7 Procesy – demony

• ***Demonem*** (ang. *daemon*) nazywany jest proces, który wykonuje usługi działając w drugim planie (ang. *background*) i nie ma sterującego terminala. Jego procesem macierzystym jest proces `init`.

ps -axj – pokazuje procesy ? - demony

• Typowy demon systemowy jest procesem, który charakteryzuje się następującymi cechami:

- jest rozpoczynany raz, podczas inicjowania pracy systemu,
- trwa przez cały czas pracy systemu,
- większość czasu spędza na oczekiwaniu na jakieś zdarzenie, po którym wykonują swoją pracę,
- często powołuje nowe procesy w celu obsłużenia zamówienia.

• Czynności, które trzeba wykonać, aby utworzyć proces-demon:

1. uruchomić proces w drugim planie,
2. umieścić proces w jego prywatnej grupie procesów,
3. odłączyć proces od terminala sterującego,
4. ustawić maskę trybu dostępu do plików,
5. przełączyć proces do znanego, bezpiecznego katalogu,
6. zamknąć odziedziczone deskryptory plików,
7. otworzyć deskryptory standardowego we-wy,
8. zapewnić wzajemne wykluczanie się kopii procesów,
9. zapamiętać identyfikator procesu,
10. oczekiwać na zakończenie procesów potomnych,
11. ignorować sygnały zewnętrzne.

7.1 Uruchomienie w drugim planie

- Do uruchomienia w drugim planie najlepiej wykorzystać funkcję fork:

```
i=fork();
if (i<0) { /* ujemna wartość oznacza błąd */
      fprintf(stderr, "blad funkcji fork()");
      exit(1);
}
if (i) {
      /* wartość dodatnia: proces macierzysty */
      exit(0); /* zakończ proces macierzysty */
}
      /* tutaj dalsza część programu,
      która wykonuje się jako proces potomny */
```

- Zakończenie procesu macierzystego po wywołaniu funkcji `fork()` oznacza, że proces demona zostaje osierocony i jego procesem macierzystym staje się proces `init`. Gwarantuje to poprawne uprzątnięcie procesu demona po jego zakończeniu. Proces potomny otrzymał swój identyfikator `PID`, różny od identyfikatora procesu macierzystego `PPID`. W ten sposób gwarantujemy, że proces nie jest przywódcą grupy, co jest wymagane na przykład w funkcji `setsid`.

7.2 Umieszczenie w prywatnej grupie procesów

- Proces potomny dziedziczy od procesu macierzystego przynależność do grupy procesów.
- Serwer powinien działać niezależnie od innych procesów. Aby nie otrzymywać sygnałów przeznaczonych dla grupy procesu macierzystego, serwer musi się odłączyć od tej grupy.
- Umieszczenie serwera o określonym identyfikatorze we własnej, nowej grupie procesów zależy od systemu Unix. W standardzie Posix.1 należy użyć funkcji `setsid` (Tworzy nową sesję, jeśli proces wywołujący nie jest liderem grupy. Proces staje się liderem sesji.)

7.3 Odłączenie od terminala sterującego

- Proces potomny dziedziczy powiązanie procesu macierzystego z terminalem sterującym.
- Serwer działający w tle musi odłączyć się od terminala sterującego, aby na jego działanie nie miały wpływu sygnały z terminala.
- Schemat postępowania zależy od wersji systemu Unix.
    - Posix.1: można użyć funkcji `setsid`. Proces staje się przywódcą sesji, przywódcą nowej grupy procesów, nie ma terminala sterującego
- Należy również zablokować możliwość odzyskania terminala sterującego. Proces może odzyskać terminal sterujący, jeśli będzie wysyłać komunikat na przykład do pliku `/dev/tty` lub `/dev/console`. Tutaj również sposób postępowania zależy od systemu:
    - jeśli terminal sterujący może być odzyskany przez przywódcę sesji, wystarczy zakończyć działanie przywódcy sesji:

```
if (pid=fork()) != 0) //brak reakcji na błąd
     exit(0); /* proces macierzysty kończy pracę*/
/*pierwszy potomek */
  setsid(); /* zostaje przywódcą grupy i sesji */
/* zablokowanie sygnału wysyłanego do wszystkich procesów
grupy podczas kończenia procesu przywódcy */

  signal(SIGHUP,SIG_IGN);
  if (pid=fork()) != 0)
     exit(0); /*pierwszy potomek kończy pracę*/

/* teraz będzie działał drugi potomek, który nie jest już
   przywódcą sesji i grupy */
```

7.4 Ustawienie bieżącego katalogu

- Podczas trwania procesu jądro systemu zachowuje otwarty jego roboczy katalog bieżący. Oznacza to, że administrator nie może na przykład odmontować systemu plików, w którym katalog ten się znajduje.
- Proces powinien być przeniesiony do takiego katalogu, w którym będzie mógł działać nieograniczenie długo nie utrudniając zarządzania systemem. Na przykład może to być katalog główny. Schemat postępowania w tym przypadku:

```
chdir ("/");
```

- Demon może również wymagać pewnego określonego katalogu, w którym umieszcza swoje pliki. Nie istnieje jeden katalog, który byłby najwłaściwszy dla wszystkich demonów.

7.5 Ustawienie maski trybu dostępu do plików

- Proces dziedziczy po przodku maskę trybu dostępu do pliku.
- Prawa dostępu do tworzonych plików wyznaczane są na podstawie kodu trybu dostępu podawanego w funkcji `open()` oraz wartości maski: wykonywana jest bitowa operacja and na kodzie trybu dostępu i bitowej negacji maski. Niezależnie od tego, jaki tryb dostępu proces określi w wywołaniu funkcji `open()`, system nie przydzieli szerszych uprawnień niż te określone maską.
- Wartość maski ustawia się za pomocą funkcji `umask()`.
- Przykład:

```
umask(027); /* ustaw wartość maski na 027 (właściciel – wszystko, grupa rx, inni nic
                */

umask(0); /* ustaw wartość maski na 000 – wszyscy wszystko */
```

- Ograniczenia obowiązują do następnego wywołania funkcji `umask`.

7.6 Zamknięcie otwartych deskryptorów plików

- Proces potomny dziedziczy po macierzystym kopie deskryptorów plików, które były otwarte w kontekście procesu macierzystego w chwili utworzenia procesu potomnego. Proces działający w tle powinien wykonać operację zamknięcia pliku na wszystkich odziedziczonych deskryptorach, aby zapobiec przetrzymywaniu zasobów.

- Można:

  - wykorzystać funkcję `getdtablesize()`, która podaje rozmiar tablicy deskryptorów procesu:

    ```
    for (i=getdtablesize()-1; i>=0; --i) close(i);
    ```

  - przyjąć pewną liczbę `n` i zamknąć wszystkie pierwsze `n` deskryptorów:

    ```
    #define MAXFD 64

    for (i=0; i<MAXFD; i++) close(i);
    ```

  - w standardzie Posix.1 zdefiniowana jest stała `OPEN_MAX`, określająca maksymalną liczbę otwartych plików .

7.7 Deskryptory standardowego we-wy

- Wiele procedur bibliotecznych odwołuje się do trzech standardowych deskryptorów we-wy, które w związku z tym muszą być otwarte. Są to:

  - standardowe wejście (ang. *standard input*, deskryptor 0) ,
  - standardowe wyjście danych (ang. *standard output*, deskryptor 1) ,
  - standardowe wyjście dla komunikatów o błędach (ang. *standard error*, deskryptor 2) .

- W takim przypadku można otworzyć standardowe deskryptory i przypisać je urządzeniu obojętnemu, czyli takiemu, które pozwala poprawnie wykonać operacje we-wy, ale czyni to nieskutecznie.

- Przykładem takiego urządzenia jest urządzenie obsługiwane przez plik specjalny `/dev/null`. Przy próbie wprowadzania danych z tego urządzenia zawsze zgłaszany jest stan końca pliku, dane wyprowadzane są odrzucane. W rezultacie wykonywane operacje są operacjami pustymi.

- Otwieranie standardowych deskryptorów:

```
fd=open("/dev/null",O_RDWR); /* std. wejście */

/* dup – tworzy kopię deskryptora i przydziela mu najmniejszy z
możliwych numerów */

dup(fd); /* standardowe wyjście danych */

dup(fd); /* standardowe wyjście dla błędów */
```

7.8 Wzajemne wykluczanie egzemplarzy demona

- Jeśli nie chcemy inicjować działania więcej niż jednej kopii demona w danym czasie, musimy wprowadzić wzajemne wykluczanie się egzemplarzy demona.

- Programowa realizacja wzajemnego wykluczania demonów może być zrealizowana za pomocą pliku blokady (ang. *lock file*):

    - każdy serwer ma swój własny plik blokady,

    - proces wykonujący program demona rozpoczynając działanie usiłuje ustanowić kontrolę nad przypisanym mu plikiem blokady,

    - jeśli żaden inny proces nie zablokował tego pliku, to proces uzyskuje kontrolę nad plikiem; jeśli inny proces, wykonujący ten sam program zajął już plik, to próba zakończy się niepowodzeniem.

- Ustawianie / usuwanie blokady na pliku

```
#include <sys/file.h>
int flock(int fd, int operation);
```

Ustawia lub usuwa blokadę na otwartym pliku o deskryptorze `fd`. Parametr *`operation`* przyjmuje jedną z wartości:

`LOCK_SH`
: Umieść blokadę dzieloną (ang. *shared lock*). Wówczas więcej niż jeden proces może współdzielić blokadę dla danego pliku w danym czasie.

`LOCK_EX`
: Umieść blokadę wyłączną (ang. *exclusive lock*). Tylko jeden proces może trzymać taki plik w danym czasie.

`LOCK_UN`
: Usuń blokadę dla tego pliku przez ten proces.

Wywołanie `flock()` może zablokować się jeżeli plik został zablokowany przez inny proces. Aby tego uniknąć należy dołączyć `LOCK_NB` (przez OR) do dowolnej z powyższych operacji.

W przypadku sukcesu zwraca 0. Przy błędzie zwraca -1 i ustawia `errno`.

- Przykład:

```
#define LOCKF /var/spool/serwer.lock
lf=open(LOCKF, O_RWDR|O_CREAT, 0640);
if (lf < 0) /* błąd podczas otwierania pliku */
     exit(1);
if (flock(lf, LOCK_EX|LOCK_NB))
   exit(-1); /* nie udało się zablokować pliku */
```

7.9 Zarejestrowanie identyfikatora procesu demona

- Demon często zapamiętuje identyfikator swojego procesu w pliku o ustalonej nazwie. Dzięki temu można szybko odnaleźć ten identyfikator, bez potrzeby przeglądania listy wszystkich procesów działających w systemie.
- Takim plikiem może być na przykład plik blokady serwera.
- Przykład:

```
char pbuf[10]; /* pid w postaci napisu */

      /* zamień liczbę binarną na dziesiętną */

sprintf(pbuf,"%6d\n",getpid());

      /* plik blokady jest już otwarty */

write(lf,pbuf,strlen(pbuf));
```

7.10 Ignorowanie sygnałów zewnętrznych

- Należy zdecydować, których sygnałów demon będzie używał do sterowania działaniem, a które mają być ignorowane.
- Jeśli na przykład sygnał `HANGUP` ma być ignorowany przez demon, to program powinien zawierać wywołanie:

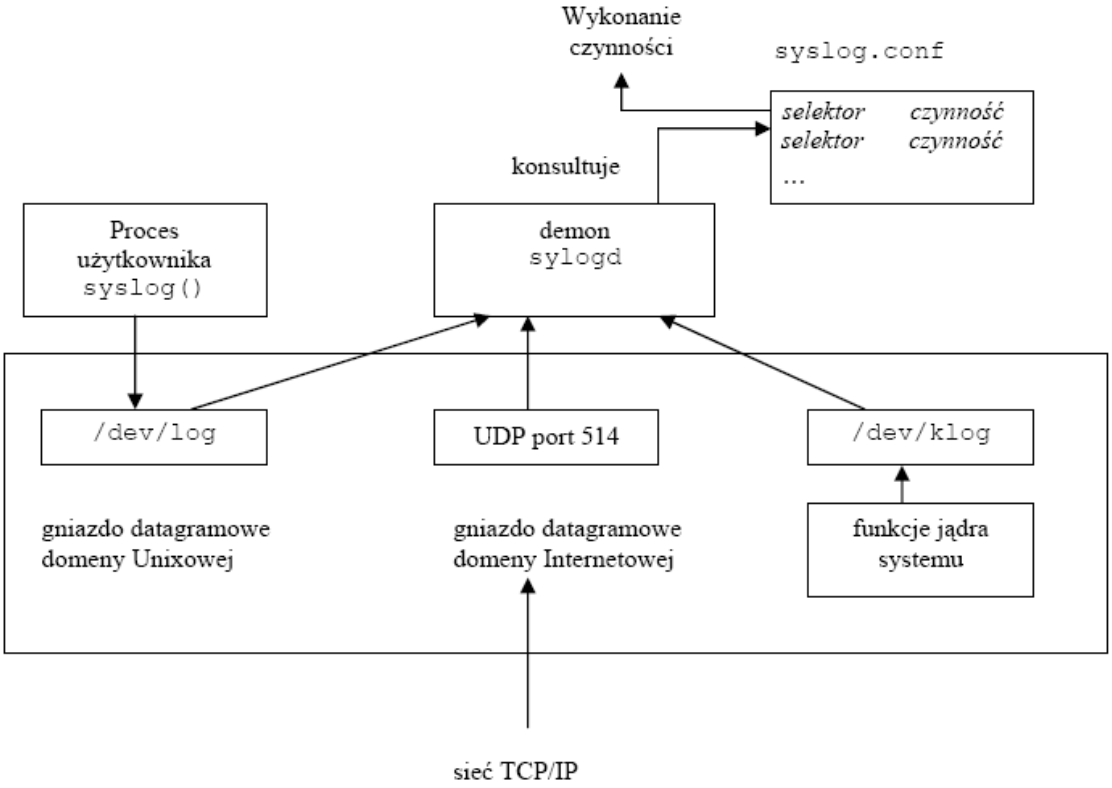
```
signal(SIGHUP,SIG_IGN)
```

7.11 Zapisywanie komunikatów do dziennika

- Dwa rozwiązania:
    - przesyłanie komunikatów do standardowego wyjścia błędów, o ile zostało ono zdefiniowane,
    - przesyłanie komunikatów do własnego dziennika,
    - przesyłanie komunikatów do centralnie sterowanego dziennika.
- W Unixie istnieje usługa o nazwie `syslog`, która pozwala centralnie zarządzać komunikatami o błędach pochodzącymi z różnych programów.

- Usługa ta oparta jest o model klient-serwer. W jej skład wchodzą:
    - serwer `syslogd`,
    - funkcje biblioteczne, włączane do procesów-klientów, które pozwalają przesłać komunikaty do serwera.

7.12 Usługa syslog Unixa BSD 4.3.



- Demon `syslogd` nasłuchuje komunikatów, których źródłem może być:
    - proces wykonywany na lokalnym komputerze (`/dev/log`),
    - jądro systemu na lokalnym komputerze (`/dev/klog`),
    - proces na innym komputerze (UDP port 514).
- Rejestrowane komunikaty zawierają informacje o źródle komunikatu, funkcji i priorytecie przypisanym komunikatowi, oraz właściwą treść komunikatu.
- Programy, które chcą się komunikować z demonem `syslogd` przesyłają do niego komunikaty za pomocą funkcji `syslog`.

- Po otrzymaniu komunikatu `syslogd` sprawdza w pliku `syslog.conf`, co ma zrobić z otrzymanym komunikatem. Może przesłać otrzymany komunikat:
  - do wskazanego pliku lub urządzenia,
  - do wskazanego użytkownika lub wszystkich użytkowników,
  - do innego programu,
  - do demona `syslogd` na innym komputerze.

7.12.1 Klasyfikacja komunikatów obsługiwanych przez syslogd

- Programy wysyłające komunikaty podzielone są na klasy usług (ang. *facilities*).

Klasa usług	Znaczenie
LOG_AUTH	komunikaty dotyczące bezpieczeństwa i uprawnień
LOG_CRON	demon `cron` (okresowe wykonywanie zadań)
LOG_DAEMON	inne demony systemowe
LOG_KERN	komunikaty jądra systemu
LOG_LOCAL0 do LOG_LOCAL7	używane lokalnie
LOG_MAIL	podsystem poczty
LOG_SYSLOG	wewnętrzne komunikaty demona `syslogd`
LOG_USER	różne komunikaty poziomu użytkownika

- Jeżeli proces nie określi klasy usługi, domyślnie przyjmowana jest wartość LOG_USER.

- W każdej klasie wyróżnia się 8 poziomów priorytetów uszeregowanych hierarchicznie.

Poziom	**Wartość**	**Znaczenie**
LOG_EMERG	0	system jest niedostępny, rozesłać do wszystkich
LOG_ALERT	1	operację wykonać natychmiast
LOG_CRIT	2	warunki krytyczne (np. awaria dysku)
LOG_ERR	3	warunki awaryjne, ale nie krytyczne
LOG_WARNING	4	ostrzeżenie
LOG_NOTICE	5	sytuacja wymaga uwagi
LOG_INFO	6	komunikaty informacyjne
LOG_DEBUG	7	komunikaty diagnostyczne

- Jeżeli proces nie określi poziomu, domyślnie przyjmowana jest wartość `LOG_NOTICE`.

7.12.2 Funkcje obsługujące przesyłanie komunikatów do syslogd

```
#include <syslog.h>


void openlog(char *ident, int option, int facility);

void syslog(int priority, char *format,...);

void closelog(void);
```

- Funkcja `openlog` otwiera połączenie z demonem `syslogd`. Określa komunikat, który będzie wyświetlany razem ze standardowym komunikatem demona `syslogd` (argument `ident`), opcję (argument `option`), która określa sposób przesyłania komunikatu (na przykład `LOG_PID` oznacza, że do każdego komunikatu ma być dołączony identyfikator PID procesu) oraz klasę komunikatów (argument `facility`). Przykład:

```
openlog("nazwa_programu", LOG_PID, LOG_USER);
```

- Funkcja `syslog` generuje komunikat o określonym priorytecie:

```
syslog(LOG_INFO,"Połączenie z %s",nazwa_hosta);
```

- Funkcja `closelog` zamyka deskryptor, który był używany do komunikowania się z `syslogd`.

7.13 Przykład funkcji tworzącej proces-demon

```
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int daemon_start(const char *pname, int facility)
     /* pname - wskaźnik do nazwy procesu
        facility - klasa usług dla syslogd
     */
{
int i;
pid_t pid;

/* Utwórz nowy proces */
pid=fork();
if (pid == -1)
     return -1;
else if (pid !=0)
     exit(EXIT_SUCCESS); /*proces macierzysty kończy działanie */
/* Utwórz nową sesję i grupę procesów */
if (setsid() == -1)
     return -1;
/* Zmiana katalogu roboczego */
if (chdir("/") == -1)
     return -1;

/* Zerowanie maski tworzenia pliku */
umask(0);

/* Zamknięcie wszystkich otwartych plików */
for (i=getdtablesize()-1; i>=0; --i)
     close(i);

/* Przekierowanie 0,1,2 do /dev/null */
open("/dev/null",O_RDWR); /* stdin */
dup(0); /* stdout */
dup(0); /* stderr */
/* do rejestracji komunikatów wykorzystywany będzie syslogd */
openlog(pname,LOG_PID,facility);
return 0;
}
```

• Użycie funkcji w serwerze: na początku kodu serwera umieszczamy wywołanie:

```
daemon_start(argv[0],0);
```

Drugi argument równy 0, oznacza, że program serwera został zakwalifikowany do klasy LOG_USER. Zamiast 0 można tutaj umieścić dowolną z wartości wymienionych w tabeli klas usług demona `syslogd`.

7.14 Funkcja `daemon()`

- Od wersji BSD 4.4 istnieje funkcja biblioteczna `daemon()`, która wykonuje podstawowe działania związane z przekształceniem zwykłego procesu w proces-demon.

```
#include <unistd.h>

int daemon (int nochdir, int noclose);
```

- Funkcja ta odłącza proces od terminala sterującego i wykonuje go w drugim planie jako demon.
- Argument `nochdir=0` oznacza, że bieżącym katalogiem procesu ma zostać katalog główny /. Wartość ≠ 0 oznacza pozostawienie w bieżącym katalogu odziedziczonym z procesu macierzystego.
- Argument `noclose ≠ 0` oznacza, że standardowe wejście, wyjście podstawowe i diagnostyczne mają być przeadresowane do /`dev/null`.

```
daemon(0, 0);
```