Procesy

dr inż. Sławomir Samolej
Katedra Informatyki i Automatyki
Politechnika Rzeszowska

Program przedmiotu oparto w części na materiałach opublikowanych na:
http://wazniak.mimuw.edu.pl/

oraz

Na materiałach opracowanych przez
dr inż. Jędrzeja Ułasiewicza:
jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl

POSIX

- Standard POSIX służy ujednoliceniu systemów rodziny Unix
- Od systemów zgodnych z POSIX oczekuje się, że nie będzie potrzeby wprowadzania żadnych modyfikacji w kodach źródłowych aplikacji dla nich przygotowanych żeby dokonać przeniesienia aplikacji.
- Oznacza to, że wszystkie systemy operacyjne zgodne ze standardem POSIX będą oferować funkcje o tych samych nazwach realizujące taką samą funkcjonalność w systemie.
- Funkcje omawiane w ramach tego wykładu są zgodne z POSIX.

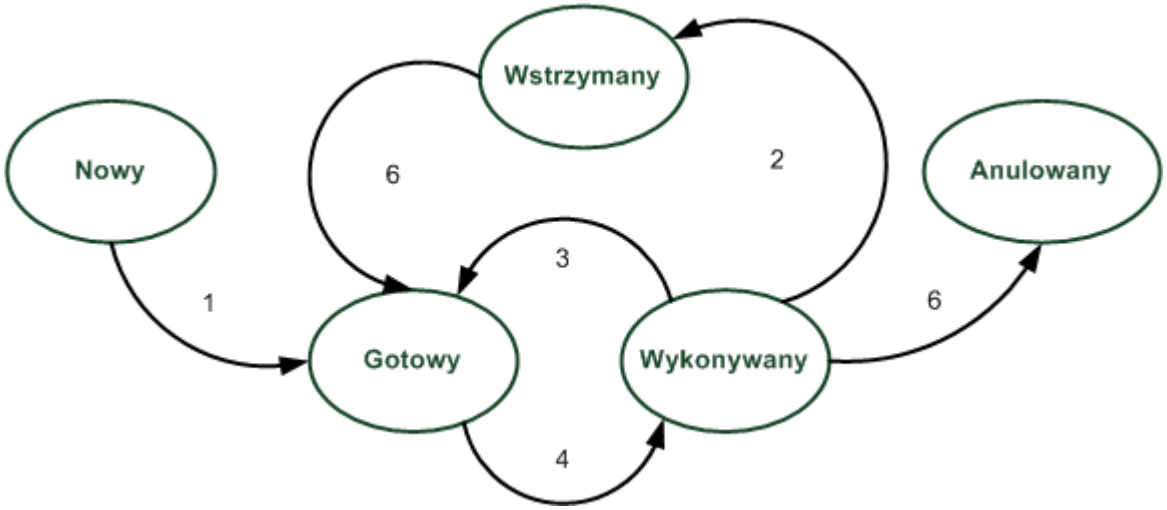
Proces

- **Proces** to przestrzeń adresowa i pojedynczy wątek sterujący, który działa w tej przestrzeni, oraz potrzebne do tego zasoby systemowe
- Minimalne zasoby do wykonywania się procesu to:
  - Procesor
  - Pamięć
  - Urządzenia wejścia-wyjścia
- W systemach wielozadaniowych w istocie każda instancja działającego programu jest procesem.

Przypomnienie – uruchomienie procesu

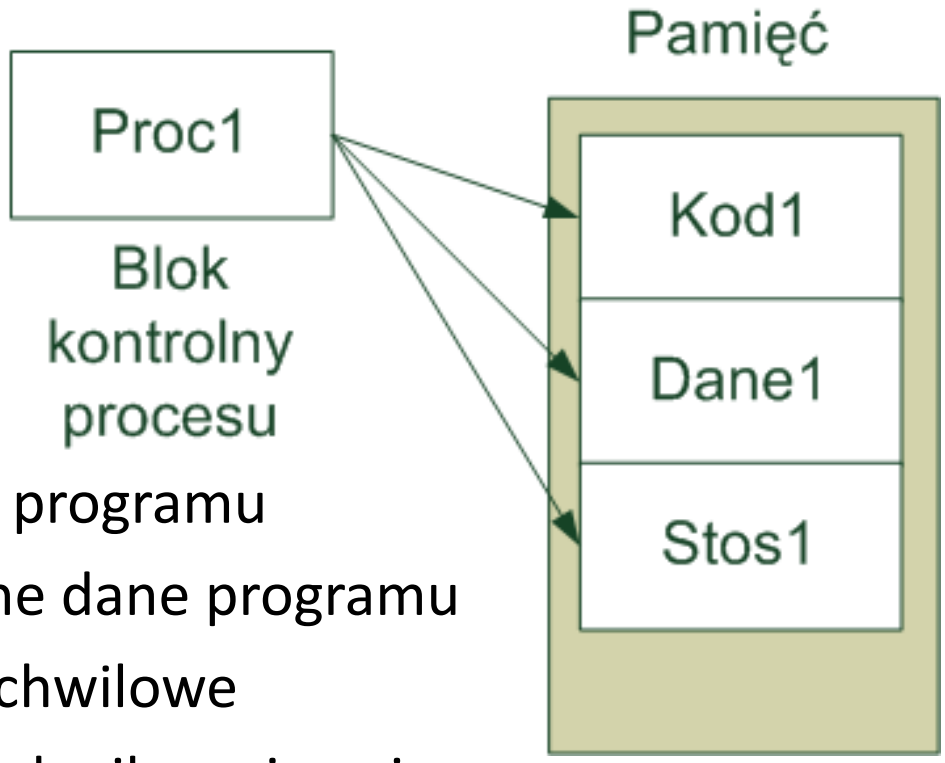
- Podczas uruchamiania programu w systemie operacyjnym następuje przeniesienie kodu programu z jakiegoś nośnika (najczęściej systemu plików) do pamięci operacyjnej.
- Kod programu jest „otaczany” pewnymi strukturami danych identyfikującymi go na czas wykonywania i zapewniającymi jego ochronę (tzw. blok kontrolny procesu).
- Następuje także powiązanie (planowanie przydziału) z programem zasobów systemu mikroprocesorowego, z których będzie korzystał.
- Zasobami przyszłego procesu są czas procesora, pamięć, system plików oraz urządzenia wejścia-wyjścia. Tak przygotowany do wykonywania program staje się **procesem**, który jest gotowy do wykonywania.

Przypomnienie – podstawowe stany procesu



1. Utworzenie procesu
2. Proces żąda zasobu, który nie jest dostępny
3. Wystąpiło przerwanie (wywłaszczenie) lub proces zwolnił procesor dobrowolnie
4. Proces został wybrany do wykonywania
5. Potrzebny zasób został zwolniony (przerwanie wej/wyj lub inicjatywa bieżącego procesu)
6. Zakończenie procesu

Przypomnienie – struktury danych powiązane z procesem



- Segment kodu – instrukcje programu
- Segment danych – statyczne dane programu
- Segment stosu – zmienne chwilowe
- Segment sterty – zmienne chwilowe jawnie alokowane i zwalniane przez programistę
- Blok kontrolny procesu – dane które o procesie przechowuje system operacyjny

Przypomnienie – blok kontrolny procesu

- Identyfikator procesu (PID, grupa procesów)
- Uwierzytelnienia (id. użytkownika i grupy)
- Indywidualność (dot. biblioteki emulacji)
- Wektor argumentów (parametry wywołania programu)
- Wektor środowiska (zmienne NAZWA=WARTOŚĆ)
- Kontekst planowania: rejestry, priorytety, nieobsłużone sygnały
- Informacje rozliczeniowe
- Tablica plików
- Tablica obsługi sygnałów
- Kontekst pamięci wirtualnej

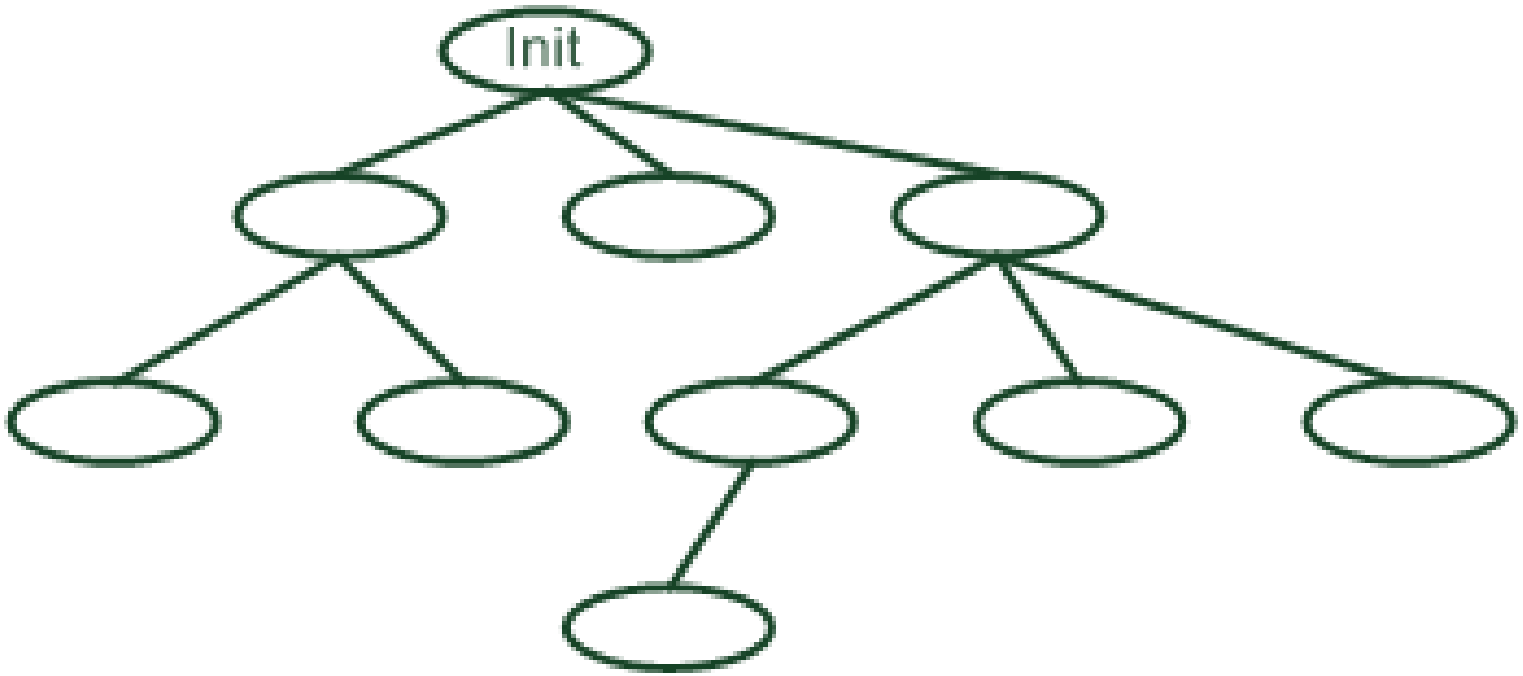
Tożsamość procesu

Środowisko procesu (dziedziczone od rodzica)

Kontekst procesu

Drzewo procesów

- Procesy tworzą drzewo
- Każdy proces ma dokładnie jeden proces – rodzic
- Procesy mogą posiadać wiele procesów-dzieci



Polecenia Linux zarządzające procesami

- **ps** – pobieranie informacji o aktywnych procesach
  ps –ef – wszystkie procesy w systemie
  ps – axjf – drzewo procesów
  ps – sLf – informacja o wątkach
- **top** – dynamiczna lista aktywnych procesów w systemie
- **pstree** – drzewo procesów w systemie
- **monitor systemu**
- **nice** – uruchomienie programu z zadanym priorytetem
- **renice** – zmiana priorytetu uruchomionego procesu
- **kill** – przesłanie sygnału do procesu pracującego w syytemie operacyjnym

Funkcje informujące i zarządzające procesami

Funkcja	Opis
pid_t getpid(void);	Zwróć swój identyfikator procesu
pid_t getppid(void);	Zwróć identyfikator procesu macierzystego
unsigned int sleep(unsigned int seconds);	„Uśpij” bieżący wątek/proces na zadaną ilość sekund
int pause(void);	Zatrzymaj wykonywanie programu do momentu, gdy otrzyma on sygnał, który go „zabije” lub który zostanie obsłużony

Uruchomienie programu z innego programu

```
#include <stdlib.h>
int system (const char *string);
```

```
#include <stdlib.h>
#include <stdio.h>
int main()
{
printf("Running ps with system\n");
system("ps -ax");
printf("Done.\n");
exit(0);
}
```

```
$ ./system1
Running ps with system
PID TTY STAT TIME COMMAND
1 ? S 0:05 init
2 ? SW 0:00 [keventd]
...
1262 pts/1 S 0:00 /bin/bash
1273 pts/2 S 0:00 su -
1274 pts/2 S 0:00 -bash
1463 pts/1 S 0:00 oclock -transparent -
geometry 135x135-10+40
1465 pts/1 S 0:01 emacs Makefile
1480 pts/1 S 0:00 ./system1
1481 pts/1 R 0:00 ps -ax
Done.
```

Wada: nowy program jest uruchamiany przez powłokę.

Zastępowanie procesu (1)

- Rodzina funkcji: **exec*** (execl, execlp, execle, execv, execvp, execve) zastępuje bieżący proces innym, tworzonym na podstwaie podanych argumentów.

```
#include <unistd.h>
#include <stdio.h>
int main()
{
printf("Running ps with execlp\n");
execlp("ps", "ps", "-ax", 0);
printf("Done.\n");
exit(0);
}
```

```
$ ./pexec
Running ps with execlp
PID TTY STAT TIME COMMAND
1 ? S 0:05 init
2 ? SW 0:00 [keventd]
...
1262 pts/1 S 0:00 /bin/bash
1273 pts/2 S 0:00 su -
1274 pts/2 S 0:00 -bash
1465 pts/1 S 0:01 emacs Makefile
1514 pts/1 R 0:00 ps –ax
```

exec*()

```
#include <unistd.h>
char **environ;
int execl(const char *path, const char *arg0, ..., (char *)0);
int execlp(const char *path, const char *arg0, ..., (char *)0);
int execle(const char *path, const char *arg0, ..., (char *)0, char *const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[], char *const envp[]);
```

- path – nazwa programu
- arg0 … argn – argumenty programu
- dla wersji funkcji execv – argumenty mogą być przekazane przez tablicę argv
- Funkcje z przyrostkiem p przeszukują zmienną środowiskową PATH (echo $PATH)
- Zmienna environ może zawierać wartość nowego środowiska programu
- W funkcjach execle i execve jest dodatkowa zmienna do przekazania tablicy ciągów, która zostanie wykorzystana jako nowe środowisko programu

Zastępowanie procesu (2)

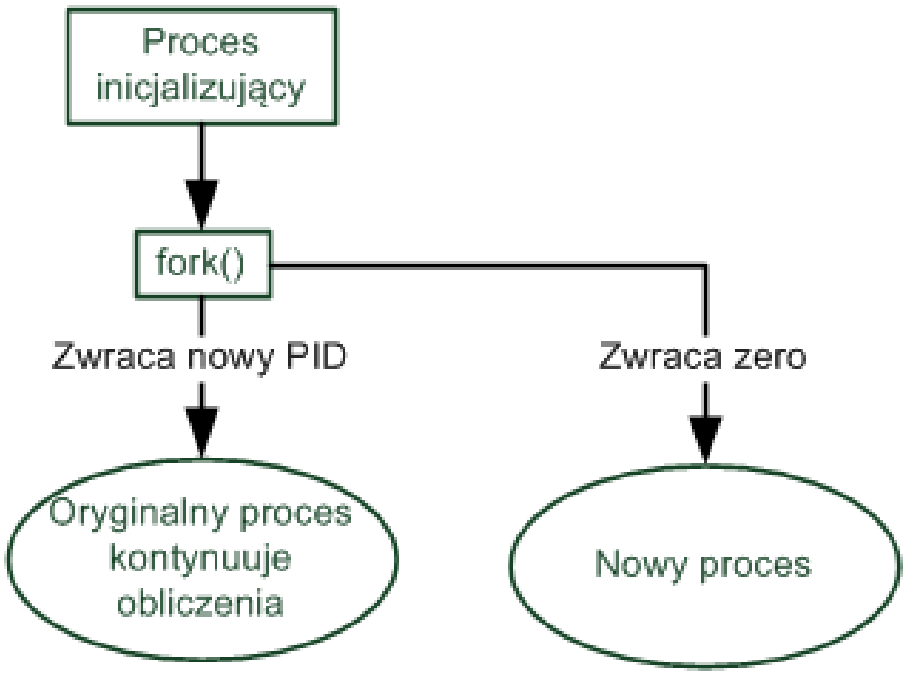
- Jeśli wystąpiło „zastąpienie” procesu to PID procesu jest taki sam jak procesu, w którym nastąpiło wywołanie funkcji exex*
- Efekt jest taki, jakby program rozpoczął wykonywać nowy kod z nowego pliku wykonywalnego, określonego w argumentach funkcji exec*
- Funkcje exec* zwykle nie powracają, chyba, że wystąpi błąd. Wtedy funkcja zwraca wartość -1 i ustawiana jest zmienna **errno**.
- Nowy proces uruchomiony przez exec* dziedziczy między innymi deskryptory pliku. Zamykane są natomiast wszystkie strumienie katalogowe, otwarte w pierwotnym procesie.

Na marginesie: zmienna **errno**

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
int main(int argc, char **argv)
{  int files = argc - 1;
   FILE *file;   char *fname;   int i;
   if (!files) { fprintf(stderr, "Usage: %s <files>\n", argv[0]);
                 return 1;}
   for (i = 1; i <= files; ++i) {
        fname = argv[i];
        file = fopen(fname, "r");
        if (file == NULL) {
           fprintf(stderr, "Error while trying to open '%s': %s\n", fname,
           strerror(errno));}
        else {  fprintf (stderr, "'%s' opened successfully\n", fname);
                fclose (file);}
    }
        return 0;
}
```

Duplikowanie procesu (1)

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```




```
pid_t new_pid;
new_pid = fork();
switch(new_pid) {
case -1 : /* Error */
break;
case 0 : /* We are child */
break;
default : /* We are parent */
break;
}
```

Duplikowanie procesu (2)

- Wywołanie systemowe tworzy nowy proces potomny, który jest identyczny z procesem wywołującym, ale posiada unikatowy PID, a jego PPID jest ustawiany na proces wywołujący.
- Nowy proces wywołujący dziedziczy po procesie macierzystym
  - Przestrzeń danych (zmienne)
  - Otwarte deskryptory i strumienie katalogowe
- Wywołanie fork w procesie macierzystym zwraca PID nowego procesu potomnego
- Nowy proces pracuje dokładnie tak samo, jak oryginał, ale w procesie potomnym fork zwraca zero (dzięki temu procesy mogą rozróżnić, który jest który)

Duplikowanie procesu – przykładowa aplikacja

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{       pid_t pid; int n;
        char *message;
        printf("fork program starting\n");
        pid = fork();
        switch(pid)
        {       case -1:        perror("fork failed");
                                exit(1);
                case 0:         message = "This is the child";
                                n = 5;
                                break;
                default:        message = "This is the parent";
                                n = 3;
                                break;
}       for(; n > 0; n--) {
        puts(message); sleep(1);
        }
        exit(0);}
```

Rezultat działania programu

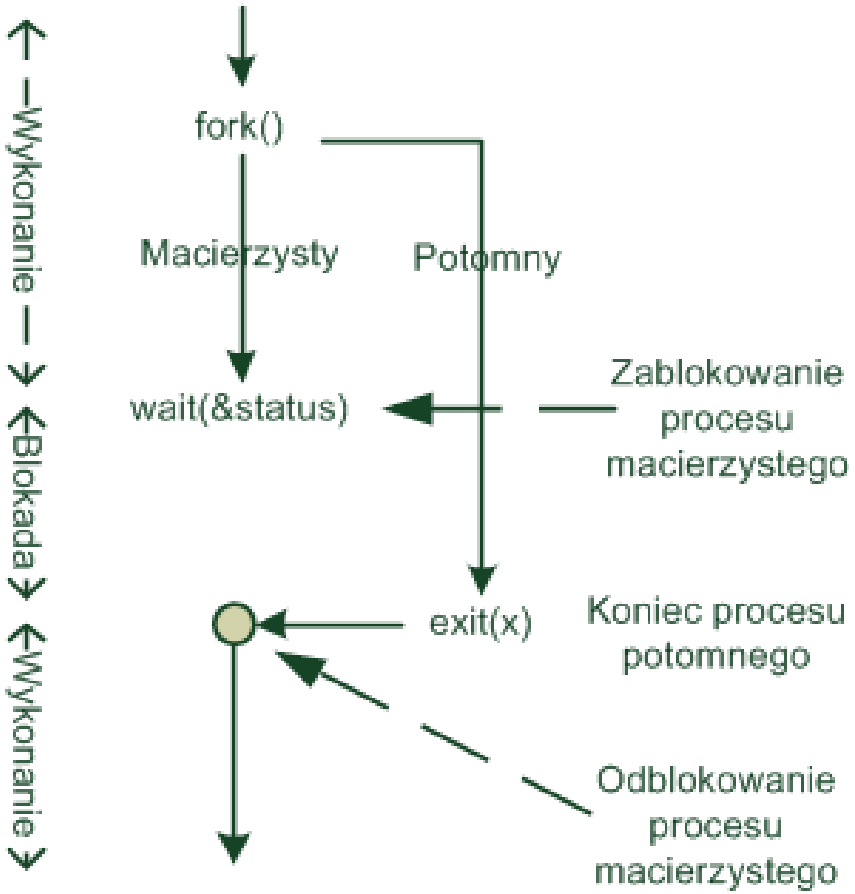
```
$ ./fork1
fork program starting
This is the parent
This is the child
This is the parent
This is the child
This is the parent
This is the child
$ This is the child
This is the child
```

- Program działa jako 2 procesy
- Proces potomny wysyła komunikat 5 razy, macierzysty -3
- Proces macierzysty kończy działanie przed zakończeniem procesu dziecka
- W komunikaty wmieszał się znak powłoki…

Oczekiwanie na proces (1)

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_val);
```

- Gdy proces potomny nie zakończył się funkcja wait powoduje zablokowanie procesu macierzystego aż do zakończenia się procesu potomnego. Gdy ten się zakończy zwracany jest jego PID oraz status
- Gdy proces potomny zakończył się zanim wykonano funkcję wait nie występuje blokada procesu macierzystego. Funkcja zwraca PID zakończonego procesu oraz jego status
- Gdy brak jakichkolwiek procesów potomnych funkcja wait zwraca –1.



Odczytywanie statusu przechwyconego przez funkcję wait

Makro	Definicja
WIFEXITED(stat_val)	Niezerowe, jeśli proces potomny normalnie zakończył pracę
WEXITSTATUS(stat_val)	Jeśli WIFEXITED jest niezerowe, zwraca kod wyjściowy procesu potomnego
WIFSIGNALED(stat_val)	Niezerowe, jeśli proces potomny zakończył pracę po otrzymaniu nie przechwyconego sygnału
WTERMSIG(stat_val)	Jeśli WIFSIGNALED jest niezerowe, zwraca numer sygnału
WIFSTOPPED(stat_val)	Niezerowe, jeśli proces potomny zatrzymał się po otrzymaniu sygnału
WSTOPSIG(stat_val)	Jeśli WIFSTOPPED jest niezerowe, zwraca numer sygnału

Zastosowanie wait – przykładowa aplikacja

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
int main()
{ pid_t pid;  char *message;  int n;
  int exit_code;
  printf("fork program starting\n");
  pid = fork();
 switch(pid)
{ case -1:  perror("fork failed");
            exit(1);
 case 0:    message = "This is the child";
            n = 5;
            exit_code = 37;
            break;
default:    message = "This is the parent";
            n = 3;
            exit_code = 0;
            break;
}
```

```
for(; n > 0; n--) {  puts(message);
                     sleep(1);
}

if (pid != 0) {
int stat_val;
pid_t child_pid;
child_pid = wait(&stat_val);
printf("Child has finished: PID = %d\n",
child_pid);
if(WIFEXITED(stat_val))
printf("Child exited with code %d\n",
WEXITSTATUS(stat_val));
else
printf("Child terminated abnormally\n");
}
exit(exit_code);
}
```

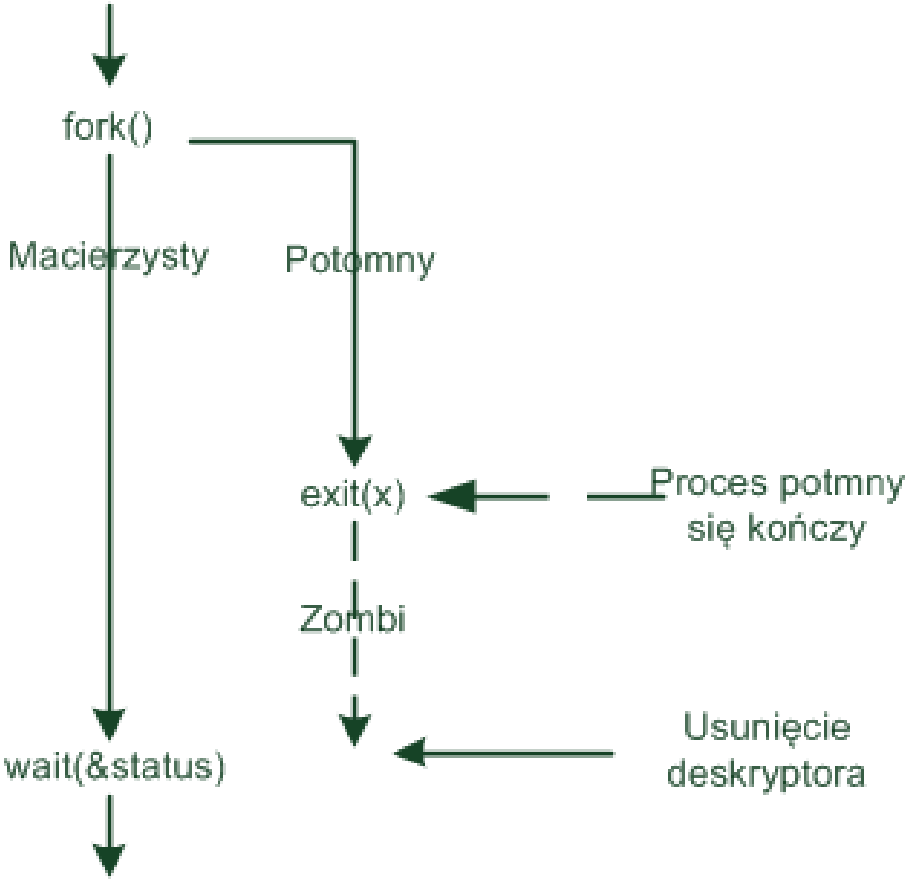
Rezultat programu

```
$ ./wait
fork program starting
This is the child
This is the parent
This is the parent
This is the child
This is the parent
This is the child
This is the child
This is the child
Child has finished: PID = 1582
Child exited with code 37
$
```

- Proces macierzysty korzysta z wywołania wait i zawiesza swoje działanie
- Kiedy zostaje wywołane exit w programie potomnym program macierzysty wznawia pracę
- Program macierzysty przechwytuje wartość zwracaną przez wait i informuje o statusie zakończenia procesu potomnego

Procesy zombi

- Kiedy proces potomny kończy pracę, jego powiązanie z procesem macierzystym jest podtrzymywane, dopóki ten nie zakończy działania albo nie wykona wait
- Proces potomny nie jest już aktywny, ale pozostaje po nim wpis w tablicach opisu procesów



Wprowadzenie procesu zombie – przykładowa aplikacja

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{       pid_t pid; int n;
        char *message;
        printf("fork program starting\n");
        pid = fork();
        switch(pid)
        {       case -1: perror("fork failed");
                         exit(1);
                case 0:  message = "This is the child";
                         n = 3;
                         break;
                default: message = "This is the parent";
                         n = 5;
                         break;
}       for(; n > 0; n--) {puts(message);
        sleep(1);
        }
        exit(0);}
```

```
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
004 S 0 1273 1259 0 75 0 - 589 wait4 pts/2 00:00:00 su
000 S 500 1465 1262 0 75 0 - 2569 schedu pts/1 00:00:01 emacs
000 S 500 1603 1262 0 75 0 - 313 schedu pts/1 00:00:00 fork2
003 Z 500 1604 1603 0 75 0 - 0 do_exi pts/1 00:00:00 fork2 <defunct>
000 R 500 1605 1262 0 81 0 - 781 - pts/1 00:00:00 ps
```

Gdy proces macierzysty zakończy swoją pracę nieprawidłowo, proces potomny otrzyma proces macierzysty z identyfikatorem 1 (init).

Oczekiwanie na określony proces

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

Wywołanie funkcji waitpid z parametrami:

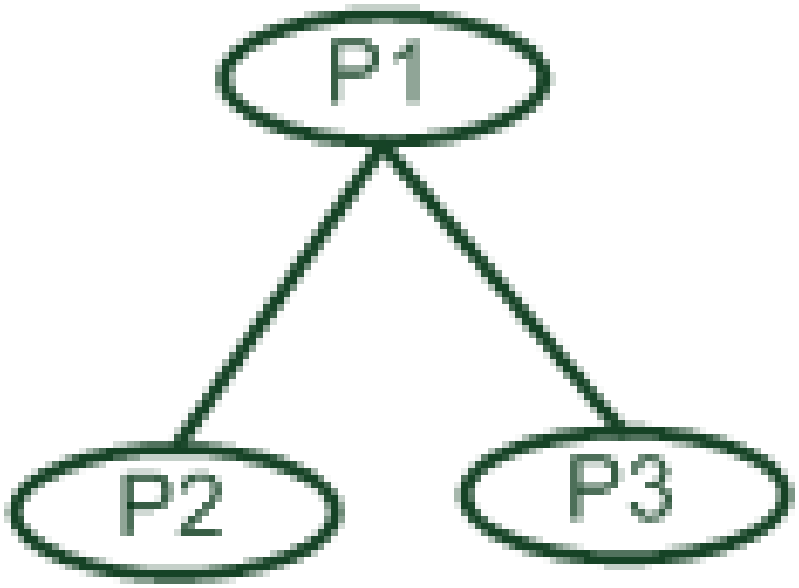
waitpid(child_pid, (int *) 0, WNOHANG);

pozwala na monitorowanie stanu procesu potomnego.
Funkcja zwróci:

- 0 – jeśli proces potomny nie zakończył pracy
- PID potomka, jeśli proces potomny zakończy pracę normalnie
- -1 i ustawi errno, gdy zostanie wykryty błąd

Użycie fork do utworzenia 2 procesów potomnych

```
Kod wspólny
if(fork() == 0) {
   Kod procesu potomnego P2
    exit(0);
}
if(fork() == 0) {
    Kod procesu potomnego P3
    exit(0);
 }
Kod procesu macierzystego P1
```



Użycie fork do tworzenia kaskady procesów potomnych

```
Kod wspólny
if(fork() == 0) {
  if(fork() == 0 {
     Kod procesu potomnego P3
  } else {
     Kod procesu potomnego P2
  }
 }
Kod procesu macierzystego P1
```

Wątki - zapowiedź

- Procesy w Linux mogą współpracować, przesyłać między sobą komunikaty, przerywać sobie nawzajem pracę, a nawet współdzielić segmenty pomięci.
- Są jednak osobnymi bytami i „niezbyt chętnie” wymieniają się swoimi zmiennymi.
- Istnieje możliwość pisania aplikacji współbieżnych w oparciu o **wątki**, gdzie w ramach jednego procesu powołuje się do życia pewne pod-procesy, które w naturalny sposób współdzielą obszar danych należących do głównego procesu
- Programowaniu wielowątkowemu poświęcony będzie osobna część wykładu.