Systemy operacyjne – studia stacjonarne 2023/24

Lab 7. Funkcje systemowe związane z plikami, środowiskiem, czasem. Wątki.

1. Operacje na plikach.

Programy użytkownika systemu Linux odwołują się do jądra systemu poprzez funkcje systemowe. Tylko za ich pośrednictwem możliwe jest np. korzystanie z systemu plików, czy też mechanizmów komunikacji między procesami. Wywołania funkcji niskopoziomowych (open, read, write, close) odwołują się bezpośrednio do sterowników urządzeń. Oznacza to przełączenie do pracy w trybie jądra przy każdym takim wywołaniu i wiąże się często z małą wydajnością. Dlatego w systemie Linux dostępne są biblioteki, które optymalizują wykonywane operacje, jak np. standardowa biblioteka wejścia/wyjścia z plikiem nagłówkowym stdio.h, stosująca bufory o rozmiarach dostosowanych do sprzętu.

Niskopoziomowe funkcje dostępu do plików korzystają z deskryptorów plików, natomiast funkcje wysokopoziomowe używają odpowiadających im strumieni plikowych. Każdy uruchomiony program ma dostępne trzy strumienie plikowe, zadeklarowane w pliku stdio.h, którym odpowiadają deskryptory:

	deskryptor	strumień plikowy
standardowe wejście	0	stdin
standardowe wyjście	1	stdout
standardowe wyjście błędu	2	stderr

Każdy otwarty plik w programie otrzymuje deskryptor, będący kolejną liczba całkowitą.

Ćwiczenie 1:

Napisz program w języku C, który testuje prawo SUID do pliku: Do pliku **dane** w swoim katalogu domowym zablokuj prawa dostępu dla członków grupy. W programie **program** otwórz plik dane do odczytu i zapisu i zapisz do niego jakieś dane, po czym odczytaj te dane. Ustaw prawa dostępu, które umożliwiają członkom grupy uruchomienie programu. Przetestuj program i poproś użytkownika należącego do tej samej grupy o przetestowanie jego działania. Ustaw prawo SUID do pliku i ponownie wykonaj testy.

2. Nisko i wysokopoziomowe operacje wejścia/wyjścia.

Przykłady poniżej ilustrują różnice między nisko i wysokopoziomowymi operacjami wejścia/wyjścia.

Pliki źródłowe przykładów do pobrania z katalogu : /home/inf-prac/wojtas.jan/Dydaktyka/SO/LAB7

Porównaj sposoby wyprowadzania informacji z programu.

```
#include <stdio.h>
#include <unistd.h>
//niskopoziomowo bez buforowania
int main()
{
int i=0;
while (i<20){
        write(1,"1",1);
        sleep(1);
        i++;
        }
i=0;
while (i<10){
        write(2,"2",1);
        sleep(1);
        i++;
      }
}
```

```
#include <stdio.h>
#include <unistd.h>
//wysokopoziomowo
int main()
{
int i=0;
while (i<10){
           printf("1");
           sleep(1);
           //fflush(stdout);
           i++;
           }
i=0;
while (i<10) {
           fprintf(stderr,"2");
           sleep(1);
           i++;
           }
}
```

Do programowego kopiowania plików można używać zarówno funkcji niskopoziomowych jak i wysokopoziomowych. Ich efektywność może być różna i zależy od przyjętych parametrów. Do badania

czasu trwania procesu zastosujemy polecenie systemowe **time**, które wyświetla czas działania procesu, będącego parametrem wywołania z wyszczególnieniem czasu w trybie systemowym, w trybie użytkownika oraz czasu rzeczywistego.

Ćwiczenie 2:
Porównaj czasy kopiowania pliku o rozmiarze 10MB używając funkcji niskopoziomowych: oraz funkcji ze standardowej biblioteki we/wy. W każdym przypadku kopiowanie powinno odbywać się po znaku oraz za pomocą buforów o rozmiarze 1024B, 2048B oraz 4096B.
Czasy kopiowania (tryb systemowy) zbierz w tabelce i przedstaw wnioski. Programy powinny mieć jak najprostszą formę, tak aby czas ich wykonania odpowiadał czasowi kopiowania.

Przykładowe programy:

```
//Program kopiujący znak po znaku z wykorzystaniem funkcji niskopoziomowych
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
char c;
int we, wy;
we=open("we", O_RDONLY);
wy=open("wy", O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
while(read(we,&c,1)==1)
        write(wy,&c,1);
}
```

```
//Program kopiujący znak po znaku z wykorzystaniem funkcji wysokopoziomowych
#include <stdio.h>
int main()
{
FILE *we,*wy;
int c;
we=fopen("we","r");
wy=fopen("wy","w");
if((we!=NULL)&&(wy!=NULL))
        while((c=fgetc(we))!=EOF)
        fputc(c,wy);
else
        printf("blad otwarcia\n");
}
```

```
//Program kopiujący blokami o rozmiarze 1024B z wykorzystaniem funkcji niskopoziomowych
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
int main()
{
char blok[1024];
int we, wy;
int liczyt;
we=open("we", O_RDONLY);
wy=open("wy", O_WRONLY|O_CREAT,S_IRUSR|S_IWUSR);
while((liczyt=read(we,blok,sizeof(blok)))>0)
        write(wy,blok,liczyt);
}
```

```
//Program kopiujący blokami o rozmiarze 1024B z wykorzystaniem funkcji wysokopoziomowych
#include <stdio.h>
int main()
{
char blok[1024];
FILE *we,*wy;
int c;
we=fopen("we","r");
wy=fopen("wy","w");
if((we!=NULL)&&(wy!=NULL))
        while(fgets(blok,1024,we))
                fputs(blok,wy);
else
        printf("blad otwarcia\n");
}
```

Dodatkowe informacje:
http://www.linux.pl/man/index.php?command=creat
http://www.linux.pl/man/index.php?command=fopen

3. Operacje na katalogach.

Grupa funkcji systemowych dedykowanych do programowego przetwarzania katalogów odwołuje się do strumienia katalogowego, czyli wskaźnika do struktury **DIR**. Wpisy katalogowe używają struktury **dirent**. Poniższy program wykorzystuje funkcję **opendir** otwierającą katalog, funkcję **readdir** odczytującą wpis z katalogu oraz funkcję **closedir** zamykającą katalog.

```
#include <stdio.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int main(int argc, char * argv[])
{
DIR *dirp;
struct stat status;
struct dirent *direntp;
dirp=opendir(argv[1]);
while((direntp=readdir(dirp))!=NULL)
  {
    printf("\t%s\n",direntp->d_name);
        lstat(direntp->d_name,&status);
        printf("czas dostępu=%d\nrozmiar=%d\n",status.st_atime,status.st_size);
        if(S_ISDIR (status.st_mode))
                printf("katalog\n");
  }
closedir(dirp);
}
```

Dodatkowe informacje:
http://www.linux.pl/man/index.php?command=opendir
http://www.linux.pl/man/index.php?command=readdir
http://www.linux.pl/man/index.php?command=closedir
http://www.linux.pl/man/index.php?command=lstat

4. Funkcje systemowe związane ze środowiskiem.

   Za pomocą funkcji **getenv** można odczytywać wartość zmiennej środowiskowej, funkcja **putenv** umożliwia zdefiniowanie nowej zmiennej środowiskowej procesu:

```
#include <stdlib.h>
char *getenv(const char *nazwa);
int putenv(const char *ciag);
```

   http://www.linux.pl/man/index.php?command=getenv

   http://www.linux.pl/man/index.php?command=putenv

   Przykład.

   Program wywołany z jednym parametrem odczytuje wartość zmiennej środowiskowej wskazanej parametrem, natomiast wywołany z dwoma parametrami definiuje nową zmienną i nadaje jej wartość według drugiego parametru (będą komunikaty: Naruszenie ochrony pamięci (core dumped) 😊)

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
int main(int liczarg, char *tablarg[])
{
char *zmienna, *wart;
zmienna=tablarg[1];
wart=getenv(zmienna);
if(wart)
        printf("zmienna %s ma wartosc %s\n",zmienna,wart);
else
    printf("zmienna %s nie ma wartosci\n",zmienna);
char *string;
wart=tablarg[2];
string=malloc(strlen(zmienna)+strlen(wart)+2);
if(!string)
        {
        fprintf(stderr,"brak pamieci\n");
        exit(1);
        }
strcpy(string, zmienna);
strcat(string,"=");
strcat(string,wart);
if(putenv(string)!=0)
        {
        fprintf(stderr,"blad\n");
        free(string);
        exit(1);
        }
wart=getenv(zmienna);
if(wart)
        printf("nowa wartosc %s jest %s\n", zmienna,wart);
else
    printf("nowa wart %s jest null\n", zmienna);

exit(0);
}
```

5. Funkcje związane z czasem.

   Funkcja **time** zwraca aktualny czas w postaci liczby sekund, jakie upłynęły od 1.01.1970 roku (Coordinated Universal Time (UTC)). Wynik trafia również w miejsce wskazane przez argument:

```
#include <time.h>
time_t time(time_t *tloc);
```

Przykłady wywołania funkcji time.

```
#include<time.h>
#include<stdio.h>
#include<unistd.h>
#include<stdlib.h>
int main()
{
time_t czas;
time_t *ti;
printf("czas=%ld\n",time(ti));
sleep(5);
time(&czas);
printf("czas=%ld\n",czas);
sleep(2);
czas=time(ti);
printf("czas=%ld\n",czas);
exit(0);
}
```

Funkcje **gmtime** i **localtime** zamieniają czas niskopoziomowy - podany za pomocą typu time_t na strukturę tm, która zawiera następujące pola:

int tm_sec – sekundy <0; 60>
int tm_min – minuty <0; 60>
int tm_hour – godziny <0; 23>
int tm_mday – dzień miesiąca <1; 31>
int tm_mon – miesiąc <0; 11>; 0 – oznacza styczeń
int tm_year – rok-1900; rok 2009 to 109
int tm_wday –dzień tygodnia <0; 6>; 0–to niedziela
int tm_yday – dzień w roku <0; 365>
int tm isdst – strefa czasowa

```
#include <time.h>
struct tm *gmtime(const time_t *czas);
struct tm *localtime(const time_t *czas);
```

Funkcja localtime zwraca czas lokalny, uwzględniając strefę czasową

Do przekształcenia czasu przedstawionego w formacie struktury tm na typ time_t służy funkcja mktime:

```
time_t mktime(struct tm *wskczas);
```

Istnieją funkcje asctime i ctime, przekształcające czas na ściśle określony format.

```
#include <time.h>
char *asctime(const struct tm *wskczas);
char *ctime(const time_t *wartczas);
```

Jeśli czas jest typu time_t wywołanie funkcji time(czas) jest równoważne wywołaniu: asctime(localtime(czas)).

Fragment kodu:

```
czas=time(ti);
printf("ctime: %s\n",ctime(ti));
```

powoduje wyświetlenie informacji:

```
ctime: Fri Jun 5 23:50:31 2009
```

Do wyspecyfikowanego przez użytkownika formatowania czasu służą funkcje **strftime** i **strptime**. Funkcja strftime przekształca czas wskazany przez wskczas podany w formacie struktury tm według wskazanego formatu, a wynik zapisuje w stringu s.

```
#include <time.h>
```

```
size_t strftime(char *s, size_t maxrozmiar, const char *format, struct tm *wskczas);
```

Wybrane specyfikatory formatu:

%a, %A	skrócona, pełna nazwa dnia tygodnia
%b, %B	skrócona, pełna nazwa miesiąca
%c	data i godzina
%H	godzina
%Y	rok-1900
%p	a.m lub p.m

Funkcja **strptime** wczytuje datę w postaci ciągu znaków (bufor) i wypełnia strukturę tm (wskczas) wg. zadanego formatu (format).

```
#include <time.h>
char *strptime(const char *bufor, const char *format, struct tm *wskczas);
```

Poniżej program testujący funkcje związane z czasem. Program sprawdza jaką datą jest czas = 0, zamienia datę: 1970-01-01:01:00:00 z powrotem na typ time_t, a następnie podaje dzień tygodnia odpowiadający wprowadzonej w zadanym formacie dacie.

```
#include <stdio.h>
#include <time.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
int main()
{
struct tm *wczas;
time_t czas;
char bufor[1024];
czas=0;
wczas=gmtime(&czas);
printf("czas=%ld\n",czas);
strftime(bufor, 1024, "%Y-%m-%d:%T", wczas);
printf("czas 0  to: %s\n", bufor );
char znak;
strcpy(bufor,"1970-01-01:01:00:00");
printf("bufor=%s\n",bufor);
znak = strptime(bufor, "%Y-%m-%d:%T", wczas);
czas = mktime(wczas);
printf("zamiana na time_t   --- czas= %ld\n", czas);
printf("Podaj date <YYYY-MM-DD>\n");
fgets(bufor, 12,stdin);
strptime(bufor, "%Y-%m-%d", wczas);
strftime(bufor, 1024, "%A", wczas);
printf("dzien tygodnia= %s\n", bufor);
}
```

Dodatkowe informacje:

https://man.linux.pl/?section=&command=gmtime/localtime/mktime/asctime/ctime

https://man.linux.pl/?section=&command=strptime

https://man.linux.pl/?section=&command=strftime

6. Wątki.

Do wywołania funkcji niezbędne są następujące pliki nagłówkowe:

```
#include <pthread.h>
```

Tworzenie wątku: pthread_create().

Pliki nagłówkowe	<pthread.h>		
Prototyp	int pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*func)(void *), void *arg);		
Zwracana wartość	sukces	porażka	
	0	-1	

tid - identyfikator wątku;
attr - atrybuty wątku (zazwyczaj NULL);
func - funkcja wątku;
arg - argument wątku przekazywany do funkcji wątku przy rozpoczęciu jego wykonywania (struktura);
Po uruchomieniu programu przez funkcję exec() następuje utworzenie jednego wątku nazywanego wątkiem głównym (main). Każdy dodatkowy wątek tworzymy przy pomocy funkcji pthread_create(). Do każdego wątku wewnątrz procesu można odnosić się za pośrednictwem identyfikatora wątku (tid). W razie pomyślnego utworzenia nowego wątku, funkcja pthread_create() przekazuje identyfikator tego wątku jako wartość wskazywaną przez argument tid. Każdy wątek ma wiele atrybutów: swój priorytet, początkowy rozmiar stosu, informację o tym czy powinien zostać wątkiem-demonem oraz różne inne informacje. Jeżeli tworząc nowy wątek, chcemy określić wartości tych atrybutów, to możemy zainicjować zmienną typu pthread_attr_t i wskaźnik do tej zmiennej przekazać do funkcji pthread_create(), powodując zastąpienie domyślnych wartości atrybutów wątku. Zazwyczaj korzysta się z wartości domyślnych, przekazując wskaźnik pusty w miejscu attr.

Przyłączenie wątku: pthread_join().

Pliki nagłówkowe	<pthread.h>		
Prototyp	int pthread_join(pthread_t tid, void **status);		
Zwracana wartość	Sukces	porażka	
	0	-1	

tid - identyfikator wątku (przekazany przez pthred_create);
status - jeżeli jest niepustym wskaźnikiem (NULL) to w miejscu wskazywanym przez jego wartość zostaje zachowana wartość zwrócona przez wątek;
funkcja pthread_join() jest odpowiednikiem funkcji waitpid() dla procesów.

Odłączenie wątku: pthread_detach().

Pliki nagłówkowe	<pthread.h>		
Prototyp	int pthread_detach(pthread_t tid);		
Zwracana wartość	Sukces	porażka	
	0	-1	

tid - identyfikator wątku (przekazany przez pthred_create);
Nie można czekać na zakończenie wykonywania wątku odłączonego. Jeżeli jeden wątek powinien otrzymać informację o zakończeniu wykonywania innego wątku, to ten inny wątek powinien pozostać przyłączalny. Funkcja pthread_detach() modyfikuje określony wątek powodując jego odłączenie.

Pobranie wartości identyfikatora wątku: pthread_self().

Pliki nagłówkowe	<pthread.h>		
Prototyp	pthreat_t pthread_self(void);		
Zwracana wartość	Sukces	Porażka	
	Identyfikator wywołującego wątku	-1	

funkcja pthread_self jest odpowiednikiem funkcji getpid() przekazującej identyfikator procesu;

Zakończenie wątku: pthread_exit().

Pliki nagłówkowe	<pthread.h>		
Prototyp	void pthread_exit(void *status);		
Zwracana wartość	Sukces	Porażka	

Jeżeli wątek nie jest odłączony, to jego identyfikator oraz stan końcowy są przechowywane dla przyszłego wywołania funkcji pthread_join() przez inny wątek wewnątrz tego samego procesu. Argument status nie może wskazywać na obiekt lokalny w kontekście wywołującego wątku, ponieważ taki obiekt zniknie po zakończeniu wykonywania tego wątku;
Inne sposoby zakończenia wątku:

- Wątek kończy działanie z chwilą, gdy następuje powrót z funkcji, od której zaczęło się wykonywanie wątku. Ponieważ funkcja jest deklarowana jako wskaźnik void *, więc jej wynik jest stanem końcowym wątku;
- Proces (a wraz z nim wszystkie jego wątki) kończy działanie (gdy następuje powrót z jego funkcji main() lub któryś z wątków wywołuje funkcję exit();

Dodatkowe informacje:
http://www.linux.pl/man/index.php?command=pthread_create
http://www.linux.pl/man/index.php?command=pthread_join
http://www.linux.pl/man/index.php?command=pthread_detach
http://www.linux.pl/man/index.php?command=pthread_exit
http://www.linux.pl/man/index.php?command=pthread_self

Ćwiczenie 3.
Skopiuj do swojego katalogu domowego, pliki watek.c oraz watek1.c znajdujące się w katalogu:
/home/inf-prac/wojtas.jan/Dydaktyka/SO/Projekty/WATKI
W trybie pracy krokowej przeanalizuj sposób działania funkcji związanych z tworzeniem, przyłączaniem, odłączaniem i zakończeniem pracy wątków.

Ćwiczenie 4.
Napisać program złożony w 2 wątków. Wątek główny losowo generuje elementy tablicy np. int tab[2][10]. Wątek 1 liczy sumę elementów pierwszego wiersza tablicy. Wątek 2 liczy sumę elementów drugiego wiersza tablicy. Wątek główny liczy sumę całkowitą z sum cząstkowych wyznaczonych przez poszczególne wątki.

*Treści oznaczone kursywą pochodzą z różnych źródeł internetowych.