Systemy operacyjne

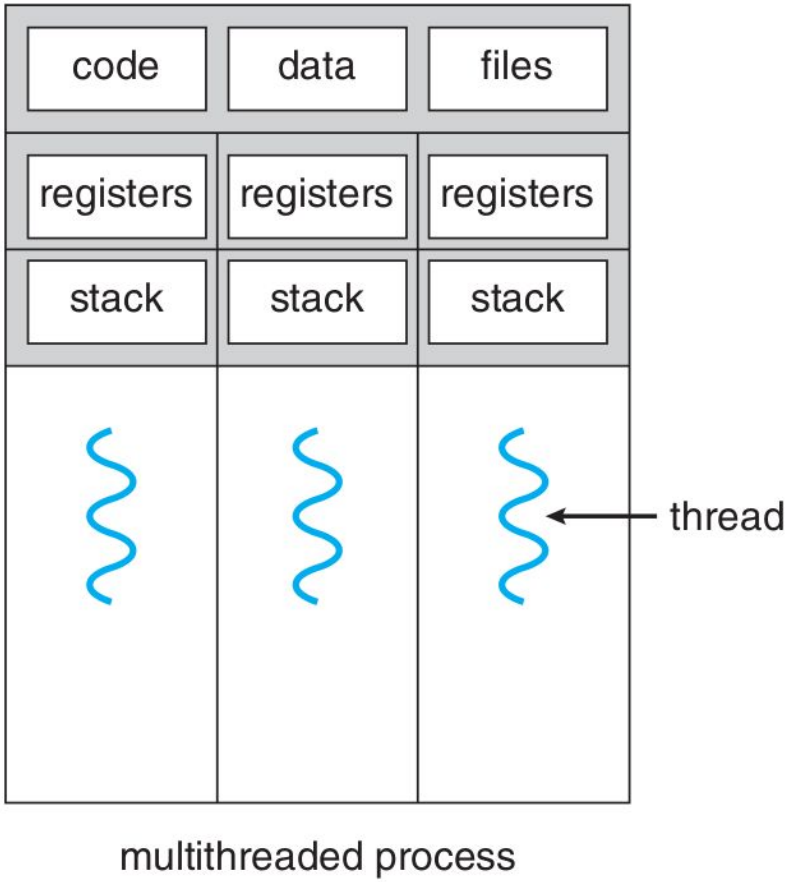
Wykład 3: Procesy

Procesy: wprowadzenie

Proces to środowisko uruchomieniowe składające się: z przestrzeni adresowej, załadowanego programu, przydzielonej pamięci, otwartych plików i innych zasobów.

Proces można zasiedlić niezależnymi kontekstami wykonania instrukcji → **wątkami**.

Proces bez wątku nie ma sensu!



multithreaded process

Procesy: motywacja

Pożądane cechy środowiska uruchomieniowego:

1. Toleruje awarie programu:
   - nie pozwala zakłócić działania innych programów
   - potrafi obsłużyć błędy wykonania programu
   - po zakończeniu pracy programu zasoby wracają do środowiska
2. Wirtualizuje zasoby – program nie musi:
   - kontrolować jaka jego część leży w pamięci fizycznej
   - obserwować liniowego czasu rzeczywistego (z wyjątkiem zadań RT)
   - wiedzieć, że współdzieli urządzenie (np. HDD) z innymi programami
3. Udostępnia:
   - abstrakcje → pliki zamiast bloków dyskowych
   - zunifikowany interfejs do urządzeń zewnętrznych
   - metody komunikacji z innymi programami → gniazda, skrzynki pocztowe

Sposoby na „niegrzeczne” programy

Izolacja przestrzeni adresowych: tryb pracy procesora daje izolację jądro ↔ proces, tablice stron: proces ↔ proces.

Jądro tłumaczy wyjątki i pułapki na **sygnały**. Daje programom możliwość obsługi tych zdarzeń albo kończy ich działanie.

Jądro śledzi wszystkie zasoby przydzielone procesowi. **Uchwyty zasobów** (ang. *resource handle*) przechowuje w strukturze opisującej proces (PCB).

Wywłaszczanie nie pozwala zdominować czasu procesora przez błędnie działający program (np. nieskończona pętla).

Maszyna wirtualna (1)

Programy nie muszą wiedzieć jak są wykonywane!

Stronicowanie na żądanie: załadujemy tylko taki fragment kodu i danych programu, z jakiego rzeczywiście korzysta.

Wymiana: jeśli fragment pamięci już się nie przyda to przenieśmy go do pamięci drugorzędnej.

Wywłaszczanie: wstrzymywanie i wznawianie programu, bez jego wiedzy i bez wpływu na jego poprawność.

Wirtualny czasomierz: mierzy czas wykonania programu, ale tylko wtedy, gdy są wykonywane jego instrukcje.

Maszyna wirtualna (2)

Multipleksowanie: metoda koordynowania dostępu do zasobów współdzielonych → wypożyczanie na kwant czasu.

Wywołania systemowe: instrukcje realizujące zadania, których szczegółów wykonania program nie powinien znać.

Interfejs plików: zunifikowany dostęp do danych → obsługuje pamięć drugorzędną, komunikację lokalną i zdalną między programami, dostęp do urządzeń I/O.

Do jednej maszyny można przypisać wiele **wirtualnych procesorów**, które będą wykonywać wątki programu.

SO: wsparcie dla wykonania programów

1. Ładowanie, startowanie, wstrzymywanie, wznawianie i kończenie programów.
2. Monitorowanie zużycia zasobów → naliczanie kosztów, co zrobić gdy program próbuje zdominować inne?
3. Autoryzacja dostępu do zasobów → czy program jest uprawniony, żeby …
4. Nadzorowanie wykonania → czy robi to co powinien?
5. Śledzenie wykonania → odpluskwianie (ang. *debugging*).
6. Grupowanie procesów → zadania, polityki szeregowania.

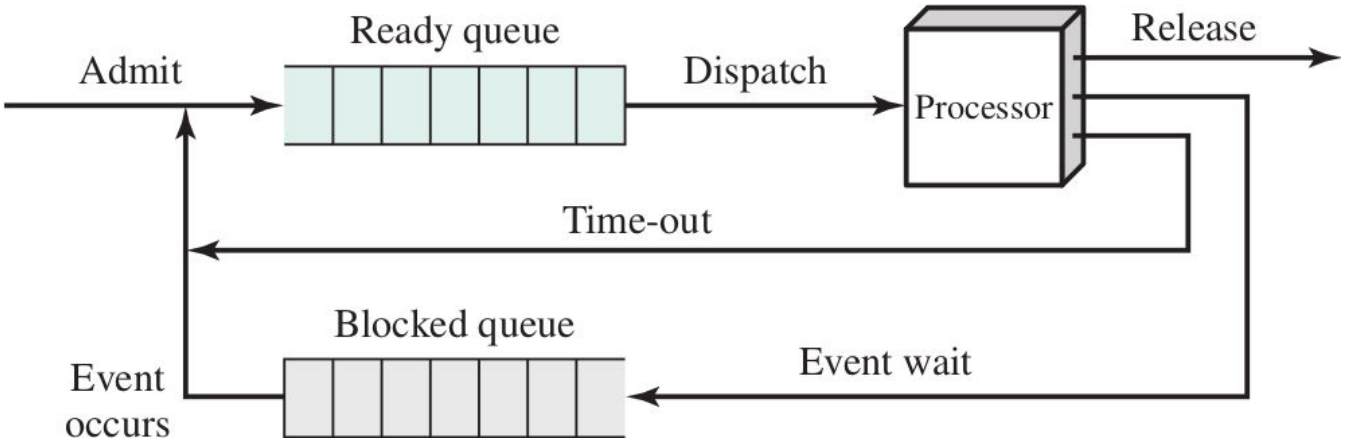
Przełączanie procesów

Przełączanie procesów polega na przełączeniu przestrzeni adresowych, wstrzymaniu wirtualnego czasomierza T_1, wymianie kontekstu procesora, wznowieniu czasomierza T_2.

1. Wątek może zostać **uśpiony** (przez zawołanie `cv_wait`) w wyniku wykonania w jądrze **operacji blokującej**.
2. W trakcie przetwarzania wywołania systemowego lub procedury obsługi przerwania może zostać wybudzony wątek o wyższym **priorytecie**.
3. W wyniku upłynięcia kwantu czasu (przerwanie zegarowe) należy **wstrzymać** dany wątek.

Szeregowanie procesów

Dyspozytor (ang. *dispatcher*) uruchamiany, kiedy wątek chce zostać uśpiony lub zrzec się czasu procesora (ang. *yield*): `mi_switch(9)`. Wywołuje procedurę **planisty** (ang. *scheduler*) wybierającą następnika do uruchomienia: `sched_nextlwp(9)`. Proces się zablokował → trafia do jednej z kolejek procesów zablokowanych, w p.p. wraca do kolejki procesów gotowych.



Model maszyny procesu

Maszyna rzeczywista	**Maszyna wirtualna**
zbiór instrukcji (ISA)	wywołania systemowe
wznawialne instrukcje	wznawialne wywołania systemowe
przerwania, pułapki, wyjątki	sygnały
procedury obsługi przerwań	procedury obsługi sygnałów
blokowanie przerwań	maskowanie sygnałów
stos obsługi przerwania	stos obsługi sygnału

Jądro (z reguły) nie jest zainteresowane obserwacją wykonania pojedynczych instrukcji programu.

Interweniuje wyłącznie w przypadku nadejścia przerwania sprzętowego, wywołania wyjątku procesora lub pułapki.

Sygnały: wysyłanie

- **synchroniczne** → związane z wykonaniem instrukcji
- **asynchroniczne** → zdarzenia zewnętrzne: budzik, przerwanie (CTRL+C), zakończenie procesu potomnego

SO tłumaczy wyjątki i pułapki na sygnały synchroniczne.
W tym przypadku jądro **wysyła sygnał** do wątku.

Sygnały asynchroniczne to prymitywna metoda komunikacji między programami i zgłaszanie zdarzeń przez jądro. Wątki mogą wysyłać sygnały do procesów!

Pytanie: Jak jądro rozróżnia błąd strony od błędu dostępu?

Sygnały: doręczanie

Jądro ma kilka opcji: zignorować sygnał, zakończyć proces, wywołać **procedurę obsługi sygnału** (ang. *signal handler*), której kod jest wykonywany przez jeden z wątków procesu.

Kiedy jądro sprawdza, czy należy doręczyć sygnał?

1. Przed wejściem w stan uśpienia.
2. Po wyjściu ze stanu uśpienia.
3. Przed powrotem do przestrzeni użytkownika `userret(9)`.

Jeśli sygnały mogą przerywać sen, to mówimy wtedy o **śnie przerywalnym** (ang. *interruptible sleep*).

Process Control Block

Struktura jądra przechowująca informacje o procesie i zasobach z których korzysta!

- Kontekst procesora (zawartość rejestrów, SP, PC, …)
- Informacje dla planisty (zużycie procesora, priorytet)
- Stan procesu
- Identyfikatory, uprawnienia
- Obraz pamięci (opis stanu przestrzeni adresowej)
- Informacje rozliczeniowe (pomiar zużycia zasobów)
- Uchwyty do używanych zasobów (pliki, IPC, …)

PCB są przechowywane w globalnej liście procesów.

Tożsamość i uprawnienia procesów

W systemach wieloużytkownikowych wymagana **kontrola dostępu** (ang. *access control*) do zasobów. Procesy mają **tożsamość** (ang. *identity*) użytkownika, z reguły tego który je utworzył. Tożsamość należy potwierdzić przy pomocy mechanizmu **uwierzytelniania** (ang. *authentication*).

Proces widzi zasoby w obrębie przydzielonych **przestrzeni nazw** → system plików. Prosi jądro o przydzielenie zasobu → otwarcie pliku. Jeśli proces ma odpowiednie **uprawnienia** (ang. *permissions*) to dostaje **uchwyt** do zasobu.

Katalog roboczy procesu → wyróżniony element przestrzeni nazw.

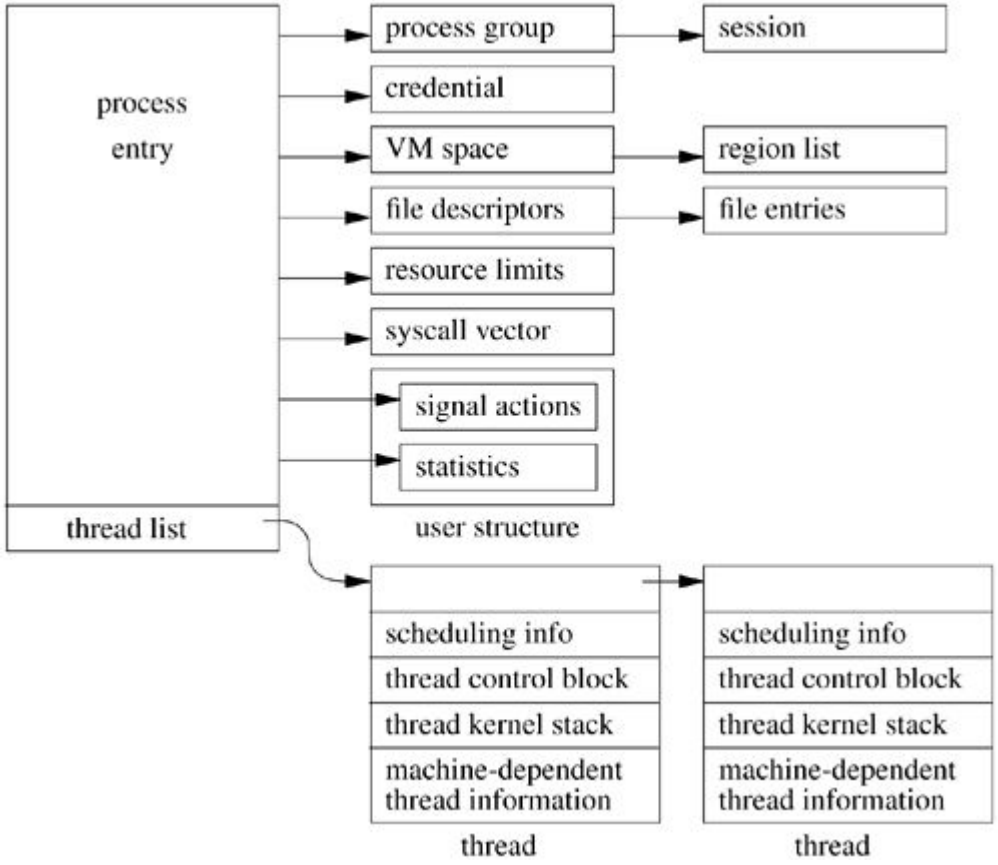
Procesy w systemach uniksowych

Unix: Zasoby procesu

Linux (`task_struct`)
FreeBSD (`proc`, `thread`)
NetBSD (`proc`, `lwp`)

Linux i BSD prezentują różne podejście do zarządzania wątkami!

Wątek posiada po jednym stosie dla przestrzeni użytkownika i jądra.



Tworzenie procesów

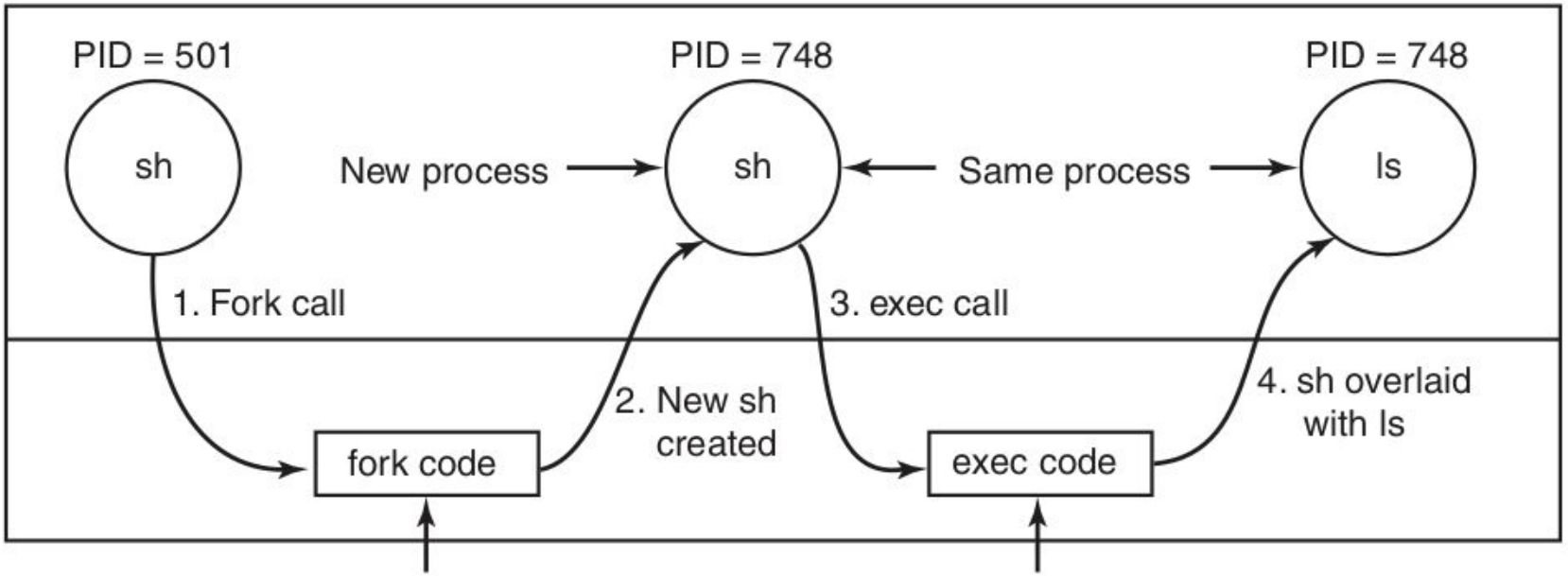
- Inicjalizacja systemu (proces `init`)
- Wywołanie systemowe (`fork`)
- Żądanie użytkownika (funkcja powłoki)
- Utworzenie zadania wsadowego (**demony**)

By załadować nowy program do pamięci, należy utworzyć nowy **obraz procesu** z pliku wywołaniem `execve`.

Funkcja biblioteczna **`fork`** korzysta z wywołania systemowego `clone` (Linux) lub `rfork` (FreeBSD).

Te z kolei potrafią tworzyć inne byty (np. wątki, kontenery, …)

UNIX: Uruchamianie programów



Allocate child's task structure
Fill child's task structure from parent
Allocate child's stack and user area
Fill child's user area from parent
Allocate PID for child
Set up child to share parent's text
Copy page tables for data and stack
Set up sharing of open files
Copy parent's registers to child

Find the executable program
Verify the execute permission
Read and verify the header
Copy arguments, environ to kernel
Free the old address space
Allocate new address space
Copy arguments, environ to stack
Reset signals
Initialize registers

Kończenie pracy procesów

- normalne zakończenie pracy (`exit`)
- zakończenie w wyniku błędu (`abort`, `assert`, `raise`)
- awaria procesu (`SIGSEGV`, `SIGILL`, `SIGBUS`, …)
- błąd krytyczny (`SIGKILL` od jądra)
- zniszczenie przez inny proces (`kill`, …)

Można oczekiwać na zakończenie potomków **`waitpid`** lub dowolnych procesów **`wait4`**. Odbierzemy informację czy zakończył się normalnie, z błędem, czy w wyniku awarii.

Gdy proces się zakończy, można podejrzeć zużycie zasobów (czas CPU, pamięć, operacje I/O, …) z użyciem **`getrusage`**.

Sygnały

Pełnione funkcje:

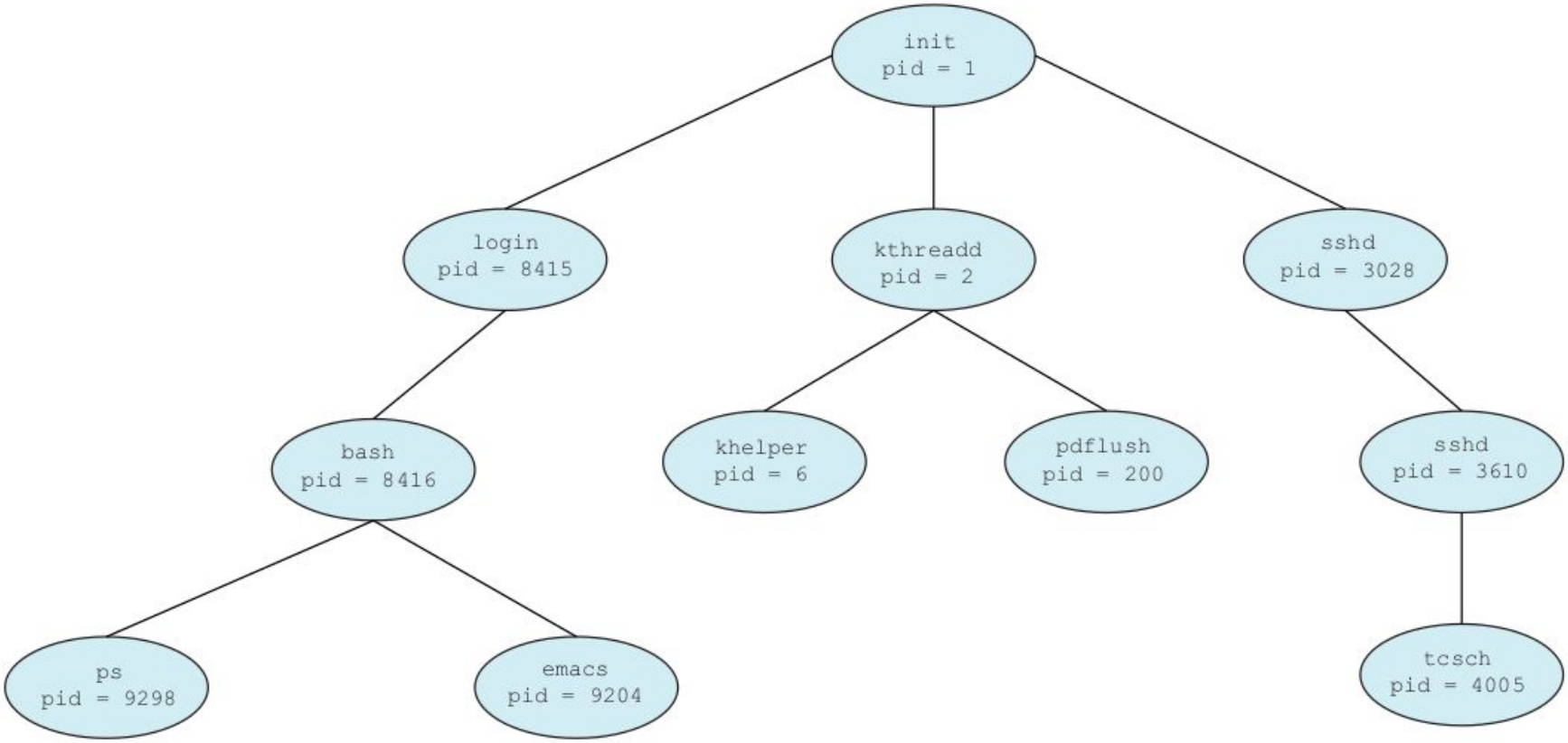
- komunikacja międzyprocesowa
- tłumaczenie wyjątków procesora i pułapek
- sytuacje wyjątkowe
- zarządzanie procesami

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Wywołania systemowe: maskowanie (`sigprocmask`), odpytywanie (`sigpending`), oczekiwanie (`sigsuspend`), ustalanie procedury obsługi (`sigaction`), powrót z procedury obsługi (`sigreturn`), wysyłanie (`kill` i `killpg`), …

Domyślne reakcja na otrzymanie sygnału? `signal(7)`

Drzewo procesów



Prezentacja narzędzia `pstree`!

Relacja rodzic ↔ potomek

W systemach uniksowych tak, ale w *WinNT* opcjonalnie.

Czytanie identyfikatorów: `getpid`, `getppid`, `getpgrp`.

Wszystkie procesy poza **`init`** muszą mieć rodzica. Co się stanie jeśli rodzic umrze? Ktoś musi przygarnąć **sieroty**.

Zakończonego procesu nie można od razu usunąć. Co gdy rodzic chce zobaczyć zużycie zasobów lub **kod wyjścia**? Proces najpierw przechodzi do stanu **`ZOMBIE`**.

Możliwe grupowanie procesów w **zadania** (ang. *job*).

Pomijamy zarządzanie sesjami i zadaniami – charakterystyczne dla systemów uniksowych!

Uprawnienia, użytkownicy i grupy

Globalna przestrzeń nazw dla zasobów → system plików.

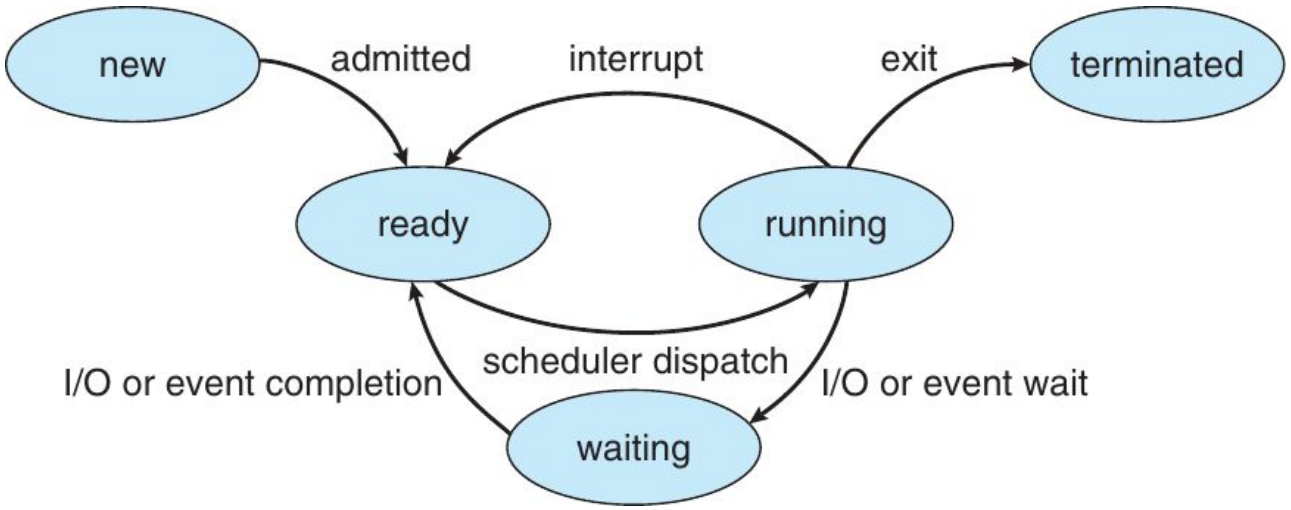
Tożsamość procesu: **użytkownik**, **grupa podstawowa**, **grupy rozszerzone**. Wyróżniony przez jądro użytkownik `root` pełni rolę administratora.

Proces dziedziczy zestaw **umiejętności** (`capabilities`), których dobrowolnie może się zrzekać → bezpieczeństwo.

Prezentacja narzędzi `getent` i `id`!

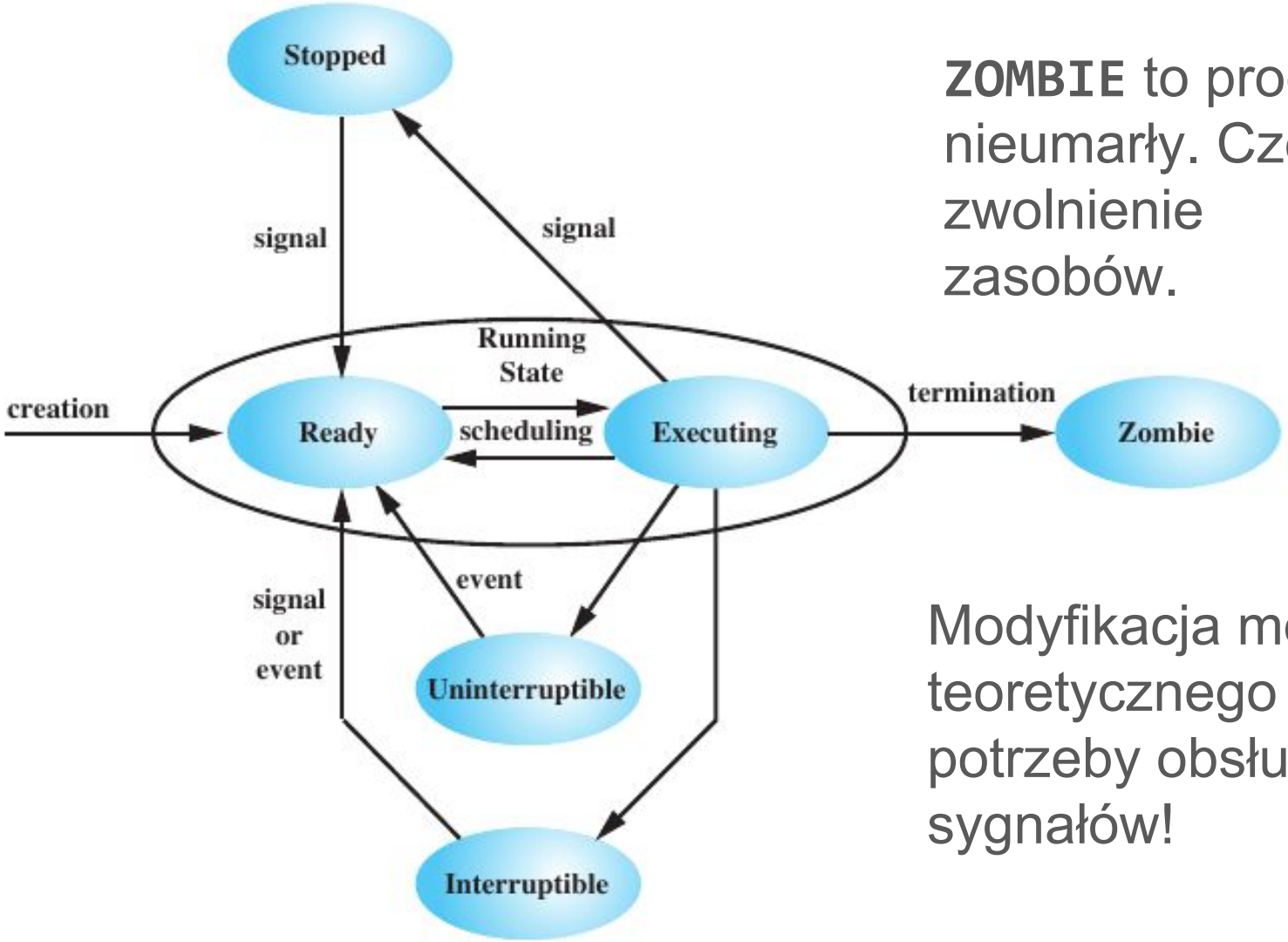
Baza danych dostępna przez usługę `NSS`. Zestaw narzędzi systemowych do modyfikacji bazy danych np.: `useradd`.

Stan procesu: model teoretyczny



- **`NEW`**: brakuje zasobów
- **`READY`**: gotowy do uruchomienia (`runqueue`)
- **`WAITING`**: oczekuje na zdarzenie (`sleepqueue`, `turnstile`)
- **`RUNNING`**: uruchomiony na procesorze (`pcpu`)
- **`TERMINATED`**: oczekuje na zwolnienie zasobów (`exit`)

Linux: stan procesu



`ZOMBIE` to proces nieumarły. Czeka na zwolnienie zasobów.

Modyfikacja modelu teoretycznego na potrzeby obsługi sygnałów!

Sen płytki i głęboki

Ile możemy czekać na wybudzenie procesu? Zwolnienie blokady (~1 µs), dane z dysku (~10 ms), pakiet sieciowy (~1s), naciśnięcie klawisza (nigdy?).

Część akcji szybko się zakończy. Ich przerwanie może być szkodliwe (np. założenie blokady się nie powiodło?). Zbyt częste rozpatrywanie sygnałów kosztuje.

`UNINTERRUPTIBLE` → sen **ograniczony** (ang. *bounded*).

`INTERRUPTIBLE` → musimy mieć możliwość wybudzenia procesu, który zasnął na **nieograniczoną** ilość czasu.

Śledzenie wykonania procesów

Z użyciem wywołania `ptrace` można:

- podłączyć się do istniejącego procesu by go odpluskwiać
- czytać i modyfikować stan procesu (rejestry, pamięć)
- przechwytywać sygnały wysyłane do procesu
- nasłuchiwać na zdarzenia (`fork`, `exec`)
- śledzić wywołania systemowe
- przełączyć wykonywanie programu w tryb krokowy

Prezentacja narzędzia `strace`!

Pytania?