SO Lista 3

**Zad1**

Process memory layout (The linux programming interface):

Text: the instructions of the program.

Data: the static variables used by the program.

* Initialized (global i static)
* Uninitialized ( global i static, bss! (block started by symbol)

Odzielone ponieważ wystarczy trzymać na dysku tylko ile tego jest później wstawić same zera

Heap: an area from which programs can dynamically allocate extra memory.

Stack: a piece of memory that grows and shrinks as functions are called and

return and that is used to allocate storage for local variables and function call

linkage information.

Kernel

Stack

Memory Mapping

Heap

BSS

Data

Text

puste

Procesowi zostaje przyznana pamięć wirtualną do której tylko ma dostęp.

Na samej górze znajdują się funkcje jądra podłączone do procesu, jednak użytkownik nie ma do nich dostępu.

Na górze mapy pamięci procesu znajdują się argumenty wywołania procesu i zmienne środowiskowe.

Używamy pmap z -x dla nazw kolumn hehe

Address: start address of map

Kbytes: size of map in kilobytes

RSS: resident set size in kilobytes

Dirty: dirty pages (both shared and private) in kilobytes

Mode: permissions on map: read, write, execute, shared, private (copy on write)

Mapping: file backing the map, or '[ anon ]' for allocated memory, or '[ stack ]' for the program stack

Offset: offset into the file

Device: device name (major:minor)

Obraz (process image) dane wymagane do wczytania (lub alokowania (stos, sterta)) do pamięci aby można było uruchomić program

* Text (kod)
* Data (jak w layout)
* Stack
* Heap
* Process control block

Wszystko jest wczytywane leniwie, po page fault

$ pmap $(pgrep xeyes)

Address Kbytes RSS Dirty Mode Mapping

0000000000400000 16 16 0 r-x-- xeyes

0000000000400000 0 0 0 r-x-- xeyes < - - kod

0000000000603000 4 4 4 r---- xeyes

0000000000603000 0 0 0 r---- xeyes < - - process control block

0000000000604000 4 4 4 rw--- xeyes

0000000000604000 0 0 0 rw--- xeyes < - - argumenty i zmienne srodowiskowe?

Stos to [stack]

Sterta to [anon]

**Zad2**

# Blok kontrolny procesu - struktura danych zawierająca wszystkie istotne

informacje o nim

# Uchwyt do zasobu - struktura dająca dostęp do pliku

488: struct filedesc \*p\_fd; /\* (b) Open files. \*/

# Atrybuty - ? nie wiem które to, obstawiam że UID i tego typu rzeczy

487: struct ucred \*p\_ucred; /\* (c) Process owner's identity. \*/

# Blok kontrolny wątku - analogicznie do procesu

206: struct proc \*td\_proc; /\* (\*) Associated process. \*/<Paste>

Współdzielą wszystko to co jest w procesie?

**Zad3**

Dostępnych jest sześć stanów procesu: gotowy, wykonywany, przerywalny, nieprzerywalny, zatrzymany i zombie.

Proces,który jest aktywny, to proces gotowy lub wykonywany. Aktywny gość może być przerzucany przez zarządcę/dyspozytora pomiędzy gotowością a wykonywaniem, bo wykonywany = akurat zajmuje czas procesora (proces czekający na odczyt z dysku czy z I/O nie jest ani gotowy, ani wykonywany , o tym za chwilę).

Przerywalny proces to taki, który czeka na sygnał/zdarzenie, np. otwarta konsola może być w tym stanie, kiedy czeka na komendy do wpisania przez użytkownika; jeśli jakiś druk czeka na zwolnienie drukarki, to też jest w tym stanie; gra gotowa do odpalenia która czeka, aż Steam się załaduje i powie o tym grze.

Nieprzerywalny proces różni się tym od przerywalnego, że czeka na pewną akcję wykonaną bezpośrednio przez sprzęt - odczyt z dysku twardego, odczyt z płyty DVD, zrobienie grzanek w tosterze czy przesłanie jakiegoś protokołu z modemu.

Synchroniczne i asynchroniczne są zdarzenia a nie stany.

Synchroniczne:

* I/O
* Czekanie na inny proces

Asynchroniczne:

* sygnał

Zatrzymany proces może być wznowiony tylko po pozytywnej akcji jakiegoś innego procesu. Na Windowsie 7 często wyświetla się okienko pod tytułem “program medieval2.exe przestał działać” i to znaczy, że właśnie ten proces czy szereg procesów z programu medieval2.exe został zatrzymany. Te procesy mogą zostać wznowione po pracy jakiegoś debuggera albo innego cudaka. ( z tym przestał działać to raczej zombie, debbuger może stopować i wznawiać debbugowany proces)  
  
Proces zombie już nie żyje (znaczy został zakończony), ale jego struktura dalej znajduje się na liście procesów - chyba chodzi o to, żeby system uznawał ten proces za jeszcze działający. Być może jakiś inny proces, który jeszcze działa, potrzebuje widzieć zombie jako jeszcze aktywny proces czy coś. (Np. do zebrania informacji o tym procesie PCB)

Jeszcze raz: nieprzerywalny proces czeka na sprzęt, przerywalny na inny proces/użytkownika/dostępność zasobów.

<http://szymonwieloch.pl/synchroniczne-vs-asynchroniczne-operacje-wejscia-wyjscia/>

Nie mogę nigdzie znaleźć dokumentacji dotyczącej tego, kto wykonuje przejścia, więc trzeba się pewnie logicznie tego domyślić.  
  
Jądro systemu operacyjnego: tu mamy zarządcę/dyspozytora, czyli przełączanie procesów aktywnych pomiędzy gotowością a wykonywaniem. Oczywiście jądro może tworzyć procesy. Jak najbardziej odpowiada za terminację i zatrzymywanie procesów.

Kod sterowników: najprostsze, bo sterowniki odpowiadają za przełączanie pomiędzy procesem aktywnym a asynchronicznym (to te procesy czekające na sprzęt). Zawiadamia o zwolnienieniu lub blokowaniu I/O.  
  
Proces użytkownika i admina: (jądro tworzy procesy) Może wysyłać sygnały (user swoim, admin wszystkim), w tym stopować procesy(ctr + z). Yield oddaje cpu innym wątkom.

Obrazek: 215 str. W pdf

**Zad4**

Hierarchia procesów oznacza, że procesy tworzą drzewo.

ps (pokazuje procesy powiązane z terminalem, z którego została wywołana)

ps -A (pokazuje wszystkie procesy)

ps -a (wszystkie powiązane z terminalem, z którego została wywołana poza właścicielami grup procesów (właścicielem jest proces bash))

ps -C cmd (filtruje po nazwie procesu)

ps -u user (filtruje po nazwie właściciela procesu)

Wyświetlenie informacji dla wybranego procesu:

ps -p 1 -o pid,ppid,uid,pgid,comm

Właścicielem procesu init jest proces “0”, co oznacza pewnie brak właściciela.

Znaki w kolumnie STAT:

D nieprzerywalne spanie (uninterruptible sleep)

R chodzi

S przewarywalne spanie (interruptible sleep)

T zatrzymany

X dead (should never be seen)

Z zombie

Dodatkowe znaki:

< wysoki priorytet

N niski priorytet

L has pages locked into memory (for real-time and custom IO)

s lider grupy procesów

l jest wielowątkowy

+ jest w grupie procesów pierwszoplanowych

Wątki jądra są synami procesu “kthreadd”, więc ich PPID jest równy 2. (H przy ps pokazuje wątki)

Uruchamiając pstree wątkami powinny być nazwy w {} wyświetlonego drzewa.

**Zad5**

Sygnał - mały int, powiadomienie procesu że wystąpiło wydarzenie

Rola:

* software interrupts
* Procesy mogą przesyłać do siebie sygnały (prymitywna forma komunikacji)
* Można zdefiniować w swoim programie “signal handler” i wykonać w ten sposób pewien kod po otrzymaniu sygnału np. przed końcem życia procesu
* Debuggery używają, mogą np. Zatrzymać program, jeśli proces otrzyma dany sygnał

Sygnały z jądra:

* Hardware exception : Sprzęt wykrył błąd i powiadamia jądro. Jądro wysyła sygnał do odpowiedniego procesu (div 0; brak dostępu do pamięci; malformed machine-language instruction)
* Od użytkownika: C -c, C-z (interrupt, suspend)
* Wydarzenie oprogramowania:(wejście dostępne; timer off; proces dziecko terminated; okno terminala resized)

Domyślne działania przy odbiorze sygnału (W zależności od sygnału)

* Zignorowany - The signal is ignored; that is, it is discarded by the kernel and has no effect on the process. (The process never even knows that it occurred.)
* Nadnormalne zakończenie (Jak exit() zamiast normalnego)
* Generowanie pliku rdzenia zrzutu i normalne zakończenie
* Zatrzymany
* Wznowiony

Obsłużęnie sygnału

* Domyślna akcja(kill)
* Zignorowanie
* Signal handler
* Blokowany na później

Proces odbiera sygnał od razu(przerwanie i zmiana kontekstu) lub w najbliższym planowym uruchomieniu (dostep do CPU? - tak, patrz zadanie 3.)

Jeśli proces jest zatrzymany, to odbiera sygnały dopiero gdy jest wznowiony (chyba, że to sygnał KILL - wtedy proces umiera).

Nie można zignorować: SIGSTOP, SIGKILL

Pozwolenie na wysłanie sygnału [kill()] znać PID procesu do którego chce wysłać sygnał

For a process to have permission to send a signal it must either be privileged (under Linux: have the CAP\_KILL capability), or the real or effective user ID of the sending process must equal the real or saved set-user-ID of the target process. In the case of SIGCONT it suffices when the sending and receiving processes belong to the same session.

# proces może wysyłać sygnały w określonych warunkach

- jeżeli posiada uprzywilejowanie (CAP\_KILL) może wysłać do każdego innego procesu

- jeżeli jest nieuprzywilejowany, może wysłać sygnał do procesu jeżeli jego real user ID lub effective user ID zgadzają się z real user ID lub saved set-user-ID celu

- sygnał SIGCONT może zostać wysłany do procesów z tej samej sesji

SIGSEGV: One example of when you may restart the program is when you have memory obtained using [mmap(2)](http://www.kernel.org/doc/man-pages/online/pages/man2/mmap.2.html) that is read-only, you may check if the signal handler that the cause of segmentation fault was writing to this memory (using data from siginfo\_t) and use [mprotect(2)](http://www.kernel.org/doc/man-pages/online/pages/man2/mprotect.2.html) to change the protection of this memory. How practical is it? I don't know.

Używa się do debugowania - można w ten sposób np. wypisać stacktrace.

SIGILL : VMM może używać, żeby symulować uprzywilejowane operacje przez system gościa.

**Zad6**

Uruchomienie procesu:

xeyes

Szukanie procesu:

ps aux | grep xeyes (na liście procesów możemy odczytać PID)

Lub:

pgrep xeyes (od razu zwraca PID)

Uruchomienie i szukanie jednocześnie:

xeyes && echo &!

Różnice między kill, pkill, xkill:

kill numer\_procesu

pkill nazwa\_procesu (usunie wszystkie procesy o pasujących nazwach)

xkill (klikamy kursorem myszki na okienko, co zabije odpowiadający mu proces)

Lista sygnałów, które można wysłać:

kill -l

Domyślnie wysyłany jest sygnał SIGTERM (prosi proces o zakończenie pracy, ale proces może zignorować ten sygnał).

Wznawianie procesu po jego zatrzymaniu:

kill -CONT numer\_procesu

fg

Inspekcja pliku procesu:

cd

cd /proc

cd numer\_procesu

cat status (na samym dole będzie SigPnd, co oznacza maskę zgłoszonych sygnałów)

Wysyłanie sygnałów z polecenia zadania:

kill -USR1 numer\_procesu 10

kill -USR2 numer\_procesu 12

kill -HUP numer\_procesu 1

kill -INT numer procesu 2

SigQ - lista zkolejkowanych sygnałów/maksymalna liczba skolejkowanych sygnałów

SigPnd - zgłoszone sygnały

ShdPnd - zgłoszone, współdzielone sygnały

SigBlk - blokowane sygnały

SigIgn - ignorowane sygnały

SigCgt - złapane sygnały

Pierwszy wykona się sygnał o najniższym numerze, czyli SIGHUP.

**Zad7**

Przerobie to po polnocy.  
http://webcache.googleusercontent.com/search?q=cache:Y-3d5XeSakQJ:web.cs.wpi.edu/~cs502/f07/Lecture%2520Notes%2520%28f07%29/Week%252002,%2520Unix-Linux-Windows%2520Processes.ppt+&cd=3&hl=pl&ct=clnk&gl=pl&client=ubuntu  
  
Dla użytkowników i innych procesów proces jest identyfikowany przez Process ID (PID).

W systemie operacyjnym procesy są reprezentowane przez pola w Process Table (PT).

PID to wskaźnik na pole PT.

Wskaźnik na PT to Process Control Block (PCB).

PCB to duża struktura, która zawiera informacje o procesie (lub odnośniki do tych informacji).

Linux - PCB zdefiniowane w task\_struct – ponad 70 pól.

Zobacz include/linux/sched.h

NT – PCB zdefiniowane w EPROCESS – około 60 pól.

Użytkownicy dostają przywileje od admina. Przywileje mówią co dany użytkownik może robić z obiektem.

Unix/Linux – Read|Write|eXecute by user, group and “other” (i.e., “world”)

WinNT – Access Control List

Procesy “dziedziczą” przywileje od użytkownika.  
  
Unix/Linux

Tworzenie nowego (child) procesu - fork();

Wkłada nowy wpis do PCB

Prawie że klonuje dany proces

Kopiuje przestrzeń adresową ojca

Kopiuje zasoby w jądrze (np. pliki)

Wsadza nowe PCB na Ready queue - chodzi o to, że ten proces-dziecko jest gotowy do działania i czeka teraz na czas procesora

**Return from fork() call**

**0 for child**

**child PID for parent - WAŻNE!!!**  
  
Unix & Linux:–

int exec (char \*prog, char \*\*argv)

Check privileges and file type

Loads program “prog” into address space

Replacing previous contents!

Execution starts at main()

Initializes context – e.g. passes arguments

\*argv

Place PCB on ready queue

Preserves, pipes, open files, privileges, etc.  
  
fork() followed by exec()

Creates a new process as clone of previous one

First thing that clone does is to replace itself with new program  
  
Windows/NT – combines fork & exec

CreateProcess(10 arguments)

Not a parent child relationship

Note – privileges required to create a new process

Pytanie o copy-on-write **(niżej wersja pl)**:  
When a fork() system call is issued, a copy of all the pages corresponding to the parent process is created, loaded into a separate memory location by the OS for the child process. But this is not needed in certain cases. **Consider the case when a child executes an "exec" system call (which is used to execute any executable file from within a C program) or exits very soon after the fork(). When the child is needed just to execute a command for the parent process, there is no need for copying the parent process' pages, since exec replaces the address space of the process which invoked it with the command to be executed.**

In such cases, a technique called copy-on-write (COW) is used. With this technique, when a fork occurs, the parent process's pages are not copied for the child process. Instead, the pages are shared between the child and the parent process. Whenever a process (parent or child) modifies a page, a separate copy of that particular page alone is made for that process (parent or child) which performed the modification. This process will then use the newly copied page rather than the shared one in all future references. The other process (the one which did not modify the shared page) continues to use the original copy of the page (which is now no longer shared). This technique is called copy-on-write since the page is copied when some process writes to it.

**W skrócie:** kopiowanie procesu forkiem wsadza kopię wszystkich stron związanych z ojcem do innego miejsca w pamięci. Tak żeby dziecko miało swoje. No i motyw jest teraz taki, że czasem nie warto tego robić (pogrubiony jest przykład). W tym wypadku kopiowanie przy zapisie pozwala nam nie kopiować stron ojca dla dziecka, tylko te strony są współdzielone przez rodzinę. Dopiero kiedy któreś z nich modyfikuje stronę, osobną kopię (tę zmodyfikowaną) dajemy temu, kto zmieniał coś w stronie. Od tej pory nasz delikwent będzie już używał swojej strony, a nie tej współdzielonej. Inne procesy, które nie modyfikowały strony, dalej używają oryginalnej kopii, która już nie jest współdzielona z procesem, który ma własną zmodyfikowaną. Ta technika nazywa się kopiowanie przy zapisie, no bo stronę kopiujemy dopiero wtedy, kiedy jakiś proces grzebie w tej stronie.

Proces czy wątek tworzymy widelcem, much wow. Nie wiem, o co tu chodziło Cahirowi.

<http://man7.org/linux/man-pages/man2/clone.2.html>  
  
Clone() różni się od widelca tym, że pozwala dziecku dzielić z ojcem parę rzeczy **such as the memory space, the table of file descriptors, and the table of signal handlers.**

One use of clone() is to implement threads: multiple threads of control in a program that run concurrently in a shared memory space.

Tych flag clone() jest od chuja, ale część mówi o tym, co dziecko i tata mogą współdzielić i to jest ważne. Inne służą do tego, że otwierać procesy w nowych przestrzeniach nazw.  
  
**BONUS - z książki:**  
*In UNIX, there is only one system call to create a new process: fork . This call*

*creates an exact clone of the calling process. After the fork , the two processes, the*

*parent and the child, have the same memory image, the same environment strings,*

*and the same open files. That is all there is. Usually, the child process then ex-*

*ecutes execve or a similar system call to change its memory image and run a new*

*program. For example, when a user types a command, say, sort, to the shell, the*

*shell forks off a child process and the child executes sort. The reason for this two-*

*step process is to allow the child to manipulate its file descriptors after the fork but*

*before the execve in order to accomplish redirection of standard input, standard*

*output, and standard error.*

*In Windows, in contrast, a single Win32 function call, CreateProcess , handles*

*both process creation and loading the correct program into the new process. This*

*call has 10 parameters, which include the program to be executed, the com-*

*mand-line parameters to feed that program, various security attributes, bits that*

*control whether open files are inherited, priority information, a specification of the*

*window to be created for the process (if any), and a pointer to a structure in which*

*information about the newly created process is returned to the caller. In addition to*

*CreateProcess , Win32 has about 100 other functions for managing and synchro-*

*nizing processes and related topics.*

*In UNIX, the child’s initial address space is a copy of the parent’s, but there are definitely two distinct address spaces involved; no writable memory is shared.  
In Windows, the parent’s and child’s address spaces are different from the start.*

**Zad8**

# ltrace oraz strace służą do śledzenia wywołań bibliotecznych oraz systemowych danego programu

ltrace program # wywołania biblioteczne

ltrace -S program # biblioteczne oraz systemowe

strace program # tylko wywołania systemowe

# Dla aplikacji wielowątkowych możemy użyć flagi -f by śledzić wszystkie wątki oraz procesy przezeń tworzone

strace -f program

ltrace -f program

# Aby zliczyć ilość wywołań możemy użyć flagi -c

strace -c program

ltrace -c program

# Aby obserwować i na koniec podliczyć wywołania

strace -C program

# funkcjonalność niedostępna w ltrace

# Jeśli chcemy ograniczyć wyjście jedynie do interesującyh nas wywołań możemy użyć flagi -e

strace -e open,read,write program

ltrace -e open+scanf+printf program

# Powyższe flagi można łączyć

strace -ce read,write program

# Warto pamiętać, że flaga -e nie działa dla wywołań systemowych dla ltrace

ltrace -Se ... program # tutaj filtrujemy funkcje biblioteczne, nie wywołania systemowe

# Przykładowe wywołania

ltrace -fe fopen+fcanf program # obserwuj wszystkie otwarcia i zapisy do pliku programu i jego dzieci

ltrace -fce printf program # zlicz wszystkie printf'y programu i dzieci

strace -ce read,write program # zlicz wywołania read oraz write

Programy do sprawdzania

1. [Prosty program jednowątkowy](https://github.com/mbuszka/university_operating-systems/blob/master/assignments/list-3/example_simple.c)
2. [Program tworzący wiele procesów](https://github.com/mbuszka/university_operating-systems/blob/master/assignments/list-3/example_fork.c)