Procesy, pliki, potoki, sygnały - uzupełnienie

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

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 podstawie 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.

Blokowanie dostępu do pliku

```
#include <sys/file.h>
int lockf(int fd, int cmd, off_t len);
```

Parametry funkcji:

fd	Uchwyt pliku
cmd	Specyfikacja operacji: F_LOCK, F_ULOCK, F_TEST,F_TLOCK
len	Zakres blokowania (o ile bajtów od bieżącego położenia plik ma być zablokowany) (0 - blokowany jest cały plik)

Wartości zwracane:

-1	Błąd
>0	Sukces

Specyfikacje operacji:

F_LOCK	Zablokuj dostęp do pliku na długości zakres od pozycji bieżącej
F_ULOCK	Zwolnij dostęp do pliku.
F_TEST	Testuj czy fragment pliku jest zablokowany przez inny proces
F_TLOCK	Testuj czy fragment pliku jest zablokowany przez inny proces. Gdy nie to zajmij plik

Kontrolowany zapis 2 procesów do 1 pliku - przykład

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

int main()
{
   int fd, i; pid_t pid;
   char buf1[]="To jest tekst1\n";
   char buf2[]="To jest tekst2\n";
   printf("Tworze proces potomny...\n");
   fd=open("plik1.txt",O_CREAT|O_WRONLY,0777);
   pid=fork();
   switch(pid)
   {   case -1: perror("Blad...\n");
                exit(1);
       case  0: for(i=0;i<10;i++)
                { lockf(fd,F_LOCK,0);
                  write(fd,buf1,sizeof(buf1));
                  sleep(2);
                  lockf(fd,F_ULOCK,0);  } exit(112);
default: { int stat_val;
                for(i=0;i<10;i++)
                {
                  if(lockf(fd,F_TEST,0)==-1)
                  {  printf("Zablokowany...\n");
                  }
                  else
                  {   lockf(fd,F_LOCK,0);
                      write(fd,buf2,sizeof(buf2));
                      lockf(fd,F_ULOCK,0);
                  }
                  sleep(1);
                }
                wait(&stat_val);
                exit(0);
               }
}}
```

Potoki procesowe

```
#include <stdio.h>
FILE *popen(const char *command, const char *open_mode);
int pclose(FILE *stream_to_close);
```

- popen
  - uruchamia inny programu jako nowy proces oraz przekazuje lub odbiera od niego dane
  - parametr *command* jest nazwą programu, który należy uruchomić wraz ze wszystkimi parametrami wywołania, parametr *open_mode* musi mieć wartość *r* lub *w.*
- pclose
  - czeka do momentu zakończenia uruchomionego procesu
  - zwraca kod wyjściowy uruchomionego procesu
  - jeśli proces uruchamiający wykona funkcję wait tracimy wówczas kod wyjściowy, *pclose* zwróci *-1*, a zamienna errno będzie zawierała wartość ECHILD

Przykład – użycie popen i pclose

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{  FILE *read_fp;
   char buffer[BUFSIZ + 1];
   int chars_read;
   memset(buffer, '\0', sizeof(buffer));
   read_fp = popen("uname -a", "r");
   if (read_fp != NULL)
   { chars_read = fread(buffer, sizeof(char), BUFSIZ, read_fp);
     if (chars_read > 0)
     { printf("Output was:-\n%s\n", buffer);
     }
     pclose(read_fp);
     exit(EXIT_SUCCESS);
   }
   exit(EXIT_FAILURE);
}
```

Output was:-
Linux gw1 2.4.20-8 #1 Thu Mar 13 17:54:28 EST 2003 i686 i686 i386 GNU/Linux

Wysyłanie danych ze pomocą popen

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
int main()
{ FILE *write_fp;
  char buffer[BUFSIZ + 1];
  sprintf(buffer, “Once upon a time, there was...\n”);
  write_fp = popen(“od -c”, “w”);
  if (write_fp != NULL)
  { fwrite(buffer, sizeof(char), strlen(buffer), write_fp);
    pclose(write_fp);
    exit(EXIT_SUCCESS);
  }
  exit(EXIT_FAILURE);
}
```

```
$ ./popen2
0000000 O n c e u p o n a t i m e
0000020 , t h e r e w a s . . . \n
0000037
```

Przekazywanie większej ilości danych

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{ FILE *read_fp;
  char buffer[BUFSIZ + 1];
  int chars_read;
  memset(buffer, '\0', sizeof(buffer));
  read_fp = popen("ps -ax", "r");
  if (read_fp != NULL)
  { chars_read = fread(buffer, sizeof(char),
                 BUFSIZ, read_fp);
    while (chars_read > 0)
    {  buffer[chars_read - 1] = '\0';
       printf("Reading:-\n %s\n", buffer);
       chars_read = fread(buffer, sizeof(char),
                          BUFSIZ, read_fp);
    }
    pclose(read_fp);
    exit(EXIT_SUCCESS);
  }
  exit(EXIT_FAILURE);
}
```

```
$ ./popen3
Reading:-
PID TTY STAT TIME COMMAND
1 ? S 0:04 init
2 ? SW 0:00 [kflushd]
3 ? SW 0:00 [kpiod]
4 ? SW 0:00 [kswapd]
5 ? SW< 0:00 [mdrecoveryd]
...
240 tty2 S 0:02 emacs draft1.txt
Reading:-
368 tty1 S 0:00 ./popen3
369 tty1 R 0:00 ps -ax
...
```

Potoki i exec

Producent:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main()
{ int data_processed; int file_pipes[2];
  const char some_data[] = “123”;
  char buffer[BUFSIZ + 1]; pid_t fork_result;
  memset(buffer, ‘\0’, sizeof(buffer));
  if (pipe(file_pipes) == 0)
  {  fork_result = fork();
     if (fork_result == (pid_t)-1)
     {   fprintf(stderr, “Fork failure”); exit(EXIT_FAILURE);}
     if (fork_result == 0)
     { sprintf(buffer, “%d”, file_pipes[0]);
       (void)execl(“pipe4”, “pipe4”, buffer, (char *)0);
       exit(EXIT_FAILURE);}
     else
     { data_processed = write(file_pipes[1], some_data,
       strlen(some_data));
       printf(“%d - wrote %d bytes\n”, getpid(), data_processed);}
   }
exit(EXIT_SUCCESS);}
```

Konsument:

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int argc, char *argv[])
{
int data_processed;
char buffer[BUFSIZ + 1];
int file_descriptor;
memset(buffer, ‘\0’, sizeof(buffer));
sscanf(argv[1], “%d”, &file_descriptor);
data_processed = read(file_descriptor,
buffer, BUFSIZ);
printf(“%d - read %d bytes: %s\n”,
getpid(), data_processed, buffer);
exit(EXIT_SUCCESS);
}
```

```
$ ./pipe3
980 - wrote 3 bytes
981 - read 3 bytes: 123
```

Jak to działa?

- Program producenta korzysta z funkcji pipe, aby stworzyć potok, a następnie z funkcji fork, aby utworzyć nowy proces
- Zapisuje w buforze „bufor” deskryptor pliku do odczytu otrzymany z funkcji pipe
- W ramach procesu potomnego wywoływana jest funkcja exec uruchamiająca inny program (konsument), do której jako parametry wywołania przekazywany jest deskryptor pliku do odczytu.
- Program konsumenta pobiera deskryptor pliku do odczytu i z niego czyta.

Sygnały - wprowadzenie

- Sygnał to zdarzenie w systemie Linux powstałe w odpowiedzi na zaistnienie pewnych okoliczności
- Po otrzymaniu sygnału proces może podjąć określone czynności
- Sygnały są generowane przez niektóre błędy (naruszenie segmentów pamięci, błędy jednostki zmiennoprzecinkowej)
- Mogą być generowane przez powłokę i programy obsługi terminala, aby wykonać przerwanie.
- Mogą być też jawnie wysyłane z jednego procesu do drugiego w celu przesłania informacji lub modyfikacji pracy procesu.
- Na poziomie interfejsu można:
  - Generować sygnały
  - Przechwytywać sygnały
  - Wykonywać na ich podstawie pewne czynności
  - Zignorować (ale nie wszystkie).

Tablica sygnałów

Nazwa sygnału	Opis
SIGABORT	*Przerwanie procesu
SIGALRM	Zegar alarmowy
SIGFPE	*Wyjątek związany z jednostką zmiennoprzecinkową
SIGHUP	Zawieszenie
SIGILL	*Nielegalna instrukcja
SIGINT	Przerwanie z terminala
SIGKILL	„Zabójstwo” (nie można go przechwycić ani zignorować)
SIGPIPE	Zapis do potoku bez odbiorcy
SIGQUIT	Sygnał zakończenia terminala
SIGSEGV	*Dostęp do niewłaściwego segmentu pamięci
SIGTERM	Zakończenie
SIGUSR1	Sygnał 1 zdefiniowany przez użytkownika
SIGUSR2	Sygnał 1 zdefiniowany przez użytkownika

Reakcja procesów na sygnały

- Jeśli proces otrzyma jeden z wymienionych wyżej sygnałów i nie jest przygotowany na jego przechwycenie, to zostaje natychmiast zakończony
- W przypadkach sygnałów oznaczonych * mogą zostać podjęte w systemie czynności zależne od implementacji
- W chwili takiego zakończenia procesu w jego bieżącym katalogu tworzony jest plik „core” zawierający zrzut pamięci

Sygnały dodatkowe

Nazwa sygnału	Opis
SIGCHLD	Proces potomny zatrzymał się lub zakończył pracę
SIGCONT	Wznowienie wykonania, o ile proces był zatrzymany
SIGSTOP	Zatrzymanie wykonywania (nie można go przechwycić ani zignorować)
SIGTSTP	Sygnał zatrzymania terminala
SIGTTIN	Proces pracujący w tle próbuje przeprowadzić odczyt
SIGTTOU	Proces pracujący w tle próbuje przeprowadzić zapis

- SIGCHILD można zastosować do sterowania procesami potomnymi
- Pozostałe, poza SIGCONT powodują zatrzymanie otrzymujących je procesów
- Normalnie skonfigurowany terminal po przyciśnięciu kombinacji Ctrl+C powoduje wysłanie sygnału SIGINT do pierwszoplanowego procesu uruchomionego na tym terminalu.

Manipulowanie sygnałami

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

- Funkcja signal przyjmuje 2 parametry:
  - Sygnał, który należy przechwycić lub zignorować (sig)
  - Funkcję, którą należy wywołać po otrzymaniu sygnału, która:
    - Musi przyjmować pojedynczy argument tupu int, a sama być typu void
- Funkcja signal zwraca z kolei funkcję tego samego typu – poprzednią wartość funkcji ustawionej do obsługi sygnału, albo jedną z wartości specjalnych:
  - SIG_IGN = zignorować sygnał
  - SIG_DFL = przywrócić domyślne zachowanie
- W systemie Linux domyślne zachowanie przywracane jest automatycznie

Prosta obsługa sygnałów - przykład

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
void ouch(int sig)
{
printf("OUCH! - I got signal
%d\n", sig);
(void) signal(SIGINT, SIG_DFL);
}
int main()
{
(void) signal(SIGINT, ouch);
while(1) {
printf("Hello World!\n");
sleep(1);
}
}
```

```
$ ./ctrlc1
Hello World!
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
Hello World!
Hello World!
^C
$
```

Sygnał jest domyślnie obsługiwany tylko raz. Do ponownej obsługi trzeba procedurę obsługi ponownie „ustanowić" (pojawia się możliwość niezdążenia z ponowną obsługą).

Wysyłanie sygnałów

Wysłanie sygnału do dowolnego procesu:

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Funkcja alarm planuje dostarczenie sygnału *SIGALRM* za *seconds* sekund

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds);
```

Symulator funkcji alarm - budzik

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
static int alarm_fired = 0;
void ding(int sig)
{ alarm_fired = 1;
}
int main()
{ pid_t pid;
  printf("alarm application starting\n");
  pid = fork();
  switch(pid) {
  case -1:   /* Failure */
             perror("fork failed");
             exit(1);
  case 0:    /* child */
             sleep(5);
             kill(getppid(), SIGALRM);
             exit(0);
  }
```

```
/* if we get here we are the parent process */
printf("waiting for alarm to go off\n");
(void) signal(SIGALRM, ding);
pause();
if (alarm_fired) printf("Ding!\n");
printf("done\n");
exit(0);
}
```

```
$ ./alarm
alarm application starting
waiting for alarm to go off
<5 second pause>
Ding!
done
$
```

Jeśli zastosujemy pause, a sygnał już został wysłany do naszego procesu, to możemy spowodować „zawieszenie" procesu.

Odporny interfejs sygnałowy

```
#include <signal.h>
int sigaction( int sig, const struct sigaction *act,
               struct sigaction *oact);
```

- Struktura sigaction jest zdefiniowana w pliku nagłówkowym signal.h i musi się składać przynajmniej z następujących elementów:
  - void (*) (int) sa_handler — funkcja, SIG_DFL lub SIG_IGN
  - sigset_t sa_mask — sygnały, które należy zablokować w sa_handler
  - int sa_flags — modyfikatory reakcji na sygnał
- Funkcja sigaction ustala reakcję na sygnał sig. Jeśli oact ma wartość inną niż null, sigaction zapisuje poprzednią reakcję na sygnał w lokacji wskazanej przez oact
- Jeśli act ma wartość null, to jest to jedyny rezultat działania sigaction; jeśli zaś act jest różne od null, wówczas ustawiana jest reakcja na określony sygnał
- Wewnątrz struktury sigaction, na którą wskazuje argument act, sa_handelr jest wskaźnikiem do funkcji wywoływanej po otrzymaniu sygnału sig.

Przykładowy program stosujący sigaction

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void ouch(int sig)
{ printf("OUCH! - I got signal %d\n", sig);
}

int main()
{
  struct sigaction act;
  act.sa_handler = ouch;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGINT, &act, 0);
  while(1) {
            printf("Hello World!\n");
            sleep(1);
           }
}
```

```
$ ./ctrlc2
Hello World!
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^C
OUCH! - I got signal 2
Hello World!
Hello World!
^\
Quit
$
```