2.6. Dlaczego aplikacje zależą od systemu operacyjnego

W zasadzie aplikacje skompilowane w jednym systemie operacyjnym nie są wykonywalne w innych systemach operacyjnych. Gdyby były, świat byłby lepszym miejscem, a w wyborze systemu operacyjnego kierowalibyśmy się jego wygodą i własnościami, a nie tym, które aplikacje są w nim dostępne.

Opierając się na tym, co wcześniej powiedzieliśmy, możemy obecnie dostrzec część tego problemu – każdy system operacyjny rozporządza swoistym zbiorem wywołań systemowych. Wywołania systemowe są częścią zbioru usług świadczonych przez system operacyjny na użytek aplikacji. Gdyby nawet wywołania systemowe dało się w jakiś sposób ujednolicić, inne przeszkody utrudniałyby wykonywanie programów użytkowych w różnych systemach operacyjnych. Jeśli jednak używaliście kilku systemów operacyjnych, to zapewne z niektórych takich samych aplikacji dało się w nich skorzystać. Jak to możliwe?

Są trzy sposoby udostępnienia aplikacji do pracy w wielu systemach operacyjnych:

1. Można napisać aplikację w języku interpretowanym (takim jak Python lub Ruby), którego interpretery są dostępne w wielu systemach operacyjnych. Interpreter czyta kolejne wiersze programu źródłowego, wykonując zawarte w nich instrukcje za pomocą równoważnych im ciągów rozkazów z zestawu właściwego danej maszynie, i używa wywołań właściwych rodzimemu systemowi operacyjnemu. Cierpi na tym wydajność w stosunku do rdzennych aplikacji, a interpreter dostarcza tylko podzbiór cech każdego z systemów operacyjnych, ograniczając być może zbiory własności kojarzonych aplikacji.
2. Aplikację można napisać w języku, który uwzględnia maszynę wirtualną z wykonywaną aplikacją. Maszyna wirtualna jest częścią pełnego środowiska wykonawczego (RTE) danego języka. Przykładem tej metody jest Java. Java ma RTE zawierające ładowacz, weryfikator bajtokodu i inne składowe, które wprowadzają aplikację Javy do maszyny wirtualnej Java. To RTE zostało **przeniesione** (*ported*) do wielu systemów operacyjnych (lub opracowane dla nich od zera), poczynając od komputerów głównych, na smartfonach kończąc. Teoretycznie dowolna aplikacja Javy może pracować w takim środowisku wykonawczym, gdziekolwiek jest ono dostępne. Systemy tego rodzaju wykazują wady podobne do wad interpreterów, które omówiliśmy wcześniej.
3. Budowniczy aplikacji może wykorzystać standardowy język lub API, w którym kompilator generuje binaria w maszynowym i specyficznym dla systemu operacyjnego języku. Aplikacja musi być przeniesiona do każdego systemu operacyjnego, w którym będzie wykonywana. To przeniesienie może zajmować dużo czasu i trzeba je wykonywać dla każdej nowej wersji aplikacji, a w konsekwencji testować i czyścić z błędów. Być może najlepiej znanym przykładem jest POSIX API i jego zbiór standardów utrzymywania

zgodności kodu źródłowego między różnymi odmianami uniksopodobnych systemów operacyjnych.

Z teoretycznego punktu widzenia te trzy podejścia zdają się zapewniać proste rozwiązania problemu budowy aplikacji, które mogą działać w różnych systemach operacyjnych. Jednak ogólny brak mobilności aplikacji ma kilka przyczyn, a każda z nich sprawia, że opracowywanie aplikacji międzyplatformowych jest nie lada zadaniem. Dostarczane wraz z systemami operacyjnymi biblioteki zawierają na poziomie aplikacji interfejsy API wyposażone w takie możliwości jak interfejsy graficzne, więc aplikacja pomyślana tak, aby wywoływała jeden zbiór interfejsów API (powiedzmy, tych, które są dostępne pod systemem iOS w iPhonie Apple'a) nie będzie działać w systemie operacyjnym, który tych API nie ma (np. Android). Na niższych poziomach w systemie istnieją inne wyzwania, w tym następujące:

- Każdy system operacyjny ma jakiś format binarny aplikacji, który narzuca układ nagłówka, instrukcji i zmiennych. Te komponenty muszą się znajdować w pliku wykonywalnym w określonych miejscach zdefiniowanych struktur, aby system operacyjny mógł otworzyć plik, załadować aplikację i właściwie ją wykonać.
- Jednostki centralne mają różne zbiory rozkazów i tylko aplikacja zawierająca odpowiednie rozkazy może być poprawnie wykonana.
- Systemy operacyjne dysponują wywołaniami systemowymi, które umożliwiają aplikacjom zamawianie rozmaitych czynności, takich jak tworzenie plików i otwieranie połączeń sieciowych. Wywołania systemowe różnią się w różnych systemach operacyjnych pod wieloma względami, m.in. specyficznymi argumentami i ich uporządkowaniem, sposobem uaktywniania przez aplikację funkcji systemowych, ich numeracją i liczbą, znaczeniem i zwracanymi wynikami.

Jest kilka podejść pomocnych w pokonywaniu tych architektonicznych różnic, lecz nie dających kompletnych rozwiązań. Na przykład w Linuxie – i w prawie każdym systemie uniksowym – dla wykonywalnych plików binarnych przyjęto format ELF. Mimo że ELF stanowi powszechny standard w systemach Linux i UNIX, nie jest związany z żadną konkretną architekturą komputera, nie gwarantuje więc, że plik wykonywalny będzie działał na różnych platformach sprzętowych.

Interfejsy API, jak już wspomniano, określają pewne funkcje na poziomie aplikacji. Na poziomie architektonicznym stosuje się **binarny interfejs aplikacji** (*application binary interface* – **ABI**) definiujący sposób współpracy różnych komponentów kodu binarnego w danym systemie operacyjnym i danej architekturze. ABI określa szczegóły niskiego poziomu, w tym długość adresu, metody przekazywania parametrów do wywołań systemowych, organizację stosu w fazie wykonywania, binarny format bibliotek systemowych i rozmiar typów danych

– by wymienić tylko niektóre. ABI jest zazwyczaj zdefiniowany dla określonej architektury (istnieje na przykład ABI dla procesora ARMv8). Tak więc ABI jest odpowiednikiem API na poziomie architektonicznym. Jeżeli binarny plik wykonywalny został skompilowany i skonsolidowany zgodnie z konkretnym ABI, powinien dać się wykonać w różnych systemach udostępniających ten ABI. Ponieważ jednak konkretny ABI jest zdefiniowany dla określonego systemu operacyjnego działającego w określonej architekturze, interfejsy ABI w niewielkim stopniu przyczyniają się do zapewniania zgodności międzyplatformowej.

Podsumowując, wszystkie te różnice oznaczają, że dopóki interpreter, RTE lub binarny plik wykonywalny nie zostaną napisane i skompilowane w określonym systemie operacyjnym dla jednostki centralnej określonego typu (np. dla Intela lub ARMv8), aplikacja nie będzie działała. Wyobraźmy sobie, ile trzeba pracy, aby program w rodzaju przeglądarki Firefox działał pod systemami Windows, macOS, w różnych wydaniach Linuxa, w systemach iOS i Android, i jeszcze niekiedy na CPU o różnych architekturach.

2.7. Projektowanie i implementowanie systemów operacyjnych

W tym podrozdziale omówimy zagadnienia projektowania i implementowania systemu operacyjnego. Nie istnieją – rzecz jasna – gotowe rozwiązania takich problemów, lecz niektóre podejścia zostały sprawdzone w praktyce.

2.7.1. Założenia projektowe

Pierwszym zagadnieniem przy projektowaniu systemu jest określenie celów i specyfikacji systemu. Na najwyższym poziomie projekt systemu będzie w dużym stopniu zależał od wyboru sprzętu i typu systemu – czy ma to być system dla konwencjonalnego komputera biurkowego lub laptopa, czy ma być mobilny, rozproszony, czy pracujący w czasie rzeczywistym.

Pominąwszy poziom najwyższy, sformułowanie wymagań może być znacznie trudniejsze. Wymagania te można zasadniczo podzielić na dwie grupy: **cele użytkownika** i **cele systemu**.

Użytkownicy oczekują od systemu operacyjnego pewnych oczywistych cech. Powinien on być wygodny i łatwy w użyciu, łatwy do nauki, niezawodny, bezpieczny i szybki. Jest zrozumiałe, że ze względu na brak powszechnej zgody co do sposobu osiągania tak określonych celów, tego rodzaju specyfikacje nie są zbyt przydatne w projektowaniu systemu.

Podobny zestaw wymagań mogą sformułować osoby, których zadaniem jest taki system zaprojektować, utworzyć, utrzymywać i obsługiwać. System operacyjny powinien być łatwy do zaprojektowania, realizacji i pielęgnowania; powinien być elastyczny, niezawodny, wolny od błędów i wydajny. I znowu, tak postawione wymagania są niejasne i mogą być interpretowane przeróżnie.