

**Procesy** – Oddzielne programy, działające niezależnie od siebie

**Wątki** – Ścieżka wykonania programu w obrębie danego procesu

**Zasoby** – To, do czego ma dostęp program i z czego korzysta

Głównymi motywacjami do wprowadzenia procesów i wątków było:

* Odpowiedni podział przestrzeni adresowej między procesami
* Utworzenie abstrakcji dla każdego z procesów, dzięki której każdy proces korzysta z „własnej” przestrzeni adresowej
* W przypadku procesów dużo „płacimy” za tworzenie i niszczenie procesów i zmiany kontekstów. Wątki są tańsze, wszystkie wątki procesu dzielą między sobą tę samą pamięć, nie musimy przełączać kontekstu i alokować pamięci tak jak w przypadku nowego procesu.
* Dzięki wątkom łatwiej jest nam rozdzielić obliczenia na kilka procesorów
* Dzięki procesom i wątkom nie musimy robić zadań po kolei, a przeplatać je ze sobą (daje to lepszą wydajność i złudzenie wykonywania wielu operacji w tym samym czasie)
* Zastosowanie wątków i procesów jest bardziej responsywne (możemy jakieś operacje wykonywać w tle, dzięki czemu nie musimy czekać aż zakończy się dane zadanie w celu rozpoczęcia drugiego)

**Różnice**:

Oddzielne dla każdego **procesu**:

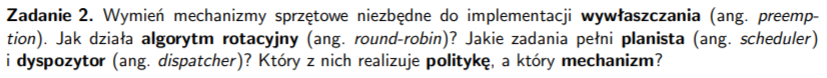
* Przestrzeń adresowa
* Zmienne globalne
* Uchwyty do plików
* Procesy-dzieci
* Sygnały

Oddzielne dla każdego **wątku**:

* Program counter
* Rejestry
* Stos
* Stan

Wątki mają wspólne:

* Przestrzeń adresowa,
* Kod
* Dane
* Zasoby systemowe (otwarte pliki, sygnały)



**Wywłaszczanie** – Zawieszanie aktualnie wykonywanych zadań i oddanie kontroli systemowi operacyjnemu bez kooperacji zadań, które zostają zawieszone. Dzięki temu system operacyjny ma kontrolę nad wykonywanymi zadaniami i planowaniem wykonywania.

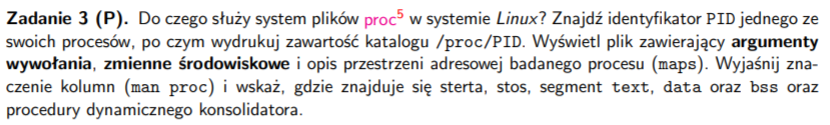
**Algorytm rotacyjny** – Algorytm, który jest używany do wybierania procesów, które mają być aktualnie wykonane. Procesy są dodawane do kolejki FIFO. Proces jest pobierany z kolejki, przydzielany jest mu czas wykonania (kwant czasu). Jeśli nie zdąży się cały wykonać w przydzielonym czasie, jest zawieszany i dodawany na koniec kolejki. Potem kolejny element jest pobierany, przydziela mu się kwant czasu itd.

**Planista** (scheduler) – Specjalne oprogramowanie systemowe, które planuje wykonywanie procesów po sobie. Głównym zadaniem jest wybór procesów, które mają być wznowione w danym czasie.

**Dyspozytor** (dispatcher) – Jego działanie następuje po planiście. Kiedy planista wybierze process, dyspozytor umieszcza go w odpowiednim miejscu, aby był gotowy do wykonania i oddaje kontrolę procesowi.

**Politykę** (round-robin) realizuje **planista**, **mechanizm** realizuje **dyspozytor**.

* Sposób na odzyskanie kontroli przez system operacyjny, zawieszając wykonywanie zadania. Przydaje się do tego „zegar przerwania”(?) oraz przerwania systemowe. Podczas przerwania aktualne zadanie zostaje „zawieszone” i po obsłużeniu przerwania, jest wznawiane (te lub inne, decyduje o tym SO). Zegar jest potrzebny, ponieważ jeśli dane zadanie nie wywołuje przerwań systemowych, mogłoby działać wiecznie i SO nigdy mógłby nie odzyskać kontroli.
* Sposób na zapobieganie wyłączenia powyższej funkcji. Jeśli mechanizm wywłaszczania nie może zostać odizolowany od zadania, któreś zadanie mogłoby zwyczajnie wyłączyć mechanizm i działać wiecznie.



System plików **proc** w systemie Linux zawiera w sobie informacje o przestrzeni adresowej procesów. Możemy tam znaleźć m.in informacje o tym pod jakimi adresami leży sterta, stos czy sekcje procesu. Proc to specjalny “wirtualny” system plików. Zawiera mnóstwo informacji o procesach uruchomionych w systemie. Nie zawiera „realnych” plików, lecz informacje o uruchomieniu.

**Argumenty wywołania** – Argumenty, z jakimi został wywołany program. Znajdują się w /proc/[pid]/cmdline.

**Zmienne środowiskowe** – Ich zapis dla danego procesu znajduje się w /proc/[pid]/environ. Są to takie zmienne globalne dla programu, które ułatwiają dostęp do ścieżek czy plików. Przykład:

USERNAME=debian. Pozwalają nam na odwoływanie się w łatwy sposób np. do nazwy użytkownika.

Wywołujemy:

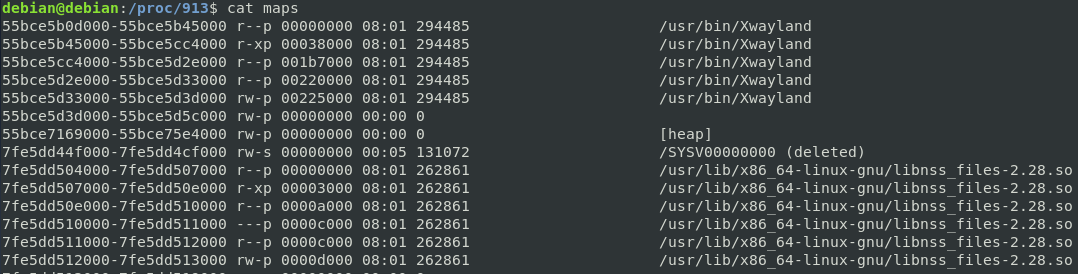
ps -eo user,pid,ppid,pgid,tid,pri,stat,wchan,cmd | grep X

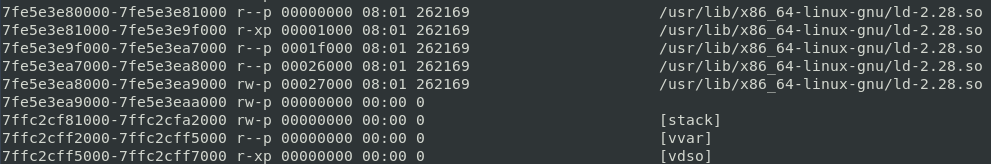


Znaleźliśmy 2 procesy, w tym drugi to nasz proces, który właśnie wywołaliśmy powyższą komendą. Proces górny o pid 913 to X-serwer. Zbadamy ten proces.

Cd /proc/913

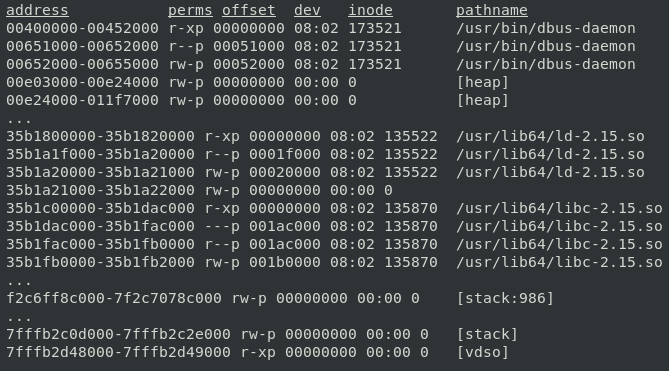
Cat maps





Możemy tutaj znaleźć zakres adresów, pod którymi leży np. stos procesu czy sterta. Możemy się także dowiedzieć czy dany fragment pamięci jest read-only, executable czy nie.

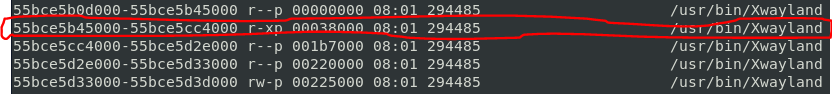
Wywołując polecenie man dla proc możemy dowiedzieć się co oznaczają poszczególne kolumny:



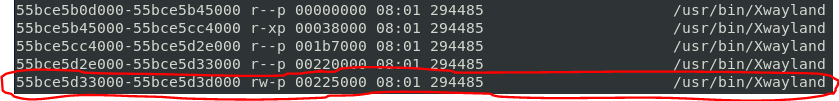
Mamy od lewej:

* Address - Adres danego elementu
* Perms - Uprawnienia (read/write/execute/private,shared). Jeśli process próbuje skorzystać z fragmentu pamięci w sposób zablokowany przez uprawnienia, dostajemy Segmentation Fault.
* Offset - Położenie danego elementu względem początku mapowanego pliku (jeśli jest mapowanie, jeśli nie – 0)
* Dev – Jeśli region był mapowany z pliku, to jest oznaczenie gdzie znajduje się ten plik w kodzie szesnastkowym
* Inode – Numer pliku jeśli region był mapowany z pliku
* Pathname – ścieżka do pliku, który jest zmapowany na fragment pamięci. Dla anonimowych regionów to miejsce jest puste. Poza tym mamy też specjalne oznaczenia: [stack], [heap] oraz [vdso]. Vdso – vitrual dynamic shared object – używany przez przerwania systemowe, aby wywołać jądro. [vvar] – zmienne jądra.

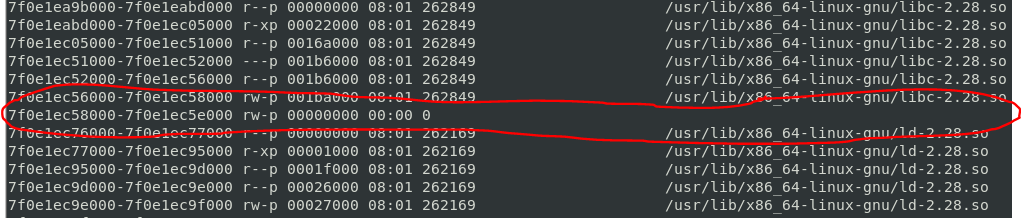
Stertę i stos widzimy na załączonych zrzutach ekranu. Sekcja text to sekcja, którą możemy czytać i wykonywać, ale nie możemy do niej zapisywać. Czyli uprawnienia: r-xp



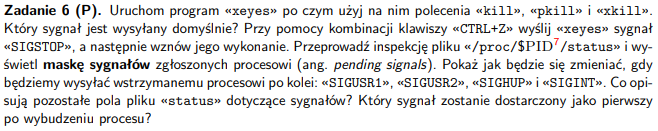
Sekcja data jest w miejscu oznaczonym uprawnieniami rw-p. Tą sekcję możemy czytać i zapisywać, ale nie możemy wykonywać.



Sekcję bss da się łatwo zauważyć z uwagi na pustą kolumnę ze ścieżką:



(tutaj akurat sekcja bss dla libc-2.28.so)

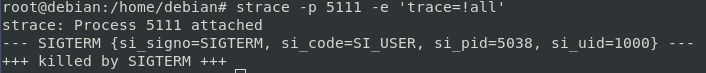


Uruchamiam w terminal process xeyes. Poszczególne sygnały wysyłane do tego programu będę śledził w innym terminalu poleceniem:

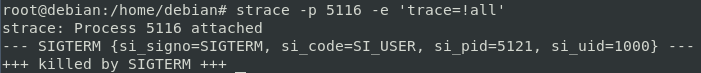
Strace -p [pid] -e ‘trace=!all’

To odfiltruje mi wszystko co zbędne i wyświetli tylko sygnały, które wysyłam do programu.

* Kill powoduje SIGTERM



* Pkill także powoduje SIGTERM

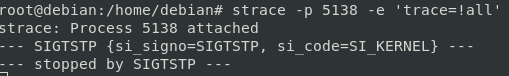


* Xkill powoduje wypisanie: exited with 1. SIGHUP?





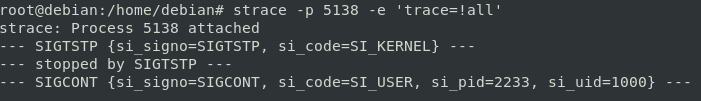
Uruchamiam po raz kolejny xeyes i wysyłam za pomocą CTRL + Z sygnał do procesu:



Wznawiam wykonanie procesu za pomocą polecenia:

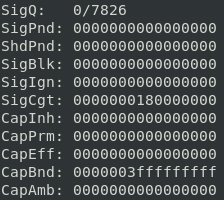
Fg xeyes

Strace pokazał mi nowy wpis, sygnał o kontynuowaniu procesu:

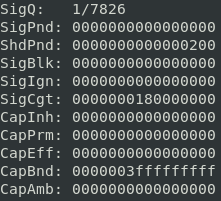
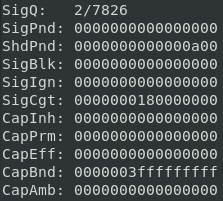
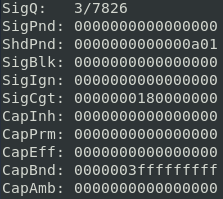
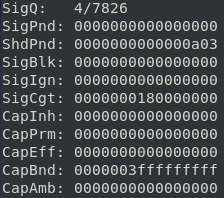


Dokonuję inspekcji pliku /proc/[pid]/status. Przyjrzymy się wpisom zaczynającym się od „Sig” (to są nasze **Maski Sygnałów**).

W tym momencie wpisy wyglądają tak:



Teraz po kolei wysyłam procesowi sygnały: SIGUSR1, SIGUSR2, SIGHUP, SIGINT. Maska zmienia się następująco:

Jak widzimy, zmienia się pole „ShdPnd” oraz „SigQ”. Pole „ShdPnd” to maska bitowa zapisująca sygnały, które czekają na wysłanie do zatrzymanego procesu. Pole „ShdPnd” to licznik, który wskazuje ile sygnałów jest w kolejce to wysłania do procesu. Pozostałe pola:

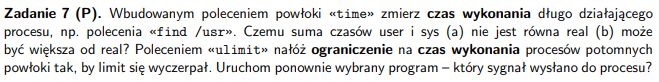
SigPnd – Signal Pending – Sygnały czekające na obsłużenie

SigBlk – Signal Blocked – Sygnały zablokowane (proces nie jest w tym momencie gotowy, aby je otrzymać). Jeśli jakikolwiek z takich przyjdzie do procesu, zostanie wrzucony do pending.

SigIgn – Signal Ignored – Sygnały, które nie zabiją procesu

SigCgt – Signal Caught – Sygnały “złapane”, czyli sygnały, które proces potrafi obsłużyć

Po przywróceniu procesu jako pierwszy zostanie wysłany sygnał **SIGHUP**.



Wykonuję polecenie: time find /usr. Wynik polecenia to:



‘real’ to całkowity czas wykonania polecenia. Czyli tyle, ile zajęło sumarycznie wykonanie polecenia find.

‘user’ to czas, jaki procesor spędził w „user-mode”.

‘Sys’ to czas, jaki procesor spędził w “kernel-mode”.

Przyczyny tego, że suma się nie zgadza:

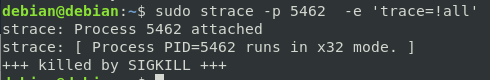
* Jeśli User+sys < real

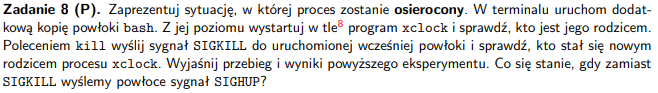
Jeśli możemy podzielić pracę na kilka wątków, łączne czasy user + sys zmierzą nam to ile czasu spędziliśmy w tych trybach obsługując zadanie, przy czym czas ‘real’ będzie mniejszy (bo mogliśmy wykonać kilka rzeczy na raz).

* Jeśli User+sys > real

Czasy „user” oraz „sys” nie uwzględniają czasu uśpienia procesu (np. podczas czekania na dane lub czas procesora). Polecenie find /usr intensywnie pobiera dane z dysku oraz wypisuje je na ekran (w dużej mierze opiera się na operacjach wejścia/wyjścia). Poza tym, działa dość długo (drzewo katalogów /usr jest dość duże), więc czas uśpienia tego procesu będzie zauważalny przez dużą dysproporcję w czasach.

Ustawiłem poleceniem ulimit -t 15 maksymalny czas CPU dla procesu na 15 sekund. Użyłem polecenia z zadania 9 (polecenie z tego zadania wykonywało się za szybko): echo „” | awk ‘{for (;;) {}}’. W osobnym terminalu uruchomiłem polecenie strace dla działającego w tle polecenia. Proces został zabity poprzez wysłanie sygnału SIGKILL.



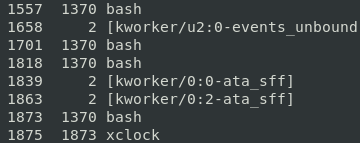


**Proces osierocony** – Każdy proces w systemie ma swojego rodzica (poza init). Proces, którego rodzic został zakończony lub zabity to Orphan (sierota). Takie procesy są dowiązywane do procesu init w momencie zabicia rodzica.

Uruchamiam konsolę i w niej *xclock&.*

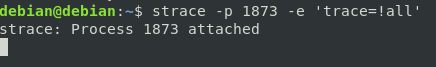


Sprawdzam pid konsoli, która uruchomiła xlock: *ps -eo pid,ppid,cmd*



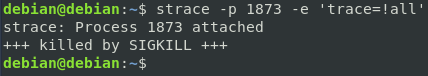
Widzimy, że szukanym pidem jest 1873 (jego ppid to pid basha nad nim, więc rodzicem jest bash o pid 1873).

W innej konsoli zaczynam obserwację sygnałów dla pid 1873:

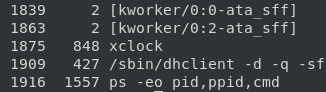


W innej konsoli wysyłam procesowi sygnał SYGKILL:

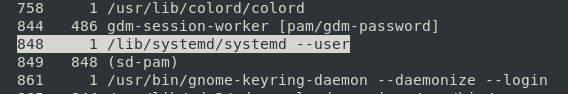




Jak widzimy, proces terminala został zabity. Jednak xclock nadal działa w tle. Zobaczmy jego ppid:



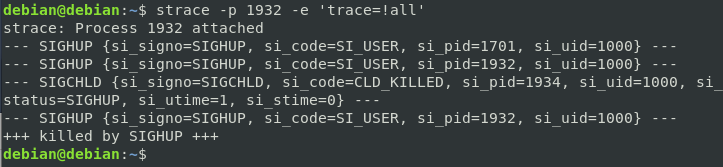
Widzimy, że ma ppid = 848. Pid 848 to proces:



Jest to taki „delegat”(?) do zarządzania różnymi procesami użytkownika. Jego rodzic to init. Czyli pokrywa się to z naszą teorią, osierocony proces jest dzieckiem init (tutaj nie bezpośrednio).

Próbujemy teraz z sygnałem SIGHUP:





Jak widzimy, terminal został zabity, a wraz z nim jego dziecko – xclock.