Jednym z podstawowych pojęć w systemie operacyjnym jest **proces**. Pojęcie to jest ściśle związane z pojęciami **program** oraz **zadanie**, ale w kontekście Unixa są to 3 różne obiekty i należy je rozróżniać. Najłatwiej wyobrazić sobie program. Ci którzy pisali i kompilowali programy wiedzą co to jest - plik binarny w kodzie maszynowym gotowy do uruchomienia. **Skrypty**, o których będzie więcej później, jak widać nie pasują do tej definicji bo wymagają programu interpretera, który je przeczyta i dopiero przetłumaczy na kod maszynowy, aby mogły być wykonane.

W momencie kiedy uruchamiamy program system operacyjny przydziela mu pamięć, ładuje go tam i przekazuje sterowanie - tak to wygląda nieco uproszczając, ale jeszcze wrócimy do tej kwestii. W ten sposób powstaje proces. Zatem proces to wykonywany program. W zasadzie jest to tylko egzemplarz, bo ten sam program możemy uruchomić wielokrotnie i utworzymy wiele różnych procesów. Na przykład, jeśli w danym momencie w systemie zalogowanych jest kilku użytkowników i wszyscy będą używać kompilatora C, to może się zdarzyć, że program kompilatora `cc` będzie uruchamiany równolegle przez kilku z nich. W ten sposób powstanie kilka różnych procesów kompilatora `cc`. Inny przykład, to uruchomionych kilka programów `httpd` serwera Apache realizujących usługę HTTP i oczekujących na żądania pobrania strony WWW przez klientów.

Pojęcie zadania zostawmy na później. Tymczasem wystarczy powiedzieć, że zadanie to proces powiązany z konkretnym terminalem i powłoką shell.

Tak jak to zostało powiedziane wcześniej każdy proces ma przypisany pewien obszar pamięci RAM. Jest to jeden z zasobów systemu używany przez proces. Inne zasoby jakich proces używa to: czas procesora, urządzenia wejścia-wyjścia oraz pliki. Zasoby te poza czasem procesora mogą być współdzielone przez różne procesy. Gdy na przykład mamy uruchomionych 50 egzemplarzy programu `httpd` to przydzielanie każdemu z nich odrębnego obszaru pamięci byłoby ogromnym marnotrawstwem. Cześć pamięci tych 50 procesów jest identyczna i nie zmienia się w trakcie ich działania - jest to **segment kodu** - więc może być współdzielony. Już **segment danych** to co innego. Tam przechowywane są dane procesu i niemal nigdy się nie zdarzy, aby dwa takie segmenty były identyczne, bo zawierają specyficzne informacje procesu. Fragment pamięci RAM komputera, wydzielony przez system operacyjny i dostępny dla danego procesu to jego **przestrzeń adresowa**. Poza tym wydzielonym kawałkiem RAMu proces nic więcej nie widzi. Taka separacja jest konieczna, aby jeden proces nie podejrzał i nie zniszczył danych innego procesu. System operacyjny dba, aby do tego nigdy nie doszło. Próba odwołania się do niedozwolonego obszaru pamięci, czyli *segmentation fault*, to najczęściej zgłaszany przez system błąd w trakcie wykonania programu i jest on spowodowany nie tyle celowym działaniem programu, ale błędem w jego algorytmie, na przykład, nieprawidłową gospodarką pamięcią, odczytem lub zapisem do tablicy poza jej zadeklarowanym rozmiarem.

System Unix został zaprojektowany tak, że cała dostępna pamięć RAM podzielona jest na dwa rozłączne obszary: **przestrzeń jądra** (kernel space) oraz **przestrzeń użytkowa** (user space, userland). Pomysł zaczerpnięto z pierwowzoru jakim był system Multics. Podział ten jest ściśle związany z poziomami zabezpieczeń obecnymi w procesorach. Najwyższy stopień zabezpieczeń w procesorach Intel to *ring 0*, w procesorach ARM to *system mode*. Na tym poziomie działa jądro i dzięki temu ma dostęp do wszelkich zasobów sprzętowych komputera. Najniższy poziom u Intela to *ring 3* a w ARM to *user mode*. Na tym poziomie wykonywane są programy z userland. Unix jak większość systemów używa tylko dwóch poziomów zabezpieczeń sprzętowych (pośrednie ring 1 i 2 nie są używane). Zarządzanie procesami odbywa się w jądrze systemu operacyjnego.

Plik ma unikalny inode, użytkownik ma unikalny UID, grupa ma unikalny GID itd. Proces ma także swój unikalny, numeryczny identyfikator PID, czyli Process Identifier. Każdy proces opisany jest wieloma parametrami. Poza PID najważniejsze z nich to:

- PPID - identyfikator procesu rodzica. Każdy proces poza jedynym wyjątkowym procesem uruchamianym jako pierwszy w systemie jest uruchamiany przez inny proces. Ten uruchamiający proces zwiemy rodzicem. Gdy w shellu wykonujemy polecenie `cat`, to rodzicem procesu `cat`, jest proces shella.
- UID - ID użytkownika, który uruchamia dany proces.
- GID - ID grupy podstawowej użytkownika, który uruchamia dany proces.
- EUID - efektywne ID użytkownika uzyskane w trakcie uruchamiania programu z ustawioną flagą SUID.
- EGID - efektywne ID ugrupy uzyskane w trakcie uruchamiania programu z ustawioną flagą SGID.
- PRI - priorytet z jakim działa proces.
- STIME - czas uruchomienia procesu (start time).
- TIME - efektywny, sumaryczny czas procesora jaki został przydzielony procesowi od momentu uruchomienia.
- STATE - stan procesu.
- CMD - nazwa programu i jego argumenty z jakimi proces został uruchomiony.

Możemy je zobaczyć na liście procesów wyświetlanej przez program `ps`. Bez dodatkowych opcji `ps` pokaże tylko uproszczoną listę procesów w danym shellu. Z parametrem `-e` (every) pokaże wszystkie procesy w systemie, natomiast z parametrem `-f` (full) pokaże pełną listę z dodatkowymi informacjami o procesach. Oto początkowy fragment takiej pełnej listy:

```
sirius# ps -ef | head
  UID   PID  PPID   C    STIME TTY  TIME CMD
 root     0     0   0 12:43:11 ?    1:02 sched
 root     1     0   0 12:43:30 ?    0:00 /sbin/init
 root     2     0   0 12:43:30 ?    0:00 pageout
 root     3     0   0 12:43:30 ?    0:30 fsflush
 root   341     1   0 12:43:45 ?    0:00 /opt/cfw/sbin/dhcpd -q e1000g1
 root     7     1   0 12:43:30 ?    0:01 /lib/svc/bin/svc.startd
 root     9     1   0 12:43:30 ?    0:03 /lib/svc/bin/svc.configd
 root   171     1   0 12:43:44 ?    0:00 /usr/lib/picl/picld
 root   681   680   0 12:43:47 ??   0:00 /usr/openwin/bin/fbconsole -n -d :0
```

Proces o PID równym 0 to dyspozytor (scheduler). To wyjątkowy proces bo nie ma rodzica. Jest on uruchamiany jako pierwszy przy starcie jądra systemu. Drugi proces (`/sbin/init`) na liście ma PID równe 1 i jego rodzicem jest proces o PID wynoszącym 0, czyli scheduler. Właścicielem powyższych procesów jest root. Z czasu uruchomienia schedulera można wywnioskować, że system został zbootowany o godzinie 12:43. Potwierdza to wynik programu `last`:

```
sirius# last -2
mariusz   console      :0       Sun Dec  6 12:55   still logged in
reboot    system boot           Sun Dec  6 12:43
```

Aby zobaczyć listę najbardziej aktywnych procesów w systemie używamy programu `top`. W nagłówku `top` wyświetla wiele cennych statystyk o systemie od czasu zbootowania, miedzy innymi średnie obciążenie i użycie pamięci.

```
sirius# top
last pid:  2797;  load avg:  0.17,  0.23,  0.30;  up 0+06:07:11
104 processes: 103 sleeping, 1 on cpu
```

```
CPU states: 98.1% idle,  0.2% user,  1.7% kernel,  0.0% iowait,  0.0% swap
Kernel: 7744 ctxsw, 5 trap, 5156 intr, 6382 syscall, 1 flt
Memory: 32G phys mem, 21G free mem, 4103M total swap, 4041M free swap

  PID USERNAME LWP PRI NICE  SIZE   RES STATE    TIME    CPU COMMAND
  996 mariusz   29  39    0 4469M 4387M sleep   72:22  1.69% VirtualBox
  682 mariusz    1  59    0  466M  413M sleep   18:01  0.03% Xorg
  955 mariusz   56  49    0 1259M  645M sleep   49:36  0.02% firefox-bin
 1785 mariusz    2  49    0  235M  122M sleep    0:16  0.02% acroread
  898 root       1  59    0 2396K 1504K sleep    0:21  0.01% rpc.rstatd
 2273 mariusz    1  59    0 9652K 4840K sleep    0:00  0.01% dtterm
  911 mariusz   63  49    0  454M  157M sleep    1:05  0.01% thunderbird-bin
```

Ponieważ wszystko w Unixie jest plikiem, więc konsekwentnie proces też jest plikiem. Mamy tutaj katalog `/proc`, gdzie przechowywane są metadane procesów.

```
sirius# ls -l /proc | head
total 216
dr-x--x--x   5 root     root         864 gr  6 12:43 0
dr-x--x--x   5 root     root         864 gr  6 12:43 1
dr-x--x--x   5 mariusz  staff        864 gr  6 12:59 1001
dr-x--x--x   5 mariusz  staff        864 gr  6 13:08 1060
dr-x--x--x   5 mariusz  staff        864 gr  6 13:08 1061
dr-x--x--x   5 mariusz  staff        864 gr  6 13:08 1063
dr-x--x--x   5 mariusz  staff        864 gr  6 13:08 1081
dr-x--x--x   5 mariusz  staff        864 gr  6 13:08 1082
dr-x--x--x   5 mariusz  staff        864 gr  6 13:08 1084
```

Jak widać, proces to formalnie katalog. Właściciel i grupa są takie jak UID i GID procesu. Zajrzyjmy do katalogu odpowiadającemu procesowi schedulera o PID równym 0:

```
sirius# ls -l /proc/0
total 21
-rw-------   1 root     root           0 gr  6 12:43 as
-r--------   1 root     root         168 gr  6 12:43 auxv
dr-x------   2 root     root          48 gr  6 12:43 contracts
-r--------   1 root     root          32 gr  6 12:43 cred
--w-------   1 root     root           0 gr  6 12:43 ctl
lr-x------   1 root     root           0 gr  6 12:43 cwd ->
dr-x------   2 root     root          32 gr  6 12:43 fd
-r--------   1 root     root           0 gr  6 12:43 ldt
-r--r--r--   1 root     root         120 gr  6 12:43 lpsinfo
-r--------   1 root     root         816 gr  6 12:43 lstatus
-r--r--r--   1 root     root         536 gr  6 12:43 lusage
dr-xr-xr-x   3 root     root          64 gr  6 12:43 lwp
-r--------   1 root     root           0 gr  6 12:43 map
dr-x------   2 root     root          32 gr  6 12:43 object
-r--------   1 root     root           0 gr  6 12:43 pagedata
dr-x------   2 root     root          64 gr  6 12:43 path
-r--------   1 root     root          72 gr  6 12:43 priv
-r--r--r--   1 root     root         336 gr  6 12:43 psinfo
-r--------   1 root     root           0 gr  6 12:43 rmap
lr-x------   1 root     root           0 gr  6 12:43 root ->
-r--------   1 root     root        1536 gr  6 12:43 sigact
-r--------   1 root     root        1136 gr  6 12:43 status
-r--r--r--   1 root     root         256 gr  6 12:43 usage
-r--------   1 root     root           0 gr  6 12:43 watch
-r--------   1 root     root           0 gr  6 12:43 xmap
```

Mamy tam małe pliki binarne opisujące stan danego procesu.

Tak jak pliki, procesy mają swego właściciela, grupę i prawa dostępu. W zasadzie jest to konsekwencja koncepcji według, której wszystko jest plikiem. Jeśli nie jesteśmy właścicielem procesu i nie jesteśmy odpowiedniej grupie to na przykład nie możemy wymusić zakończenia tego procesu.

W momencie, gdy uruchamiamy program system operacyjny wykonuje następujące operacje:

1. Utworzenie przestrzeni adresowej.
2. Wypełnienie struktury metadanych opisujących proces.
3. Kopiowanie kodu i danych z programu wykonywalnego do przestrzeni adresowej procesu.
4. Mapowanie współdzielonych zasobów w przestrzeń adresowa procesu.
5. Dołączenie procesu do kolejki procesów oczekujących na przydzielenie czasu procesora i ustalenie priorytetu wykonania.
6. Ustawienie stanu procesu na działający.

Każdy proces może być w jednym z następujących stanów:

- nowy, w trakcie tworzenia,
- aktywny, aktualnie wykonywany przez procesor,
- czekający na dostęp do zasobów,
- uśpiony,
- zakończony, przeznaczony do zniszczenia,
- zombie.

Proces czeka na dostęp do zasobów, gdy na przykład otwieramy stronę w przeglądarce, a system w tym czasie pobiera aktualizacje. Wówczas wysłanie żądania z przeglądarki nastąpi dopiero wtedy, gdy program aktualizacji nie tyle zakończy swoją pracę, ale zostanie odebrany mu czas procesora i nastąpi mała przerwa w przesyłaniu danych przez sieć. W ten sposób zasoby w postaci łącza sieciowego staną się dostępne przeglądarce. I odwrotnie, aby program aktualizujący mógł kontynuować, przeglądarka musi zrobić sobie małą przerwę.
W sytuacji, gdy proces nie jest w stanie kontynuować działania ponieważ nie ma dostępu do wymaganych zasobów mówimy, że proces jest **głodzony**.

Proces uśpiony to taki, który został z jakiegoś powodu zablokowany przez system. Jego wykonanie jest zawieszone. Taki proces nie dostaje czasu procesora. W modelu 5 stanowym systemu operacyjnego procesy czekające na zasoby i uśpione traktowane są na równi jako zablokowane.

Proces zombie to proces, które zakończył swoje działanie, ale w tablicy procesów pozostaje jego PID, aby rodzic mógł odczytać stan zakończenia swego procesu potomnego. Po odczycie stanu przez rodzina proces zombie jest usuwany z tablicy. Proces zombie to tylko wpis w tablicy procesów, nie zajmuje on żadnych zasobów poza tym.

W systemach z podziałem czasu bardzo ważną rolę odgrywa dyspozytor (scheduler), który przydziela czas procesora i w ten sposób decyduje, który proces w danym momencie jest aktywny. Musi uwzględnić zasoby sprzętowe. Ilość procesorów i rdzeni może być różna. Różne procesy mogą być wykonywane z różnymi priorytetami. Niektóre procesy czekają na zasoby itd. To komplikuje algorytm szeregowania. Z podziałem czasu nierozłącznie związane jest przełączanie kontekstu (context switching). Jak wiemy przy podziale czasu mamy wrażenie, że procesy wykonywane są równolegle podczas, gdy w rzeczywistości dostają czas procesora na krótko, w małych odstępach czasu. W momencie kiedy dyspozytor wstrzymuje jeden proces, aby przydzielić czas procesora drugiemu należy zapamiętać całe środowisko w jakim wykonywany jest pierwszy proces, czyli stan rejestrów, licznik programu, stos, listę otwartych plików itd. a odtworzyć te informacje dla drugiego

procesu, aby mógł bez zakłóceń kontynuować swoje działanie. To przełączenie stanu procesora z jednego procesu na drugi nazywa się właśnie przełączaniem kontekstu. Jest to dość kosztowna operacja w sensie czasu wykonania. Duża ilość przełączeń kontekstowych w systemie świadczy o dużym jego obciążeniu lub nieprawidłowo działającym algorytmie szeregowania. Wartość tę podaje `top` w nagłówku `Kernel` jako `ctxsw`.

Czasem zdarza się tak, że proces z powodu błędu w swoim algorytmie zawiesza się, przestaje reagować na zdarzenia i wysyłane do niego sygnały. Dyspozytor musi mieć możliwość przerwać wykonanie takiego procesu by odzyskać blokowane przez niego zasoby. Generalnie dyspozytor powinien mieć możliwość przerwania lub zablokowania dowolnego procesu dla zapewnienia stabilności całemu systemowi. Nazywa się to **wywłaszczaniem**. Jest to podstawowa cecha systemów z podziałem czasu.

Ponieważ przełączenie kontekstowe procesów jest kosztowne, aby lepiej wykorzystać zasoby sprzętowe, większą ilość rdzeni procesora w ramach jednego procesu może być wykonywanych wiele **wątków** (thread), czasem zwanych jako lekkie procesy (lightweight processes). O ile procesy są od siebie izolowane, mają odrębne przestrzenie adresowe, to wątki współdzielą wszystkie zasoby dostępne procesowi. Z tego względu przełączanie procesora między wątkami jest znacznie łatwiejsze i trwa krócej niż przełączanie kontekstowe. Aby lepiej sobie wyobrazić sens wątków, rozważmy przykład programu do obróbki grafiki. Powiedzmy, że opracowaliśmy duży, złożony obraz i wydajemy polecenie zapisania go na nośniku. Gdy zapis takiego pliku trwa 15 sekund, co jest zauważalne, to na ten czas aplikacja nie będzie reagować na operacje I/O z myszki i klawiatury. Będziemy oglądać klepsydrę, albo jakiś gadżet przez czas zapisu. A co jeśli zapis trwa 15 minut? Idziemy zrobić kawę? Ale ile można pić kawy? Gdyby rozdzielić zadania interakcji z użytkownikiem i zapisu, to główny wątek zarządzający procesu w momencie kliknięcia Zapisz tworzy odpowiedni wątek roboczy zapisujący dane na nośniku i od razu wraca do wątku interakcji. Owszem, komputer trochę spowolni bo ma dane do zapisania, ale międzyczasie możemy otwierać następny obraz do obróbki.

Wątki mają kilka istotnych zalet:

- utworzenie i zakończenie wątku zajmuje znacznie mniej czasu niż w przypadku procesu,
- przełączanie kontekstu pomiędzy wątkami tego samego procesu trwa krócej niż przełączanie kontekstowe procesów,
- wątki jednego procesu mogą komunikować się ze sobą bez pośrednictwa systemu operacyjnego,
- wykorzystujemy możliwości maszyn wieloprocesorowych SMP (Symmetric MutliProcessing).

Jedyną chyba wadą wątków jest to, że z uwagi na brak izolacji, błędy w kodzie jednego wątku mogą spowodować zakłócenia w pracy pozostałych wątków i zawiesić cały proces. No cóż zawsze jest coś za coś - nie można mieć wszystkiego.

Do obsługi procesów przez administratora wspomniany został niezbędnik `ps` oraz `top`. Warto wymienić tutaj program `pgrep`, który pozwala odszukać PID procesu według nazwy programu:

```
sirius# ps -ef | fgrep fire
 mariusz   955     1   0 12:56:15 ?        72:39
/opt/sfw/lib/firefox/firefox-bin
    root  4152  2296   0 23:17:07 pts/6    0:00 fgrep fire

sirius# pgrep fire
955
```

Jak widać można z pełnej listy procesów odfiltrować przy pomocy `fgrep` lub `grep` proces, który szukamy, ale łatwiej to zrobimy przy pomocy `pgrep`.

W Solaris mamy program prstat bardzo podobny do top, ale pozwalający obejrzeć parametry poszczególnych wątków:

```
sirius# prstat
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/NLWP
   996 mariusz  4469M 4387M cpu0    43    0   1:57:36 1,5% VirtualBox/29
   955 mariusz  1437M  878M sleep   49    0   1:12:45 0,2% firefox-bin/57
  1441 mariusz   254M  131M sleep   49    0   0:02:33 0,1% soffice.bin/5
   682 mariusz   468M  414M sleep   59    0   0:27:12 0,1% Xorg/1
```

Na tej liście widać, że proces `firefox-bin` ma 57 wątków. Zobaczmy te, które są aktywne:

```
sirius# prstat -L
   PID USERNAME  SIZE   RSS STATE  PRI NICE      TIME  CPU PROCESS/LWPID
   682 mariusz   468M  414M sleep   59    0   0:27:11 1,0% Xorg/1
   996 mariusz  4469M 4387M sleep   39    0   0:59:59 0,9% VirtualBox/12
   955 mariusz  1437M  890M sleep   31    0   0:41:50 0,8% firefox-bin/1
   996 mariusz  4469M 4387M sleep   44    0   0:48:32 0,7% VirtualBox/11
   955 mariusz  1437M  890M sleep   41    0   0:00:16 0,2% firefox-bin/633
   955 mariusz  1437M  890M sleep   42    0   0:00:24 0,1% firefox-bin/14
   955 mariusz  1437M  890M sleep   44    0   0:16:07 0,1% firefox-bin/53
  1441 mariusz   254M  131M sleep   49    0   0:02:29 0,1% soffice.bin/1
   876 mariusz    13M 8988K sleep   59    0   0:00:08 0,0% dtwm/1
```

Najbardziej aktywne są wątki numer 1, 663, 14 i 53. Wątki numerowane są w momencie tworzenia, tak więc numer wątku 663 świadczy o tym, że program Firefox od uruchomienia utworzył co najmniej 663 wątki.