Technologie informacyjne
przed 2 wykładem

Andrzej Giniewicz

06.03.2024

Po przeglądzie składni Pythona, przechodzimy do opisu podstaw działania systemów operacyjnych oraz pracy z powłoki tekstowej.

1 Najważniejszy program

Najważniejszym programem działającym na komputerze, jest system operacyjny. System operacyjny zajmuje się tym, abyśmy my jako użytkownicy i twórcy oprogramowania nie byli świadomi jego istnienia, lub byli minimalnie świadomi jego istnienia. Rolą systemu operacyjnego, jest przede wszystkim ukrycie przed nami detali sprzętowych. We wczesnych czasach komputerów PC, kiedy jednym z częściej stosowanych systemów na komputerach domowych był DOS, twórcy programów mieli bezpośredni dostęp do sprzętu. Powodowało to sytuacje takie, że musieli przygotowywać się na najróżniejsze konfiguracje sprzętowe, których na szczęście było wtedy niewiele. Niemniej jednak, instalując grę, musieliśmy z listy obsługiwanych przez nią kart dźwiękowych i graficznych wybrać odpowiednie. Dzięki takiej optymalizacji z jednej strony twórcy mieli dostęp do 100% możliwości, z drugiej niestety, kiedy wyszła nowa karta graficzna lub dźwiękowa, traciliśmy możliwość skorzystania z niej w grze. Podobne sytuacje miały miejsce w dowolnych aspektach zarządzania komputerem. Na szczęście jednak czasy DOSa się skończyły. Z czasem zaczęła pojawiać się koncepcja sterowników. Sterownik to program lub biblioteka, dostarczony przez producenta sprzętu. Sterownik integruje się z systemem operacyjnym i ukrywa przed nami fizyczne detale. Sterowniki często udostępniają swoją funkcjonalność za pomocą ogólnoprzyjętych interfejsów, czyli API. Był to bardzo ważny krok. Od teraz to już nie producent gry lub programu musiał martwić się, jakie na świecie są popularne karty dźwiękowe, tylko jakie są popularne interfejsy programistyczne. Najczęściej na systemie operacyjnym było jedno wiodące API do danej funkcjonalności, więc nie było to zadanie trudne. Taki proces ukrywania detali sprzętowych przed programami, nazywamy wirtualizacją. Programy dziś nie otrzymują dostępu do fizycznej pamięci, tylko do wirtualnej pamięci, zarządzanej przez system operacyjny. System zajmuje się przydzielaniem wirtualnych zasobów pomiędzy różne programy, dzięki czemu mogą je współdzielić i na przykład dwa programy mogą korzystać z Internetu, choć mamy jedną kartę sieciową.

Podsumowując ten historyczny wywód, głównym zyskiem z posiadania systemu operacyjnego jest to, że ukrywa on przed nami detale, abyśmy mogli się skupić na ogólnym działaniu. System dba o przydział zasobów, ale też bezpieczeństwo. Jeden użytkownik komputera nie powinien móc przeglądać plików innego użytkownika. Włamanie się na komputer z Internetu powinno być utrudnione, jeśli system działa odpowiedzialnie. Również powinno być trudno napisać działający złośliwy kod, który będzie działał na szkodę użytkownika. System w końcu powinien pozwolić użytkownikowi szybko uruchomić inne programy, przełączać się pomiędzy nimi — a poza tym, usunąć się w cień. Najlepiej, jeśli codziennie nie jesteśmy bombardowani masą powiadomień, które nas nie interesują, jeśli podczas nagrywania wykładu nie usłyszymy, że „baza wirusów została zaktualizowana" lub nie zobaczymy okienka mówiącego, że komputer został zaktualizowany i zostanie uruchomiony za 15 minut. Jeśli takie rzeczy się dzieją, denerwujemy się na system, bo przeszkadza nam w naszej pracy. Dlatego im bardziej jesteśmy świadomi istnienia systemu operacyjnego, tym dla niego gorzej.

Systemów operacyjnych jest dużo. Najpopularniejszym systemem na komputerach osobistych jest dziś Windows firmy Microsoft. Historia systemu Windows sięga 1985 roku. W tamtych czasach, aż do 1993 roku do Windows 3.11, była to graficzna nakładka na system MS-DOS, który funkcjonował od 1981 roku. Przez graficzną nakładkę, rozumiemy graficzny interfejs użytkownika działający jako program w systemie DOS. Kolejne wersje rozwijały się w dwóch kierunkach, systemu Windows opartego na DOS, o nazwach Windows 95, Windows 98, Windows Me; oraz systemu Windows NT napisanego od podstaw, o nazwach Windows NT 3, Windows NT 4, Windows 2000. Rodzina Windows NT była przeznaczona dla użytkowników profesjonalnych i na serwery, natomiast rodzina Windows 9x dla użytkowników domowych. Począwszy od systemu Windows XP i Windows Server 2003, oba jego warianty były oparte na kodzie Windows NT, co jest prawdą do dziś. Oznacza to, że obecne wersje systemu Windows nie są oparte na systemie DOS, ale ich kod źródłowy sięga roku 1993, kiedy system ten był dominujący.

Dużo wcześniej, bo w 1971 roku powstał system Unix. Był to system nowoczesny, nawet w porównaniu z późniejszymi systemami. Zawierał koncepcję wielu użytkowników, był o wiele bezpieczniejszy oraz powstawał z myślą o komunikacji sieciowej w czasach, gdy od powstania Internetu dzieliło nas dużo czasu. System Unix był produktem firmy Bell Labs, tej samej, w której powstał język C. Pod koniec lat 70-tych wydzielił się z niego system BSD, którego warianty funkcjonują i są dostępne do dzisiaj. Do najpopularniejszych systemów rodziny BSD należą macOS, FreeBSD, NetBSD, OpenBSD i DragonFly BSD. W latach 80-tych z Unixa również wydzielił się system PWB, na którego bazie powstała rodzina systemów System V. Z niej z kolei wywodzi się między innymi rozwijany do dziś system Solaris. Ponieważ powstało tak wiele wariantów systemu Unix, w 1988 roku wprowadzono standard POSIX, który ustalał wspólne API oraz zbiór narzędzi, które powinny być dostępne na każdym systemie Unix. Opublikowanie standardu spowodowało, że zaczęły powstawać również inne systemy, które były zgodne z POSIX, ale nie wykorzystywały kodu Unixa. Należy do nich głównie Linux powstały w 1991 roku, który choć formalnie nie należy do rodziny Unix, jest zgodny ze standardem POSIX, więc charakteryzuje je wiele wspólnych

cech. Obecnie na jądrze Linuxa działa bardzo dużo różnych systemów operacyjnych, w tym jeden z popularniejszych systemów mobilnych — Android.

Patrząc na daty dla najpopularniejszych systemów, których sięgają pierwsze wersje poszczególnych systemów, można wskazać 1971 rok dla macOS (początek rodziny Unix), 1991 rok (początek rodziny Linux) oraz 1993 rok (początek współczesnych systemów Windows). Aby różne narzędzia mogły współistnieć tyle lat, konieczna jest specjalizacja. Według statystyk[1], Windows obsługuje 15,9% stron internetowych, natomiast wiadomo, że Linux działa na 41,5% serwerów. Co na pozostałych 42,6%? Są strony działające na systemach rodziny Unix i Linux, które są na tyle zabezpieczone, że boty zbierające statystyki nie są w stanie tego jednoznacznie sprawdzić. Procent wykorzystania macOS na serwerach jest mniejszy niż 0,1%. Ogólnie, wiadomo, że 84,4% wszystkich stron internetowych działa na jakimś systemie związanym z Unixem lub Linuxem. Jeśli chodzi o systemy mobilne[2], 71,4% stanowi Android (oparty na Linuxie) a 27,8% iOS (oparty na macOS) — łącznie te dwa systemy stanowią 99,2% rynku urządzeń mobilnych. Sytuacja wygląda nieco inaczej dla komputerów osobistych[3], gdzie Windows posiada 72,2%, macOS 15,4% i Linux 4%. Na podstawie tych danych, zdecydowanie widać specjalizację systemów.

1. Na serwerze, najczęściej spotkamy Linuxa, potem Windowsa;
2. Na komputerze osobistym, najczęściej spotkamy Windowsa, potem macOS;
3. Na urządzeniu mobilnym najczęściej spotkamy Linuxa (wariant Android), potem macOS (wariant iOS).

Nieprzypadkowo, każdy z tych systemów wymieniony jest w tym rankingu dwa razy — gdyby jakiś dominował we wszystkich trzech kategoriach, prawdopodobnie wyparłby konkurencję, podczas gdy równowaga zapewnia zdrową rywalizację. W związku z tym — musimy godzić się z różnorodnością systemów operacyjnych i z tym, że w zależności od tego, czym się zajmujemy, spotkamy się z różnymi ich wariantami.

2 Jak system operacyjny widzi inne programy

Gdy każemy systemowi operacyjnemu uruchomić program, ładuje go do pamięci. Uruchomiony program jest przedstawiony jako proces. Każdy proces ma swój unikatowy numer, nazywany ID procesu (PID, ang. Process Identification). System operacyjny zajmuje się tym, aby każdy proces otrzymał odrębne zasoby, co stanowi ważne zabezpieczenie. Proces w systemie może utworzyć inne procesy, tak zwane procesy potomne — w efekcie w systemie tworzy się drzewo (jak drzewo genealogiczne) powiązanych ze sobą procesów. Proces potomny nazywamy dzieckiem procesu, który go utworzył. Proces, który utworzył dziecko,

[1] Statystyki dla serwerów oparte są na publicznych stronach internetowych `https://w3techs.com/technologies/overview/operating_system`, stan na 1 marca 2024.

[2] Statystyki dla systemów mobilnych pochodzą z `https://gs.statcounter.com/os-market-share/mobile/worldwide`, stan na 1 marca 2024.

[3] Statystyki dla systemów na komputery osobiste pochodzą z `https://gs.statcounter.com/os-market-share/desktop/worldwide`, stan na 1 marca 2023.

jest jego rodzicem. Komunikacja pomiędzy procesami jest powolna, ale w pełni bezpieczna. Proces może jedynie przekazać pewne informacje innemu procesowi[4]. Jeśli proces przestanie działać, można wymusić jego zamknięcie. Wymuszone zamknięcie procesu nazywamy jego zabijaniem.

2.1 Procesy osierocone

Zabicie procesu, który miał procesy potomne, powoduje ich osierocenie. Istnieją trzy różne strategie radzenia sobie w sytuacji osierocenia procesu. System operacyjny może zabić również procesy potomne, inny proces może je adoptować lub proces może pozostać procesem osieroconym. To ostatnie rozwiązanie może prowadzić do wycieków pamięci, zatem jest najrzadziej stosowane.

Sytuacja, w której system operacyjny zabija również procesy potomne, nazywa się wyłączaniem kaskadowym (ang. cascading termination). Jest ono obecne na niektórych systemach operacyjnych jako jedyne możliwe rozwiązanie, natomiast na innych zwykle jest dostępne jako opcja. Biorąc pod uwagę, jakie są nasze potrzeby podczas zajęć z technologii informacyjnych, najczęstszymi przypadkami będzie kończenie zadania w powłoce tekstowej poprzez wysłanie sygnału SIGINT (przerwij, z angielskiego „signal interrupt", gdy naciśniemy Ctrl-C) albo SIGHUP (rozłączenie, z angielskiego „signal hang up", gdy zamykamy okienko z powłoką tekstową lub wpisujemy „exit" w BASHu). W obu tych przypadkach domyślnym zachowaniem aplikacji jest propagowanie tych sygnałów i kaskadowe kończenie procesów potomnych. To samo dzieje się na większości systemów operacyjnych, gdy kończymy zadanie w graficznym menadżerze zadań — system traktuje grupę procesów jako jedno zadanie i zamyka je kaskadowo.

Zachowanie polegające na adopcji jest domyślnym na systemach operacyjnych z jądrem Linux w wersji 3.4 lub większej — na systemie tym, gdy proces kończy działanie, sam staje się procesem zombie, natomiast wszystkie jego procesy potomne, procesami osieroconymi. Proces zombie istnieje w systemie, do czasu aż nie zostanie zżęty (ang. reaping). Obowiązek zżęcia procesu zombie spoczywa na jego rodzicu. Jeśli rodzic procesu, który przestaje działać, sam już nie działa, mogłoby to prowadzić do wycieku zasobów i uniemożliwienia tworzenia nowych procesów. W tym celu wprowadzono pojęcie adopcji. Każdy osierocony proces jest adoptowany przez najbliższy w hierarchii (dziadek lub któryś z pradziadków wyżej) proces oznaczony w systemie jako podżniwiarz (subreaper). Podżniwiarz przejmuje osierocone procesy i żnie je, gdy te staną się zombie. Jeśli w hierarchii nie ma podżniwiarza, przejmuje je proces o numerze 1 — korzeń drzewa i jedyny proces bez rodzica — proces ten pełni rolę żniwiarza i przejmuje osierocone procesy, które nie podlegają pod żadnego podżniwiarza.

[4]Komunikacja międzyprocesowa ma swój odpowiednik w wysyłaniu listów — trwa dłużej, ale zapewnia prywatność i kontrolę nad tym, jakie informacje są udostępniane.

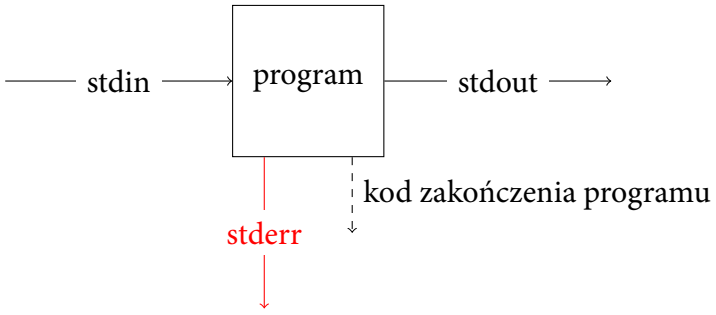
2.2 Wiele wątków w jednym procesie

Procesy mogą mieć wiele wątków. Wątek jest podobny do procesu potomnego, ale nie jest oddzielnym bytem, tylko działa w ramach procesu tworzącego. Oznacza to, że zasoby są współdzielone przez wszystkie wątki jednego procesu. Wobec tego są dużo szybsze, ale o wiele bardziej niebezpieczne[5] i niedeterministyczne, czyli mogące cechować się pewną losowością co do kolejności wykonywania operacji pomiędzy wątkami.

Rdzeń procesora może jednocześnie wykonywać jeden wątek jednego procesu[6]. System operacyjny zajmuje się tym, aby wszystkie wątki wszystkich procesów były obsługiwane naprzemiennie, dzięki czemu użytkownik ma wrażenie programów działających równolegle. Techniki programowania obejmujące komunikację między procesami oraz współdzielenie pamięci przez wątki, nazywamy właśnie programowaniem równoległym lub współbieżnym.

3 Program z lotu ptaka

Jeśli popatrzymy na program z lotu ptaka, pomijając całkowicie jego budowę, możemy myśleć o nim jak o czarnym pudełku wykonującym pewne zadanie. W systemach opartych na Unix i Linux główną ideą jest to, żeby pudełka wykonywały jak najmniej zadań, a najlepiej były odpowiedzialne za jedno zadanie. Wtedy łatwiej kontrolować to, czy są poprawnie zaprogramowane. I tak na przykład to dlatego komendy `pwd`, `cd` oraz `ls` są oddzielne — każda z nich robi jedną rzecz, choć razem są często używane do nawigacji po strukturze katalogów na dysku twardym. Każde czarne pudełko w pewien sposób komunikuje się z resztą świata. Komunikacja ta odbywa się za pomocą strumieni oraz tak zwanego kodu zakończenia programu w postaci jednej liczby całkowitej.



Na rysunku powyżej zaznaczone zostały trzy standardowe strumienie:

stdin standardowy strumień wejścia, domyślnie w terminalu jest to tekst z klawiatury, w Pythonie funkcja `input` odczytuje dane ze strumienia wejściowego,

stdout standardowy strumień wyjścia, domyślnie w terminalu jest to tekst wyświetlany na ekranie, w Pythonie funkcja `print` zapisuje dane do strumienia wyjściowego,

[5] Współdzielona pamięć przez wątki przypomina trochę lodówkę w akademiku — jeśli coś zapiszemy pod jakimś adresem, wystarczy się obejrzeć i inny wątek mógł już z tego skorzystać.

[6] Niekiedy jeden fizyczny rdzeń dzielony jest na więcej niż jeden rdzeń logiczny. Wtedy system operacyjny widzi więcej rdzeni, niż fizycznie znajduje się w procesorze. Taka technologia nazywa się Hyper-Threading.

stderr standardowy strumień błędów, domyślnie w terminalu jest to tekst wyświetlany na ekranie, w Pythonie trzeba skorzystać z dodatkowych parametrów funkcji `print`, aby zapisać tekst na standardowe wyjście.

Strumień `stderr` jest przeznaczony do komunikowania błędów w działaniu programu w sposób czytelny dla użytkownika. Do komunikacji błędu działania programu w sposób czytelny dla komputera, służy kod zakończenia programu. Kod zakończenia programu to liczba, którą interpretujemy w następujący sposób:

wartość zero oznacza, że program zakończył się poprawnie,

wartość niezerowa oznacza, że program zakończył się z błędem.

Interpretacja konkretnej wartości kodu błędu jest zależna od programu i powinna być sprawdzona w dokumentacji, choć często jedyny kod błędu, jaki jest zwracany to `1`. W języku Python, jeśli program zakończy się bez wyjątków, zostanie zwrócona wartość zero. Jeśli zakończy się z wyjątkiem, zostanie zwrócona wartość niezerowa. Aby wcześniej zakończyć działanie programu, możemy użyć funkcji `exit` z modułu `sys`. W funkcji tej podajemy kod zakończenia programu jako liczbę. Wartość domyślna to zero, czyli `sys.exit()` zakończy program bez błędu a przykładowo `sys.exit(-11)` zakończy program z kodem zakończenia programu ustawionym na -11.

4 Przekierowanie strumieni i rury

Każdy strumień posiada swój numer będący liczbą naturalną. BASH przydziela numer 0 do `stdin`, numer 1 do `stdout` oraz numer 2 do `stderr`. BASH rezerwuje dla użytkownika również numery od 3 do 9, natomiast dwucyfrowe i większe numery strumieni są przeznaczone dla plików otwieranych przez programy. Domyślnie strumień zero służy do wczytywania danych z klawiatury a strumienie 1 i 2 do wypisywania danych na ekranie. Komenda `echo` wypisuje informacje na ekranie.

```
echo ”Hello World”
```

Jeśli chcemy, możemy przekierować strumień `stdout` do pliku za pomocą `>` (nadpisuje plik jeśli istniał) lub `>>` (dopisuje do pliku).

```
echo ”Hello World” > przywitanie.txt
echo ”Hello to you too!” >> przywitanie.txt
```

Po wykonaniu tych komend na ekranie się nic nie pojawiło, ale powstał plik o nazwie `przywitanie.txt`. Zawartość pliku możemy wyświetlić komendą `cat`.

```
cat przywitanie.txt
```

Za pomocą komendy `wc` możemy sprawdzić, ile linii, słów i znaków jest w `stdin`. Po uruchomieniu komendy

```
wc
```

na ekranie nic się nie pojawi. Nie ma też znaku zachęty. Możemy jednak pisać. Napiszmy kilka linijek, na przykład

```
Hello World
Hello to you too!
```

Po drugiej linii wciśnijmy Enter. Nic się nie stanie, ponieważ klawiatura jest cały czas przekierowana do aplikacji, która czeka na kolejne linie tekstu. Aby wskazać terminalowi zakończenie strumienia, naciskamy skrót klawiaturowy CTRL-D. W odpowiedzi pojawią się trzy liczby, kolejno liczba linii, liczba słów i liczba znaków. Jeśli zdecydujemy, że chcemy przerwać komendę bez kontynuowania, możemy nacisnąć CTRL-C.

Pisanie tekstu w ten sposób nie jest najwygodniejsze, ale mamy ten sam tekst zapisany w pliku. Możemy przekierować strumień plik do strumienia wejściowego za pomocą symbolu <.

```
wc < przywitanie.txt
```

Przekierowanie strumienia do pliku, jeśli chcemy tylko policzyć liczbę słów i linii wydaje się zbędne. W takiej sytuacji możemy użyć rury, czyli symbolu |. Symbol rury powoduje podłączenia wejścia komendy po prawej do wyjścia komendy po lewej.

```
echo ”Hello World” | wc
```

Jeśli chcemy uruchomić kilka komend jedna po drugiej, możemy je oddzielić średnikami.

```
echo ”Hello World”; echo ”Hello to you too!”
```

Niestety, jeśli spróbujemy przekierować wyjście do komendy `wc`, tylko druga z komend zostanie przekierowana, na ekranie pojawi się „Hello World” oraz statystyki dla drugiego tekstu. Aby przekierować strumień z obu komend, musimy użyć nawiasów klamrowych

```
{ echo ”Hello World”; echo ”Hello to you too!”; } | wc
```

Zwróćmy uwagę, że w nawiasach klamrowych spacje wokół nich są istotne, dodatkowo ostatnia komenda `echo` również musi kończyć się średnikiem.

Przy łączeniu kilku komend za pomocą średnika lub rury, ignorowane są kody zakończenia programu oprócz ostatniego. Oznacza to, że jedyny kod zakończenia programu w przykładzie powyżej, będzie pochodził z komendy `wc`, niezależnie od tego, czy komendy `echo` zakończą się z kodem zero, czy z niezerowym kodem (kodem błędu).

4.1 Bardziej złożone przekierowania

W przykładach powyżej nie modyfikowaliśmy strumienia `stderr`, wszystko, co do niego trafia, wciąż ląduje na ekranie. Bardzo możliwe, że chcielibyśmy przekierować właśnie strumień błędów do pliku, aby po zakończeniu działaniu programu przejrzeć błędy, ale żeby w trakcie działania programu nie przeszkadzały nam w analizie wyników ze `stderr`.

Operatory `<`, `>` oraz `>>` to w rzeczywistości skróty dla `0<`, `1>` oraz `1>>`, gdzie `0` i `1` to odpowiednio `strin` oraz `stdout`. Aby przekierować `stderr`, czyli strumień numer `2`, do pliku, musimy użyć komendy `2>` lub `2>>` (jeśli chcemy dopisywać). W jednej komendzie może pojawić się wiele przekierowań.

```
wc < input.txt > wyniki.txt 2>> error.log
```

W hipotetycznym kodzie powyże zawartość pliku `input.txt` jest przekierowana do wejścia komendy `wc`, wyniki trafiają do pliku `wynik.txt` (za każdym wywołaniem nowego), natomiast błędy są doczepiane do pliku `error.log`.