Opis algorytmu `wait`

Leopold Wegner

19 grudnia 2000

1 Wprowadzenie

W Linuxie do synchronizacji działania procesu macierzystego z zakończeniem lub zatrzymaniem procesu potomnego służą funkcje: `wait` i `waitpid` (występują także funkcje `wait3, wait4` - styl systemu BSD).

Funkcja `wait(int *status)` wstrzymuje proces do momentu aż jeden z jego procesów potomnych przestanie działać (zakończy się lub zatrzyma).

Funkcja `waitpid(pid_t pid, int *status, int options)` działa podobnie jak `wait`, lecz czeka na potomka o identyfikatorze spełniającym warunek wywołania (argument `pid`) oraz uwzględnia podczas działania przekazane flagi (argument `options`).

Funkcje `wait3(int *status, int options, struct rusage *rusage)` i `wait4(pid_t pid, int *status, int options, struct rusage *rusage)` są jakby odpowiednikami funkcji `wait` i `waitpid`, z tym że w obu wywołaniach tych funkcji należy podać argument określający flagi (zmienna `options`) oraz wskaźnik do struktury `rusage` (argument `rusage`) opisującej zużycie zasobów przez proces potomny.

Implementacją powyższych funkcji jest funkcja `sys_wait4(pid_t pid, unsigned int *status, int options, struct rusage *ru)` realizująca opisywany algorytm `wait`.

2 Argumenty wywołania funkcji

Wartości argumentu `pid` mogą być następujące:

- >0 oznacza czekanie na potomka o identyfikatorze równym `pid`
- $=0$ oznacza czekanie na potomka o identyfikatorze grupy równym identyfikatorowi grupy procesu wywołującego `wait`
- $=$ -1 oznacza czekanie na dowolnego potomka
- $<$ -1 oznacza czekanie na potomka o identyfikatorze grupy równym wartości bezwzględnej z `pid`

Argument `status` to adres, pod który zostanie zapisana przyczyna zatrzymania lub zakończenia działania przez potomka.

Wartość argumentu `options` może być alternatywą następujących flag:

`WNOHANG` funkcja nie zasypia w oczekiwaniu na zakończenie lub zatrzymanie procesu potomnego tylko od razu kończy działanie i zwraca zero (oczywiście musi najpierw zostać znaleziony proces potomny, którego numer pid spełnia warunki nałożone podczas wywołania funkcji, jeśli takiego procesu nie ma zwracany jest błąd)

`WUNTRACED` funkcja bierze pod uwage wszystkich potomoków będących w stanie `TASK_STOPPED`, jeśli nie ustawiona to zwraca uwagę tylko na tych potomków w stanie `TASK_STOPPED` którzy mają ustawioną flagę śledzenia `PF_PTRACE` (używana przez proces śledzący do zauważenia procesu śledzonego)

`__WCLONE` flaga ta jest używana do czekania na wątki jądra, które po śmierci wysyłają do procesu macierzystego sygnał różny od `SIGCHLD`, jeśli jest ustawiona to bierzemy tylko pod uwagę zakończenie się takiego wątku wpp. rozpatrujemy tylko te procesy potomne, które wysyłaja sygnał `SIGCHLD`

Argument `ru` jest wskaźnikiem do sruktury `rusage`, która opisuje zużycie zasobów przez proces potomny (na którgo koniec działania zaczekaliśmy). Po wywołaniu funkcji `sys_wait4`, w strukturze wskazywanej przez `ru` zostaną zaktualizowane następujące pola:

- `ru_utime` opisuje czas zużyty w trybie użytkownika
- `ru_stime` opisuje czas zużyty w trybie jądra
- `ru_minflt` opisuje ilość wystąpień błędów braku strony nie wymagających korzystania z urządzeń wymiany
- `ru_majflt` opisuje ilość wystąpień błędów braku strony wymagających korzystania z urządzeń wymiany
- `ru_nswap` opisuje ilość wymian stron Wszystkie te pola będą się odnosiły od zużycia zasobów przez proces potomny i jego dzieci (procesy potomne procesu potomnego)

3 Procesy brane pod uwagę

Funkcje `wait, waitpid (wait3, wait4)` odnoszą się do potomków będących w stanie `TASK_ZOMBIE` lub `TAKS_STOPPED` (ewentualnie podczas wywołania funkcji nałożone są jeszcze warunki na identyfikatory procesów potomnych).

STAN `TASK_ZOMBIE` Proces osiąga ten stan na dwa sposoby - wywołując funkcję `exit` lub gdy otrzyma sygnał, dla którego czynnością domyślną jest zakończenie procesu (proces nie zmienił domyślnej obsługi tego sygnału). W pierwszym przypadku proces do pola `exit_code` (jedno z pól struktury `task_struct` opisującej dany proces) wpisuje argument `exit-a` przesunięty o 8 bitów w lewo, w drugim przypadku do pola `exit_code` jest wpisywany numer sygnału, który spowodował zakończenie się procesu. Jądro systemu zwalnia wszystkie zasoby używane przez proces zombie (stan `TASK_ZOMBIE`) lecz cały czas utrzymuje stan danego procesu, aż proces macierzysty nie wywoła funkcji `wait`.

STAN `TASK_STOPPED` Proces osiąga ten stan gdy otrzyma sygnał `SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOUT`. Numer sygnału, który otrzymał proces zostaje wpisany do pola `exit_code`.

4 Wartości przekazywane przez funkcję

Sytuacja normalna.

Funkcja `wait` (i pozostałe) przekazują dwie wartości:

- zwraca identyfikator zakończonego lub zatrzymanego procesu lub 0 jeśli była ustawiona flaga `WNOHANG` i proces posiada potomków o określonych w wywołaniu funkcji numerach pid lecz żaden z nich nie jest w stanie `TASK_STOPED` lub `TASK_ZOMBIE`
- jeśli został podany adres w wywołaniu funkcji (argument `status`) to zostaje umieszczona tam przyczyna zakończenia lub zatrzymania procesu - jeśli był to proces w stanie `TASK_ZOMBIE` to przekazana zostaje wartość pola `exit_code` znalezionego potomka, jeśli prosec był w stanie `TASK_STOPPED` to wartość `pola exit_code` zostaje przesunięta o 8 bitów w lewo, a w miejsce 8 młodszych bitów zostaje wpisana wartość 0x7f. Taka organizacja zwracanej wartości rozróżnia wszystkie przypadki.

Proces potomny	8 starszych bitów	8 młodszych bitów
wywołał `exit`	argument `exit`	0x00
zabity sygnałem	0x00	numer sygnału (*)
zatrzymany	numer sygnału	0x7f

(*) - najstarszy bit jest ustawiony jeśli proces wygenerował plik `core` (zrzut pamięci)

Sytuacja błędna.

Jeśli wystąpił błąd to funkcje zwracają -1, a zmienna `errno` może otrzymać wartość

- `ECHILD` - proces nie ma potomków określonych przez argument `pid`
- `EINVAL` - argument `options` jest niepoprawny
- `ERESTARTSYS` - w trakcie oczekiwania na potomka proces otrzymał nieprzewidziany sygnał (różny od `SIGCHLD` i nie zablokowany)

5 Opis funkcji sys_wait4

Implementacją funkcji wait i waitpid (wait3, wait4) jest systemowa funkcja sys_wait4(pid_t pid, unsigned int *status, int options, struct rusage *ru). Funkcja ta realizuje algorytm czekania procesu macierzystego na koniec działania procesu potomnego. Niniejsza sekcja dokładnie opisuje działanie algorytmu funkcji.

5.1 Opis najważniejszych zmiennych i struktur danych użytych w funkcji sys_wait4

Struktury danych:
task_struct - struktura opisująca proces, w funkcji używane są następujące pola tej stuktury

- wait_chldexit - kolejka oczekiwania na śmierć potomka (używana przez ojca wykonującego wait)
- p_cptr - wskaźnik do najmłodszego potomka
- p_opptr - wskaźnik do orginalnego ojca procesu
- p_pptr - wskaźnik do bieżącego ojca procesu
- p_osptr - wskaźnik do starszego brata
- pid - numer pid procesu (identyfikator procesu)
- pgrp - numer pgrp grupy procesu
- exit_signal - sygnał wysyłany do ojca po zakończeniu działania procesu, standardowo ustawiany na SIGCHLD lecz może ulec zmianie przy klonowaniu procesu
- state - stan w jakim znajduje się proces
- exit_code - kod zakończenia procesu
- flags - flagi procesu, w naszym przypadku używane do sprawdzenia czy proces jest w stanie śledzenia
- times - rekord opisujący czas zużyty przez proces w trybie jądra i użytkownika (sam, wraz z potomkami)

Zmienne:

- current - jest to wskaźnik do struktury task_struct opisującej proces macierzysty (wywołujący funkcję wait)
- takslist_lock - zmienna używana do obsługi (tworzenia) sekcji krytycznych w kodzie funkcji (używanych jedynie w systemie wieloprocesorowym, w celu poprawnego wykonania funkcji sys_wait4), na maszynie jednoprocesorowej jest bez znaczenia

5.2 Opis algorytmu funkcji sys_wait4

Najpierw funkcja sprawdza poprawność argumentu `options` (czyli użytych flag) - w wypadku wykrycia niedozwolonej wartości zostaje zwrócony błąd (`EINVAL`).

Później proces dodaje się do swojej kolejki `wait_chldexit` (pole struktury `task_struct`). Robi to po to aby proces potomny, kończąc działanie mógł poinformować go (obudzić go ze spania wewnątrz `wait-a`) o swojej śmierci (funkcja `notify_parent`).

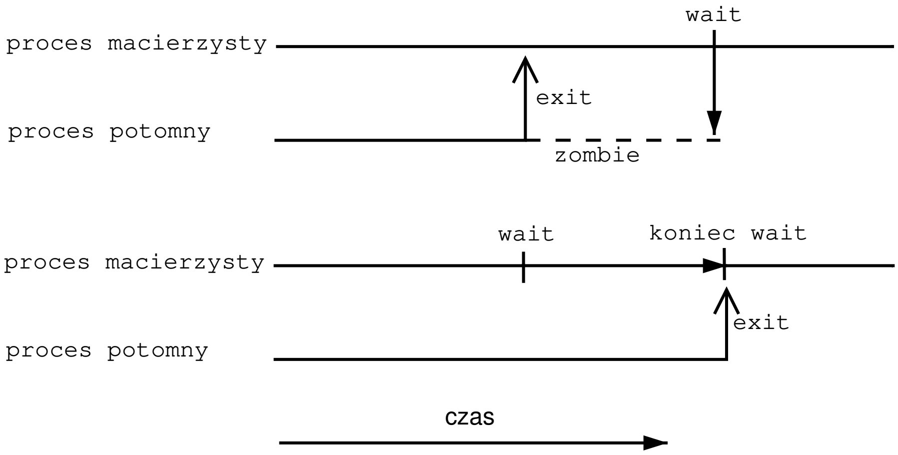
Następnie proces zmienia swój stan na `TASK_INTERRUPTIBLE` (dopuszczający przerwanie) i zaczyna przeglądać swoich wszystkich potomków. Zmiana stanu następuje przed sprawdzaniem swoich potomków aby móc "złapać" sygnał śmierci potomka, który skończył działanie w trakcie wykonywania się funkcji. Dla każdego przeglądanego potomka sprawdzany jest jego numer pid - jeśli numer nie spełnia warunku wywołania funkcji (argument `pid`) to potomek jest pomijany. Kolejnym krokiem jest sprawdzenie ustawienia flagi `__WCLONE` (opisane powyżej). Jeśli dany potomek jest "odpowiedni" (przeszedł pozytywnie powyższe sprawdzenia) wtedy funkcja sprawdza stan w jakim się znajduje dany proces. Wszystkie procesy nie będące w stanie `TASK_ZOMBIE` lub `TASK_STOPPED` sa pomijane.

W przypadku znalezienia potomka w stanie `TASK_STOPPED` sprawdzane jest jego `pole exit_code`. Jeśli pole to równe jest zeru to znaczy, że proces macierzysty już czekał na tego potomka i nie zostaje on brany teraz pod uwagę. Gdy pole `exit_code` jest różne od zera, sprawdzane jest ustawienie flagi `WUNTRACED` i pola `flags` potomka (opisane powyżej). Jeśli proces przeszedł przez to sprawdzenie to znaczy, że został odnaleziony odpowiedni potomek. Zostaje wtedy zaktualizowana struktura wskazywana przez argument `ru` (struktura `rusage`), a pod adres `status` (argument funkcji) zostaje przepisana przyczyna zatrzymania procesu (opis w tabeli powyżej). Ponadto do `pola exit_code` znalezionego potomka wpisywana jest wartość zero - aby nie brać po uwagę danego potomka w następnym wywołaniu funkcji.

W przypadku znalezienia potomka w stanie `TASK_ZOMBIE` aktualizowane jest pole `times` procesu macierzystego (dodany zostaje czas zużyty przez potomka w trybie jądra i użytkownika), podobnie jak powyżej zostaje zakutalizowana wartość struktury `ru` oraz argumentu `status`. Oprócz tego sprawdzane jest czy znaleziony potomek poinformował o swojej śmierci odpowiedniego ojca (istnieje możliwość, że potomek poinformował nie tego ojca co powinien, taka sytuacja występuje podczas używania funkcji `sys_ptrace`). Jeśli się tak zdarzyło, że poinformowano nie tego ojca co trzeba to funcja informuje orginalnego ojca o śmierci potomka (funkcja `notify_parent`), w przeciwnym przypadku usuwa pozostałości po zakończonym potomku (zostaje zwolniona pozycja w tablicy procesów zajmowana przez zombie-go).

Gdy został odnaleziony odpowiedni potomek to funkcja zwraca numer pid znalezionego procesu. Jeśli nie został odnaleziony żaden potomek o odpowiednim numerze pid to funkcja zwraca błąd (`ECHILD`). Natomiast jeśli istniał taki potomek ale nie był w stanie `TASK_ZOMBIE` ani `TASK_STOPPED` to zwracane jest zero pod warunkiem ustawionej flagi `WNOHANG`. Kiedy flaga ta nie jest ustawiona to proces macierzysty zasypia (funkcja `schedule`) wcześniej jeszcze sprawdzając czy podczas działania funkcji nie przyszedł jakiś nieprzewidziany (różny od `SIGCHLD`) i niezablokowany sygnał. W takim przypadku funkcja zwraca błąd (`ERESTARTSYS`). Proces macierzysy w trakcie spania może zostać obudzony przez sygnał - po obudzeniu zaczyna przeglądać swoich potomków od początku. W każdym przypadku kończenia działania przez funkcje, proces macierzysty zostaje usunięty ze swojej kolejki `wait_chldexit`, a jego stan jest zmieniany na `TASK_RUNNING`.

Ogólne działanie funkcji przedstawia rysunek.



Rysunek 1: Ogólne działanie funkcji `wait`

6 Algorytm funkcji `sys_wait4` zapisany w pseudojęzyku

Dane wejściowe: `pid_t pid` - identyfikator procesu na którego będziemy czekać
`unsigned int *status` - tutaj zostanie wpisana przyczyna zakończenia działania potomka
`int options` - flagi sterujące działaniem funkcji
`struct rusage *ru` - wskaźnik do struktury opisującej zużycie zasobów przez potomka

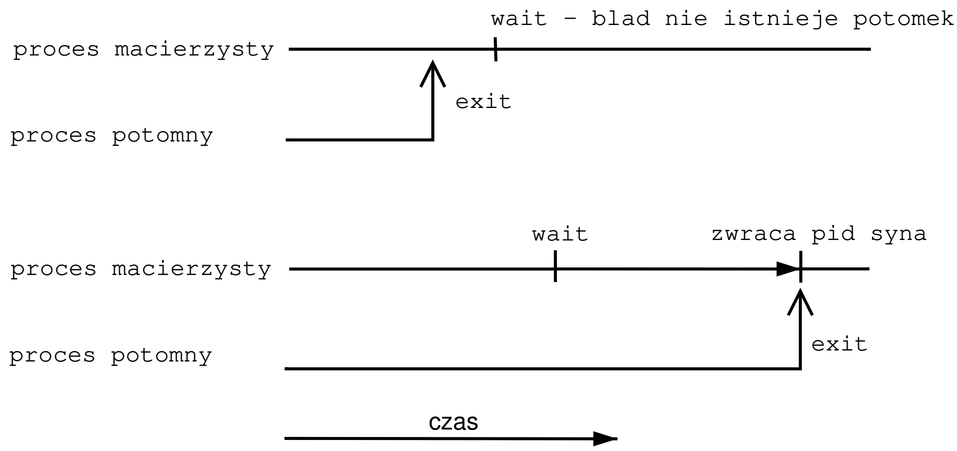
Wynik: identyfikator procesu potomnego na którego zaczekaliśmy, aktualizacja argumentu `status`

jeśli niepoprawny argument `options` **to**
  zwróć błąd (`EINVAL`)
koniec jeśli
wstaw proces do swojej kolejki `wait_chldexit` (pole `task_struct`)
zmień stan procesu wywołującego funkcję na `TASK_INTERRUPTIBLE` (dopuszczający przerwania)
wykonuj w pętli
  przejdź do kolejnego potomka
  jeśli identyfikator potomka nie spełnia warunku wywołania funkcji **to**
    omiń tego potomka
  koniec jeśli
  jeśli (ustawiona flaga `__WCLONE` i wartość pola `exit_signal` potomka wynosi `SIGCHLD`) lub (flag nie jest ustawiona i `exit_signal` jest różny od `SIGCHLD`) **to**
    omiń tego potomka
  koniec jeśli
  jeśli stan potomka = `TASK_STOPPED` **to**
    jeśli proces był już uwzględniony we wcześniejszym wywołaniu funkcji **to**
      omiń tego potomka
    koniec jeśli
    jeśli (ustawiona flaga `WUNTRACED` i potomek nie ma ustawionej flagi `PF_PTRACE`) lub (nie

ustawiona flaga `WUNTRACED` i potomek ustawił flagę `PF_PTRACE`) **to**
omiń tego potomka
koniec jeśli
zapisz dane o zużyciu zasobów przez potomka do struktury `ru`
wpisz pod adres `status` numer sygnału, który zatrzymał proces (wartość pola `exit_code` odpowiednio przesunięta i sformatowana)
do pola `exit_code` potomka wpisz 0 (inforamacja o tym że już czekaliśmy na danego potomka)
zwróć numer pid znalezionego procesu
koniec jeśli
jeśli stan potomka = `TASK_ZOMBIE` **to**
zwiększ procesowi wywołującemu funkcję czas wykorzystania procesora przez potomka (tryb jądra i użytkownika)
zapisz dane o zużyciu zasobów przez potomka do struktury `ru`
wpisz pod adres `status` wartość pola `exit_code` potomka
jeśli powiadomiono o zakończeniu działania potomka nie tego ojca co trzeba **to**
informuj orginalnego ojca o zakończeniu działania przez potomka
w przeciwnym przypadku
zwolnij miejsce w tablicy procesów po zakończonym potomku
koniec jeśli
zwróć numer pid znalezionego procesu
koniec jeśli
koniec pętli
jeśli nie ma procesu spełniającego warunki na numer pid **to**
zwróć błąd (`ECHILD`)
koniec jeśli
jeśli ustawiona flaga `WNOHANG` i każdy procesy o odpowiednim numerze pid działa **to**
zwróć 0
koniec jeśli
jeśli w trakcie działania funkcji nadszedł nieoczekiwany sygnał (różny od `SIGCHLD` i niezablokowany) **to**
zwróć błąd (`ERASTARTSYS`)
koniec jeśli
zaśnij (dopuszczając przerwania)
po obudzeniu ponownie przeglądaj synów
w każdym przypadku kończenia działania przez funkcje, usuń proces macierzysty ze swojej kolejki `wait_chldexit` i zmień jego stan na `TASK_RUNNING`

7 Funkcja `wait` i ignorowanie śmierci potomka

Proces może ustawić obsługę sygnału `SIGCHLD` na `SIG_IGN` (ignorować sygnał `SIGCHLD`). W takim wypadku proces macierzysty nie jest zainteresowany stanem końcowym swoich procesów potomnych, więc nie ma sensu utrzymywać w systemie procesów potomnych w stanie `TASK_ZOMBIE`. Dlatego też w momencie obsługi sygnału `SIGCHLD` - jeśli proces go ignoruje - jądro samo będzie wywoływać w pętli funkcję `sys_wait4` (z opcją `WNOHANG`) aż nie będzie procesów zombie. Pomimo tego w linuxie (wersja 2.2.17 i wcześniejsze) jeśli proces będzie spał w `wait` i przyjdzie do niego sygnał `SIGCHLD` (który proces ignoruje) to nie zdąży obsłużyć sygnału (tzn. sprawdzić po otrzymaniu sygnału, że ten sygnał jest przez niego ignorowany) zanim przeszuka swoich synów, więc znajdzie zombie-go, usunie go i zwróci jego pid. Powyższe systuacje przedstawia rysunek.



Rysunek 2: Funkcja `wait` i ignorowanie śmierci potomka

8 Zadania

Zadanie 1 Jak wygląda wywołanie funkcji `sys_wait4` odpowiadające wywołaniu funkcji `wait(s)` (`s` - zmienna int*)?

Zadanie 2 Załóżmy, że nie istniej sygnał śmierci potomka (`SIGCHLD`). Zamiast wysyłania takiego sygnału, funkcja `exit` wywołana przez kończącego działanie potomka budzi proces macierzysty - jeśli spał w `sys_wait4` to budzi się, znajduje zombie-go i działa zgodnie z algorytmem funkcji, jeśli natomiast nie spał wewnątrz `sys_wait4` to budzenie nie ma żadnego skutku. Jaki problem może wystąpić z tą implementacją?

Zadanie 3 Jak zadziała następujący program, wywołany z argumentem i bez?

```
#include <signal.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
  pid_t pid;
  int w;

  switch (pid=fork()) {
    case -1:
      perror("Blad przy forku\n");
```

```
        exit(-1);
      case 0:
        sleep(3);
        exit(0);
      default:
        signal(SIGCHLD,SIG_IGN);
        if (argc > 1)
          sleep(6);
        w=wait(0);
        if (w<0)
          perror("Blad przy wykonywaniu wait()");
        printf("Wartosc zwrocona przez wait: %d\n",w);
        exit(0);
    }
  };
```

9 Rozwiązania zadań

Zadanie 1 Wywołaniu `wait(s)` odpowiada wywołanie `sys_wait4(-1, s, 0, NULL)`.

Zadanie 2 Problem z taką implementacją funkcji czekania na zakończenie działania przez potomka polega na tym, że nie można usunąć procesu zombie aż do czasu gdy proces macierzysty nie wywoła funkcji `wait`. W przypadku gdy proces macierzysty będzie tworzył wielu potomków, lecz nigdy nie będzie wywoływał funkcji `wait` dojdzie do zapełnienia tablicy procesów procesami zombie (oczywiście dojdzie do takiej sytuacji wtedy gdy proces macierzysty nie będzie ignorował śmierci potomka, w przypadku ignorowania sygnału `SIGCHLD`, jądro systemu samo będzie wywoływać funkcję `sys_wait4`, usuwając wszystkie procesy zombie).

Zadanie 3 W przypadku wywołania programu bez argumentu, proces macierzysty zaśnie w funkcji `wait`, podczas gdy potomek będzie się jeszcze wykonywał. Gdy potomek zakończy działanie wyśle do ojca sygnał `SIGCHLD`. Proces macierzysty zostanie obudzony, lecz jak to zostało opisane w sekcji powyżej, "nie zdąży się zorientować" że sygnał, który otrzymał jest przez niego ignorowany. W rezultacie zostanie znaleziony proces potomny w stanie `TASK_ZOMBIE` i funkcja `wait` zwróci numer pid odszukanego potomka.

Kiedy program zostanie wywołany z argumentem, funkcja `wait` zwróci błąd (brak procesu potomnego). Stanie się tak ponieważ proces macierzysty ignoruje śmierć potomka. W przeciwieństwie do przypadku opisywanego powyżej sygnał śmierci potomka przyjdzie do procesu zanim wywoła on funkcję `wait`. W takiej sytuacji jądro systemu wykona funkcję `sys_wait4` zamiast procesu macierzystego, usuwając z systemu proces potomny, który po skończonym działaniu zamienił się w zombie-go. W rezultacie gdy proces macierzysty wywoła funkcję `wait`, nie znajdzie żadnego procesu potomnego i zwróci błąd.

10 Literatura i źródła

1. Bach M. J. *Budowa systemu operacyjnego UNIX* Warszawa, WNT 1995
2. Stevens W. R. *Programowanie zastosowań sieciowych w systemie Unix* Warszawa, WNT 1995
3. Źródła jądra:
   (a) `kernel/exit.c` - funkcja `sys_wait4`
   (b) `kernel/signal.c` - funkcja `notify_parent`
   (c) `arch/i386/kernel/ptrace.c` - funkcja `sys_ptrace`