Infrastruktura do zarządzania urządzeniami i sterownikami urządzeń w systemie operacyjnym FreeBSD z częściową implementacją w systemie operacyjnym Mimiker

(The device and device drivers infrastructure in FreeBSD operating system with partial implementation in Mimiker operating system)

Jan Mazur Wojciech Moczulski

Praca inżynierska

Promotor: dr Piotr Witkowski

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

3 września 2018

Streszczenie

System komputerowy nie jest niepodzielną jednostką funkcjonalną. Składa się na niego wiele, często bardzo różnych, urządzeń wymagających odmiennego traktowania. Szczególne traktowanie każdego urządzenia z osobna wprowadzałoby chaos, stąd potrzeba spójnego podsystemu zarządzającego szeroką gamą urządzeń w jednolity sposób.
W tej pracy opisaliśmy organizację systemu komputerowego, rodzaje urządzeń oraz sposoby ich połączenia. Przedstawiamy również jeden ze sposobów zarządzania urządzeniami i ich sterownikami - podsystem NewBus systemu operacyjnego FreeBSD. Omówimy proces wykrywania i inicjalizacji urządzeń, ich organizację w systemie operacyjnym oraz rezerwację zasobów sprzętowych. Na koniec przedstawimy implementację infrastruktury do zarządzania urządzeniami w systemie operacyjnym Mimiker wzorowaną na infrastrukturze NewBus.

Computer system is not an indivisible functional unit. It is made of many, often very different devices, requiring different handling. Special treatment of every device in a system would cause chaos, thus need for consistent subsystem responsible for handling a wide range of devices in an unified way.
In this paper, we described computer system structure, types of devices and ways they are connected. We present one way of managing devices and their drivers - NewBus subsystem in FreeBSD operating system. We will discuss process of detecting and initializing devices, their organization in operating system and hardware resources reservation. At the end we will present implementation of infrastructure responsible for handling devices in Mimiker operating system inspired by NewBus infrastructure.

Spis treści

Rozdział 1.

Wprowadzenie

1.1. Czym jest komputer? Rola urządzeń zewnętrznych

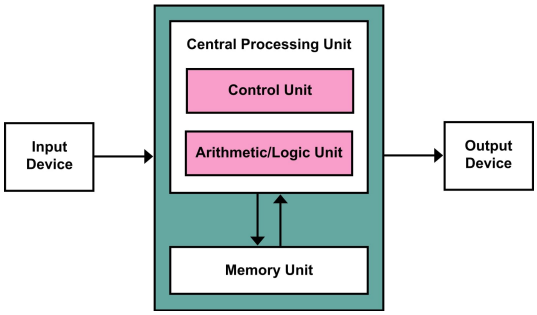
Komputer z założenia jest programowalną maszyną liczącą. Oznacza to, że potrafi przeprowadzać obliczenia na podstawie podanych przez użytkownika instrukcji. Instrukcje te muszą w wyczerpujący sposób opisywać obliczenia które mają zostać wykonane, oraz muszą być podane w sposób możliwy do zinterpretowania przez maszynę. Na przykład zakodowane przy pomocy tablic programowych, kart perforowanych, taśm magnetycznych, bądź zadane z użyciem dalekopisu. Maszyna po zdekodowaniu instrukcji, bez ingerencji człowieka, przeprowadza obliczenia. Gdy obliczenia się zakończą, ich wynik powinien być przekazany użytkownikowi, na przykład poprzez te same media, które służyły do przekazania wejściowych instrukcji maszynie.

W podobny do wyżej opisanego sposób, John von Neumann w 1945 roku zdefiniował działanie *automatycznego systemu liczącego*[1]. W tak działającej maszynie naturalnym wydało mu się wyodrębnienie kilku komponentów, z których każdy miał ściśle określone zadanie. Wśród komponentów były takie jak jednostka licząca, jednostka kontrolna, pamięć, oraz moduły odpowiedzialne za pobieranie instrukcji zakodowanych na zewnętrznych dla maszyny nośnikach i zapisywanie na tychże nośnikach wyników obliczeń.

W opisanej przez Neumanna architekturze urządzenia spełniają rolę interfejsu pomiędzy maszyną, a człowiekiem. Służą do obustronnej komunikacji, pozwalają zdefiniować dane wejściowe, a następnie po skończonych obliczeniach prezentują dane wyjściowe. Prawdopodobnie stąd nazwa **urządzenia wejścia-wyjścia (ang. I/O devices, Input/Output devices)**.

Aby prowadzić interakcję z jednostką liczącą, urządzenia tego typu są niezbędne.

Mimo upływu ponad 70 lat, wpływ Architektury von Neumanna dalej jest widoczny w nowoczesnych systemach komputerowych.



Rysunek 1.1: Uproszczony schemat architektury von Neumanna [62]

1.2. Rozwój urządzeń

Postęp technologiczny sprawiał, że komputery stawały się coraz bardziej interaktywne, pozwalały na większą swobodę interakcji z użytkownikiem niż systemy z przetwarzaniem wsadowym [11][6].

Pierwsze systemy operacyjne z podziałem czasu (ang. timesharing) [9], Potrafiły obsługiwać wiele urządzeń wejścia-wyjścia pracujących na rzecz wielu różnych użytkowników jednocześnie. Potrzeba trwałego przechowywania danych spowodowała pojawienie się urządzeń pamięci masowej. Począwszy od nośników magnetycznych, przez optyczne na dyskach półprzewodnikowych kończąc.

Z czasem cena komputera jak i jego wielkość drastycznie się zmniejszyły. Dawniej komputer zajmował powierzchnię sporego pomieszczenia. Już nie tylko wielkie korporacje i instytucje rządowe mogły sobie pozwolić na posiadanie zaawansowanej maszyny obliczeniowej ale także osoby prywatne. Tak powstały **komputery osobiste (ang. Personal Computer - PC)**.

Aby komputery były coraz bardziej powszechne i dostępne, nie mogły już wymagać specjalnie wyszkolonych operatorów do ich obsługi. Potrzebne były dużo prostsze metody interakcji, tak aby jak najwięcej osób w krótkim czasie mogło nauczyć się obsługi komputera. Wymagało to zarówno rozwoju oprogramowania (szczególnie systemu operacyjnego), jak i sprzętu.

Archaiczne, drogie, często skomplikowane i nieporęczne już urządzenia wejścia-wyjścia takie jak te wymienione wcześniej, zostały zastąpione przez klawiatury, monitory, ekrany dotykowe, czy nawet systemy interakcji głosowej i wizualnej.

Każdy kolejny etap rozwoju urządzeń sprawiał, że stawały się one coraz bardziej skomplikowane, a tym samym złożoność sprzętowych i programowych mechanizmów ich obsługi rosła.

Komputery PC [21] wyrosły na wysoce modularne systemy, w których użytkownik może z powodzeniem modyfikować możliwości swojej maszyny poprzez podłączenie urządzenia na karcie rozszerzeń obsługującej pewien ustalony protokół ko-

munikacji. Urządzenia peryferyjne stały się naturalnym sposobem an rozszerzanie możliwości systemu. Liczba urządzeń jest w zasadzie nieograniczona, a każde z nich musi być odpowiednio obsłużone przez system.

Każde nowe urządzenie, którego maszyna ma być być świadoma, z którym wymienia dane, musi być odpowiednio oprogramowane. Kod obsługujący dane urządzenie nazywamy **sterownikiem urządzenia**. Jednakże sterowniki nie obsługują urządzeń bezpośrednio. Wchodzą one w interakcję z układem (bądź zbiorem układów elektronicznych) zwanym **kontrolerem**. Pośredniczą one w komunikacji między systemem operacyjnym a konkretnym urządzeniem (lub zbiorem urządzeń) oferując systemowi operacyjnemu przystępniejszy protokół komunikacji (wciąż jednak bardzo złożony) [5].

1.3. Kontrolery

Urządzenia z czasem stawały się coraz bardziej skomplikowane, stąd również komunikacja z nimi stawała się bardziej zawiła. Aby masowo produkować urządzenia w łatwy sposób rozszerzające możliwości komputera, wprowadzono standardy komunikacji z urządzeniami. Np. dowolny kontroler dysku SATA [26] powinien obsłużyć dowolny dysk SATA.

Obecnie niemal każde urządzenie wejścia-wyjścia jest podłączone do systemu komputerowego za pośrednictwem układu zwanego **kontrolerem** którego zadaniem jest udostępnienie systemowi operacyjnemu prostszego interfejsu (który pomimo wszystko jest bardzo złożony) komunikacyjnego z urządzeniem zewnętrznym [23][5].

Kontroler natomiast komunikuje się z urządzeniem poprzez ustandaryzowany protokół danego typu **magistrali**, taki jak na przykład nie używane już SCSI [14] i ATA/IDE [15], czy powszechnie występujące PCI i USB, przybliżone w rozdziałach 3.2. i 3.3..

Sztandarowym przykładem zadania które pełni kontroler jest tłumaczenie logicznych adresów bloków dyskowych [3] (LBA) na adresowanie oparte na wyborze cylindra, sektora i głowicy (CHS) [7]. Konwersja ta nie należy do trywialnych z tego powodu, że cylindry wewnętrzne mają mniej sektorów od zewnętrznych, oraz ze względu na możliwe wystąpienie wadliwych sektorów i ich przemapowanie na sektory poprawne. Po wyznaczeniu adresu CHS, kontroler dokonując odczytu bądź zapisu musi zmienić położenie głowicy z cylindra nad którym obecnie się znajduje na pozycję nad docelowym cylindrem. Następnie musi poczekać, aż docelowy sektor w wyniku rotacji dysku przesunie się pod głowicę. Wtedy wykonywany jest odczyt bądź zapis i liczone są sumy kontrolne. Logika zawarta w kontrolerach przypomina niewielkie wbudowane komputery, które zostały zaprogramowane przez dostawcę sprzętu do wykonywania tej pracy.

Obecnie używane kontrolery najczęściej są zintegrowane z płytą główną, ale

mogą być też podłączone do odpowiednich magistral jako karty rozszerzeń.

1.4. Magistrale

Urządzenia w systemie komputerowym wymagają odpowiedniej organizacji. Muszą być ze sobą fizycznie połączone, aby wchodzić ze sobą w interakcję. Służą do tego **magistrale** [5].

Magistralą nazywamy *zespół linii przenoszących sygnały oraz układów wejścia-wyjścia służących do przesyłania sygnałów między połączonymi urządzeniami w systemach mikroprocesorowych* [12].

W większości magistral można wyodrębnić trzy podstawowe szyny (kanały, linie): szynę sterującą (kontrolną), adresową i danych. Sygnały elektryczne na pierwszej z nich interpretowane są jako wybór aktualnie podejmowanej akcji na magistrali, takiej jak np. odczyt i zapis danych. Za pomocą szyny adresowej wybierane jest miejsce z, bądź do którego będzie wykonywany transfer. Ostatnia szyna, jak sama nazwa wskazuje, służy do przesyłania danych.

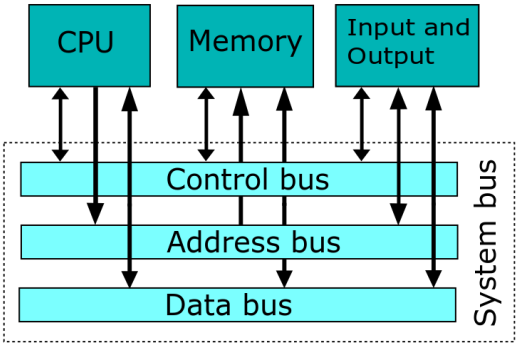
Magistrale można podzielić ze względu na typ transmisji na **równoległe** (np. PCI [3.2.]) i **szeregowe** (np. PCI-express [16]). Pierwszy typ posiada wiele linii (przewodów), co pozwala na równoległą transmisję bitów słowa danych. Trudnością jest zapewnienie dotarcia wszystkich bitów przesyłanej wiadomości w tym samym czasie. W drugim przypadku wiadomości wysyłane są szeregowo, jednym pasmem (ang. lane), bit po bicie, niczym pakiety w sieci ethernet. Zazwyczaj używa się wielu równoległych pasm w połączeniu punkt-punkt [13] (jak w PCI-express).

Komunikacja między urządzeniami podłączonymi do magistrali musi być dobrze zorganizowana. Wyobraźmy sobie telekonferencję wielu osób. Gdyby wszyscy mówili jednocześnie, nikt nie byłby w stanie zrozumieć nikogo. Analogiczna sytuacja dotyczy magistral.

W przypadku, gdy więcej niż jedno urządzenie na magistrali może inicjować transakcje potrzebny jest arbitraż. Standard magistrali danego typu może specyfikować jak ma on wyglądać, bądź zezwalać na zewnętrzną implementację arbitra (np. w PCI [17]).

Z magistralą najczęściej skojarzony jest kontroler potrafiący obsługiwać urządzenia danego typu. Np. kontroler magistrali SATA potrafi obsługiwać wszystkie dyski SATA. Kontrolerami, które wspomagają komunikację na magistrali są PIC i DMA, które zostaną omówione w rozdziałach 2.4. i 2.5..

Urządzenie inicjujące transakcję, mające w danej chwili kontrolę nad magistralą nazywamy **nadzorcą (ang. master)** . Pozostałe urządzenia nazywamy wtedy **podległymi (ang. slave)**.



Rysunek 1.2: Magistrala systemowa [63]

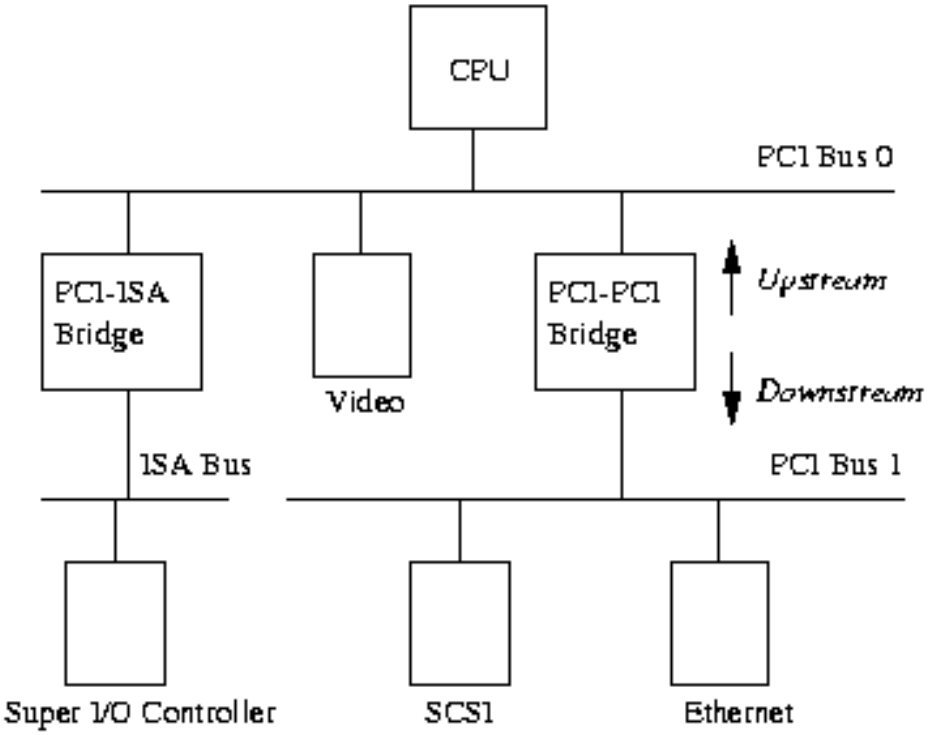
Większość magistrali o wysokiej przepustowości, pozwala każdemu urządzeniu podłączonemu do niej, na inicjacje operacji [50]. Mechanizm ten nazywamy *bus mastering*, bądź *first-party DMA*. Każde urządzenie może ubiegać się o dostęp do magistrali, może wyjść z inicjatywą. Pozwala to na komunikację między urządzeniami z pominięciem procesora, czyli na **bezpośredni dostęp do pamięci (ang. Direct Memory Access - DMA)**. *First-party DMA* różni się od *third-party DMA*, tym że w pierwszym przypadku to urządzenie dokonuje operacji na pamięci, a w drugim specjalny układ - kontroler DMA.

Przy starcie system komputerowy musi zidentyfikować podłączone do niego urządzenia. Obecnie niemal wszystkie nowoczesne magistrale (jak np. PCI czy USB) posiadają protokół detekcji urządzeń, który pozwala wykryć jakie urządzenia są podłączone i jak się z nimi komunikować. Przestarzała, lecz wciąż używana magistrala ISA [3.1.] nie posiada protokołu detekcji, przez co system musi w pewien sposób założyć z góry, że dane urządzenie jest podłączone w odpowiedni sposób [4.5.].

Najprostszy, najwcześniej stosowany model systemu komputerowego z jedną magistralą spinającą wszystkie urządzenia nazywany jest systemem z **magistralą systemową (ang. system bus)** . Schematycznie przedstawiony na rysunku 1.4.. Taki model pozwalał obniżyć koszty jak i zmniejszyć rozmiar maszyny, zwiększając modularność systemu. Używany na przykład w komputerze IBM PC z użyciem magistrali ISA.

Mimo, że nowoczesne systemy komputerowe pod względem organizacji sprzętu wyglądają dużo bardziej skomplikowanie, to model ten dobrze się sprawdza jako reprezentacja sprzętu dla programisty systemu operacyjnego nie zajmującego się sterownikami urządzeń.

Poza podstawowym przesyłaniem danych większość magistral pozwala urządzeniom zgłaszać przerwania [2.4.], oraz dopuszcza możliwość transferu danych za pomocą DMA [2.5.].



Rysunek 1.3: Mostki i magistrale [69]

1.5. Mostki

W nowoczesnych systemach komputerowych istnieje wiele podłączonych do siebie magistral z różnymi protokołami. Potrzebujemy w takim wypadku mechanizmu, który pozwoli tłumaczyć protokół jednej szyny na drugi. Z pomocą przychodzą układy zwane **mostkami** [5]. Możemy chcieć tłumaczyć protokół magistrali ISA na PCI, bądź nawet PCI na PCI kiedy magistrala PCI jest podłączona do innej magistrali PCI. Standardowo w systemie występuje jedna magistrala PCI i jak każda magistrala tego typu ma ona limit urządzeń. Aby móc ich podłączyć więcej, rozwiązaniem jest właśnie mostek typu PCI-PCI.

1.6. Organizacja systemu komputerowego

Na przestrzeni lat, używano wielu różnych sposobów połączenia komponentów komputera. Jednym z podstawowych jest wymieniona wcześniej 1.4. koncepcja jednej szyny systemowej [24].

W miarę jak procesory i pamięci stawały się coraz szybsze, zdolność jednej ma-

gistrali do obsługi całego ruchu stawała się ograniczona. Z natury niezbyt szybkie urządzenia wejścia-wyjścia dzieliły jeden kanał komunikacji z szybkimi i kluczowymi dla systemu urządzeniami. Pojawiły się więc inne warianty organizacji komponentów w systemie [5].

W przypadku procesorów firmy Intel magistrala najbliżej procesora nosi nazwę *Front Side Bus* [18]. Intel ostatecznie upublicznił informację o niej, stąd niektóre procesory AMD również posiadają takową. W niektórych kontekstach nazwa ta jest również wykorzystywana jako nazewnictwo przyprocesorowych magistral niezależnie od producenta.

Jedną z koncepcji, będącą do niedawna w powszechnym użyciu jest architektura **mostu północnego** [20][28] i **mostu południowego** [19][28]. Została ona wprowadzona przez firmę Intel.

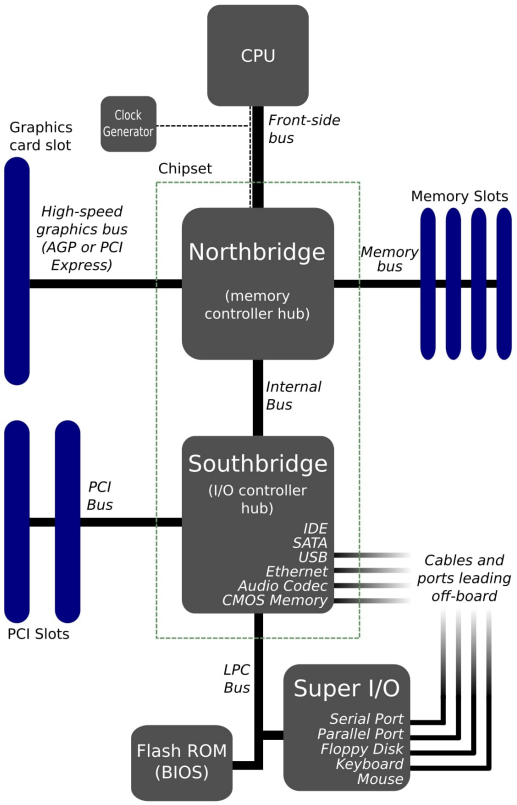
Rozdzielenie wynika z potrzeby jak najszybszej interakcji procesora z pamięcią operacyjną jak i kartą graficzną. Na tej ścieżce komunikacji znajdował się most północny, który do procesora podłączony był magistralą *Front Side Bus*. Dodatkowo do mostu północnego podłączony most południowy miał obsługiwać wszystkie akcje związane z wejściem i wyjściem. W tym wypadku nie jest wymagana tak duża przepustowość jak dla pamięci RAM czy GPU, ponieważ urządzenia IO z natury są wolne i nie pod nie optymalizuje się system komputerowy.

Początkowo mostki były ze sobą połączone magistralą PCI, jednakże z czasem połączenie to zastąpiono nowszymi: DMI (Direct Media Interface) - w przypadku Intela, oraz UMI (Unified Media Interface) w przypadku AMD.

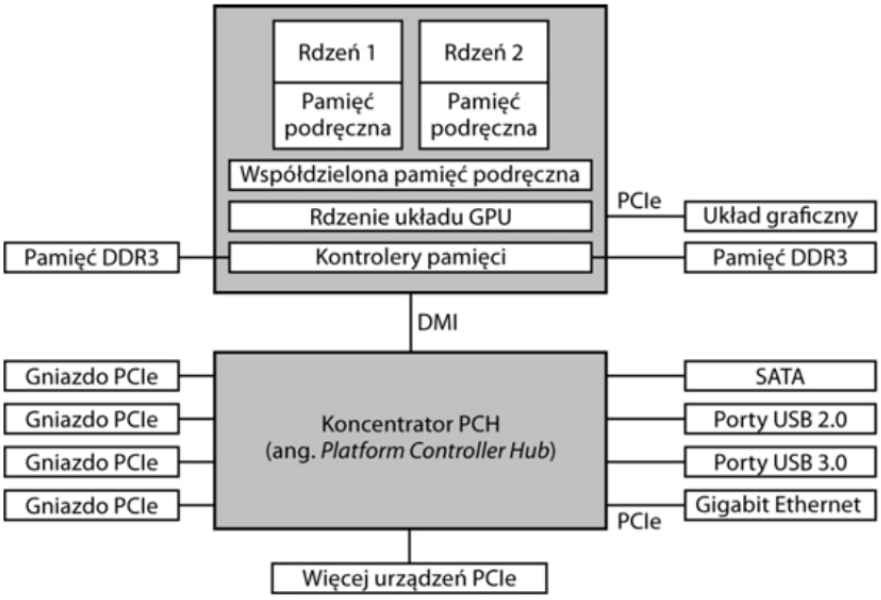
Przykładem systemu komputerowego opartego na architekturze mostów północnego i południowego jest platforma Malta z procesorem MIPS, pod którą działa system operacyjny Mimiker. Zostanie ona omówiona w rozdziale 8.1..

Z czasem z powodów wydajnościowych kontrolery pamięci i GPU zostały zintegrowane z procesorem, a funkcje mostu południowego przejął **koncentrator PCH (ang. Platform Controller Hub)** używany w systemach komputerowych opartych o procesor architektury x86. Połączenie między CPU a PCH realizowane było przez DMI w kolejnej wersji.

Obecnie dwie najnowsze architektury komunikacji komponentów w systemie komputerowym to *Quick Path Interconnect* [30] firmy Intel i *Hypertransport* [29] firmy AMD, które nie mieszczą się w zakresie tej pracy. Są one sprzętowo zaimplementowane na płytach głównych. Głównie służą do łączenia wielu procesorów realizując między innymi protokoły spójności pamięci podręcznych [48]. W dużym uproszczeniu bardziej przypominają sieć połączeń punkt-punkt [13] z przesyłaniem komunikatów, niż magistrale w standardowym rozumieniu.



Rysunek 1.4: Architektura mostu północnego i południowego [64]



Rysunek 1.5: System komputerowy oparty na procesorze architektury x86 z PCH [65]

Rozdział 2.

Sposoby komunikacji z urządzeniami i ich zasoby

2.1. Zasoby

Sterowniki urządzeń oprócz standardowego, dynamicznego rezerwowania pamięci wymagają również rezerwacji innego rodzaju zasobów [31]. Najważniejszymi i najczęściej używanymi z nich są linie przerwań, zakresy portów wejścia-wyjścia, jak i zakresy pamięci używanej przez mechanizm MMIO, oraz kanały DMA [4][58]. Obecność tych zasobów wynika bezpośrednio ze sprzętowej organizacji systemu komputerowego jak i charakterystyki samego urządzenia. Urządzenia jak i kontrolery mogą udostępniać pamięć, posiadać zestaw rejestrów, bądź reagować na przerwania czy obsługiwać transfery DMA. Charakterystyka jak i sposoby użycia danych zasobów do komunikacji z systemem operacyjnym zostaną omówione w następnych sekcjach pracy, natomiast mechanizmy ich rezerwacji zostaną przedstawione na przykładzie systemu FreeBSD w rozdziale poświęconym temuż systemowi .

2.2. Komunikacja

Każdą komunikację z urządzeniem jesteśmy w stanie uprościć do wysyłania i odbierania danych, co czasem jest wspomagane przerwaniami. Do głośnika będziemy wysyłali dane reprezentujące dźwięk, z mikrofonu będziemy je odbierali. Klawiatura po wciśnięciu przycisku informuję system za pomocą przerwania o nowych danych czekających na odczyt.

Najczęściej wyróżnia się trzy uzupełniające się wzajemnie rodzaje komunikacji. Najprostsze i nierzadko najlepsze jest regularne **odpytywanie (ang. pooling)** urządzenia. Czasem jednak się nie jest to satysfakcjonujące rozwiązanie. Na przykład klawiatura bardzo rzadko (z punktu widzenia komputera) ma coś do zasygnalizo-

wania. Ale kiedy zostanie wciśnięty przycisk, chcielibyśmy, aby system operacyjny dowiedział się o tym jak najszybciej. Temu właśnie służą przerwania. Jest to sposób, w jaki urządzenie może samo zainicjować komunikację, a nie czekać na inicjatywę ze strony systemu operacyjnego. W celu zoptymalizowania obsługi transferów danych, wymyślono mechanizm **DMA - Direct Memory Access**, dzięki któremu procesor może zainicjować transfer danych, zająć się czymś innym, a kiedy dane będą gotowe, odczytać je z pamięci operacyjnej, co trwa znacznie krócej, niż odczyt bezpośrednio z dysku.

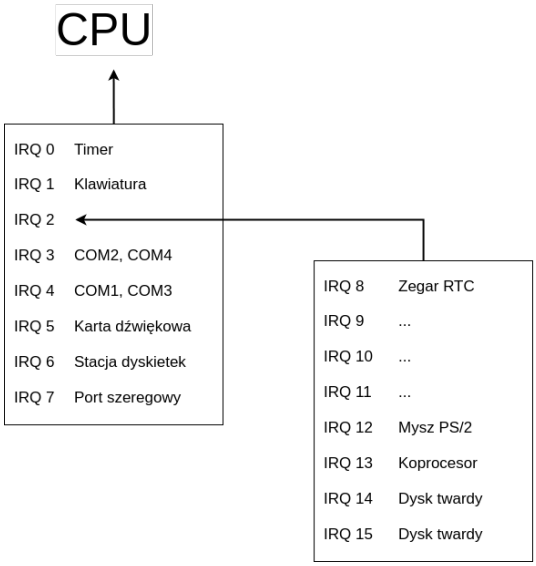
2.3. PIO - Programowane wejście-wyjście

Programowane wejście-wyjście [32] jest zdecydowanie najprostszym mechanizmem IO. Procesor na polecenie sterownika inicjuje transfer danych z urządzeniem, następnie czeka na informację zwrotną. Niestety takie podejście jest uciążliwe, jeśli urządzenia i transfer danych są wolniejsze, niż sam procesor, co zazwyczaj ma miejsce.

Istnieją dwa sposoby implementacji tego mechanizmu. Pierwszy z nich - **MMIO - ang. Memory Mapped IO - wejście-wyjście mapowane na pamięć** [33] - polega na użyciu adresów z tej samej przestrzeni, dla pamięci operacyjnej oraz urządzeń. To sprawia, że interakcja z urządzeniami odbywa się za pomocą dokładnie tych samych instrukcji procesora, oraz w większości tej samej infrastruktury, co dostęp do pamięci. Urządzenia w tym przypadku są podłączone do tej samej szyny adresowej, co pamięć operacyjna.

Drugie podejście to **PMIO - ang. Port Mapped IO - wejście-wyjście mapowane na porty** - gdzie pamięć urządzeń znajduje się w odrębnej przestrzeni adresowej. Jest to bardziej skomplikowany sposób implementacji PIO, ponieważ wymaga dodatkowych instrukcji procesora, dlatego raczej nie znajduje ono zastosowania w prostych procesorach RISC oraz w systemach wbudowanych. Jeśli PMIO zostało zaimplementowane z użyciem odrębnej fizycznej szyny adresowej, może to oznaczać zysk na wydajności.

PMIO nie zabiera adresów z fizycznej przestrzeni, co było dużą zaletą w przypadku 16 i 32 bitowych procesorów. W układach 64-bitowych przestaje mieć to znaczenie. PMIO w zasadzie składa się z dwóch instrukcji - LOAD i STORE. Chcąc dokonać inkrementacji wartości w rejestrze urządzenia, musimy wykonać co najmniej trzy instrukcje - LOAD, INCREMENT, STORE. Przy MMIO można użyć każdej instrukcji która działa na pamięci, czyli na przykład w przypadku x86 wystarczy jedna instrukcja INC.



Rysunek 2.1: Sposób połączenia dwóch układów Intel 8259

2.4. Interrupt driven io

Przerwania są mechanizmem pozwalającym na zainicjowanie komunikacji przez urządzenie, a nie procesor. Jest to niezwykle ważne w przypadku urządzeń, które wymagają uwagi rzadko, ale natychmiast po wystąpieniu obsługiwanego przez nie zdarzenia. Przykładem jest klawiatura, karta sieciowa i przycisk na obudowie. Regularne ich odpytywanie byłoby skrajnie nieopłacalne, ponieważ prawie zawsze okazywałoby się, że nie wymagają uwagi. Zbyt rzadkie odpytywanie z kolei sprawiałoby, że urządzenie byłoby mało responsywne.

Tradycyjnie, przerwanie było zgłaszane przez zmianę stanu na odpowiednim wyprowadzeniu procesora lub kontrolera przerwań. W momencie wykrycia przerwania przez procesor, przekazuje on sterowanie do odpowiedniej procedury systemu operacyjnego, której zadaniem jest zapisanie kontekstu aktualnie wykonywanego procesu, odnalezienie odpowiedniego sterownika i przekazanie kontroli zarejestrowanej przez niego procedurze obsługi przerwań. Następnie system operacyjny przywraca kontekst uprzednio wykonywanego procesu i oddaje mu sterowanie.

Procesor nie posiada osobnych wyprowadzeń do obsługi każdego urządzenia z osobna. Zamiast tego, podłączało się do niego układy wspomagające nazywane **PIC - ang. Programmable Interrupt Controller - Programowalny Kontroler Przerwań**. Tradycyjnie stosowano układ Intel 8259[35], który był w stanie obsłużyć 8 linii przerwań (IRQ), samemu używając jednej. Nie było to wystarczająco, więc dokładano drugi taki układ, podłączając go do portu nr 2 pierwszego. To dawało w sumie 15 wolnych linii przerwań i to przez długi czas wystarczało.

Większość linii było na stałe przypisanych do konkretnych urządzeń:

Oczywiście powodowało to problemy. Autorzy listy nie mogli przewidzieć jaki sprzęt powstanie w przyszłości.

Układy PIC obsługują maski oraz priorytety. Dwa przerwania mogą nadejść jednocześnie. W takim przypadku to o wyższym priorytecie zostanie przetworzone pierwsze. Jeśli przerwanie nadejdzie w trakcie obsługi innego, zazwyczaj chcemy wywłaszczyć poprzednie tylko jeśli ma ono niższy priorytet od nowego.

Maski przerwań służą do wyłączenia konkretnych linii, co jest potrzebne na przykład, gdy nie chcemy, aby w trakcie obsługi danego przerwania, nadeszło drugie identyczne. Także przerwania muszą zostać wyłączone na czas wykonywania kodu, który musi zostać wykonany atomowo, co w tym wypadku znaczy, że zmiana kontekstu w trakcie jego wykonania mogłaby mieć złe skutki. Przykładem tego jest zapisanie kontekstu wątku przy przełączeniu kontekstu.

W internecie nadal można znaleźć instrukcje typu "jak rozwiązać konflikt między myszą podłączoną przez port COM 1 oraz modemem podłączonym przez port COM 3". Oba te urządzenia domyślnie używały linii IRQ4. Rozwiązać taki problem dało się na wiele sposobów, na przykład przez konfigurację w biosie, fizyczną zmianę slotu, do którego urządzenie jest podłączone lub przez przestawienie odpowiednich zworek w sprzęcie. Podłączonych urządzeń było coraz więcej, więc zarządzanie tym ręcznie byłoby problematyczne. Na szczęście powstały interfejsy z automatyczną konfiguracją.

Wraz z nadejściem procesorów wielordzeniowych, zwykły PIC przestał wystarczać. Z pomocą przyszedł **APIC - ang. Advanced Programmable Interrupt Controller**, który zasadniczo składa się z dwóch modułów. Pierwszym jest **Local APIC - LAPIC**, jeden dla każdego rdzenia procesora, często w niego fizycznie wbudowany. Każdy z nich obsługuje 255 linii przerwań.

LAPIC mogą służyć także do przekazywania Między-Procesorowych przerwań (**IPI - Inter-Processor Interrupt**). W ten sposób rdzenie procesora są w stanie komunikować się między sobą na przykład w celu nakazania innym procesorom wyczyszczenia TLB (TLB shootdown), lub pamięci podręcznej MMU, gdy stanie się ona nieważna.

Drugim modułem jest **I/O APIC**, zazwyczaj przypadający w jednym egzemplarzu na każdy kontroler magistrali. I/O APIC jest w stanie obsłużyć zazwyczaj 24 linie przerwań, a jako, że samych układów może być wiele, ilość linii przerwań nie stanowi już problemu. Każdy I/O apic trzyma w swojej pamięci informację, do którego LAPIC powinien przekazać które przerwanie.

Wraz ze wzrostem zapotrzebowania na ilość linii przerwań oraz prędkość, zaczęto stosować Message Signalled Interrupts - **Przerwania Sygnalizowane Wiadomością - ang. Message Signaled Interrupts - MSI** [37]. Polega to na przesyłaniu krótkiej informacji opisującej przerwania poprzez główny kanał komunikacji, zamiast stosowania osobnych ścieżek do obsługi przerwań. Najczęściej służy do tego MMIO.

Urządzenie chcąc wyzwolić przerwania, zapisuje jej opis pod ustalony adres, a docelowy APIC wyłapuje je i przekazuje do procesora.

PCI wspiera MSI od wersji 2.2. W PCI-Express jest to jedyny sposób dostarczania przerwań. MSI zwiększa liczbę obsługiwanych przerwań i znacząco upraszcza implementację sprzętu. Dzieje się to kosztem utrudnienia projektowania urządzeń. Współdzielenie szyny danych z pamięcią może w niektórych przypadkach rozwiązać problem wyścigów między przerwaniem, a zapisem do pamięci przez urządzenie. Nie potrzebna jest w takim wypadku dodatkowa synchronizacja, co oznacza zysk na wydajności.

Specjalnym rodzajem przerwań są przerwania synchroniczne. Nie są one wyzwalane przez sprzęt, lecz są wynikiem działania programu, a więc działają synchronicznie względem niego. Przykładem jest implementacja wywołań systemowych w niektórych architekturach - odpowiednia instrukcja powoduje przerwanie, które jest obsługiwane przez jądro, które następnie określa jakie wywołanie systemowe miało miejsce. Innym przykładem są wyjątki, takie jak dzielenie przez zero czy błąd stronicowania.

Specjalny plik /proc/interrupts pokazuje liczbę przerwań, które miały miejsce na każdej z linii.

Listing 2..1: Uproszczony plik /proc/interrupts [1]

```
1        CPU0     CPU1
2   0:        9        0  IR-IO-APIC  2-edge       timer
3   1:   219100        0  IR-IO-APIC  1-edge       i8042
4   8:        0        1  IR-IO-APIC  8-edge       rtc0
5   9:    11112       51  IR-IO-APIC  9-fasteoi    acpi
6  12:  1772765        0  IR-IO-APIC  12-edge      i8042
7  18:        0        0  IR-IO-APIC  18-fasteoi   i801_smbus
8  40:        0        0  DMAR-MSI    0-edge       dmar0
9  41:        0        0  DMAR-MSI    1-edge       dmar1
10 42:        0        0  IR-PCI-MSI  458752-edge  PCIe PME
11 43:        0        0  IR-PCI-MSI  464896-edge  PCIe PME
12 44:        0        0  IR-PCI-MSI  466944-edge  PCIe PME
13 45:        0        0  IR-PCI-MSI  49152-edge   snd_hda_intel:card0
14 46:  2941213        0  IR-PCI-MSI  512000-edge  ahci[0000:00:1f.2]
15 47:        0      448  IR-PCI-MSI  327680-edge  xhci_hcd
16 48:        0        0  IR-PCI-MSI  360448-edge  mei_me
17 49:  9885017        0  IR-PCI-MSI  32768-edge   i915
18 50:  9825069        0  IR-PCI-MSI  1048576-edge iwlwifi
19 51:      770       21  IR-PCI-MSI  1572864-edge nvkm
20 52:        0     4488  IR-PCI-MSI  442368-edge  snd_hda_intel:card1
21 (...)
22 TLB:  776285  786805  TLB shootdowns
23 (...)
```

[1] Wydruk pochodzi z systemu *Manjaro 17.1.11 Hakoila*, z jądrem w wersji *x86_ 64 Linux 4.17.12-*

Pierwsza kolumna to nazwa przerwania, zazwyczaj będąca numerem, który jest także jego priorytetem. Następnych kilka to liczba przerwań wygenerowanych na danym procesorze przez dane przerwanie. Wyraźnie widać, że większość z nich była kierowana do jednego lub dwóch procesorów. Reszta tabeli mówi o sposobie dostarczenia przerwania oraz nazwie urządzenia. System operacyjny stara się równomiernie rozłożyć przerwania na procesory. Da się to także zrobić ręcznie w łatwy sposób.

2.5. DMA - Direct Memory Access

Zarówno PIO, jak i przerwania nie są w stanie obsłużyć dużych transferów danych w optymalny sposób. Dla przykładu przy komunikacji ze stacją dysków CD, przerwania same w sobie nie nadawałyby się w ogóle. Mogą one tylko wspomagać PIO, informując o gotowości urządzenia. PIO z kolei zablokowałoby procesor na cały czas odczytu, co jest marnotrawstwem możliwości procesora, ponieważ większość czasu spędziłby czekając na dane.

Z pomocą przychodzi moduł **DMA - ang. Direct Memory Access - bezpośredni dostęp do pamięci** [39], czyli układ który wykonuje transfery między pamięcią i urządzeniami (także między dwoma miejscami w pamięci, lub między dwoma urządzeniami), bez ciągłego zaangażowania procesora, który w tym przypadku odpowiada jedynie za odpowiednie zaprogramowanie DMA.

Wyróżnia się dwa rodzaje transferów DMA - Third-party DMA i First-party DMA. Third-party DMA oznacza kontroler, który wykonuje zlecone przez procesor transfery. Taki kontroler DMA może być wielokanałowy, a więc prowadzić więcej niż jeden transfer na raz. First-party DMA z kolei pozwala urządzeniom na wykonywanie transferów do i z pamięci bez zaangażowania procesora.

DMA zazwyczaj działa w jednym z dwóch trybów - Burst Mode - Trybie Wiązki, lub Cycle Stealing Mode - Trybie Podkradania Cykli. W pierwszym przypadku transfer odbywa się dużymi blokami, pomiędzy dwoma miejscami w pamięci lub urządzeniami, a magistrale które biorą udział w danym transferze są całkowicie temu dedykowane, to znaczy, że nawet procesor nie może z nich korzystać. W cycle stealing mode, kontroler DMA przesyła kilka bajtów pomiędzy urządzeniem a pamięcią, po czym oddaje szynę do użytku przez CPU. Procesor przetworzywszy dane, zwraca kontrolę układowi DMA, który dokonuje kolejnego transferu. Ten sposób jest co prawda wolniejszy, ale sprawia, że dane przetwarzane są na bieżąco.

1.1-MANJARO, i procesorem *Intel Core i5-5200U @ 4x 2.7GHz.*

Rozdział 3.

Omówienie wybranych magistral.

3.1. ISA

Magistrala ISA [40] powstała jako magistrala systemowa [1.4.] komputerów linii IBM PC w 1981 roku. Zaprojektowana została z myślą o podłączaniu kart rozszerzeń do systemu komputerowego z możliwością bus masteringu [1.4.].

System operacyjny z urządzeniami peryferyjnymi podłączanymi pod magistralę ISA musi mieć szczegółową wiedzę na temat podłączanego sprzętu i jego organizacji. Oznacza to, że użytkownik systemu musi ręcznie konfigurować każde podłączane urządzenie, czyli ustawić parametry takie jak linie przerwań, adresy portów IO, kanały DMA [2.5.][2.1.] [1].

Tego rodzaju problemy spowodowały pojawienie się systemu ISA PnP (ISA Plug and Play [43]), który wprowadzał modyfikacje do sprzętu, BIOSu oraz systemu operacyjnego, które pozwalały na automatyczną konfigurację urządzeń. Standard ten jednak nie przyjął się ponieważ nie został wystarczająco szybko dopracowany, oraz mało urządzeń z niego korzystało. Dodatkowo magistralę ISA zaczęła wypierać magistrala PCI [3.2.].

Obecnie większość nowoczesnych komputerów nie posiada fizycznej magistrali ISA, jednak wszystkie systemu architektury x86 posiadają wirtualną magistralę ISA przez którą wbudowane w płytę główną kontrolery mogą świadczyć proste usługi jak na przykład monitorowanie temperatury i napięcia podzespołów.

3.2. Peripherial Component Interconnect - PCI

Nie będziemy omawiali magistrali PCI w całości, a skupimy się na aspektach ważnych z punktu widzenia programisty systemu operacyjnego. Omówimy między

[1] Na przykład w systemie FreeBSD takim mechanizmem jest *device.hints* [82]

innymi sposób wykrywania urządzeń podłączonych do magistrali, adresowanie urządzeń i alokację oferowanych przez nie zasobów pamięciowych.

Magistrala PCI pojawiła się jako następca przestarzałej już magistrali ISA. Służy ona w głównej mierze do podłączania kart rozszerzeń do płyty głównej w komputerach PC. Poza tym może również łączyć kontrolery zintegrowane z płytą główną (np. most północny z południowym).

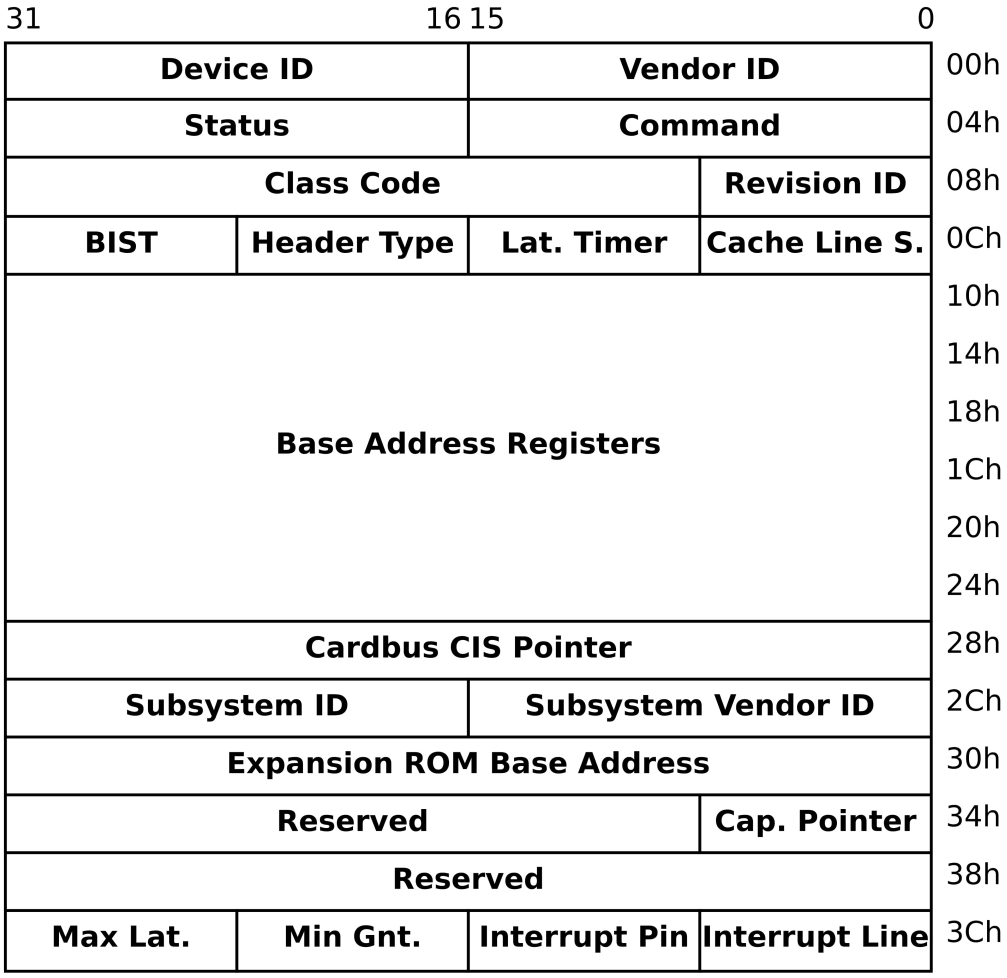
Magistrala PCI posiada mechanizm **Plug and Play** [42]. Termin ten oznacza, że magistrala daje możliwość detekcji i konfiguracji urządzeń do niej podłączonych bez wcześniejszej manualnej konfiguracji, bądź interwencji użytkownika. Wszelakie konflikty między urządzeniami również mogą być rozwiązane automatycznie.

System jest w stanie zidentyfikować podłączone urządzenie (w przypadku PCI-express nawet podłączone w trakcie działania systemu (ang. *hot plug*)), na podstawie numerów identyfikacyjnych, a następnie odpowiednio dobrać sterownik. Informacje o danym urządzeniu zapisane są w ustandaryzowanej, znajdującej się na każdym urządzeniu PCI, przestrzeni zwanej **przestrzenią konfiguracyjną**. Jest to pamięć do której możemy zapisywać i czytać z niej. Znajdziemy tam takie informację jak numer producenta danego sprzętu (VID - Vendor ID), klasę urządzenia (Class Code), oraz numer wewnętrznie nadany przez producenta pozwalający zidentyfikować konkretny model urządzenia (DID - Device ID).

Pola Subsystem ID (SSID) i Subsystem Vendor ID (SVID) identyfikują konkretny model urządzenia (np. karty rozszerzeń). VID identyfikuje producenta chipsetu, a SVID producenta karty rozszerzeń na której znajduje się owy chipset. SSID wybierany jest z tej samej puli co DID. Jako przykład mogą posłużyć karty graficzne, gdzie producentem chipsetu będzie nVidia, a producentem całej karty będzie Asus.

Listing 3..1: Część wydruku komendy lspci -v -nn [2]

```
03:00.0 3D controller [0302]: NVIDIA Corporation GM108M
[GeForce 840M] [10de:1341]
  Subsystem: ASUSTeK Computer Inc. GM108M [GeForce 840M]
    [1043:12df]
  Flags: bus master, fast devsel, latency 0, IRQ 51
  Memory at f6000000 (32-bit, non-prefetchable) [size=16M]
  Memory at e0000000 (64-bit, prefetchable) [size=256M]
  Memory at f0000000 (64-bit, prefetchable) [size=32M]
  I/O ports at e000 [size=128]
  Expansion ROM at f7000000 [disabled] [size=512K]
  Capabilities: [60] Power Management version 3
  Capabilities: [68] MSI: Enable+ Count=1/1 Maskable
  Capabilities: [78] Express Endpoint, MSI 00
  Capabilities: [250] Latency Tolerance Reporting
  Capabilities: [258] L1 PM Substates
  Kernel driver in use: nouveau
  Kernel modules: nouveau
```



Rysunek 3.1: Przestrzeń konfiguracyjna PCI [66].

Na listingu 3..1 klasa (Class Code) urządzenia to *3D controller* identyfikowana numerem 0x0302. Producentem (VID) jest NVIDIA Corporation (0x10de), a samo urządzenie (DID) o nazwie GeForce 840M ma wewnętrznie nadany numer 0x1341. SVID to ASUSTek Computer Inc. (0x1043) a SSID to Geforce 840M (0x12df).

Numery VID i SVID nadawane są przez organizację PCI-SIG [25]. Na podstawie pary VID, DID, oraz Class Code, system operacyjny może automatycznie dobrać odpowiedni sterownik. Korzystając z SVID i SSID system może sparować z urządzeniem sterownik implementujący pewne dodatkowe funkcje wprowadzone przez producenta karty.

Dostępy do przestrzeni konfiguracyjnej można wykonywać na dwa sposoby. Pierwszy (*legacy*), zwany *Configuration Access Mechanism* pozwala na zapis i odczyt przestrzeni konfiguracyjnej przez porty IO o adresach 0xCF8 i 0xCFC. Ich symboliczne, zwyczajowe nazwy to `PCI_CONFIG_ADDRESS` i `PCI_CONFIG_DATA`.

[2]Wydruk pochodzi z systemu *Manjaro 17.1.11 Hakoila*, z jądrem w wersji *x86_ 64 Linux 4.17.12-1.1-MANJARO*, i procesorem *Intel Core i5-5200U @ 4x 2.7GHz*

Drugi mechanizm zwany *Memory-Mapped Configuration* mapuje przestrzenie konfiguracyjne urządzeń w fizyczną (a z pomocą systemu operacyjnego w wirtualną) przestrzeń adresową. Dostępny jest w następcy standardowego PCI zwanym **PCI-express**.

Dostęp do portów `PCI_CONFIG_ADDRESS` i `PCI_CONFIG_DATA` jest możliwy, w zależnośći od architektury, poprzez instrukcje `in` i `out` jak na przykład w architekturze x86, albo rejestry te mogą być dostępne poprzez adresowanie pamięci, tak jak na przykład na platformie Malta z procesorem MIPS.

Przy pisaniu bądź czytaniu z przestrzeni konfiguracyjnej w trybie *legacy*, musimy wyspecyfikować magistralę PCI (może być ich wiele w systemie), urządzenie i funkcję, oraz przesunięcie (ang. *offset*) rejestru w przestrzeni konfiguracyjnej.

Poniżej wydruk komendy `lspci`. W pierwszej kolumnie znajdują się adresy urządzeń PCI w formacie `MAGISTRALA:URZĄDZENIE:FUNKCJA`, zapisane szesnastkowo. Pod jednym numerem urządzenia, może być wiele różnych funkcji danego urządzenia, które mogą przez system być traktowane jako osobne byty (tak jak w wypadku urządzenia o numerze 1c poniżej), bądź mogą to być różne urządzenia zorganizowane w bardziej kompaktowy sposób (np. urządzenie o numerze 1f na listingu 3..2).

Listing 3..2: Część wydruku komendy lspci [3]

```
00:00.0 Host bridge: Intel Corporation Broadwell-U Host Bridge
        -OPI (rev 09)
00:02.0 VGA compatible controller: Intel Corporation HD
        Graphics 5500 (rev 09)
00:03.0 Audio device: Intel Corporation Broadwell-U Audio
        Controller (rev 09)
00:14.0 USB controller: Intel Corporation Wildcat Point-LP USB
        xHCI Controller (rev 03)
00:1b.0 Audio device: Intel Corporation Wildcat Point-LP High
        Definition Audio Controller (rev 03)
00:1c.0 PCI bridge: Intel Corporation Wildcat Point-LP PCI
        Express Root Port #1 (rev e3)
00:1f.0 ISA bridge: Intel Corporation Wildcat Point-LP LPC
        Controller (rev 03)
00:1f.2 SATA controller: Intel Corporation Wildcat Point-LP
        SATA Controller [AHCI Mode] (rev 03)
00:1f.3 SMBus: Intel Corporation Wildcat Point-LP SMBus
        Controller (rev 03)
02:00.0 Network controller: Intel Corporation Wireless 7265
        (rev 59)
03:00.0 3D controller: NVIDIA Corporation GM108M [GeForce 840M]
        (rev a2)
```

[3] Wydruk pochodzi z systemu *Manjaro 17.1.11 Hakoila*, z jądrem w wersji *x86_64 Linux 4.17.12-1.1-MANJARO*, i procesorem *Intel Core i5-5200U @ 4x 2.7GHz*

Odpowiednio zakodowany w słowie maszynowym adres zapisujemy w porcie `PCI_CONFIG_ADDRESS`.

Listing 3..3: Odczyt z przestrzeni konfiguracyjnej

```
1 uint32_t pci_addr = 0x80000000 | bus << 16 | device << 11
2                     | function <<  8 | offset;
3 sysOutLong(PCI_CONFIG_ADDRESS, address);
4
5 return (uint16_t)((sysInLong (PCI_CONFIG_DATA) >>
6        ((offset & 2) * 8)) & 0xffff);
```

Przy dostępie do `PCI_CONFIG_DATA` wymagane jest ustawienie 31-szego bitu zwanego *ECD (Enable Config Data)*.

Po odpowiednim zaadresowaniu, odczyt i zapis wykonujemy przez port `PCI_CONFIG_DATA`, korzystając z instrukcji in i out, bądź w przypadku zmapowania tego rejestru w pamięć, za pomocą zwykłych instrukcji operujących na pamięci. W powyższym przykładzie zadanie to spełniają funkcje `sysOutLong` i `sysInLong`.

W przypadku zmapowanych w pamięć przestrzeni konfiguracyjnych, system operacyjny powinien dla każdego potencjalnie podłączonego urządzenia zarezerwować 4 KiB miejsca. Przestrzenie konfiguracyjne dostępne są wtedy poprzez C-ową tablicę postaci `config_space[bus][device][function]`. Daje to 256 możliwych wyborów magistrali, po 32 urządzenia, po 8 funkcji, gdzie każda przestrzeń konfiguracyjna jest wielkości 4 KiB. Razem 256 MiB pamięci potrzebnej na obsługę jednej magistrali PCIe. Zatem na 32-bitowych systemach może dojść do sytuacji, gdzie fizycznie mamy 4 GiB pamięci RAM, ale 256 MiB jest nieużywanych, ponieważ to miejsce zajmują przestrzenie konfiguracyjne urządzeń PCIe.

Następnymi ważnymi rejestrami w przestrzeni konfiguracyjnej są *Base Address Registers (w skrócie BAR)*. Jeśli urządzenie dysponuje zasobami pamięciowymi, to w BAR zakodowane są informacje mówiące ile pamięci udostępnia urządzenie, oraz jak wykonywać do niej dostępy z poziomu systemu.

BAR dzielą się na 2 rodzaje ze względu na sposób w jaki można wykonywać dostępy do udostępnianego przez urządzenie zasobu. Istnieją BAR typu `MEMORY` i typu `IO PORT`. Zasoby pierwszego typu muszą być odwzorowane w fizyczną przestrzeń adresową odpowiadającą pamięci RAM, natomiast zasoby drugiego typu przechowujące adres portu, mogą przechowywać dowolny adres przestrzeni portów IO. Dostęp do odpowiadających nim przestrzeni wykonuje się kolejno za pomocą mechanizmów MMIO i PMIO.

Przy pierwszej interakcji z urządzeniem w BAR zakodowana jest wielkość zasobu pamięciowego, jego typ, jak i flagi. Tabelki 3.1 i 3.2 prezentują kodowanie początkowych informacji w BAR.

[4] `https://wiki.osdev.org/PCI#Base_Address_Registers`

Tablica 3.1: Kodowanie informacji w BAR typu MEMORY

31-4	**3**	**2-1**	**0**
wyrównany do 16 bajtów adres bazowy	bit prefetchable	typ[4]	zawsze 0

Tablica 3.2: Kodowanie informacji w BAR typu IO

31-2	**1**	**0**
wyrównany do 4 bajtów adres bazowy	zarezerwowane	zawsze 1

System czytając wszystkie możliwe przestrzenie konfiguracyjne jest w stanie zidentyfikować wszystkie podłączone urządzenia. Po ich zidentyfikowaniu czyta rejestry BAR każdego z urządzeń, dekodując zawarte w nich informacje. Dla każdego BAR po zarezerwowaniu odpowiedniej ilości miejsca w przestrzeni adresowej, bądź w przestrzeni IO, w zależności od typu BAR, system wpisuje pierwszy adres zarezerwowanej przestrzeni do rejestru BAR.

3.3. USB

Najpopularniejszym w dzisiejszych czasach standardem podłączania urządzeń zewnętrznych jest **USB - Universal Serial Bus - Uniwersalna Magistrala Szeregowa**. Tym samym złączem możemy podłączyć drukarkę, kamerę, dysk zewnętrzny i klawiaturę. Występuje w kilku rozmiarach - standardowe montowane w komputerach A, czasem spotykane w drukarkach B, dawniej popularne mini USB A i B, bardzo częste w telefonach Micro, oraz zdobywające coraz większą popularność USB - C.

Standard mówi, że istnieje dokładnie jeden kontroler nadrzędny - master [1.4.]. Tylko on może inicjować komunikację (a więc nie ma tu typowych linii przerwań).

Pozostałe urządzenie są albo bezpośrednio podłączone do niego, albo przez huby, zwane też koncentratorami. Tworzy to topologię drzewa. Urządzenia są adresowane za pomocą 7 - bitowego identyfikatora, z czego 0 jest zarezerwowane. Zatem teoretycznie jeden kontroler może obsłużyć do 127 urządzeń. Jednak jednak pozwala on na przepustowość wynoszącą maksymalnie 480 Mbit/s w standardzie 2.0, dlatego czasem stosuje się więcej niż jeden.

Każde urządzenie USB może zarejestrować do 32 kanałów logicznych - 16 wejścia i 16 wyjścia. Dzięki nim, jedno fizyczne urządzenie może obsługiwać więcej, niż jedną funkcję, na przykład kamera internetowa może mieć osobne kanały dla mikrofonu, kamery oraz przycisków sterowania. Mogą one transmitować dane krótkimi wiadomościami, lub potokami. Są trzy tryby potoków: izochroniczny, masowy (bulk) oraz przerwania.

Tryb izochroniczny gwarantuje minimalną przepustowość, ale nie gwarantuje poprawnej transmisji. Ma on zastosowanie na przykład przy komunikacji z mikrofonem. Tryb masowy (bulk) - gwarantuje poprawny transfer, ale nie minimalną przepustowość, która jest zależna od innych transferów. Przykładem zastosowania jest pamięć masowa. Tryb przerwaniowy gwarantuje minimalną częstotliwość próbkowania.

Warto zauważyć, że przerwania są tu zaimplementowane z użyciem bardzo częstego próbkowania, ale dokonywanego przez kontroler a nie przez procesor i sterownik urządzenia, dzięki czemu nie ma to wpływu na wydajność reszty systemu. Tryb przerwaniowy jest używany w urządzeniach takich jak mysz i klawiatura.

Standard USB został zaprojektowany z myślą o Plug and Play oraz hot plug. Urządzenia USB możemy podłączyć lub odłączyć z działającego komputera w dowolnym momencie. Wyjątkiem są nośniki pamięci, do których zapis może być wspomagany pamięcią podręczną, w takim wypadku należy najpierw dane zsynchronizować (polecenie **sync** w linuksie).

Sterownik jest wybierany automatycznie przez system na podstawie informacji dostarczonych przez urządzenie - VID, PID, klasa urządzenia. Numer VID jest nadawany producentowi przez organizację standaryzacyjną.

Standard stał się niezwykle popularny dzięki swojej prostocie. Przewód w najpopularniejszym standardzie 2.0, zawiera tylko dwie linie przenoszące dane. Prostota protokołu ułatwia implementację w urządzeniach. Dzięki klasom urządzeń, producent sprzętu nie musi zazwyczaj dostarczać sterownika.

Informacje na temat urządzeń można zobaczyć używając polecenia *lsusb -v* w systemie linux.

Rozdział 4.

Urządzenia i sterowniki w systemach operacyjnych

4.1. Sterowniki jako moduły jądra

W przypadku architektury mikrojądra sterownik może być specjalnym programem przestrzeni użytkownika. W systemach uniksowych kod sterowników urządzeń wykonuje się zazwyczaj z uprawnieniami jądra ze względu na monolityczną architekturę. Nie chcielibyśmy jednak ponownie kompilować jądra za każdym razem gdy dodajemy nowy sterownik. Dodatkowo posiadany system komputerowy w zasadzie nigdy nie korzysta ze wszystkich dostępnych sterowników. Z tych powodów, w większości systemów, sterowniki ładowane są do systemu dynamicznie jako tzw **moduły jądra**. Nazywane w środowisku linuxowym LKM od *Loadable Kernel Module* i KLD w środowisku BSD [22].

Moduły jądra to dowolny kod, który może być załadowany do jądra w dynamiczny sposób, czyli nawet podczas pracy systemu. Dodanie kodu w taki sposób nie wymaga ponownej kompilacji reszty kodu. Moduł jądra może również zostać odczepiony zwialniając zaalokowane wcześniej zasoby. Aby osiągnąć taki efekt moduły jądra muszą implementować interfejs, który został zadany przez jądro systemu. Kod modułu jądra, niezależnie od systemu operacyjnego, musi najczęściej wyspecyfikować co najmniej procedurę reagującą na zdarzenia związane z modułem takie jak jego załadowanie i odczepienie, oraz zdefiniować strukturę opisującą ten moduł, która będzie przeczytana przez system operacyjny i umożliwi załadowanie modułu. W systemach opartych na jądrze Linux, moduły jądra mają strukturę bibliotek dzielonych, a jądro implementuje konsolidator dynamiczny.

Minusem modułów jądra może być ich użycie przez złośliwe oprogramowanie (tzw. rootkity [27]).

Z tego powodu systemy wymagające wysokiego poziomu bezpieczeństwa mogą

mieć systemy operacyjne skompilowane statycznie, z wyłączoną możliwością ładowania modułów. Drugim minusem może być drobny narzut czasowy związany z obsługą modułów. Dodatkowo, kod modułu po załadowaniu zazwyczaj nie zajmuje z kodem jądra ciągłego zakresu adresów - występuje fragmentacja. Przez to wykonywanie kodu może wykorzystywać więcej wpisów TLB [22].

W niektórych systemach operacyjnych np. przeznaczonych na mniejsze urządzenia (IOT, wbudowane, RTOS), czy w przypadku niektórych systemów o architekturze mikrojądra wybór składowych części jądra jest możliwy tylko podczas kompilacji. Statycznie skompilowane jądro pozwala pominąć całą skomplikowaną logikę odpowiadającą za zarządzanie modułami jądra.

Jak oprogramowanie klienckie korzysta z funkcjonalności sterownika? Wiele systemów operacyjnych stosuje filozofię pochodzącą z systemów z rodziny Unix, polegającą na reprezentowaniu urządzeń za pomocą plików. Za wywołaniami systemowymi dotyczącymi plików stoją różne semantyki w zależności od tego czy mamy do czynienia ze standardowym plikiem dyskowym będącym ciągiem bajtów, czy może ze specjalnym typem np. urządzeniem.

W przypadku pliku reprezentującego urządzenie, jego sterownik może implementować w zasadzie dowolne zachowanie dla każdego odnoszącego się do niego plikowego wywołania systemowego.

Między innymi w taki sposób jak wyżej opisany jądra udostępnia kod użytkownikowi, jednak samo jądro może taki kod wykorzystywać w dowolny sposób wołając funkcje zarejestrowane przez moduł jądra pod wyspecyfikowanym interfejsem.

Moduł jądra może również utworzyć **pseudourządzenia**. Są to pliki logicznie będące urządzeniem w systemie, ale nie odpowiadające żadnemu urządzeniu fizycznemu. Takie urządzenia w systemie GNU/Linux znajdują się zwyczajowo w katalogu /dev. Są to takie urządzenia jak: `urandom`, `tty0`, `zero` czy `null`.

4.2. Struktura kodu sterownika

Wywołania procedur zdefiniowanych w sterowniku wynikają z różnych zdarzeń. Procedury, których wywołanie wynika z jakiegoś żądania systemu operacyjnego (np. wywołania systemowego) względem sterownika, nazywamy **górną połówką (ang. top half)** [1]. Wykonują się one synchronicznie w kontekście wątku jądra oczekującego na obsługę żądania IO i mogą **spać**, czyli dobrowolnie oddać sterowanie. Najczęściej jest to konsekwencją próby zajęcia zajętej już blokady, bądź czekania na zmiennej warunkowej.

Drugi zestaw procedur zwanych **dolną połówką (ang. bottom half)** to pro-

[1] Posługujemy się tutaj nomenklaturą FreeBSD. W przypadku jądra Linux nazewnictwo jest odwrotne.

cedury obsługi przerwań (ang. Interrupt Service Routines - ISR) pochodzących od urządzeń. Z natury są asynchroniczne, stąd nie mogą zależeć od stanu konkretnego procesu. Przyjmijmy, że kod dolnej połówki wykonuje w kontekście przerwanego wątku, oraz z wyłączonymi przerwaniami[2].

Z tego powodu dolną połówkę należy jak najszybciej opuścić, aby nie dopuścić do niepożądanych zdarzeń. Gdyby wykonanie dolnej połówki trwało zbyt długo, obsługa przerwań zagłodziłaby inne wątki. W przypadku jeszcze dłuższego wykonania mogłyby pojawić się efekty takie jak np. gubienie przerwań, które z reguły całkowicie uniemożliwiają stabilną pracę systemu. Tym bardziej kod dolnej połówki nie może spać, zatem jedynym dozwolonym środkiem synchronizacji w takim wypadku będzie blokada wirująca.

Kolejnym zestawem funkcji są procedury odpowiadające za inicjalizację i konfigurację urządzenia, oraz reagowanie na akcje takie jak, wyłączenie, uśpienie, czy wybudzenie urządzenia. W systemie FreeBSD są one używane przede wszystkim przez podsystem NewBus podczas procesu zwanego **autokonfiguracją**.

4.3. Drzewo urządzeń

W dużej części systemów operacyjnych programowa reprezentacja urządzeń wynika z ich organizacji w sprzęcie - przypomina drzewo.

W systemie GNU/Linux polecenie *lshw* pozwala wyświetlić owe drzewo. Prezentujemy jednak przykład drzewa urządzeń w systemie FreeBSD ze względu na jego wykorzystanie w dalszych rozdziałach.

Listing 4..1: Przykładowe drzewo urządzeń w systemie FreeBSD [70]

```
1  root0
2    description: System root bus
3    devclass: root, drivers: nexus
4    children: nexus0
5
6  nexus0
7      devclass: nexus, drivers: acpi, legacy, npx
8      children: np0, legacy0
9
10   legacy0
11       description: legacy system
12       devclass: legacy, drivers, eisa, isa, pcib
13       children: eisa0, pcib0
14
15     pcib0 /* southbridge */
16         description: Intel 82443BX (440 BX) host to PCI bridge
```

[2] Jest to uproszczenie. Systemy operacyjne stosują bardzo różne mechanizmy do obsługi przerwań. Przypadek FreeBSD zostanie opisany w rozdziale 5.4.

```
17          devclass: pcib, drivers: pci
18          children: pci0
19      pcib1
20          description: PCI-PCI bridge
21          devclass: pcib, drivers: pci
22          children: pci1
23
24        pci0
25            description: PCI bus
26            devclass: pci, drivers: ahc, eisab, atapci, xl ...
27            children: agp0, pcib1, isab0, atapci0, ahc0, xl0
28
29          atapci0
30              description: Intel 82371AB PIIX4 IDE controller
31              devclass: atapci, drivers: atadisk, atapicd
32              children: atadisk0, atadisk1
33              class: mass storage, subclass: ATA
34              I/O ports: 0xffa0-0xffaf
35
36            atadisk0
37                devclass: atadisk, drivers: none
38                interrupt request lines: 0xe
39                I/O ports: 0x1f0-0x1f7, 0x3f6
40            atadisk1
41                devclass: atadisk, drivers: none
42                interrupt request lines: 0xf
43                I/O ports: 0x170-0x177, 0376
```

W korzeniu drzewa zazwyczaj znajduje się logiczne urządzenie nazywane rootdev bądź nexus. Wszystkie inne urządzenia są jego potomkami. W liściach często znajdują się fizyczne urządzenia, a na węzłach reprezentacja mostków, magistral i kontrolerów.

Widoczne są również zarezerwowane zasoby takie jak zakresy portów wejścia-wyjścia, czy przerwania.

4.4. Urządzenia znakowe i blokowe

Różne typy urządzeń inaczej obsługują transfer danych z, i do urządzenia. Urządzenia pozwalające na swobodny dostęp (ang. random access) do bloków danych stałej wielkości nazywamy **urządzeniami blokowymi**. Są to na przykład dyski twarde. **Urządzenia znakowe** charakteryzują się strumieniowym przesyłaniem danych i nie pozwalają na swobodny dostęp. Tego typu urządzeniem jest na przykład karta dźwiękowa.

Naturalnie sterowniki różnych typów urządzeń będą się różnić. Można wtedy takie sterowniki nazywać odpowiednio sterownikami blokowymi bądź znakowymi.

Jednakże nazwy te nie zawsze są ściśle związane z typem obsługiwanego urządzenia, a raczej z interfejsem wystawianym przez sterownik do systemu operacyjnego i użytkownika, stąd pojawiające się nazwy **interfejs znakowy sterownika**, oraz **interfejs blokowy sterownika**.

Sterowniki blokowe przyjmują zlecenia zapisu bądź odczytu bloków. Zlecenia te, oraz dane są buforowane i keszowane przez jądro systemu operacyjnego. Proces ten wymaga bardzo dużej i skomplikowanej logiki w systemie.

Interakcja ze sterownikami znakowymi korzysta z tego samego interfejsu co operacje na zwykłych plikach. W przypadku systemów POSIX można czytać i pisać za pomocą procedur `read()` i `write()`. Operacje te nie wymagają buforowania ani keszowania, a ich obsługa jest znacznie prostsza.

W systemie operacyjnym FreeBSD blokowy interfejs do sterowników jest wycofywany z użycia oraz wysoce niezalecany w żadnym przypadku. Głównym powodem jest niedeterminizm wykonania zlecanych operacji. Mianowicie, ze względu na buforowanie i keszowanie zlecanych operacji zapisów, rzeczywiste zapisy mogły być wykonane w innej kolejności niż zostały zlecone. Uniemożliwia to przewidzenie stanu dysku w dowolnym momencie czasu, co dalej czyni prawie niemożliwym analizę i naprawę zniszczonych struktur danych na dysku (np. systemów plików).

Dodatkowo rzeczywiste zapisy nie są wykonywane od razu. Powoduje to, że w przypadku błędu, system operacyjny nie jest w stanie stwierdzić która ze zleconych operacji zawiniła.

We FreeBSD interfejs znakowy dla urządzeń dyskowych nazywany jest **surowym interfejsem (ang. raw-device interface)**.

4.5. Start systemu

Przy starcie komputera, wstępna konfiguracja urządzeń dokonywana jest przez oprogramowanie zapisane w nieulotnej pamięci na płycie głównej, czyli tzw. **firmware**. Jest ono zależne od płyty głównej i architektury procesora. Przykładami może być BIOS [52] i wypierające go UEFI [53] w architekturze x86. Oprócz niezbędnej konfiguracji sprzętu, ich głównym zadaniem jest załadowanie programu zwanego **bootloaderem**, który będzie pełnił dalsze kroki w celu wystartowania systemu.

W przypadku starszych urządzeń i magistral nie wspierających mechanizmu Plug and Play [3.2.], urządzenia miały przypisane stałe numery przerwań, oraz adresy portów wejścia-wyjścia. Z czasem zaczęto wprowadzać urządzenia wyposażone w przełączniki DIP bądź zworki. Za ich pomocą użytkownik mógł manipulować przypisanymi do urządzenia numerami przerwań, oraz adresami portów wejścia-wyjścia, szczególnie w przypadku ewentualnych konfliktów.

System operacyjny rzadko wiernie polega na wstępnej konfiguracji przez firm-

ware (bądź bootloader), dlatego często sam dokonuje powtórnej i dokładniejszej konfiguracji. W przypadku starszych standardów należy opisać zasoby podłączonych urządzeń w odpowiednim pliku konfiguracyjnym.

Inicjalizacja większości systemów operacyjnych podzielona jest na etapy. W każdym z nich wykonują się ściśle zdefiniowane operacje. Jednym z etapów, może być np. inicjalizacja urządzeń i ich sterowników.

Rozdział 5.

Wprowadzenie do architektury jądra systemu FreeBSD

Wszelki cytowany w następnych rozdziałach kod pochodzi z systemu FreeBSD w wersji 11. Przeglądarki kodu dostępne są pod adresami `fxr.watson.org` i `bxr.su`. Zdecydowana większość przytaczanego kodu pochodzi z pliku `subr_bus.c` [59]. Zachęcamy do samodzielnej analizy kodu źródłowego, ponieważ kod opisany w pracy jest tylko małą częścią całości.

5.1. Podstawowe struktury danych

System FreeBSD posiada wygodną implementację podstawowych spośród dynamicznych struktur danych. Najpowszechniej używanymi strukturami danych w jądrze FreeBSD są listy jedno i dwukierunkowe. Ich implementacja jest tzw. biblioteką nagłówkową (ang. header-only library), czyli zawiera się w jednym pliku nagłówkowym, w tym przypadku `queue.h` [56].

Plik ten nie definiuje żadnych funkcji, ani nie deklaruje żadnych zmiennych. Cała funkcjonalność opiera się na makrodefinicjach. Powoduje to, że plik ten może być z powodzeniem zaimportowany do dowolnego innego pliku źródłowego, a nieużywane funkcjonalności zostaną pominięte podczas kompilacji.

Plik `queue.h` zawiera makrodefinicje obsługujące 4 struktury danych:

- `SLIST` - listę jednokierunkową
- `STAILQ` - listę jednokierunkową ze wskaźnikiem na koniec listy
- `LIST` - listę dwukierunkową
- `TAILQ` - listę dwukierunkową ze wskaźnikiem na koniec listy

Implementacja powyższych struktur nie przypomina jednak kontenerów w standardowym rozumieniu. Jest to intruzyjna struktura danych [8].

Na przykładzie `TAILQ` przybliżymy koncepcje list intruzyjnych. W strukturach intruzyjnych, dane odpowiedzialne za zarządzanie strukturą są rozproszone pomiędzy elementy należące do owej struktury. Czyli struktura będąca na liście musi definiować jako własne pole strukturę, która będzie odpowiadać, za przechowywanie informacji o następniku i poprzedniku.

Strukturę tę definiuje makro `TAILQ_ENTRY`:

```
1 #define TAILQ_ENTRY(type)    \
2   struct {                   \
3     struct type *tqe_next;   \
4     struct type **tqe_prev;  \
5     (...)                    \
6   }
```

Wskaźnik `tqe_next` wskazuje na następną strukturę na liście, a wskaźnik `tqe_prev` pokazuje na zmienną `tqe_next` poprzedniej struktury nawleczonej na listę.

Przy używaniu listy należy zdefiniować jej głowę. Służy do tego makrodefinicja `TAILQ_HEAD` zdefiniowana następująco:

```
1 #define TAILQ_HEAD(name, type)  \
2   struct name {                 \
3     struct type *tqh_first;     \
4     struct type **tqh_last;     \
5     (...)                       \
6   }
```

Głowa listy może zostać zainicjalizowana w sposób statyczny (podczas kompilacji) przez makro `TAILQ_HEAD_INITIALIZER`, bądź dynamicznie przez makro `TAILQ_INIT`. Obydwie metody ustawiają pole `tqh_first` na `NULL`, i pole `tqh_last` na adres pola `tqh_first`.

Listing 5..1: Wstawianie do listy

```
1 TAILQ_INSERT_HEAD(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME)
2 TAILQ_INSERT_TAIL(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME)
```

Makra z listingu 5..1 służą do dodawania elementu na koniec, bądź początek listy.

Listing 5..2: Wstawianie do listy

```
1 TAILQ_INSERT_BEFORE(TYPE *listelm, TYPE *elm, TAILQ_ENTRY NAME)
2 TAILQ_INSERT_AFTER(TAILQ_HEAD *head, TYPE *listelm, TYPE *elm,
3   TAILQ_ENTRY NAME)
```

Z kolei makra z listingu 5..2 pozwalają wstawiać nowy element za, bądź przed innym elementem w liście.

Makro `TAILQ_FOREACH(TYPE *var, TAILQ_HEAD *head, TAILQ_ENTRY NAME)` pozwala na przejście po całej liście w pętli podstawiając za `var` kolejne adresy struktur nawleczonych na listę.

Makro `TAILQ_REMOVE(TAILQ_HEAD *head, TYPE *elm, TAILQ_ENTRY NAME)` usuwa element z listy.

Listing 5..3: Przykład użycia TAILQ [85]

```
1  TAILQ_HEAD(tailhead, entry) head = TAILQ_HEAD_INITIALIZER(head);
2
3  struct tailhead *headp;            // Głowa listy.
4  struct entry {
5    int number;
6    TAILQ_ENTRY(entry) entries;      // Element na liście.
7  } *n1, *n2, *n3, *np;
8
9  TAILQ_INIT(&head);                 // Inicjalizacja listy.
10
11 n1 = malloc(sizeof(struct entry)); // Wstawienie na początek.
12 TAILQ_INSERT_HEAD(&head, n1, entries);
13
14 n1 = malloc(sizeof(struct entry)); // Wstawienie na koniec.
15 TAILQ_INSERT_TAIL(&head, n1, entries);
16
17 n2 = malloc(sizeof(struct entry)); // Wstawienie za element.
18 TAILQ_INSERT_AFTER(&head, n1, n2, entries);
19
20 n3 = malloc(sizeof(struct entry)); // Wstawienie przed element.
21 TAILQ_INSERT_BEFORE(n2, n3, entries);
22
23 TAILQ_REMOVE(&head, n2, entries);  // Usuwanie.
24 free(n2);
25
26 /* Przejście listy w przód. */
27 int i = 0;
28 TAILQ_FOREACH(np, &head, entries)
29   np->number = i++;
30
31 /* Przejście listy w tył. */
32 TAILQ_FOREACH_REVERSE(np, &head, tailhead, entries)
33   np->number = 1337;
```

W podobny sposób zdefiniowane są drzewiaste struktury danych w pliku `tree.h` [57]. Znajduje się tam implementacja drzew splay i drzew czerwono-czarnych.

5.2. Moduły jądra

Każdy moduł jądra jest reprezentowany przez strukturę `module_t` [71]. Struktura ta zawiera między innymi nazwę modułu, jego unikalny numer, wskaźnik na **uchwyt zdarzeń modułu** i argument przekazywany do uchwytu.

Każdy moduł musi implementować co najmniej **uchwyt zdarzeń modułu**, oraz wywoływać makro `DECLARE_MODULE`.

Uchwyt zdarzeń modułu jest funkcją o sygnaturze:

```
1 typedef int (*modeventhand_t)(module_t mod,
2   int /* modeventtype_t */ what, void *arg);
```

Listing 5..4: Zdarzenia związane z modułami

```
1 typedef enum modeventtype {
2   MOD_LOAD,
3   MOD_UNLOAD,
4   MOD_SHUTDOWN,
5   MOD_QUIESCE
6 } modeventtype_t;
```

Kiedy moduł jest ładowany do jądra, wywoływany jest uchwyt tego modułu z argumentem `what` równym `MOD_LOAD`. Kod modułu jądra wykonuje w tym momencie operacje związane z inicjalizacją. Np. tworzy pseudourządzenie w ścieżce /dev.

W przypadku odczepienia modułu (kldunload), uchwyt modułu najpierw jest wywoływany z argumentem `what` ustawionym na `MOD_QUIESCE`, aby zweryfikować, czy moduł może być w danej chwili odczepiony. Jeśli tak, to dopiero wtedy uchwyt wywoływany jest z argumentem `MOD_UNLOAD`. Możliwe jest również siłowe odczepienie modułu `kldunload -f` bez powyższego sprawdzenia, jednak może to powodować różne nieporządane skutki, takie jak np. wycieki pamięci. Nie chcemy odczepiać modułu, który aktualnie przetwarza jakieś dane, np. trzyma wskaźniki na pamięć, która musi zostać zwolniona. Moduł przy odczepieniu musi zwolnić wszystkie zajęte przy inicjalizacji zasoby, oraz wycofać zmiany poczynione w systemie np. usunąć utworzone uprzednio pseudourządzenie.

Kiedy system się wyłącza argument `what` jest równy `MOD_SHUTDOWN`. Moduły mogą przed wyłączeniem systemu chcieć uspójnić jakiś stan, np. poprzez zapisanie jakichś informacji na dysk twardy.

Kiedy moduł nie obsługuje danego zdarzenia, uchwyt powinien zwrócić `EOPNOTSUPP`.

Makrodefinicja `DECLARE_MODULE` rejestruje moduł w systemie.

```
1 DECLARE_MODULE(name, moduledata_t data, sub, order);
```

Argument `name` identyfikuje moduł w systemie. Pod argumentem `data` powinna

znajdować się opisująca moduł, wypełniona struktura `moduledata_t` wyglądająca następująco:

```
typedef struct moduletada {
  const char      *name;
  modeventhand_t  evhand;
  void            *priv;
} moduledata_t;
```

`name` jest nazwą modułu, `evhand` jest wskaźnikiem na uchwyt modułu, a `priv` wskazuje na prywatne dane modułu.

Argument `sub` makra `DECLARE_MODULE` specyfikuje w jakim **etapie inicjalizacji systemu** powinien być wywołany uchwyt modułu związany ze zdarzeniem `MOD_LOAD` inicjalizujący dany moduł.

Etapy inicjalizacji służą podsystemowi `sysinit` [74], do stopniowego ładowania funkcjonalności jądra w dobrze ustalony i kontrolowany sposób. W tej pracy interesować nas będą tylko etapy `SI_SUB_DRIVERS` oraz `SI_SUB_CONFIGURE` spośród wszystkich kilkudziesięciu.

Listing 5..5: Etapy inicjalizacji systemu

```
enum sysinit_sub_id {
  ...
  SI_SUB_DRIVERS   = 0x3100000, /* inicjalizacja sterowników */
  SI_SUB_CONFIGURE = 0x3800000, /* konfiguracja urządzeń */
  SI_SUB_VFS       = 0x4000000, /* konfiguracja wirtualnego
                                   systemu plików */
  ...
}
```

Argument `order` makra `DECLARE_MODULE` wyznacza kolejność inicjalizacji w obrębie danego etapu. Ostateczna kolejność jest reprezentowana przez liczbę całkowitą, na przykład: `SI_SUB_CONFIGURE + SI_ORDER_FOOURTH`.

```
enum sysinit_elem_order {
  SI_ORDER_FIRST
  SI_ORDER_SECOND
  SI_ORDER_THIRD
  SI_ORDER_FOOURTH
  SI_ORDER_MIDDLE
  SI_ORDER_ANY
}
```

W przypadku modułów jądra będących sterownikami, argument ten najczęściej będzie przyjmował wartość `SI_ORDER_MIDDLE`.

Możliwe jest również wyspecyfikowanie wersji modułu za pomocą makra `MODULE_VERSION(name, int version);`.

Inne moduły mogą wtedy wymagać jako zależności innego modułu w danej wersji wywołując makro:

```
1 MODULE_DEPEND(name, moddepend, int minversion, int prefversion,
2               int maxversion);
```

Informuje ono jądro podczas inicjalizacji systemu, że moduł `name` zależy od modułu `moddepend`, w wersjach od `minversion` do `maxversion`, z preferowaną wersją `prefversion`. Jądro jest w stanie przy starcie rozwiązywać tego typu zależności. W przypadku ładowania modułów ręcznie podczas pracy systemu makro to nie jest brane pod uwagę.

Istnieją również makra rejestrujące moduły implementujące pewne ustalone funkcjonalności jak na przykład pseudourządzenia (makro `DEV_MODULE`), wywołania systemowe (makro `SYSCALL_MODULE`), czy sterowniki (makro `DRIVER_MODULE`).

Przykładowy kod modułu jądra może wyglądać następująco:

```
1  #include <sys/param.h>
2  #include <sys/kernel.h>
3  #include <sys/module.h>
4
5  static int
6  hello_handler(module_t mod, int /*modeventtype_t*/ what,
7                void *arg)
8  {
9    switch(what){
10     case MOD_LOAD:
11       uprintf("It's showtime.\n");
12       return 0;
13     case MOD_UNLOAD:
14       uprintf("You have been terminated.\n");
15       return 0;
16     default:
17       return EOPNOTSUPP;
18   }
19 }
20
21 static moduledata_t mod_data= {
22   "hello",
23   hello_handler,
24   NULL
25 };
26
27 MODULE_VERSION(hello, 1);
28 MODULE_DEPEND(hello, other_mod, 1, 3, 4);
29 DECLARE_MODULE(foo, mod_data, SI_SUB_EXEC, SI_ORDER_ANY);
```

Więcej przykładów kodu modułów jądra znaleźć można w witrynie *bxr.su* pod adresem `http://bxr.su/FreeBSD/share/examples/kld/`.

5.3. kobj, czyli obiektowość w C

kobj, czyli podsystem systemu FreeBSD oferujący możliwość programowania zorientowanego obiektowo w jądrze [83][44][45]. System opiera się na koncepcie interfejsów, które opisują metody (oraz czasem definiują domyślną ich implementację). Klasy są listą funkcji implementujących metody interfejsów. Obiekty są pojedynczymi instancjami klas.

Nowe interfejsy i klasy mogą być tworzone dynamicznie, podczas pracy systemu.

Aby skorzystać z podsystemu kobj należy najpierw utworzyć interfejs. Są to pliki o sufiksie .m przetwarzane przez skrypt *src/sys/kern/makeobjops.pl* generujący odpowiednie nagłówki, struktury, kod metod i procedury ich wyszukiwania, w postaci plików .c i .h. Skrypt ten szuka następujących słów kluczowych: `#include`, `INTERFACE`, `CODE`, `METHOD`, `STATICMETHOD`, and `DEFAULT`.

Wyrażenie następujące po słowie kluczowym `#include` jest kopiowane do pliku wyjściowego. Po słowie kluczowym `INTERFACE` występują nazwa definiowanego interfejsu. W pliku wyjściowym nazwa każdej metody będzie miała nazwę interfejsu w prefiksie: `[nazwa interfejsu]_[nazwa metody]`. Słowo `METHOD` definiuje metodę w następujący sposób: `METHOD [zwracany typ] [nazwa metody] [obiekt; [pozostałe argumenty]] DEFAULT [domyślna funkcja];` Na przykład:

```
1 INTERFACE foo;
2
3 METHOD int bar {
4   kobj_t baz;
5   int qux;
6 } DEFAULT foo_ugly_hack;
```

Definiujemy typ zwracany, nazwę metody, kolejne jej argumenty i domyślnie wywoływaną metodę w przypadku braku implementacji metody definiowanej. Pierwszym argumentem metody zawsze musi być obiekt podsystemu kobj.

Słowo kluczowe `CODE` powoduje skopiowanie kodu po nim występującego do pliku wyjściowego. Mechanizm ten może zostać użyty np. do generowania domyślnych implementacji metod.

```
1   CODE {
2     static int foo_ugly_hack(kobj_t baz, int qux){
3       return printf("emacsem przez sendmail");
4     }
5   }
```

`STATICMETHOD` jest wykorzystywane w sposób analogiczny do `METHOD`, jednakże nie wymaga się aby obiekt `kobj_t` był pierwszym argumentem. Metody te wywoływane są w kontekście całej klasy a nie pojedynczego obiektu.

Skrypt *makeobjobs.pl* w kontekście powyższych przykładów generuje następujący kod:

Listing 5..6: Część kodu z wygenerowanego pliku nagłówkowego

```
1 extern struct kobjop_desc foo_bar_desc;
2 typedef int foo_bar_t(kobj_t baz, int qux);
3 static __inline void FOO_BAR(kobj_t baz, int qux)
4 {
5         kobjop_t _m;
6         KOBJOPLOOKUP(((kobj_t)baz)->ops,foo_bar);
7         ((foo_bar_t *) _m)(baz, qux);
8 }
```

Wywołanie metody wykonuje się poprzez makro `FOO_BAR(baz, qux)`. Makro `KOBJOPLOOKUP` znajduje implementację metody `bar` w **tablicy metod obiektu** `baz` i przypisuje jej adres do zmiennej `_m`. Metody są identyfikowane unikalnie poprzez strukturę `kobjop_desc`. Jeśli odpowiednia metoda nie zostanie odnaleziona, wykonana zostanie funkcja będąca domyślną implementacją.

Aby utworzyć obiekt implementujący dany interfejs należy zdefiniować jego **tablicę metod**.

```
1  static int my_bar(kobj_t baz, int qux)
2  {
3          return 0x6834783072;
4  }
5
6  kobj_method_t foo_methods = {
7          /* foo interface */
8          KOBJMETHOD(bar, my_bar),
9          { 0, 0 }
10 };
```

Następnie trzeba zdefiniować używaną klasę: `DEFINE_CLASS(foo, foo_methods, sizeof(struct kobj));` Aby używać obiektów danej klasy musi ona zostać *skompilowana*. Oznacza to przetworzenie definicji danej klasy, aby wyszukiwanie metod było szybkie. System kobj w większości przypadków robi to leniwie w momencie pierwszego użycia klasy. Jednak aby wymusić ten proces wcześniej, użytkownik może użyć procedury:

```
1 void kobj_class_compile(kobj_class_t cls);
```

Teraz może zostać utworzony obiekt:

```
1 kobj_t baz = kobj_create(foo_class, M_FOO, M_WAITOK);
```

a jego metody zawołane używając wygenerowanych makr, np: `FOO_BAR(baz, 2);`. Obiekt powinien zostać zwolniony za pomocą funkcji `kobj_delete`. W powyższym przykładzie pamięć którą zajmuje obiekt jest przydzielana dynamicznie (stąd flagi `M_FOO`, oraz `M_WAITOK` [72]). Kiedy klasa nie jest już używana również powinna zostać zwolniona używając funkcji `kobj_class_free`.

Jeśli użytkownik sam chce zarządzać pamięcią powinien użyć procedury:

```
1 void kobj_init(kobj_t obj, kobj_class_t cls);
```

która zwiąże dany obiekt z daną klasą.

5.4. Synchronizacja w procedurach obsługi przerwań

W systemie operacyjnym FreeBSD definiuje się dwa rodzaje uśpienia wątku. **Uśpienie ograniczone (ang. bounded sleep)**, czasem nazywane blokowaniem i **uśpienie nieograniczone (ang. unbounded sleep)**, nazywane również spaniem. W kontekście gdzie rozróżnienie typów uśpienia nie ma znaczenia, spaniem nazywa się dobrowolne oddanie sterowania przez wątek [86].

W przypadku uśpienia ograniczonego, jedynym zasobem potrzebnym do wznowienia wykonania wątku jest czas procesora, który musi zostać poświęcony na wykonanie innego wątku trzymającego współdzielony zasób podlegający wzajemnemu wykluczeniu, który zablokowany wątek chce przejąć. W szczególności wykonanie ciągu wątków zablokowanych na danym zasobie zawsze powinno postępować, ponieważ zawsze któryś z wątków jest aktualnie wykonywany na procesorze. Aby uzyskać taki efekt wymaga się, aby żaden wątek uśpiony ograniczenie, nie czekał pośrednio, lub bezpośrednio na zasób zajęty przez wątek uśpiony nieograniczenie. Dodatkowo żeby uniknąć sytuacji **odwrócenia priorytetów**, wątek uśpiony ograniczenie na danej blokadzie, podnosi priorytet obecnemu właścicielowi blokady do swojego priorytetu, jeśli ten jest większy. Mechanizm ten nazywa się **dziedziczeniem priorytetów** i zaimplementowany jest z użyciem struktury danych zwanej **turnstile**.

Inaczej sytuacja wygląda w przypadku uśpienia nieograniczonego. W tym przypadku wątek czeka na zewnętrzne zdarzenie, które jest potencjalnie nieograniczone czasowo. Np. oczekiwanie na wciśnięcie klawisza przez użytkownika.

Obsługa przerwania sprzętowego we FreeBSD składa się z procedury typu **filter** [88], części wykonywanej w wątku przerwania - **ithread** [87][55], bądź obu naraz [54]. Procedury typu filter wykonują się w dolnej połówce i mogą zlecić część pracy do wykonania w wątku przerwania, który wykonuje się niemal jak górna połówka, jednakże z pewnymi obostrzeniami.

Procedura obsługi przerwania typu filter wykonuje się w kontekście przerwanego wątku, z wyłączonym wywłaszczaniem [89], oraz zależnie od architektury z wyłączoną częścią, bądź wszystkimi przerwaniami [60][61]. Z powyższych powodów taka

Tablica 5.1: Środki synchronizacji w ISR w zależności od typu procedury

	spin mtx	mutex/rw	rmlock	sleep rm	sx/lk	sleep
interrupt filter	ok	no	no	no	no	no
interrupt thread	ok	ok	ok	no	no	no
bounded/unbounded	-	B	B	U	U	U

procedura nie może dobrowolnie zrzec się procesora, czyli nie może w żaden sposób spać. Stąd jedynym dozwolonym środkiem synchronizacji jest blokada wirująca.

Wątki przerwań **ithread** agregują procedury obsługi przerwań. Znaczy to, iż w takim wątku może zostać zarejestrowane wiele ISR. Procedury w ithread mają własny kontekst i mogą się blokować, ale mimo wszystko nie powinny wykonywać się zbyt długo, ponieważ mają najwyższy priorytet i mogłyby zagłodzić inne wątki. Nie mogą też spać nieograniczenie. Plusem jest za to możliwość wykorzystywania większej ilości środków synchronizacji. Jednak z zastrzeżeniem, że nie mogą one powodować nieograniczonego uśpienia. Są to na przykład muteksy czy blokady typu rmlock (ang. read-mostly lock) [86].

Programista pisząc kod odpowiadający za obsługę przerwania musi niezwykle uważać, jakich środków synchronizacji używa, ponieważ użycie niewłaściwego (niezgodnego z polityką zakładania blokad danego systemu operacyjnego), zazwyczaj kończy się bardzo trudnymi do zdiagnozowania problemami takimi jak zakleszczenia, głodzenie i inne.

W przypadku FreeBSD samo użycie mechanizmu przerwań w kodzie jest dość proste. Sterownik, aby zarejestrować ISR woła procedurę [88]:

```
1 int bus_setup_intr(device_t dev, struct resource *r, int flags,
2   driver_filter_t filter, driver_intr_t ithread, void *arg,
3   void **cookiep);
```

Rozdział 6.

Urządzenia we FreeBSD

6.1. Struktury device i driver oraz urządzenie root

W infrastrukturze odpowiedzialnej za urządzenia i sterowniki, w oczywisty sposób centralną rolę stanowią struktury danych reprezentujące owe byty [73][75]. Przedstawimy je już teraz, a w następnych rozdziałach równolegle z tłumaczeniem całej infrastruktury, będziemy wyjaśniać dlaczego owe byty są reprezentowane właśnie w taki sposób i jaką rolę pełnią pola struktur.

Struktura `driver_t` jest zdefiniowana w następujący sposób:

```
1 typedef struct kobj_class   driver_t
```

a więc jest klasą w podsystemie `kobj`. Urządzenia implementują metody z interfejsu tej klasy. System operacyjny woła te metody właśnie za pomocą abstrakcyjnego interfejsu, który dokładniej zostanie opisany w dalszej części pracy.

Listing 6..1: Reprezentacja urządzenia

```
1  typedef struct device {
2
3    KOBJ_FIELDS; /* device_t rozszerza kobj_t */
4
5    /* Hierarchia urządzeń. */
6    TAILQ_ENTRY(device) link;    /* element na liście urządzeń
7                                    rodzica */
8    TAILQ_ENTRY(device) devlink; /* element na globalnej liście
9                                    urządzeń */
10   device_t        parent;      /* rodzic tego urządzenia  */
11   device_list_t   children;    /* lista urządzeń będących
12                                   dziećmi tego urządzenia */
13   driver_t        *driver;     /* skojarzony sterownik */
14   devclass_t      devclass;    /* klasa urządzenia */
15   int             unit;        /* numer jednostkowy urządzenia
16   char*           nameunit;    /* name ++ unit  */
```

```
17   char*           desc;         /* zależny od sterownika opis
18                                    urządzenia  */
19   int             busy;      /* device_busy()[1] */
20   device_state_t  state;     /* stan urządzenia  */
21   uint32_t        devflags;  /* flagi na użytek podsystemu do
22                                 zarządzania urządzeniami */
23   u_int           flags;     /* prywatne flagi urządzenia */
24   u_int           order;     /* device_add_child_ordered()[2] */
25   void            *ivars;    /* prywatne dane szyny */
26   void            *softc;    /* prywatne dane
27                                 urządzenia/sterownika */
28   ...
29 } device_t;
```

Struktura `device_t` jest abstrakcyjną reprezentacją urządzenia sprzętowego takiego jak na przykład karta rozszerzeń, magistrala, mostek, czy kontroler. Rozszerza ona strukturę `kobj_t` i jako obiekt podsystemu kobj jest związywana z odpowiednią klasą sterownika `driver_t`.

Nomenklatura FreeBSD mówi, że **szyną** jest każde urządzenie (`device_t`) posiadające dzieci. Może to być zarówno urządzenie reprezentujące magistrale jak i mostek. **Urządzeniem** w zależności od kontekstu nazywane jest fizyczne urządzenie sprzętowe, bądź reprezentacja takowego, czyli konkretna struktura `device_t`. Ponadto urządzenia z nazwą kończącą się na b są mostkami (np. pcib1), i do nich podłączone są urządzenia będące szynami (np. pci0) 4..1.

System operacyjny w statyczny sposób definiuje jedno urządzenie `root`, które jest wyjątkowe, ponieważ znajduje się w korzeniu drzewiastej hierarchii urządzeń. Jest ono ładowane jako moduł jądra za pomocą standardowego makra rejestrującego moduły w następujący sposób: `DECLARE_MODULE(rootbus, root_bus_mod, SI_SUB_DRIVERS, SI_ORDER_FIRST);`. Uchwyt zdarzeń modułu urządzenia `root` przy załadowaniu modułu, tworzy strukturę `device_t` reprezentującą owe urządzenie, po czym statycznie przypisuje sterownik do tego urządzenia. Sterownik ten implementuje metody z interfejsów `bus` i `device` [6.5.], jednak nie implementuje metod które są wymagane do procesu autokonfiguracji, ponieważ nie bierze w nim takiego udziału jak inne sterowniki.

[1] DEVICE_GET_STATE(9) - https://www.freebsd.org/cgi/man.cgi?query=device_busy&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports

[2] DEVICE_ADD_CHILD(9) - https://www.freebsd.org/cgi/man.cgi?query=device_add_child_ordered&manpath=FreeBSD+11.2-RELEASE+and+Ports

Listing 6..2: Uchwyt zdarzeń modułu urządzenia root.

```
devclass_t root_devclass;
device_t root_bus;

int root_bus_module_handler(module_t mod, int what, void* arg)
{
  switch (what) {
  case MOD_LOAD:
    TAILQ_INIT(&bus_data_devices);
    kobj_class_compile((kobj_class_t) &root_driver);
    root_bus = make_device(NULL, "root", 0);
    root_bus->desc = "System root bus";
    kobj_init((kobj_t) root_bus, (kobj_class_t) &root_driver);
    root_bus->driver = &root_driver;
    root_bus->state = DS_ATTACHED;
    root_devclass = devclass_find_internal("root", NULL, FALSE);
    devinit();
    return (0);

  case MOD_SHUTDOWN:
    device_shutdown(root_bus);
    return (0);

  default:
    return (EOPNOTSUPP);
  }

  return (0);
}
```

Przy ładowaniu modułu uchwyt inicjuje globalną listę urządzeń `bus_data_devices`, oraz *kompiluje* klasę `root_driver`. Następnie tworzy urządzenie o nazwie `root` z **numerem jednostkowym (ang. unit number)** 0, które nie posiada rodzica jako urządzenia nadrzędnego i ustawia jego krótki opis.

Numery jednostkowe pozwalają odróżnić urządzenia o tej samej nazwie, co pozwala za pomocą nazwy i numeru jednostkowego jednoznacznie zidentyfikować urządzenie w systemie (pola `unit` i `nameunit` struktury `device_t`). Ponadto są one indeksem w tablicy wskaźników na urządzenia w odpowiedniej strukturze `devclass`. Zazwyczaj są to różne instancje tego samego urządzenia, np. `atadisk0` i `atadisk1` będące dwoma różnymi urządzeniami dysków twardych protokołu ATA.

Następnie inicjalizowany jest obiekt `root_bus` w klasie `root_driver`. Urządzenie jako obiekt jest wiązane ze sterownikiem jako klasą w podsystemie kobj. Dodatkowo wskaźnik do sterownika jest zapisywany w urządzeniu, a stan urządzenia ustawiany na `DS_ATTACHED` [92].

Listing 6..3: Typ opisujący stan urządzenia.

```
typedef enum device_state {
  DS_NOTPRESENT = 10,     /* niewykonane bądź niepomyślne
                             próbkowanie */
  DS_ALIVE = 20,          /* pomyślne próbkowanie */
  DS_ATTACHING = 25,      /* wywoływana metoda attach */
  DS_ATTACHED = 30,       /* wywołana metoda attach */
  DS_BUSY = 40            /* urządzenie jest otwarte */
} device_state_t;
```

Następnie ustawiana jest zmienna **root_devclass**, będąca wskaźnikiem o globalnym zasięgu, do której przypisany zostanie adres klasy urządzeń `root` uprzednio utworzonej przy tworzeniu urządzenia przez procedurę `make_device`. Klasy urządzeń zostaną omówione w sekcji dalszej części pracy .

Procedura `devinit` inicjalizuje podsystemy `devctl` [76][77] i `sysctl` [78][79] służące do zarządzania urządzeniami przez użytkownika podczas pracy systemu z poziomu konsoli.

Poniżej struktury używane w uchwycie modułu, oraz makrze rejestrującym moduł.

Listing 6..4: Tablica metod obiektu

```
static kobj_method_t root_methods[] = {
  /* Metody z interfejsu device. */
  KOBJMETHOD(device_shutdown, bus_generic_shutdown),
  KOBJMETHOD(device_suspend,  bus_generic_suspend),
  KOBJMETHOD(device_resume,   root_resume),

  /* Metody z interfejsu bus. */
  KOBJMETHOD(bus_print_child, root_print_child),
  KOBJMETHOD(bus_read_ivar,   bus_generic_read_ivar),
  KOBJMETHOD(bus_write_ivar,  bus_generic_write_ivar),
  KOBJMETHOD(bus_setup_intr,  root_setup_intr),
  KOBJMETHOD(bus_child_present,   root_child_present),
  KOBJMETHOD(bus_get_cpus,    root_get_cpus),

  KOBJMETHOD_END
};
```

Urządzenie `root` nie implementuje własnych metod `device_shutdown`, ani `device_suspend`. Korzysta z generycznych funkcji (`_generic_` w nazwie), które zawierają podstawową logikę jakiej wymaga podsystem sterowników od takiej metody. Generyczne funkcje zazwyczaj są opakowywane przez sterowniki aby uzyskać różnego rodzaju dodatkowe funkcjonalności.

Funkcja `bus_generic_shutdown` powoduje wywołanie metody `device_shutdown` każdego urządzenia które jest dzieckiem urządzenia `root`. Analogicznie działa funkcja `bus_generic_suspend`.

Urządzenie `root` implementuje metodę `device_resume` za pomocą funkcji `root_resume`. Implementacja ta oprócz wywołania generycznej funkcji `bus_generic_resume`, w razie błędu powiadamia w odpowiedni sposób podsystem `devctl` o błędzie.

Listing 6..5: Definicje struktur sterownika i danych modułu

```
1  static driver_t root_driver = {
2    "root",
3    root_methods,
4    1,          /* brak prywatnego kontekstu */
5  };
6
7  static moduledata_t root_bus_mod = {
8    "rootbus",
9    root_bus_module_handler,
10   NULL
11 };
```

Załadowanie modułu urządzenia root i jego utworzenie odbywa się w etapie inicjalizacji [5..5] `SI_SUB_DRIVERS`. Również i w tym etapie ładowane są sterowniki innych urządzeń.

6.2. Klasy urządzeń - devclasses

Każde urządzenie należy do **klasy urządzeń** reprezentowanej przez strukturę `devclass_t` [80]. Klasy przede wszystkim grupują sterowniki obsługujące urządzenia danej klasy. Przy inicjalizacji urządzenia przeszukiwane są wszystkie sterowniki w klasie. Sterownik najlepiej obsługujący dane urządzenie zostaje z nim skojarzony. Np. klasa pci zawiera sterowniki mogące obsługiwać urządzenia podłączane do magistrali pci. Drugim zadaniem devclasses jest zachowanie odwzorowania pomiędzy przyjazną dla użytkownika nazwą urządzenia (np. *pci1*), a odpowiadającą mu strukturą `device_t`. Odwzorowanie to pamiętane jest jako nazwa danej klasy urządzeń (pole `name`), oraz numer jednostkowy danego urządzenia.

Cała hierarchia klas urządzeń tworzona jest dynamicznie, w etapie `SI_SUB_DRIVERS` jak i `SI_SUB_CONFIGURE`. Nowe klasy tworzone są głównie przy ładowaniu modułu sterownika do systemu, oraz przy tworzeniu urządzeń (`device_t`) np. w wyniku procedury `make_device`.

Przypuśćmy, że urządzenie `pci1` wykryje kontroler dysku ATA. Tworzy nowe urządzenie i odpowiadającą mu strukturę `device_t`, po czym dodaje ją do listy

Listing 6..6: Struktura devclass

```
struct devclass {
  TAILQ_ENTRY(devclass) link; /* element na globalnej liście
                                 devclasses */
  devclass_t  parent;     /* rodzic w hierarchii devclasses */
  driver_list_t  drivers;     /* lista sterowników w klasie */
  char  *name;          /* nazwa danej klasy */
  device_t  *devices;   /* tablica urządzeń indeksowana
                           numerami jednostkowymi */
  int  maxunit;    /* ilość elementów w powyższej tablicy */
  int  flags;
  (...)
};
```

swoich dzieci. Sterownik kontrolera dysków ATA - `atapci`, został na etapie `SI_SUB_DRIVERS` dodany do klasy urządzeń `pci`. Stało się to w uchwycie zdarzeń modułu sterownika. Informacja aby zakwalifikować ten sterownik do tej właśnie klasy została przekazana w makrodefinicji `DRIVER_MODULE` przez parametr `busname`.

Gdy urządzenie *pci1* zidentyfikuje już wszystkie swoje dzieci, zaczyna przeprowadzać na nich dalsze etapy inicjalizacji. Należy dobrać im sterowniki. Każde urządzenie spośród wszystkich sterowników w klasie rodzica wybiera ten który pasuje najlepiej (względem wartości zwracanej podczas próbkowania sterowników; omówione w dalszej części pracy). Nazwa urządzenia najczęściej ustawiana jest przez wybrany sterownik. Np. przy dobraniu sterownika *atapci* urządzeniu nadana zostanie nazwa *atapci0*. Kiedy sterownik został wybrany, urządzenie jest rejestrowane w klasie urządzeń o tej samej nazwie co nazwa urządzenia.

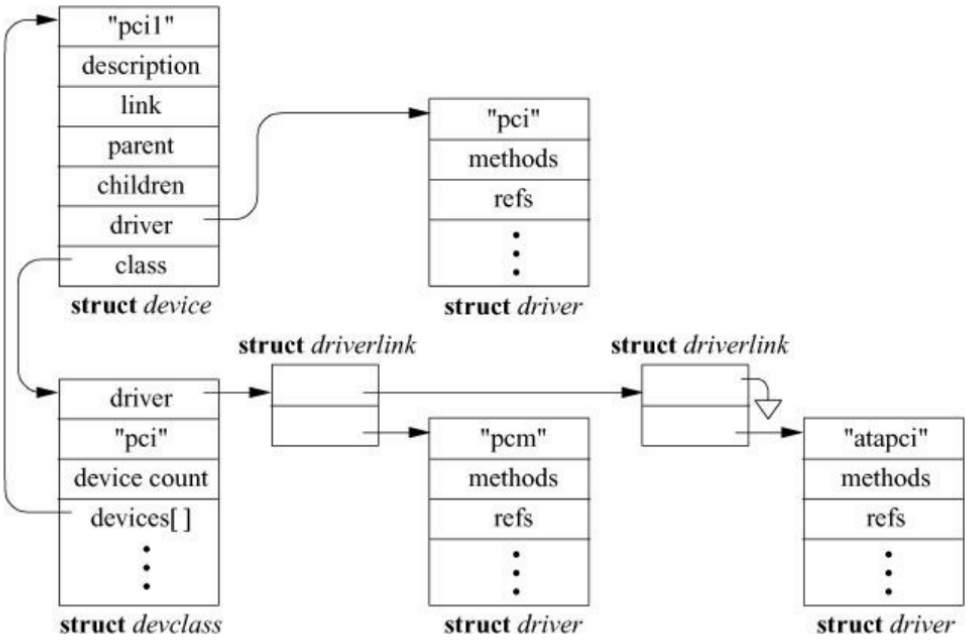
Taka klasa urządzeń zostanie utworzona na żądanie, chyba że została ona utworzona na etapie `SI_SUB_DRIVERS`, ponieważ któryś sterownik przekazał właśnie taki parametr w makrze `DRIVER_MODULE`.

Rejestracja urządzenia w klasie urządzeń polega m.in. na zaalokowaniu następnego wolnego numeru jednostkowego, który jest indeksem tego urządzenia w liście wskaźników na urządzenia w danej klasie (pole `devices` w strukturze devclass). Pełną nazwą urządzenia będzie np. *atapci0*.

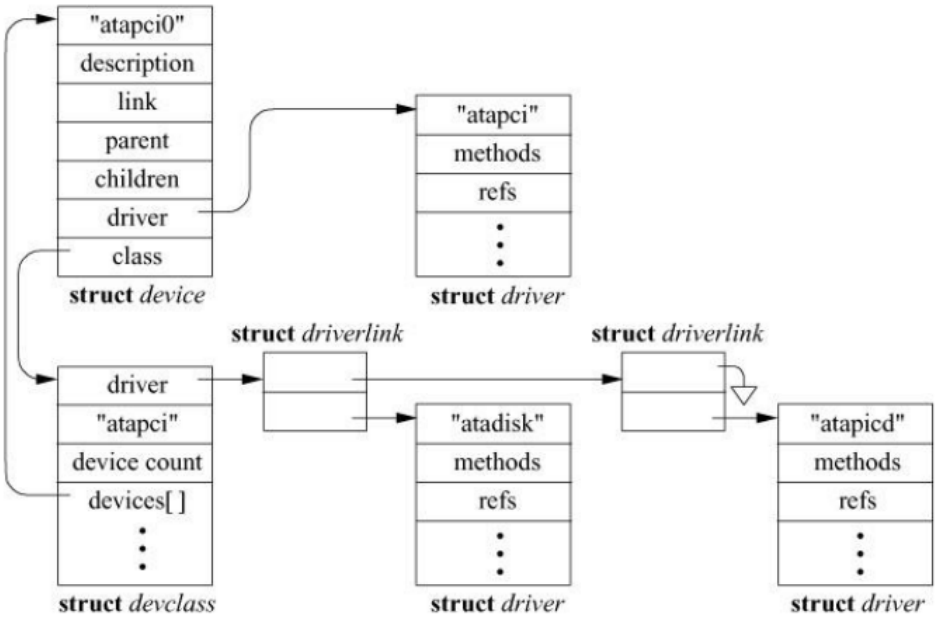
Poniższe procedury są najbardziej podstawowymi procedurami związanymi z klasami urządzeń.

```
static devclass_t devclass_find_internal(const char *classname,
                       const char *parentname, int create);

int devclass_add_driver(devclass_t dc, driver_t *driver,
                     int pass, devclass_t *dcp);
```



Rysunek 6.1: Struktury devclasses [67].



Rysunek 6.2: Struktury devclasses [68].

Pierwsza z tych procedur znajduje daną (`classname`) klasę urządzeń w systemie, bądź jeśli takiej nie znajdzie, może ją utworzyć (`create`). Znaleziona (bądź utworzona klasa), jest podczepiana pod klasę rodzica (`parentname`), która jeśli nie istnieje również może zostać utworzona.

Druga z procedur dodaje sterownik (`driver`) do danej klasy urządzeń (`dc`). Klasa ta najczęściej jest klasą o nazwie równej parametrowi `busname` makra `DRIVER_MODULE`. Ponadto inicjalizowana jest struktura devclass tego właśnie sterownika poprzez argument `dcp`. W tej procedurze, również jak w poprzedniej, klasy urządzeń mogą być tworzone na żądanie jeśli jeszcze nie istnieją. Funkcja ta dodatkowo kompiluje sterownik jako obiekt w podsystemie `kobj`.

System posiada globalną listę klas urządzeń:

```
static devclass_list_t devclasses = TAILQ_HEAD_INITIALIZER(devclasses);
```

Listing 6..7: Struktura driverlink

```
1 struct driverlink {
2   kobj_class_t  driver;
3   TAILQ_ENTRY(driverlink) link;  // lista sterowników w klasie
4   int pass;  // zmienna używana w podsystemie buspasses
5   TAILQ_ENTRY(driverlink) passlink;
6 };
```

Warte wspomnienia są też następujące funkcje:

```
1 int devclass_add_device(devclass_t dc, device_t dev)
2 int devclass_alloc_unit(devclass_t dc, device_t dev, int *unitp)
```

Odpowiednio dodają urządzenie do danej klasy oraz wyznaczają i rezerwują odpowiedni numer urządzenia w danej klasie.

6.3. Wstępna konfiguracja urządzeń

Etapy inicjalizacji systemu związane z infrastrukturą do zarządzania urządzeniami sprzętowymi i sterownikami to `SI_SUB_DRIVERS` i `SI_SUB_CONFIGURE`, w takiej właśnie kolejności.

Listing 6..8: Etapy inicjalizacji systemu

```
1 enum sysinit_sub_id {
2   ...
3   SI_SUB_DRIVERS    = 0x3100000, /* inicjalizacja sterowników */
4   SI_SUB_CONFIGURE  = 0x3800000, /* konfiguracja urządzeń */
5   ...
6 }
```

Podczas etapu inicjalizacji `SI_SUB_DRIVERS` ładowane są moduły jądra jądra, które zawierają kod sterowników. W pierwszej kolejności jest to kod sterownika urządzenia `root`, omówiony wcześniej.

Każdy moduł będący sterownikiem urządzenia sprzętowego wywołuje makro `DRIVER_MODULE(name, busname, driver, devclass, evh, arg)`.
Argument `busname` specyfikuje do jakiego typu urządzenia (np. magistrali, mostu) podłączone jest dane urządzenie. Sterownik trafi zatem do klasy urządzeń grupującej sterowniki obsługujące urządzenia podłączane do urządzenia typu `busname`. Argumenty evh i arg pozwalają przekazać **rozszerzający uchwyt zdarzeń modułu** i jego argument.

Makro to dalej rozwija się do makra `EARLY_DRIVER_MODULE_ORDERED(name, busname, driver, devclass, evh, arg, order, pass)`, z argumentem `order` [5.2.] równym `SI_ORDER_MIDDLE`. Argument pass jest wartością używaną w mechanizmie `buspasses` omówionym w rozdziale 7.4.. Makro to definiuje następujące struktury:

Listing 6..9: Makro EARLY_DRIVER_MODULE_ORDERED

```
1  #define EARLY_DRIVER_MODULE_ORDERED(name, busname, driver,     \
2    devclass, evh, arg, order, pass)                             \
3                                                                 \
4  static                                                         \
5  struct driver_module_data name##_##busname##_driver_mod = {    \
6    evh, arg,                                                    \
7    #busname,                                                    \
8    (kobj_class_t) &driver,                                      \
9    &devclass,                                                   \
10   pass                                                         \
11 };                                                             \
12                                                                \
13 static moduledata_t name##_##busname##_mod = {                 \
14   #busname "/" #name,                                          \
15   driver_module_handler,                                       \
16   &name##_##busname##_driver_mod                               \
17 };                                                             \
18 DECLARE_MODULE(name##_##busname, name##_##busname##_mod,       \
19   SI_SUB_DRIVERS,order)
```

Listing 6..10: Definicja struktury driver_module_data

```
1 struct driver_module_data {
2   int     (*dmd_chainevh)(struct module *, int, void *);
3   void        *dmd_chainarg;
4   const char  *dmd_busname;
5   kobj_class_t    dmd_driver;
6   devclass_t  *dmd_devclass;
7   int     dmd_pass;
8 };
```

efektywnie czyniąc z kodu sterownika moduł jądra, a następnie wywołuje makro `DECLARE_MODULE(name##_##busname, name##_##busname##_mod, SI_SUB_DRIVERS, order)` odpowiadające za zarejestrowanie modułu jądra w odpowiednim etapie inicjalizacji systemu [5..5], czyli w `SI_SUB_DRIVERS`.

Podstawową rzeczą jaką system operacyjny musi wiedzieć o module jądra, to jak wygląda **uchwyt zdarzeń modułu**. W przypadku sterowników uchwyt zdarzeń modułu wygląda następująco:

```
1  int
2  driver_module_handler(module_t mod, int what, void *arg)
3  {
4    struct driver_module_data *dmd;
5    devclass_t bus_devclass;
6    kobj_class_t driver;
7    int error = 0;
8    int pass;
9
10   dmd = (struct driver_module_data *)arg;
11   bus_devclass = devclass_find_internal(dmd->dmd_busname, NULL,
12     TRUE);
13
14   switch (what) {
15   case MOD_LOAD:
16     if (dmd->dmd_chainevh)
17       error = dmd->dmd_chainevh(mod,what,dmd->dmd_chainarg);
18
19     pass = dmd->dmd_pass;
20     driver = dmd->dmd_driver;
21     error = devclass_add_driver(bus_devclass, driver, pass,
22       dmd->dmd_devclass);
23       break;
24   case MOD_UNLOAD:
25     error = devclass_delete_driver(bus_devclass,
26       dmd->dmd_driver);
27     if (!error && dmd->dmd_chainevh)
28       error = dmd->dmd_chainevh(mod,what,dmd->dmd_chainarg);
29     break;
30   case MOD_QUIESCE:
31     ...
32   }
33   return (error);
34 }
```

Widzimy po uchwycie zdarzeń modułu sterowników, że główną operacją przy ładowaniu modułu sterownika jest inicjalizacja **klas urządzeń (ang. devclasses** [6.2.], za pomocą omówionych wcześniej procedur. Najpierw znajdowana jest klasa urządzenia, do której będzie podłączone urządzenie związane z tym sterownikiem.

Następnie sterownik ten dodawany jest do owej klasy, oraz jest kompilowany jako klasa kobj. Również jeśli podana jest procedura rozszerzającego uchwytu zdarzeń modułu (`dmd_chainevh`), jest ona wykonywana. Jest to procedura o takim samym prototypie jak uchwyt zdarzeń modułu, pozwala rozszerzać standardową logikę przy ładowaniu sterownika.

Gdy moduł sterownika jest odczepiany, z odpowiedniej klasy urządzeń owy sterownik jest usuwany. Dodatkowo może zostać wykonany rozszerzający uchwyt zdarzeń modułu.

6.4. Przykładowe urządzenie

Pokażemy przykładową definicję sterownika kompletnie zmyślonego urządzenia - CFD (ang. Completely Fabricated Device), która posłuży lepszej orientacji przy czytaniu następnych rozdziałów.

Listing 6..11: Szkielet kodu sterownika

```
1  static int cfd_probe(device_t);
2  static int cfd_attach(device_t);
3  static int cfd_detach(device_t);
4  static int cfd_frob(device_t, int, int);
5  static int cfd_twiddle(device_t, char *);
6
7  static device_method_t cfd_methods[] = {
8    /* Methods from the device interface */
9    DEVMETHOD(device_probe, cfd_probe),
10   DEVMETHOD(device_attach, cfd_attach),
11   DEVMETHOD(device_detach, cfd_detach),
12
13   /* Methods from the bogo interface */
14   DEVMETHOD(bogo_frob, cfd_frob),
15   DEVMETHOD(bogo_twiddle, cfd_twiddle),
16
17   /* Terminate method list */
18   DEVMETHOD_END
19 };
20
21 static driver_t cfd_driver = {
22   "cfd",
23   cfd_methods,
24   sizeof(struct cfd_softc)
25 };
26
27 static devclass_t cfd_devclass;
28
29 DRIVER_MODULE(cfd, bogo, cfd_driver, cfd_devclass, NULL, NULL);
```

Każdy sterownik poza `root` wykorzystuje następujące makrodefinicje: `#define device_method_t kobj_method_t;`
`#define DEVMETHOD KOBJMETHOD`.

Powyższy kod można traktować jako minimalną implementację sterownika urządzenia sprzętowego. Musi on implementować co najmniej procedury `probe`, `attach` i `detach` z interfejsu `device`. Zostaną one omówione przy omawianiu procesu autokonfiguracji [7.1.]. Widzimy również, że urządzenie implementuje metody z hipotetycznego interfejsu `bogo`. Mogą to być interfejsy `bus`, bądź specyficzne dla danej magistrali takie jak np. `pci`. Niezbędna jest definicja struktury `driver_t cfd_driver`, oraz deklaracja struktury `devclass_t cfd_devclass`, których adresy są przekazywane za pomocą makra `DRIVER_MODULE` podsystemowi sysinit.

6.5. Interfejsy w sterownikach

Kod sterowników w większości opiera się na implementowaniu metod z różnych interfejsów zrealizowanych za pomocą podsystemu kobj.

Bez interfejsów zmiana kodu sterownika mogłaby wymagać ponownej kompilacji innych części jądra. W dodatku modyfikacja jak i ponowna kompilacja niektórych przestarzałych sterowników może z różnych powodów nie być już możliwa. Z tych powodów stabilny interfejs jest kluczowy przy programowaniu sterowników.

Najważniejszym interfejsem, który muszą implementować wszystkie urządzenia jest interfejs `device` będący trzonem podsystemu NewBus odpowiedzialnego za dopasowanie sterowników do urządzeń i dalszą ich inicjalizację. Zostanie dokładniej omówiony w dalszej części pracy.

W kodzie sterowników urządzeń, definiowane są również obsługiwane procedury z innych interfejsów. Na przykład interfejsem, który implementują szyny jest interfejs `bus`, opisany w pliku bus_if.m Istnieją także interfejsy specyficzne dla sterowników magistral konkretnego typu (np. PCI), bądź inne definiowane przez programistę wedle uznania.

W interfejsach, dla niektórych metod zaimplementowane są wywołania dymyślne, jak i wywołania generyczne mające obsługiwać podstawowe przypadki.

Rozdział 7.

Autokonfiguracja

7.1. NewBus

Kiedy wszystkie sterowniki urządzeń są zarejestrowane w systemie, jądro przechodzi do następnego etapu inicjalizacji [5..5] - SI_SUB_CONFIGURE. W tym etapie zachodzi proces zwany **autokonfiguracją**.

Podsystem NewBus zapewnia stabilny interfejs dla programistów sterowników urządzeń, oraz szereg abstrakcji pozwalający w wygodny sposób przeprowadzać ich inicjalizację. Zarządzanie urządzeniami jak i ich działanie również się upraszcza do wywoływania odpowiednich metod z interfejsu.

W pliku *autoconf.c*, zdefiniowane są trzy procedury odpowiadające za autokonfigurację: `configure_first`, `configure` i `configure_final`.

Listing 7..1: Rejestracja procedur w podsystemie sysinit.

```
1 SYSINIT(configure1, SI_SUB_CONFIGURE, SI_ORDER_FIRST,
2         configure_first, NULL);
3 SYSINIT(configure2, SI_SUB_CONFIGURE, SI_ORDER_THIRD,
4         configure, NULL);
5 SYSINIT(configure3, SI_SUB_CONFIGURE, SI_ORDER_ANY,
6         configure_final, NULL);
```

Pierwsza i trzecia z nich wykonują się odpowiednio przed i po skonfigurowaniu i utworzeniu drzewa urządzeń. Zapewniają docelowemu procesowi autokonfiguracji pewne warunki wstępne jak i wykonują pewne operacje po zakończeniu autokonfiguracji. Np. procedura `configure_first` dodaje urządzenie `nexus` jako dziecko urządzenia `root`.

Procedura `configure` jest punktem startowym autokonfiguracji. Wywoływana jest procedura `root_bus_configure`, która następnie woła procedurę `bus_set_pass(BUS_PASS_DEFAULT)` [7.4.] rozpoczynającą dynamiczne tworzenie drzewa urządzeń, wraz z ich inicjalizacją.

W procesie konfiguracji urządzeń należy zidentyfikować wszystkie urządzenia w systemie jak i znaleźć odpowiadające im sterowniki. Jeżeli szyna/magistrala nie jest w stanie zidentyfikować podłączonych do niej urządzeń, używa metody `identify` z interfejsu `device`. Metoda ta identyfikuje (w zasadzie w dowolny sposób) i dodaje nowe urządzenie do danej szyny. Niekiedy procedura ta po prostu zakłada, że dane urządzenie jest podłączone do systemu.

Sterownik każdego urządzenia sprzętowego musi implementować co najmniej metody `probe`, `attach`, oraz `detach` z interfejsu `device`.

Zidentyfikowane urządzenia podłączone do danej szyny są przekazywane do metody `probe` sterowników w klasie urządzeń owej szyny.

Metoda `probe` sterownika służy do sprawdzenia jaki stopień zgodności deklaruje on z danym urządzeniem. System operacyjny wywołuję tę metodę, z danym urządzeniem jako argumentem, dla każdego sterownika w klasie urządzeń szyny do której jest podłączony. Sterownik którego metoda probe zwróciła największą wartość (mniejszą, bądź równą zero; wartości większe od 0 oznaczają błąd wykonania), jest wybierany jako najlepiej pasujący i przechodzi do dalszego etapu konfiguracji. Proces ten w autorskim tłumaczeniu nazywamy **próbkowaniem**. Jeżeli żaden ze sterowników nie zwrócił zadowalającej wartości, urządzenie nie jest obsługiwane.

Listing 7..2: Zwracane wartości probe

```
1  #define BUS_PROBE_SPECIFIC  0  /* Sterownik specyficzny dla
2                                    danego urządzenia. */
3  #define BUS_PROBE_VENDOR   (-10) /* Sterownik dostarczony
4                                      przez producenta. */
5  #define BUS_PROBE_DEFAULT   (-20)  /* Domyślny sterownik. */
6  #define BUS_PROBE_LOW_PRIORITY (-40) /* Starszy sterownik. */
7  #define BUS_PROBE_GENERIC  (-100)  /* Generyczny sterownik dla
8                                        danego typu urządzeń. */
9  #define BUS_PROBE_HOOVER   (-1000000) /* Sterownik obsługujący
10                                          każde urządzenie danej
11                                          magistrali. */
12 #define BUS_PROBE_NOWILDCARD  (-2000000000) // Brak dopasowania.
```

Kiedy najlepszy sterownik został odnaleziony, wywoływana jest jego metoda attach. W metodzie tej dokonuje się właściwa inicjalizacja danego urządzenia, specyficzna dla niego. Rezerwowane są zasoby i rejestrowane procedury obsługi przerwań. Między innymi sterownik rezerwuje miejsce na prywatne dane danej instancji urządzenia (pole `softc` w strukturze `device_t`) [91].

Zadaniem szyny podczas attach, oprócz inicjalizacji samej siebie jest enumeracja podłączonych do niej urządzeń sprzętowych, utworzenie reprezentujących je urządzeń programowych (struktur `device_t`), a następnie wykonanie na nich sekwencji probe i attach. Procedurą realizującą tą sekwencję jest `bus_generic_attach` [81]. Sterownik swoje prywatne dane może umieścić pod wskaźnikiem będącym polem `ivars`

struktury `device_t` [90].

Urządzenia nie posiadające podległych urządzeń są liśćmi w drzewie urządzeń i nie kontynuują procesu autokonfiguracji.

7.2. Zarządzanie zasobami sprzętowymi

Urządzenia potrzebują zasobów, którymi najczęściej są obszary pamięci. Sterowniki w trakcie inicjalizacji urządzeń muszą zarezerwować zasoby oraz przypisać je do urządzeń.

Rman - Resource Manager - Menadżer Zasobów jest abstrakcją pomagającą w zarządzaniu zasobami magistral, którymi najczęściej jest pamięć. Wprowadza ona dwa pojęcia - region, czyli ciągły zakres zasobów oraz zasób (resource) będący jednym z wielu fragmentów tego regionu.

Najpierw należy strukturę typu ‘rman‘ zainicjalizować - używając funkcji `rman_init(struct rman *rm)`, następnie przekazuje mu się region funkcją `rman_manage_region(struct rman *rm, rman_res_t start, rman_res_t end)`.

Każda rezerwacja zasobu, o której można myśleć jak o alokacji, odbywa się za pomocą funkcji:

```
1 rman_reserve_resource(struct rman *rm, rman_res_t start,
2   rman_res_t end, rman_res_t count, u_int flags, device_t dev));
```

lub funkcji z innym interfejsem. Wszystkie te funkcje mają wspólną implementację.

Za pomocą argumentów wspomnianej funkcji, możemy wyspecyfikować rozmiar zasobu (`count`), oraz oczekiwania dotyczące jego lokalizacji (`start`, `end`). Możemy podać dokładny adres, przedział, w którym zasób ma zostać zarezerwowany lub pozwolić na dowolny adres.

Przez flagi możemy przekazać do jakiego rozmiaru ma być wyrównany adres. Możemy także nakazać, aby zasób był współdzielony, co jest potrzebne na przykład w przypadku magistrali ISA, gdzie wiele urządzeń dzieli tę samą pamięć.

Warto zauważyć, że jesteśmy w stanie wewnątrz zasobu utworzyć region, co tworzy drzewiastą strukturę.

W idealnym świecie istniałby region obejmujący całą fizyczną przestrzeń adresową. Jednym z jej dzieci (lub wnuków albo nawet prawnuków) byłby zasób - pamięć na którą mapowane jest PCI. Ten zasób jednocześnie byłby regionem w którym zmapowane BARy PCI urządzeń. Jednym z tych urządzeń mógłby być kontroler innej magistrali. Urządzenie podłączone do niej byłoby w końcu liściem w tym drzewie.

Urządzenia nie mają dostępu do rmanów trzymanych przez szyny, do których są podczepione. Dlatego o zasoby muszą prosić poprzez metodę `bus_alloc_resource`,

którą każdy sterownik magistrali implementuje osobno. Zazwyczaj jej działanie nie wybiega daleko poza zawołanie `rman_reserve_resource` w trzymanym przez siebie rmanie, lub zawołanie metody `bus_alloc_resource` u rodzica. W drugim przypadku można użyć gotowej metody `bus_generic_alloc_resource`.

7.3. Bus space

Jedno urządzenie może być podłączone na wiele różnych sposobów. Twórca sterownika nie wie, przez jakie szyny danych jest ono obsługiwane. Architektury komputera czasem wymuszają stosowanie MMIO, czasem PMIO. Z tego powodu systemy operacyjne oferują warstwę abstrakcji nad interakcją z pamięcią urządzenia, która we FreeBSD nazywa się **bus_space** [84]. Pozwala ona na użycie dokładnie tego samego kodu sterownika, niezależnie od architektury i organizacji platformy sprzętowej.

```
1 bus_space_map(tag, address, size, flags, handle_ptr)
```

Funkcja `bus_space_map` jako pierwszy argument bierze tag, który jednoznacznie wskazuje na typ przestrzeni adresowej, np MMIO lub PMIO w przypadku x86. W przypadku architektury MIPS, gdzie dostępne jest tylko MMIO, będzie dostępny tylko jeden tag. Funkcja ta tworzy uchwyt na zasób pod adresem ‘address‘ i rozmiarze ‘size‘.

```
1 bus_space_read_1(tag, handle, offset)
2 bus_space_read_2(tag, handle, offset)
3 bus_space_read_4(tag, handle, offset)
4 bus_space_read_8(tag, handle, offset)
5
6 bus_space_write_1(tag, handle, offset, value)
7 bus_space_write_2(tag, handle, offset, value)
8 bus_space_write_4(tag, handle, offset, value)
9 bus_space_write_8(tag, handle, offset, value)
```

Powyższe funkcje pozwalają na zapis lub odczyt danych wielkości 1, 2, 4 lub 8 bajtów. Każda z nich jako argumenty bierze tag, wcześniej utworzony uchwyt oraz przesunięcie względem początku danego obszaru.

Powyższe funkcje nie gwarantują wykonania w tej samej kolejności, co ich wywołania. W celu uczynienia operacji synchronicznymi, należy użyć funkcji:

```
1 bus_space_barrier(space, handle, offset, length, flags)
```

Która jako jako flagi bierze jedną z poniższych wartości, lub obie:

```
1 BUS_SPACE_BARRIER_READ
2 BUS_SPACE_BARRIER_WRITE
```

Zatrzymują one wykonanie kodu do momentu zakończenia wszystkich operacji danego typu.

Wszystkie powyższe metody mają osobną implementację dla każdej architektury procesora. Na przykład w przypadku x86, funkcje `bus_space_write_*`, bezpośrednio piszą do pamięci lub portów IO, zależnie od przekazanego taga jako pierwszy argument. W przypadku MIPSa i ARMa, wołany jest kod szyny, który w większości przypadków po prostu pisze pod dany adres pamięci.

7.4. buspasses

Pierwotna implementacja podsystemu NewBus zakładała inicjalizację wszystkich urządzeń za pomocą jednego przejścia (i jednoczesnego tworzenia) drzewiastej hierarchii urządzeń. Podczas tego procesu każda szyna odpowiedzialna była za zidentyfikowanie swoich dzieci i przekazanie dalej procesu inicjalizacji do każdego z nich.

Mechanizm ten działa bardzo dobrze w przypadku urządzeń, które są inicjalizowane zawsze przed urządzeniami które od nich zależą, np. reprezentacji magistral, bądź mostów.

Jednakże są odmienne przypadki. Np. kontrolery przerwań, takie jak 8259A w architekturze x86. Programowa reprezentacja tego kontrolera jest w hierarchii dzieckiem szyny ISA, która z kolei jest dzieckiem mostu PCI-ISA, który podpięty jest do szyny PCI. Inicjalizacja urządzeń podłączonych do szyny PCI będzie przeprowadzona wcześniej, a one nierzadko korzystają z kontrolera przerwań. Obejścia stosowane w takich wypadkach wprowadzają kod zależny od platformy, poza podsystemem NewBus.

Rozwiązaniem powyższych problemów jest dodanie mechanizmu **buspasses** [2]. Zakłada on wielokrotne przechodzenie drzewa urządzeń i inicjalizowanie ich. Sterownik może próbkować urządzenia wtedy i tylko wtedy, gdy nadany mu **pass level** jest mniejszy niż ogólnosystemowy **pass level**.

Logika ta zaimplementowana jest w większości w procedurach `bus_generic_probe`, oraz `bus_generic_attach`. Kiedy szyna wywołuje te metody, aby rozpocząć inicjalizację swoich dzieci, inicjalizacja ta dokonywana jest tylko na tych które mają odpowiedni pass level.

System musi zarządzać poziomami pass level. Implementacja przejścia hierarchii urządzeń musi być efektywna w przypadku rzadko rozłożonych poziomów pass level. System trzyma więc listę aktywnych pass level - tych, które są niepuste. Lista ta jest wyznaczana przy ładowaniu modułów sterowników (argument `pass` w makrze `DEVICE_MODULE`).
Aktualny poziom jest trzymany w globalnej zmiennej `bus_current_pass`. Pass le-

vel może zostać zwiększony korzystając z procedury `bus_set_pass`, której wywołanie pośrednio powoduje ponowne przeskanowanie całego drzewa urządzeń z nowym poziomem pass level. Podniesiony poziom nie może zostać zmniejszony. Powyższa procedura może powodować wielokrotne przejście hierarchii urządzeń po kolei dla aktywnych pass levels, zatrzymuje się gdy lista aktywnych będzie pusta, bądź natrafi na pass level wyższy niż aktualny. Wszystkie mniejsze niż aktualny pass level są pomijane.

Ponowne przeskanowanie drzewa urządzeń dokonywane jest za pomocą procedury z interfejsu `bus`: `bus_new_pass`. Istnieje również generyczna funkcja: `bus_generic_new_pass`. Procedura ta przejście całej hierarchii urządzeń od samego urządzenia root0. Działa ona następująco: jeśli któreś z dzieci danego urządzenia używa nowego pass level, to jego procedura identify jest wołana, i dodawany jest nowy pass level. Następnie jeśli dziecko ma już dobrany sterownik, to wołana jest jego implementacja metody `bus_new_pass`. W przeciwnym wypadku próbkowanie urządzenia wykonywane jest ponownie.

Listing 7..3: Poziomy pass level

```
1  #define BUS_PASS_ROOT       0   /* Używany przez root0. */
2  #define BUS_PASS_BUS        10  /* Magistrale i mostki. */
3  #define BUS_PASS_CPU        20  /* Urządzenia CPU. */
4  #define BUS_PASS_RESOURCE   30  /* Enumeracja zasobów. */
5  #define BUS_PASS_INTERRUPT  40  /* Kontrolery przerwań. */
6  #define BUS_PASS_TIMER      50  /* Zegary. */
7  #define BUS_PASS_SCHEDULER  60  /* Włączenie planisty. */
8  #define BUS_PASS_DEFAULT    __INT_MAX /* Ostatni poziom. */
9
10 #define BUS_PASS_ORDER_FIRST    0
11 #define BUS_PASS_ORDER_EARLY    2
12 #define BUS_PASS_ORDER_MIDDLE   5
13 #define BUS_PASS_ORDER_LATE 7
14 #define BUS_PASS_ORDER_LAST 9
```

Teraz kontrolery przerwań mogą normalnie uczestniczyć w inicjalizacji jako urządzenia podsystemu NewBus, bez zbędnych obejść.

Rozdział 8.

Mimiker

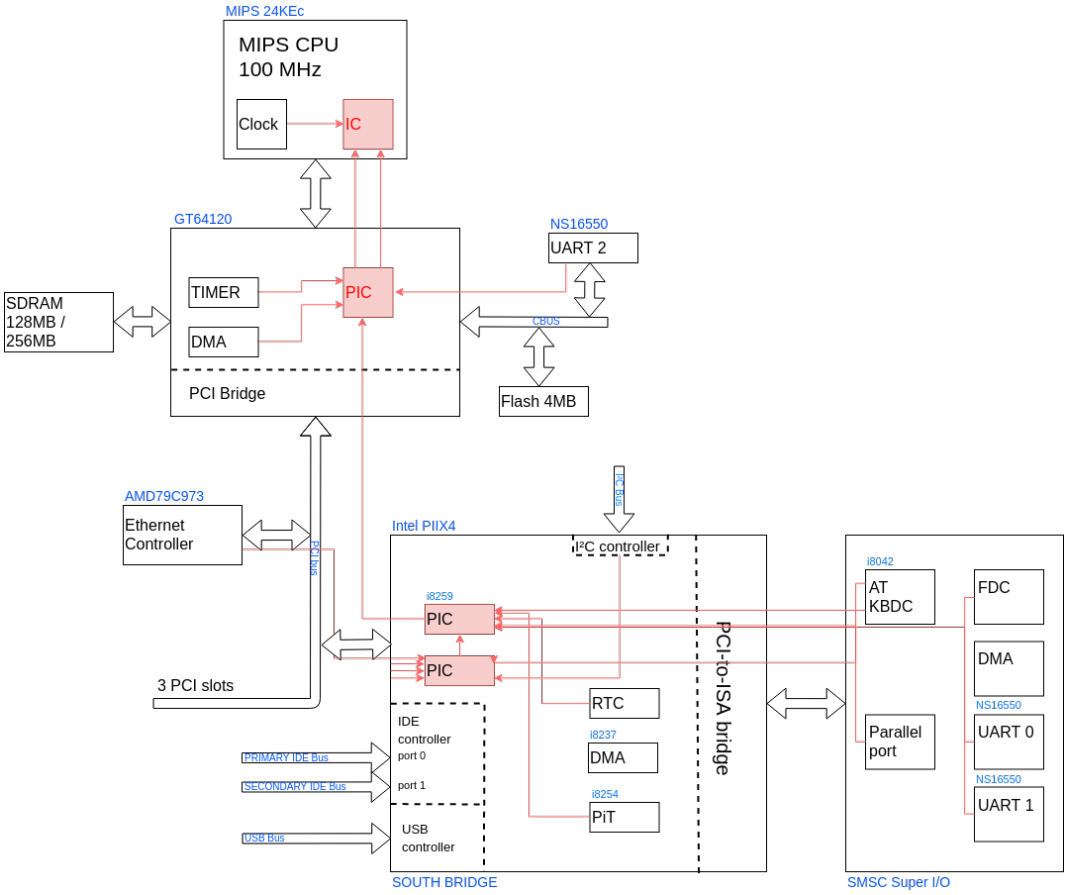
Mimiker jest prostym unixo-podobnym systemem operacyjnym powstającym od listopada 2015 na naszej uczelni. Powstał tylko w celach edukacyjnych, bez ambicji na bycie lepszym od już istniejących systemów pisanych przez całe rzesze ekspertów. Duża część kodu naśladuje FreeBSD. Mimiker jest dostępny na licencji BSD pod adresem `mimiker.ii.uni.wroc.pl`. W momencie pisania tego tekstu, lista kontrybutorów składa się z 19 osób.

Mimiker działa na platformie Malta z procesorem MIPS, której opis znajduje się w następnym rozdziale. Przy rozwoju Mimikera nie używamy fizycznej płytki, lecz emulatora Qemu, ponieważ jest to dla nas znacznie wygodniejsze. Quemu nie emuluje wszystkiego, co byśmy chcieli, na przykład wyścigów między głównym potokiem procesora, a FPU. Mimiker od dawna nie był testowany na prawdziwym sprzęcie i najprawdopodobniej uruchomienie go na nim wymagałoby wielu poprawek.

W niedalekiej przyszłości chcemy zacząć wspierać platformy wieloprocesorowe, a następnie przenieść system na architekturę ARM, używając popularnej płytki Raspberry Pi. Od dłuższego czasu staramy się pisać kod tak, aby jak najmniejsza jego ilość była zależna od platformy sprzętowej, dzięki czemu tak duża zmiana, będzie wymagała przepisania tylko niektórych mechanizmów. Polega to na wyraźnym oddzieleniu kodu specyficznego dla danej platformy - na przykład zapisu kontekstu wątku przy przełączeniu - od kodu niezależnego od platformy - na przykład planisty systemowego.

8.1. Płytka ewaluacyjna Malta

W celu umożliwienia pracy projektantom sprzętu oraz oprogramowania, nad mikroprocesorem, na rynku pojawiły się płytki ewaluacyjne. Są to obwody drukowane zawierające dany procesor, oraz współpracujące z nim komponenty, na przykład sprzętowy debugger, zegar czasu rzeczywistego, czy port szeregowy. Szczególnie na początku, płytki ewaluacyjne były dostarczane przez producenta danego mi-



Rysunek 8.1: Platforma Malta

kroprocesora, ale nigdy nie było to regułą. Z czasem takie urządzenia zaczęły być wykorzystywane także przez hobbystów, co doprowadziło do powstania urządzeń o możliwościach daleko wybiegających poza pracę z samym procesorem. Zaczęły one przypominać kompletne komputery.

Malta jest już dosyć starą i trudną do zdobycia w dzisiejszych czasach płytką ewaluacyjną w rozmiarze ATX, stworzoną przez firmę MIPS Technologies w celu umożliwienia pracy nad układami mips32 i mips64. Na płytce znajdziemy kontrolery PCI, IDE, USB, zegar czasu rzeczywistego, most południowy PIIX4, PS/2, ethernet. Jest to bardzo popularny sprzęt (jak na tę architekturę), wspierają ją systemy takie jak Linux, FreeBSD oraz NetBSD.

8.2. MIPS

Architektura MIPS[46], na której pracujemy, kiedyś była używana w komputerach osobistych, najlepszych superkomputerach, oraz konsolach do gier takich jak PlayStation 1, PlayStation 2, PlayStation portable. Dziś w większości została wy-

parta przez architektury z rodziny x86 oraz ARM. Można ją nadal znaleźć w systemach wbudowanych, na przykład w najtańszym sprzęcie sieciowym.

Oryginalnie procesor był 32-bitowy, jednak powstały także wersje 64-bitowe. Jest to architektura typu RISC, oraz load/store - czyli z możliwie prostymi instrukcjami oraz z wyraźnym rozdzieleniem tych odpowiedzialnych za interakcję z pamięcią od tych odpowiedzialnych za obliczenia. Z powodu prostoty, często znajduje zastosowanie w celach edukacyjnych.

Specyficzny dla tej architektury jest Branch Delay Slot - zgodnie ze specyfikacją, jedna instrukcja bezpośrednio po instrukcji skoku, zostanie wykonana zawsze, nawet jeśli skok w inne miejsce się wykona. Adresy instrukcji muszą być wyrównane do jej rozmiaru, czyli na przykład czterobajtowa instrukcja nie może znajdować się pod adresem kończącym się cyfrą 3, gdyż taki adres nie jest wielokrotnością liczby 4.

Procesory MIPS wspierają 4 koprocesory, czyli procesory pomocnicze. Pierwszym z nich jest zawsze koprocesor systemowy, który odpowiada za takie mechanizmy jak tłumaczenie adresów z wirtualnych na fizyczne lub obsługa wyjątków. Drugim jest zazwyczaj FPU, który nie jest jednak obowiązkowy. Pozostałe dwa nie są zdefiniowane przez standard. Na przykład dodatkowym koprocesorem w konsoli do gier PlayStation był odpowiedzialny za wspomaganie obliczeń grafiki trójwymiarowej.

8.3. Co dopisaliśmy

Podczas pracy z Mimikerem zastaliśmy poważny problem, polegający na przydzielaniu BARom PCI adresów, które nachodziły na sztywno zmapowaną pamięć magistrali ISA.

Z pomocą przyszedł RMAN - bardzo podobny resource manager do opisanego wcześniej ale trochę prostszy.

Rezerwacja, nazwana u nas alokacją, odbywa się za pomocą funkcji:

```
1 resource_t *
2 rman_alloc_resource(rman_t *rm, rman_addr_t start,
3   rman_addr_t end, size_t count, size_t bound,
4   res_flags_t flags, device_t *dev);
```

Cała implementacja resource managera zajmuje niewiele ponad 100 linii kodu. O każdej instancji struktury `rman_t` możemy myśleć jak o liście zasobów (struktur `resource_t`). Początkowo zawiera on jeden zasób odpowiadający całemu jego rozmiarowi. Każda alokacja znajduje pierwszy wolny zasób będący w stanie pomieścić rozmiar zadany jako argument i jeśli jest on większy, wykraja pozostałą część jako nowy zasób (w niektórych przypadkach nawet jako dwa zasoby - przed i za zaalokowanym). Ponadto możliwe jest wyspecyfikowanie, aby alokacja miała miejsce w konkretnym miejscu, za pomocą argumentów `start` i `end`.

Alokacja zasobów współdzielonych nie jest obsługiwana, ponieważ struktura resource_t zawiera wskaźnik na urządzenie. Współdzielony zasób musiałby wskazywać na wiele urządzeń - właścicieli, a to komplikuje implementację, która z założenia miała być bardzo prosta.

Po wprowadzeniu kodu odpowiedzialnego za zarządzanie zasobami do Mimikera należało wprowadzić do infrastruktury sterowników interfejs, który pozwalałby korzystać z kodu menedżera zasobów. Zaimplementowaliśmy interfejs podobny do opisanego wcześniej interfejsu `bus_*_resource` z FreeBSD. Uprościliśmy interfejs do jednej funkcji:

```
1 resource_t *
2 bus_resource_alloc(device_t *dev, resource_type_t type, int rid,
3   rman_addr_t start, rman_addr_t end, size_t size,
4   unsigned flags)
```

Użycie powyższej funkcji jest takie samo jak w analogicznej funkcji z FreeBSD omówionej we wcześniejszych rozdziałach.

Naszym docelowym założeniem było, aby każdy dostęp do zasobów urządzeń był wykonywany poprzez uzyskaną wcześniej za pomocą powyższej funkcji reprezentacją zasobu `resource_t`. Zasób ten pobierany był z menedżera zasobów nadrzędnej szyny. Podczas procesu konfiguracji urządzeń i sterowników, każda szyna pobiera od rodzica (za pomocą bus_resource_alloc) odpowiedni zasób, którym inicjalizowany jest resource manager owej szyny. Z tego resource managera podrzędne względem tej szyny urządzenia będą mogły prosić o zasoby. W związku z tym każda szyna, której dzieci mogą prosić o zasoby musi implementować funkcję obsługującą takie żądanie i wyeksportować wskaźnik na nią w strukturze `bus_methods_t`.

Taki mechanizm pozwala na dynamiczne przydzielanie zasobów, oraz pozwala wykryć i uniknąć konfliktów przy ich przydzielaniu.

Napisany przez nas kod można znaleźć pod linkiem `https://github.com/cahirwpz/mimiker/pull/447`.

Rozdział 9.

Zalecana dalsza lektura

- Andrew S. Tanenbaum, Herbert Bos - *Systemy operacyjne*
- Marshall Kirk McKusick, George V. Neville-Neil, Robert N.M. Watson - *The Design and Implementation of the FreeBSD Operating System*
- Paul E. McKenney - *Memory Barriers: a Hardware View for Software Hackers*
- Joseph Kong - *FreeBSD Device Drivers: A Guide for the Intrepid*
- Alessandro Rubini, Jonathan Corbet - *Linux Device Drivers*
- Bruce Jacob, David T. Wang, Spencer W. Ng - *Memory Systems: Cache, DRAM, Disk*

Bibliografia

[1] John von Neumann, *First Draft of a Report on the EDVAC*

[2] John H. Baldwin, *Multiple Passes of the FreeBSD Device Tree*, https://people.freebsd.org/~jhb/papers/bsdcan/2009/article.ps

[3] Bruce Jacob, David T. Wang, Spencer W. Ng, *Memory Systems: Cache, DRAM, Disk*, rozdział 18

[4] Alessandro Rubini, Jonathan Corbet, *Linux Device Drivers*, wydanie II, Using resources

[5] Andrew S. Tanenbaum, Herbert Bos, *Systemy operacyjne*, wydanie IV, rozdział 1.3

[6] Andrew S. Tanenbaum, Herbert Bos, *Systemy operacyjne*, wydanie IV, rozdział 1.2

[7] Andrew S. Tanenbaum, Herbert Bos, *Systemy operacyjne*, wydanie IV, rozdział 1.3.3

[8] Jiri Soukup, Code Farms Inc., *Intrusive Data Structures*

[9] Wikipedia, *Time-sharing*, https://en.wikipedia.org/wiki/Time-sharing

[10] Wikipedia, *Von Neumann architecture*, https://en.wikipedia.org/wiki/Von_Neumann_architecture

[11] Wikipedia, *Batch processing*, https://en.wikipedia.org/wiki/Batch_processing

[12] Wikipedia, *Magistrala komunikacyjna* , https://pl.wikipedia.org/wiki/Magistrala_komunikacyjna

[13] Wikipedia, *Point-to-point*, https://en.wikipedia.org/wiki/Point-to-point_(telecommunications)

[14] Wikipedia, *SCSI*, https://en.wikipedia.org/wiki/SCSI

[15] Wikipedia, *Parallel ATA*, https://en.wikipedia.org/wiki/Parallel_ATA

[16] Wikipedia, *PCI-express*, https://en.wikipedia.org/wiki/PCI_Express

[17] Wikipedia, *Conventional PCI - Arbitration*, https://en.wikipedia.org/wiki/Conventional_PCI#Arbitration

[18] Wikipedia, *Front Side Bus*, https://en.wikipedia.org/wiki/Front-side_bus

[19] Wikipedia, *Southbridge (computing)*, https://en.wikipedia.org/wiki/Southbridge_(computing)

[20] Wikipedia, *Northbridge (computing)*, https://en.wikipedia.org/wiki/Northbridge_(computing)

[21] Wikipedia, *Pleonazm*, https://pl.wikipedia.org/wiki/Pleonazm

[22] Wikipedia, *Loadable kernel module*, https://en.wikipedia.org/wiki/Loadable_kernel_module

[23] Wikipedia, *Device controller*, https://simple.wikipedia.org/wiki/Device_controller

[24] Wikipedia, *Bus (computing)*, https://en.wikipedia.org/wiki/Bus_(computing)

[25] PCI-SIG, https://pcisig.com

[26] *Serial ATA International Organization (SATA-IO)*, https://sata-io.org/

[27] Dino Dai Zovi, *Kernel Rootkits*, https://www.sans.org/reading-room/whitepapers/threats/kernel-rootkits-449

[28] Information Technology Resource and Forum Just2Good, *Introduction to the Chipset*, http://just2good.co.uk/chipset.php

[29] HyperTransport Consortium, *HyperTransport Overview*, https://www.hypertransport.org/ht-overview

[30] Intel, *An Introduction to the Intel® QuickPath Interconnect*, https://www.intel.com/content/www/us/en/io/quickpath-technology/quick-path-interconnect-introduction-paper.html

[31] Eli Billauer, *The Right Way: Managed Resource Allocation in Linux Device Drivers*, http://haifux.org/lectures/323/haifux-devres.pdf

[32] I/O Techniques, *Programmed I/O*, http://inputoutput5822.weebly.com/programmed-io.html

[33] Wikipedia, *Memory-mapped I/O*, https://en.wikipedia.org/wiki/Memory-mapped_I/O

[34] *Interrupts and Interrupt Handlers*, https://notes.shichao.io/lkd/ch7/

[35] Intel, *8259A PROGRAMMABLE INTERRUPT CONTROLLER (8259A/8259A-2)*, https://pdos.csail.mit.edu/6.828/2014/readings/hardware/8259A.pdf

[36] jcm's blog, *Everything you know about interrupts is wrong*, http://www.jonmasters.org/blog/2007/12/12/everything-you-know-about-interrupts-is-wrong/

[37] Wikipedia, *Message Signaled Interrupts*, https://en.wikipedia.org/wiki/Message_Signaled_Interrupts

[38] Wikipedia, *Advanced Programmable Interrupt Controller*, https://en.wikipedia.org/wiki/Advanced_Programmable_Interrupt_Controller

[39] Wikipedia, *Direct memory access*, https://en.wikipedia.org/wiki/Direct_memory_access

[40] Wikipedia, *Industry Standard Architecture*, https://en.wikipedia.org/wiki/Industry_Standard_Architecture

[41] Beyond Logic, *USB in a NutShell - Making sense of the USB standard*, https://www.beyondlogic.org/usbnutshell/usb1.shtml

[42] Wikipedia, *Plug and Play*, https://en.wikipedia.org/wiki/Plug_and_play

[43] Wikipedia, *Legacy Plug and Play*, https://en.wikipedia.org/wiki/Legacy_Plug_and_Play

[44] Artykuł na temat podsystemu `kobj`, https://people.freebsd.org/~jake/kobj/kobj.txt

[45] *FreeBSD Architecture Handbook*, rozdział 3.3: Kernel Objects - Using Kobj, https://www.freebsd.org/doc/en/books/arch-handbook/kernel-objects-using.html

[46] *MIPS32® 24K® Processor Core Family Software User's Manual*, https://people.freebsd.org/~adrian/mips/MD00343-2B-24K-SUM-03.11.pdf

[47] *MIPS® Malta^{TM}-R Development PlatformUser's Manual*, https://manualzz.com/doc/7240054/mips%C2%AE-malta%E2%84%A2-r-development-platform-user-s-manual

[48] Wikipedia, *Cache coherence*, https://en.wikipedia.org/wiki/Cache_coherence

[49] Wikipedia, *Bus mastering*, https://en.wikipedia.org/wiki/Bus_mastering

[50] PC GUIDE, *PCI Bus Mastering*, http://www.pcguide.com/ref/mbsys/buses/types/pciMastering-c.html

[51] HARDWAREBOOK, *ISA*, http://www.hardwarebook.info/ISA#Bus_Mastering

[52] WIKIPEDIA, *BIOS*, https://en.wikipedia.org/wiki/BIOS

[53] UNIFIED EXTENSIBLE FIRMWARE INTERFACE FORUM, http://www.uefi.org/

[54] Wątek listy mailingowej FreeBSD - *Understanding the FreeBSD locking mechanism*, https://groups.google.com/forum/#!topic/muc.lists.freebsd.hackers/FYk8EtGY-is

[55] FREEBSD ARCHITECTURE HANDBOOK, *SMP design*, https://www.freebsd.org/doc/en/books/arch-handbook/smp-design.html

[56] Kod źródłowy FreeBSD - plik queue.h, http://fxr.watson.org/fxr/source/sys/queue.h?v=FREEBSD11

[57] Kod źródłowy FreeBSD - plik tree.h, http://fxr.watson.org/fxr/source/sys/tree.h?v=FREEBSD11

[58] Kod źródłowy FreeBSD - plik resource.h, http://fxr.watson.org/fxr/source/mips/include/resource.h?v=FREEBSD11

[59] Kod źródłowy FreeBSD - plik subr_bus.c, http://fxr.watson.org/fxr/source/kern/subr_bus.c

[60] Kod źródłowy FreeBSD - plik intr_machdep.c, http://fxr.watson.org/fxr/source/mips/mips/intr_machdep.c

[61] Kod źródłowy FreeBSD - plik exception.S, http://fxr.watson.org/fxr/source/mips/mips/exception.S

[62] Obrazek - uproszczony schemat architektury von Neumanna, https://commons.wikimedia.org/wiki/File:Von_Neumann_Architecture.svg

[63] Obrazek - magistrala systemowa, https://commons.wikimedia.org/wiki/File:Computer_system_bus.svg

[64] Obrazek - architektura mostów północnego i południowego, https://commons.wikimedia.org/wiki/File:Motherboard_diagram.svg

[65] ANDREW S. TANENBAUM, HERBERT BOS, *Systemy operacyjne*, wydanie IV, rysunek 1.12

[66] Obrazek - przestrzeń konfiguracyjna PCI, https://commons.wikimedia.org/wiki/File:Pci-config-space.svg

[67] Marshall Kirk McKusick, George V. Neville-Neil, Robert N.M. Watson, *The Design and Implementation of the FreeBSD Operating System*, wydanie II, rysunek 8.12

[68] Marshall Kirk McKusick, George V. Neville-Neil, Robert N.M. Watson, *The Design and Implementation of the FreeBSD Operating System*, wydanie II, rysunek 8.13

[69] The Linux Documentation Project, *PCI*, `https://www.tldp.org/LDP/tlk/dd/pci.html`

[70] Marshall Kirk McKusick, George V. Neville-Neil, Robert N.M. Watson, *The Design and Implementation of the FreeBSD Operating System*, wydanie II, listing 8.15

[71] Podręcznik systemowy `module(9)`, `https://www.freebsd.org/cgi/man.cgi?query=module&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports`

[72] Podręcznik systemowy `malloc(9)`, `https://www.freebsd.org/cgi/man.cgi?query=malloc&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports`

[73] Podręcznik systemowy `driver(9)`, `https://www.freebsd.org/cgi/man.cgi?query=driver&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports`

[74] Podręcznik systemowy `sysinit(9)`, `https://www.freebsd.org/cgi/man.cgi?query=SYSINIT&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports`

[75] Podręcznik systemowy `device(9)`, `https://www.freebsd.org/cgi/man.cgi?query=device&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports`

[76] Podręcznik systemowy `devctl(8)`, `https://www.freebsd.org/cgi/man.cgi?query=devctl&sektion=8&manpath=FreeBSD+11.2-RELEASE+and+Ports`

[77] Podręcznik systemowy `devctl(3)`, `https://www.freebsd.org/cgi/man.cgi?query=devctl&sektion=3&manpath=FreeBSD+11.2-RELEASE+and+Ports`

[78] Podręcznik systemowy `sysctl(8)`, `https://www.freebsd.org/cgi/man.cgi?query=sysctl&sektion=8&manpath=FreeBSD+11.2-RELEASE+and+Ports`

[79] Podręcznik systemowy `sysctl(3)`, `https://www.freebsd.org/cgi/man.cgi?query=sysctl&sektion=3&manpath=FreeBSD+11.2-RELEASE+and+Ports`

[80] Podręcznik systemowy `devclass(9)`, `https://www.freebsd.org/cgi/man.cgi?query=devclass&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports`

[81] Podręcznik systemowy `bus_generic_attach(9)`, `https://www.freebsd.org/cgi/man.cgi?query=bus_generic_attach&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports`

[82] Podręcznik systemowy device.hints(5), https://www.freebsd.org/cgi/man.cgi?query=device.hints&sektion=5&manpath=FreeBSD+11.2-RELEASE+and+Ports

[83] Podręcznik systemowy kobj(9), https://www.freebsd.org/cgi/man.cgi?query=kobj&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports

[84] Podręcznik systemowy bus_space(9), https://www.freebsd.org/cgi/man.cgi?query=bus_space&manpath=FreeBSD+11.2-RELEASE+and+Ports

[85] Podręcznik systemowy queue(3), https://www.freebsd.org/cgi/man.cgi?query=queue&sektion=3&manpath=FreeBSD+11.2-RELEASE+and+Ports

[86] Podręcznik systemowy locking(9), https://www.freebsd.org/cgi/man.cgi?query=locking&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports

[87] Podręcznik systemowy ithread(9), https://www.freebsd.org/cgi/man.cgi?query=ithread&sektion=9&manpath=FreeBSD+10.0-RELEASE

[88] Podręcznik systemowy bus_setup_intr(9), https://www.freebsd.org/cgi/man.cgi?query=bus_setup_intr&manpath=FreeBSD+11.2-RELEASE+and+Ports

[89] Podręcznik systemowy critical_enter(9), https://www.freebsd.org/cgi/man.cgi?query=critical_enter&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports

[90] Podręcznik systemowy device_set_ivars(9), https://www.freebsd.org/cgi/man.cgi?query=device_set_ivars&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports

[91] Podręcznik systemowy device_get_softc(9), https://www.freebsd.org/cgi/man.cgi?query=device_get_softc&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports

[92] Podręcznik systemowy device_busy(9), https://www.freebsd.org/cgi/man.cgi?query=device_busy&sektion=9&manpath=FreeBSD+11.2-RELEASE+and+Ports