jak napisaliśmy, maska jest zachowana przy `execu`.

2. Najpierw strzelamy `SIGINT`, potem `SIGUSR2`, na końcu `SIGUSR1`

   W tej sytuacji wypisane będą trzy kropki, bo `SIGINT` zostanie olany, `SIGUSR2` zablokowany, więc dopiero `SIGUSR1` ubije nam kota.

TL;DR: Przy execu maski się dziedziczą, handlery nie. Ale heca.

2.3.3 Zadanie 3

Liczba 1613 to jakiś *reaper* pewnie, a na Minixie 1 to *init*. Kiedy proces-rodzic spada z rowerka, jego dziecko dostaje te procesy jako „nowe" parent procesy czy coś takiego (żeby dziecko się nie zmieniło w zombie).

2.3.4 Zadanie 4

1. utworzenie nowego procesu
2. zablokowanie procesu
3. koniec procesu
4. przerwanie zegarowe
5. przerwanie IO

2.3.5 Zadanie 5

My tego nie mieliśmy, ale chillera utopia, program jest trochę różny w różnych latach i nie będzie tematów, których nie było.

That being said, możemy się zastanowić co może to robić.

Są takie przypadki, gdy chcemy przerwać „zamówienie" złożone do drivera. Proces może po prostu umrzeć i już nie ma komu odpowiedzieć. Jest to dość istotne kiedy:

1. próbujemy coś czytać z wolnego nośnika (np. dysku magnetycznego lub płyty CD)
2. czytamy coś z dysku sieciowego i mamy wolny internet

2.3.6 Zadanie 6

My to w ogóle omawialiśmy???? Otóż nie omawialiśmy driverów :(

W każdym razie odpowiedź to:

1. wspóldzielona pamięc
2. porty io
3. przerwania

2.3.7 Zadanie 7

Zalety stronicania na żądanie:

1. Nie rezerwuje pamięci, która jest nieużywana – zmniejsza narzut pamięciowy.
2. Więcej pamięci w pamięci (bo nie zaśmiecamy tym czego jeszcze nie użyliśmy) = speed
3. Szybszy start procesów (zwłaszcza jak mamy ich wiele jednocześnie).

Wady stronicowania na żądanie:

1. Opóźnienie przy pierwszym dostępie do pamięci związane z obsługą page faulta przez system operacyjny.
2. Bardziej złożona obsługa takiego systemu
3. Często dostajemy page faultem w ryj

2.3.8 Zadanie 8

2.3.9 Odpowiedź oczekiwana

Chodzi o mechanizm, który pod spodem przekazuje nam `argv`.

Zanim system odpali program to tworzy on stos, który na początku zawiera dane z pliku. Na to wszystko wkładane są argumenty i dopiero wtedy odpala proces.

2.3.10 Spekulacje

`int argc` oraz `char** argv`, które jest tablicą wskaźników zakończoną `NULL`em

2.3.11 Zadanie 9

Odpowiedź oczekiwana

Trzeba jakoś wymusić, żeby strona w pamięci była dwukrotnie zmapowana w przestrzeni procesu. Innymi słowy – żeby w pamieci fizycznej `arr[42]` odpowiadało dwóm miejscom w pamięci wirtualnej. W ten sposób inkrementując `arr[42]` zinkrementujemy też np. `arr[4138]` co zwiększy nam sumę o 2.

Możemy do tego doprowadzić celowo używając `mmap` (w dość głupi sposób). W szczególności możemy zmapować dwa razy ten sam plik na naszą pamięć i ogarnąć sprawę tak, żeby `arr` wskazywał na pamięć gdzie jest zmapowany tenże plik.

Podobną sztuczkę możemy zrobić ze stronami i w ten sposób nasz program ma „deterministyczne undefined behavior”

Czy jest to bez sensu? Tak. Czy jest to możliwe? Tak. :|

Spekulacje

Nie bardzo umiemy ustalić co może być tutaj dobrą odpowiedzią. Podejrzewamy, że mieliśmy zaalokowane mniej pamięci niż wynosi wartość zmiennej *size* (hyhyhy, trolololo): chodzi o fakt, że wolno nam czytać wszystko z naszej pamięci (czyli ze stron które dostaliśmy), tj. system operacyjny nas nie zabije w ten sposób. Natomiast problem jest w tym, że rozmiar stron jest niezależny od tego ile pamięci sobie zażyczyliśmy.

W związku z tym kod:

Może wypisać absolutnie cokolwiek nie rzucając przy tym błędu. welp, standardy.

2.3.12 Zadanie 10

A dlaczego miałaby być wyzerowana? :|

Nie rozmawialiśmy o tym skąd się może to wziąć, chyba po prostu za stary egzamin robimy :)

Obsługa sygnałów może nam tutaj trochę namieszać. W momencie przyjścia sygnału odpala się handler, który musi działać na jakimś stosie. Stos ten rozszerzy nam tymczasowo zwykły stos programu, a po zakończeniu obsługi zostaną tam śmieci z handlera.

Ponadto dokładany jest też stan procesu oraz `siginfo` (przed stosem handlera).

Nikt tego nie sprząta bo po co.

2.4 Egzamin 2018/2019

2.4.1 Zadanie 1

Odpowiedź to 14. Pierwszy wątek będzie mieć 3 dzieci:

1. Dziecko które wypisze sumarycznie 3 kropki i będzie mieć dwójkę dzieci:

2.4.2 Zadanie 2

Select działa tutaj tak, że informuje nas o możliwości poprawnego odczytu na pipe'ach. Generalnie to ten kod działałby fajnie, gdyby nie to że odczyt z pipe'a który jest zamknięty po drugiej stronie klasyfikowany jest jako poprawny odczyt. To oznacza, że gdy któryś z producentów przestanie produkować i zamknie pipe po swojej stronie, *select* cały czas będzie nas informować, że możliwy jest poprawny odczyt z któregoś pipe'ów. Jako, że wszystko jest owinięte w pętlę *while* dostajemy aktywne czekanie.

2.4.3 Zadanie 3

We do a little trolling: *alarm* mówi systemowi, by po jakimś czasie posłal nam sygnał `SIGALRM`. Tymczasem ten sygnał nie ma ustawionego handlera przez nasz program, więc gdy nic nie wpiszemy przez te 3 sekundy to poleci sygnał i nasz program zostanie przerwany. Whoopsie!

2.4.4 Zadanie 4

Patrz: 2.3.3

2.4.5 Zadanie 5

Nie robi się `printf` ani `exit` w handlerze.

2.4.6 Zadanie 6

Zwykły szary użytkownik nie może wywołać funkcji *sus_fork*, bo komunikację z kernelem wykonują serwery i drivery (np. *PM* czy *VFS*). Dla użytkowników przeznaczone są zwykłe system calle, które dzwonią do serwerów.

2.4.7 Zadanie 7

1. Powstanie nowego wątku
2. Wykopanie na koniec kolejki w związku ze zbyt długim czasem pracy
3. Wykopanie z początku kolejki i ponowne zakolejkowaniu po zablokowaniu się procesu na jakimś wywołaniu

2.4.8 Zadanie 8

Obstawiamy, że driver ten chce nam jakoś pomóc i doczytując sobie radośnie od razu bliższe sektory na dysku, by przyśpieszyć przyszłe dostępy do pamięci w tych okolicach (zakłada, że będą nam potrzebne – w sumie często słusznie).

Potwierdzone przez dr Kozika.

2.4.9 Zadanie 9

2.4.10 Odpowiedź oczekiwana

Początek pliku mamy od razu, a koniec pliku dopiero jak odczytamy bloki pośrednie. Asymptotycznie to jest czas logarytmiczny, ale w praktyce to drzewo jest bardzo rozgałęzione a dysk ma ograniczony rozmiar więc będzie to jakaś mała stała (rzędu 3).

2.4.11 Spekulacje

I-node trzyma plik jako takie drzewko, które zawiera wskaźniki do bezpośrednich danych oraz wskaźniki do kolejnego poziomu na którym znowu są bezpośrednie dane oraz kolejne wskaźniki...

Generalnie do pierwszych bajtów jest się łatwiej dostać bo nie wymagają skoków do tych pośrednich bloków, a żeby odczytać coś z końca pliku to trzeba wykonać jakoś logarytmicznie wiele skoków i odczytów z dysku. (NOTE: Tego z logarytmicznością za kija nie jesteśmy pewni bo w sumie to nie rozumiemy co się dzieje, elo)

2.4.12 Zadanie 10

page fault frequency mówi nam jak często programy odwołują się do „nie swojej' pamięci. To znaczy – obiecaliśmy im, że mają pamieć (pomijając przypadki że programista sam strzelił sobie w stopę, i wyleciał poza stronę i zarobił segfaultem na twarz), ale w celach optymalizacyjnych nie przydzieliśmy im fizycznie stron w pamięci. Oczywiście procesor nakrzyczy na program, że co on odwala i wtedy system operacyjny powinien grzecznie przeprosić i oddać programowi jego pamięć.

Ponadto, gdy przydzielimy już fizycznie jakąś pamięć programowi, ale ten od dłuższego czasu jej nie używał (a nam zaczyna brakować RAMu) możemy zrobić drobny fikołek i zapisać stan tej strony na dysk twardy, a pamięć fizyczną udostępnić innemu procesowi (który jej potrzebuje na teraz). Gdy proces który ordynarnie okradliśmy będzie chciał danych które sobie tam zapisał to się odwoła do tej pamięci w swojej przestrzeni adresów; wówczas memory management unit z CPU się oburzy i wywali page fault systemowi, a system spojrzy na ten wyjątek i powie „no tak, śmieszna sprawa, już oddaję tę pamięć", weźmie to co miał na dysku i odda stronę okradzionemu procesowi.

Podczas gdy ten mechanizm umożliwia nam pracę nawet wtedy, gdy programy wymagają sumarycznie więcej RAMu niż my go mamy, jeśli wszyscy aktywnie korzystają ze swojej pamięci to takie page faulty będą latać bardzo często i w sumie to nasze sztuczki będą powodować straszne mulenie. Stąd też fajnie by system obserwował ten wskaźnik; jeśli zobaczy, że page fault frequency jest zbyt duże, po prostu wywali jakiś proces do swapa (no bo inaczej się już dać nie będzie).

2.4.13 Zadanie 11

Printf jest buforowany trollololo. (A pamięć się kopiuje, kurczę beka)

2.5 Egzamin 2019/2020

2.5.1 Zadanie 1

Zdefiniujmy sobie radośnie oznaczenia:

1. p – PID programu
2. q – PID dziecka tego programu
3. r – PID reapera
4. x – PID rodzica naszego programu

Pierwsza linijka jest zawsze taka sama: p, x.

Znaczy, HAHA ŻARTUJEMY, bufor z printfa może się skopiować do dziecka jeśli się nie wypisało i będziemy mieć straszny syf. Nie rozpatrujemy tu tego przypadku.

Co do pozostałych dwóch, będą one wyglądać różnie zależnie od tego czy rodzic zdążył już spaść z rowerka przed dzieckiem i kto pierwszy co wypisał. Należy zauważyć, że teoretycznie reaper może być również rodzicem naszego programu. Wtedy r i x to to samo. Super.

Możemy zatem zaobserwować:

```
p x
0 q p
q p x
```

albo

```
p x
q p x
0 q p
```

albo

```
p x
q p x
0 q r
```

2.5.2 Zadanie 2

Generalnie to dup po prostu da nam inny sposób „odnoszenia" się do pliku? Kinda? Na przykładzie to działa w ten sposób, że jak sobie otworzę jakiś plik openem i będę z niego czytać, a potem zduplikuje deskryptor na niego za pomocą *dup*, to po próbie odczytu dalej czytamy z miejsca w którym wcześniej byliśmy; z kolei, jak otworzymy ten sam plik openem (jeszcze raz) na innym deskryptorze to będziemy czytać go na tamtym deskryptorze od początku (a na wcześniejszym od tego miejsca gdzie byliśmy).

A formalniej rzecz biorąc, *open* robi nam *open file description* i ustawia nam wskaźnik do niego na jakimś *file descriptor*; *dup* z kolei po prostu ordynarnie sklonuje nam *file descriptor* (ale *ofd* na które wskazujemy pozostanie to samo). Patrz: pytanie piąte z 2022 roku

2.5.3 Zadanie 3

Generalnie jak przeczytamy ten kod, to zauważymy, że pytanie jest o to, czy i kiedy `alarm` nas ubije (dziecka nie ubije, załączamy dokumentację forka).

Cytując dokumentację:

> alarm() arranges for a SIGALRM signal to be delivered to the calling process in seconds seconds.
>
> If seconds is zero, any pending alarm is canceled.
>
> In any event any previously set alarm() is canceled.

oraz

> Alarms created by alarm() are preserved across execve(2) and are not inherited by children created via fork(2).

W takim razie nieważne ile każemy spać dziecku, wypisze ono dwie kropki – jedną wewnątrz ifa, a drugą zaraz przed returnem.

Jeśli ustawimy alarm na wystarczająco późno (tak, aby oba sleepy się wykonały a dziecko zakończyło), to rodzic wypisze swoją kropkę i łącznie zaobserwujemy trzy. W przeciwnym razie na wyjściu będą tylko dwie kropki dziecka.

2.5.4 Zadanie 4

Sygnały się nie kolejkują, a więc jak wiele dzieci zakończy swą działalność to nasz program może wywołać swój handler tylko raz; to z kolei spowoduje, że będą dzieci które nie zostaną zwaitpidowane. Ups.

2.5.5 Zadanie 5

exit w handlerze, dobranoc (jeśli koniecznie (dalece niezalecane) chcemy wyleźć z programu w handlerze to używamy *_exit*)

2.5.6 Zadanie 6

2.5.7 Zadanie 7

Co robi kernel MINIXa?

Otóż wykonuje najbardziej niezbędne i podstawowe zadania systemu operacyjnego (trochę w kontraście do linuxa, gdzie kernel ma nieco większą odpowiedzialność.

Do takich zadań należą:

1. komunikacja między procesami
2. obsługa urządzeń oraz przerwań
3. IO
4. obsługa zegara

2.5.8 Zadanie 8

Nie jestem dumny z tego zadania

dr hab. Jakub Kozik

Generalnie to zadanie jest lekko zepsute.

`open` powoduje powiązania deskryptora z i-nodem, ale najpierw trzeba go znaleźć w systemie plików. Potrzebujemy więc z grubsza tyle odczytów ile wynosi głębokość (liczona w katalogach) ścieżki pliku, który otwieramy.

`lseek` z tego co rozumiem to zdaje się nic nie rusza tylko ustawia jakieś dane w tablicy deskryptorów

`read` czyta sobie reprezentację pliku do znalezienia odpowiedniego bajtu w pliku (no i czyta te kilka sektorów z dysku)

2.5.9 Zadanie 9

RS jest takim Guciem wśród serwerów, który siedzi i pilnuje, żeby serwery i drivery, które mają działać działały – jak coś umrze z niewiadomych przyczyn to RS próbuje to postawić na nogi.

Dodatkowo RS trzyma też informacje o endpointach serwerów. Jeśli włączymy nowy serwer, ale nie wiemy jaki ma numer, to możemy zapytać o to właśnie RSa.

2.5.10 Zadanie 10

To już widzieliśmy w poprzednich latach. :)

2.6 Egzamin 2020/2021

2.6.1 Zadanie 1

To zadanie jest śmieszne.

Rodzic po forku woła jeszcze raz maina, wypisuje kropkę, widzi f równe 1 i sobie exituje.

Natomiast dziecko odpala ten sam proces ponownie za pomocą execa (przy czym to jest nowy proces, bo exec and stuff, więc f jest wyzerowane). W ten oto piękny sposób cały cykl życia się powtarza i otrzymujemy proces wypisujący kropki w nieskończoność. Jednocześnie zauważmy, że w dowolnym momencie pracują maksymalnie 2 procesy równolegle, więc raczej nie walniemy w limit liczby procesów. Nawet Minix się na tym nie krztusi, a to już jest osiągnięcie.

2.6.2 Zadanie 2

Zakładając że exec się uda wypisze on dwie kropki i umrze.

Nastepnie po wysłaniu SIGINT do dzieci nie stanie się nic, bo handler = SIGIGN.

Po SIGUSR1 do całej grupy procesów wypiszą się kolejne dwie kropki gdyż odpali się handler na całej grupie procesów.

SIGUSR2 zamorduje nas wszystkich, więc finalnym outputem będzie

..

..

2.6.3 Zadanie 3

Żeby rozwiązać to zadanie trzeba wiedzieć co robi *fcntl* i flaga `FD_CLOEXEC`.

fcntl to jest taka fajna funkcja do kontrolowania deskryptorów (między innymi ich flag). `FD_CLOEXEC` to taka flaga deskryptora, która mówi, że w przypadku gdybyśmy robili execa to ten deskryptor należy zamknąć. Część ludzi mogła używać tego w swoich shellach na zasadzie magicznej linijki która naprawia problemy życiowe.

W kodzie co robimy to bierzemy sobie flagi które nasz deskryptor z numerem 1 (*stdout*) obecnie ma, po czym dorzucamy flagę `FD_CLOEXEC` do tego zestawu flag. To powoduje, że po obu execach zamknie się nam stdout. *cat* będzie usiłował wypisać coś na ten deskryptor, ale mu nie wyjdzie (bo się zamknął po execu) w związku z czym wywali na *stderr* komunikat, że mu się nie udało nic wypisać bo deskryptor *stdout* jest walnięty.

2.6.4 Zadanie 4

Flagi w `mkfifo` zdają się jedynie ustawiać pozwolenia, żeby użytkownik mógł z niej czytać i pisać więc tym się nie przejmujemy.

Pamiętamy, że `open` blokuje aż oba końce są otwarte, ale tutaj mamy taką zabawną sytuację, że robimy `O_RDWR`, a to znaczy, że otwieramy oba końce naraz.

W takim razie dziecko, otworzy sobie fifo, wpisze do niego TIC oraz odczyta z niego TIC jeszcze zanim rodzic zdąży wstać ze swojego `sleep(3)`. Oczywiście po zakończeniu roboty z fifo wypisze elegancko `TICTIC` na wyjście.

Po tym wszystkim budzi się rodzic, który chciałby coś odczytać z pustego fifo; `read` mu powie, że spoko, gość na drugim końcu z pewnością coś zaraz napisze. Problem jest taki, że na drugim końcu jest ten sam proces, który właśnie wisi na readzie.

Całe szczęście mamy alarm, który ubije rodzica, który czekałby tak w nieskończoność.

Ostatecznie na wyjściu zobaczymy jedynie `TICTIC`

2.6.5 Zadanie 5

Sygnały się nie kolejkują, a ten handler „odczekuje" tylko jedno dziecko gdy przychodzi mu `SIGCHLD`: jako że jednocześnie widać, że dzieci będą kończyć się mniej więcej w tym samym momencie, jest niemal pewne że przyjdzie mu mniej sygnałów niż jego dzieci się zakończyło, a więc nie wszystkie dzieci zostaną „odczekane". Tym samym powstaną procesy zombie.

2.6.6 Zadanie 6

Dirty mówi, czy ktoś coś wpisał na stronę, czyli czy będzie trzeba ją zapisywać z powrotem na dysk. Jeśli strona jest czysta to fajnie, nie musimy absolutnie nic robić.

Accessed mówi, że ktoś niedawno odczytywał z tej strony, czyli jest ona względnie potrzebna. System operacyjny sobie co jakiś czas zeruje ten bit na stronach, żeby mieć mniej więcej pojęcie czy ta strona jest bardzo używana.

2.6.7 Zadanie 7

`HARD_INT` oznacza *Hardware Interrupt*. W tym przypadku jako przerwanie systemowe dostajemy pewnie od jakiegoś urządzenia pełniącego rolę zegara i oznaczają że powinniśmy przesunąć czas o kolejny tick czy coś w tym stylu.

2.6.8 Zadanie 8

Pierwszy odczyt jest wolny bo wymaga odczytów z dysku (potencjalnie nawet z wielu rozrzuconych sektorów). Najpewniej Linux cache'uje sobie w RAMie zawartość tego pliku, dzięki czemu kolejne odczyty mogą jeszcze z niej skorzystać bez konieczności czytania z dysku ponownie.

2.6.9 Zadanie 9

Bo VFS ma swój stały endpoint, a IPC nie, więc za każdym razem trzeba pytać RSa który proces to IPC.

2.6.10 Zadanie 10

Bo tak jest po prostu szybciej i prościej.

Jeśli wszystko byłoby przekazywane przez wskaźnik to proces, który odbiera wiadomość musiałby sam odczytać/skopiować napis spod wskaźnika, który wskazuje na pamięć kogoś innego. Mechanizm, którego trzeba by tutaj użyć jest siłą rzeczy bardziej kosztowny niż po prostu odczytanie z wiadomości, która należy do odbiorcy.

2.7 Egzamin 2021/2022

2.7.1 Zadanie 1

Na początku powiedzmy sobie jasno: *kill 0* ubija wszystkie procesy w grupie procesów. To oznacza, że w outpucie litera `C` pojawi się maksymalnie 2 razy (zanim dzieci obu równoległych procesów ubiją całą grupę).

Może jeszcze być taka heca, że `fork` się nie udał i zwrócił nam -1, a `kill -1` strzela do **wszystkich** w systemie. Na szczęście w naszym scenariuszu wszyscy to te same procesy, które ubija `kill 0`.

Oczywiście na początku pojawi się tylko jedna litera `A`. Pozostaje pytanie o to, co dzieje się z literami `B` – musi pojawić się co najmniej jedna, ale, podobnie, mogą być też dwie.

Prowadzi nas to do następujących możliwości (hopefully żadnej nie pominęliśmy):

1. `AB`
2. `ABB`
3. `ABC`
4. `ABBC`
5. `ABBCC`
6. `ABCB`
7. `ABCBC`

W pewnym momencie dziecko zabije wszystkich, którzy jeszcze żyją. Pytanie tylko kiedy.

2.7.2 Zadanie 2

2.7.3 Zadanie 3

Patrz: 2.4.13

2.7.4 Zadanie 4

Na początku warto powiedzieć, co tu się w ogóle dzieje: każde wywołanie programu (poza takim gdy `c` wynosi 0) się forkuje i execuje siebie samego z argumentem `c` pomniejszonym o 1, który ma czytać wiadomości od swojego ojca, który zmienił się w :kota:.

Czyli, efektywnie, jak się to rozrysuje i chwilę pomyśli wyjdzie nam taki łańcuch kotów, które przepisują to co im napisano na swój output; jest to pewnego rodzaju głuchy telefon.

Problem jest jeden: deskryptory. Nie zamykamy deskryptorów. Duplikujemy je *dup2*, ale nie zamykamy deskryptorów, które dostaliśmy przy utworzeniu pipe'a. Ponieważ deskryptory dziedziczą się do naszych dzieci, a nasze dzieci w dodatku majstrują kolejne pipe'y, szybko dołazimy do wniosku, że liczba deskryptorów jest liniowa od liczby `c`. A to już jest pewien problem, bo taki Minix dosyć szybko umrze dla wielu deskryptorów. W sumie Linux też, ale nieco wolniej. W każdym razie: chcemy po prostu zamykać deskryptory z pipe'ów po ich zduplikowaniu za pomocą *dup2*.

2.7.5 Zadanie 5

Najlepiej chyba wyjaśnić jak to działa pod spodem: w kernelu każdy deskryptor reprezentowany jest przez strukturę `struct_fd`. Każda z tych struktur trzyma wskaźnik do *open file description* właśnie; *ofd* trzyma informacje takie jak to, z jakimi flagami otwarty jest plik lub to gdzie mamy w nim swój „kursor". Wiele *fd* może wskazywać na to samo *ofd* (np. możemy je klonować za pomocą funkcji *dup*).

Co do flag: według jakiejś stronki *GNU* obecnie istnieje tylko jedna flaga dla *fd* i jest to `FD_CLOEXEC`, która powoduje że deskryptor się zamyka po jakimś execute.

Natomiast *ofd* ma tych flag już więcej, na przykład `O_APPEND`, mówiąca o tym że dopisujemy do końca pliku. Fajne, nie?

2.7.6 Zadanie 6

1. *Ready* – Jeżeli proces czeka aż procesor pozwoli mu zacząć dziamać
2. *Blocked* – Jeśli proces czeka na jakieś zdarzenie (np. na odczyt *read* czy dostanie odpowiedzi na wiadomośc na Minixie)
3. *Swapped* – Jeśli system uzna że nie ma odpowiednio dużo RAMu i musi zapisać stan jakiegoś procesu na dysk twardy to może czasem tak zrobić; wtedy status procesu jest *swapped*.
4. *Zombie* – Proces umarł, ale rodzic jeszcze go jeszcze nie zwaitował, więc system musi trzymać o nim jakąś informację w tabeli procesów (np. status z jakim się on zakończył).

Patrz: `https://www.geeksforgeeks.org/states-of-a-process-in-operating-systems/`

2.7.7 Zadanie 7

Patrz: 2.4.12

2.7.8 Zadanie 8

Dobrym pomysłem jest, aby dziecko wykonało się najpierw, jako że, chances are, dziecko wywoła sobie radośnie exec. Jednocześnie, należy pamiętać, że dziecko i rodzic powinni mieć (teoretycznie) całkowicie oddzielną pamięć; nie musimy jednak jej klonować przy forku dopóki któryś z tych procesów actually czegoś nie napisze do pamięci (bo do tego momentu fakt, że pamięć między nimi jest współdzielona ujdzie nam na sucho).

Jako, że dzieci mają tendencję do execowania siebie samych bez pisania do pamięci, fajnie byłoby, aby wykonały się najpierw (bo rodzic po forku może chcieć coś napisać do pamięci, co wymusi kopiowanie całej pamięci – niespecjalnie efektywne, biorąc pod uwagę że pamięć dziecka po execu i tak pójdzie na śmietnik).

2.7.9 Zadanie 9

Patrz: 2.5.9

2.7.10 Zadanie 10

Strony mogą być dowolnie przemapowane na pamięć fizyczną, więc tak naprawdę jak ustalimy na ilu stronach i jak są rozmieszczone zmienne to dostaniemy odpowiedź na pytanie.

Strona pierwsza powinna być od jakiegoś adresu zerowego do 4095, strona druga od 4095 do 8 tysięcy z hakiem, strona trzecia od 8 tysięcy z hakiem + 1 do ponad dwunasty tysięcy. Stąd mamy, że X znajdzie się na stronie drugiej, a Y i Z znajdą się na stronie trzeciej.

Teraz z tego jak te zmienne są trzymane na wirtualnych stronach musimy wywnioskować, gdzie znajdują się w pamięci fizycznej. Wiemy na pewno, że jako że Y i Z są na tej samej stronie wirtualnej (i Z jest po Y) to będą one również na tej samej stronie fizycznej, w tej samej kolejności. X nie jest z nimi na tej samej stronie, więc w pamięci fizycznej może być przed nimi lub po nich. To zostawia nas z dwoma permutacjami:

1. `XYZ`
2. `YZX`

2.8 Egzamin 2022/2023

2.8.1 Zadanie 1

Trzy.

`setsid` tworzy nową, niezależną od innych procesów sesję.

`kill 0 SIGUSR1` wyśle sygnał do wszystkich procesów w tej samej sesji, więc w obrębie jednej takiej grupy tylko jeden proces wypisze kropkę i od razu zabije wszystkich pozostałych.

Takie sesje będą trzy – sesja początkowego programu oraz jego dwóch wnuków, więc zobaczymy trzy kropki.

2.8.2 Zadanie 2

Zobaczymy litery „a", „c" oraz „d".

Wynika to z faktu, że `printf` jest buforowany a `_exit` bezceremonialnie kończy program.

`write` nie jest buforowany, więc od razu zobaczymy jego wyjścia.

`exit` natomiast grzecznie flushuje bufory zanim zakończy program.

W takim razie tylko kombinacja `printf` + `_exit` nie zostanie wyświetlona.

2.8.3 Zadanie 3

Bo jest to kod biblioteki dynamicznie ładowanej (`so` oznacza Shared Object) więc może być tak, że wiele procesów naraz używa tego kodu. Pisanie po współdzielonej pamięci nie jest natomiast zbyt dobrym pomysłem.

Moglibyśmy dać każdemu procesowi osobną kopię, ale to nam zjada pamięć, bo te biblioteki nie są współdzielone bez powodu (są duże).

2.8.4 Zadanie 4

`SIGKILL` oraz `SIGSTOP`

2.8.5 Zadanie 5

Proces zombie, czyli taki, którego technicznie nie ma, ale jego rodzic nie wywołał `wait`, więc wpis wisi nadal w tabeli procesów.

2.8.6 Zadanie 6

Nie ustawiamy ani `act.sa_mask` ani `act.sa_flags`, więc może tam być **absolutnie cokolwiek**, więc może się stać cokolwiek.

2.8.7 Zadanie 7

Jeśli proces nie wykorzystał w pełni swojego kwantu czasu to po odblokowaniu zostaje wrzucony na początek swojej kolejki.

2.8.8 Zadanie 8

Pewnie jakiś śmieszek wpisał jeden bajt na pozycji `6205` i zostawił wcześniej pusty plik. Na szczęście i-node'y są sprytne i nie trzymają bloków, których nie ma. Te 4KB wynika z faktu, że musimy trzymać ten blok w którym dane są.

2.8.9 Zadanie 9

Jak sama nazwa wskazuje, robimy kopię pamięci **wirtualnej**, czyli takiej, która jest adresowana dla każdego procesu osobno. Siłą rzeczy musimy znać identyfikatory procesów, żeby znać fizyczną lokację adresów.

2.8.10 Zadanie 10

System cache'uje plik. Pierwsze użycie czyta z dysku, kolejne dwa czytają z RAMu.